Architectures Logicielles et Matérielles

P. Amblard, J.-C. Fernandez,
F. Lagnier, F. Maraninchi,
P. Sicard, Ph. Waille

Table des matières

Introduction

Ce qu'on trouvera dans ce livre

Ce livre suit d'assez près l'enseignement dispensé en Licence d'informatique à l'Université Joseph Fourier de Grenoble. L'enseignement a le même titre : *Architectures Logicielles et Matérielles*. Il est dispensé en environ 150 heures de cours, Travaux Dirigés et Travaux Pratiques.

L'objectif est d'expliquer à de futurs spécialistes d'informatique le fonctionnement de l'ordinateur. Pour cela nous faisons un certain nombre de choix, nous prenons parti.

Pour *comprendre* le fonctionnement, il faut se placer du point de vue du *concepteur* d'ordinateur. Le lecteur trouvera donc dans ce livre une démarche de conception de machines. Il ne s'agit pourtant pas de lui faire croire au réalisme de cette conception.

En effet la véritable conception d'une machine, c'est-à-dire de son matériel — du microprocesseur à la mémoire, en passant par la carte graphique — et de son logiciel — du système d'exploitation aux compilateurs — représente des centaines de milliers d'heures de travail de spécialistes. Nous ne décrivons qu'une partie du travail, en choisissant les points qui nous semblent les plus significatifs dans cette conception. D'autre part nous insistons sur les liaisons entre différents aspects de la conception. En particulier, l'une des idées fortes de ce livre est l'étroite complémentarité des aspects logiciels et matériels des ordinateurs. L'idée centrale, et le chapitre central de ce livre, montrent donc comment du matériel exécute du logiciel.

Le contenu de ce livre ne devrait pas se périmer, sauf si des principes vraiment nouveaux apparaissent en informatique et se généralisent.

Ce qu'on ne trouvera pas dans ce livre

En revanche ce livre ne décrit pas les aspects les plus avancés utilisés dans les machines actuelles. Ces aspects font l'objet d'enseignements spécifiques de systèmes d'exploitation, de compilation ou d'architectures des machines, dans lesquels, en général, on ne se préoccupe que d'un aspect. Ce livre constitue un prérequis pour de tels enseignements car il montre les relations entre les 3 domaines.

Parmi les thèmes très intéressants que nous avons délibérement écartés (et réservés pour le tome 2 !) figurent :

- L'étude fine des fonctionnalités d'un système d'exploitation particulier. Beaucoup de nos références sont inspirées d'UNIX[1].
- L'étude de la hiérarchie mémoire (cache et mémoire virtuelle), que nous passons totalement sous silence.
- L'étude détaillée d'un langage d'assemblage d'un processeur donné. Beaucoup de nos références sont inspirées du SPARC[2] ou du Motorola 68000[3].
- L'étude des techniques de conception de circuits micro-électroniques. Par exemple nous ne parlons ni de consommation, ni de circuits asynchrones.
- L'étude des techniques d'optimisation des performances des processeurs. Nous ne développons pas les techniques de pipeline, ni celles de réordonnancement dynamique du flot d'exécution des instructions.
- Les entrées/sorties très particulières que constituent les accès d'un ordinateur à un réseau, ce qui demanderait un développement spécifique.

Comment lire ce livre ?

Méthode de travail

On peut lire ce livre comme un roman, de la première à la dernière page.

On peut également le lire avec une salle de Travaux Pratiques à portée de la main, pour essayer toutes les techniques évoquées, les comparer, les analyser en détail, etc.

On peut essayer de résoudre tous les exercices et envoyer les solutions aux auteurs, qui se feront un plaisir de les corriger :

`Paul.Amblard@imag.fr` `Jean-Claude.Fernandez@imag.fr`
`Fabienne.Lagnier@imag.fr` `Florence.Maraninchi@imag.fr`
`Pascal.Sicard@imag.fr` `Philippe.Waille@imag.fr`

On peut enfin essayer de trouver des erreurs, de fond et de forme, et on y parviendra certainement.

Thèmes

On peut privilégier une approche centrée sur les langages de programmation, leur traduction et la façon dont ils sont exécutés. Sur la figure 0.1 cela correspond aux flèches en traits gras.

---

[1] marque déposée, et dans la suite de l'ouvrage nous ne préciserons plus que les noms de systèmes et de machines sont, évidemment, déposés.

[2] marque déposée

[3] marque déposée

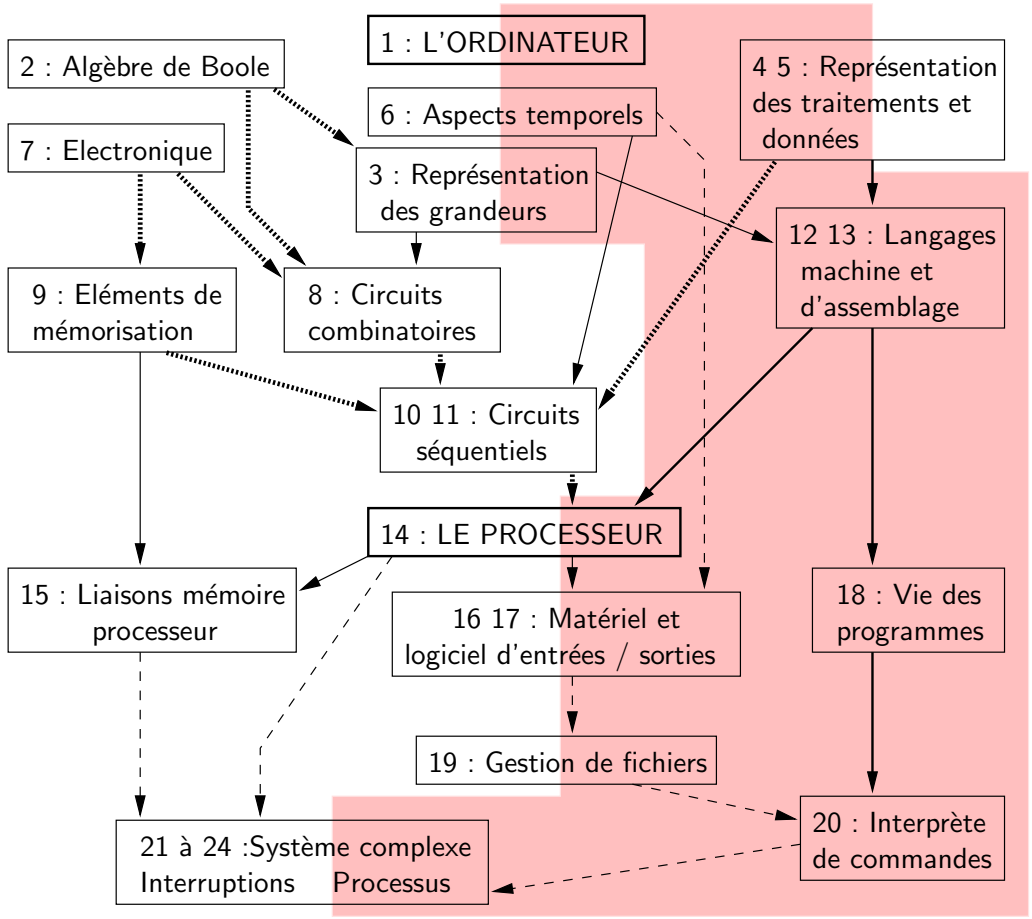


FIG. 0.1 – Relations de dépendance des principales idées utilisées dans les 24 chapitres. La zone grisée correspond plutôt au monde du logiciel, la zone blanche au matériel.

On peut privilégier une approche de conception des circuits digitaux et d'architecture de machine. Sur la figure 0.1 cela correspond aux flèches en traits larges et hachurés.

On peut privilégier une approche centrée sur l'architecture de haut niveau et les systèmes d'exploitation. Sur la figure 0.1 cela correspond aux flèches en traits pointillés.

Il n'en reste pas moins que les auteurs ont cherché à mettre l'accent sur la globalité et la complémentarité des 3 approches.

Index

Les mots en italique apparaissent souvent en index. Dans l'index, les numéros de page en gras indiquent les occurrences de définition des mots. Les autres numéros indiquent des occurrences d'utilisation des mots, parfois antérieures à leur définition, parfois postérieures.

Remerciements

Les idées, principes, techniques, outils, méthodes, présentés dans ce livre ne sont pas les résultat de nos découvertes. Nous avons reçu des enseignements, puis nous avons lu, essayé, enseigné. Sans ceux qui nous ont précédés ce livre n'existerait pas. Sans celles et ceux qui ont enseigné avec nous le module *Architectures Logicielles et Matérielles* au fil des années il serait sans doute plus pauvre. En particulier Catherine, Danielle, Joëlle, Jean-Louis et Jean-Paul reconnaîtront certaines de leurs bonnes influences. Les mauvaises viennent d'ailleurs !

Chapitre 1

Qu'est-ce qu'un ordinateur ?

Un *ordinateur* est une machine, presque toujours électronique, qui exécute des programmes. Ces programmes traitent des données. Une machine électronique est un objet. Par opposition, les programmes et les données sont des informations. Cette opposition est celle qui existe entre *matériel* et *logiciel*. L'ensemble du livre est consacré à montrer de façon détaillée comment ces deux univers se rencontrent pour former l'*architecture* de l'ordinateur. Dans ce premier chapitre, nous faisons un très rapide survol permettant de situer les notions avant de les décrire de façon détaillée.

> *Le paragraphe 1. décrit ce qu'est une information et sa représentation. Cela nous permet de parler de programmes. Puis nous décrivons une machine à exécuter les programmes et nous insistons sur la notion d'exécution dans le paragraphe 2. Cela nous permet au paragraphe 3. de montrer les différents matériels et logiciels présents dans l'ordinateur. Nous évoquons enfin les usages de l'ordinateur au paragraphe 4.*

1. Notion d'information

Une *information* est une entité abstraite, liée à la notion de connaissance. Nous nous intéressons naturellement aux informations d'un point de vue technique en informatique, non d'un point de vue journalistique. Nous donnons différentes facettes de l'information et séparons l'étude des informations de celle des objets.

1.1 Quelques aspects d'une information

Nous avons besoin pour cerner la notion d'information de donner l'origine possible d'une information et de montrer la nécessité de ses représentations pour pouvoir envisager les manipulations d'informations dans les ordinateurs.

1.1.1 Origine d'une information

Une information peut être en relation avec une grandeur physique, l'origine étant par exemple mécanique (forme, dimensions, emplacements d'objets, intensité d'une force), électromagnétique (amplitude, fréquence ou phase d'un signal électrique, d'une onde électromagnétique), électrochimique (PH d'un liquide, potentiel électrochimique d'une cellule nerveuse).

1.1.2 Nom, valeur et combinaison d'informations

Une information a un *nom* : "la température mesurée au sommet de la Tour Eiffel", "le caractère tapé au clavier", "le montant de mon compte en banque".

Une information a une *valeur* à un certain moment : 37 degrés, 'A', 5 000 F.

La plus *petite* information possible est une réponse par oui ou par non (on parle de réponse *booléenne*) : le nombre est pair ou impair, le caractère est une lettre ou pas une lettre, le point de l'écran est allumé ou éteint, la lettre est une majuscule ou non, la touche de la souris est enfoncée ou non. Une telle petite information constitue un *bit*.

L'ensemble des valeurs possibles peut être fini (comme pour les caractères), ou potentiellement infini (comme pour mon compte en banque !). Un ensemble infini de valeurs peut présenter des variations *continues*, c'est-à-dire qu'entre deux valeurs possibles il y a une valeur possible. C'est le cas pour la température. Les variations sont *discrètes* dans le cas contraire. Le solde de mon compte en banque peut être de 123,45 F ou de 123,46 F, mais pas d'une valeur entre les deux, car la banque arrondit les sommes au centime le plus proche.

Différentes informations peuvent se combiner soit dans l'espace (les montants des comptes en banque de différents clients) soit dans le temps (l'historique des variations de mon compte).

Les combinaisons dans l'espace contiennent un nombre fini d'éléments. En revanche un système informatique traite des informations qui peuvent varier un nombre non borné de fois au fil du temps. Il suffit de maintenir le système en état de marche.

1.1.3 Représentation et codage

Une information a une *représentation* sous forme de grandeur(s) physique(s) associée à une convention, ou *code*, d'interprétation. Si une information est représentée dans un code inconnu, elle n'est pas compréhensible.

La grandeur physique peut être la position d'une aiguille sur un appareil de mesure. On passe parfois par une représentation intermédiaire sous forme de suite de lettres et de chiffres, représentés à leur tour par une grandeur physique (traces sur un papier par exemple). Pour l'aiguille sur un cadran on parle de représentation *analogique* ; si l'intermédiaire des chiffres est mis en jeu on parle de représentation *numérique* ou *digitale*. Cette différence se retrouve

entre les disques anciens et les disques compacts. Il est parfois nécessaire de réaliser par un dispositif électronique une *conversion* entre ces deux types de représentation.

Un chiffre *binaire*, 0 ou 1, suffit à représenter un bit. Un *vecteur de bits* constitue un *mot*. Les mots de 8 bits sont des *octets*.

Une même information peut être représentée dans l'ordinateur de façons diverses : le caractère frappé au clavier est d'abord connu comme un couple de coordonnées d'une touche au clavier (la touche en deuxième colonne de la troisième ligne), puis par une séquence de variations de potentiel sur une ligne électrique liant le clavier et l'ordinateur (combinaison temporelle), puis par un vecteur de chiffres binaires dont les composantes sont les unes à côté des autres en mémoire (combinaison spatiale), puis par une représentation sous forme de matrice de points allumés/éteints sur l'écran.

Pour les informations structurées complexes (en raison des combinaisons) le codage constitue un *langage*. Les programmes sont écrits dans des langages de programmation, les figures sont décrites dans des langages de description de figures, etc.

Dans le langage courant on assimile souvent l'information, sa valeur, sa représentation.

1.2 Utilisation des informations dans l'ordinateur

Dans les ordinateurs les informations sont mémorisées, transmises et traitées. Nous retrouvons cette triple fonction dans le paragraphe 3.1.3 En informatique l'association du nom d'une information et de la représentation de la valeur constitue une *variable*.

1.2.1 Stockage (ou mémorisation) des informations

On peut copier, c'est-à-dire créer un nouvel exemplaire de l'information en lui associant un nouveau représentant physique. Mais c'est toujours la même information : elle est simplement matérialisée plusieurs fois. On peut aussi détruire un exemplaire de l'information : elle disparaîtra avec son dernier représentant. Une information est stockée dans une *mémoire* si on ne veut pas qu'elle disparaisse.

1.2.2 Transmission des informations

Les informations traitées dans l'ordinateur peuvent provenir de dispositifs matériels (capteur de température par exemple). Elles peuvent provenir d'un utilisateur via un clavier, une souris, ...Une information sortante, sous la forme d'une tension sur un fil électrique, peut influencer un matériel par l'intermédiaire d'un actionneur, comme un déclencheur d'alarme. Différents systèmes d'*interface* permettent à l'ordinateur de communiquer avec le monde extérieur.

Les informations peuvent être transmises d'un point à un autre. Des liaisons par fils électriques ou par ondes électro-magnétiques (radio, infra-rouge, visible, ...) nous sont familières. A l'intérieur d'un ordinateur la distance est parfois de moins d'un micron (10^{-6} m). Quand une fusée transmet vers la Terre des images de l'espace, la distance est de plusieurs millions de kilomètres. Les réseaux permettent les transmissions entre ordinateurs.

Il arrive que le codage de l'information comporte une certaine redondance. Cela peut permettre, si l'on garde l'information en excès, de détecter des erreurs de transmission, ou, si le débit d'information est une priorité, de compresser la représentation avant de la transmettre.

1.2.3 Traitement des informations : données, programmes

On peut réaliser des opérations de combinaison d'informations pour générer de nouvelles informations. Dans le cas des ordinateurs, il s'agit très souvent d'opérations arithmétiques de calcul et de comparaison. Etymologiquement l'*ordinateur* met de l'ordre.

Il existe des informations qui décrivent ces traitements appliqués à d'autres informations : "Diviser la distance parcourue par le temps de trajet. Le résultat est la vitesse" ; "Comparer deux caractères et déterminer le premier dans l'ordre alphabétique" ; "Convertir une information représentée selon le code 1 pour la représenter selon le code 2". Des enchaînements de tels ordres constituent des *programmes*. Les autres informations sont nommées *données*. Les ordres élémentaires sont des *instructions*. Une instruction indique un changement d'*état* dans l'ordinateur. L'état de la machine avant l'instruction est différent de son état après.

Attention, les instructions peuvent être considérées comme des données à un certain moment. Par exemple quand le programmeur imprime son programme, les instructions du programme d'impression traitent le programme comme un texte ordinaire ; de même le *compilateur* traite le programme à compiler comme une donnée.

On dit parfois que l'informatique concerne le traitement de l'information, mais il serait plus exact de parler du traitement d'une représentation de l'information. Cette représentation peut être finie (dans l'espace) ou infinie (dans le temps).

1.3 Information par rapport à objet, logiciel par rapport à matériel

Enfonçons quelques portes ouvertes pour distinguer la notion d'information de celle d'objet. La distinction est de même nature que celle qui distingue le logiciel du matériel.

Un objet peut être dupliqué. Cela donne deux objets. Si la représentation d'une information est dupliquée il n'y a toujours qu'une information. Mais

il y a probablement deux supports physiques. Les informations peuvent être mémorisées, évidemment pas les objets.

Une information peut voyager par téléphone ou par courrier électronique. Un objet ne le peut pas.

Produire un objet suppose de la matière première. La production d'objet est une activité économique du secteur secondaire. Produire une information demande de la matière grise. La production d'information est une activité du secteur tertiaire.

Lors de la réalisation d'un objet, des défauts de fabrication peuvent apparaître. Une information peut être considérée comme vraie ou fausse, mais elle n'a pas de défaut de fabrication.

Un objet peut tomber en panne, se dégrader au fil du temps. Une information peut être accessible ou non, dans un code compréhensible ou non. Le support de la représentation d'une information peut s'abîmer, la représentation peut disparaître.

1.4 Objet et description d'objet

Attention à ne pas confondre l'objet matériel et sa description ; la description de l'objet *est* une information. Ainsi la description d'un ordinateur n'a pas de défauts de fabrication, ne peut tomber en panne, est reproductible, voyage sur un fil.

Par contre l'ordinateur lui-même est un objet composé de fils, de silicium, de tôlerie, de ventilateurs. Sa description est une information codée graphiquement dans un schéma ou textuellement par un ensemble d'équations ou de formules. Il existe des langages de description de matériel informatique.

Pour obtenir l'objet il faut savoir *réaliser* la description. Le résultat de la fabrication de l'objet ordinateur doit être *testé*. On doit vérifier que l'objet est conforme à sa description du point de vue du fonctionnement. Ce test vise la découverte de défauts de fabrication. Après un temps de bon fonctionnement, on peut refaire un test pour découvrir ou localiser des pannes. Les défauts de conception sont d'une autre nature : ils concernent une différence entre la description de l'ordinateur et l'intention du concepteur. On peut les assimiler aux *bogues* des programmes. Les programmes n'ont pas de défauts de fabrication. Ils peuvent comporter des fautes de typographie, de syntaxe ou des erreurs de conception.

2. L'ordinateur : une machine qui exécute

L'ordinateur est un objet. Il exécute des informations (les programmes) à propos d'autres informations (les données). Un ordinateur correspond à un certain moule, un *modèle de calcul*.

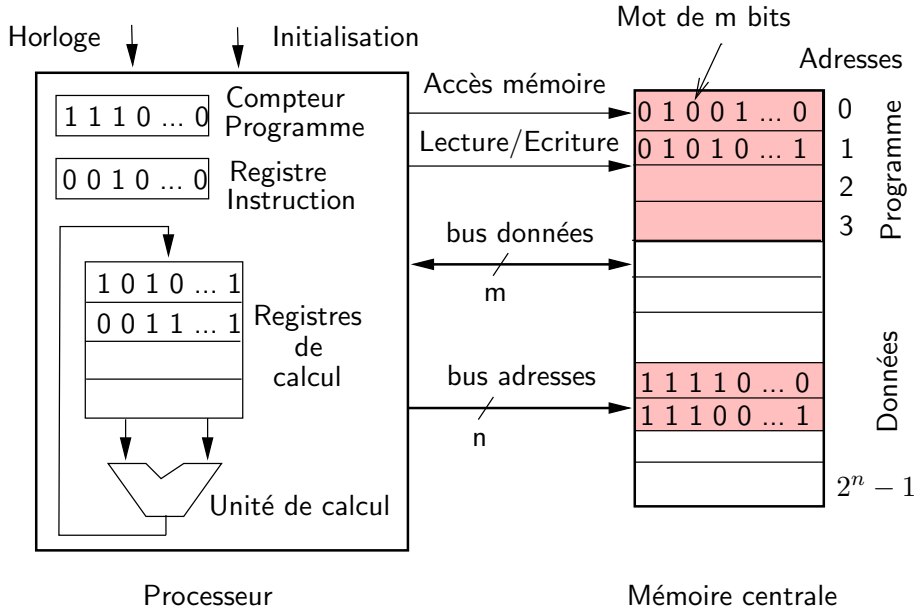


FIG. 1.1 – Architecture simplifiée d'une machine de Von Neumann

2.1 Modèle de calcul, machine de Turing

Un modèle de calcul comporte un ensemble de transformations applicables à un ensemble de données. Il comporte aussi l'ensemble des règles de composition de ces transformations. Prenons un exemple en géométrie où calculer signifie dessiner : le *calcul* par la règle et le T glissant. En géométrie plane, en n'utilisant que la règle et le T glissant, il est possible de *calculer* la parallèle à une droite passant par un point donné, la perpendiculaire à une droite passant par un point, l'orthocentre d'un triangle, etc. L'utilisation de la règle et du T glissant constitue un modèle de calcul.

Si l'on ajoute le compas, on obtient un autre modèle de calcul, plus puissant, c'est-à-dire permettant de construire d'autres figures.

En informatique, la machine abstraite de Turing est un modèle de calcul. Le mathématicien britannique Turing [Gir95, Tur54, Las98] a défini une classe de fonctions calculables en composant (éventuellement récursivement) des fonctions élémentaires. Il a défini un modèle abstrait de machine et montré que cette machine pouvait effectivement calculer la classe de fonctions définie. Ce modèle est un maximum, on ne connaît pas de modèle plus puissant. Cette machine est toutefois très rudimentaire, sa programmation est donc ardue. Obtenir un résultat suppose de nombreuses opérations élémentaires. La machine de Turing suppose l'existence d'un dispositif de mémorisation de dimension infinie. Ce n'est donc pas un modèle réaliste.

2.2 L'architecture de Von Neumann

Les travaux réalisés autour du mathématicien hongrois Von Neumann [BGN63] constituent le fondement de l'*architecture* des ordinateurs actuels. Du point de vue théorique, on a pu démontrer que le modèle *concret* de Von Neumann possède les propriétés de la machine *abstraite* de Turing.

Il y a quelques modèles de calcul en informatique qui ne sont pas de ce type : par exemple le calcul par réseaux de neurones formels.

Pratiquement tous les modèles informatiques de traitement se retrouvent dans la catégorie générale des *automates*, ou *systèmes séquentiels*.

Les principes de la machine de Von Neumann, que nous allons décrire, sont encore en oeuvre dans la quasi totalité des ordinateurs contemporains. Il y a eu, naturellement, de nombreuses améliorations. Une machine de Von Neumann (voir Figure 1.1) stocke des représentations des informations digitales, en binaire. Elle comporte deux éléments : une *mémoire* et une *unité centrale*. On parle plus facilement aujourd'hui de *processeur* plutôt que d'unité centrale. Habituellement les *machines parallèles* à plusieurs processeurs ne sont pas considérées comme des machines de Von Neumann.

2.2.1 La mémoire centrale

Les informations sont codées sous forme numérique. Les instructions, les caractères, les couleurs, etc., sont représentés par des suites de chiffres binaires. Les informations sont stockées dans une mémoire dans des emplacements numérotés nommés *mots*. Le numéro d'un emplacement est son *adresse*. Le maintien de la correspondance entre le *nom* de l'information et l'*adresse* du mot mémoire où est rangée une de ses représentations est une tâche difficile et une préoccupation permanente en informatique.

Une *écriture* dans la mémoire associe une valeur à une adresse (on parle aussi d'*affectation*). Après une écriture, on peut exécuter une ou plusieurs lectures de la même information. La *lecture* fournit la valeur associée à cette adresse.

La mémoire est à affectations multiples : on peut écrire successivement plusieurs valeurs dans un mot. Chaque écriture associe une nouvelle valeur à l'adresse. Elle induit un changement de l'état de la machine, en détruisant l'association précédente. Elle n'est pas réversible : il n'est pas possible d'annuler la nouvelle association pour accéder à nouveau à l'ancien contenu.

La mémoire contient des données et des programmes constitués de suite d'instructions. Le codage de l'information est tel que rien ne permet de reconnaître une représentation de donnée et une représentation d'instruction. Cette distinction n'aurait pas de sens puisqu'un programme peut même créer des données qui sont en fait des instructions. Cette possibilité est ce qui donne toute sa spécificité aux ordinateurs. Cela oppose le modèle de type Von Neumann à celui dit de Harvard ou de Manchester dans lequel il existe deux mémoires respectivement dédiées aux données et aux instructions.

2.2.2 Le processeur

Le processeur *exécute* les instructions. Les instructions sont généralement exécutées dans l'ordre où elles sont écrites dans la mémoire, mais certaines instructions peuvent introduire des *ruptures* de cette séquentialité. La règle générale est donc la correspondance entre l'ordre de rangement en mémoire et l'ordre d'exécution.

L'instruction en cours d'exécution est repérée par son adresse. Cette adresse est stockée dans une partie du processeur appelé *pointeur d'instruction*, *compteur ordinal* ou *compteur programme*.

Le processeur est doté au minimum de deux éléments de mémorisation particuliers appelés des *registres* : le compteur ordinal déjà cité et le *registre d'instruction* dans lequel le processeur stocke une copie de l'instruction en cours d'exécution.

Le processeur exécute cycliquement la tâche suivante dite d'*interprétation des instructions* ou d'*exécution des instructions* :

- Lecture de l'instruction à exécuter : le processeur transmet à la mémoire l'adresse de l'instruction à lire, autrement dit le contenu du compteur ordinal, et déclenche une opération de lecture. Il reçoit en retour une copie de l'instruction qu'il stocke dans son registre d'instruction.
- Décodage : le processeur examine le contenu du registre d'instruction et détermine l'opération à effectuer. Si le contenu du registre ne correspond pas à une instruction valide, c'est une erreur. En effet, en fonctionnement normal, le compteur programme pointe sur un mot mémoire contenant une instruction. Le décodage est le moyen de vérifier qu'une information est bien une instruction.
- Exécution : le processeur effectue l'opération décrite par l'instruction. Cette exécution met souvent en jeu une *unité de calcul* ou *Unité Arithmétique et Logique*. Cet opérateur effectue des calculs sur les données stockées dans des registres de calcul ou *accumulateurs*.
- Sélection de l'instruction suivante : le processeur calcule l'adresse de l'instruction suivante. Cela se fait le plus souvent en ajoutant 1 au compteur ordinal.

Une instruction de *saut* ou *branchement* force une rupture de l'ordre implicite d'exécution des instructions défini par leur position dans la mémoire. Son exécution consiste à stocker dans le registre compteur ordinal une autre adresse que celle obtenue implicitement par incrémentation de ce dernier. En utilisant des branchements, on peut faire en sorte que l'exécution globale du programme comporte plusieurs exécutions d'une même instruction. Le texte d'un programme est donc une représentation finie d'un comportement qui dure éventuellement indéfiniment. C'est l'essence même de la programmation. Un *algorithme* est un texte de taille finie.

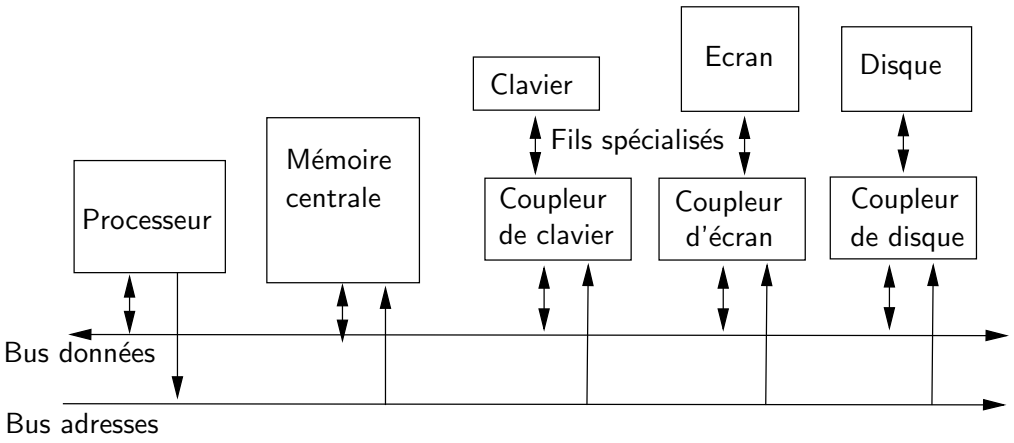


FIG. 1.2 – Architecture matérielle simplifiée d'un ordinateur

2.2.3 Liaisons entre le processeur et la mémoire

Le processeur dialogue avec la mémoire via trois sortes de fils électriques groupés en paquets nommés *bus* : 1) le *bus d'adresse* transmet du processeur vers la mémoire l'information adresse. 2) le *bus de données* transporte le contenu de l'emplacement mémoire auquel on accède. Le terme de *bus de valeur* aurait été plus explicite : le processeur peut interpréter la valeur qui transite sur ce bus comme une donnée ou comme une instruction. 3) des signaux complémentaires précisent à quel instant a lieu l'accès (Accès mémoire) et dans quel sens (Lecture/Ecriture). La figure 1.2 montre une telle organisation.

2.2.4 Langages du processeur : langage machine et langage d'assemblage

Pour être interprétées par le processeur, les instructions d'un programme doivent être représentées selon un certain code et stockées dans la mémoire centrale dans un format appelé *langage machine*. Le langage machine décrit l'ensemble des instructions comprises par le processeur et la convention de codage pour que celles-ci soient exécutables. On parle de *jeu* ou de *répertoire* d'instructions. Le codage d'une instruction est un vecteur de 0 et de 1.

Donnons quelques exemples de ce que peuvent être les instructions : "Ajouter 258 et le nombre contenu en mémoire à l'adresse 315 puis ranger le résultat à l'adresse 527", "Si le nombre rangé en mémoire à l'adresse 124 est positif, alors sauter à l'exécution de l'instruction à l'adresse 471, sinon continuer en séquence, c'est-à-dire passer à l'instruction suivante dans la mémoire". Différents sous-vecteurs, ou *champs*, représentent alors la valeur *immédiate* de l'*opérande* 258, ou bien l'adresse *directe* 315, ou encore le code *opération* addition.

Un programme écrit en langage machine a l'apparence d'une suite de

chiffres binaires peu évocatrice pour un programmeur humain. C'est pourquoi on utilise un langage dit d'*assemblage*, dans lequel on décrit exactement les mêmes instructions, mais sous une forme textuelle plus facile à manipuler que des paquets de 0 et de 1.

Le programe écrit en langage d'assemblage doit nécessairement être traduit en langage machine pour être exécuté. Le programme qui effectue cette traduction est l'*assembleur*. Un abus de langage fréquent confond le langage et le traducteur. On emploie l'expression impropre de *programmation en assembleur* pour *programmation en langage d'assemblage*.

On parle de code *source* à propos d'un programme écrit dans le langage d'assemblage et de format *objet* ou de *format exécutable* à propos du résultat de la traduction.

Le résultat de la traduction est une donnée qui peut être en mémoire, en cours d'exécution ou non. Il passe de l'état de donnée à celui de programme en cours d'exécution au moment du *lancement*. Le lancement est une opération spéciale qui change le statut du programme. Elle consiste à affecter au compteur ordinal du processeur l'adresse de la première instruction du programme à exécuter.

Chaque processeur a son propre langage machine. Un programme en langage machine écrit pour le processeur X ne peut pas, en général, s'exécuter sur un processeur Y. Dans le cas contraire X et Y sont dits *compatibles*.

Il existe des machines à jeu d'instructions complexe *Complex Instruction Set Computer* ou plus restreint *Reduced Instruction Set Computer*.

3. Où sont le matériel et le logiciel ?

Dans une machine informatique se trouvent du matériel et des logiciels. Une fonction réalisée par du logiciel sur une certaine machine peut être réalisée par du matériel sur une autre. C'est le cas pour certains calculs ou transcodages complexes, par exemple le calcul sur des réels représentés dans le codage *virgule flottante*. La mise en oeuvre d'un algorithme par un programme est classique. Sa mise en oeuvre par du matériel, par des techniques d'*algorithmique câblée*, est moins connue car moins facile à expérimenter sur un simple ordinateur personnel.

3.1 Matériel

Le matériel est pourtant plus directement accessible à la vue. Nous allons l'examiner selon trois critères : son aspect, sa technologie et sa fonction.

3.1.1 Aspect du matériel

Une première approche du matériel consiste à le considérer selon son aspect. Un ordinateur peut ressembler à une caisse surmontée d'un écran. N'oublions

pas qu'un ordinateur peut parfois être une armoire, ou une carte imprimée, voire simplement une *puce* ou *circuit* comme sur votre carte bancaire. L'écran n'est pas nécessaire à l'ordinateur. Ce n'est qu'un moyen de communiquer entre la machine et l'être humain.

3.1.2 Technologie du matériel

Une deuxième classification d'éléments matériels se base sur les phénomènes mis en oeuvre.

Certains systèmes sont purement électriques ou électroniques. Ces systèmes sont organisés selon une hiérarchie correspondant à la technologie de réalisation utilisée : dans les caisses, il y a des cartes imprimées, sur lesquelles sont soudés des boîtiers. Dans les boîtiers il y a des (le plus souvent une seule) puces comprenant des *transistors*, résistances et condensateurs.

Pour l'acquisition de données externes, on utilise souvent des systèmes mécaniques ou électromécaniques. Les claviers et souris sont de ce type. On a plus généralement des *capteurs* de pression, d'accélération, etc., et des *actionneurs* de mouvements divers. L'ensemble peut constituer un *robot*. Différents capteurs ou actionneurs peuvent se trouver sous forme de puce ou de composants séparés. Les *microsystèmes* réunissent sur une seule puce capteurs, actionneurs et l'électronique de traitement. Des systèmes électromécaniques sont utilisés notamment pour la lecture ou l'enregistrement sur supports magnétiques ou optiques.

Certains systèmes sont électro-optiques comme les écrans, les diodes électroluminescentes, les caméras ou appareils photo numériques, les lecteurs de code-barre, les scanners, etc.

Les ordinateurs pneumatiques, où la pression dans des tubes tient lieu de courant électrique, sont assez rares.

3.1.3 Fonctions du matériel

La troisième façon de caractériser le matériel est de le faire d'après sa fonction. Les éléments matériels ont différents types de fonctions : de *mémorisation*, de *traitement* et de *communication*.

La *mémorisation* stocke des informations dans la machine. Le coût et la durée du stockage et des opérations de copie dépendent fortement du mode de représentation physique.

Si une information était représentée par un champ de menhirs, le stockage prendrait de la place, la duplication serait difficile (sauf pour Obélix). La durée de stockage serait en revanche de plusieurs siècles.

Dans l'ordinateur, l'information est représentée par des signaux électriques de faible puissance. La copie est rapide et de faible coût énergétique. La durée de vie dépend éventuellement d'une source d'alimentation électrique.

On distingue classiquement la *mémoire principale* et la *mémoire secondaire*. La mémoire principale est directement accessible dans l'ordinateur. Elle com-

porte une partie de *mémoire vive* et une partie de *mémoire morte*. Quand on coupe l'alimentation électrique, la mémoire morte ne perd pas les informations qui y sont inscrites. La mémoire morte ne peut pas être facilement modifiée. La mémoire secondaire contient des informations moins directement accessibles par le processeur. Il faut passer par une *interface*. Ainsi les disques souples ou durs sont des mémoires secondaires. Elles sont généralement permanentes : l'information y reste en l'absence d'alimentation électrique. La carte perforée a longtemps constitué un support de stockage en informatique. Son avantage est de pouvoir être lue directement par l'utilisateur humain.

Une mémorisation a lieu aussi dans le processeur qui garde temporairement des copies de certaines informations dans ses registres.

La fonction de *traitement* est assurée par le processeur. Il peut lire ou écrire le contenu de la mémoire principale. Il peut ensuite, comme on l'a vu, exécuter les instructions lues.

D'autres circuits ont des fonctions de *communication* entre le processeur et la mémoire ou entre le processeur et le monde extérieur. Ces circuits d'interfaçage et de communication sont des *coupleurs*. Les communications avec le monde extérieur se font à travers des *périphériques* comme les claviers, souris, lecteur/graveur/enregistreurs de disques. D'autres types de coupleurs permettent de connecter l'ordinateur à d'autres ordinateurs via un *réseau*. Dans les applications industrielles où une chaîne de production est pilotée par ordinateur il serait incongru de considérer la chaîne comme un périphérique ! Du point de vue du programmeur c'est pourtant le cas.

3.2 Les programmes et les données

Les programmes et les données peuvent être enregistrés sur des supports magnétiques ou en mémoire vive ou morte. Ils peuvent être présents (en partie) dans le processeur : à un instant donné l'instruction en cours d'exécution est dans dans le registre d'instruction du processeur. Cette information est dupliquée, on ne l'enlève pas de la mémoire pour l'exécuter. Les programmes peuvent être affichés à l'écran ou écrits sur une feuille de papier.

Sur un même disque optique ou magnétique, ou dans une mémoire, on peut trouver le texte source d'un programme et le format objet correspondant. Quand on achète un logiciel on n'achète généralement que le code objet. L'éditeur se protège ainsi contre la possibilité pour le client de modifier le logiciel. On achète aussi le plus souvent des données : dictionnaire du vérificateur orthographique, images des jeux, etc.

3.3 La vie du matériel et des programmes

Le matériel a une vie très simple : avant la mise sous tension, il ne fait rien. Certaines informations sont stockées en mémoire morte ou en mémoire secondaire. Aucun traitement n'a lieu ; à la mise sous tension, il se produit une

réinitialisation automatique (reset), qui fait démarrer le système matériel dans son *état initial*, et donc lance l'exécution du logiciel. Une réinitialisation peut avoir lieu à n'importe quel instant sur commande de l'utilisateur.

Tout ordinateur (sauf de très rares exceptions) est rythmé par un signal périodique nommé l'*horloge*. Ce signal cadence les changements d'états dans l'ordinateur. Un ordinateur dont la fréquence d'horloge est de 250 Mhz (Mégahertz) change d'état avec une période de 4 ns (nanosecondes, c'est-à-dire 4.10^{-9} secondes). L'existence de cette horloge permet de gérer une *pendule* qui donne l'heure à l'utilisateur et date ses fichiers. La précision n'a pas besoin d'être à la nanoseconde près évidemment. En revanche elle doit permettre de changer de siècle !

La vie des programmes est plus agitée ! Certains programmes sont inscrits en mémoire morte en usine, par le constructeur de l'ordinateur. Pour la construction de petits ordinateurs spécialisés, la technologie d'inscription des mémoires mortes est accessible assez facilement. Certains programmes comme les noyaux de systèmes d'exploitation ou les jeux sur cartouche, sont généralement sur un tel support.

Certains programmes se trouvent en mémoire vive. Ils n'y apparaissent pas par génération spontanée. Ils sont le résultat d'un cycle : édition, sauvegarde, traduction éventuelle, chargement (d'un support secondaire vers la mémoire vive). Ces étapes sont généralement suivies d'un lancement.

L'ordinateur comporte les outils logiciels nécessaire à ces actions : *éditeur de texte*, *gestion de fichiers*, *traducteur*, *chargeur*, *lanceur* sont pilotés par un utilisateur humain par l'intermédiaire d'un enchaîneur de travaux : l'*interprète de commandes*.

Dans les vieilles machines on pouvait entrer des programmes en mémoire et forcer le compteur programme directement en binaire, avec des interrupteurs à deux positions. Il n'y avait plus qu'à appuyer sur un bouton pour lancer l'exécution. C'était le bon temps !

4. Fonctionnalités des ordinateurs

Cette partie décrit différents usages de l'ordinateur. Cela nous permet ensuite de distinguer l'ordinateur de différentes machines *programmables* qui ne sont pas des ordinateurs.

4.1 Les usages de l'ordinateur

Distinguons deux usages bien différents des ordinateurs.

Certains ordinateurs ont atteint une destination finale ; c'est le cas par exemple de la console de jeux, du traitement de texte de la dactylographe, du système de réservation de la compagnie aérienne, de la station de travail en bu-

reau d'étude de mécanique, du contrôleur de programmation du magnétoscope, ou de la calculette *programmable*.

D'autres ordinateurs n'ont pas encore atteint ce stade. Pour l'instant ils ne servent qu'à des informaticiens pour écrire des programmes. Ils ne servent encore qu'à la mise au point d'une certaine destination finale. Souvent un même ordinateur peut servir à développer des jeux, un traitement de texte ou une calculette par *simulation*.

Certains ordinateurs peuvent être utilisés des deux manières : les programmeurs de la compagnie aérienne changent les programmes sur la machine sans interrompre les réservations, ce qui n'est pas forcément simple.

Sur certains ordinateurs il est possible d'utiliser des logiciels dits de *Conception Assistée par Ordinateur* (*CAO*) pour concevoir des voitures, des moteurs ou les puces qui seront à la base d'un futur ordinateur.

4.1.1 Fonctionnalités des ordinateurs pour non programmeurs

Remarquons que ces machines sont souvent qualifiées de *programmables*, comme c'est le cas pour les magnétoscopes.

Dans une telle machine il faut pouvoir introduire des informations et lancer des exécutions. Si l'on regarde de plus près, il faut pouvoir introduire des données (les textes du traitement de texte, les opérandes de la calculette non programmable) et des programmes dans le langage de la machine programmable visible. Par exemple pour le magnétoscope : enregistrer la chaîne x, de *h1* à *h2* heures, pendant N jours, tous les M jours. Le processeur n'exécute pas *directement* ce type de programmes qui ne sont pas écrits en langage machine. Le programme en langage machine qui est exécuté considère *enregistrer, x, h1, h2, N* et M comme des données. Ces *programmations* sont *interprétées* par un programme en langage machine qui est fourni avec la machine. Ce programme est totalement invisible pour l'utilisateur.

Par ailleurs il faut pouvoir lancer un tel programme en langage machine qui prend ces paramètres (*enregistrer*, *h1*,..) et s'exécute en tenant compte de leurs valeurs.

Cette double fonctionnalité permettant une phase de programmation et une phase d'exécution n'est pas facile à comprendre pour les utilisateurs non informaticiens. L'informaticien qui devra un jour écrire un mode d'emploi d'une telle machine doit s'en souvenir.

Dans de telles machines l'utilisateur peut parfois installer des programmes nouveaux qu'il se procure : jeux, nouvelle version de traitement de texte, etc. Ils sont déjà en langage machine ; il faut pouvoir mémoriser ces programmes sur un disque et les lancer. On est très proche alors d'un ordinateur.

4.1.2 Fonctionnalités des ordinateurs pour programmeurs

Dans ces machines il faut pouvoir écrire des programmes et les traduire en langage machine puis les charger et les lancer. La traduction d'un pro-

gramme écrit dans un langage *de haut niveau* en un texte en langage machine est une *compilation*. Il y a donc des programmes qui permettent d'écrire, sauver, traduire, lancer des programmes. Sur les ordinateurs utilisés pour le développement de programmes, les programmes peuvent, comme sur le magnétoscope, être interprétés.

Sur les ordinateurs compliqués où plusieurs programmeurs travaillent en même temps chacun veut quand même avoir l'impression d'avoir un ordinateur pour lui tout seul. C'est le cas de l'ordinateur de la compagnie aérienne qui gère les places, permet la mise au point de programmes, etc.

L'ensemble des outils permettant l'édition, la sauvegarde et le lancement de programmes pour un ou plusieurs utilisateurs constitue un *système d'exploitation*. Un système d'exploitation comporte 2 parties :

- Une partie *basse* fortement dépendante des caractéristiques du matériel comme le type de processeur ou les types de périphériques connectés (souris à 1, 2 ou 3 boutons, claviers AZERTY ou QWERTY, lecteurs de disquettes avec une vitesse de rotation plus ou moins grande). Des *bibliothèques* de programmes de gestion des périphériques, nommées les *pilotes de périphériques*, sont toujours livrées avec l'ordinateur ou avec le périphérique. L'installation de ces pilotes (*drivers* en anglais) cause bien des soucis aux utilisateurs novices. Cette partie basse comporte aussi les outils permettant de gérer plusieurs utilisateurs simultanés de l'ordinateur.
- Une partie *haute* utilisant les primitives de la précédente pour offrir des services de plus haut niveau. Par exemple : après une édition de texte, on le sauvegarde en utilisant le programme de *gestion de fichiers*. Ce gestionnaire vérifie si le fichier existe déjà, si sa date enregistrée est bien antérieure, etc. Mais la prise en compte de la vitesse de rotation du disque n'est pas du même niveau. Le système de gestion de fichiers suppose ces aspects plus *bas* déjà résolus. De même l'envoi d'un *courriel* (ou *mél*) utilise des parties de programmes qui ne dépendent pas du nombre de boutons de la souris.

4.2 Tout ce qui est programmable est-il un ordinateur ?

On rencontre de nombreux appareils électroménagers dotés d'un *séquenceur* ou *programmateur*. Ce dernier leur permet d'enchaîner automatiquement certaines actions, selon un ordre immuable figé lors de la construction de la machine (l'utilisateur a parfois le choix entre plusieurs séquences prédéfinies). C'est le cas des machines à laver.

Une machine à laver a un comportement cyclique complexe, avec rétroaction de l'environnement. Un moteur qui tourne déclenche ou non des actions selon la *programmation* manifestée par la position de certains points de contacts électriques ; les actions continuent ou s'arrêtent selon le temps écoulé, les informations provenant du détecteur de température, le niveau d'eau, etc. Les actions correspondant aux contacts sont faites dans l'ordre où les contacts sont touchés par un contacteur électrique.

On pourrait imaginer un comportement plus complexe dans lequel une action est ou n'est pas faite selon le résultat de l'action précédente. Imaginons un détecteur d'opacité de l'eau de rinçage : si l'eau est trop opaque, un rinçage supplémentaire a lieu.

Le matériel informatique a un tel comportement. Le processeur peut être assimilé à un moteur qui tourne. Le compteur programme, qui évolue périodiquement, évoque ce comportement : il passe devant des contacts, il pointe successivement sur des instructions, et effectue les actions correspondantes. Si les contacts disent de s'arrêter ou d'aller plus loin dès que l'action est terminée, cela se produit. Les intructions peuvent être conditionnelles. Elles peuvent comporter des ruptures de séquence. Dans ce cas les instructions ne sont plus exécutées dans l'ordre où elles sont écrites.

L'informatique est toutefois plus complexe qu'une simple machine à laver car *un programme peut avoir comme résultat de créer et d'écrire dans la mémoire un programme et de lui passer la main*, c'est-à-dire de le lancer. Les machines à laver n'en sont pas capables.

5. Plan du livre

Le livre comporte six parties.

La première partie donne des fondements pour toute l'informatique, logicielle et matérielle. Les outils mathématiques ne sont pas présentés ici pour eux-mêmes mais pour être utilisés dans la suite. Les mots *binaire, information, bit, automate, booléen, représentation, état, langage* seront alors familiers.

La deuxième partie donne les techniques propres au matériel. Nous y décrivons toutes les étapes qui permettent de représenter et traiter les vecteurs de 0 et de 1 sur du matériel. Les mots *puce, système séquentiel, mémoire, circuit, transistor* ne poseront plus de problème.

La troisième partie donne les techniques propres au logiciel. Après cette partie, on sait tout sur *langage, langage d'assemblage, langage machine, saut, branchement, registre.*

La quatrième partie est centrale. On y explique *comment* le processeur exécute les instructions. Ceci est fait de façon détaillée, en s'appuyant sur les connaissances acquises dans les trois premières parties. Après cette partie on a compris comment du matériel peut traiter du logiciel.

La cinquième partie donne tous les éléments pour construire un ordinateur au sens où nous venons de le définir. Cela suppose des ajouts de matériel autour du processeur et de la mémoire et l'introduction de programmes constituant le système d'exploitation. Après ce chapitre, on sait, de façon détaillée, comment *marche* l'ordinateur et comment on le conçoit. On pourrait donc s'arrêter là.

La sixième partie est nécessaire pour le professionnel de l'informatique. On montre comment peut être mis en place le système qui permet d'accepter plusieurs utilisateurs effectuant plusieurs tâches simultanément, ou tout au

moins avec l'apparence de la simultanéité.

Première partie

Outils de base de l'algorithmique logicielle et matérielle

Chapitre 2

Algèbre de Boole et fonctions booléennes

George Boole, mathématicien anglais, a utilisé pour la première fois en 1850 une algèbre à 2 éléments pour l'étude de la logique mathématique. Il a défini une algèbre permettant de modéliser les raisonnements sur les propositions vraies ou fausses. Etudiée après Boole par de nombreux mathématiciens, l'Algèbre de Boole a trouvé par la suite de nombreux champs d'application : réseaux de commutation, théorie des probabilités, recherche opérationnelle (étude des alternatives).

Les premières applications dans le domaine des calculateurs apparaissent avec les relais pneumatiques (ouverts ou fermés). Aujourd'hui, les ordinateurs sont composés de transistors électroniques fonctionnant sur 2 modes : bloqué ou passant (Cf. Chapitres 7 et 8). Ils utilisent une arithmétique binaire (Cf. Chapitre 3). L'algèbre de Boole constitue un des principaux fondements théoriques pour leur conception et leur utilisation. Les circuits sont des implémentations matérielles de fonctions booléennes.

Les fonctions booléennes peuvent être représentées et manipulées sous différentes formes. Ces représentations ont des intérêts variant suivant de nombreux critères. Selon la technologie de circuit cible, certaines représentations sont plus adéquates pour arriver à une implémentation optimisée. Une représentation peut bien convenir à certains types de fonctions et devenir très complexe, voire impossible à utiliser pour d'autres. Enfin, selon l'outil de CAO (Conception assistée par ordinateur) utilisé, certaines formes sont acceptées (car bien adaptées à une représentation sur machine) ou non.

Le paragraphe 1. présente les principales définitions concernant cette algèbre et les fonctions booléennes. Les différents moyens de représenter ces fonctions booléennes sont énumérés dans le paragraphe 2. Le paragraphe 3. décrit les différentes manipulations que l'on peut effectuer sur ces représentations afin d'obtenir des formes permettant par la suite une implémentation physique à moindre coût.

1. Algèbre de Boole

1.1 Opérations

Soit l'ensemble $\mathcal{B} = \{0, 1\}$. On définit une relation d'ordre total sur cet ensemble en posant : $0 \leq 1$. A partir de cette relation d'ordre, on définit les opérations suivantes sur les éléments de $\mathcal{B}$:

Addition : $x + y = max(x, y)$
Multiplication : $x.y = min(x, y)$
Complémentation : $\bar{x} = 0$ si $x = 1$ et $\bar{x} = 1$ si $x = 0$

On utilise les termes de *somme*, *produit* et *complément* pour les résultats de l'addition, de la multiplication et de la complémentation. Le résultat de ces opérations est détaillé dans la table suivante :

a	b	$a+b$	$a.b$	$\bar{a}$
0	0	0	0	1
1	0	1	0	0
0	1	1	0	-
1	1	1	1	-

1.2 Définition

Soit $\mathcal{A}$ un ensemble non vide comportant deux éléments particuliers notés 0 et 1. On définit sur l'ensemble $\mathcal{A}$ deux opérations binaires notées + et . et une opération unaire notée $^-$.

$(\mathcal{A}, 0, 1, +, ., ^-)$ est une *algèbre de Boole* s'il respecte les axiomes suivants :

1. L'addition et la multiplication sont commutatives et associatives.

   $\forall a \in \mathcal{A}, \forall b \in \mathcal{A} : a + b = b + a$ et $a.b = b.a$
   $\forall a \in \mathcal{A}, \forall b \in \mathcal{A}, \forall c \in \mathcal{A} : (a + b) + c = a + (b + c)$ et $(a.b).c = a.(b.c)$

   Remarque : On pourra ainsi noter de façon équivalente $(a.b).c$ ou $a.b.c$; de même : $a + (b + c)$ ou $a + b + c$.

2. 0 est élément neutre pour l'addition et 1 est élément neutre pour la multiplication.

   $$\forall a \in \mathcal{A} : 0 + a = a$$
   $$\forall a \in \mathcal{A} : a.1 = a$$

3. L'addition et la multiplication sont distributives l'une par rapport à l'autre : $\forall a \in \mathcal{A}, \forall b \in \mathcal{A}, \forall c \in \mathcal{A} : (a + b).c = a.b + a.c$ et $(a.b) + c = (a + c).(b + c)$.

   Remarque : L'usage a consacré la priorité de la multiplication sur l'addition comme dans la notation algébrique usuelle. Par souci de simplification d'écriture, on notera de façon équivalente : $(a.b) + c$ ou $a.b + c$.

4. Pour tout élement, la somme d'un élément et de son complémentaire est égale à 1 et le produit d'un élément et de son complémentaire est égal à 0 : $\forall a \in \mathcal{A} : \bar{a} + a = 1$ et $\forall a \in \mathcal{A} : a.\bar{a} = 0$.

1.3 Exemples d'Algèbres de Boole

L'algèbre de Boole la plus simple est définie sur l'ensemble à deux éléments : $\mathcal{B} = \{0, 1\}$. Pour l'étude des raisonnements sur les propositions logiques, il existe des synonymes pour les noms des éléments de cet ensemble et des opérations ; on parle alors de faux et vrai (au lieu de 0 et 1) et des opérateurs et et ou (au lieu de la multiplication et de l'addition). Les définitions et les propriétés mathématiques restent identiques. Ces termes sont utilisés aussi dans l'étude des circuits logiques.

L'ensemble des parties d'un ensemble E (noté $\mathcal{P}(\mathrm{E})$) muni des opérations d'intersection ensembliste (correspondant à .), d'union ensembliste (correspondant à +) et de complémentaire ensembliste dans E (correspondant à $^-$) forme une algèbre de Boole. L'ensemble vide correspond à 0 et l'ensemble E à 1.

L'ensemble des nuplets de booléens muni des opérations d'addition, de multiplication et de complémentation étendues aux vecteurs forme une algèbre de Boole. $(0, 0, \ldots, 0)$ correspond à 0 et $(1, 1, \ldots, 1)$ à 1.

$(x_1, x_2, ..., x_n) + (y_1, y_2, ..., y_n) = (x_1 + y_1, x_2 + y_2, ..., x_n + y_n)$
$(x_1, x_2, ..., x_n).(y_1, y_2, ..., y_n) = (x_1.y_1, x_2.y_2, ..., x_n.y_n)$
$\overline{(x_1, x_2, ..., x_n)} = (\bar{x_1}, \bar{x_2}, ..., \bar{x_n})$

1.4 Principaux théorèmes

Théorème de dualité : Si $(\mathcal{A}, 0, 1, +, ., ^-)$ est une algèbre de Boole alors $(\mathcal{A}, 1, 0, ., +, ^-)$ est aussi une algèbre de Boole.

Ainsi les axiomes et règles de simplification peuvent se présenter sous deux formes duales, l'une se déduisant de l'autre en remplaçant les + par des . et les 1 par des 0 et inversement.

Règles de simplification booléenne :

$\bar{\bar{a}} = a$	$\overset{duale}{\longleftrightarrow}$	$\bar{\bar{a}} = a$
$a + 1 = 1$	$\overset{duale}{\longleftrightarrow}$	$a.0 = 0$
$a + a = a$	$\overset{duale}{\longleftrightarrow}$	$a.a = a$
$a + a.b = a$	$\overset{duale}{\longleftrightarrow}$	$a.(a + b) = a$
$a + \bar{a}.b = a + b$	$\overset{duale}{\longleftrightarrow}$	$a.(\bar{a} + b) = a.b$
$a.b + \bar{a}.b = b$	$\overset{duale}{\longleftrightarrow}$	$(a + b).(\bar{a} + b) = b$
$a.b + \bar{a}.c + b.c = a.b + \bar{a}.c$	$\overset{duale}{\longleftrightarrow}$	$(a + b).(\bar{a} + c).(b + c) = (a + b).(\bar{a} + c)$

x_1	x_2	x_3	y
0	0	0	**1**
0	0	1	**1**
0	1	0	**0**
0	1	1	**0**

x_1	x_2	x_3	y
1	0	0	**1**
1	0	1	**1**
1	1	0	**0**
1	1	1	**1**

FIG. 2.1 – Table de vérité de la fonction : $y = f(x_1, x_2, x_3)$

Règles de De Morgan

$$\overline{a.b} = \bar{a} + \bar{b} \quad \overset{duale}{\longleftrightarrow} \quad \overline{a+b} = \bar{a}.\bar{b}$$

On peut généraliser à n variables :

$$\overline{x_1.x_2.\ldots.x_n} = \bar{x}_1 + \bar{x}_2 + \ldots + \bar{x}_n \quad \overset{duale}{\longleftrightarrow} \quad \overline{x_1 + x_2 + \ldots + x_n} = \bar{x}_1.\bar{x}_2.\ldots.\bar{x}_n$$

2. Fonctions booléennes

2.1 Fonctions booléennes simples

2.1.1 Définitions

On appelle *fonction booléenne simple* une application de $\{0,1\}^n$ dans $\{0,1\}$:

$$(x_1, x_2, ..., x_n) \overset{f}{\longmapsto} f(x_1, x_2, ..., x_n)$$

$(x_1, x_2, ..., x_n)$ est appelée *variable booléenne générale*. f est appelée *fonction à n variables*. Une valeur donnée de $(x_1, x_2, ..., x_n)$ est appelée *point* de la fonction.

La façon la plus simple de définir une fonction est de donner la liste de ses valeurs en chaque point. On peut le faire sous la forme d'un tableau que l'on appelle aussi *table de vérité*. La figure 2.1 donne la table de vérité d'une fonction à 3 variables.

L'ensemble des points de la fonction forme le *domaine* de la fonction. On dit qu'une fonction couvre tous les points pour lesquelles elle vaut 1 (sous-ensemble du domaine pour lequel la fonction vaut 1). La fonction f définie par la table 2.1 couvre les points $(0,0,0)$, $(0,0,1)$, $(1,0,0)$,$(1,0,1)$ et $(1,1,1)$.

> **Remarque :** Une fonction booléenne peut servir à représenter un ensemble : la fonction vaut 1 en chacun des points appartenant à l'ensemble. On parle de *fonction caractéristique*.

2.1.2 Les fonctions à 2 variables

Les fonctions à deux variables sont définies sur les 4 points $(0,0), (0,1), (1,0), (1,1)$. En chacun de ces 4 points une certaine fonction peut prendre une des deux valeurs 0 ou 1. Il y a donc $2^4 = 16$ fonctions possibles.

x_1	x_2	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

x_1	x_2	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

FIG. 2.2 – Les tables de vérité des 16 fonctions à deux variables

Les tables de vérité des 16 fonctions à deux variables sont listées dans la figure 2.2. f_1 et f_7 correspondent respectivement à la multiplication et l'addition algébriques vues auparavant.

2.1.3 Duale d'une fonction

On appelle *duale* d'une fonction f la fonction notée f^* telle que : $f^*(X) = \overline{f(\bar{X})}$. On dit qu'une fonction est autoduale si $f^*(X) = f(X), \forall X$.

2.2 Fonctions booléennes générales

On appelle *fonction booléenne générale* une application F de $\{0,1\}^n$ dans $\{0,1\}^m$:

$(x_1, x_2, \ldots, x_n) \overset{F}{\longmapsto} (f_1(x_1, x_2, ..., x_n), f_2(x_1, x_2, ..., x_n), \ldots, f_m(x_1, x_2, ..., x_n))$.
Une fonction booléenne générale est un m-uplet de fonctions simples :

$F = (f_1, f_2, \ldots, f_m)$.

2.3 Relations d'ordre

L'ordre défini sur $\mathcal{B}$ est étendu aux variables générales et aux fonctions booléennes.

La relation d'ordre partiel sur les variables booléennes générales est définie par : $(x_1, x_2, ..., x_n) \leq (y_1, y_2, ..., y_n)$ si et seulement si $\forall j, x_j \leq y_j$. Par exemple $(0,0,1) \leq (0,1,1)$. En revanche, $(1,0,1)$ et $(0,1,0)$ ne sont pas comparables.

La relation d'ordre partiel sur les fonctions booléennes simples est définie comme suit. La fonction f est inférieure à la fonction g si et seulement si pour tout point P : $f(P) \leq g(P)$; c'est-à-dire si tous les points couverts par f sont couverts par g.

> **Remarque :** Si f et g sont respectivement les fonctions caratéristiques des ensembles A et B, $f \leq g$ signifie que A est inclus dans B.

La relation d'ordre partiel sur les fonctions booléennes générales est définie comme suit. La fonction générale $F = (f_1, f_2, \ldots, f_m)$ est inférieure à la fonction $G = (g_1, g_2, \ldots, g_m)$ si pour tout i dans $1..m$, on a $f_i \leq g_i$.

2.4 Fonctions phi-booléennes

Une fonction booléenne partielle est une fonction booléenne dont la valeur n'est pas définie en chaque point. Dans la pratique les fonctions partielles sont utilisées pour définir des fonctions dont la valeur en certains points est indifférente ou dont la valeur des entrées en certains points est impossible.

On peut coder une fonction partielle f par une fonction totale dont le codomaine est complété par une valeur appelée Φ. La valeur Φ est associée aux points non déterminés de f. Une telle fonction est dite phi-booléenne.

Définition On appelle fonction *phi-booléenne* une application f de $\{0,1\}^n$ dans $\{0,1,\Phi\}$: $(x_1, x_2, ..., x_n) \stackrel{f}{\longmapsto} f(x_1, x_2, ..., x_n)$.

> **Remarque :** Le choix de la lettre Φ vient de sa forme qui peut être vue comme la superposition d'un 1 et d'un 0.

Exemple E2.1 : Une fonction phi-booléenne
Soit la fonction à 4 variables $f(x_1, x_2, x_3, x_4)$ qui vaut 1 si l'entier compris entre 0 et 9, codé en binaire sur 4 bits correspondant aux valeurs de x_1, x_2, x_3, x_4, est pair et 0 sinon. Cette fonction est partielle puisque sa valeur est indifférente pour les points correspondant à des valeurs comprises entre 10 et 15. On peut la coder en fonction phi-booléenne en associant la valeur Φ à chacun des points $(1,0,1,0)$, $(1,0,1,1)$, $(1,1,0,0)$, $(1,1,0,1)$, $(1,1,1,0)$ et $(1,1,1,1)$.

Bornes d'une fonction phi-booléenne Soit f une fonction phi-booléenne. La borne supérieure de f est obtenue en remplaçant tous les Φ par des 1. Elle est notée SUP(f). La borne inférieure de f est obtenue en remplaçant tous les Φ par des 0. Elle est notée INF(f). Si nous étendons la relation d'ordre donnée sur $\mathcal{B}$ sur $\{0,1,\Phi\}$ en posant $0 \leq \Phi \leq 1$, nous avons : INF(f) $\leq$ f $\leq$ SUP(f). Le tableau ci-dessous donne les bornes supérieure et inférieure d'une fonction phi-booléenne :

x_1	x_2	f	INF(f)	SUP(f)
0	0	Φ	0	1
1	0	1	1	1
0	1	Φ	0	1
1	1	0	0	0

3. Représentation des fonctions booléennes

Comme nous l'avons vu précédemment la façon la plus simple de représenter une fonction est de donner la liste de ses valeurs. Cette représentation, dite *en extension*, n'est malheureusement plus possible dès que le nombre de variables augmente. En effet une fonction à n variables comporte 2^n valeurs. De nombreux types de représentation plus compactes, dites *en compréhension* existent. Leur utilisation varie principalement suivant leur degré de complexité et de facilité de manipulation à des fins d'implémentation matérielle (Cf. Chapitre 8).

Nous donnons dans cette partie trois types de représentations, très utilisées aujourd'hui : les expressions algébriques, les tableaux de Karnaugh et les BDD. Outre les représentations des fonctions simples nous montrons comment représenter une fonction générale à l'aide de la représentation des m fonctions simples qui la composent.

3.1 Expressions algébriques

3.1.1 Définitions

Expression booléenne algébrique : Toute fonction booléenne peut être représentée par une expression algébrique construite à partir des noms des variables simples de la fonction, des constantes 0 et 1, et des opérations de l'algèbre de Boole. Par exemple, $f(x_1, x_2) = x_1.(x_2 + x_1.\bar{x_2})$ ou $g(x_1, x_2) = 1.(0 + x_1.\bar{x_2})$.

Cette représentation n'est pas unique. Par exemple, $x_1.(x_2 + x_1.\bar{x_2}) + \bar{x_2}$ et $x_1 + \bar{x_2}$ sont deux expressions algébriques d'une même fonction.

Littéral : On appelle *littéral* l'occurrence d'une variable ou de son complément dans une expression algébrique. Les littéraux apparaissant dans l'expression de la fonction f définie ci-dessus sont : $x_1, x_2, \bar{x_2}$.

Monôme : On appele *monôme* un produit de p littéraux distincts. Par exemple, $x_1.x_2.\bar{x_3}$. Un monôme est dit *canonique* pour une fonction s'il contient toutes les variables de la fonction. Chaque ligne de la table de vérité correspond à un monôme canonique. On note dans le monôme $\bar{x}$ si la variable x vaut 0, x si elle vaut 1. Dans la table 2.1 la deuxième ligne correspond au monôme canonique $\bar{x_1}.\bar{x_2}.x_3$.

Forme polynômiale : On dit qu'une expression est sous forme *polynômiale* si elle est écrite sous forme de somme de monômes. Par exemple, $f(x_1, x_2) = x_1 + x_1.\bar{x_2}$.

3.1.2 Théorème de Shannon

Soit $f(x_1, x_2, ..., x_n)$ une fonction simple de $\mathcal{B}^n$ dans $\mathcal{B}$:

$\forall i \in 1..n$
$f(x_1, x_2, ..., x_n) = \bar{x_i}.f(x_1, x_2, \ldots, 0, ..., x_n) + x_i.f(x_1, x_2, \ldots, 1, ..., x_n)$

$f(x_1, \cdots, x_{i-1}, 1, x_{i+1}, \cdots, x_n)$ et $f(x_1, \cdots, x_{i-1}, 0, x_{i+1}, \cdots, x_n)$ sont appelés *cofacteurs* positif et négatif de f par rapport à la variable x_i. Ils sont notés respectivement f_{x_i} et $f_{\overline{x_i}}$.

La décomposition de Shannon sur la variable x_i s'écrit : $f = \overline{x_i}.f_{\overline{x_i}} + x_i.f_{x_i}$. Cette décomposition est unique.

Il existe la forme duale du théorème de Shannon :
$f(x_1, x_2, ..., x_n) = (\bar{x_i} + f(x_1, x_2, \ldots, 1, ..., x_n)).(x_i + f(x_1, x_2, \ldots, 0, ..., x_n))$

3.1.3 Formes de Lagrange

En appliquant successivement le théorème de Shannon sur toutes les variables de la fonction, on obtient une forme polynômiale composée de tous les monômes canoniques affectés de coefficients correspondant aux valeurs de la fonction.

Par exemple, pour une fonction à deux variables on obtient :

$f(x_1, x_2) = \bar{x_1}.f(0, x_2) + x_1.f(1, x_2)$
$f(x_1, x_2) = \bar{x_1}.(\bar{x_2}.f(0, 0) + x_2.f(0, 1)) + x_1.(\bar{x_2}.f(1, 0) + x_2.f(1, 1))$
$f(x_1, x_2) = \bar{x_1}.\bar{x_2}.f(0, 0) + \bar{x_1}.x_2.f(0, 1) + x_1.\bar{x_2}.f(1, 0) + x_1.x_2.f(1, 1)$

Cette forme est appelée *première forme de Lagrange*. Toute fonction possède une et une seule forme de ce type. C'est une expression canonique. On simplifie en général cette forme en supprimant tous les monômes dont le coefficient est 0 et en enlevant les coefficients à 1.

Exemple E2.2 : Première forme de Lagrange d'une fonction

Soit h une fonction à deux variables définie par la table ci-contre. Son expression algébrique sous la première forme de Lagrange est :
$h(x_1, x_2) = \bar{x_1}.\bar{x_2}.1 + \bar{x_1}.x_2.1 + x_1.\bar{x_2}.0 + x_1.x_2.0 =$
$\bar{x_1}.\bar{x_2} + \bar{x_1}.x_2$ *qui se simplifie en* $\bar{x_1}$.

x_1	x_2	y
0	*0*	**1**
0	*1*	**1**
1	*0*	**0**
1	*1*	**0**

En utilisant la forme duale du théorème de Shannon, on obtient la *deuxième forme de Lagrange*, un produit de sommes appelées *monales*.

3.1.4 Expressions polynômiales des fonctions à 2 variables

La figure 2.3 donne l'expression polynômiale des 16 fonctions de deux variables booléennes.

En logique, la somme est aussi appelée disjonction alors que dans le domaine des circuits, c'est l'opération ou exclusif qui est appelée disjonction.

Fonctions	Expressions	Noms usuels
f_0	0	
f_1	$x_1.x_2$	et, and, produit
f_2	$x_1.\bar{x_2}$	
f_3	x_1	
f_4	$\bar{x_1}.x_2$	
f_5	x_2	
f_6	$\bar{x_1}.x_2 + x_1.\bar{x_2}$	ou exclusif
f_7	$x_1 + x_2$	ou, or, somme
f_8	$\overline{x_1 + x_2}$	ni, non ou, nor
f_9	$\bar{x_1}.\bar{x_2} + x_1.x_2$	conjonction
f_{10}	$\bar{x_2}$	complément de x_2
f_{11}	$x_1 + \bar{x_2}$	
f_{12}	$\bar{x_1}$	complément de x_1
f_{13}	$\bar{x_1} + x_2$	implication
f_{14}	$\overline{x_1.x_2}$	exclusion, non et, nand
f_{15}	1	tautologie

FIG. 2.3 – Expression polynômiale des fonctions à deux variables

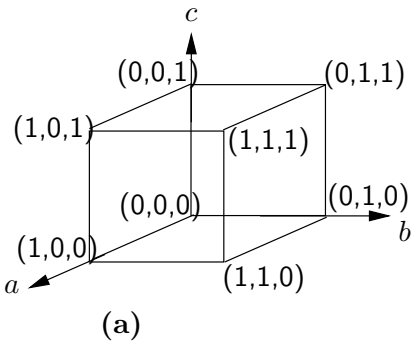
3.2 Tableaux de Karnaugh

3.2.1 Définition

Un *tableau de Karnaugh* est une représentation particulière de la table de vérité permettant de manipuler facilement (à la main) les différentes formes algébriques polynômiales d'une fonction ; nous le définissons ici et verrons au paragraphe 4. comment l'utiliser.

Un tableau de Karnaugh se présente comme une table à plusieurs entrées, chaque variable de la fonction apparaissant sur une des entrées. Par exemple, la figure 2.7 représente un tableau de Karnaugh pour une fonction à 3 variables et la figure 2.5 le tableau de Karnaugh d'une fonction à 4 variables.

Dans un tableau de Karnaugh, une seule variable change de valeur entre deux cases voisines verticalement ou horizontalement (on parle de cases adjacentes). Dans l'exemple de la figure 2.5, entre les cases de la deuxième et la troisième colonne seule la variable a change. Le tableau peut être vu comme un hypercube où chaque sommet correspond à un point de la fonction. Deux sommets sont adjacents s'il existe dans l'hypercube une arête entre eux (Cf. Figure 2.4).



	00	01	11	10
0	*	O		O
1	O			

(b)

FIG. 2.4 – a) Représentation sur un hypercube à 3 dimensions d'une fonction à trois variables a, b et c. b) Présentation du tableau de Karnaugh associé ; les cases marquées d'un O sont adjacentes à la case marquée d'une étoile.

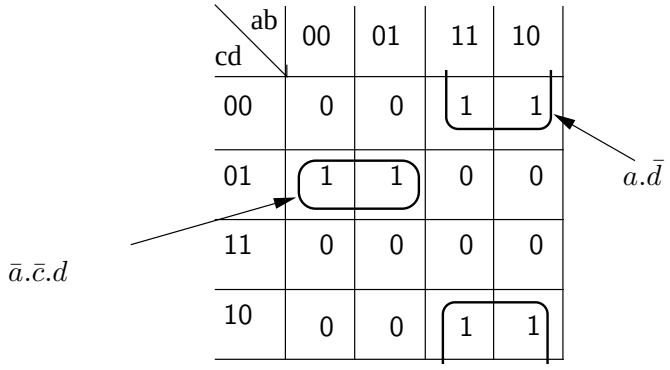


FIG. 2.5 – Un tableau de Karnaugh à 4 variables

3.2.2 Obtention d'une somme de monômes à partir d'un tableau de Karnaugh

En dimension 2, les colonnes et lignes d'un tableau de Karnaugh sont agencées de telle façon qu'un monôme de la fonction corresponde à un rectangle de 2^n cases adjacentes portant la valeur 1. Un tel regroupement de cases correspond à la simplification d'une somme de monômes en un seul monôme. Les cases de la première ligne (resp. colonne) sont adjacentes à celle de la dernière. Ainsi les 4 cases des coins d'un tableau de Karnaugh à 4 variables peuvent aussi former un monôme. Une fois les regroupements effectués l'obtention des variables du monôme se fait aisément. Ce sont celles qui ne changent pas de valeur entre les différentes cases correspondant au monôme.

Sur l'exemple de la figure 2.5, le monôme correspondant au regroupement de 4 cases est $a.\bar{d}$ puisque a possède la valeur 1 pour ces 4 cases et d possède la valeur 0 pour ces 4 cases. Il correspond à la simplification suivante à partir des 4 monômes canoniques : $a.b.\bar{c}.\bar{d} + a.\bar{b}.\bar{c}.\bar{d} + a.b.c.\bar{d} + a.\bar{b}.c.\bar{d} = a.b.\bar{d}.(c + \bar{c}) +$

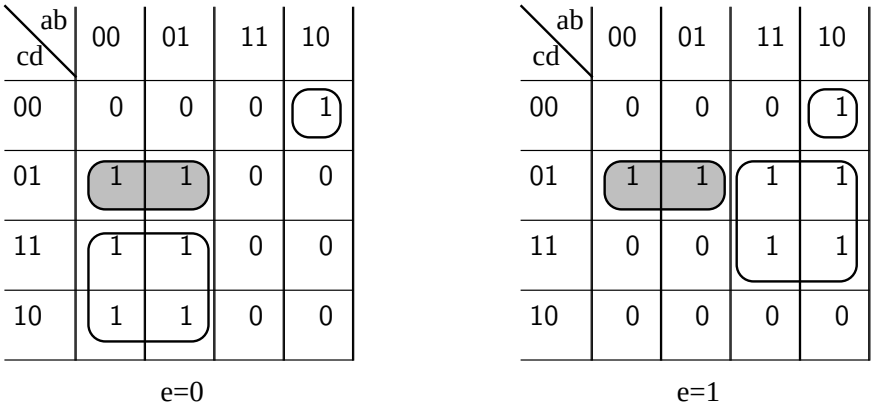


FIG. 2.6 – Un tableau de Karnaugh à 5 variables

$a.\bar{b}.\bar{d}.(c+\bar{c}) = a.b.\bar{d} + a.\bar{b}.\bar{d} = a.\bar{d}.(b+\bar{b}) = a.\bar{d}.$

Ce type de représentation est bien adapté aux fonctions de 2 à 5 variables. Les fonctions à 5 variables peuvent être représentées sur deux tableaux de Karnaugh à 4 variables (l'un pour une des variables à 0, l'autre pour cette même variable à 1). Deux cases situées à la même place sur les 2 tableaux sont adjacentes. Sur la figure 2.6, les 2 regroupements grisés sont un seul monôme : $\bar{a}.\bar{c}.d$. Il correspond à la simplification à partir des 4 monômes canoniques suivants : $\bar{a}.\bar{b}.\bar{c}.d.\bar{e}+\bar{a}.b.\bar{c}.d.\bar{e}+\bar{a}.\bar{b}.\bar{c}.d.e+\bar{a}.b.\bar{c}.d.e = \bar{a}.\bar{c}.d.\bar{e}.(b+\bar{b})+\bar{a}.\bar{c}.d.e.(b+\bar{b}) = \bar{a}.\bar{c}.d.\bar{e} + \bar{a}.\bar{c}.d.e = \bar{a}.\bar{c}.d.(e+\bar{e}) = \bar{a}.\bar{c}.d.$

L'expression polynômiale de la fonction définie par les tableaux de Karnaugh de la figure 2.6 est $\bar{a}.\bar{c}.d + \bar{a}.c.\bar{e} + a.d.e + a.\bar{b}.\bar{c}.\bar{d}$.

On peut procéder de la même manière pour des fonctions à 6 variables en dessinant 4 tableaux à 4 variables, au-delà cela devient inextricable.

3.2.3 Obtention d'un produit de monaux

On peut obtenir facilement une forme composée des monaux d'une fonction (forme duale) à partir de son tableau de Karnaugh. Pour cela on regroupe les cases adjacentes comportant des 0. Les variables du monal sont celles qui ne changent pas mais sont données sous forme complémentée par rapport à leur valeur.

L'expression algébrique sous forme produit de monaux de la fonction f définie par le tableau de Karnaugh de la figure 2.7 est : $f(a,b,c) = (a+\bar{c})(\bar{b}+\bar{c})$.

3.3 Graphes de décision binaire

Les *graphes de décision binaire* (en anglais *Binary Decision Diagram* : BDD) ont été introduits par Akers et Bryant dans les années 80 ([Bry86]). Ils sont utilisés dans les outils de C.A.O. de synthèse logique depuis une di-

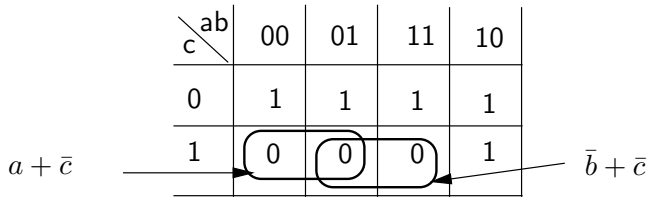


FIG. 2.7 – Monaux sur un tableau de Karnaugh

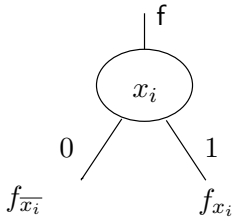


FIG. 2.8 – La décomposition de Shannon représentée par un arbre binaire

zaine d'années. Ils permettent de représenter et de manipuler des fonctions booléennes de grande taille.

Nous allons définir tout d'abord les arbres de Shannon, un BDD étant un graphe acyclique orienté ayant les mêmes chemins que l'arbre de Shannon associé mais dans lequel il n'y a pas de redondances. Tout l'intérêt des BDD est que l'on sait les construire, à coût algorithmique intéressant, à partir d'une autre représentation (par exemple, une forme algébrique) sans avoir à construire l'arbre de Shannon (Cf. Paragraphe 4.3).

3.3.1 Arbre de Shannon

On peut représenter la décomposition de Shannon par un arbre binaire où la racine est étiquetée par la variable de décomposition, le fils droit par le cofacteur positif et le fils gauche par le cofacteur négatif (Cf. Figure 2.8).

Si l'on itère la décomposition de Shannon avec cette représentation sur les deux cofacteurs, pour toutes les variables de f, on obtient un arbre binaire, appelé *arbre de Shannon*, dont les feuilles sont les constantes 0 et 1 et les noeuds sont étiquetés par les variables de la fonction (Cf. Figure 2.9-a sans tenir compte des parties grisées).

Un tel arbre est une représentation équivalente à la table de vérité de la fonction. Les valeurs de la fonction se trouvent sur les feuilles de l'arbre. Pour une valeur donnée de la fonction, la valeur de chaque variable est donnée par l'étiquette de l'arc emprunté pour aller de la racine à la feuille correspondante. Sur l'exemple de la figure 2.9-a, la fonction f a comme première forme de Lagrange : $f(a,b,c) = \bar{a}.\bar{b}.\bar{c} + \bar{a}.b.\bar{c} + a.\bar{b}.\bar{c} + a.b.\bar{c} + a.b.c$.

Une fois fixé un ordre total sur les variables, étant donné l'unicité de la décomposition de Shannon, la représentation sous forme d'arbre de Shannon

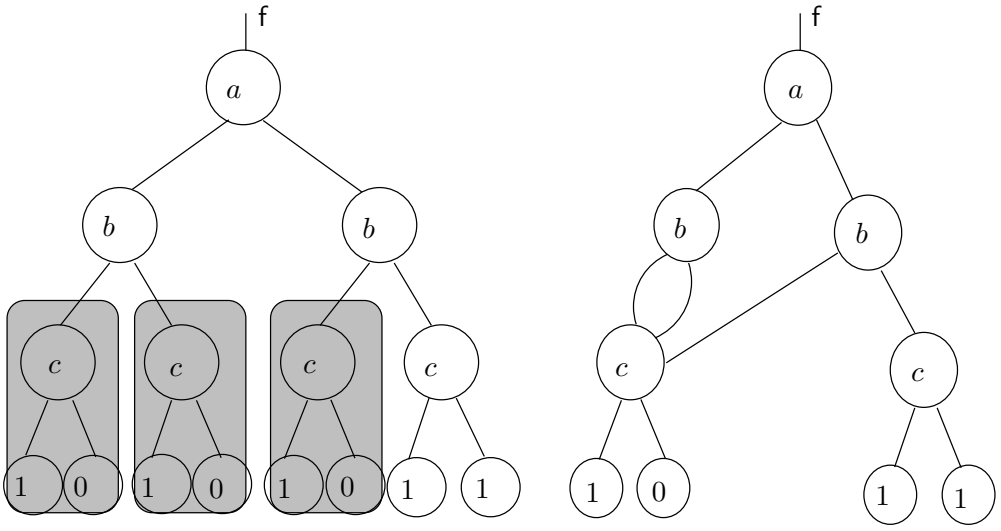


FIG. 2.9 – L'arbre de Shannon d'une fonction f à 3 variables a, b, c avec l'ordre : $a \ll b \ll c$ et le résultat de la mise en commun de 3 sous-arbres identiques.

est unique.

3.3.2 Redondance dans les arbres de Shannon

On définit la taille d'un arbre de Shannon par le nombre de ses noeuds. Un arbre de Shannon est de taille $2^n - 1$, n étant le nombre de variables de la fonction représentée.

Il existe dans cette représentation des redondances. Par exemple, certains sous-arbres sont identiques. La figure 2.9-b montre la mise en commun de trois sous-arbres identiques (ceux qui sont en grisés sur la partie a de la figure).

En considérant les feuilles comme des sous-arbres élémentaires, le graphe ne possède plus qu'un seul noeud à 1 et un seul noeud à 0. Pour l'exemple de la figure 2.9 on obtient le graphe de la figure 2.10-a.

On peut également éliminer les noeuds tels que tous les arcs sortants ont la même cible. Sur l'exemple précédent, on supprime ainsi deux noeuds (Cf. Figure 2.10-b).

Le graphe sans redondance est appelé graphe de décision binaire réduit. Dans le cas où il possède le même ordre de décomposition des variables sur tous ses chemins, on parle de ROBDD (en anglais *Reduced Ordered BDD*). Un ROBDD est encore canonique. La taille effective du ROBDD dépend de l'ordre choisi pour les variables. Un problème est de trouver un ordre optimal.

D'autres méthodes de simplification de BDD consistent à ajouter des informations supplémentaires sur les arcs, on parle alors de BDD typés. Le lecteur pourra trouver des informations détaillées sur ces méthodes dans [Bry86].

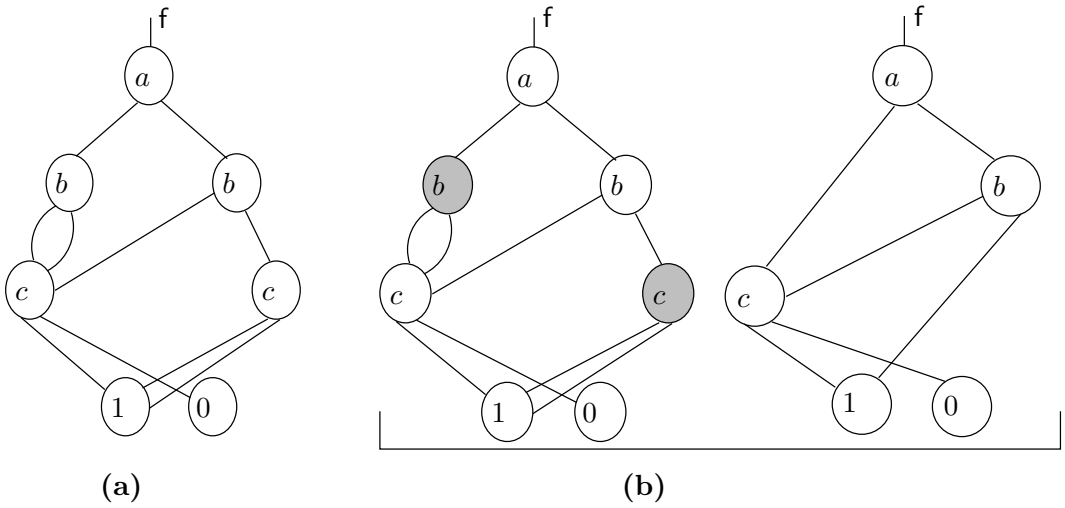


FIG. 2.10 – a) La mise en commun des feuilles à 1 et à 0. b) La suppression de noeuds qui n'apportent pas d'information.

4. Manipulation de représentations de fonctions booléennes

Un *circuit* est en général spécifié par une fonction booléenne. Nous verrons dans le chapitre 8 comment peut être réalisé le processus de synthèse d'une fonction booléenne vers une technologie cible. Selon les outils, la forme de départ de la fonction dans ce processus influe énormément sur la forme du circuit résultant et en particulier sur ses performances en terme de rapidité de calcul, surface en millimètres carrés, consommation électrique, etc. Ces critères de performance peuvent être traduits de façon plus ou moins précise en critères simples d'optimisation sur les formes des fonctions booléennes. Nous justifierons ces critères au chapitre 8.

Nous indiquons dans ce paragraphe quelques formes particulières et quelques méthodes pour manipuler l'expression d'une fonction booléenne à des fins d'optimisation en vue d'une implémentation physique.

Les premières méthodes basées sur les tableaux de Karnaugh ont vu le jour dans les années 50 [Kar53]. Elles permettent d'obtenir une forme polynômiale minimisée de façon manuelle. Des algorithmes permettant de trouver une forme polynômiale minimale ont ensuite été développé dans les années 60 et 70 [Kun65, Kun67]. Devant l'augmentation de la complexité des fonctions et des formes nécessaires pour une implémentation dans des technologies de plus en plus variées, des méthodes basées sur des représentations plus compactes, en particulier graphe de décision binaire, ont vu le jour depuis.

4.1 Formes particulières pour l'expression d'une fonction booléenne

La plupart des technologies cibles actuelles (Cf. Chapitre 8) nécessitent des décompositions des fonctions en expressions d'une forme particulière.

Pour certaines cibles technologiques une forme non polynômiale, appelée forme factorisée, peut être nécessaire. Le critère de minimisation est le nombre de littéraux apparaissant dans l'expression de la fonction. Par exemple, la forme algébrique de la fonction : $f(a,b,c,d) = \bar{a}.(\bar{b}.(\bar{c}+d)) + a.b$ possède 6 littéraux. Des méthodes ont été mises au point permettant d'automatiser la recherche de formes factorisées à nombre de littéraux minimal. Le lecteur pourra trouver des détails sur ces méthodes dans [BRWSV87]. Il est à noter que ces méthodes sont aujourd'hui souvent employées sur des formes en ROBDDs.

On peut vouloir aboutir à une représentation de forme quelconque mais à nombre de variables limité, par exemple, un ensemble de fonctions d'au plus 8 variables. On peut souhaiter une forme polynômiale à nombres de monômes et de variables limités, par exemple, un ensemble de fonctions d'au plus 10 monômes possédant chacun au plus 6 variables. Il peut aussi être nécessaire de représenter les fonctions avec un ensemble de formes fixées. Ce sont en général des formes de petite taille (nombre de variables inférieur à 6) avec des formes polynômiales ou factorisées fixées strictement. On parle alors de bibliothèque. On devra par exemple aboutir à un ensemble de fonctions possédant une des formes suivantes : $a+b$, $\bar{a}.\bar{b}$, $a.b$, $\bar{a}.b + a.c$ ou $\bar{a}.\bar{b} + a.b$.

Le problème est donc dans tous les cas de partir d'une fonction booléenne et d'aboutir à un ensemble de fonctions respectant la ou les formes imposées par la technologie cible. Des algorithmes propres à chaque technologie ont été développés.

Les critères d'optimisation sont alors le nombre de fonctions (qui est lié à la surface du circuit résultant) et le nombre d'étages de sous-fonctions imbriquées (qui est lié au temps de calcul du circuit). Le nombre d'étages maximal est appelé *chemin critique*.

Exemple E2.3 Soit la fonction f définie par l'expression algébrique : $f(a,b,c,d) = \bar{a}.b.c + a.\bar{b}.\bar{c} + c.d + \bar{c}.\bar{d}$. Regardons les solutions auxquelles on peut aboutir avec des formes cibles différentes.

Si l'on veut aboutir à des fonctions possédant au plus trois variables : $f(a,b,c,d) = SF + c.d + \bar{c}.\bar{d}$ avec $SF = \bar{a}.b.c + a.\bar{b}.\bar{c}$. Le nombre de littéraux est 11. Le nombre de fonctions imbriquées maximal est égal à 2 : (f, SF). Une nouvelle fonction appelée sous-fonction SF a été introduite pour parvenir à un ensemble de fonctions respectant le critère.

Si l'on veut aboutir à des fonctions possédant au plus deux monômes de trois variables : $f(a,b,c,d) = SF_1 + SF_2$ avec $SF_1 = \bar{a}.b.c + a.\bar{b}.\bar{c}$ et $SF_2 = c.d + \bar{c}.\bar{d}$. Deux sous fonctions ont été introduites. Le nombre de littéraux est 12 . Le chemin critique est égal aussi à 2.

Si l'on veut aboutir à des fonctions de la forme $a + b$ ou $\bar{a}.\bar{b}$ ou $a.b$ ou $\bar{a}.b + a.c$ ou $\bar{a}.\bar{b} + a.b$: $f(a, b, c, d) = SF_1 + SF_2$ avec $SF_1 = \bar{a}.SF_3 + a.SF_4$ et $SF_2 = c.d + \bar{c}.\bar{d}$. De plus $SF_3 = b.c$ et $SF_4 = \bar{b}.\bar{c}$. Le nombre de littéraux est 14. Le chemin critique est égal ici à 3 (f, SF_1, SF_3).

4.2 Obtention d'une forme polynômiale

De nombreuses technologies de réalisation des circuits nécessitent une forme polynômiale. Le critère d'optimisation est alors le nombre de monômes apparaissant dans la forme polynômiale puis le nombre de variables dans ces monômes. Un certain nombre de techniques de simplification respectant ce critère ont été mises au point et intégrées dans les outils modernes de CAO.

Nous montrons ici comment obtenir une forme polynômiale réduite à partir d'un tableau de Karnaugh. Cette méthode peut être pratiquée à la main pour des fonctions ayant moins de 6 variables.

4.2.1 Définitions

Monôme premier : un monôme m est dit *premier* pour une fonction f si l'on a : $m \leq f$ et s'il n'existe pas $m' \neq m$ tel que $m \leq m'$ et $m' \leq f$.

Base : on appelle *base* d'une fonction booléenne une forme polynômiale de la fonction composée uniquement de monômes premiers. On appelle *base complète* la base composée de tous les monômes premiers de la fonction.

Base irrédondante : une base est dite *irrédondante* si, dès que l'on ôte l'un des monômes qui la composent, on n'a plus une base de la fonction. Une fonction booléenne peut avoir plusieurs bases irrédondantes.

Le but de la minimisation est donc d'obtenir une base irrédondante possédant le minimum de monômes.

4.2.2 Obtention d'une forme polynômiale minimale à partir d'un tableau de Karnaugh

On peut obtenir de façon très visuelle les monômes premiers sur un tableau de Karnaugh à 2 dimensions en cherchant les pavés de cases adjacentes valant 1, les plus grands possible.

Exemple E2.4 La fonction représentée sur la figure 2.11 possède 5 monômes premiers : $M1 = b.\bar{c}$, $M2 = \bar{b}.c.d$, $M3 = \bar{a}.\bar{b}.\bar{d}$, $M4 = \bar{a}.\bar{c}.\bar{d}$, $M5 = \bar{a}.\bar{b}.c$. Elle possède 2 bases irrédondantes : $M1 + M2 + M4 + M5$ et $M1 + M2 + M3$. L'expression minimale (en nombre de monômes) de la fonction est : $f = b.\bar{c} + \bar{b}.c.d + \bar{a}.\bar{b}.\bar{d}$

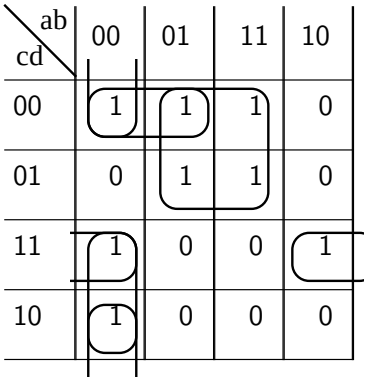


FIG. 2.11 – La base complète d'une fonction booléenne

4.2.3 Problème de la minimisation polynômiale

Le nombre de monômes premiers et de bases irrédondantes d'une fonction peut être très grand et une solution très longue à trouver. Le problème de la détermination d'une base irrédondante minimale est un problème NP-complet. La méthode consiste à trouver la base complète puis à extraire toutes les bases irrédondantes par essais successifs de suppression de chaque monôme. Devant l'accroissement de la taille des fonctions à manipuler, les outils de CAO sont pourvus de minimiseurs qui sont basés sur des algorithmes à base d'heuristiques diverses.

Cas des fonctions phi-booléennes : Les monômes premiers d'une fonction phi-booléenne sont ceux de sa borne supérieure. Une base d'une fonction phi-booléenne est une somme de monômes premiers telle que tous les points de la borne inférieure sont couverts par au moins un de ses monômes premiers. La méthode pour trouver une forme minimale d'une fonction phi-booléenne consiste à trouver tous les monômes premiers de la borne supérieure puis à trouver toutes les bases irrédondantes par rapport à sa borne inférieure.

Une fonction phi-booléenne est une fonction totale codant une fonction partielle et telle que chacun des Φ correspond à une valeur indifférente. On peut par conséquent associer à ce Φ la valeur 1 ou bien la valeur 0. Pratiquement, on va remplacer certains Φ par des 0 et d'autres par des 1 de façon à obtenir un minimum de monômes.

Exemple E2.5 Le monôme représenté sur le tableau de Karnaugh de la figure 2.12-a n'est pas premier pour f car il est plus petit que $a.c$ qui est un monôme de la borne supérieure. Les deux monômes $a.c$ et $b.\bar{c}.d$ de la figure 2.12-c suffisent pour couvrir les points de la borne inférieure.

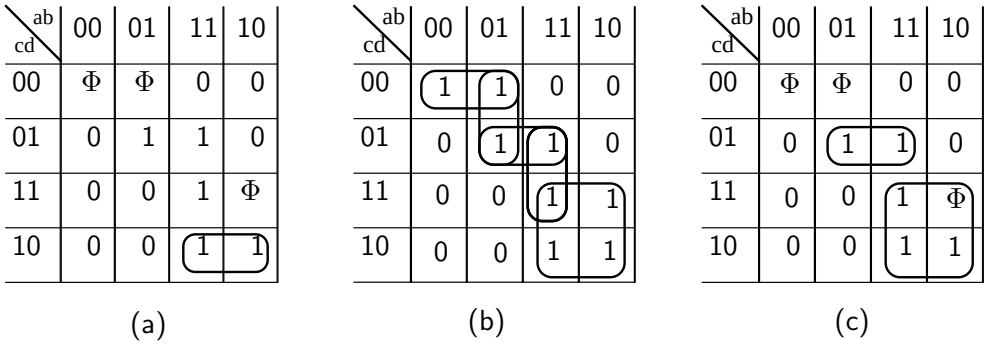


FIG. 2.12 – a) Une fonction Phi-Booléenne f. b) La base complète de la borne supérieure de f. c) La seule base irrédondante de f.

Cas des fonctions générales : Dans le cas d'une fonction générale $F = (f_1, f_2, \ldots, f_t)$ de $\mathcal{B}^n$ dans $\mathcal{B}^t$, le critère de minimisation est le nombre de monômes de l'ensemble des t fonctions simples qui composent la fonction générale.

Un *monôme général* M est un couple $(m, (v_1, \ldots, v_t))$ où m est un monôme d'une des fonctions f_i $(i = 1, \ldots, t)$ et $(v_1, \ldots, v_t)$ est un vecteur booléen. Il définit la fonction générale : $(v_1.m, \ldots, v_t.m)$. Par exemple, le monôme général associé à un monôme m ne figurant que dans l'expression de la fonction f_2 est : $(m, (0, 1, 0, \ldots, 0))$; le monôme général associé au monôme p figurant dans l'expression de chacune des fonctions f_i $(i = 1, \ldots, t)$ est : $(p, (1, 1, \ldots, 1))$.

Soient M et M' deux monômes généraux : $M = (m, (v_1, v_2, \ldots, v_t))$ et $M' = (m', (v'_1, v'_2, \ldots, v'_t))$.

On définit un ordre sur les monômes généraux : $M \leq M' \Leftrightarrow (m \leq m')$ et $(v_1, v_2, \ldots, v_t) \leq (v'_1, v'_2, \ldots, v'_t)$.

De la même façon que pour une fonction simple, on définit les *monômes généraux premiers*. Un monôme général M est dit premier pour une fonction générale F si l'on a : $M \leq F$ et s'il n'existe pas $M' \neq M$ tel que $M \leq M'$ et $M' \leq F$.

La méthode de minimisation en utilisant des tableaux de Karnaugh consiste à prendre tous les regroupements de cases maximaux sur plusieurs tableaux à la fois. Un regroupement de plusieurs cases dans un des tableaux de Karnaugh peut ne pas être maximal dans ce tableau mais correspondre à un monôme premier général parce qu'il apparaît dans plusieurs tableaux à la fois.

Pour obtenir tous ces monômes premiers généraux on cherche d'abord tous les monômes premiers de chaque fonction composant la fonction générale. On fait ensuite tous les produits possibles de ces monômes entre eux, le produit de deux monômes généraux étant défini par : $M.M' = (m.m', (v_1 + v'_1, v_2 +$

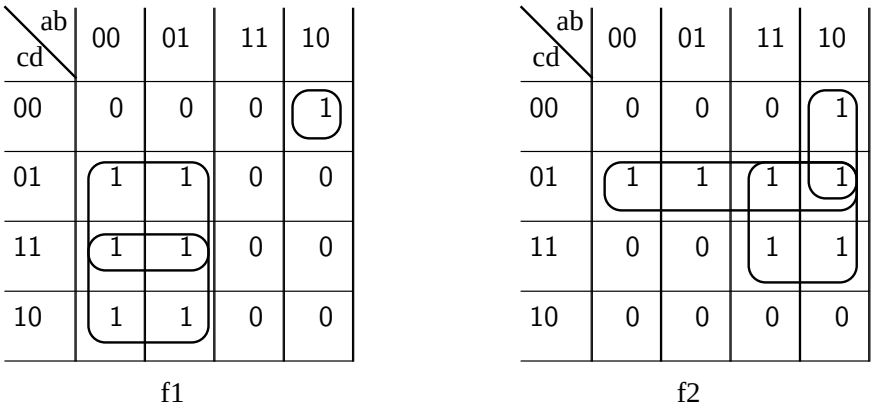


FIG. 2.13 – Les bases complètes des fonctions f_1 et f_2

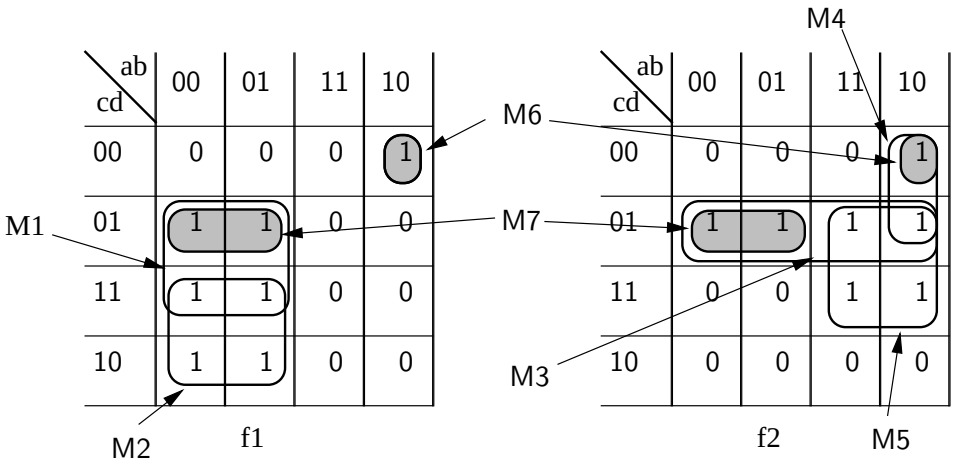


FIG. 2.14 – La base complète de la fonction générale $F = (f_1, f_2)$

$v'_2, \ldots, v_t + v'_t)$). Enfin, on ne garde que les monômes les plus grands.

Exemple E2.6 Sur la figure 2.13 sont représentées les deux bases complètes des fonctions f_1 et f_2. Ces deux bases sont irrédondantes.

Sur la figure 2.14 est représentée la base complète de la fonction générale $F = (f_1, f_2)$. Les monômes grisés sont des monômes généraux premiers obtenus par produits des monômes : $(a.\bar{b}.\bar{c}.\bar{d}, (1,0))$ et $(a.\bar{b}.\bar{c}, (0,1))$ pour $(a.\bar{b}.\bar{c}.\bar{d}, (1,1))$ et $(\bar{a}.d, (1,0))$ et $(\bar{c}.d, (0,1))$ pour $(\bar{a}.\bar{c}.d, (1,1))$.

Dans cet exemple, le monôme général $M7 = (\bar{a}.\bar{c}.d, (1,1))$ est premier car il n'existe pas de monôme plus grand que $\bar{a}.\bar{c}.d$ à la fois dans f_1 et f_2. Le monôme général $M5 = (a.d, (0,1))$ est premier car dans f_2, il n'existe pas de monôme plus grand que $(a.d)$ et $(a.d)$ n'est pas un monôme de f_1. La fonction générale $F = (f_1, f_2)$ possède 7 monômes premiers généraux $M1 = (\bar{a}.d, (1,0)), M2 = (\bar{a}.c, (1,0)), M3 = (\bar{c}.d, (0,1)), M4 = (a.\bar{b}.\bar{c}, (0,1)), M5 = (a.d, (0,1)), M6 = (a.\bar{b}.\bar{c}.\bar{d}, (1,1)), M7 = (\bar{a}.\bar{c}.d, (1,1))$. Les deux bases irrédondantes générales de

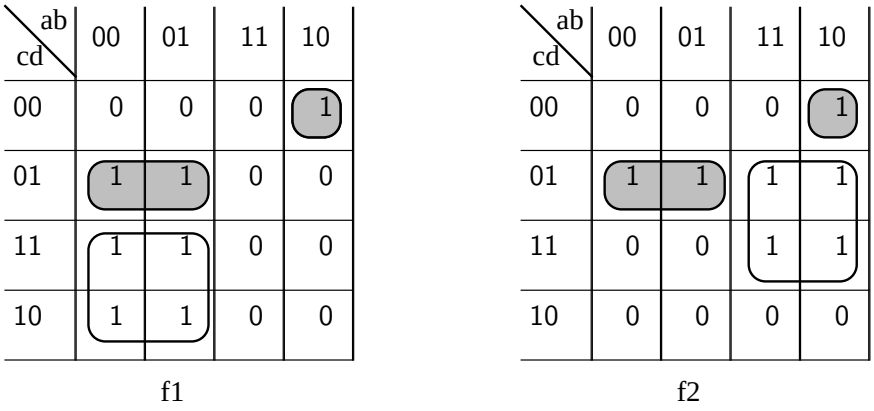


FIG. 2.15 – La base irrédondante minimale de la fonction générale F

F sont : $M1+M2+M3+M5+M6$ et $M2+M5+M6+M7$ et les 4 monômes de la base irrédondante minimale : $M6$, $M7$, $M5$ et $M2$.

4.3 Obtention de BDDs réduits ordonnés

L'utilisation des BDDs est aujourd'hui largement répandue car c'est une représentation très compacte et particulièrement adaptée à l'obtention de formes factorisées et à la décomposition en sous-fonctions.

Nous donnons ici les principes de base pour construire un BDD réduit ordonné à partir d'une expression booléenne algébrique. Le lecteur pourra trouver la justification et des détails d'implémentation logicielle de cette construction dans [KB90].

Nous avons vu qu'un BDD est un graphe de Shannon dans lequel il n'y a pas de redondances. Il s'agit de construire le BDD sans construire l'arbre de Shannon complet. Pour cela, on construit récursivement le BDD en évitant de fabriquer un sous-arbre déjà construit.

Soit un ordre donné sur les variables de la fonction. On effectue à chaque étape de la récursion la décomposition de Shannon suivant la variable courante et l'on construit le BDD à partir des BDDs des cofacteurs positif et négatif de f. Pour fixer les idées, nous donnons figure 2.16 un algorithme de spécification fonctionnelle de la construction d'un BDD à partir d'une expression algébrique booléenne. Lors de la mise en oeuvre de cet algorithme, il faut éviter de construire deux fois le même objet.

La fonction RepCanonique fabrique un nouveau BDD à partir d'une variable et de deux BDDs différents, si le BDD à construire n'existe pas, dans le cas contraire elle donne le graphe correspondant au BDD qui existait déjà. Pour la mettre en oeuvre, il faut définir une table avec adressage dispersé (hashcode en anglais), la fonction de dispersion portant sur la variable et les pointeurs des BDD fils. La comparaison de deux BDDs consiste en la comparaison des pointeurs associés aux racines des graphes.

Notations :
- /1\ : le BDD représentant la valeur 1
- /0\ : le BDD représentant la valeur 0
- /G, r, D\ : un arbre binaire de racine r, de fils gauche G et de fils droit D

Fonction principale :

LeBdd (e : une expression algébrique) $\longrightarrow$ un BDD

{ *e étant une expression booléenne, LeBdd(e) est le BDD associé à e.* }

LeBdd (1) = /1\
LeBdd (0) = /0\
LeBdd (e1 op e2) = TBop (LeBdd(e1), LeBdd(e2))
LeBdd (op e) = TUop (LeBdd(e))

Fonctions intermédiaires

TBop (b1, b2 : deux BDD) $\longrightarrow$ un BDD

{ *b1 et b2 sont deux BDD. Il existe une fonction TBop par opérateur binaire traité : elle fabrique le BDD résultat de l'application de l'opérateur en question aux deux BDD b1 et b2.* }

TUop (b : un BDD) $\longrightarrow$ un BDD

{ *b est un BDD. Il existe une fonction TUop par opérateur unaire pris en compte : elle produit le BDD résultat de l'application de l'opérateur au BDD b* }

RepCanonique (x une variable ; b1,b2 : deux BDD) $\longrightarrow$ un BDD

{ *RepCanonique(x, b, b) = b. x étant une variable, b1 et b2 deux BDDs différents, RepCanonique(x, b1, b2) est le BDD de racine x, de fils gauche b1 et de fils droit b2. Ce BDD n'est construit que s'il n'existe pas dans l'ensemble des BDD déjà construits* }

Exemple pour l'opérateur OR

{ *On applique les règles de simplification triviales associées à l'opérateur or : vrai or e = vrai, faux or e = e, e or e = e* }

TBor (1,b) = /1\
TBor (0,b) =b
TBor (b, 1) = /1\
TBor (b, 0) = b
TBor (b, b) = b
TBor ($b1$, $b2$) =
- selon $b1$, $b2$ { $b1 \neq b2$ }
  - /Ax, x, A$\bar{x}$\ = $b1$ et /Bx, x, B$\bar{x}$\ = $b2$:
    - RepCanonique (x, TBor(Ax, Bx), TBor(A$\bar{x}$, B$\bar{x}$))
  - /Ax, x, A$\bar{x}$\ = $b1$ et /By, y, B$\bar{y}$\ = $b2$:
    - si $x \ll y$ alors RepCanonique (x, TBor(Ax, $b2$), TBor(A$\bar{x}$, $b2$))
    - sinon RepCanonique (y, TBor($b1$, By), TBor($b1$, B$\bar{y}$))

FIG. 2.16 – Construction d'un BDD à partir d'une expression algébrique booléenne

5. Exercices

E2.7 : Proposition logique
Un étudiant dit : je vais faire du ski s'il fait beau ou s'il ne fait pas beau et que je n'ai pas d'examen à réviser. Cet étudiant est-il sérieux ? sportif ? Pour répondre donner une forme plus simple de cette proposition logique.

E2.8 : Expression booléenne algébrique de la majorité
Trois personnes doivent voter bleu ou rouge. Démontrer en passant par les expressions booléennes algébriques correspondantes que si la majorité est pour le bleu alors, s'ils changent tous d'avis, la majorité sera pour le rouge.

E2.9 : De Morgan
Démontrer les formules de De Morgan à partir des tables de vérité des fonctions somme, produit et complément.

E2.10 : Règles de simplification booléenne
Démontrer les règles de simplification suivantes à partir des axiomes de l'algèbre de Boole.

$$a + a.b = a$$
$$a + \bar{a}.b = a + b$$
$$a.b + \bar{a}.b = b$$
$$a.b + \bar{a}.c + b.c = a.b + \bar{a}.c$$

E2.11 : Expression booléenne
Donner une expression booléenne de la fonction $f(a, b, c)$ qui vaut 1 si et seulement si la majorité de ses trois variables vaut 1.

E2.12 : Ou exclusif
Démontrer que l'opérateur ou-exclusif (noté $\oplus$) défini par $x_1 \oplus x_2 = \bar{x_1}.x_2 + x_1.\bar{x_2}$ est associatif.

E2.13 : Théorème de Shannon
Démontrer la première forme du théorème de Shannon.

E2.14 : Formes de Lagrange
Obtenir la deuxième forme de Lagrange de la fonction $f(x_1, x_2) = \bar{x_1}.x_2 + \bar{x_1}.\bar{x_2}$ à partir de la deuxième forme du théorème de Shannon.

E2.15 : Poids d'un vecteur booléen
On appelle poids d'un vecteur booléen le nombre de 1 de ce vecteur. le vecteur $(0, 1, 1, 0)$ a un poids de 2. Donner des expressions booléennes des fonctions simples p_2, p_1 et p_0 qui correspondent au codage en base 2 du poids d'un vecteur de 4 variables booléennes x_1, x_2, x_3, x_4.

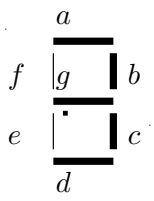


FIG. 2.17 – Représentation du chiffre 3 sur un afficheur 7 segments

E2.16 : Tableau à 4 variables
Donner une base irrédondante minimale de la fonction $f(a,b,c,d) = \bar{a}.\bar{b}.\bar{c} + a.\bar{b}.\bar{d} + \bar{a}.b.d + b.\bar{c}.\bar{d} + a.b.c + a.\bar{c}.d + \bar{a}.c.\bar{d}$.

E2.17 : Tableau à 5 variables
Donner une base irrédondante minimale de la fonction $f(a,b,c,d,e) = a.\bar{b} + \bar{b}.c.e + \bar{a}.\bar{b}.d.\bar{e} + \bar{a}.b.d.e + a.\bar{c}.d.\bar{e}$.

E2.18 : Afficheur 7 segments
Soit un afficheur à cristaux liquides comportant 7 segments, comme il y en a sur la plupart des calculettes.

On veut réaliser les 7 fonctions booléennes (a, b, c, d, e, f, g) à 4 variables (e_4, e_3, e_2, e_1) qui correspondent aux 7 segments (Cf. Figure 2.17). La fonction booléenne vaudra 1 si le segment doit être allumé pour la représentation du chiffre décimal donné en binaire par (e_4, e_3, e_2, e_1). Par exemple le chiffre 3 ($e_3e_2e_1e_0 = 0011$) en entrée donne $a = b = c = g = d = 1$.

Ces 7 fonctions sont phi-booléennes car on ne veut représenter que les chiffres décimaux (0...9). Sur 4 bits on a aussi les entrées de 10 à 15 qui correspondent donc à des points à Φ pour les 7 fonctions. Attention le 6 et le 9 pouvant avoir différentes représentations sur l'afficheur, nous choisirons ici de représenter le 6 avec le segment a allumé et le 9 avec le segment d allumé.

Donner les tableaux de Karnaugh de ces 7 fonctions. Donner une base irrédondante minmale générale de la fonction générale $F = (a,b,c,d,e,f,g)$.

Donner les ROBDDs de a, b et c avec différents ordres sur les variables.

Chapitre 3

Représentation des grandeurs

Dans le chapitre 1 nous avons vu que l'ordinateur ne traite pas véritablement l'*information* mais ses représentations. La représentation de l'information se fait à travers un code. Pour des raisons technologiques qui apparaissent dans le chapitre 7 la représentation de toute information est un vecteur de booléens, ou bits. Les bits sont identifiés individuellement, le plus souvent par un simple numéro. On parle de *représentation digitale* de l'information. Physiquement un booléen, ou bit, est l'état d'un fil électrique. L'ordinateur étant alimenté par un générateur continu, la tension basse (la masse) représente le 0 (ou Faux), la tension haute (l'alimentation) représente le 1 (ou Vrai). Il existe cependant de nombreuses applications où des appareils de mesure donnent des tensions électriques proportionnelles à la grandeur mesurée. On parle dans ce cas de représentation *analogique* de l'information. Pour être traitées par un ordinateur standard (on dit *numérique*, ou *digital*, par opposition à analogique) ces tensions sont converties par des circuits spécifiques (Convertisseurs Analogiques Numériques, ou, a contrario, Numériques Analogiques).

Dans ce chapitre, nous donnons dans le paragraphe 1. les éléments de ce que signifie un codage par des booléens. Nous étudions ensuite les représentations des nombres, et, ce qui va avec, les techniques de calcul sur ces représentations. Nous distinguons la représentation des naturels (au paragraphe 2.), et celle des entiers relatifs (au paragraphe 3.). La représentation de nombres réels est brièvement évoquée dans le paragraphe 6., celle des caractères dans le paragraphe 5. La notion de taille de la représentation est présente en permanence.

1. Notion de codage d'informations

1.1 Codage binaire

A un instant donné N fils électriques sont chacun à 0 ou à 1. Il est nécessaire d'identifier chacun des fils par un numéro, par exemple entre 0 et $N-1$. L'en-

semble des N fils peut se trouver dans une des 2^N configurations possibles. Les N fils peuvent représenter 2^N informations différentes. On parle aussi des 2^N valeurs possibles d'une information. Il y a là une différence entre le vocabulaire courant et un vocabulaire technique.

Pour évaluer le nombre de valeurs différentes représentables sur N bits, il est commode d'avoir en tête les valeurs des petites puissances de 2 et les ordres de grandeurs des grandes : $2^0 = 1$; $2^1 = 2$. Les puissances suivantes sont 4, 8, 16, 32, 64, 128, $2^8 = 256$ et $2^{10} = 1024$. Comme 1000 est proche de 1024, il est facile de compléter la suite : $2^{10} \approx 10^3$, $2^{20} \approx 10^6$, $2^{30} \approx 10^9$, $2^{40} \approx 10^{12}$.

Les préfixes d'unités correspondants sont `kilo`, `méga`, `giga`, `téra`. Un kilobit correspond donc à 1024 bits et non à 1000 bits.

Repérer un élément parmi un ensemble de 256 éléments suppose de le localiser par un numéro codé sur 8 bits. Dans certains contextes ce numéro est appelé une *adresse*. Repérer un élément parmi un ensemble de 4 giga-éléments suppose de le localiser par un numéro codé sur 32 bits.

La notation de logarithme à base 2 est parfois utilisée : si $2^N = M, \log_2 M = N$; ainsi pour représenter P valeurs différentes il faut au moins R bits, où R est l'entier immédiatement supérieur au logarithme à base 2 de P. Ainsi $\log_2 2048 = 11$ et pour représenter 2050 valeurs différentes il faut 12 bits.

La correspondance entre la représentation par un vecteur de booléens et la valeur se fait par une convention, un *code*. L'ensemble des valeurs codables est caractéristique du domaine (nombres, couleurs...) Par exemple, si une gamme de température va de - 10 à + 40 degrés, et si la température est codée sur 9 bits, la précision peut être de l'ordre du dizième de degré ($2^9 = 512$ codes possibles pour 50 degrés). Si la température est codée sur 12 bits la précision est de l'ordre du centième ($2^{12} = 4096$ codes possibles pour 50 degrés).

1.2 Un exemple : le codage des couleurs

On trouve dans la documentation du micro-ordinateur Commodore 64 le tableau de la figure 3.1 indiquant le code sur 4 bits $b_3b_2b_1b_0$ des 16 couleurs affichables par cette machine. On trouve dans la documentation de micro-ordinateurs PC (carte CGA) le tableau de la figure 3.2, donnant un autre codage.

La question *Comment est représenté* rouge *?*, ou *Que représente 0 0 1 0 ?* n'a de sens que si le code est précisé. De même, la conversion d'un code à l'autre n'a un sens que pour les couleurs qui sont représentées dans les deux codes (brun, bleu pâle, ...).

Les deux couleurs noir et cyan ont le même codage dans les deux codes, ce qui est fortuit. Dans le deuxième code, chaque bit a une interprétation. Le bit 3 correspond à la présence d'une composante pâle, le bit 2 à la présence d'une composante rouge, le bit 1 au vert et le bit 0 au bleu. On trouve souvent le sigle RGB (Red, Green, Blue) dans ce contexte. Une telle interprétation

$b_3b_2b_1b_0$		$b_3b_2b_1b_0$		$b_3b_2b_1b_0$	
0 0 0 0	noir	0 1 0 1	vert	1 0 1 0	rose
0 0 0 1	blanc	0 1 1 0	bleu	1 0 1 1	gris foncé
0 0 1 0	rouge	0 1 1 1	jaune	1 1 0 0	gris moyen
0 0 1 1	cyan	1 0 0 0	orange	1 1 0 1	vert pâle
0 1 0 0	violet	1 0 0 1	brun	1 1 1 0	bleu pâle
				1 1 1 1	gris pâle

FIG. 3.1 – Codage des couleurs du Commodore 64

$b_3b_2b_1b_0$		$b_3b_2b_1b_0$		$b_3b_2b_1b_0$	
0 0 0 0	noir	0 1 0 1	violet	1 0 1 0	vert pâle
0 0 0 1	bleu	0 1 1 0	brun	1 0 1 1	cobalt
0 0 1 0	vert	0 1 1 1	gris	1 1 0 0	rose
0 0 1 1	cyan	1 0 0 0	noir pâle	1 1 0 1	mauve
0 1 0 0	rouge	1 0 0 1	bleu pâle	1 1 1 0	jaune
				1 1 1 1	blanc

FIG. 3.2 – Codage des couleurs pour PC, carte CGA

individuelle de chaque bit n'existe pas dans le premier code.

1.3 Codage 1 parmi n

Un codage particulier est utilisé dans certaines applications matérielles ou logicielles : le codage appelé *1 parmi n*. Pour ce code, on utilise autant de bits que d'informations à coder. Pour reprendre l'exemple des couleurs, chacune serait codée sur b_{15},.. b_0, et chaque bit correspondrait alors à une couleur.

2. Les naturels

2.1 Représentation des entiers naturels

2.1.1 Numération de position

Les entiers naturels peuvent être écrits de différentes façons (voir par exemple [Ifr94]). Le système des Romains est encore présent dans certaines notations, les dates des livres par exemple.

La meilleure représentation est la numération de position dans une base choisie. En base 10, ou système décimal, on choisit 10 symboles différents, les 10 chiffres décimaux $0, 1, \ldots, 9$. Ils représentent les valeurs des 10 premiers naturels. Les naturels suivants s'écrivent avec plusieurs chiffres : un chiffre des unités, un chiffre des dizaines, des centaines, des milliers, etc.

Si un naturel X s'écrit en base β sur N chiffres $x_{N-1}\ x_{N-2}...\ x_1\ x_0$, la correspondance entre la valeur du naturel X et celles des chiffres est donnée

par l'équation :

$$X = \sum_{i=0}^{N-1} \beta^i \times \text{valeur}(x_i) \quad \text{ou, pour simplifier : } X = \sum_{i=0}^{N-1} \beta^i x_i$$

La correspondance est telle que l'écriture d'un naturel dans une base donnée est unique. Dans la suite nous ne précisons plus que c'est toujours la valeur du chiffre (et non le chiffre lui-même) qui intervient dans les expressions arithmétiques. En base β, sur N chiffres, tous les naturels compris au sens large entre 0 et $\beta^N - 1$ sont représentables. Les nombres plus grands peuvent être représentés par leur reste modulo β^N. C'est ce qui se produit sur les compteurs kilométriques des voitures : si le compteur a 5 chiffres, quand on a parcouru 100 012 kilomètres, le compteur marque 00 012. Une situation analogue a lieu pour les angles où on ne donne que le représentant dans l'intervalle $[0, 2.\pi[$. En informatique on rencontre le terme de chiffre de poids faible, pour le chiffre des unités et, si un naturel est écrit sur 7 chiffres, celui de chiffre de poids le plus fort pour le chiffre des millions. L'usage dans la vie courante est de ne pas écrire les 0 en poids forts. A certains endroits, pour des raisons matérielles, c'est une obligation. Le compteur kilométrique des voitures par exemple. En informatique, on les écrit très souvent. Les machines ont un format pour représenter les nombres, c'est-à-dire un nombre de chiffres pré-établi. Quand ce nombre de chiffres est mal choisi, comme par exemple représenter une date avec seulement deux chiffres décimaux pour l'année, les conséquences sont ennuyeuses. Dans les documents écrits où il y a risque d'ambiguïté, on écrit la base en indice. La base elle-même est alors écrite en base décimale (ou base 10). Par exemple, le nombre qui s'écrit 147 en base 10 s'écrit 1042 en base 5 :

$147_{10} = 10010011_2 = 173_9 = 93_{16} = 1042_5 = 12110_3$

Il existe une autre représentation conventionnelle : le décimal codé en binaire (DCB) dans laquelle chaque chiffre décimal est écrit en binaire sur 4 bits. On a alors $147_{10} = 0001\ 0100\ 0111_{\text{dcb}}$.

Technique de conversion Pour montrer la technique de conversion d'une base à une autre, prenons deux exemples.

– Passage de la base 10 à la base 5.

$$\begin{aligned}
147_{10} &= 29_{10} && && && \times 5 && +\boxed{2} \\
&= (5 && && \times 5 && +\boxed{4}) \times 5 && +2 \\
&= ((1 && \times 5) && +\boxed{0}) \times 5 && +4) \times 5 && +2 \\
&= (((0 \times 5) + \boxed{1}) && \times 5 && +0) \times 5 && +4) \times 5 && +2
\end{aligned}$$

Les restes successifs dans la division par 5 sont 2, 4, 0 et 1. Le chiffre des unités est 2, celui de rang supérieur est 4, etc. $147_{10} = 1042_5$, c'est-à-dire : $147_{10} = 1 \times 5^3 + 0 \times 5^2 + 4 \times 5^1 + 2 \times 5^0$.

– Passage de la base 2 à la base 10.
$10010011_2 = 1 \times 2^7 + 1 \times 2^4 + 1 \times 2^1 + 1 \times 2^0$
$10010011_2 = 1 \times 128_{10} + 1 \times 16_{10} + 1 \times 2 + 1 = 147_{10}$

2.1.2 Représentations binaire, hexadécimale

Ce principe de la numération de position peut s'appliquer avec une base entière quelconque supérieure ou égale à 2. Si la base est 2 on parle de numération *binaire*, pour la base 8 de numération *octale*, pour la base 16 d'*hexadécimal*. En base 2, sur N chiffres, tous les naturels compris au sens large entre 0 et $2^N - 1$ sont représentables. On pourrait parler de chiffre des *deuzaines*, des *quatraines*, ou des *seizaines* mais ces mots ne sont pas dans le lexique courant. L'intérêt de la base 2 est que les deux chiffres 0 et 1 peuvent facilement être représentés par les deux booléens 0 et 1, ou par les états bas et haut d'un fil électrique. Ce codage des nombres est le seul utilisé dans les ordinateurs. Nous emploierons les termes de *chiffres binaires* ou de *bits* indistinctement.

Les 16 chiffres hexadécimaux sont $0, 1, \ldots, 8, 9, A, B, C, D, E, F$ représentant les valeurs des naturels de 0 à 15. On a évidemment valeur$(A) = 10, \ldots,$ valeur$(F) = 15$.

On peut aisément convertir un nombre écrit en base 16 vers la base 2 et réciproquement. Il suffit pour cela de convertir par tranches de 4 chiffres binaires, ou d'un chiffre hexadécimal. Ainsi dans l'exemple suivant figurent les deux écritures l'une en dessous de l'autre :

3	4	7	B	8
0011	0100	0111	1011	1000

En effet le chiffre hexadécimal B représente le naturel 11_{10}, qui en binaire s'écrit 1011, et $347B8_{16} = 00110100011110111000_2$.

On remarque là une analogie avec le passage de l'écriture décimale à l'écriture en langue naturelle. Ici figurent l'écriture d'un nombre en base décimale et son écriture en français (les espaces ne sont pas significatifs) :

104	730	105
cent quatre	millions sept cent trente	mille cent cinq

2.2 Opérations sur les vecteurs booléens représentant les naturels

Etant donnés deux naturels A et B, représentés respectivement par M et N bits, on cherche à trouver un procédé de calcul, ou de fabrication, des bits représentant $A + B, A - B, A \times B, \ldots$ (une approche très complète se trouve dans [Mul89]).

Nous devons nous préoccuper du nombre de bits nécessaires pour représenter le résultat du calcul. Pour la somme c'est $\max(M, N) + 1$, pour le produit $M + N$. Pour simplifier ce problème, nous supposerons, sauf mention contraire, que les deux nombres sont codés sur N bits, et que N est une puissance de 2. Si ce n'est pas le cas, il est toujours possible de compléter A ou B en poids forts. Si A et B sont codés sur 2^p bits, $A + B$ est codé sur $2^p + 1$ bits et $A \times B$ est codé sur 2^{p+1} bits. La somme de deux nombres codés sur N chiffres est représentable sur $N + 1$ chiffres. Le chiffre de poids fort de cette somme est égal à 0 ou 1. Ceci est valable dans toutes les bases.

2.2.1 Extension et réduction de format

Si un naturel est codé sur N bits et qu'il faille le coder sur M bits, avec $M > N$, il suffit d'ajouter des 0 en poids forts. A l'inverse, si la représentation de C a k bits à 0 en poids forts, C peut être représenté sur k bits de moins.

Par exemple $00001100_2 = 1100_2$.

2.2.2 Addition

A et B étant représentés sur N bits, $a_{N-1}, \ldots, a_0$, et $b_{N-1}, \ldots, b_0$, la somme S de A et B s'écrit sur $N + 1$ bits $s_N, \ldots, s_0$

Deux questions se posent : comment obtenir les s_i à partir des a_i et des b_i, et peut-on représenter S sur N bits ?

Pour obtenir les chiffres de la somme, examinons brièvement le procédé pour la base 10 que l'on apprend à l'école : les a_i et b_i sont compris entre 0 et 9. La base est 10. On applique un procédé itératif, en commençant par les poids faibles, et en propageant une retenue d'étage (ou tranche) de 1 chiffre en étage de 1 chiffre vers les poids forts.

A chaque étage i le calcul du *report* [1] sortant rep_{si} de l'étage est fonction des chiffres a_i et b_i de A et B à cet étage et du report entrant dans cet étage rep_{ei}. Le report entrant dans l'étage 0 est évidemment nul. Le report sortant de l'étage j, rep_{sj} est le report entrant de l'étage j+1 $rep_{e(j+1)}$.

$$rep_{si} = 1 \text{ si } a_i + b_i + rep_{ei} \geq 10 \text{ et}$$
$$rep_{si} = 0 \text{ si } a_i + b_i + rep_{ei} < 10$$

le chiffre s_i de la somme à l'étage i est la somme modulo 10 de a_i, b_i et rep_{ei}, c'est-à-dire :

$$s_i = a_i + b_i + rep_{ei} - 10 \text{ si } rep_{si} = 1 \text{ et}$$
$$s_i = a_i + b_i + rep_{ei} \text{ si } rep_{si} = 0$$

[1] Nous emploierons le terme de *report* pour l'addition et, plus loin, celui d'*emprunt* pour la soustraction. La langue courante utilise le terme de *retenue* dans les deux cas.

a_i	b_i	rep_{ei}	$rep_{si} = maj(a_i, b_i, rep_{ei})$	$s_i = \oplus(a_i, b_i, rep_{ei})$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

FIG. 3.4 – Table d'addition

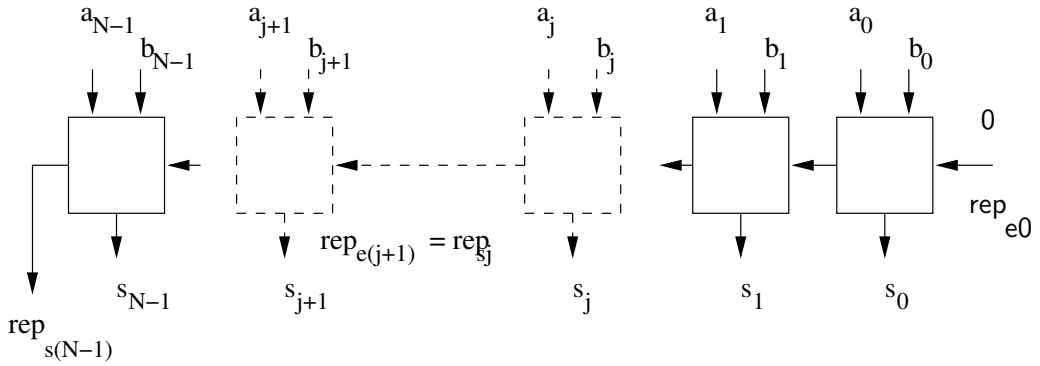


FIG. 3.3 – Schéma de principe d'un additionneur

En binaire le procédé est identique mais les a_i et b_i valent 0 ou 1. Les deux fonctions de calcul du report et du chiffre de somme sont définies pour des triplets de nombres 0 ou 1.

La fonction report sortant d'un *étage* d'addition binaire est la fonction *majorité*. On la note maj. maj(x, y, z) vaut 1 si et seulement si $x + y + z \geq 2$.

La fonction calculant le chiffre de somme est la *somme modulo 2* de 3 entiers. On la note $\oplus$.

$$\oplus(x, y, z) = (x + y + z) \text{ si maj } (x, y, z) = 0$$
$$\oplus(x, y, z) = (x + y + z - 2) \text{ si maj } (x, y, z) = 1.$$

Les *tables d'addition* pour un étage sont données par la figure 3.4.

Pour l'addition de deux nombres à N bits, les formules sont données par les équations suivantes qui utilisent les reports intermédiaires r_i où r_{i+1} est la retenue sortante de l'étage numéro i :

$$s_0 = \oplus(a_0, b_0, 0) \qquad r_1 = \text{maj } (a_0, b_0, 0)$$
$$s_i = \oplus(a_i, b_i, r_i) \qquad r_{i+1} = maj(a_i, b_i, r_i) \quad \forall i.\ 1 \leq i \leq N - 1$$

De plus, puisque la somme est sur $N + 1$ bits, $s_N = r_N$.

Dans ce cas s_N s'appelle souvent *La* retenue. Si on cherche à représenter la somme S sur N bits, ce n'est possible que si s_N vaut 0. Si s_N vaut 1, la somme est trop grande et ne peut être représentée sur N bits.

> **Remarque :** Dans les processeurs, après une addition, ce bit de retenue est disponible dans le mot d'état sous le nom de bit *indicateur* C (en anglais report se dit *Carry*). Il vaut 1 si la somme de deux naturels codés sur N bits n'est pas représentable sur N bits. On utilisera ce bit dans la programmation en langage machine au chapitre 12.

2.2.3 Multiplication et division entière par une puissance de 2

On sait multiplier par 10 un nombre écrit en base 10 en ajoutant un 0 en poids faible de sa représentation. Il en est de même en base 2. Et si l'on ajoute deux 0 en poids faible de l'écriture binaire, on obtient l'écriture binaire du nombre multiplié par 4. Le produit d'un nombre sur N bits par 2^k s'écrit sur $N + k$ bits, les k bits de poids faibles étant à 0.

On sait obtenir la représentation décimale du quotient entier par 10 d'un naturel en ôtant le chiffre des unités de sa représentation décimale. De même pour la base 2, si l'on supprime 3 chiffres en poids faible, on obtient l'écriture du quotient entier par 8. Le quotient entier d'un nombre sur N bits par 2^k s'écrit sur $N - k$ bits.

On sait obtenir la représentation décimale du reste modulo 10 en ne gardant que le chiffre des unités de la représentation décimale. De même pour la base 2, si l'on garde les 4 chiffres en poids faible, on obtient l'écriture du reste modulo 16. Le reste modulo 2^k d'un nombre s'écrit sur k bits.

Le tableau suivant illustre différentes multiplications et divisions entières. Tous les nombres y sont écrits en binaire sur 6 bits. Il y a donc parfois des zéros en poids forts.

Ecriture de N en décimal	Ecriture de N en binaire	Ecriture de $N \times 2$ en binaire	Ecriture de $N/4$ en binaire	Ecriture de N mod 8 en binaire
5	000101	001010	000001	000101
13	001101	011010	000011	000101
29	011101	111010	000111	000101
28	011100	111000	000111	000100
35	100011	impossible	001000	000011

2.2.4 Multiplication générale

Si deux naturels A et B sont codés sur N bits, leur produit est codé sur $2 \times N$ bits. Si N vaut 1, le produit de A et B est facile à calculer. Sinon, comme dans l'exercice E3.14 on décompose A et B en parties faible et forte.

Le produit P est la somme des 4 produits partiels :

$$P1 = A_{\text{fort}} \times B_{\text{fort}} \times 2^{N/2} \times 2^{N/2}$$
$$P2 = A_{\text{fort}} \times B_{\text{faible}} \times 2^{N/2}$$
$$P3 = A_{\text{faible}} \times B_{\text{fort}} \times 2^{N/2}$$
$$P4 = A_{\text{faible}} \times B_{\text{faible}}$$

Remarquons qu'effectuer l'addition de $P1$ et $P4$ est très facile. L'un des deux nombres n'a que des 0 là où l'autre a des chiffres significatifs.

Une autre expression du produit reprend simplement l'écriture binaire : puisque $14_{10} = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$, on a aussi $14_{10} \times 13 = 1 \times 2^3 \times 13 + 1 \times 2^2 \times 13 + 0 \times 2^1 \times 13 + 1 \times 2^0 \times 13$. On a vu que le calcul du produit de 13 par une puissance de 2 revient à écrire 1101 suivi du bon nombre de 0. La multiplication se réduit alors à une suite d'additions et de décalages.

2.2.5 Soustraction

La différence $D = A - B$ de deux naturels A et B n'est définie que si $A > B$.

> **Remarque :** Un problème est celui de la détection des cas valides et invalides. En informatique comparer deux nombres nécessite de calculer leur différence. On ne peut donc pas raisonnablement comparer deux nombres *avant* de calculer leur différence si elle existe. Dans un système informatique, on calcule toujours ce que l'on croit être la différence, puis on se préoccupe de savoir si la différence est représentable ou non. Si oui, elle est le résultat obtenu.

Comme pour l'addition, l'opération se fait tranche par tranche, en commençant par les poids faibles et avec propagation d'un bit d'emprunt vers les poids forts. Le calcul fait apparaître le bit d'emprunt à l'étage de poids plus fort. On peut produire une table de soustraction en base 2, analogue à la table d'addition du paragraphe 2.2.2, tenant compte du bit d'emprunt *entrant* e_e et faisant apparaître le bit d'emprunt *sortant* e_s et le bit de résultat d_i (Cf. Figure 3.5-(a)).

On a le résultat suivant : Si A et B sont codés sur N bits la différence est un naturel (c'est-à-dire $A \geq B$) si et seulement si l'emprunt sortant de l'étage de rang $N - 1$ est nul.

Reprenons la table d'addition du paragraphe 2.2.2, en remplaçant systématiquement les bits de reports entrants et sortants par leur complémentaire booléen. De même remplaçons le bit de l'opérande B par son complémentaire. On retrouve la table de la soustraction (Cf. Figure 3.5-(b)).

Si l'on applique l'algorithme d'addition avec A et le complémentaire de B et si l'on prend soin de complémenter en entrée et en sortie tous les bits de report, on obtient l'algorithme de soustraction de $A - B$.

(a)

a_i	b_i	e_e	e_s	d_i
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

(b)

a_i	$\overline{b_i}$	$\overline{r_e}$	$\overline{r_s}$	s_i
0	1	1	0	0
0	1	0	1	1
0	0	1	1	1
0	0	0	1	0
1	1	1	0	1
1	1	0	0	0
1	0	1	0	0
1	0	0	1	1
a_i	b_i	e_e	e_s	d_i

FIG. 3.5 – (a) Table de soustraction et (b) table d'addition modifiée

On peut aussi utiliser les expressions pour chaque étage :

$e_s = 1$ si $a_i < b_i + e_e$ c'est-à-dire si $a_i - (b_i + e_e) < 0$
$e_s = 0$ si $a_i \geq b_i + e_e$
$d_i = a_i - (b_i + e_e)$ si $e_s = 0$
$d_i = 2 + a_i - (b_i + e_e)$ si $e_s = 1$

Remarque : Dans les processeurs, après une soustraction, le complémentaire du bit d'emprunt sortant est disponible dans le mot d'état sous le nom de bit *indicateur* C. Il vaut 0 si la différence de deux naturels codés sur N bits est positive, donc représentable sur N bits.

3. Les relatifs

3.1 Représentation des entiers relatifs

Pour représenter des entiers relatifs par un vecteur de N booléens, la première idée qui vient à l'esprit est de représenter la valeur absolue sur $N-1$ bits et de réserver un bit pour le signe. Cette idée simple est correcte. On parle de *représentation en signe et valeur absolue*. Une autre représentation est habituellement utilisée. Elle présente des similitudes avec la représentation d'un angle quelconque par un nombre réel dans l'intervalle $[-\pi, +\pi[$. Nous allons la décrire d'abord de façon très intuitive. Pour cela nous considérons des nombres sur 4 chiffres. Le même raisonnement vaut pour toute autre taille. Quand on achète une voiture neuve, le compteur kilométrique indique 0000. Il indique ensuite 0001, puis 0002. Les voitures à compteur binaire feraient apparaître 0001 puis 0010. Imaginons que le compteur décompte lorsque l'on roule en marche arrière. Avec une voiture neuve, il afficherait successivement 9999, puis 9998. Un compteur binaire montrerait 1111, puis 1110.

On *décide* de représenter -1 par 1111, puis -2 par 1110, comme sur le compteur kilométrique. Reste à fixer une convention. En effet si l'on roule quinze

kilomètres en marche arrière le compteur affiche aussi 0001, et l'on risque de croire que l'on a parcouru 1 kilomètre en marche avant ! La convention habituelle est la suivante : les relatifs strictement positifs vont de 0001 à 0111 (soit de 1 à 7) ; les relatifs strictement négatifs vont de 1111 à 1000 (soit de -1 à -8) ; 0 reste codé 0000.

La convention est choisie pour que le bit de poids fort de la représentation soit un bit de signe. Il vaut 1 pour les nombres strictement négatifs. Sur N bits les nombres représentables sont ceux de l'intervalle $[-2^{N-1},\ 2^{N-1} - 1]$. Ce système s'appelle codage en *complément à 2* (parfois complément à 2^N puisqu'il y a N bits). Au passage remarquons que l'intervalle des nombres représentables n'est pas symétrique par rapport à 0. C'est obligatoire. Avec une base paire, on représente un nombre pair de nombres. Il ne peut y en avoir autant de strictement positifs que de strictement négatifs, sauf si 0 a deux représentations.

De façon moins intuitive, si un relatif Y s'écrit en complément à 2 sur N chiffres binaires : $y_{N-1},\ y_{N-2}, \ldots, y_1,\ y_0$, la correspondance entre la valeur du relatif Y et celles des chiffres est donnée par l'équation :

$$Y = (-2^{N-1} \times y_{N-1}) + \sum_{i=0}^{N-2} 2^i \times y_i$$

ou, ce qui est équivalent, par :

$$Y = (-2^{N} \times y_{N-1}) + \sum_{i=0}^{N-1} 2^i \times y_i$$

La correspondance est telle que l'écriture est unique comme pour le cas des naturels dans une base donnée.

Remarquons que si l'on considère les deux vecteurs binaires représentant un relatif et son opposé, et si l'on interprète ces deux vecteurs comme les représentations en binaire de deux naturels, la somme de ces deux naturels est 2^N. C'est l'origine du nom *complément à* 2^N. Ainsi, sur 4 bits, 0101 code 5. En complément à 2, sur 4 bits -5 est représenté par 1011. En base 2, 1011 représente le naturel 11, et $11 + 5 = 16$.

Conversion Une difficulté notable vient d'apparaître, la même que pour les couleurs en début de chapitre. La question *Que représente 100011 ?* ou *Comment est représenté l'entier K ?* n'a pas de sens. Il faut à chaque fois préciser dans quelle convention, binaire pur ou complément à 2. Comme pour les couleurs, on peut avoir besoin de convertir d'une convention à l'autre les nombres qui sont représentables dans les deux (comme le brun et le bleu pâle, pour les couleurs). Pour les nombres sur N bits ce sont les nombres de l'intervalle $[0, 2^{N-1} - 1]$. Ils ont la même représentation dans les deux codes (comme le cyan et le noir qui ont le même code dans l'exemple des couleurs).

3.2 Opérations sur les vecteurs booléens représentant les relatifs

3.2.1 Extension et réduction de format

Si un relatif Y est codé sur N bits, il suffit de reporter le bit de signe de Y en poids forts $M - N$ fois pour obtenir son codage M bits (avec $M > N$).

Si les k bits de plus forts poids de la représentation de C sont identiques, C peut être représenté sur $k - 1$ bits de moins. On ne perd pas le *bit de signe*. Par exemple : $11111010_{c2} = 1010_{c2}$.

3.2.2 Addition

Soient A et B représentés en complément à 2 par $a_{N-1},\ a_{N-2}, \ldots, a_1,\ a_0$ et $b_{N-1},\ b_{N-2}, \ldots, b_1,\ b_0$. On a :

$$A = (-2^{N-1}) \times a_{N-1} + \sum_{i=0}^{N-2} 2^i \times a_i, \qquad B = (-2^{N-1}) \times b_{N-1} + \sum_{i=0}^{N-2} 2^i \times b_i$$

Comme pour les naturels, déterminons si la somme peut être représentable sur N bits et comment les bits de la somme peuvent être exprimés. On pose :

$$\alpha = \sum_{i=0}^{N-2} 2^i \times a_i, \quad \beta = \sum_{i=0}^{N-2} 2^i \times b_i, \quad \gamma = (\alpha + \beta) \text{ modulo } 2^{N-1}$$

avec :

$$\alpha \in [0, 2^{N-1} - 1], \quad \beta \in [0, 2^{N-1} - 1], \quad \gamma \in [0, 2^{N-1} - 1].$$

On a évidemment :

$$A = -2^{N-1} \times a_{N-1} + \alpha$$

et, de même,

$$B = -2^{N-1} \times b_{N-1} + \beta.$$

Soit r_e défini par :

$$\alpha + \beta = r_e \times 2^{N-1} + \gamma.$$

r_e vaut donc 1 ou 0. C'est le report sortant du calcul de $\alpha + \beta$. γ est la somme $\alpha + \beta$ privée de son bit de poids fort r_e.
Soit S la somme de A et de B.

$$S = -2^{N-1} \times (a_{N-1} + b_{N-1}) + (\alpha + \beta) \quad = \quad -2^{N-1} \times (a_{N-1} + b_{N-1} - r_e) + \gamma$$

Les valeurs possibles de $a_{N-1} + b_{N-1} - r_e$ sont -1, 0, 1 ou 2 puisque les trois nombres a_{N-1}, b_{N-1}, r_e sont des chiffres binaires.

Nombre de bits nécessaires pour représenter S La première question est : S est-il représentable sur N bits en complément à 2 ? C'est-à-dire a-t-on $-2^{N-1} \leq S \leq 2^{N-1} - 1$? Examinons les deux cas où la réponse est non.

Premier cas : $S < -2^{N-1}$

$$\begin{array}{rl} -2^{N-1} \times (a_{N-1} + b_{N-1} - r_e) + \gamma & < -2^{N-1} \\ -2^{N-1} \times (a_{N-1} + b_{N-1} - r_e - 1) & < -\gamma \\ 2^{N-1} \times (a_{N-1} + b_{N-1} - r_e - 1) & > \gamma \end{array}$$

Puisque $\gamma \in [0, 2^{N-1} - 1]$, cette inégalité ne peut être vérifiée avec certitude que si

$$a_{N-1} + b_{N-1} - r_e - 1 \geq 1 \text{ c'est-à-dire si } a_{N-1} + b_{N-1} - r_e - 1 = 1.$$

Ce qui ne se produit que si $a_{N-1} = b_{N-1} = 1$ et $r_e = 0$. Si l'on pose $r_s = maj(a_{N-1}, b_{N-1}, r_e)$, on a dans ce cas $r_s = 1 = \overline{r_e}$.

Deuxième cas : $S > 2^{N-1} - 1$

$$\begin{array}{rl} -2^{N-1} \times (a_{N-1} + b_{N-1} - r_e) + \gamma & > 2^{N-1} - 1 \\ -2^{N-1} \times (a_{N-1} + b_{N-1} - r_e) + \gamma & \geq 2^{N-1} \\ -2^{N-1} \times (a_{N-1} + b_{N-1} - r_e + 1) & \geq -\gamma \\ 2^{N-1} \times (a_{N-1} + b_{N-1} - r_e + 1) & \leq \gamma \end{array}$$

Cette inégalité ne peut être vérifiée avec certitude que si

$$a_{N-1} + b_{N-1} - r_e + 1 \leq 0 \text{ c'est-à-dire si } a_{N-1} + b_{N-1} - r_e + 1 = 0$$

Ce qui ne se produit que si $a_{N-1} = b_{N-1} = 0$ et $r_e = 1$.
Dans ce cas $r_s =$ maj $(a_{N-1}, b_{N-1}, r_e) = 0 = \overline{r_e}$.

Dans tous les autres cas $-2^{N-1} \leq S \leq 2^{N-1} - 1$, c'est-à-dire pour

$$\begin{array}{ll} a_{N-1} = b_{N-1} = 0, & r_e = 0 \\ a_{N-1} = b_{N-1} = 1, & r_e = 1 \\ a_{N-1} = 1,\ b_{N-1} = 0, & r_e \text{ quelconque} \\ a_{N-1} = 0,\ b_{N-1} = 1, & r_e \text{ quelconque} \end{array}$$

la somme S de A et B est représentable sur N bits en complément à 2. On a alors $r_s = r_e$. Le tableau suivant récapitule les différents cas.

a_{N-1}	b_{N-1}	r_e	r_s	Interprétation	$a_{N-1} + b_{N-1} - r_e$
1	1	0	1	Premier cas : $S < -2^{N-1}$	2
0	0	1	0	Deuxième cas : $S > 2^{N-1} - 1$	-1
0	0	0	0	Somme représentable	0
1	1	1	1	Somme représentable	1
1	0	x	x	Somme représentable	$\overline{x}$
0	1	x	x	Somme représentable	$\overline{x}$

Notons $s = \oplus(a_{N-1}, b_{N-1}, r_e)$. Deux expressions booléennes décrivent la valeur du bit de débordement V après une addition :

$$V = a_{N-1}.b_{N-1}.\overline{s} + \overline{a_{N-1}}.\overline{b_{N-1}}.s$$
$$V = r_s \text{ ou exclusif } r_e$$

La première apparaît souvent dans les documents des constructeurs de processeurs. L'exercice E3.9 propose de montrer l'équivalence des deux expressions.

L'interprétation est facile : a_{N-1} étant interprété comme le bit de signe d'un opérande, b_{N-1} comme l'autre et s comme le bit de signe du résultat calculé par le processeur, le cas $a_{N-1} = 1, b_{N-1} = 1, s = 0$ signifierait que la somme de deux négatifs est positive. Cela se produit si $r_e = 0$.

Calcul des bits de S On se pose une deuxième question : comment calculer la représentation en complément à 2 de S, si elle existe, c'est-à-dire comment trouver le vecteur $s_{N-1}, s_{N-2}, \ldots, s_1, s_0$ tel que

$$S = -2^{N-1} \times s_{N-1} + \sum_{i=0}^{i=N-2} 2^i \times s_i$$

On sait que

$$S = \quad -2^{N-1} \times (a_{N-1} + b_{N-1} - r_e) + \gamma, \quad \text{avec } \gamma \in [0, 2^{N-1} - 1]$$

En identifiant bit à bit les deux écritures, on voit que pour $i \in [0, N-2]$, les s_i ne sont rien d'autres que les chiffres binaires de γ.

De plus, puisque $a_{N-1} + b_{N-1} - r_e$ vaut 0 ou 1, car S est représentable sur N bits, alors $-(a_{N-1} + b_{N-1} - r_e) = \oplus(a_{N-1}, b_{N-1}, r_e)$.

On a reconnu dans r_e et r_s les reports entrant et sortant du dernier étage d'addition binaire *normale* des vecteurs a_i et b_i. Ce qui signifie que les chiffres binaires de l'écriture de S s'obtiennent de la même façon que les chiffres binaires de la somme des deux naturels représentés en binaire pur par les a_i et les b_i. C'est là tout l'intérêt du codage en complément à 2.

> **Remarque :** Cette propriété est d'une portée pratique considérable.
> Elle signifie que le même mécanisme d'addition peut ajouter deux vecteurs binaires sans avoir à tenir compte de l'interprétation, binaire pur ou complément à 2, qui est faite des opérandes et du résultat. Les chiffres binaires du résultat, si celui-ci est représentable, sont identiques quelle que soit l'interprétation.
>
> On retrouvera cette propriété dans le chapitre 12 où l'on verra que la même instruction du langage machine convient pour l'addition, indépendamment du code choisi, et dans le chapitre 8 où l'on verra que le même circuit combinatoire additionneur convient pour l'addition indépendamment du code choisi.

Mais, attention, l'information disant si le résultat est représentable ou non n'est pas la même. En binaire pur le résultat de l'addition est représentable si et seulement si $r_s = 0$. En complément à 2 le résultat de l'addition est représentable si et seulement si $r_s = r_e$.

L'exercice corrigé E3.6 donne des exemples qui concrétisent ces équations.

Ecriture de l'opposé Soit A un relatif représenté sur N bits en complément à 2 par $a_{N-1}\ a_{N-2}, \ldots, a_1\ a_0$. On a :

$$A \ = \ (-2^N \times a_{N-1}) + \sum_{i=0}^{N-1} 2^i \times a_i$$

Complémentons chaque bit de A (en remplaçant a_i par $1 - a_i$), le résultat est un nombre A' défini par :

$$A' \ = \ -2^N \times (1 \ - \ a_{N-1}) + \sum_{i=0}^{N-1} 2^i \times (1 \ - \ a_i)$$

Si l'on ajoute A et A' modulo 2^N on obtient -1. $A + A' = -1$, c'est-à-dire $A = -A' - 1$, c'est-à-dire $-A = A' + 1$ (toutes ces égalités sont modulo 2^N).

Cela donne le procédé technique pour obtenir la représentation de l'opposé de A : on forme le complémentaire bit à bit A' de A et on lui ajoute 1. Comme l'opération est modulo 2^N, on ne tient pas compte d'éventuels reports. Un autre procédé consiste à recopier tous les bits en commençant par les poids faibles jusqu'au premier 1 inclus puis à inverser les suivants.

Attention toutefois car, sur N bits, l'opposé de -2^{N-1} n'est pas représentable.

3.2.3 Soustraction

Puisque l'addition est connue, ainsi que le passage à l'opposé, la soustraction ne pose pas de problèmes : il suffit de se souvenir que $A - B = A + -(B)$.

Comme pour l'addition, les constructeurs de processeurs donnent l'expression booléenne du bit V de débordement après une soustraction :

$$V = a_{N-1}.\overline{b_{N-1}}.\overline{s} + \overline{a_{N-1}}.b_{N-1}.s$$

L'exercice E3.9 revient sur cette expression.

3.2.4 Multiplication et division par une puissance de 2

Multiplier par 2 un nombre codé en complément à 2 se fait, comme pour un naturel, en ajoutant un 0 en poids faible.

Diviser par 2 consiste, comme pour les naturels, à décaler tous les chiffres d'une position vers les poids faibles, mais c'est la partie entière du quotient qui est obtenue.

La différence notable est que si l'on travaille sur un nombre de bits fixé, ce décalage doit se faire en maintenant le bit de poids fort, le bit de signe.

Cela explique pourquoi dans les jeux d'instructions des processeurs il y a toujours deux types de décalages vers les poids faibles, l'un nommé logique, dans lequel un 0 est inséré en poids fort, l'autre nommé arithmétique où le bit de signe est maintenu.

La division par 2 des entiers relatifs, qui revient à diviser par 2 la valeur absolue de l'entier en conservant son signe, nécessite quelques précautions pour les entiers négatifs impairs.

Le décalage arithmétique ne tient en effet aucun compte de la valeur du bit de poids faible. Or changer de 0 à 1 le bit de poids faible d'un entier pair a pour effet d'en augmenter la valeur absolue s'il est positif ou nul, et au contraire de la diminuer s'il est négatif.

Pour en tenir compte, il faut au préalable ajouter 1 aux seuls entiers négatifs avant d'effectuer le décalage vers les poids faibles. Si l'entier est pair, ceci ne modifie que le bit de poids faible qui est ensuite ignoré lors du décalage.

Si l'entier est impair, cette opération le ramène à l'entier pair de valeur absolue immédiatement inférieure. Ainsi, pour l'entier -7, on appliquera en fait le décalage sur l'entier -6.

Ecriture de N en décimal	Ecriture de N en complément à 2	Ecriture de $N \times 2$ en complément à 2	Ecriture de $N/4$ en complément à 2
13	001101	011010	000011
29	011101	impossible	000111
-6	111010	110100	111101
-7	111001	110010	111110
-21	101011	impossible	110110

4. Lien entre l'arithmétique et les booléens

Le fait que les chiffres binaires 0 et 1 se représentent par les booléens 0 et 1 amène souvent à faire des amalgames de types. Ainsi on assimile parfois $\overline{a}$ et $1 - a$ (en traitant le booléen a comme un entier). En déduire l'existence d'une soustraction booléenne est une grosse erreur.

Les vecteurs booléens peuvent représenter des nombres, on vient de le voir. On a vu dans le chapitre 2 que les opérations booléennes existent aussi sur les vecteurs : l'addition booléenne, nommée aussi OU bit à bit, la multiplication booléenne, nommée aussi ET bit à bit et la complémentation.

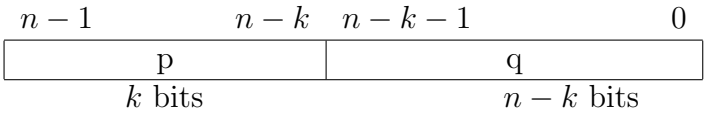
Que signifieraient ces opérations appliquées à des vecteurs représentant des entiers ? Elles gardent leurs propriétés algébriques, mais sont peu intéressantes arithmétiquement. L'addition booléenne ne correspond pas à l'addition des naturels ou des relatifs représentés par les deux vecteurs. De même pour la multiplication. On obtient, par exemple, sur 8 bits :

$$01110000_2 \text{ ET } 01011011_2 = 112_{10} \text{ ET } 91_{10} = 01010000_2 = 80_{10}$$
$$01110000_2 \text{ OU } 01011011_2 = 112_{10} \text{ OU } 91_{10} = 01111011_2 = 123_{10}$$

La seule opération intéressante pour l'arithmétique est la troncature : pour tronquer un naturel A, représenté sur N bits, à P bits (avec $P < N$), il suffit de calculer le ET entre A et un vecteur ayant des 0 en poids forts et P 1 en poids faibles : $0 \ldots 01 \ldots 1$. Ce vecteur représente le naturel $2^P - 1$.

On a donc A ET $(2^P - 1) = A$ modulo 2^P.

Si un naturel X est codé sur n bits, on peut le décomposer en deux naturels p et q, respectivement codés sur k et $n - k$ bits. Si p est la partie poids fort et q la partie poids faible, selon le tableau :



on a les relations suivantes :

$$X = p \times 2^{n-k} + q, \qquad q = X \text{ modulo } 2^{n-k}, \qquad p = X \text{ div } 2^{n-k}$$

Le ET, le OU et le OU exclusif sur les vecteurs de N bits servent aussi : à connaître le bit de rang i d'un nombre X (en calculant X ET 2^i) ; à forcer à 0 le bit de rang i d'un nombre X (par X ET $(2^N - 1 - 2^i)$) ; à forcer à 1 le bit de rang i d'un nombre X (par X OU 2^i) ; à inverser le bit de rang i d'un nombre X (par X OUEX 2^i).

5. Les caractères

Les caractères alphabétiques, numériques, typographiques (parenthèse, virgule, etc.) et certains caractères non imprimables (fin de ligne, fin de fichier, etc.) sont habituellement représentés sur 7 bits selon un code normalisé nommé code ASCII pour *American Standard Code for Information Interchange* (Cf. Figure 3.6).

Le code ASCII est tel que : l'entier représentant un chiffre vaut la valeur du chiffre plus 48 ; les entiers correspondant aux codes de deux lettres sont ordonnés comme les deux lettres dans l'alphabet si les deux lettres sont toutes les deux en majuscules ou en minuscules ; la différence entre le code d'une majuscule et de la minuscule correspondante est 32, c'est-à-dire une puissance de 2.

Sur une machine UNIX la commande `man ascii` fournit en hexadécimal le tableau des codes correspondant aux caractères. Comme on le voit sur la figure 3.6, 23_{16} est le code hexadécimal de # et 20_{16} celui de l'espace ; `del`, de code $7F_{16}$, est le caractère d'effacement. Les codes inférieurs à 1F représentent des caractères non affichables.

Ce code ne permet pas de représenter les lettres accompagnées de diacritiques (accents, cédille, tréma, tilde, petit rond, etc.) dans les langues qui les utilisent (c'est-à-dire presque toutes les langues européennes !). Des extensions à 8 bits, puis à 16 sont proposées (UNICODE), mais les standards sont difficiles à établir. Le problème de pouvoir coder en binaire l'ensemble de toutes les formes écrites des principales langues écrites du monde n'est pas encore totalement résolu. Cela pose de nombreux problèmes lors des transmissions de fichiers contenant des textes.

20	␣	21	!	22	"	23	#	24	$	25	%	26	&	27	'
28	(| 29 |)	2A	*	2B	+	2C	,	2D	-	2E	.	2F	/
30	0	31	1	32	2	33	3	34	4	35	5	36	6	37	7
38	8	39	9	3A	:	3B	;	3C	<	3D	=	3E	>	3F	?
40	@	41	A	42	B	43	C	44	D	45	E	46	F	47	G
48	H	49	I	4A	J	4B	K	4C	L	4D	M	4E	N	4F	O
50	P	51	Q	52	R	53	S	54	T	55	U	56	V	57	W
58	X	59	Y	5A	Z	5B	[	5C	\	5D	]	5E	^	5F	_
60	‘	61	a	62	b	63	c	64	d	65	e	66	f	67	g
68	h	69	i	6A	j	6B	k	6C	l	6D	m	6E	n	6F	o
70	p	71	q	72	r	73	s	74	t	75	u	76	v	77	w
78	x	79	y	7A	z	7B	{	7C	|	7D	}	7E	~	7F	del

FIG. 3.6 – Code ASCII

6. Les nombres réels, la virgule flottante

Les nombres rationnels sont généralement représentés par un couple d'entiers. Mais ils sont peu utilisés en tant que tels dans les ordinateurs à bas niveau. Seuls les langages évolués les manipulent et le soin de réaliser les opérations est alors à la charge du compilateur ou de l'interpréteur. Les nombres réels sont représentés et manipulés à bas niveau dans la plupart des ordinateurs contemporains. Il existe des circuits de calcul sur des réels et, par voie de conséquence, des instructions dans le langage machine qui manipulent des réels. Ces réels sont-ils des irrationnels ? Evidemment non. Des réels non rationnels ont nécessairement une suite infinie non périodique de *décimales*. Les représenter en base 2 ne change rien : ils ont une suite infinie non périodique de *duomales*.

On représente en machine un ensemble fini de réels, en fait des rationnels, selon une technique très proche de la représentation dite scientifique des calculettes. Au lieu de représenter $-123,5$ par -1.235×10^2, on le représente par $-1,1110111 \times 2^6$ (car $123_{10} = 1111011_2$). $-1,1110111$ reçoit le nom de *mantisse* et 6 celui d'*exposant*.

La représentation en décimal en notation scientifique a toujours un chiffre (un seul) avant la virgule, chiffre qui n'est 0 que pour la représentation de 0. La même propriété vaut pour le binaire et le seul chiffre possible avant la virgule étant 1, il n'est pas nécessaire de le représenter explicitement. On parle de 1 caché, et c'est ce qui explique la composante $(1+fr)$ dans le tableau ci-dessous.

Le nombre de chiffres de la mantisse fixe la précision représentable. L'exercice E3.15 sensibilise à la précision dans une représentation basée sur le même principe que la virgule flottante.

La norme I.E.E.E. 754 fixe les formats possibles de la mantisse, de l'exposant, du signe. Selon cette norme, il existe 3 formats de représentation : les réels sont codés sur 32, 64 ou 128 bits. Dans chaque cas la représentation

Taille totale	32 bits	64 bits	128 bits
Taille de S $0 \leq s \leq 1$	1 bit	1 bit	1 bit
Taille de E $0 \leq e \leq 2^{8,11,15}$	8 bits	11 bits	15 bits
Taille de F $0 \leq f \leq 2^{23,52,112}$	23 bits	52 bits	112 bits
Valeur de la partie fractionnaire fr	$fr = f \times 2^{-24}$	$fr = f \times 2^{-53}$	$fr = f \times 2^{-113}$
Valeur normale de e	$0 < e < 255$	$0 < e < 2047$	$0 < e < 32767$
Valeur de X cas normal $e \neq 0, f \neq 0$	$(-1)^s \times 2^{e-127}$ $\times(1 + fr)$	$(-1)^s \times 2^{e-1023}$ $\times(1 + fr)$	$(-1)^s \times 2^{e-16383}$ $\times(1 + fr)$
Valeur de X si $e = 0$ $X = 0$ si de plus $f = 0$	$(-1)^s \times 2^{-126}$ $\times(0 + fr)$	$(-1)^s \times 2^{-1022}$ $\times(0 + fr)$	$(-1)^s \times 2^{-16382}$ $\times(0 + fr)$
Cas particuliers : $e =$	255	2047	32767

FIG. 3.7 – Représentation des réels

comporte 3 champs nommés S (signe), E (exposant) et F (mantisse, ou plutôt partie *f*ractionnaire). Nommons s, e, f le naturel représenté par le champ de bits S, E, F et fr la valeur de la partie fractionnaire.

Le tableau de la figure 3.7 donne les correspondances entre s, e et f et la valeur du réel X représenté selon la taille. Les cas particuliers correspondent aux cas infinis.

7. Exercices

E3.1 : Expression booléenne d'une propriété arithmétique
Considérons un naturel A codé sur N bits. Donner l'expression booléenne caractérisant les bits de A pour que $10 \times A$ soit aussi codable sur N bits.

E3.2 : Reste modulo $2^N - 1$
Retrouver dans ses cahiers d'école élémentaire la technique de la preuve par 9. Se remémorer comment l'on obtient le reste modulo 9 d'un naturel à partir de son écriture en base 10 (à chaque fois qu'il y a au moins 2 chiffres on les ajoute). Ecrire un nombre en octal. Appliquer la technique précédente sur les chiffres octaux. Vérifier que l'on obtient le reste modulo 7. Calculer de même le reste modulo 15 d'un naturel à partir de son écriture hexadécimale.

E3.3 : Manipulation du complément à 2
Ecrire sur 4 bits les relatifs de -8 à +7. Ecrire sur 5 bits les relatifs de -16 à +15. Se persuader que la définition intuitive, celle du compteur de voiture, et les deux équations donnant la correspondance entre valeur et écriture donnent

bien les mêmes résultats. Se persuader de l'unicité de l'écriture. Repérer -8 (sur 4 bits), et -16 (sur 5) comme un cas particulier dont l'opposé n'est pas représentable.

E3.4 : Ecriture des nombres à virgule
Se persuader que l'écriture 0,011 (en base 2), peut valablement représenter le nombre 0,375 (en décimal), c'est-à-dire 1/4 + 1/8. Les nombres à virgule représentables en base 2 et les nombres représentables en base 10 ne sont pas les mêmes. Dans quel sens est l'inclusion, pourquoi ?

E3.5 : Comparaison d'entiers
Pour comparer deux entiers une solution est de calculer leur différence. Mais ce n'est pas nécessaire. La comparaison ayant pour but de dire si les deux entiers sont égaux, et, sinon, quel est le plus grand, trouver des algorithmes de comparaisons de deux entiers à partir de leurs représentations binaires sur N bits :

- dans le cas où les deux nombres sont naturels,
- dans le cas où les deux sont signés (et représentés en complément à 2),
- dans le cas où un nombre est signé et l'autre non.

On pourra compléter cet exercice après l'étude des circuits combinatoires.

E3.6 : Additions en binaire pur et en complément à 2
Dans le tableau 3.8, on montre des résultats d'addition. La table se présente comme une table d'addition, lignes et colonnes. Elle est donc symétrique. Chaque information numérique est représentée de 4 façons : un vecteur de 4 bits, écrits en petits chiffres ; un naturel compris entre 0 et 15 (son écriture en binaire est le vecteur de 4 bits) ; un entier relatif entre -8 et +7 (son écriture en complément à 2 est le vecteur de 4 bits).

Dans chaque case du tableau figurent ces 3 représentations, la valeur du report sortant r_3 provenant de l'addition restreinte aux 3 premiers bits, la valeur du report sortant r_4 provenant de l'addition sur 4 bits. Les résultats corrects sont encadrés. Les résultats incorrects ne le sont pas.

L'objet de l'exercice est de retrouver, d'après ce tableau, les modes de calcul des indicateurs C et V précisant respectivement si le résultat est correct ou non en binaire (pour C) et en complément à 2 (pour V). On peut faire le même travail pour la soustraction. La table n'est pas symétrique dans ce cas.

E3.7 : Signification et test des indicateurs
Quels sont les entiers codables sur 32 bits en complément à 2 et dont la valeur absolue est aussi codable sur 32 bits en complément à 2 ?

Pour résoudre la suite de cet exercice, il faut connaître la programmation en langage d'assemblage.

Dans le programme suivant en langage d'assemblage, il manque un mnémonique d'instruction de branchement conditionnel, il a été remplacé par

	(0011) 3_b $+3_{c2}$	(0100) 4_b $+4_{c2}$	(0101) 5_b $+5_{c2}$	(1011) 11_b -5_{c2}	(1100) 12_b -4_{c2}	(1101) 13_b -3_{c2}
(0011) 3_b $+3_{c2}$	(0110) $\boxed{6_b}$ $\boxed{+6_{c2}}$ $r_3 = 0$ $r_4 = 0$	(0111) $\boxed{7_b}$ $\boxed{+7_{c2}}$ $r_3 = 0$ $r_4 = 0$	(1000) $\boxed{8_b}$ -8_{c2} $r_3 = 1$ $r_4 = 0$	(1110) $\boxed{14_b}$ $\boxed{-2_{c2}}$ $r_3 = 0$ $r_4 = 0$	(1111) $\boxed{15_b}$ $\boxed{-1_{c2}}$ $r_3 = 0$ $r_4 = 0$	(0000) 0_b $\boxed{0_{c2}}$ $r_3 = 1$ $r_4 = 1$
(0100) 4_b $+4_{c2}$	(0111) $\boxed{7_b}$ $\boxed{+7_{c2}}$ $r_3 = 0$ $r_4 = 0$	(1000) $\boxed{8_b}$ -8_{c2} $r_3 = 1$ $r_4 = 0$	(1001) $\boxed{9_b}$ -7_{c2} $r_3 = 1$ $r_4 = 0$	(1111) $\boxed{15_b}$ $\boxed{-1_{c2}}$ $r_3 = 0$ $r_4 = 0$	(0000) 0_b $\boxed{0_{c2}}$ $r_3 = 1$ $r_4 = 1$	(0001) 1_b $\boxed{+1_{c2}}$ $r_3 = 1$ $r_4 = 1$
(0101) 5_b $+5_{c2}$	(1000) $\boxed{8_b}$ -8_{c2} $r_3 = 1$ $r_4 = 0$	(1001) $\boxed{9_b}$ -7_{c2} $r_3 = 1$ $r_4 = 0$	(1010) $\boxed{10_b}$ -6_{c2} $r_3 = 1$ $r_4 = 0$	(0000) 0_b $\boxed{0_{c2}}$ $r_3 = 1$ $r_4 = 1$	(0001) 1_b $\boxed{+1_{c2}}$ $r_3 = 1$ $r_4 = 1$	(0010) 2_b $\boxed{+2_{c2}}$ $r_3 = 1$ $r_4 = 1$
(1011) 11_b -5_{c2}	(1110) $\boxed{14_b}$ $\boxed{-2_{c2}}$ $r_3 = 0$ $r_4 = 0$	(1111) $\boxed{15_b}$ $\boxed{-1_{c2}}$ $r_3 = 0$ $r_4 = 0$	(0000) 0_b $\boxed{0_{c2}}$ $r_3 = 1$ $r_4 = 1$	(0110) 6_b $+6_{c2}$ $r_3 = 0$ $r_4 = 1$	(0111) 7_b $+7_{c2}$ $r_3 = 0$ $r_4 = 1$	(1000) 8_b $\boxed{-8_{c2}}$ $r_3 = 1$ $r_4 = 1$
(1100) 12_b -4_{c2}	(1111) $\boxed{15_b}$ $\boxed{-1_{c2}}$ $r_3 = 0$ $r_4 = 0$	(0000) 0_b $\boxed{0_{c2}}$ $r_3 = 1$ $r_4 = 1$	(0001) 1_b $\boxed{+1_{c2}}$ $r_3 = 1$ $r_4 = 1$	(0111) 7_b $+7_{c2}$ $r_3 = 0$ $r_4 = 1$	(1000) 8_b $\boxed{-8_{c2}}$ $r_3 = 1$ $r_4 = 1$	(1001) 9_b $\boxed{-7_{c2}}$ $r_3 = 1$ $r_4 = 1$
(1101) 13_b -3_{c2}	(0000) 0_b $\boxed{0_{c2}}$ $r_3 = 1$ $r_4 = 1$	(0001) 1_b $\boxed{+1_{c2}}$ $r_3 = 1$ $r_4 = 1$	(0010) 2_b $\boxed{+2_{c2}}$ $r_3 = 1$ $r_4 = 1$	(1000) 8_b $\boxed{-8_{c2}}$ $r_3 = 1$ $r_4 = 1$	(1001) 9_b $\boxed{-7_{c2}}$ $r_3 = 1$ $r_4 = 1$	(1010) 10_b $\boxed{-6_{c2}}$ $r_3 = 1$ $r_4 = 1$

FIG. 3.8 – Table d'addition

`bxx`. A l'état initial, le registre `i0` contient une valeur entière x. A l'état final, le registre `i2` contient 1 si la valeur absolue de x est codable sur 32 bits en complément à 2, et alors `i3` contient cette valeur absolue ; le registre `i2` contient 0 si cette valeur absolue n'est pas codable ; dans ce cas la valeur de `i3` n'est pas pertinente.

```
          cmp    i0, 0       ! si i0 ≥ 0
          bge    pos
   neg :  subcc  0, i0, i3   ! i3 prend pour valeur (-i0)
          bxx    spe
          mov    1, i2       ! OK prend pour valeur vrai
          ba     fin
   pos :  mov    1, i2
          mov    i0, i3      ! si pos. la valeur absolue est le nombre
          ba     fin
   spe :  mov    0, i2       ! OK prend pour valeur faux
   fin :                     ! état final
```

Pourrait-on remplacer le `bge` de la deuxième ligne par un `bpos` ? Par quel mnémonique faut-il remplacer `bxx` ?

E3.8 : Arithmétique saturée

En arithmétique saturée, il n'y a ni retenue, ni débordement. Quand le résultat est *trop grand*, il est remplacé par le plus grand nombre représentable dans le système de numération utilisé. Quand le résultat est *trop petit*, il est remplacé par le plus petit nombre représentable dans le système de numération utilisé.

Ainsi sur 8 bits, avec des exemples écrits en décimal :

- En binaire pur : $200_{10} +_{SatBinpur} 80_{10} = 255_{10}$ au lieu de 280_{10}
  $80_{10} -_{SatBinpur} 200_{10} = 0_{10}$ au lieu de -120_{10}
- En complément à 2 : $100_{10} +_{SatCompl2} 80_{10} = 127_{10}$ au lieu de 180_{10}
  $-80_{10} -_{SatCompl2} 100_{10} = -128_{10}$ au lieu de -180_{10}

Question 1 : *Pour résoudre cette question, il faut connaître la programmation en langage d'assemblage.*

On suppose que A et B sont deux entiers, codés sur 32 bits (attention les exemples sont sur 8 bits). Ils sont rangés dans des registres 32 bits d'un processeur ne disposant pas des opérations en format saturé. Comme les opérations en arithmétique saturée n'existent pas, il convient de les remplacer par un petit programme qui produise le résultat voulu. Ecrire les instructions qui effectuent la soustraction saturée en binaire pur de A et B et range le résultat dans un registre.

Ecrire les instructions qui effectuent l'addition saturée en complément à 2 de A et B et range le résultat dans un registre.

Question 2 : *On peut revenir sur cet exercice après le chapitre sur les circuits combinatoires.*

Donner la description détaillée d'une Unité Arithmétique qui effectue sur deux entiers A et B : l'addition et la soustraction (A + B ou A − B), en binaire

pur et en complément à deux, en arithmétique normale et en arithmétique saturée, selon 3 bits de commande.

> **Remarque :** Les opérations en arithmétique saturée font partir de l'extension MMX du jeu d'instruction des processeurs PENTIUM de INTEL. Elles servent notamment en représentation des images. Si un octet représente le niveau de gris d'un pixel, par exemple 0 pour noir et 255 pour blanc, on peut éclaircir un pixel en augmentant sa luminosité [2], mais il ne faut pas aller au-delà de 255.

E3.9 : Expression du bit de débordement
Soit maj $(x, y, z) = x.y + x.z + y.z$. Montrer que

$$\overline{\text{maj } (x, y, z)} = \text{maj } (\overline{x}, \overline{y}, \overline{z})$$

On note $\oplus$ le OUEXCLUSIF ou XOR. Montrer que

$$a.b.(a \oplus b) = \overline{a}.\overline{b}.(a \oplus b) = 0$$

On pose les équations booléennes :

$$\begin{array}{l} s = a \oplus b \oplus r_e \qquad \text{(on note parfois } s = \oplus(a, b, r_e)) \\ r_s = \text{maj } (a, b, r_e) \end{array}$$

On connaît deux expressions décrivant la valeur du bit de débordement V après une addition :

$$V = a.b.\overline{s} + \overline{a}.\overline{b}.s \quad \text{ et } \quad V = r_s \oplus r_e$$

Montrer que les deux expressions sont équivalentes. Le bit V pour la soustraction est donné par :

$$V = a_{N-1}.\overline{b_{N-1}}.\overline{s} + \overline{a_{N-1}}.b_{N-1}.s$$

Montrer que là aussi $V = r_s \oplus r_e$.

E3.10 : Relation entre binaire pur et complément à 2
Soit un vecteur de bits $y_{N-1}\ y_{N-2}, \ldots, y_1\ y_0$. Soit Y_b le naturel représenté par ce vecteur pour l'interprétation binaire pur. Soit Y_{c2} le relatif représenté par ce vecteur pour l'interprétation complément à 2. Donner des relations entre Y_b, Y_{c2} et y_{N-1}.

E3.11 : Représentation de la somme de deux entiers relatifs
Montrer que, si l'addition de deux nombres relatifs codés en complément à deux déborde, alors la retenue C est l'inverse du bit de signe : $V \Longrightarrow C = \overline{(N)}$

La figure 3.9 donne quelques éléments de réponse. On y représente les 8 cas possibles de valeurs pour le bit poids fort dans une addition en complément à deux. Trouver des entiers codés sur 4 bits dans $[-8, 7]$ pour chacun des cas. Retrouver chacun des cas dans le tableau 3.8. Faire le même travail pour la soustraction.

a_P	b_P	r_e	$r_s = maj(a_P, b_P, r_e)$	$s_P = \oplus(a_P, b_P, r_e)$	$V = r_e \oplus r_s$
signe A	signe B		indic. C	indic. N	indic. V
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	0	1	0
1	0	1	1	0	0
1	1	0	1	0	1
1	1	1	1	1	0

FIG. 3.9 – Représentation de la somme de deux entiers relatifs.

E3.12 : Récupération du résultat d'une addition qui déborde (cas des entiers positifs)

On considère deux entiers positifs A et B, et l'entier $U = A + B$. On suppose que A et B sont représentés en binaire pur sur 32 bits, respectivement dans les registres `%l0` et `%l1` du SPARC. On effectue l'addition grâce à l'instruction `ADDcc %l0, %l1, %l2`.

A et B, entiers *positifs*, étant supposés représentés sur 32 bits, sont donc dans l'intervalle $[0, 2^{32} - 1]$. Lorsque U est représentable en binaire pur sur 32 bits (c'est-à-dire lorsque $U \leq 2^{32} - 1$), on obtient sa représentation dans le registre `%l2` à l'issue de l'instruction d'addition.

Lorsque U n'est pas représentable en binaire pur sur 32 bits (c'est-à-dire $U > 2^{32} - 1$), on dit que l'addition *déborde*. Mais dans ce cas U est représentable sur 64 bits (33 suffiraient). Ecrire un programme en langage d'assemblage qui donne toujours la somme U dans deux registres `%l3, %l2`.

On peut évidemment faire l'exercice analogue pour la différence.

E3.13 : Récupération du résultat d'une addition qui déborde (cas des entiers relatifs)

On reprend l'exercice précédent, dans le cas de la représentation en complément à 2. On considère deux entiers relatifs A et B, et l'entier $U = A + B$. On suppose que A et B sont représentés en complément à deux sur 32 bits, respectivement dans les registres `%l0` et `%l1` du SPARC. On effectue l'addition grâce à l'instruction `ADDcc %l0, %l1, %l2`.

A et B, entiers *relatifs*, étant supposés représentés sur 32 bits, sont donc dans l'intervalle $[-2^{31}, 2^{31} - 1]$ Lorsque U est représentable en C2 sur 32 bits (c'est-à-dire $-2^{31} \leq U \leq 2^{31} - 1$), on obtient sa représentation dans le registre `%l2` à l'issue de l'instruction d'addition.

Lorsque U n'est pas représentable en C2 sur 32 bits (c'est-à-dire $U <$

[2] "Plus blanc que blanc, c'est quoi comme couleur ?" demandait Coluche dans un de ses textes !

-2^{31} ou $U > 2^{31} - 1$), on dit que l'addition *déborde*. Mais dans ce cas U est représentable sur 64 bits (33 suffiraient). Ecrire un programme en langage d'assemblage qui donne toujours la somme U dans deux registres `%l3, %l2`.

On peut évidemment faire l'exercice analogue pour la différence.

E3.14 : Description récursive de l'addition de 2 naturels

Décrire l'addition de deux naturels comme une opération récursive sur la taille des deux naturels, selon l'indication suivante.

Si le nombre N de bits de A et B vaut 1, la somme de A et B est facile à calculer, elle est représentable sur 2 bits. Si N est une puissance de 2 supérieure à 1, $N/2$ est entier ; on peut couper A et B en deux parties $A_{fort}A_{faible}$ et $B_{fort}B_{faible}$, chacune sur $N/2$ bits ; on a alors $A = A_{fort} \times 2^{N/2} + A_{faible}$.

Calculons un report intermédiaire r_{inter} :

$$r_{inter} = 1 \text{ si } A_{faible} + B_{faible} \geq 2^{N/2}$$
$$r_{inter} = 0 \text{ si } A_{faible} + B_{faible} < 2^{N/2}$$

On a alors, pour les poids faibles :

$$S_{faible} = A_{faible} + B_{faible} \text{ si } r_{inter} = 0$$
$$S_{faible} = A_{faible} + B_{faible} - 2^{N/2} \text{ si } r_{inter} = 1$$

et, pour les poids forts :

$$S_N = 1 \text{ si } A_{fort} + B_{fort} + r_{inter} \geq 2^{N/2}$$
$$S_N = 0 \text{ si } A_{fort} + B_{fort} + r_{inter} < 2^{N/2}$$
$$S_{fort} = A_{fort} + B_{fort} + r_{inter} \text{ si } S_N = 0$$
$$S_{fort} = A_{fort} + B_{fort} + r_{inter} - 2^{N/2} \text{ si } S_N = 1$$

E3.15 : Précision en représentation flottante

Les pièces de monnaies courantes en France sont 5, 10, 20 et 50 centimes et 1, 2, 5, 10, 20 Francs. On représente ces pièces par un code binaire.

La première partie du code est l'analogue d'une mantisse de 3 bits m_2, m_1, m_0. Elle prend les valeurs 001, 010 ou 101 pour représenter 1, 2 ou 5 (centimes, dizaine de centimes, francs ou dizaine de francs).

La deuxième partie du code est l'exposant de 10 affectant les centimes (00 pour les centimes, 01 pour les dizaine de centimes, 10 pour les Francs et 11 pour les dizaines de Francs). L'exposant est codé sur 2 bits e_1, e_0.

Les codes des différentes pièces sont donc donnés par le tableau de la figure 3.10.

Il serait possible de compléter ce code pour représenter des sommes d'argent utilisant 2 pièces. On a alors des sommes de 3, 4, 6 ou 7 unités. On obtiendrait une table d'addition pour cette représentation :

$$01001 + 10101 = 11101 (20 \text{ centimes} + 50 \text{ centimes} = 70 \text{ centimes}).$$

$m_2\ m_1\ m_0$	$e_1\ e_0$	pièce	$m_2\ m_1\ m_0$	$e_1\ e_0$	pièce
1 0 1	0 0	5 centimes	0 1 0	1 0	2 Francs
0 0 1	0 1	10 centimes	1 0 1	1 0	5 Francs
0 1 0	0 1	20 centimes	0 0 1	1 1	10 Francs
1 0 1	0 1	50 centimes	0 1 0	1 1	20 Francs
0 0 1	1 0	1 Franc			

FIG. 3.10 – Codage des valeurs de pièces de monnaie française.

Etudier la technique d'addition dans cette représentation, en particulier le cas où 50 centimes + 50 centimes font 1 Franc et autres cas semblables.

Toutefois, on a aussi :

$$00111 + 01001 = 00111(10\ \text{Francs} + 20\ \text{centimes} = 10\ \text{Francs})$$

car cette représentation ne comporte pas assez de chiffres significatifs pour distinguer 10 et 10,2.

Etudier les possibilités offertes par un allongement de la mantisse sur 6 bits par exemple. Etudier la technique d'addition nouvelle. Etudier les représentations de sommes d'argent utilisant 3, 4, ou N pièces.

Dans la représentation en virgule flottante classique, la mantisse a 24 chiffres. Cela permet de ne négliger les centimes que pour des sommes supérieures à 2^{24} centimes. C'est suffisant pour la comptabilité domestique, mais insuffisant pour une comptabilité d'entreprise par exemple.

Chapitre 4

Représentation des traitements et des données : langage d'actions

La programmation des dispositifs informatiques s'appuie sur un ensemble de modèles mathématiques simples, qui permettent de représenter formellement les données et les traitements qui leur sont appliqués. Les langages dits *de haut niveau* qu'on utilise pour écrire des programmes (Pascal, Ada, C, ...) sont des modèles de traitements et de données. Le langage machine d'un processeur particulier, ou un langage d'assemblage défini pour ce processeur, sont également des modèles de traitements, qualifiés de modèles *de bas niveau*. Cette notion de niveau correspond au niveau d'abstraction auquel on se place pour écrire des programmes : les modèles de bas niveau sont proches de la machine, alors que les modèles de haut niveau permettent de s'en abstraire ; d'ailleurs les programmes écrits en langage de haut niveau peuvent être rendus indépendants de la machine sur laquelle on les exécute. La définition rigoureuse de la sémantique de ces modèles, à tous les étages, est indispensable pour assurer la correction des diverses transformations nécessaires pour passer d'une représentation de traitement dans un langage de haut niveau à un objet exécutable par une machine.

Ceci est valable en ce qui concerne le logiciel — les étapes de la *compilation* d'un langage de haut niveau vers un langage machine particulier (Cf. Chapitres 12, 13, 18) — aussi bien que pour le matériel — les étapes de la *traduction* d'un langage de description de circuits vers une réalisation à l'aide d'une technologie particulière (Cf. Chapitres 8, 11 et 10).

Les objectifs de ce chapitre et du suivant sont : a) définir les langages et les modèles mathématiques utilisés ; b) donner les éléments nécessaires à la compréhension de l'utilisation de ces objets mathématiques pour représenter des traitements informatiques ; c) donner la première étape de traduction des modèles de haut niveau vers des modèles de plus bas niveau. L'étape suivante est la traduction en langage d'assemblage (Cf. Chapitre 13).

Le paragraphe 1. présente un petit langage d'actions (structures de données et structures de contrôle). Le paragraphe 2. étudie la représentation en mémoire des types de base et des structures de données ; nous introduisons le tableau MEM *qui modélise la mémoire d'un ordinateur. Le paragraphe 3. montre comment transformer systématiquement les affectations du langage d'actions en accès au tableau* MEM*. Le paragraphe 4. illustre sur un exemple de construction de séquence chaînée le problème de* l'allocation dynamique *de mémoire nécessaire à la manipulation des structures de données récursives comme les séquences chaînées et les arbres. Le dernier paragraphe s'intéresse à la fois aux traitements et aux données : la section 5. introduit les structures de piles et de files, en étudiant à la fois la représentation en mémoire et les algorithmes associés.*

1. Un langage d'actions

Le langage d'actions que nous décrivons brièvement ci-dessous est tiré de [SFLM93]. Nous supposons connues les notions de *variable* dans un langage de programmation impératif, de *type* des données.

1.1 Lexique : nommage des types et variables

Un algorithme commence toujours par un *lexique*, qui nomme en particulier les *types* et les *variables* utilisés :

entier18 : le type entier dans $[-2^{18-1}, 2^{18-1} - 1]$
a, b, c : des entier18

1.2 Types de base et types construits

1.2.1 Types de base

La représentation des types de base entier naturel, entier relatif, réel et caractère par des vecteurs de booléens a été vue au chapitre 3. On se donne une notation de ces types de base : entier, caractère, réel, booléen. Pour les entiers on s'autorise une spécification d'intervalle ; on écrira par exemple : entier dans [0..255].

1.2.2 Construction de types, structures de données usuelles

Nous étudions ici les structures de données offertes par les constructeurs de types usuels des langages de programmation (n-uplets, tableaux, pointeurs).

Pour décrire un type construit et le nommer, on écrit :

T : le type ...

où les pointillés doivent être complétés par une expression de type, utilisant l'un des constructeurs décrits ci-dessous.

N-uplets Le constructeur de type *n-uplet* permet de grouper des informations de types différents et de les manipuler comme un tout. On notera ces groupements par des chevrons :

```
T1 : le type ...
T2 : le type ...
Structure12 : le type < x : un T1, y : un T2 >
S : un Structure12
```

x et y sont des noms qui désignent les *champs* de la structure. T1 et T2 sont des types quelconques définis par ailleurs. Etant donné un objet S de type Structure12, on accède aux informations élémentaires du n-uplet par l'opération de sélection des champs, notée . ; on écrit ainsi S.x, S.y.

Le constructeur n-uplet correspond aux *struct* des langages C et C++, aux *record* des langages Pascal et Ada.

Tableaux Le constructeur de type *tableau* permet de grouper des informations de même type et d'y accéder par un *indice*. On note les tableaux par des crochets :

```
Elem : le type ...
Tab : le type tableau sur [...] de Elem
```

En général [...] doit être complété par la notation d'un type intervalle. En Pascal ou Ada, ces intervalles peuvent eux-mêmes être définis d'après des types énumérés généraux. En C les tableaux sont toujours définis sur un intervalle de la forme [0..N], où N est un entier strictement positif. Pour la suite de l'exposé, nous nous restreignons à des intervalles d'entiers. On écrira par exemple :

```
Tab : le type tableau sur [42..56] d'entiers
T : un Tab                                  { T est une variable de type Tab }
```

L'accès aux éléments du tableau est noté par des crochets : T[42], T[43], ou encore T[a+b], si a et b sont des noms de variables de type entier, dont les valeurs sont telles que a+b appartient à l'intervalle [42..56]. On peut aussi utiliser une notation indicée : T_{42}, T_{a+b}.

L'accès aux éléments par un indice permet de *parcourir* tous les éléments d'un tableau dans une boucle. En anticipant sur la notation des traitements (paragraphes 1.4 et 1.5), on écrit typiquement :

```
Tab : le type tableau sur [42..56] d'entiers
T : un Tab
i parcourant [42..56]
    T[i] ⟵ 2 * i
```

Pointeurs La notion de *pointeur* des langages de programmation comme Pascal, C, Ada, etc. est intimement liée à celle d'*adresse*. Nous revenons sur ce constructeur de type dans le paragraphe 2.

Le mot *pointeur* est un constructeur de type. Etant donné un type T, on appelle *pointeur de T* le type des *adresses mémoire d'objets de type T*.

L'opération de *déréférençage* s'applique à un objet de type *pointeur de* T et son résultat est un objet de type T. On la note de manière postfixée par une flèche verticale vers le haut : p↑ est l'objet dont l'adresse est p. On écrit par exemple :

T : un type ; adT : le type pointeur de T ; t1 : un T ; pt : une adT
t1 ⟵ pt↑

Les variables de type *pointeur* peuvent avoir une valeur particulière notée NIL, qui signifie *pointeur sur rien* (Cf. Paragraphe 2.4.3).

1.3 Opérateurs de base et expressions

Les expressions du langage sont formées à partir de noms de variables déclarées dans le lexique, de constantes des types de base, d'opérateurs prédéfinis et d'appels de fonctions.

1.3.1 Expression conditionnelle et opérateurs booléens

Une *expression conditionnelle* a la forme suivante : si C alors E1 sinon E2, où C est une expression de type booléen et E1, E2 deux expressions de même type, quelconque. Noter que les 2 expressions ci-dessous sont équivalentes, bien que différemment factorisées :

(si C1 alors E1 sinon E2) + (si C1 alors E3 sinon E4)
si C1 alors E1+E3 sinon E2+E4

Pour les booléens, on considère les opérateurs de base et, ou, non, ouexcl, etc. hérités de l'algèbre de Boole (Cf. Chapitre 2). On y ajoute les opérateurs booléens dits *séquentiels* (ou *non stricts*) etpuis, oualors (en Ada : `andthen`, `orelse`).

La sémantique de ces opérateurs peut être décrite par une transformation en expression conditionnelle :

expr1 etpuis expr2 { *est identique à :* } si expr1 alors expr2 sinon faux
expr1 oualors expr2 { *est identique à :* } si expr1 alors vrai sinon expr2

1.3.2 Opérateurs sur les nombres et les caractères

Opérations arithmétiques : On utilisera toutes les opérations arithmétiques usuelles : addition, multiplication, division, soustraction, etc., sur les types numériques introduits ici, c'est-à-dire le type entier et le type réel.

Pour les entiers strictement positifs on considère également le reste et le quotient de la division entière, en évitant les problèmes de définition dus au signe des opérandes :

reste, quotient : deux entiers > 0 ⟶ un entier > 0
{ reste(a,b) = r *et* quotient (a,b) = q *si et seulement si* $a = bq + r$, *avec* $0 \leq r < b$ }

L'opération reste est souvent appelée *modulo*.

Opérations sur les caractères : On peut introduire sur le type de base caractère des fonctions comme :

EstLettre ?, EstMajuscule ?, EstChiffre ?, ... : un caractère $\longrightarrow$ un booléen

MajusculeDe, MinusculeDe : un caractère $\longrightarrow$ un caractère

Les premières permettent de déterminer à quel sous-ensemble de caractères appartient un caractère donné. Les deuxièmes sont des fonctions de conversions. Par exemple : MajusculeDe ('a') = 'A'.

Notons que, grâce aux propriétés du code ASCII (Cf. Chapitre 3), toutes ces fonctions peuvent être codées en opérations arithmétiques ou booléennes simples sur la représentation en binaire des caractères. Par exemple, pour passer des majuscules aux minuscules il suffit d'inverser un bit, puisque l'écart entre les codes de deux lettres correspondantes est une puissance de 2.

1.4 Affectation

L'action de base dans un langage d'actions est l'*affectation*, qui permet de modifier la valeur d'une variable. On la note par une flèche orientée à gauche :

X $\longleftarrow$ expr

T[3+z].u $\longleftarrow$ expr

La partie gauche d'une affectation doit pouvoir désigner un emplacement mémoire (nous y revenons dans le paragraphe 3.) ; la partie droite est une *expression*, dont le type doit être *compatible* avec le type de la partie gauche.

Les langages de programmation proposent des notions de compatibilité de types plus ou moins riches, des vérifications statiques associées, ainsi que des conversions dynamiques implicites. Nous nous contenterons ici d'exiger que les types des parties gauche et droite soient *identiques*.

Toutefois on peut avoir besoin d'écrire x $\longleftarrow$ y, où x est un réel et y un entier. Le codage binaire des entiers étant fort différent de celui des réels (Cf. Chapitre 3), la représentation en mémoire de la variable y est nécessairement différente de celle de x.

Pour mettre en évidence la conversion que cache ainsi l'affectation, nous utiliserons des fonctions de conversion de type (ou de changement de représentation mémoire) explicites :

EntierVersRéel : un entier $\longrightarrow$ un réel

{ *EntierVersRéel (a) est le réel de valeur* a }

Naturel31 : le type entier sur $[0, 2^{32-1} - 1]$

Entier32 : le type entier sur $[-2^{32-1}, 2^{32-1} - 1]$

Naturel31VersEntier32 : un Naturel31 $\longrightarrow$ un Entier32

{ NaturelVersEntier (n) *est l'entier de valeur* n }

Nous revenons sur la traduction en assembleur de ces fonctions au chapitre 13. Nous verrons en particulier que la traduction en langage d'assemblage de la fonction Naturel31VersEntier32 est un programme vide ! Au chapitre 3, paragraphe 3.1, nous signalions déjà ce cas.

1.5 Structures conditionnelles et itératives

On se donne les constructions si ... alors ... sinon et si ... alors ... usuelles dans tous les langages de programmation impératifs. Notons que l'on peut ici omettre la partie sinon, alors que c'est impossible pour une *expression* conditionnelle, qui doit avoir une valeur dans tous les cas. Autrement dit, *ne rien faire* est une action particulière.

Noter que les 3 actions suivantes sont équivalentes :

X ⟵ (si C1 alors E1 sinon E2) + (si C1 alors E3 sinon E4)
X ⟵ (si C1 alors E1+E3 sinon E2+E4)
si C1 alors X ⟵ E1+E3 sinon X ⟵ E2+E4

Une construction moins courante est le selon, qui permet de décrire une analyse par cas exhaustive et sans duplication de cas, pour les valeurs d'une ou plusieurs expressions de type quelconque. Dans l'exemple qui suit, A1, A2 et A3 représentent des actions quelconques.

X : un entier
Y : un caractère
selon X, Y
    X ≥ 0 et Y = 'a' : A1
    X ≥ 0 et Y ≠ 'a' : A2
    X < 0 : A3

Cette structure générale doit souvent être codée par une série d'expressions conditionnelles si ... alors ... sinon enchaînées, comme en Pascal, en C, ... Les structures `case` et `switch` de ces langages ne permettent en effet que des conditions de la forme expr = constante, pour des types dont les constantes ont une notation dans le langage, c'est-à-dire les entiers, caractères, types énumérés. La structure selon à conditions quelconques existe en Lisp (`cond`), mais sa sémantique est *séquentielle* et les différentes conditions ne sont pas nécessairement exhaustives.

Nous utilisons par ailleurs 3 structures itératives : parcourant (qui correspond au `for` de Pascal, C, Ada, ...), tantque (qui correspond au `while` de Pascal, C et Ada), répéter ... jusqu'à (qui correspond au `do ... while` de C, au `repeat ... until` de Pascal, au `loop ... while` de Ada).

La sémantique de ces constructions est précisée par leur traduction en *machines séquentielles à actions* (ou *organigrammes*) au chapitre 5.

On peut déjà ramener la structure parcourant à une structure tantque :

i parcourant [a .. b] : A
    { *Est équivalent à :* }
i : un entier sur [a .. b+1]
i ⟵ a
tantque i ≤ b :
    A ; i ⟵ i + 1

1.6 Fonctions et actions paramétrées

Pour définir une *fonction* on écrira :

```
ExpressionCompliquée (a, b : deux entiers) ⟶ un entier
{ a et b sont les noms des paramètres, de type entier, de la fonction nommée
ExpressionCompliquée. Le résultat est de type entier également }
lexique local :
   x : un entier                                  { Pour des calculs intermédiaires }
algorithme
   x ⟵ (a+b)*2
   { Description du résultat de la fonction : }
   ExpressionCompliquée (a,b) : x + x*x
```

Pour définir une *action* on écrira :

```
CalculerExpressionCompliquée : une action
    (les données a, b : deux entiers ;
       { paramètres dont la valeur est utilisée par l'action }
    le résultat r : un entier) { paramètre dont la valeur est modifiée par l'action }
lexique local :
   x : un entier                                  { Pour des calculs intermédiaires }
algorithme
   x ⟵ (a+b)*2 ; r ⟵ x + x*x
```

Un contexte d'utilisation de la fonction ExpressionCompliquée et de l'action CalculerExpressionCompliquée est décrit ci-dessous :

```
u, v, w, w1, w2 : des entiers
w ⟵ ExpressionCompliquée (u, v) + ExpressionCompliquée (2*u, v−1)
CalculerExpressionCompliquée (u, v, w1) ;
CalculerExpressionCompliquée (2*u, v−1, w2) ;
w ⟵ w1+w2
```

Les noms qui apparaissent dans la liste de paramètres de la *définition* d'une action ou fonction sont appelés *paramètres formels*. Les expressions qui apparaissent entre parenthèses dans les *appels* de fonctions ou actions sont appelés *paramètres effectifs* ou *arguments*. Les paramètres effectifs données sont des expressions quelconques du type défini par le paramètre formel correspondant. Les paramètres effectifs résultats sont des expressions qui pourraient figurer en partie gauche d'affectation, c'est-à-dire qui désignent un emplacement mémoire (Cf. Paragraphe 2.2.1 du chapitre 13 pour comprendre cette contrainte).

Les noms définis dans le lexique local ont une *portée* réduite au corps de l'action ou fonction : cela signifie qu'ils ne sont pas utilisables ailleurs dans le texte d'un programme. D'autre part deux variables locales de deux actions ou fonctions différentes peuvent porter le même nom.

1.7 Entrées/Sorties

On utilisera les actions Lire et Ecrire, pour tout type de données, et avec un nombre quelconque de paramètres.

Les paramètres de Lire sont des *résultats*, ceux de Ecrire sont des *données*.

Une utilisation typique est décrite ci-dessous :

```
lexique : x, y : des entiers
Ecrire ("Donnez deux entiers : ") ; Lire (x, y)
Ecrire ("Somme des deux entiers : ", x+y)
```

2. Représentation des données en mémoire

Nous avons vu au chapitre 3 les principes de codage des types de base en binaire. Ce paragraphe traite de deux aspects : 1) la représentation binaire des valeurs des variables d'un langage de programmation (types simples, tableaux, structures, etc.), à partir du codage binaire des types de base ; 2) l'installation des variables d'un programme en mémoire.

Les choix de représentation des types structurés sont en général guidés par une notion de coût (simplicité, complexité en mémoire ou en temps) des opérations de base à réaliser sur les objets du type considéré.

2.1 Une modélisation de la mémoire : le tableau MEM

Nous introduisons le tableau MEM, comme abstraction de la mémoire d'un ordinateur. C'est un tableau à une seule dimension, indicé par les naturels d'un intervalle [0..tmem-1], et dont les éléments représentent les *unités adressables* de la mémoire d'une machine.

L'unité adressable est un vecteur de booléens. Dans une machine réelle c'est presque toujours supérieur au bit ; certaines machines ont proposé des unités adressables de 9 bits. Dans la suite de cet ouvrage nous nous intéressons — sauf mention contraire — au cas des *octets*, c'est-à-dire aux unités adressables de 8 bits. C'est une taille commode pour la représentation du type caractère en mémoire. tmem représente donc la taille de la mémoire en octets.

La notion d'unité adressable, supérieure au bit, est une manière d'exprimer que, dans une machine réelle, des contraintes de réalisation matérielle empêchent d'accéder efficacement à chaque bit de la mémoire individuellement (Cf. Chapitres 9 et 15).

2.2 Représentation en mémoire des types de base

2.2.1 Représentation en mémoire des booléens

L'idéal pour la représentation en mémoire d'une information de type booléen serait d'utiliser 1 bit ; mais il est irréaliste, pour des raisons matérielles, d'accéder à un bit individuel dans la mémoire. On choisit donc la plus petite

taille possible : une unité adressable (voir toutefois le paragraphe 2.4.2 pour le cas particulier des tableaux de booléens, où l'on peut espérer gagner de la place). Il faut convenir d'un codage des deux constantes vrai, faux parmi les 2^k configurations d'une unité adressable de k bits.

Rien n'empêche, a priori, de choisir, vrai $= 42_{10}$ et faux $= 77_{10}$ (sur un octet par exemple). Le choix du *bon* codage dépend essentiellement de la réalisation des opérations dans lesquelles intervient un opérande ou un résultat de type booléen. Il faut penser aux opérations internes du type booléen (conjonction, disjonction, ...) et à la fabrication de valeurs booléennes par comparaison de deux entiers par exemple (qui apparaît bien sûr dans si X < Y alors ... mais aussi dans des expressions de la forme : B ⟵ (X < Y)).

Pour vrai $= 42_{10}$ et faux $= 77_{10}$, il est difficile de décrire la conjonction de deux booléens a et b plus simplement que par :
si a=42 alors si b = 42 alors 42 sinon 77 sinon 77.

Dans le langage C, le choix est le suivant : 0 représente faux, *toute autre valeur* représente vrai ; une conjonction peut alors être réalisée à l'aide de l'opérateur *et logique* disponible sur tous les processeurs.

2.2.2 Représentation en mémoire des entiers

Nous avons supposé l'existence d'un type de base entier. Les types de données qui permettent de définir des entiers dans les langages de programmation usuels correspondent le plus souvent à des entiers *bornés*, c'est-à-dire à des intervalles d'entiers. En C, par exemple, on déclare des entiers en précisant leur taille et en décidant si ce sont des entiers naturels ou relatifs.

Il existe des langages, comme SCHEME [Aa91], dans lesquels les traitements d'entiers sont dits à précision *infinie*. C'est un abus de langage pour exprimer que la taille des entiers manipulables n'est pas statiquement bornée : la simple addition de deux entiers peut provoquer l'allocation d'une zone mémoire supplémentaire nécessaire à la représentation du résultat. Le terme *infini* est abusif puisque, même si l'on consacre toute la mémoire de la machine à la représentation d'un seul entier, l'intervalle des valeurs représentables n'en est pas moins borné.

2.2.3 Problème de la taille des entiers

Si le type entier du langage de haut niveau que l'on considère désigne un intervalle d'entiers suffisamment petit, les valeurs de ce type peuvent être représentées en mémoire dans une seule unité adressable. Par exemple, un octet suffit pour représenter en complément à deux les entiers de l'intervalle $[-2^{8-1}, 2^{8-1}-1]$ ou, en binaire pur, les entiers de l'intervalle $[0, 2^8-1]$ (Cf. Chapitre 3).

Si le type entier désigne un intervalle plus grand, il devient nécessaire d'utiliser *plusieurs* unités adressables pour la représentation d'une seule valeur de type entier. On utilise dans ce cas des unités adressables *contiguës*, et l'on

considère un nombre *entier* d'unités adressables. Pour représenter les entiers de l'intervalle $[-2^{18-1}, 2^{18-1} - 1]$, qui nécessitent 18 bits, on utilisera donc 3 octets. Nous avons vu au chapitre 3, paragraphes 2. et 3., comment étendre la représentation binaire d'un entier à un plus grand nombre de bits.

Notation Nous noterons taille(T) le nombre d'unités adressables nécessaires à la représentation en mémoire d'un objet de type T. Ainsi, par exemple, taille(entier dans $[-2^{18-1}, 2^{18-1} - 1]$) $= 3$, si l'unité adressable est l'octet.

2.2.4 Représentation en mémoire des entiers qui ne tiennent pas dans une unité adressable

Considérons par exemple un entier x quelconque de l'intervalle $[-2^{32-1}, 2^{32-1} - 1]$. Notons $x_{31}....x_0$ le codage en complément à deux de x, qui nécessite bien 32 bits, donc 4 octets. La suite de 32 bits $x_{31}....x_0$ doit être découpée en 4 portions de 8 bits, évidemment contigus, ce qui donne : $t_4 = x_{31}...x_{24}$, $t_3 = x_{23}...x_{16}$, $t_2 = x_{15}...x_8$, $t_1 = x_7...x_0$. Notons qu'une de ces tranches, prise isolément, n'a pas nécessairement de sens par rapport à la valeur de l'entier x. Par exemple, seul le bit de poids fort de la tranche t_4 porte l'information sur le signe de x, en cas de codage en complément à 2.

Pour représenter x en mémoire, on utilise 4 unités adressables contiguës, c'est-à-dire 4 cases consécutives du tableau MEM : MEM[a], MEM[a+1], MEM[a+2] et MEM[a+3].

Un choix subsiste sur le placement des 4 tranches t_1, t_2, t_3 et t_4 dans les cases MEM[a], MEM[a+1], MEM[a+2] et MEM[a+3]. Comme on respecte l'ordre entre les tranches, les deux choix possibles sont :

- t_1 dans MEM[a], t_2 dans MEM[a+1], t_3 dans MEM[a+2] et t_4 dans MEM[a+3] ; ce placement est appelé *petit boutiste* : les poids faibles de x apparaissent en premier, dans l'ordre des adresses.
- t_1 dans MEM[a+3], t_2 dans MEM[a+2], t_3 dans MEM[a+1] et t_4 dans MEM[a] ; ce placement est appelé *gros boutiste* : les poids forts de x apparaissent en premier, dans l'ordre des adresses.

  Remarque : L'existence de ces deux conventions différentes est une cause importante de non compatibilité entre systèmes informatiques, dès qu'il faut transférer des fichiers. Dans le domaine des réseaux, il existe un standard, c'est le choix *gros boutiste*. Sur les machines qui font le choix inverse, les données doivent être transformées avant d'être transmises. Voir aussi l'exercice E4.1.

2.3 Les accès au tableau MEM

Nous venons de voir que des variables de types simples comme les entiers peuvent nécessiter plusieurs unités adressables. Nous nous intéressons donc au problème de l'accès simultané à plusieurs unités adressables contiguës.

La situation décrite ci-dessous n'est pas la plus générale que l'on pourrait imaginer. Elle est guidée par les contraintes matérielles de liaison entre le processeur et la mémoire, que nous étudierons au chapitre 15.

Tout d'abord, nous ne nous intéressons qu'au cas de blocs d'unités adressables en nombre égal à une puissance de 2 (pour ne pas perdre d'espace d'adressage, Cf. Chapitre 15). D'autre part, sur la plupart des machines, les accès ne sont permis que lorsque l'adresse est un multiple de la taille du transfert (les autres accès ne sont pas nécessairement implémentés parce qu'ils sont moins efficaces). Cette restriction est connue sous le nom de *contrainte d'alignement mémoire*.

Les contraintes matérielles d'accès à la mémoire ont également pour conséquence que les accès simultanés à un nombre *quelconque* d'unités adressables ne peuvent pas constituer des opérations élémentaires dans une machine (un processeur) usuel. Les affectations de mémoire présentées ci-dessous, indicées par le nombre d'unités à transférer, sont en petit nombre, fixé.

Nous noterons $\longleftarrow_k$ une affectation de taille k, c'est-à-dire un transfert simultané de k unités adressables. Nous considérons par la suite les affectations :

x $\longleftarrow_1$ MEM[a]
{ *L'unité adressable d'indice* a *dans le tableau* MEM *est copiée dans la variable* x *(supposée de taille adéquate)* }

x $\longleftarrow_2$ MEM[a]
{ *Valide si* a *est multiple de 2. Les deux unités adressables d'indices* a *et* a*+1 sont copiées dans la variable* x *(supposée de taille adéquate).* }

x $\longleftarrow_4$ MEM[a]
{ *Valide si* a *est multiple de 4. Les quatre unités adressables d'indices* a, a*+1,* a*+2 et* a*+3 sont copiées dans la variable* x *(supposée de taille adéquate).* }

Il existe en général une opération élémentaire de transfert de 4 octets dans les machines dites 32 bits, une opération de transfert de 8 octets dans les machines 64 bits, ...

2.4 Représentation en mémoire des types construits

2.4.1 Représentation en mémoire des n-uplets

Les n-uplets, de même que les entiers suffisamment grands, demandent plusieurs unités adressables. On utilise lorsque c'est possible des unités contiguës. Considérons les définitions de type :

T1 : le type entier dans $[-2^{32-1}, 2^{32-1} - 1]$
T2 : le type entier dans $[-2^{16-1}, 2^{16-1} - 1]$
Structure12 : le type $<$ x : un T1, y : un T2 $>$
Structure21 : le type $<$ x : un T2, y : un T1 $>$

Une valeur de type Structure12 occupe 6 unités adressables consécutives, d'adresses a, $a+1$, ... $a+5$. Le champ x commence à l'adresse a, et le champ y à l'adresse $a+4$.

En suivant le même raisonnement que précédemment, une valeur de type Structure21 semble pouvoir occuper 6 unités adressables consécutives, d'adresses a, $a+1$, ... $a+5$. Le champ x commence à l'adresse a, et le champ y à l'adresse $a+2$. Toutefois le champ y est de taille 4 si l'unité adressable est l'octet. Si l'on veut pouvoir accéder à ce champ globalement (un seul accès mémoire), son adresse doit être un multiple de 4. De même le champ x est de taille 2, donc son adresse doit être paire.

Ces contraintes d'alignement en mémoire empêchent de placer un objet de type Structure21 à une adresse quelconque. De plus, pour satisfaire la contrainte d'alignement pour le champ y, on doit ménager un espace entre le champ x et le champ y.

Nous donnons au paragraphe 2.4.2 une solution de représentation en mémoire qui évite de perdre trop de place dans le cas d'un tableau de structures.

> *La directive d'alignement* `.align` *usuelle dans les langages d'assemblage est introduite au chapitre 12 et son utilisation dans la traduction des langages de haut niveau en langage d'assemblage est étudiée au chapitre 13.*

Remarque : Certaines machines (PENTIUM MMX, SPARC VIS) proposent des instructions spécifiques et un codage efficace pour une structure particulière qui permet de décrire une couleur : elle comporte 4 champs r, g, b, l pour les proportions de rouge, vert (*green*) et bleu, et la luminosité.

2.4.2 Représentation en mémoire des tableaux

Comme mentionné plus haut, un tableau permet de grouper des informations de même type et d'y accéder par un *indice*.

Placer les éléments du tableau dans des unités adressables consécutives permet d'exprimer simplement l'adresse d'un élément du tableau en fonction de son indice et de l'adresse de début du tableau. Le fait que l'adresse d'un élément soit ainsi *calculable* conduit à un codage simple des boucles d'accès au tableau (Cf. le paragraphe sur l'optimisation des parcours de tableaux ci-dessous).

Tableaux à une dimension Considérons le type Tab :

Tab : le type tableau sur [42..56] d'entiers dans $[-2^{32-1}, 2^{32-1}-1]$

Une valeur T de ce type nécessite $4 \times (56-42+1)$ unités adressables. 4 est le nombre d'unités nécessaires pour un élément, et $(56-42+1)$ est le nombre d'éléments du tableau. Si a est l'adresse de la première unité adressable utilisée pour T, l'élément T[i] occupe les unités d'adresses $a+d+0$, $a+d+1$, $a+d+2$ et $a+d+3$, où $d = (i-42) \times 4$.

Dans le cas particulier où l'intervalle des indices du tableau commence à 0, par exemple T : un tableau sur [0..N-1] de T', la formule qui donne l'adresse de T[i] en fonction de l'adresse a de début de T est plus simple : $d = i \times$ taille (T').

La prise en compte des contraintes d'alignement peut imposer de ménager des espaces perdus entre les éléments du tableau. Si le type T' des éléments

est tel que deux objets de type T' peuvent toujours être placés côte à côte en mémoire, il n'y a pas de place perdue. C'est le cas par exemple pour T' : le type < c1, c2, c3 : des caractères >.

En revanche, si T' est le type Structure12 étudié au paragraphe précédent, on doit ménager un espace de deux octets entre deux éléments, de manière à satisfaire la contrainte d'alignement sur des adresses multiples de 4 du champ x.

On peut conserver la formule qui donne l'adresse T[i] en fonction de l'adresse a de début de T, à condition de redéfinir la notion de taille nécessaire à la représentation d'un type. Par exemple, taille_align (Structure12) = 8, et non 6.

> **Remarque :** cette fonction correspond à la macro-notation `sizeof` du langage C, applicable à un nom de type ou à une expression typée.

Cas particulier des tableaux de booléens Nous avons vu plus haut qu'un booléen seul occupe un octet. Lorsqu'on considère un tableau de booléens, il devient intéressant d'essayer de gagner de la place en choisissant une représentation plus compacte. Considérons le tableau T défini par :

T : un tableau sur [0, N−1] de booléens

Les éléments de T peuvent être placés en mémoire à partir d'une adresse a, à raison d'un élément par bit. Le tableau complet occupe alors $N/8$ octets au lieu de N. La position de l'élément de rang i est déterminée par : le numéro de l'octet dans lequel il se trouve ; le numéro de bit dans l'octet. On obtient ces deux informations en prenant respectivement le quotient et le reste de la division entière de i par 8.

Cas particulier des tableaux de structures Soit le tableau T défini par :

TabStruct : le type tableau sur [0..N-1] de Structure21
T : un TabStruct

La représentation mémoire proposée ci-dessus pour T perd 2 octets pour chaque élément, c'est-à-dire $2 \times (N - 1)$. Si la taille mémoire est un critère important, on peut envisager une représentation mémoire tirée de la transformation suivante :

```
StructTab : le type <
            tx : un tableau sur [0..N-1] de T2 ;
            ty : un tableau sur [0..N-1] de T1 >
T : un StructTab
```

Il y a une correspondance évidente entre les objets de type TabStruct et ceux de type StructTab. Les éléments du champ tx, de taille 2, peuvent être placés côte à côte sans perte de place ; de même les éléments du champ ty. On perd éventuellement deux octets entre le tableau tx et le tableau ty, mais c'est très inférieur à $2 \times (N - 1)$.

Parcours de tableaux et optimisation Nous traitons ici un exemple classique qui permet de comprendre le codage optimisé des parcours de tableaux en langage d'assemblage, comme on l'observe dans la plupart des compilateurs. Considérons l'algorithme suivant :

```
Lexique
    N : l'entier ... ; i : un entier dans [0..N]
    T : un tableau sur [0..N−1] d'entiers dans [−2^(32−1), 2^(32−1) − 1]
algorithme
    i ⟵ 0
    tant que i < N
        T[i] ⟵ 2*i + 1
        i ⟵ i+1
```

La première transformation consiste à faire apparaître le tableau MEM qui modélise la mémoire, et l'installation des éléments de T en mémoire. On note a_T l'adresse de début de T en mémoire. On obtient :

```
lexique : E : l'entier taille_align(entier dans [−2^(32−1), 2^(32−1) − 1])
algorithme :
    i ⟵ 0
    tant que i < N
        MEM [a_T + E * i] ⟵ 2*i + 1
        i ⟵ i+1
```

La deuxième transformation consiste à ajouter une variable redondante Ad pour représenter l'adresse de l'élément courant en mémoire. Cette variable est liée à l'indice i du tableau par la propriété $\mathsf{Ad} = a_T + E * \mathsf{i}$ que l'on installe avant la boucle, et que l'on maintient en modifiant Ad lors de toute modification de i. On obtient :

```
i ⟵ 0 ; Ad ⟵ a_T + E * i
tant que i < N
    { Invariant : Ad =a_T + E * i }
    MEM [Ad] ⟵ 2*i + 1
    i ⟵ i+1 ; Ad ⟵ Ad + E
```

Remarque : La propriété qui lie Ad est i est un *invariant* de programme. Pour un exposé complet sur la notion d'invariant, voir par exemple [BB83].

Cette transformation, qui consiste à factoriser le calcul de l'adresse dans le tableau MEM et à éviter les multiplications, est une technique usuelle en compilation et optimisation des programmes (voir par exemple [CGV80]).

Tableaux à plusieurs dimensions Nous considérons ici le cas des tableaux à 2 dimensions. Le cas des tableaux à k dimensions s'en déduit avec quelques précautions (exercice E4.6).

Considérons le type Tab :

```
N, M : des entiers > 0
Tab : le type tableau sur [0..M-1, 0..N-1] de T'
```

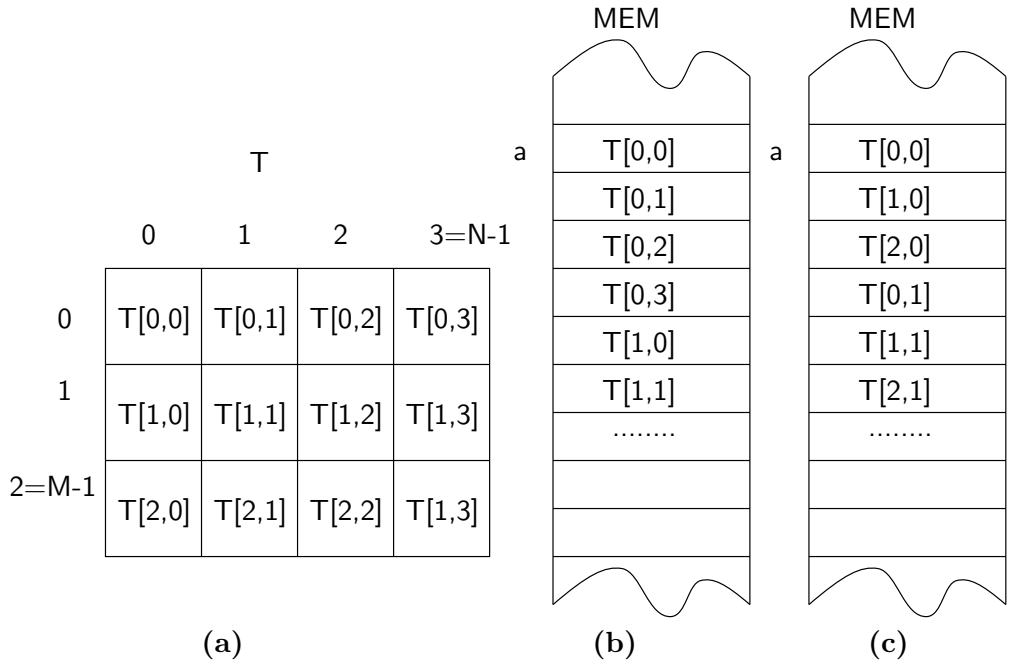


FIG. 4.1 – Représentation en mémoire des tableaux à deux dimensions

T : un Tab

La représentation de T en mémoire nécessite $N \times M \times$ taille_align(T') unités adressables.

La figure 4.1 illustre les choix de placement des éléments de T dans le tableau MEM, dans le cas où $N = 4$ et $M = 3$. Noter que la représentation de T sous forme de matrice **(a)**, et le choix de la dimension qu'on appelle *ligne* sont conventionnels ; nous convenons ici que dans l'expression T[i,j], i représente un numéro de *ligne* et j un numéro de *colonne*.

Dans le cas **(b)**, on range les éléments de T ligne par ligne, et l'adresse de l'élément T[i,j] s'exprime par la formule : $a + (i \times N + j) \times$ taille_align (T'), où a est l'adresse de début du tableau.

Dans le cas **(c)**, on range les éléments colonne par colonne, et l'adresse de l'élément T[i,j] s'exprime par la formule : $a + (j \times M + i) \times$ taille_align (T').

Noter la symétrie des deux formules.

> **Remarque :** Nous réservons le terme de tableau à deux dimensions aux structures implantées de manière contiguë. En Java, on appelle *tableau à deux dimensions* une structure de données plus compliquée qui consiste en un tableau à une dimension de pointeurs sur des tableaux à une dimension. Dans ce cas les *lignes* (ou *colonnes*) ne sont plus nécessairement contiguës.

2.4.3 Représentation en mémoire des pointeurs

NIL : un pointeur { *compatible avec tous les* pointeurs de T }
T : un type

adT : le type pointeur de T
t1 : un T ; pt : une adT
t1 ⟵ pt↑

La variable pt contient une valeur a qui est une adresse dans le tableau MEM. C'est donc un entier, d'une certaine taille majorée par la taille de la mémoire disponible de la machine.

Nous avons vu dans ce qui précède que, lorsque les valeurs des objets (structurés ou non) nécessitent plusieurs unités d'accès, celles-ci sont *contiguës*. Ainsi, pour repérer de manière non ambiguë une valeur en mémoire, il suffit de connaître : 1) l'adresse de la première unité d'accès où elle est stockée ; 2) le nombre d'unités d'accès utilisées, qui peut se déduire de son type.

Nous avons vu (paragraphe 1.2.2) que *pointeur de T* est le type des *adresses mémoire d'objets de type T*. Le type *pointeur de T* spécifie donc l'information de taille, nécessaire par exemple à la traduction des affectations comme t1 ⟵ pt↑.

On dit que pt *pointe* sur un objet qui occupe dans le tableau MEM, taille(T) unités adressables d'adresses $a + 0$, ... $a +$ taille(T) $- 1$.

La constante NIL est de type pointeur, compatible avec tous les types pointeur de T, quel que soit T. Elle représente le *pointeur sur rien*, et doit être codée par une valeur qui n'appartient pas à l'ensemble de valeurs que peuvent prendre les autres pointeurs. Avec la vision abstraite de la mémoire que nous avons adoptée jusque là, il suffit de choisir NIL : l'entier tmem, si MEM est défini sur l'intervalle [0..tmem-1]. Dans la réalité, la plupart des compilateurs choisissent de coder NIL par l'entier 0 qui est facile à tester (par convention 0 n'est alors pas une adresse valide).

3. Traduction des affectations générales en accès au tableau MEM

Considérons un type T et deux variables de type T nommées x et y, installées dans le tableau MEM à des adresses ax et ay. Dans la définition du langage d'actions utilisé, nous avons exigé que l'affectation porte sur des objets de *même type*. L'affectation se traduit donc toujours par une simple recopie du contenu d'une zone de mémoire vers une autre (pour les affectations des langages moins contraignants, qui cachent des *conversions*, nous verrons au chapitre 13, paragraphe 1.2, comment coder les fonctions de conversion introduites au paragraphe 1.4 ci-dessus). On s'intéresse ici à la traduction de l'action x ⟵ y en n'utilisant plus que les accès de taille fixée au tableau MEM décrits au paragraphe 2.3.

Lorsqu'une affectation porte sur des objets dont le type nécessite un grand nombre d'unités adressables, on ne peut pas la traduire par l'utilisation d'une affectation indicée par la taille, supposée être une opération de base dans les machines. Il faut alors traduire l'affectation par une boucle ou une séquence

d'affectations.

3.1 Affectation de structures

On peut envisager essentiellement deux méthodes : la méthode *structurelle*, dans laquelle on traduit une affectation de structures par la séquence des affectations champ par champ ; la méthode *aveugle*, dans laquelle on a oublié le type, et où l'on traduit une affectation de structures par le bon nombre d'accès au tableau MEM, de la taille la plus grande possible.

3.2 Affectation de tableaux

Considérons le programme suivant :

```
Elem : un type
T : le type tableau sur [a...b] de Elem
t1, t2 : des T ;
t1 ⟵ t2
    { est équivalent à : }
i parcourant a...b
    t1[i] ⟵ t2[i]
```

Si Elem est lui-même structuré, il faut continuer le raisonnement pour remplacer t1[i] ⟵ t2[i] par une séquence ou une boucle d'affectations plus élémentaires.

4. Utilisation des pointeurs et gestion dynamique de la mémoire

Quand on utilise des pointeurs, par exemple pour décrire la construction d'une séquence chaînée d'entiers dont le nombre d'éléments n'est connu qu'à l'exécution, la mémoire contient des données qui ne correspondent pas directement à des noms de variables définis par le programmeur. Ces données sont accessibles via des variables de type pointeur, dont les valeurs sont des adresses dans le tableau MEM.

Nous donnons figures 4.2 et 4.3 un exemple typique de construction d'une structure de données récursive.

Pour permettre la *création* et la *destruction* de cellules lors de la construction de la séquence, on utilise les actions Allouer et Libérer, qui se comportent comme des requêtes à un dispositif capable de distribuer de la mémoire : Allouer permet de réserver une zone de mémoire contiguë, en en précisant la taille ; Libérer déclare que la zone ne sera plus utilisée ; des requêtes d'allocation successives, sans libération, obtiennent des adresses de zones mémoire disjointes.

La manière la plus simple de voir les choses est de considérer que, dans un programme qui utilise des pointeurs, tout se passe comme si le programmeur

```
lexique (types et variables)
    Info : le type entier dans [0..255]
    Cellule : le type < x : une Info, suiv : un pCellule >
    pCellule : le type pointeur de Cellule
    inf, sup : des entiers ; LaSéquence : un pCellule

ConstruireSéquenceChaînée : une action
    (les données i,j : deux entiers ; le résultat p : un pCellule)
    { état final : p est l'adresse d'une première cellule qui contient l'information
    i. Celle cellule pointe sur une cellule qui contient l'information i+1. Ainsi
    de suite jusqu'à une cellule qui contient j. Cette dernière cellule ne pointe
    sur rien (suiv = NIL). }
lexique
    pcour, pprec, fictif : des pCellule ; k : un entier
algorithme
    Allouer (fictif, taille (pCellule))
    si fictif = NIL alors
        Ecrire ("Allocation impossible")
    sinon
        pprec ⟵ fictif ; fictif↑.suiv ⟵ NIL
        k ⟵ i ; pcour ⟵ fictif
        tantque pcour ≠ NIL et k ≤ j :
            Allouer (pcour, taille (pCellule))
            si pcour = NIL alors Ecrire ("Allocation impossible")
            sinon
                pcour↑.x ⟵ k ; pcour↑.suiv ⟵ NIL
            pprec↑.suiv ⟵ pcour
            pprec ⟵ pcour ; k ⟵ k+1
        p ⟵ fictif↑.suiv
        Libérer (fictif, taille (pCellule))

algorithme du programme principal :
    Ecrire ("Donnez deux entiers : ") ; Lire (inf, sup) ;
    ConstruireSéquenceChaînée (inf, sup, LaSéquence)
```

FIG. 4.2 – Algorithme de construction d'une séquence chaînée

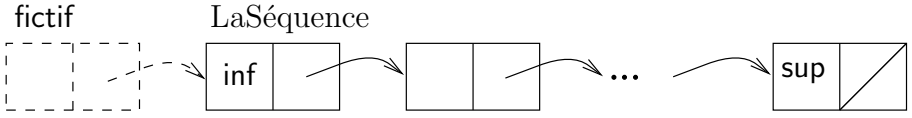


FIG. 4.3 – Une séquence chaînée

avait explicitement déclaré un grand tableau d'octets, et fourni des actions Allouer et Libérer capables de gérer l'occupation de ce tableau. C'est d'ailleurs le cas dans certaines applications où la gestion de la mémoire doit être optimisée.

Les environnements de programmation et les langages usuels offrent toutefois des actions Allouer et Libérer, que l'on peut utiliser si l'on ne se préoccupe pas particulièrement de l'efficacité des allocations. En C sous système UNIX, les fonctions `malloc` et `free` sont fournies dans une bibliothèque standard. Dans ce cas il n'est pas nécessaire que le programmeur déclare *explicitement* un tableau.

Pour comprendre exactement comment fonctionne ce dispositif d'allocation mémoire dite *dynamique*, il faut comprendre toutes les phases de traduction des langages de haut niveau en langage machine (Cf. Chapitres 12 et 13) ainsi que les étapes de la *vie d'un programme*, de l'écriture du texte jusqu'à l'installation du programme en langage machine dans la mémoire vive d'une machine, pour exécution par le processeur (Cf. Chapitres 18 et 20).

Toutefois, du point de vue du programmeur qui utilise des pointeurs comme dans l'exemple de la figure 4.2, tout se passe comme si une partie du tableau MEM était réservée pour les allocations et libérations de zones mémoire associées à des pointeurs. Ce n'est bien sûr qu'une partie de la mémoire. En effet, nous verrons dans la suite de cet ouvrage que, lors de l'exécution d'un programme utilisateur, la mémoire vive de la machine est occupée par de nombreuses informations autres que les objets du programme proprement dit. D'autre part, même si l'on ne considère que la mémoire nécessaire aux données du programme utilisateur, il faut distinguer deux zones nécessairement disjointes :

– une zone dans laquelle l'outil d'installation du programme en mémoire (le *chargeur*, Cf. Chapitre 20) place les variables du lexique global. Elles restent au même endroit pendant toute la durée de vie du programme, et elles sont toujours placées de la même manière les unes par rapport aux autres, d'une exécution à une autre. Nous verrons au chapitre 13 comment le compilateur prépare ce placement en mémoire vive, en précalculant les déplacements des différentes variables par rapport à une adresse de base qui ne sera connue que lors du chargement. Ce précalcul est qualifié d'*allocation statique*, parce qu'il est indépendant des exécutions ; seule l'adresse de base dépend de l'exécution.
– une zone dans laquelle les allocations et libérations à la demande du programme sont effectuées. Cette zone contient les zones de mémoires allouées, ainsi que les informations nécessaires à sa gestion : zones encore disponibles, zones occupées. Cette zone est appelée le *tas*.

4.1 Spécification des actions Allouer et Libérer

Les actions Allouer et Libérer peuvent donc être spécifiées et comprises en considérant qu'une partie de la mémoire est réservée à cet usage. Nous

considérons ici que le tableau MEM est partitionné en deux : une première portion $P1$, qui va de l'indice 0 à l'indice T, dans laquelle on trouve en particulier les variables du lexique global ; une deuxième portion $P2$ qui va de l'indice $T+1$ à l'indice du dernier élément tmem-1, dans laquelle on trouve les blocs alloués dynamiquement et les informations de gestion du tas.

Allouer : une action (le résultat : un pointeur ; la donnée : un entier > 0)
{ Allouer (p, n) *réserve dans la zone de mémoire comprise entre les indices* $T+1$ *et* tmem-1 *une zone contiguë de* n *éléments, démarrant sur une frontière multiple de* n. *p est l'adresse de la première unité adressable de cette zone réservée. Si l'espace disponible est déjà entièrement occupé, la valeur finale* p = NIL *exprime l'impossibilité d'allouer.*
C'est une action générique, qui convient pour tout type de pointeur. }

Libérer : une action (la donnée : un pointeur ; la donnée : un entier > 0)
{ Libérer (p, n) *restitue la zone de mémoire située entre les adresses* p *incluse et* p+n *exclue.* }

4.2 Réalisation des actions Allouer et Libérer

Les deux actions Allouer et Libérer gèrent la zone de mémoire $P2$ comprise entre les indices $T+1$ et tmem-1. Elles doivent tenir à jour un état de l'occupation des éléments de cette zone : lesquels sont libres, lesquels sont occupés, etc. Ces informations sur l'état de la zone de mémoire sont de nouvelles variables, qui peuvent être rangées dans la même zone.

L'algorithme de l'action Allouer paraît simple : il semble suffire de distribuer les portions de la zone de mémoire à gérer de manière séquentielle, dans l'ordre des demandes. Mais c'est raisonner sans tenir compte de l'action Libérer, qui peut créer des *trous*, réutilisables par des appels ultérieurs de l'action Allouer. L'algorithme se complique. Différentes politiques d'allocation de la mémoire apparaissent, selon que l'on préfère utiliser pour une nouvelle allocation : le *premier* trou de taille suffisante (dans un certain ordre d'exploration de la zone mémoire qui dépend de l'algorithme ; l'idée de prendre le premier accélère la recherche) ; le trou dont la taille est la plus proche de la taille demandée (provoque une tendance à l'émiettement) ; le trou dont la taille est la plus éloignée de la taille demandée...

Il existe une littérature prolifique sur les diverses manières de gérer ainsi une zone de mémoire où les demandes et restitutions se font dans un ordre quelconque. Le lecteur consultera par exemple [Kra85].

Le problème général de la gestion d'un espace mémoire pour l'installation dispersée de blocs est présent dans toutes les couches de l'architecture logicielle d'un ordinateur. Nous le reverrons au chapitre 19 à propos d'installation des fichiers sur un disque, puis au chapitre 20 à propos d'installation d'un programme en mémoire vive et de démarrage du système.

5. Piles, files et traitements associés

Les piles et les files sont des structures de données très utilisées dans tous les domaines de l'informatique. Nous précisons ci-dessous les opérations utilisées dans cet ouvrage. Dans certains chapitres nous serons amenés à préciser comment sont implantés les types Pile et File, et comment sont programmées les opérations de manipulation de ces types.

Dans une pile, les éléments sont extraits dans l'ordre inverse de leur ordre d'insertion (en anglais *last in, first out*, ou LIFO). Dans une file, les éléments sont extraits dans l'ordre de leur insertion (en anglais *first in, first out*, ou FIFO). Voir par exemple [BB88] pour une spécification formelle des structures de pile et de file et une étude de la programmation de ces structures (par des tableaux, des séquences chaînées, ...).

Nous considérons ici des piles et des files de taille éventuellement *bornée*, d'où la notion de pile (ou de file) *pleine*. Une pile ou une file peut également être *vide*. Ajouter un élément à une pile ou file n'est possible que si elle n'est pas pleine ; ôter un élément n'est possible que si elle n'est pas vide.

5.1 Spécification d'une pile

Elem : un type
PileElem : un type { *sans préjuger de la représentation des piles par des structures de données particulières* }
TailleMax : un entier > 0

Initialiser : une action (le résultat P : une PileElem)
{ *état final :* P *est la pile vide* }

Empiler : une action (la donnée-résultat P : une PileElem ; la donnée x : un Elem ; le résultat ok : un booléen)
{ *état initial : Notons k le nombre d'éléments présents dans la pile ; si la pile est vide :* $k = 0$ *; si la pile est pleine :* $k = TailleMax$ *. Notons* $P=\alpha_1...\alpha_k$ *le contenu de la pile.*
état final : Si $k = TailleMax$, $P = \alpha_1...\alpha_k$ *et* ok=faux *sinon,* ok=vrai *et* $P = \alpha_1...\alpha_k\ x$ }

Dépiler : une action (la donnée-résultat P : une PileElem ; le résultat x : un Elem ; le résultat ok : un booléen)
{ *état initial : Notons k le nombre d'éléments et* $P=\alpha_1...\alpha_k$ *le contenu de la pile, lorsque* $k \neq 0$.
état final : si $k=0$, *alors* ok=faux *et* x *est non spécifié sinon* ok=vrai, $x=\alpha_k$ *et* $P = \alpha_1...\alpha_{k-1}$ }

Lorsque la pile *est de taille suffisante* pour l'utilisation qui en est faite, ou lorsque qu'on ne veut pas s'intéresser au problème du débordement (c'est-à-dire une tentative d'insertion lorsque la pile est pleine), on utilisera une action

Empiler sans paramètre résultat booléen. Dans ce cas, l'état final d'une pile qui était pleine lors de l'empilement d'un élément, est non spécifié.

De même, si l'on ne s'intéresse pas au problème d'accès à la pile vide, ou si l'on sait que l'action Dépiler n'est jamais appelée avec une pile vide, on peut utiliser une action Dépiler sans paramètre résultat booléen.

5.2 Spécification d'une file

Elem : un type
FileElem : un type
TailleMax : un entier > 0
Initialiser : une action (le résultat F : une FileElem)
{ *état final :* F *est la file vide* }

Entrer : une action (la donnée-résultat F : une FileElem ; la donnée x : un Elem ;
le résultat ok : un booléen)
{ *état initial : Notons* $F=\alpha_1...\alpha_k$ *le contenu de la file ; si la file est vide :* $k = 0$ *; si la file est pleine :* $k = TailleMax$
état final : Si $k = TailleMax$, $F = \alpha_1...\alpha_k$ *et* ok=faux *sinon,* ok=vrai *et* $F = \alpha_1...\alpha_k\ x$ }

Sortir : une action (la donnée-résultat F : une FileElem ; le résultat x : un Elem ;
le résultat ok : un booléen)
{ *état initial : Notons* $F=\alpha_1...\alpha_k$ *le contenu de la file.*
état final : si k=0, alors ok=faux *et* x *est non spécifié sinon* ok=vrai, x=α_1 *et* $F = \alpha_2...\alpha_k$ }

Sous les mêmes hypothèses que pour la pile, on s'autorise les actions Entrer et Sortir sans paramètres résultats booléens.

6. Exercices

E4.1 : Codage des entiers : petit bout ou gros bout
Considérons deux chaînes de caractères dont on veut réaliser la comparaison lexicographique (autrement dit déterminer laquelle vient en premier dans l'ordre alphabétique). Ces chaînes sont représentées en mémoire de manière contiguë, chaque caractère occupe un octet et il n'y a pas de place perdue. Pour accélérer la comparaison, on utilise des opérations de comparaison d'entiers codés en binaire pur sur 32 bits, c'est-à-dire qu'on compare les caractères 4 par 4. Le choix de représentation en mémoire des entiers (petit bout ou gros bout, Cf. Paragraphe 2.2.4) a-t-il une influence sur la correction du résultat ?

E4.2 : Représentation mémoire des ensembles et codage des

opérations ensemblistes
Les vecteurs booléens peuvent représenter des ensembles, ou, plus exactement, un vecteur booléen de N bits peut représenter une partie d'un ensemble à N éléments : le bit de rang x est à 1 si et seulement si l'élément x appartient à l'ensemble. (Cf. Paragraphe 4. du chapitre 3). On considère les types :

Elem : le type entier dans $[0..31]$; EnsElem : le type ensemble d'Elems
E1, E2 : des EnsElem

Proposer une représentation mémoire des objets de type EnsElem. Combien d'octets sont-ils nécessaires ? Exprimer en termes d'opérations booléennes (et, ou, non, ...) sur la représentation mémoire de deux ensembles E1 et E2, les opérations suivantes :

E1 ∪ E2 ; E1 ∩ E2 ; E1 \ E2
E1 ⟵ E1 ∪ { x } { *avec* x *de type* Elem }
E1 ⟵ E1 \ { x } { *avec* x *de type* Elem }
x *in* E1 { *avec* x *de type* Elem }

E4.3 : Transformation des conditions boolénnes composées
Proposer une transformation de si C1 et (C2 ou non C3) alors A1 sinon A2 qui n'utilise plus d'opérateurs booléen et, ou, non.

E4.4 : Parcours de tableaux de structures
On considère l'algorithme suivant :

```
lexique
    Entier32s : le type entier sur [−2^31, 2^31 − 1]
    T : un tableau sur [0 .. N-1] de < a : un Entier32s, b : un caractère >
algorithme
    i parcourant 0 .. N-1
        T[i].a ⟵ i * 2 ; T[i].b ⟵ 'a'
```

Réécrire cet algorithme en faisant apparaître le tableau MEM et l'installation des éléments de T dans MEM, à partir d'une adresse γ.

E4.5 : Choix de représentation d'un tableau à deux dimensions
On considère trois tableaux d'entiers non signés, de dimension 2, carrés, nommés T, S et U, définis sur [0..N-1] x [0..N-1].

On veut remplir U d'après la formule : U [i,j] = T[i, j] + 2^{32} × S [j, i]. Si les tableaux T et S ont des éléments de 32 bits, U a donc des éléments de 64 bits.

Choisir une représentation en mémoire des trois tableaux qui facilite le parcours de remplissage selon la formule ci-dessus.

E4.6 : Représentation en mémoire d'un tableau à k dimensions
On considère le type suivant :

Tab : le type tableau sur [0..N_0, 0..N_1, ..., 0..N_{k-1}] d'entiers sur $[-2^{8-1}, 2^{8-1} - 1]$.
T : un Tab

Choisir une représentation en mémoire des objets de type Tab et donner la formule qui exprime l'adresse de début de l'élément T[i_0, i_1, ..., i_{k-1}] en fonction de l'adresse de début de T et des dimensions N_0, N_1, ..., N_{k-1}.

E4.7 : Transformation d'algorithme d'accès à un tableau de structures
Reprendre le développement du paragraphe sur l'optimisation des parcours de tableaux (Cf. Paragraphe 2.4.2) dans le cas où un tableau de structures est représenté en mémoire par une structure de tableaux.

E4.8 : Parcours de matrice carrée et comparaison double longueur en complément à deux

Considérons une constante entière positive N (pas trop grande) et une matrice carrée à N lignes et N colonnes :

N : un entier > 0
Matrice : un tableau sur [0..N-1, 0..N-1] d'entiers

On désire vérifier si la propriété suivante est vraie :

Pour tout i dans [1, N-1],
  Pour tout j dans [0, i-1]
    $M_{ij} < M_{ji}$

Le but de l'exercice est d'écrire un programme pour parcourir la matrice et déterminer si la propriété est vérifiée. On ne demande pas de programmer l'acquisition des éléments de la matrice.

Questions :

Q1 Choisir une valeur pour la constante N (non triviale, c'est-à-dire différente de 0, 1, 2, mais de nature à faciliter la programmation de l'algorithme de parcours. Songer en particulier à éviter les multiplications générales).

Q2 Proposer une représentation mémoire du tableau, en supposant que les éléments de la matrice sont des entiers relatifs codés en complément à deux sur 64 bits.

Q3 Donner l'algorithme demandé en notation algorithmique, en faisant apparaître le tableau MEM et le calcul des adresses des éléments.

Cet exercice se poursuit par la programmation en assembleur SPARC, exercice E13.10 du chapitre 12.

E4.9 : Programmation d'une file et d'une pile

Réaliser les actions de manipulation des piles et files décrites au paragraphe 5. :

– En rangeant les éléments dans un tableau, c'est-à-dire en considérant le type : PileElem : un tableau sur 1 .. TailleMax d'Elem

– En rangeant les éléments dans une séquence chaînée

Etudier les alternatives : pour le tableau, progression de la pile par adresses croissantes ou décroissantes et *pointeur de pile* indiquant la première case vide ou la dernière case pleine ; pour la séquence chaînée, insertion en début ou en fin.

Chapitre 5

Représentation des traitements et des données : machines séquentielles

Nous présentons ici le modèle mathématique des machines séquentielles de Moore et de Mealy. Ces modèles peuvent être utilisés pour représenter les traitements, aussi bien dans un contexte matériel que dans un contexte logiciel (où elles rejoignent la représentation classique par organigrammes).

Dans toute la suite de l'ouvrage, on utilisera indifféremment les termes de *machine séquentielle*, *machine à états finie*, *automate d'états fini*, *automate*.

> *Nous définissons les machines séquentielles simples au paragraphe 1., puis les machines séquentielles* avec actions *au paragraphe 2. Pour le logiciel, nous montrons comment traduire le langage d'actions simple en machines séquentielles avec actions au paragraphe 2.2. Pour le matériel, l'utilisation des machines séquentielles apparaît aux chapitres 10 et 11.*

1. Machines séquentielles simples

1.1 Définitions mathématiques et propriétés

Définition 5.1 : machine de Moore, machine de Mealy
Une machine de Moore est un sextuplet (Q, q_0, E, S, T, f) où :

- Q est l'ensemble des états ; $q_0 \in Q$ est l'état initial
- E (resp. S) est l'alphabet (ou vocabulaire) d'entrée (resp. de sortie)
- $T \subseteq Q \times E \times Q$ est l'ensemble des transitions ; on note (q, e, q') une transition de q à q' et on dit que l'élément e de l'alphabet des entrées est l'*étiquette* de la transition.
- $f : Q \longrightarrow S$ est la fonction qui fait correspondre un élément de l'alphabet de sortie à chaque état.

Une machine de Mealy est un quintuplet (Q, q_0, E, S, T) où :

- Q est l'ensemble des états ; $q_0 \in Q$ est l'état initial
- E (resp. S) est l'alphabet d'entrée (resp. de sortie)
- $T \subseteq Q \times E \times S \times Q$ est l'ensemble des transitions, étiquetées par des couples constitués d'un élément de l'alphabet des entrées et d'un élément de l'alphabet des sorties. □

La figure 5.1 illustre la représentation conventionnelle des automates : un cercle pour un état, une flèche étiquetée pour une transition.

1.1.1 Fonctionnement séquentiel

Le fonctionnement séquentiel des machines de Moore ou de Mealy est défini en observant quelle *séquence* de sorties est produite par la machine, lorsqu'elle réagit à une séquence d'entrées donnée.

Considérons donc une séquence d'entrées : c'est une suite d'éléments de l'alphabet d'entrées, c'est-à-dire une fonction de $\mathbb{N}$ dans E, dont les éléments seront notés de manière indicée. On notera $S_e = e_0, e_1, ..., e_n, ...$

Pour définir la réaction de la machine de Moore (Q, q_0, E, S, T, f) à la séquence d'entrées S_e, on définit la séquence $q_0, q_1, ...$ des états rencontrés :

$$\forall n \geq 0, (q_n, e_n, q_{n+1}) \in T$$

Une transition (q, e, q') exprime que, si la machine est dans l'état q, et qu'elle reçoit l'entrée e, alors elle passe dans l'état q'. La séquence de sorties $S_s = s_0, s_1, ...$ est ensuite définie par l'intermédiaire de la séquence d'états :

$$\forall n \in N, s_n = f(q_n)$$

Pour définir la réaction de la machine de Mealy (Q, q_0, E, S, T) à la séquence d'entrées S_e, on écrit directement :

$$q_0 = q_0 \qquad \forall n \geq 0, (q_n, e_n, s_n, q_{n+1}) \in T$$

1.1.2 Déterminisme et réactivité

On s'intéresse aux propriétés de *déterminisme* et *réactivité* des machines séquentielles de Moore ou de Mealy, qui sont indispensables si l'on utilise les machines comme modèle de traitements, c'est-à-dire comme des programmes (Cf. Paragraphes 1.3 et 2.). On trouvera parfois dans la littérature le terme d'automate *complet*, au lieu de *réactif* (voir par exemple [Ben91]). Intuitivement, une machine est déterministe (resp. réactive) si et seulement si, quel que soit son état, et quelle que soit la configuration de ses entrées, elle peut exécuter au plus une (resp. au moins une) transition. Une machine à la fois déterministe et réactive peut donc exécuter exactement une transition, pour chaque état et chaque entrée.

Définition 5.2 : déterminisme
On dira qu'une machine de Mealy (Q, q_0, E, S, T) est *déterministe* si et seulement si :

$$\forall q \in Q, \left.\begin{array}{l} \exists q_1 \in Q, e_1 \in E, s_1 \in S, (q, e_1, s_1, q_1) \in T \\ \wedge \\ \exists q_2 \in Q, e_2 \in E, s_2 \in S, (q, e_2, s_2, q_2) \in T \end{array}\right\} \Longrightarrow e_1 \neq e_2$$

De même, on dira qu'une machine de Moore (Q, q_0, E, S, T, f) est *déterministe* si et seulement si :

$$\forall q \in Q, \left.\begin{array}{l} \exists q_1 \in Q, e_1 \in E (q, e_1, q_1) \in T \\ \wedge \\ \exists q_2 \in Q, e_2 \in E (q, e_2, q_2) \in T \end{array}\right\} \Longrightarrow e_1 \neq e_2$$

□

Définition 5.3 : réactivité
Une machine de Mealy (Q, q_0, E, S, T) est dite *réactive* si et seulement si :

$$\forall q \in Q, \{e \in E \mid \exists q_1 \in Q, s \in S, (q, e, s, q_1) \in T\} = E$$

De même, une machine de Moore (Q, q_0, E, S, T, f) est dire réactive si et seulement si :

$$\forall q \in Q, \{e \in E \mid \exists q_1 \in Q, (q, e, q_1) \in T\} = E$$

□

Notons que lorsque la machine est déterministe, il existe une unique séquence de sorties correspondant à une séquence d'entrées. Lorsque la machine est réactive, la séquence de sorties est aussi longue que la séquence d'entrées.

1.1.3 Fonctions de transition et de sortie

Pour des machines déterministes, la relation de transition $T \subseteq Q \times E \times Q$ (Moore) ou $T \subseteq Q \times E \times S \times Q$ (Mealy) est souvent exprimée comme une *fonction*.

On définit ainsi la *fonction de transition* $g : Q \times E \longrightarrow Q$ pour les machines de Moore ; g associe à chaque couple (état, entrée) l'état de destination ; si la machine est réactive, cette fonction est totale. De la même manière, on définit pour les machines de Mealy une fonction de transition $g : Q \times E \longrightarrow Q \times S$ qui associe à chaque couple (état, entrée) l'état de destination et la sortie émise par la transition. On trouve parfois également une définition en deux fonctions, dites *de transition* et *de sortie* : $g : Q \times E \longrightarrow Q$ et $s : Q \times E \longrightarrow S$.

1.1.4 Equivalence des modèles de Moore et de Mealy

Pour toute machine M de Mealy (resp. de Moore), il existe et on peut construire une machine M' de Moore (resp. de Mealy) telle que M et M' produisent la même séquence de sorties pour une séquence d'entrées donnée. Nous donnons ici seulement l'intuition de la transformation, pour montrer que les deux modèles sont équivalents.

Pour transformer une machine de Moore en machine de Mealy, il suffit de déplacer les sorties des états sur les transitions qui y mènent. Pour transformer une machine de Mealy en machine de Moore, il suffit de déplacer les sorties associées à une transition vers l'état but de la transition. Si plusieurs transitions, portant des sorties différentes, mènent au même état, celui-ci doit être éclaté en autant d'états distincts.

Dans la suite de cet ouvrage, nous utiliserons l'un ou l'autre des modèles de Moore ou de Mealy, mais sans avoir besoin de transformer l'un en l'autre.

1.2 Application à la reconnaissance des langages réguliers

L'une des caractérisations de la classe des *langages réguliers* (on dit aussi langage *rationnel*) énonce que ces langages sont exactement les langages reconnaissables par des machines à états finies (Cf. par exemple [Ben91]).

Les reconnaisseurs de langages réguliers sont des machines de Moore qui produisent une unique sortie booléenne. Dans un état E, cette sortie est vrai si et seulement si les séquences d'entrées qui permettent d'atteindre E depuis l'état initial constituent des phrases correctes du langage à reconnaître. L'usage a consacré une notation particulière de ces machines de Moore, dans laquelle on omet la notation de la sortie : il suffit de distinguer, par exemple par des triangles, les états pour lesquels elle vaut vrai. Dans la littérature on trouvera souvent le terme d'état *final*, ou *de satisfaction*. Notons que, si l'état initial est également final, la phrase vide appartient au langage.

Les machines de Moore qui expriment la reconnaissance de langages réguliers ne sont pas nécessairement *réactives* : à partir d'un état donné, il peut ne pas exister de transition exécutable, pour un élément particulier de la séquence, et la machine peut donc se bloquer. Dans ce cas toutefois, la séquence d'entrées ne permettra jamais d'atteindre un état de satisfaction. On interprète donc les blocages de la machine comme un résultat négatif.

Elles ne sont pas non plus nécessairement *déterministes*; mais pour tout langage régulier il existe une machine séquentielle déterministe qui le reconnaît. Il existe même un algorithme de transformation d'un reconnaisseur non déterministe en reconnaisseur déterministe du même langage.

Il existe une infinité de machines de Moore à états finals pour reconnaître un langage régulier donné. Il en existe toujours une à un nombre minimal d'états.

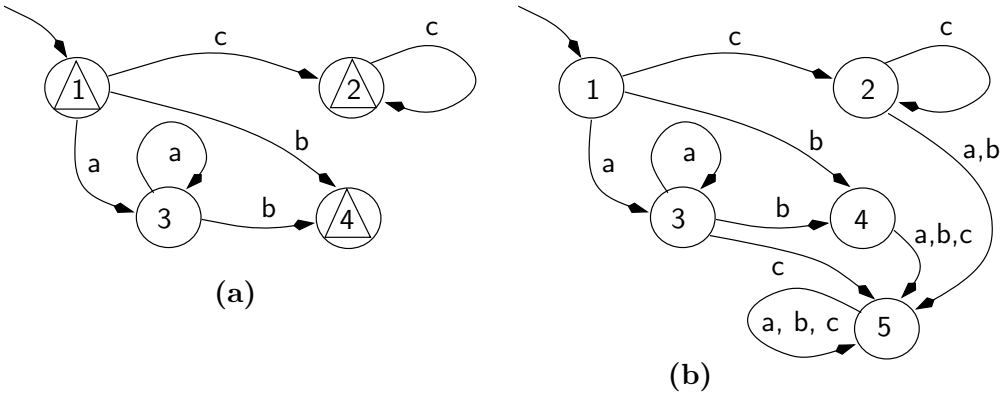


FIG. 5.1 – Reconnaissance du langage régulier a*b + c*. **(a)** Machine de Moore à états finals, avec : $Q = \{1, 2, 3, 4\}$, $E = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$, $f(1) = f(2) = f(4) =$ vrai, $f(3) =$ faux,
$T = \{(1, \mathsf{a}, 3), (1, \mathsf{b}, 4), (1, \mathsf{c}, 2), (2, \mathsf{c}, 2), (3, \mathsf{a}, 3), (3, \mathsf{b}, 4)\}$.
(b) Machine de Moore ordinaire.

Exemple E5.1 : Automate reconnaisseur du langage a*b + c*
La figure 5.1 donne une machine de Moore qui reconnaît le langage décrit par l'expression régulière a*b + c*. *L'automate donné est minimal. L'état 1 est initial. Les états 1, 2 et 4 sont finals. L'état final 2 correspond aux phrases constituées uniquement de lettres c (au moins une) ; l'état final 4 correspond à la phrase réduite à la lettre b et aux phrases de la forme* aa*b *(un nombre non nul de lettres a, puis une lettre b). Notons que dans les phrases* ca, bb *ou encore* ac, *la première lettre permet d'exécuter une transition issue de l'état initial, ensuite de quoi l'automate est bloqué. Aucune de ces phrases n'appartient au langage considéré.*

1.3 Application à la description de systèmes réactifs

Nous détaillons dans ce paragraphe un exemple de système réactif : une machine à café. Cet exemple est repris au chapitre 10 où nous montrons comment réaliser le contrôleur de la machine avec un circuit séquentiel. On donne d'autres exemples de systèmes réactifs dans le paragraphe 2.1.1 et l'exercice E10.6 du chapitre 10.

Exemple E5.2 : Machine à café
On considère une machine automatique de distribution de café, qui accepte des pièces de 1, 2 et 5 francs. Un café coûte 2 francs. Dès que le consommateur a introduit 2 francs ou plus, la machine n'accepte plus de pièces jusqu'à ce que le café soit servi. D'autre part, s'il avait introduit plus de 2 francs, la machine rend la monnaie.

On considère que la machine à café est constituée d'une partie physique et du dispositif informatique que nous appelons *contrôleur*. L'environnement du contrôleur est constitué de l'utilisateur humain et de la partie physique de la machine. Les entrées du contrôleur en provenance de l'utilisateur humain se réduisent à l'introduction de pièces (dans un exemple plus général on envisagerait le choix de la boisson). Les entrées en provenance de la partie physique de la machine sont des comptes-rendus d'activité (voir plus loin). Les sorties à destination de la partie physique de la machine sont les commandes de service du café, de fermeture de l'orifice d'introduction des pièces, de rendu de monnaie (on supposera qu'il existe un dispositif capable de calculer la somme à rendre, non décrit ici). On n'envisage pas de sorties à destination de l'utilisateur.

Nous nous intéressons ici à l'algorithme du *contrôleur* de cette machine. Le contrôleur est un exemple typique de système dit *réactif* : il interagit en permanence avec son environnement, et réagit à des *entrées* par l'émission de *sorties* appropriées. On peut le décrire par une machine séquentielle réactive, de Moore ou de Mealy. Notons que le critère mathématique de *réactivité* de la machine séquentielle correspond exactement à la nature *réactive* du système de contrôle de la machine à café : la réaction du contrôleur doit être parfaitement définie, dans chacun de ses états, pour chacune des entrées possibles.

L'algorithme à écrire analyse une séquence d'entrées et produit une séquence de sorties correspondante.

Interface d'entrée/sortie du contrôleur : Pour déterminer le vocabulaire d'entrée de la machine séquentielle décrivant le contrôleur, il convient de faire quelques hypothèses sur son environnement. On pourra considérer que les actions de l'utilisateur et le compte-rendu de la machine ne sont jamais simultanés. D'autre part des contraintes physiques comme la taille de l'orifice dans lequel on introduit les pièces empêchent sans doute d'introduire deux pièces en même temps. Les seules entrées à considérer sont donc :

- s_1, s_2, s_5 signifient respectivement que l'utilisateur a introduit une pièce de 1, 2 ou 5 francs.
- f_s est un compte-rendu d'activité de la machine : lorsqu'elle reçoit la commande de service de café, elle répond par cet acquittement de fin de service, après *un certain temps*.
- rien signifie que rien n'arrive : ni introduction de pièces, ni compte-rendu de la machine.

Le vocabulaire de sortie est $\mathcal{P}(\{R, C, B, \mathsf{AUCUNE}\})$ où R signifie : calculer et **R**endre la monnaie ; C signifie servir le **C**afé ; B signifie **B**loquage de l'orifice d'introduction des pièces ; AUCUNE signifie pas de sortie. Toutefois les seuls sous-ensembles effectivement utilisés dans la machine séquentielle qui décrit le contrôleur sont : $\{\mathsf{AUCUNE}\}$, $\{C, B\}$ et $\{R, C, B\}$.

> *Nous verrons au chapitre 10 que l'identification exacte du sous-ensemble effectivement utile du vocabulaire de sortie peut être utilisé pour proposer un codage efficace des sorties d'une machine séquentielle, lorsqu'elle est implantée par un circuit séquentiel.*

Description du comportement du contrôleur : Le comportement du contrôleur de machine à café peut être décrit par la machine de Moore de la figure 5.2 (le modèle de Moore est ici le plus approprié car la valeur des sorties est intrinsèquement définie par l'état, et ne dépend pas de l'entrée).

Cette description appelle un certain nombre de remarques. 0n suppose que l'environnement de ce contrôleur (c'est-à-dire l'ensemble formé par l'utilisateur humain et par la machine) a un comportement *correct*, c'est-à-dire que certaines successions d'entrées et de sorties du contrôleur peuvent être considérées comme impossibles : 1) Tant que l'introduction des pièces est bloquée par la machine, s_1, s_2 et s_5 ne peuvent pas survenir ; 2) Lorsque l'utilisateur humain a commandé le service du café, le compte-rendu f_s surviendra nécessairement, après un certain temps ; 3) Le compte-rendu f_s ne peut pas survenir si l'on n'a pas commandé le service du café.

Ces contraintes permettent de vérifier que les formules booléennes qui conditionnent les transitions issues d'un même état assurent bien les propriétés de déterminisme et réactivité de la machine. Par exemple, dans l'état Attente Pièces, les seules conditions envisagées sont s_1, s_2, s_5 et rien. rien correspond à la condition booléenne : $\overline{s_1}.\overline{s_2}.\overline{s_5}$. L'entrée f_s n'est pas mentionnée. En revanche, dans l'état 2F reçus s_1, s_2 et s_5 ne peuvent pas se produire et rien signifie $\overline{f_s}$.

Nous donnons figure 5.3 une séquence de monômes d'entrée et la séquence de monômes de sorties correspondante.

1.4 Codage algorithmique d'une machine séquentielle, application aux reconnaisseurs de langages réguliers

Lorsqu'un problème est décrit sous forme de machine séquentielle, il est possible de produire systématiquement un algorithme itératif dont le comportement est le comportement séquentiel de la machine.

Par exemple, l'algorithme de reconnaissance d'un langage régulier est un parcours de séquence qui calcule un booléen Appartenance. Lorsque le parcours s'arrête, ce booléen a la valeur vrai si et seulement si la séquence parcourue constitue une phrase correcte du langage considéré (c'est-à-dire si l'automate reconnaisseur s'arrête dans un état de satisfaction).

On suppose que la séquence des entrées de la machine séquentielle est accessible grâce aux primitives Démarrer, Avancer, FinDeSéq et CarCour qui permettent d'abstraire les algorithmes de traitement séquentiel (Cf. [SFLM93]). Nous construisons l'algorithme itératif par un codage systématique de la machine séquentielle de Moore qui définit le reconnaisseur. La consomma-

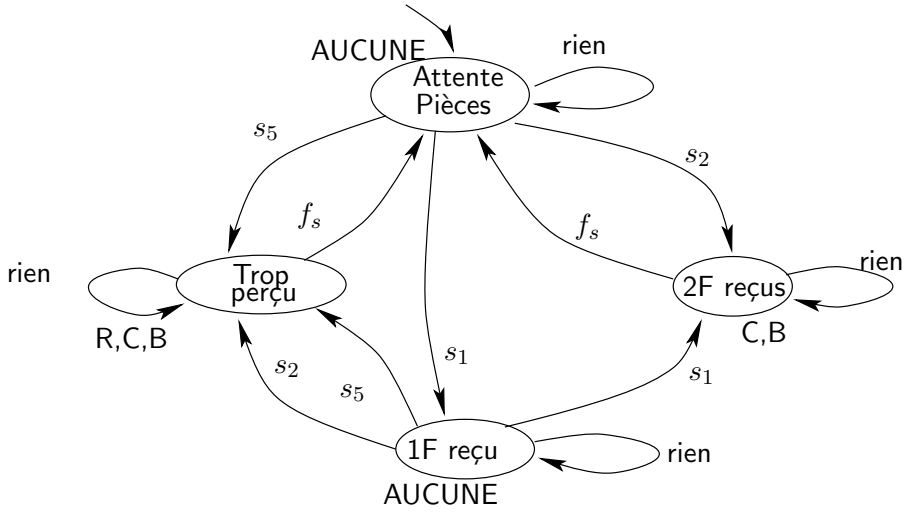


FIG. 5.2 – Comportement du contrôleur d'une machine à café (machine de Moore)

.

Entrée	Sortie	Etat courant
rien	{AUCUNE}	Attente Pièces
rien	{AUCUNE}	Attente Pièces
rien	{AUCUNE}	Attente Pièces
s_2	{AUCUNE}	Attente Pièces
rien	$\{C, B\}$	2F reçus
rien	$\{C, B\}$	2F reçus
rien	$\{C, B\}$	2F reçus
f_s	$\{C, B\}$	2F reçus
s_1	{AUCUNE}	Attente Pièces
rien	{AUCUNE}	1F reçu
s_2	{AUCUNE}	1F reçu
rien	$\{R, C, B\}$	Trop perçu
f_s	$\{R, C, B\}$	Trop perçu
...	{AUCUNE}	Attente Pièces

FIG. 5.3 – Une séquence d'exécution du contrôleur de la machine à café : chaque ligne correspond à un instant différent ; le temps passe du haut vers le bas dans le tableau.

```
Etat : le type (Un, Deux, Trois, Quatre, Erreur)
E : un Etat ; Appartenance : un booléen
E ⟵ Un ; Démarrer
tant que non FinDeSéq
   selon E
      E = Un :
         selon CarCour :
            CarCour = 'c' : E ⟵ Deux
            CarCour = 'b' : E ⟵ Quatre
            CarCour = 'a' : E ⟵ Trois
      E = Deux :
         selon CarCour :
            CarCour = 'c' : E ⟵ Deux
            CarCour = 'b' ou CarCour = 'a' : E ⟵ Erreur
      E = Trois :
         selon CarCour :
            CarCour = 'a' : E ⟵ Trois
            CarCour = 'b' : E ⟵ Quatre
            CarCour = 'c' : E ⟵ Erreur
      E = Quatre : E ⟵ Erreur
      E = Erreur : { rien à faire }
   Appartenance ⟵ (E = Un ou E = Deux ou E = Quatre)
   { Invariant : Appartenance est vrai ssi la séquence de caractères lue jusque
   là est une phrase du langage décrit par l'expression régulière a*b + c* }
   Avancer
```

FIG. 5.4 – Algorithme de reconnaissance du langage a*b + c* basé sur l'automate de la figure 5.1-b.

tion des éléments de la séquence est réalisée par un appel de la primitive Avancer. Chaque passage dans la boucle consomme exactement un élément de la séquence et représente l'exécution d'une transition de la machine. Les conditions sur l'entrée sont traduites en conditions sur l'élément courant de la séquence, accessible par la fonction Carcour. La sortie Appartenance est calculée en fin de boucle, en fonction de l'état atteint.

On suppose que la séquence d'entrée ne comporte que les caractères a, b et c. L'algorithme de reconnaissance du langage a*b + c* est donné figure 5.4.

2. Machines séquentielles avec actions

Dans le langage des actions présenté au paragraphe 1. du chapitre 4, la structuration des algorithmes est assurée par un petit nombre de constructions itératives (tant que, parcourant) ou conditionnelles. Parmi les actions

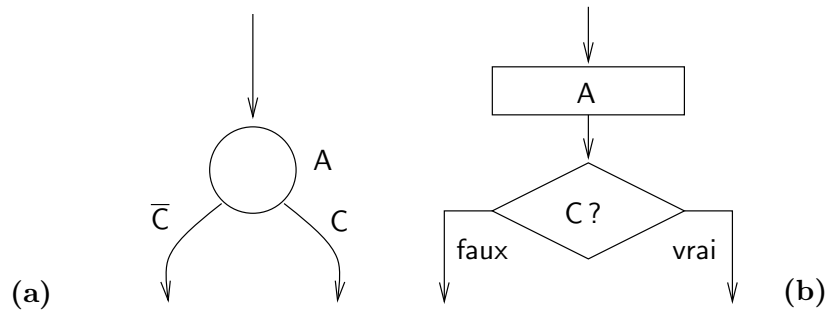


FIG. 5.5 – Machine de Moore avec actions et organigramme. **(a)** : un état de machine de Moore avec actions (C est une condition booléenne et A une action) ; **(b)** : une portion d'organigramme qui représente le même traitement.

élémentaires on trouve en particulier l'affectation.

L'idée du modèle des *machines séquentielles avec actions* — on trouve parfois dans la littérature le terme d'*automate interprété* ou de *schéma de programme avec interprétation* (Cf. par exemple [Liv78]) — est d'exprimer les structures conditionnelles et itératives d'un algorithme par les états et transitions d'une machine séquentielle. Les actions sont les sorties de la machine et constituent donc les étiquettes des transitions ou des états, selon que l'on utilise le modèle de Mealy ou le modèle de Moore. Des conditions booléennes constituent les entrées de la machine séquentielle.

2.1 Définition

On se donne un lexique (au sens défini chapitre 4) qui définit des types, des variables typées, des fonctions et des actions sans paramètres. Parmi les fonctions on distingue les *prédicats*, qui sont à résultat booléen. Le prédicat constant vrai et l'action vide vide sont toujours définis, et jouent un rôle particulier dans les manipulations de machines séquentielles à actions (Cf. Paragraphe 2.4).

Une *machine séquentielle avec actions* est une machine à états finie dont le vocabulaire d'entrée est l'ensemble des prédicats : l'évaluation d'un prédicat représente une *entrée* de la machine, au sens du paragraphe 1.1. Les transitions sont donc étiquetées par des prédicats. L'ensemble des actions constitue le vocabulaire de sortie.

Une machine de Moore avec actions est très similaire aux organigrammes classiques, ainsi que le montre la figure 5.5.

Les machines de Mealy avec actions sont étudiées dans [SFLM93]. Elles sont une extension naturelle des algorithmes obtenus comme codage systématique des machines de reconnaissance des langages réguliers (paragraphe 1.4). Nous ne les étudierons pas ici.

2.2 Représentation des structures de contrôle par des machines séquentielles avec actions

Dans le chapitre 4 nous avons défini un petit langage d'actions, et étudié la première étape de traduction des structures de données, c'est-à-dire la représentation des données complexes en mémoire. Nous obtenons donc des programmes sans structures de données, dans lesquels ne subsistent que des accès de taille 1, 2 ou 4 au tableau MEM.

Nous nous intéressons ici au codage des structures de contrôle, sauf l'appel d'action ou fonction paramétré, qui sera étudié de façon détaillée au chapitre 13.

La figure 5.6 donne la traduction des structures de contrôle usuelles en machines séquentielles avec actions. Chaque machine obtenue pour la traduction d'une structure de contrôle possède un état initial et un état final. Pour composer de telles machines, il suffit de définir comment remplacer une action A par une machine. Pour cela on remplace l'état q qui porte l'action A par le dessin complet de la machine qui représente l'algorithme de A. Les transitions issues de q deviennent issues de l'état final de la machine de A ; les transitions qui arrivent à q sont branchées sur l'état initial de la machine de A. A titre d'exemple nous donnons la machine de l'algorithme :

```
tant que C faire
   A
   tant que D faire
      B
   E
```

2.3 Définition du lexique d'une machine séquentielle avec actions

Dans ce paragraphe nous montrons comment produire une machine séquentielle avec actions à partir d'un algorithme itératif. Nous illustrons cette transformation pour l'algorithme de Bresenham, qui permet de calculer les coordonnées des points d'un segment dans un plan quadrillé. Cet exemple est repris dans le chapitre 11 où nous montrons comment obtenir un circuit à partir de cet algorithme. L'exercice E13.5 du chapitre 12 propose de programmer cet algorithme en langage d'assemblage SPARC.

2.3.1 Traceur de segments : algorithme de Bresenham

Le but de l'algorithme de Bresenham est de placer dans le plan des points de coordonnées entières qui approchent le mieux possible une droite d'équation donnée.

Le segment qui passe par les points de coordonnées $(0, 0)$ et (m, n) est supportée par la droite d'équation $y = (n/m)x$ si $m \neq 0$. Il s'agit donc de tracer

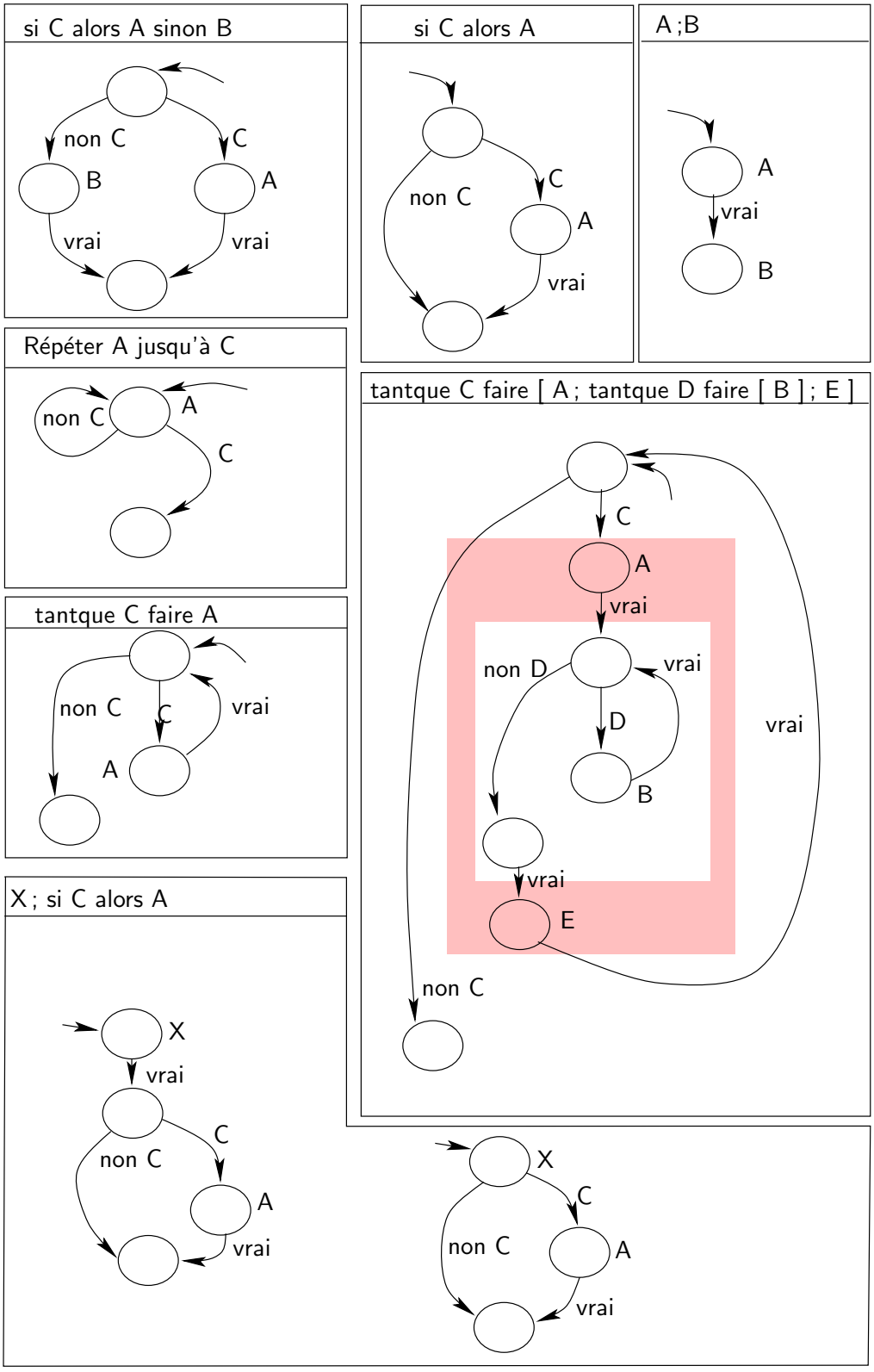


FIG. 5.6 – Traduction des structures de contrôle en machines séquentielles avec actions. Les états non étiquetés portent implicitement l'action vide.

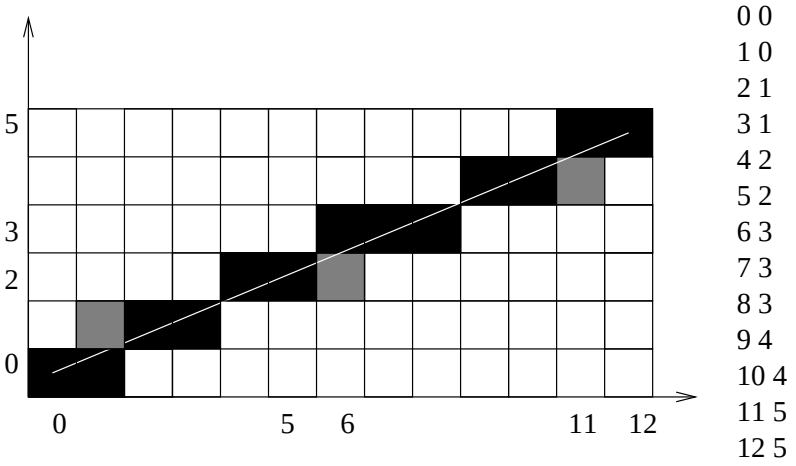


FIG. 5.7 – Tracé d'un segment dont les extrémités sont les points de coordonnées $(0,0)$ et $(12,5)$. Le trait blanc est *idéal*, les pixels noirs sont obtenus par l'algorithme, les pixels gris pourraient sembler candidats.

le segment de cette droite qui va du point $(0,0)$ au point (m,n). Les points n'ayant que des coordonnées entières, il faut noircir un ensemble de points (ou *pixels*, pour *picture element*) aussi proches que possibles de la droite idéale.

> **Remarque :** Sans perte de généralité, nous traitons le cas où $0 \leq n \leq m$. Les autres cas s'obtiennent aisément par des transformations simples où le point de coordonnées (j,k) devient $(\pm j, \pm k)$ ou $(\pm k, \pm j)$.

L'équation de la droite étant $y = (n/m)x$, avec m et n entiers, pour tout point de coordonnées entières (j,k), il est possible de calculer un écart ϵ par rapport à la droite idéale : $k = (n/m).j - \epsilon$ ou $\epsilon = (n/m).j - k$. Le critère de proximité retenu est le suivant : tout point de coordonnées (j,k) doit être tel que : $|\epsilon| \leq \frac{1}{2}$.

Evaluons la proximité relative de deux pixels par rapport à la droite idéale avec les valeurs $m = 12$, et $n = 5$ (Cf. Figure 5.7). Pour le pixel d'abcisse 1, calculons l'écart à la droite idéale de $(1,1)$ qui apparaît en grisé, et de $(1,0)$ qui est donné par l'algorithme ; pour $(1,1)$, $\epsilon = -\frac{7}{12}$ et pour $(1,0)$, $\epsilon = \frac{5}{12}$. C'est le point $(1,0)$ qui est donné par l'algorithme. Pour le point d'abscisse 6, les deux points $(6,2)$, en grisé, et $(6,3)$, en noir, donnent la même valeur de $|\epsilon|$.

De $|\epsilon| \leq \frac{1}{2}$ nous pouvons déduire :

$$\begin{array}{rcccl} -\frac{1}{2} & \leq & \epsilon & \leq & \frac{1}{2} \\ -\frac{1}{2} & \leq & (n/m).j - k & \leq & \frac{1}{2} \\ -m & \leq & 2.n.j - 2.m.k & \leq & m \\ -2m & \leq & 2.n.j - 2.m.k - m & \leq & 0 \end{array}$$

Posons $\Delta = 2.n.j - 2.m.k - m$. On remarque alors que lorsque j augmente de 1, Δ augmente de $2.n$; lorsque k augmente de 1, Δ diminue de $2.m$. La construction de l'algorithme de calcul des coordonnées des pixels successifs

```
lexique
    n : l'entier ... ; m : l'entier ...
    T : un tableau sur [0..m, 0..n] de booléens
    j, k, Δ : des entiers
algorithme
    k ⟵ 0 ; j ⟵ 0 ; Δ ⟵ − m
    { Valeur initiale de l'écart : l'abscisse j vaut 0, l'ordonnée k vaut 0, donc
    Δ = −m }
    tant que j ≤ m :
            { Invariant : 0 ≤ j ≤ m et -2*m ≤ Δ ≤ 0 }
            T_{j,k} ⟵ vrai { Le point de coord. j, k doit être affiché }
            { Pour le point suivant, on augmente j de 1 }
            j ⟵ j + 1 ; Δ ⟵ Δ + 2*n
            si Δ > 0
                { Si Δ est devenu trop grand, on le ramène à une valeur conve-
                nable en augmentant l'ordonnée courante }
                k ⟵ k + 1 ; Δ ⟵ Δ − 2*m
                { −2 ∗ m ≤ Δ ≤ 0 }
```

FIG. 5.8 – Algorithme de Bresenham

utilise cette propriété. La variable d'abscisse j est incrémentée de 1 en 1. A chaque incrémentation de j, k est mis à jour de façon à maintenir Δ entre $-2m$ et 0. Pour cela il faut soit laisser k inchangé, soit incrémenter k.

La figure 5.8 donne l'algorithme correspondant.

2.3.2 Machine séquentielle avec actions réalisant l'algorithme de Bresenham

Nous donnons Figure 5.9 le lexique des actions nécessaires à la définition de la machine séquentielle avec actions produite à partir de l'algorithme de Bresenham. La figure 5.10 décrit cette machine séquentielle.

Remarquons que cette machine a une forme particulière. Les états ne sont pas séparés si cela n'est pas nécessaire ; par exemple, l'action MajTetIncrAbs est constituée des trois actions élémentaires : $T_{j,k} \longleftarrow$ vrai, j $\longleftarrow$ j + 1 et $\Delta \longleftarrow \Delta + 2 * n$. Les prédicats se limitent à la consultation d'une variable booléenne (Fini ou Δpos). Le calcul des prédicats est systématiquement réalisé dans un état ; il pourrait parfois être intégré à un autre état : la mise à jour de Δpos pourrait, par exemple, être faite dans l'état où est réalisé l'action MajTetIncrAbs.

{ *Les variables :* }
Fini, Δpos : des booléens
{ *Les actions :* }
Init : une action : j $\longleftarrow$ 0 ; k $\longleftarrow$ 0 ; Δ $\longleftarrow$ $-$m
MajTetIncrAbs : une action
$T_{j,k}$ $\longleftarrow$ vrai ; j $\longleftarrow$ j + 1 ; Δ $\longleftarrow$ Δ + 2 * n
IncrOrdonnée : une action : k $\longleftarrow$ k + 1 ; Δ $\longleftarrow$ Δ $-$ 2 * m
CalculFini : une action : Fini $\longleftarrow$ j $>$ m
CalculΔpos : une action : Δpos $\longleftarrow$ Δ $>$ 0
{ *Les prédicats :* }
EstFini : $\longrightarrow$ un booléen : fini
ΔEstPos : $\longrightarrow$ un booléen : Δpos

FIG. 5.9 – Lexique de machine séquentielle avec actions représentant l'algorithme de Bresenham

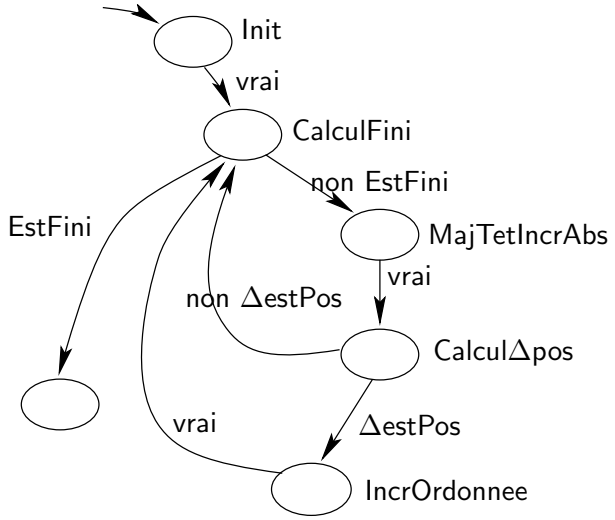


FIG. 5.10 – Machine séquentielle avec actions réalisant l'algorithme de Bresenham

2.4 Propriétés et transformations de machines séquentielles avec actions

Nous donnons ici quelques transformations des machines séquentielles à actions qui en préservent la sémantique — c'est-à-dire la séquence des actions effectuées sur les données du lexique — mais peuvent en modifier la structure. Plusieurs de ces transformations modifient le nombre d'états de la machine parcourus lors d'une séquence donnée d'actions. Lorsque l'on s'intéresse aux machines séquentielles à actions comme modèle intermédiaire dans le processus de traduction des langages de haut niveau vers un langage d'assemblage, cela a peu d'importance, et toutes les transformations seront permises. En revanche, si ce modèle de machine séquentielle est utilisé pour obtenir une réalisation matérielle de l'algorithme étudié, le nombre d'états sera en relation directe avec le *temps* d'exécution. En effet le cadencement des systèmes matériels suit assez rigoureusement la règle : *durée de séjour dans un état = une période d'horloge* ; en particulier la durée de séjour dans un état est une constante indépendante de l'état. Nous revenons sur cet aspect du problème au chapitre 11.

2.4.1 Fusion d'états

Si à la suite du processus de construction de l'algorithme deux états E1 et E2 d'une machine séquentielle à actions ne sont séparés que par une transition portant le prédicat vrai, on peut les fusionner. En effet, les propriétés de déterminisme et de réactivité des machines impliquent qu'il ne peut alors pas y avoir d'autre transition entre les deux états E1 et E2. Si les actions, A1 et A2, qu'ils portent sont *dépendantes*, l'état obtenu porte l'action A1 ; A2. Si les actions qu'ils portent sont *indépendantes*, on note A || B l'action composée portée par l'état obtenu, de préférence à A ; B ou B ; A pour rappeler que l'ordre est indifférent.

2.4.2 Eclatement d'états

Inversement, tout état portant une action composée de la forme A1 ; A2 peut être éclaté en deux états séparés par la transition portant le prédicat vrai, le premier portant l'action A1 et le deuxième l'action A2.

> **Remarque :** Dans l'exemple de Bresenham on aurait pu éclater en deux l'action MajTetIncrAbs. Le premier état porte l'action : $T_{j,k} \longleftarrow$ vrai ; j $\longleftarrow$ j + 1 ; $\Delta \longleftarrow \Delta$ + 2*n. Le deuxième : j $\longleftarrow$ j + 1 || $\Delta \longleftarrow \Delta$ + 2*n.

Nous verrons au chapitre 11 que lorsqu'il s'agit de produire un circuit synchrone pour implanter une machine séquentielle, il faut placer sur chaque état une action réalisable en 1 coup d'horloge. Cela peut imposer de décomposer des actions complexes en suites d'actions élémentaires réalisables en 1 seul coup d'horloge chacune. La machine séquentielle comporte alors une suite d'états séparés par des transitions portant le prédicat vrai.

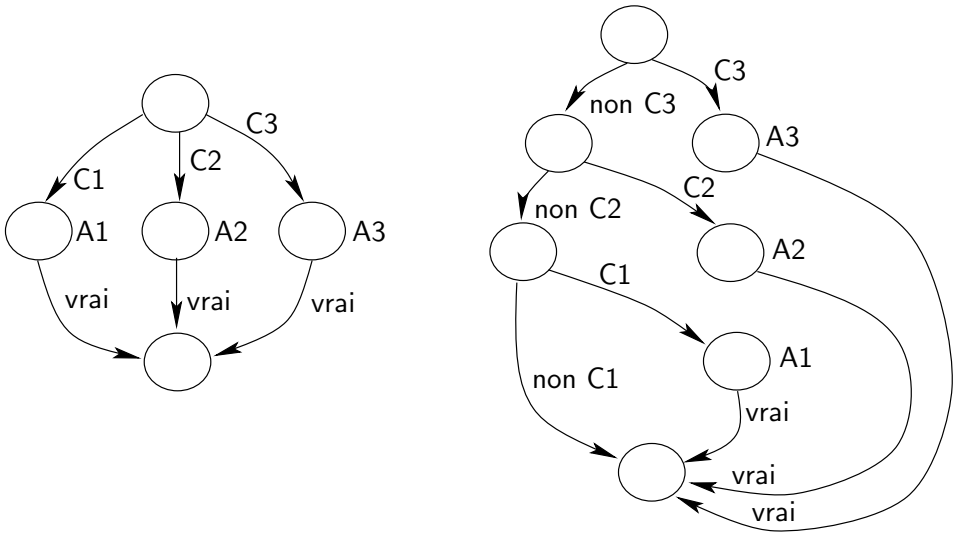


FIG. 5.11 – Transformation des tests n-aires en tests binaires

2.4.3 Transformation des branchements n-aires en branchements binaires

Que les machines séquentielles soient utilisées pour construire des circuits séquentiels synchrones (chapitre 11), ou pour produire du langage machine (chapitre 12), il faut parfois se restreindre à des branchements binaires. La transformation systématique d'une machine à branchements *n-aires* en machine à branchements uniquement binaires peut ajouter des états, et donc allonger le chemin nécessaire à l'exécution d'une action. Dans le cas logiciel comme dans le cas matériel, cet allongement du chemin se traduit par un allongement du *temps* d'exécution.

La figure 5.11 donne deux machines séquentielles correspondant à la structure conditionnelle :

selon
C1 : A1 ; C2 : A2 ; C3 : A3

La première machine possède un état à 3 transitions sortantes, pour lequel on exige : (C1 ou C2 ou C3) et non ((C1 et C2) ou (C2 et C3) ou (C1 et C3)).

La deuxième machine est à branchement binaire. Noter que le test des conditions peut se faire dans un ordre quelconque. Il existe donc 6 machines différentes ayant le même comportement. Noter également que si la condition de réactivité est bien respectée dans la machine à branchement binaire, la transition qui porte la condition non C1 est inutile.

2.4.4 Echange contrôle/données

Les deux algorithmes de la figure 5.12 produisent les mêmes résultats. La figure 5.14 représente les deux machines séquentielles avec actions associées, en utilisant le lexique décrit Figure 5.13.

```
lexique
    B1 : le booléen ... ; B2 : le booléen ... ; N : l'entier ... ; i : un entier
    T : un tableau sur [0..N] de booléens
    CondT : un entier ⟶ un booléen  { une propriété portant sur un entier }
algorithme 1 :
    i ⟵ 0
    tant que i ≤ N
        si CondT(i) alors Ti ⟵ (Ti et B1) sinon Ti ⟵ (Ti ou B2)
        i ⟵ i + 1
algorithme 2 :
    i ⟵ 0
    tant que i ≤ N
        Ti ⟵ (CondT(i) et (Ti and B1)) ou (non CondT(i) et (Ti ou B2))
        i ⟵ i + 1
```

FIG. 5.12 – Echange contrôle/données : deux algorithmes équivalents

```
{ lexique : }
    C1, C2 : des booléens
{ les actions : }
    Init : une action (la donnée-résultat i : un entier) : i ⟵ 0
    CalculC1 : une action (les données i : un entier, N : un entier) : C1 ⟵ i ≤ N
    CalculC2 : une action (la donnée i : un entier) : C2 ⟵ CondT(i)
    AndT : une action (les données : x : un booléen, i : un entier) : Ti ⟵ Ti et x
    OrT : une action (les données : x : un booléen, i : un entier) : Ti ⟵ Ti ou x
    ActCond : une action (les données : x1, x2 : deux booléens, i : un entier)
        Ti ⟵ (CondT(i) et (Ti et x1)) ou (non CondT(i) et (Ti ou x2))
{ les prédicats : }
    EstC1 : ⟶ un booléen : C1
    EstC2 : ⟶ un booléen : C2
```

FIG. 5.13 – Echange contrôle/données : lexique des machines séquentielles

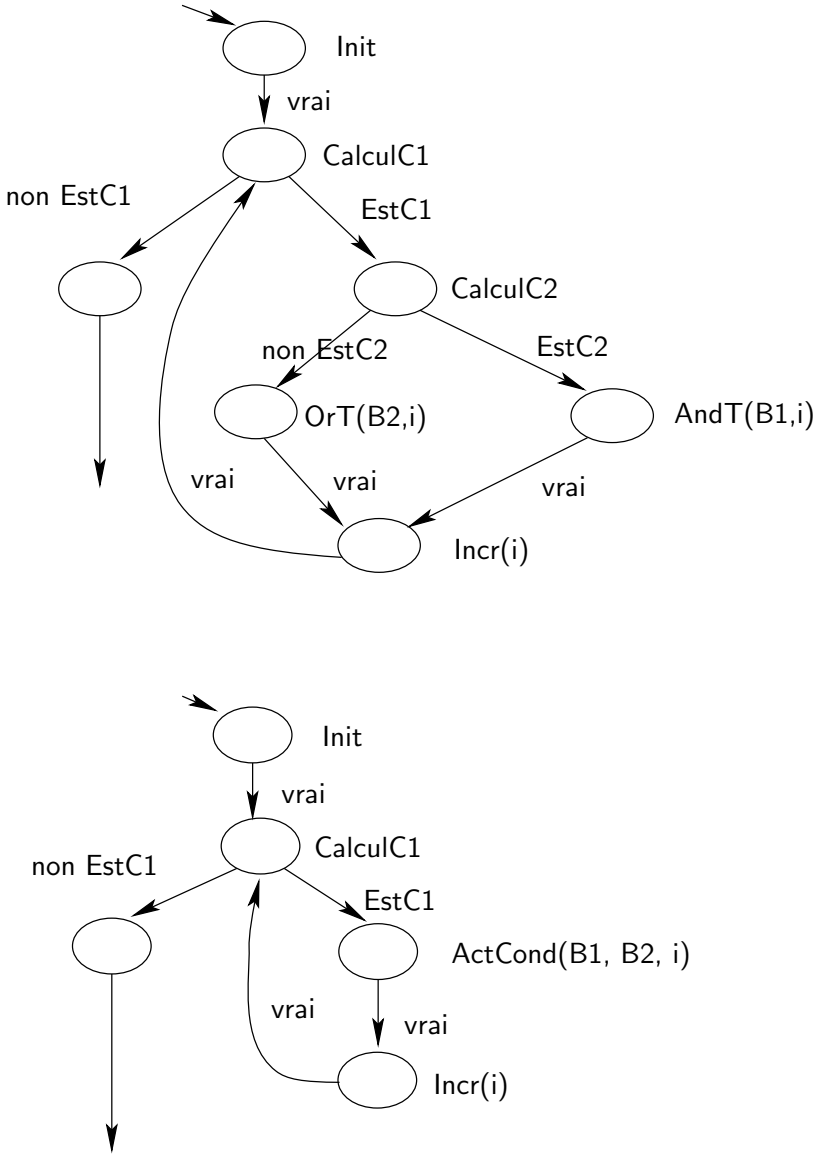


FIG. 5.14 – Deux machines séquentielles réalisant le même traitement

Dans la deuxième machine, l'utilisation de l'action ActCond permet l'économie du test portant sur C2, c'est-à-dire sur CondT.

Formellement les deux algorithmes ne sont pas équivalents. Dans le premier, une seule des deux expressions T_i et B1 et T_i ou B2 est évaluée ; dans l'autre les deux le sont. Il n'y a équivalence que si aucune des deux évaluations ne produit d'effet de bord.

Nous verrons au chapitre 11 que cette technique permettant de transférer des informations du contrôle aux données est utilisée lors de la répartition du travail entre une partie opérative et une partie contrôle. L'action ActCond correspond en matériel à l'utilisation d'un multiplexeur (Cf. Chapitre 8).

Chapitre 6

Temps, données temporelles et synchronisation

Ce chapitre est l'occasion d'introduire la notion de *temps* dans les systèmes informatiques. Quand on s'intéresse à un système informatique au niveau d'abstraction que donnent les langages de haut niveau, on peut se contenter d'une notion de temps *logique* pour raisonner sur la succession des opérations dans un programme. Cette notion de temps est qualifiée de logique parce qu'on ne s'intéresse pas à la relation avec le temps physique (même lorsque cette relation existe : pour un processeur donné et une chaîne de compilation donnée, elle est même exprimable).

En revanche, lorsqu'on s'intéresse aux modèles de traitements de bas niveau comme le langage machine, le séquencement des opérations est en rapport direct avec le temps physique. D'autre part, ne fût-ce que pour comprendre les mécanismes d'entrées/sorties, il faut s'interroger sur l'interface entre le dispositif informatique et son environnement, et sur le rapport entre les notions de temps de l'un et de l'autre : le temps de l'environnement est un temps physique *continu*; celui du système informatique est par nature *discret*.

Nous étudions tout d'abord au paragraphe 1. l'interface entre un environnement physique et un dispositif informatique réduit à une machine séquentielle (étudiée au chapitre 5). Le paragraphe 2. introduit la notion de signal logique *obtenu par discrétisation d'un signal physique continu, et la représentation de telles informations temporelles par des* chronogrammes*. Le paragraphe 3. s'intéresse aux problèmes de synchronisation de deux dispositifs informatiques connectés l'un à l'autre ; trois solutions sont envisagées, dont le protocole* poignée de mains *que nous utilisons dans les chapitres 11 et 16. Au paragraphe 4. nous reprenons l'exemple de la machine de distribution de café déjà étudiée au chapitre 5, pour préciser l'interface entre le contrôleur informatique et l'environnement physique de la machine.*

1. Interface entre un dispositif informatique et un environnement physique

Pour comprendre où intervient le *temps* dans les traitements informatiques, nous nous intéressons ici au cas où une machine séquentielle représente le fonctionnement d'un dispositif informatique directement connecté à un environnement physique.

1.1 Le temps logique discret des machines séquentielles

Bien que la définition mathématique des séquences et des machines séquentielles ne suppose pas l'introduction d'une notion de temps, il est assez naturel de parler d'*après* ou d'*avant* dans la séquence des entrées. L'indiciation des éléments de la séquence — c'est-à-dire l'ensemble des entiers naturels — est donc un bon candidat pour représenter une certaine notion de temps. Ce temps est qualifié de *logique* parce qu'on ne s'intéresse pas nécessairement à la relation entre les instants qu'il définit et un véritable temps physique. Il est dit *discret* parce que l'ensemble des entiers naturels n'est pas dense dans $\Re$ (une séquence indicée par les éléments de l'ensemble $\Re$ des réels représenterait plus naturellement un temps continu).

Tant qu'on utilise le modèle des machines séquentielles avec actions (Cf. Chapitre 5), on reste au niveau d'abstraction du logiciel. La séquence des entrées de la machine séquentielle est accessible grâce aux primitives Démarrer, Avancer, FinDeSéq et CarCour qui, dans un programme complet, seraient effectivement programmées. Elles peuvent représenter le parcours d'un tableau en mémoire, la saisie interactive au clavier, aussi bien que l'accès aux éléments d'un fichier présent sur un disque. Le fonctionnement de la machine, c'est-à-dire le déroulement de l'algorithme, dépend donc bien de paramètres de temps, comme le temps d'accès à la mémoire, le temps nécessaire pour réaliser une entrée clavier, le temps d'accès au disque, etc., mais d'une façon difficilement exprimable.

1.2 Le temps physique continu de l'environnement

Si la machine séquentielle considérée représente le fonctionnement d'un dispositif informatique directement connecté à un environnement physique, les alphabets d'entrée et de sortie représentent des informations en provenance ou à destination de cet environnement. Il faut alors exprimer précisément la relation entre les phénomènes continus qui nous intéressent dans l'environnement et la structure de séquence des entrées/sorties de la machine séquentielle.

On se ramène toujours à des phénomènes physiques que des appareils de mesure appropriés transforment en tensions électriques accessibles au dispositif informatique.

L'évolution d'une tension électrique en fonction du temps peut-être représentée par une courbe de fonction, comme illustré figure 6.1-a.

1.3 Définition de l'interface d'entrées/sorties de la machine séquentielle

Le dispositif informatique ne peut traiter que des informations *discrètes*. Nous avons vu au chapitre 3 comment ramener l'ensemble des valeurs possibles de G à un nombre fini de valeurs. On discrétise donc l'axe G en définissant une partition finie de l'ensemble des valeurs possibles, comme indiqué figure 6.1-b où il y a deux valeurs. On peut ensuite reporter les variations continues sur cette nouvelle échelle GD. On obtient une suite de paliers de longueurs quelconques, comme indiqué figure 6.1-c. Notons que deux paliers successifs sont à des hauteurs distinctes, par construction.

On va se limiter au cas des informations booléennes (pour lesquelles l'ensemble des valeurs a été partitionné en deux). Ce qui est en dessous du seuil devient la valeur la plus basse (codée par 0), et ce qui est au-dessus du seuil devient la plus haute (codé par 1).

1.4 Discrétisation du temps : interprétation synchrone ou asynchrone

Pour compléter la définition de l'interface entre l'environnement et le dispositif informatique représenté par une machine séquentielle, il faut définir la structure de séquence, c'est-à-dire décider comment la suite de paliers de la figure 6.1-c doit être interprétée en une *séquence* d'éléments de l'alphabet d'entrée, à fournir à la machine.

Il y a essentiellement deux choix : l'interprétation *asynchrone*, et l'interprétation *synchrone*, que nous exposons ci-dessous.

1.4.1 Interprétation asynchrone

En interprétation asynchrone, la structure de séquence est définie par les changements de hauteurs de paliers.

Dans le cas d'une information booléenne, qui ne comporte que deux hauteurs de paliers, on parle de *front montant* ou de *front descendant*, selon qu'on passe du niveau inférieur au niveau supérieur ou inversement. Notons que cette interprétation de la suite de paliers donne des séquences où les fronts montants et descendants alternent, par construction.

Par conséquent, quelle que soit la courbe de la grandeur mesurée, et quelle que soit la position des fronts sur l'échelle de temps physique, la séquence des hauteurs de paliers est une alternance de 0 et de 1 ; la séquence des fronts porte exactement la même information. Il n'est donc pas très intéressant de considérer la réaction d'une machine séquentielle à cette séquence d'entrées.

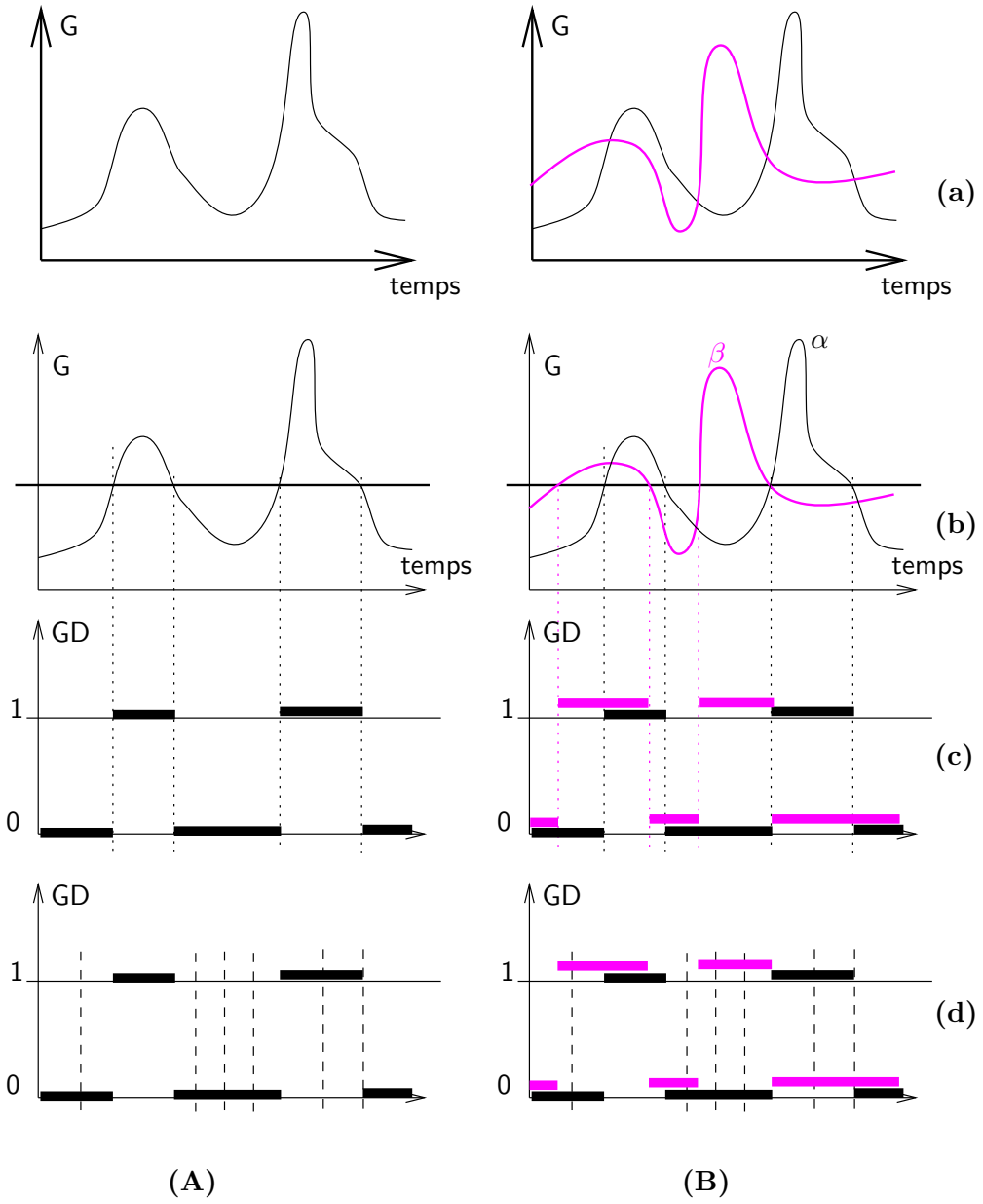


FIG. 6.1 – Séquence d'entrées correspondant à une grandeur continue de l'environnement :
a) évolution d'une grandeur continue ; b) discrétisation de l'axe G ; c) discrétisation du temps, interprétation asynchrone ; d) discrétisation du temps, interprétation synchrone.
A) Cas d'une grandeur ; B) cas de plusieurs grandeurs

En revanche, dès que l'on considère plusieurs grandeurs, les paliers (ou, de manière équivalente, les fronts) sont superposés. En associant une variable booléenne — par exemple α — à chacune des grandeurs, et en notant α la valeur 1 de cette grandeur, $\overline{\alpha}$ la valeur 0 de cette grandeur, on peut construire une séquence de monômes booléens qui reflète les superpositions de paliers. On passe à un nouvel élément de la séquence dès que l'une au moins des deux grandeurs change de palier. Pour l'exemple de la figure 6.1-Bc, on construit la séquence

$$\overline{\alpha}.\overline{\beta},\ \overline{\alpha}.\beta,\ \alpha.\beta,\ \alpha.\overline{\beta},\ \overline{\alpha}.\overline{\beta},\ \overline{\alpha}.\beta,\ \alpha.\overline{\beta},\ \overline{\alpha}.\overline{\beta}$$

Il devient intéressant de décrire des machines séquentielles capables de traiter des séquences ainsi construites.

Exemple E6.1 : Interprétation asynchrone de deux grandeurs et comptage

Considérons une machine séquentielle qui perçoit deux grandeurs α et β, et dont la sortie booléenne γ est vraie si et seulement si les deux grandeurs ont eu la même valeur un nombre pair de fois dans le passé.

En utilisant la séquence des niveaux superposés, on écrira par exemple la machine de Moore suivante :

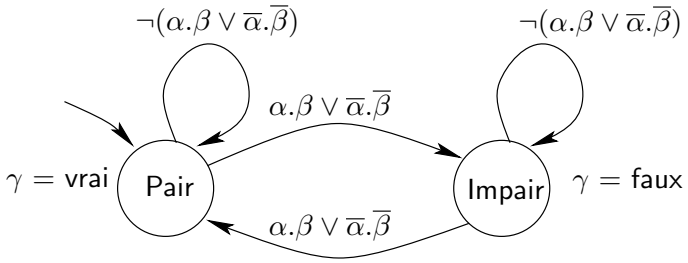


FIG. 6.2 – Machine de Moore lisant la séquence des niveaux

Pour la séquence $\overline{\alpha}.\overline{\beta},\ \overline{\alpha}.\beta,\ \alpha.\beta,\ \alpha.\overline{\beta},\ \overline{\alpha}.\overline{\beta},\ \overline{\alpha}.\beta,\ \alpha.\overline{\beta},\ \overline{\alpha}.\overline{\beta}$, la séquence de sortie est : $\gamma, \overline{\gamma}, \overline{\gamma}, \gamma, \gamma, \overline{\gamma}, \overline{\gamma}, \overline{\gamma}, \gamma$.

On peut aussi considérer que δ dénote le front montant d'une grandeur booléenne D, et $\overline{\delta}$ son front descendant. La séquence construite pour l'exemple de la figure 6.1-Bc est alors : $\beta, \alpha, \overline{\beta}, \overline{\alpha}, \beta, \alpha.\overline{\beta}, \overline{\alpha}$. Notons que l'origine des temps n'est pas considérée comme un front. D'autre part rien n'empêche d'envisager le changement simultané des deux grandeurs, d'où l'existence d'éléments de la séquence de la forme $\alpha.\overline{\beta}$.

1.4.2 Interprétation synchrone

L'interprétation synchrone est un cas particulier de l'interprétation asynchrone décrite ci-dessus pour deux grandeurs, dans lequel on considère que

l'une des grandeurs est *l'horloge* de l'autre. La grandeur choisie comme horloge définit un découpage de l'axe du temps qui permet d'*échantillonner* l'autre grandeur. Ce découpage n'est pas nécessairement régulier en temps physique ; l'axe du temps sous-jacent n'est pas découpé en intervalles de tailles égales, quoique ce soit généralement le cas avec des horloges régulées par des quartz.

En interprétation synchrone, on a donc toujours au moins deux grandeurs. Notons d'ailleurs que *synchrone* signifie littéralement *qui partage le même temps*, et qu'il faut être au moins deux pour partager quelque chose. Deux grandeurs seront dites *synchrones* si elles sont échantillonnées sur la même horloge, *asynchrones* sinon.

A partir d'une grandeur qui sert d'horloge et d'une ou plusieurs autres grandeurs, on fabrique une séquence d'entrées de la machine séquentielle en créant un élément de séquence par front d'horloge : c'est un monôme qui décrit le niveau des autres grandeurs à l'instant de ce front.

> *Nous verrons qu'une machine séquentielle peut être réalisée par un circuit séquentiel synchrone (Cf. Chapitres 10 et 11). Une horloge détermine alors les instants auxquels la machine change d'état. Un processeur peut être vu comme une machine séquentielle synchrone cadencée elle-aussi par son horloge (Cf. Chapitre 14). Il existe aussi des réalisations, dont des processeurs, asynchrones. Nous n'étudions pas cette technique dans ce livre.*

Exemple E6.2 : Machine à café (*suite de l'exemple E5.2*)
Nous envisagions une séquence d'entrées commençant par $\overline{s_1}.\overline{s_2}.\overline{s_5}, \overline{s_1}.\overline{s_2}.\overline{s_5}, \overline{s_1}.\overline{s_2}.\overline{s_5}, ...$

Si l'on utilise l'interprétation asynchrone définie ci-dessus, les entrées s_1, s_2, s_5 *et* f_s *de la machine à café sont superposées, et on en déduit une séquence d'entrées en créant un nouvel élément uniquement quand l'une au moins change. La séquence ci-dessus n'apparaît donc jamais.*

Si l'on utilise l'interprétation synchrone, en revanche, on introduit une cinquième entrée implicite : l'horloge. On construit un élément de la séquence pour chaque période d'horloge. La séquence ci-dessus peut donc apparaître.

2. Signaux logiques et représentation par des chronogrammes

Les grandeurs physiques continues dont nous avons envisagé la discrétisation sont des *signaux* physiques. Nous appellerons *signal logique* l'échantillonnage d'un tel signal physique par les fronts d'un autre signal qui sert d'horloge.

> *On étudie l'influence des problèmes de synchronisation sur la réalisation des automates synchrones dans le chapitre 10.*

L'évolution au cours du temps des horloges et des signaux logiques peut être représentée par des courbes en créneaux carrés, comme sur la figure 6.3.

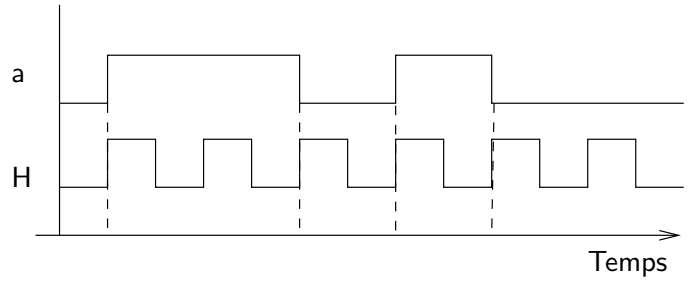


FIG. 6.3 – Un exemple de représentation de signaux logiques par des chronogrammes : H est un signal d'horloge, et a est un signal logique d'horloge H (noter que l'horloge est un signal booléen. Ici le signal a est également booléen).

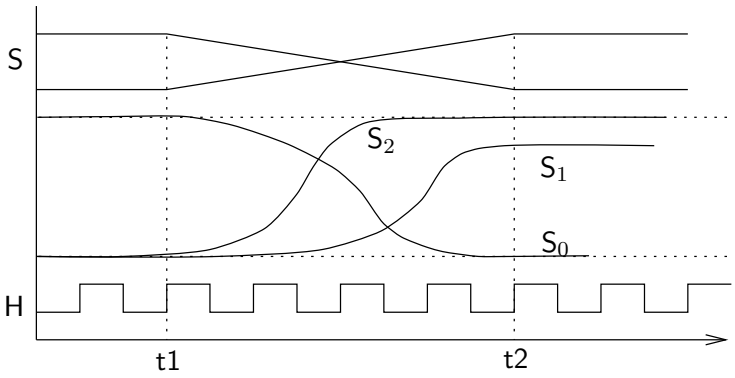


FIG. 6.4 – Représentation de l'évolution de grandeurs : la valeur S codée sur 3 bits S_0, S_1 et S_2 est momentanément instable entre les instants t1 et t2

Ces courbes sont des *chronogrammes*.

Si l'on s'intéresse au temps de changement de valeur discrète d'un signal par rapport au rythme d'une horloge H, et aux éventuels problèmes d'échantillonnage qui en découlent, on peut représenter l'évolution temporelle des grandeurs en jeu par une figure comme 6.4.

Pour représenter des valeurs indéfinies ou non significatives, nous utilisons aussi les représentations données dans la figure 6.5.

3. Problèmes de synchronisation

Nous avons envisagé jusqu'ici le cas d'un dispositif informatique connecté à un environnement physique dont il doit échantillonner les grandeurs.

Si l'on s'intéresse à plusieurs dispositifs informatiques, on peut considérer chacun comme l'environnement de l'autre : les sorties de l'un peuvent être les entrées de l'autre. Pour étudier les problèmes de synchronisation entre systèmes informatiques, on suppose que les deux systèmes sont décrits par des machines séquentielles, et que les entrées de l'un peuvent être les sorties de l'autre.

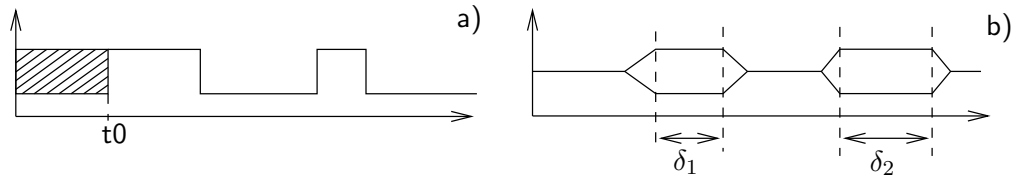


FIG. 6.5 – Représentations particulières de valeurs :
a) Signal booléen dont la valeur est indéfinie avant l'instant t0
b) La valeur n'est significative que pendant les périodes δ_1 et δ_2 ; ce type de schéma est souvent utilisé pour représenter la valeur présente sur un bus : lorsque aucun composant n'est connecté au bus sa valeur n'est pas significative.

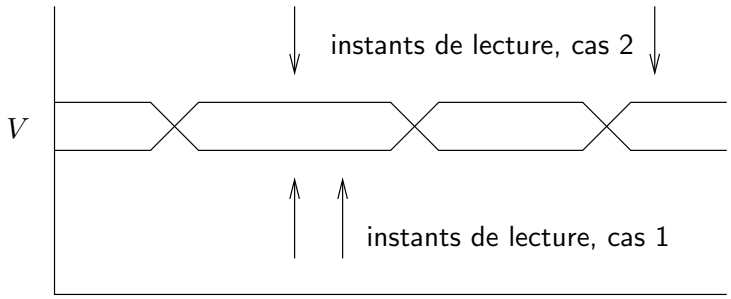


FIG. 6.6 – Accès à une valeur commune V

Si les deux dispositifs A et B reçoivent un même signal qui peut servir d'horloge commune H, ils peuvent échantillonner toutes les grandeurs sur la même horloge. Dans le cas contraire, les deux dispositifs A et B peuvent néanmoins avoir des horloges locales, c'est-à-dire utiliser chacun un signal particulier comme horloge pour échantillonner les autres signaux, mais chacun doit être considéré comme l'environnement asynchrone de l'autre.

3.1 Le problème général d'accès à un signal commun

On considère deux dispositifs informatiques appelés *récepteur* et *émetteur*, qui doivent se mettre d'accord sur une valeur V produite par l'un et consommée par l'autre. L'émetteur a un comportement cyclique : il maintient une valeur sur le fil (ou les fils) V pendant un certain temps, puis fabrique une nouvelle valeur (pendant ce temps l'état du fil est indéterminé) et la maintient sur le fil, etc.

Le récepteur a également un comportement cyclique : il accède à ce fil en lecture ; consomme la valeur (ce traitement prend un certain temps) ; accède de nouveau à V, etc.

Le problème posé comporte deux contraintes :

– Le récepteur ne doit pas consommer deux fois la même valeur

– Le récepteur ne doit pas ignorer une valeur

Si les deux dispositifs évoluent de manière complètement indépendante l'un de l'autre, les instants de lecture sont quelconques : les deux problèmes ci-dessus peuvent survenir. Voir figure 6.6 : dans le cas 1, les instants de lecture sont trop proches, le récepteur lit plus vite que l'émetteur ne produit ; dans le cas 2, les instants de lecture sont trop éloignés, le récepteur ne lit pas assez vite.

Il faut donc se débrouiller pour *synchroniser* l'émetteur et le récepteur pour l'accès à la valeur commune V. Cette synchronisation est assurée par un *protocole de communication.*

3.2 Protocole *poignée de mains* et mise en oeuvre

3.2.1 Le protocole

Pour éviter les deux cas de fonctionnement incorrect décrits par la figure 6.6, on doit assurer que :

1. le récepteur ne peut pas lire deux fois la donnée V sans avoir été prévenu par l'émetteur d'un changement entre temps ;
2. l'émetteur ne peut pas modifier la valeur de la donnée (c'est-à-dire émettre deux valeurs différentes) à moins d'avoir été prévenu par le récepteur entre temps que la première valeur a effectivement été consommée.

On introduit à cet effet deux *signaux* de synchronisation E_prêt et R_prêt. E_prêt est produit par l'émetteur et consommé par le récepteur. R_prêt est produit par le récepteur et consommé par l'émetteur. L'idée est d'assurer la synchronisation par un dialogue entre l'émetteur (E) et le récepteur (R), de la forme suivante : E est responsable de la production des valeurs V, et prévient R de l'apparition d'une nouvelle valeur — c'est le signal E_prêt ; R attend d'être ainsi prévenu pour consommer la valeur présente sur le fil ; il envoie ensuite à E un acquittement de lecture — c'est le signal R_prêt ; lorsqu'il reçoit l'aquittement de lecture en provenance de R, E peut procéder à la production d'une nouvelle valeur.

> **Remarque :** Cette idée d'un échange d'informations supplémentaires du type *j'ai écrit* et *j'ai bien lu*, pour réguler les accès en lecture/écriture à une information partagée est une idée simple et très générale. La complexité des protocoles de communication dans les réseaux informatiques tient à un autre problème : les lignes de transmission entre l'émetteur et le récepteur ne peuvent pas être considérées comme fiables, ce qui oblige à prévoir la réémission des messages et de leurs acquittements. En effet, lorsqu'un signal comme X_prêt est émis par l'un, on n'a pas de garantie de réception par l'autre.

3.2.2 Mise en oeuvre, cas général

L'émetteur a une horloge H_e et le récepteur une horloge H_r. Les deux signaux E_prêt et R_prêt donnent deux signaux logiques chacun, selon qu'ils sont échantillonnés par l'horloge de l'émetteur ou par l'horloge du récepteur. On considère les 4 signaux logiques suivants p_ep, d_ep, p_rp, d_rp (Cf. Figure 6.8) : émission du signal E_prêt (échantillonné sur H_e), détection du signal E_prêt (échantillonné sur H_r), émission du signal R_prêt (échantillonné sur H_r), détection du signal R_prêt (échantillonné sur H_e).

Le préfixe p_ indique la production du signal, le préfixe d_ indique sa détection.

E_prêt est égal à son échantillonnage sur l'horloge H_e, puisqu'il est produit sur cette horloge ; il est en revanche différent de son échantillonnage sur H_r.

Fonctionnement temporel de l'émetteur et du récepteur La figure 6.7 donne les machines de Moore décrivant le comportement temporel de l'émetteur et du récepteur, en terme des signaux logiques p_ep, d_ep, d_rp, p_rp. Chacune des machines change d'état sur les fronts de son horloge, d'après la valeur des signaux de communication à cet instant-là.

En observant le comportement des deux machines séquentielles, on peut se convaincre des propriétés suivantes :

- Le récepteur ne peut pas passer deux fois dans l'état de lecture de V sans que l'émetteur ait quitté son état d'écriture.
- Symétriquement, l'émetteur ne peut pas passer deux fois dans l'état d'écriture sans que le récepteur soit passé dans son état de lecture.

Représentation par des chronogrammes Les chonogrammes de la figure 6.8 illustrent les contraintes d'ordre sur les fronts de la donnée V et des signaux logiques p_ep, d_ep, d_rp, p_rp, imposées par le protocole *poignée de mains* ainsi que l'état courant de l'émetteur et du récepteur.

3.2.3 Mise en oeuvre : cas particuliers

Synchronisation par horloge commune ou horloges inverses : Lorsque les deux dispositifs qui communiquent échantillonnent les grandeurs sur la même horloge, le schéma de la figure 6.8 est simplifié : il n'y a pas de décalage temporel entre la production d'un signal et sa détection (si l'on néglige le délai de transmission du signal d'horloge dans les connexions physiques par rapport au temps de traversée d'un circuit combinatoire).

Le cas des horloges inverses, $H_e = \overline{H_r}$ est un cas simple où l'opposition de phase des horloges des deux systèmes résoud les problèmes d'échantillonnage et de stabilité des grandeurs échangées.

Emetteur (changements
d'état sur fronts montants de H_e)

Récepteur (changements
d'état sur fronts montants de H_r)

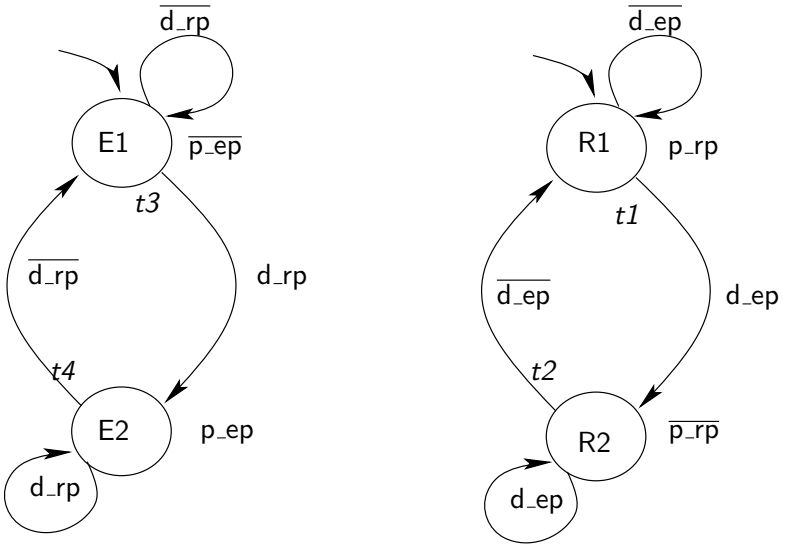


FIG. 6.7 – Machines de Moore décrivant le fonctionnement temporel de l'émetteur et du récepteur, dans le cas d'un protocole de poignée de mains. Etat E1 : attente d'émission ; Etat E2 : émission de V et attente d'acquittement de la part du récepteur. Etat R1 : attente de valeur ; Etat R2 : émission de l'acquittement et attente de prise en compte de cet acquittement par l'émetteur. Transition t1 : consommation de la valeur V ; Transition t2 : reconnaissance du fait que l'acquittement de consommation de V a été pris en compte par l'émetteur ; Transition t3 : prise en compte de l'acquittement en provenance du récepteur ; Transition t4 : reconnaissance du fait que le récepteur traite l'information envoyée par l'émetteur.

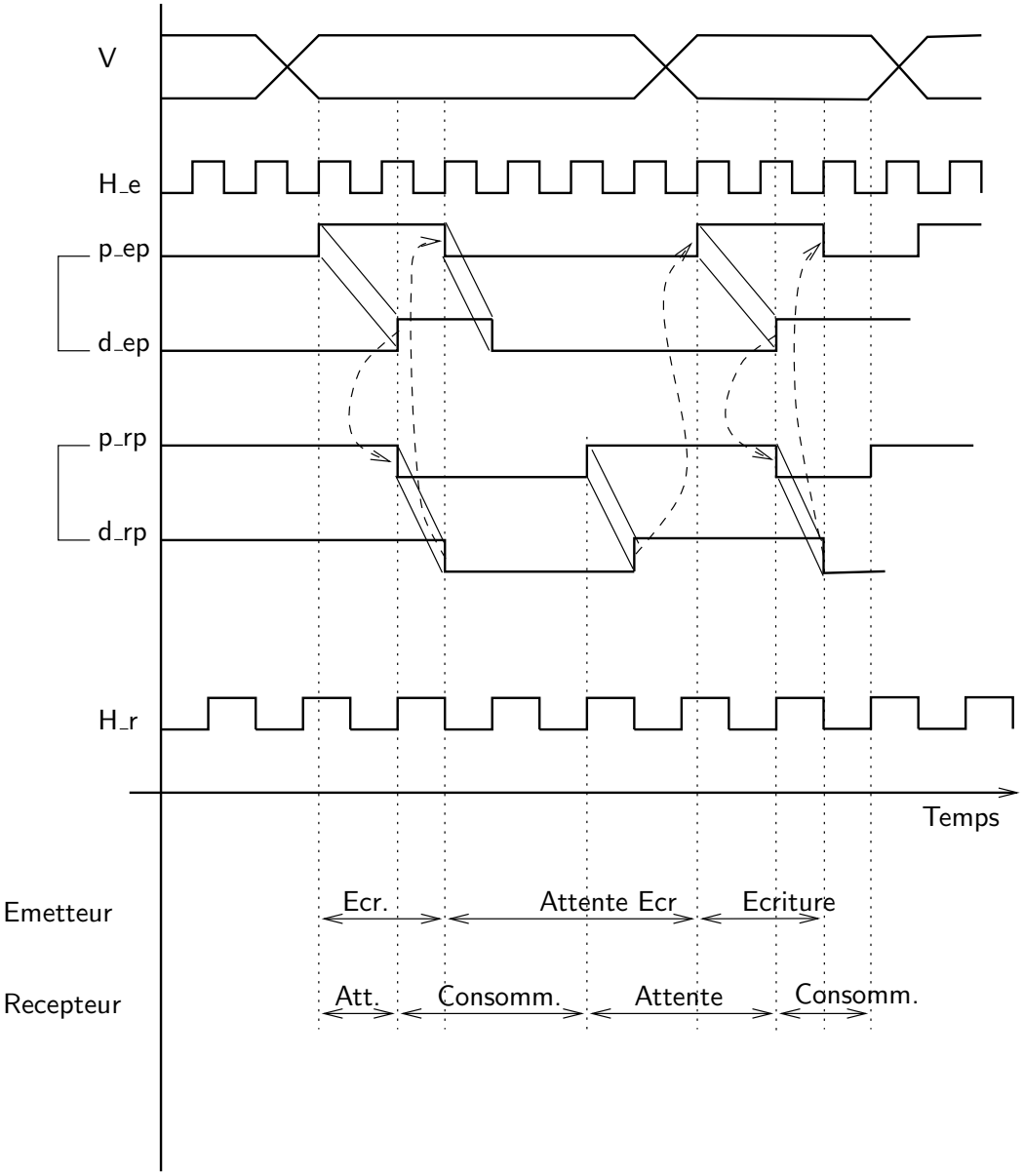


FIG. 6.8 – Comportement temporel des signaux dans un protocole poignée de mains. On a représenté : la donnée V dont les valeurs sont émises par l'émetteur, sur son horloge H_e ; l'horloge du récepteur H_r ; les signaux logiques p_ep, d_ep (resp. d_rp, p_rp) qui correspondent à l'échantillonnage du signal E_prêt (resp. R_prêt) sur les horloges de l'émetteur et du récepteur. Les courbes pointillées grasses terminées par une flèche illustrent des relations de cause à effet, déductibles du fonctionnement temporel de l'émetteur et du récepteur. Les lignes obliques en trait plein, sans flèche, illustrent les décalages temporels entre la production d'un signal, c'est-à-dire son échantillonnage sur l'horloge du producteur, et la détection de ce signal, c'est-à-dire son échantillonnage sur l'horloge du consommateur.

Synchronisation avec délai constant : Lorsque le temps de réaction (consommation) du récepteur est toujours le même, et connu lors de la construction du système qui fait communiquer les deux dispositifs informatiques, la mise en oeuvre du protocole de poignée de mains est très simplifiée : le signal d'acquittement en provenance du récepteur n'est plus un vrai signal physique : il est implicite. L'émetteur peut en effet considérer que l'acquittement *j'ai bien lu* survient n coups d'horloges après la production du signal *j'ai écrit*; il peut même arriver que $n = 1$.

> *C'est un mode de synchronisation qui peut parfois être utilisé entre le processeur (l'émetteur) et une mémoire (le récepteur) (Cf. Chapitres 14 et 15).*

Emetteur rapide : Si l'émetteur est supposé beaucoup plus rapide que le récepteur, on sait que le récepteur ne peut pas consommer deux fois la même valeur. Il suffit d'assurer que le récepteur n'ignore pas de valeur. Pour cela, on ajoute un signal de synchronisation qui permet au récepteur de signaler qu'il a consommé une valeur. L'émetteur attend cet acquittement avant de produire une nouvelle valeur. En fait le récepteur est *esclave* de l'émetteur : il n'a pas d'horloge propre, et utilise l'un des signaux émis par l'émetteur comme horloge.

Récepteur rapide : Inversement, si le récepteur est supposé beaucoup plus rapide que l'émetteur, on sait qu'aucune valeur émise ne peut lui échapper. Il suffit d'assurer qu'il ne lit pas deux fois la même valeur. Pour cela on ajoute un signal de synchronisation qui permet à l'émetteur de signaler qu'il a produit une nouvelle valeur. Le récepteur attend cet avertissement pour lire.

4. Un exemple : la machine à café

Exemple E6.3 : Machine à café (*suite de l'exemple E5.2, p 105*)
Nous reprenons l'exemple de la machine à café. Il s'agit d'étudier maintenant la définition des séquences d'entrées de la machine séquentielle qui représente le contrôleur, d'après les grandeurs physiques qui évoluent dans l'environnement de ce contrôleur.

On considère que les divers dispositifs électromécaniques de la machine à café émettent des signaux physiques que l'on échantillonne sur l'horloge du contrôleur informatique. Cette horloge est supposée beaucoup plus rapide que le temps de réaction des capteurs.

La figure 6.9 donne : l'horloge H du contrôleur ; le signal physique `s1` issu du capteur qui détecte l'insertion d'une pièce de 1F ; le signal logique `s1h` obtenu par échantillonnage de `s1` sur l'horloge du contrôleur ; le signal logique

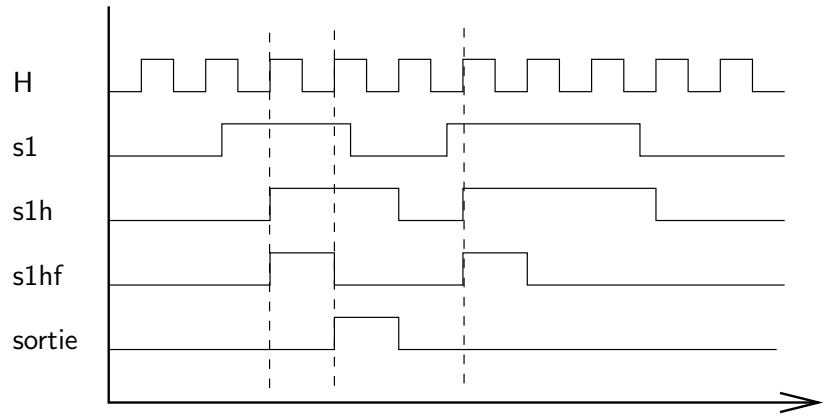


FIG. 6.9 – Signaux d'entrée et de sortie de la machine à café

s1hf obtenu par détection des fronts montants de s1h ; une sortie sortie de la machine séquentielle.

Il est nécessaire de détecter les fronts de s1h afin de fournir en entrée du contrôleur un signal logique qui indique l'insertion d'une pièce pendant au plus une période d'horloge. En effet la machine séquentielle qui représente le contrôleur change d'état à chaque période d'horloge, et risquerait sinon d'utiliser plusieurs fois le même signal pour compter une pièce de 1F. Nous verrons au chapitre 9, paragraphe 1.2.4, un dispositif matériel capable de réaliser cette détection de fronts.

Si l'entrée s1 fait passer dans un état où la sortie sortie est active, le signal logique correspondant à cette sortie est vrai dès la période d'horloge qui suit le front montant de s1h et le reste pendant toutes les périodes d'horloge où la machine séquentielle est dans le même état.

Deuxième partie

Techniques de l'algorithmique matérielle

Chapitre 7

De l'électron aux dispositifs logiques

L'objet de ce chapitre est de montrer quels phénomènes physiques élémentaires sont mis en oeuvre dans les réalisations matérielles de certaines fonctions dont, principalement, les fonctions booléennes. Ces réalisations matérielles reçoivent le nom de dispositifs logiques. Nous verrons plus loin comment combiner de telles fonctions pour réaliser les éléments d'un ordinateur. Cela se fera seulement à travers un moyen de réalisation des dispositifs : la technologie CMOS (Complementary Metal Oxyde Semiconductor). Nous ne donnons que les principes généraux. Il n'est pas question ici d'inclure un cours complet de physique ou d'électronique donnant les tenants et aboutissants de chacun des phénomènes étudiés.

> *Nous envisageons les phénomènes sous des points de vue d'abstraction croissante : l'échelle atomique, où l'on parle d'atomes et d'électrons (paragraphe 1.) ; l'échelle électrique, où l'on parle de résistances, de condensateurs et de transistors (paragraphe 2.) ; l'échelle logique, où l'on parle de fonctions booléennes (paragraphe 3.). Nous nous éloignons ainsi progressivement des phénomènes physiques pour en avoir une vision en terme d'information. Cela permet de décrire l'ensemble des circuits logiques utilisés dans les ordinateurs (paragraphe 4.). Nous donnerons aussi un bref aperçu de la fabrication des circuits, notamment en raison de l'influence qu'elle a sur les méthodes de conception (paragraphe 5.).*

1. Phénomènes à l'échelle atomique

1.1 Atomes, électrons et cristaux

1.1.1 Atomes, électrons

La matière est constituée d'atomes. Chaque atome est constitué d'un noyau et d'un cortège d'électrons appelé *nuage électronique*. Les électrons

portent chacun une charge électrique élémentaire négative et le noyau autant de charges positives qu'il y a d'électrons. On répartit les électrons selon leur énergie en niveaux d'énergie. La classification périodique des éléments de Mendeleïev donne pour chaque élément : le nombre d'électrons dans le cortège ; le nombre de niveaux d'énergie contenant des électrons ; le nombre d'électrons appartenant au niveau d'énergie le plus élevé (la *couche externe*). Extrayons une partie de cette table :

B bore	C carbone	
	Si silicium	P phosphore
Ga gallium	Ge germanium	As arsenic

Le carbone, le silicium et le germanium ont 4 électrons au niveau d'énergie le plus élevé, le bore et le gallium en ont 3, le phosphore et l'arsenic 5.

1.1.2 Cristaux

Les atomes d'un corps sont liés entre eux plus ou moins fortement et peuvent se disposer les uns par rapport aux autres selon des structures régulières : les cristaux. Le diamant et le graphite sont 2 organisations physiques différentes du même élément chimique carbone. De même il existe des variétés de silicium monocristallin et polycristallin qui sont obtenues par des procédés de fabrication différents.

1.2 Courant et conducteur

L'organisation des atomes en réseaux cristallins entraîne un élargissement des niveaux d'énergie (qui sont discrets) en bandes d'énergies (qui sont continues) et une délocalisation des électrons de plus haute énergie sur l'ensemble du réseau. Le courant électrique est un mouvement d'ensemble de particules chargées, ici les électrons. Qui dit mouvement dit énergie cinétique, donc variation de l'énergie totale de l'électron. Ceci n'est possible que s'il trouve une *place* à l'énergie correspondante dans une bande d'énergie autorisée et non pleine.

1. Si la dernière bande n'est pas pleine, l'énergie nécessaire à cette excursion est faible : on parle de *conducteur* comme le cuivre, l'or, l'aluminium.
2. Si la dernière bande est pleine et séparée de la suivante par une zone d'énergie non autorisée (*gap*), l'énergie nécessaire à la production d'un courant électrique est forte : on parle d'*isolant*. Le quartz est un cristal isolant d'oxyde de silicium. Le verre est un oxyde de silicium, isolant, mais non cristallin.
3. Il arrive que le gap soit faible, l'énergie nécessaire est alors intermédiaire : on parle de *semi-conducteur*. Le silicium et le germanium sont deux corps simples semi-conducteurs. L'arseniure de gallium est un corps composé semi-conducteur. Ces trois matériaux sont les constituants de base des

circuits électroniques. Le silicium est le plus répandu dans les composants utilisés en informatique. Le dioxyde de silicium peut être utilisé comme isolant, il peut être obtenu *facilement* à la surface du silicium.

En gagnant de l'énergie (par exemple d'origine thermique), un électron peut atteindre la bande de conduction et s'éloigner, laissant derrière lui un trou dans la bande de valence et un atome chargé positivement. Il y a donc création d'une paire (électron mobile négatif, trou fixe positif). Réciproquement, un autre électron perdant de l'énergie peut venir combler ce trou et rétablir l'équilibre électrique de l'atome. On parle alors de recombinaison électron-trou. Du point de vue électrique, il est alors commode de considérer que c'est un trou positif qui s'est déplacé dans le cristal.

Dans un semiconducteur pur il y a autant de trous que d'électrons.

1.3 Diffusion et dopage

Faites cette expérience (ou imaginez-la) : prenez un verre de thé (pas une tasse, un verre) pas trop fort mais pas trop clair, Darjeeling, Earl Grey, Lapsang-Souchong, ...au choix. A la surface du liquide déposez délicatement UNE goutte de lait. Ne remuez pas le verre et regardez par transparence. Il y a *diffusion* du lait dans le thé. Au bout d'un certain temps, en un point du verre de thé, la concentration de lait est fonction de la distance par rapport au point de dépôt de la goutte, de la concentration du thé, de la grosseur de la goutte, de la température ...

Imaginez le même phénomène de diffusion d'un solide (du phosphore) dans un autre solide (du silicium). Bien sûr il faut chauffer un peu, et on ne voit rien par transparence.

Le résultat de l'expérience précédente est intéressant en termes électriques. Les éléments silicium et phosphore sont voisins par leur structure électronique : il y a un électron de plus dans le phosphore. L'introduction de phosphore dans le silicium modifie la structure et l'équilibre atomiques. Le silicium ainsi traité est devenu meilleur conducteur. La différence de résistivité est importante. En apportant un atome de phosphore pour 100 millions d'atomes de silicium, la résistivité est divisée par un facteur de l'ordre de 30 000.

On dit que le silicium a été *dopé* ; on parle de dopage négatif puisqu'il y a excès d'électrons. Quand le silicium a reçu, par diffusion, des atomes de phosphore, TOUT SE PASSE COMME SI on avait du silicium avec des électrons libres, non liés aux atomes.

On peut aussi doper positivement le silicium en diffusant du bore qui a un électron de moins et obtenir un excès de trous.

L'intérêt du silicium est qu'il est *facilement* dopable et que le dioxyde de silicium est, lui, un obstacle au dopage. Par facilité de langage on dit souvent dopé N (pour Négatif, excès d'électrons) ou dopé P (pour Positif, excès de trous) en parlant du silicium.

Une étude plus détaillée de la physique des dispositifs semi-conducteurs se trouve dans [CW96] ou [GDS98].

2. Phénomènes à l'échelle électrique

2.1 Rappels d'électricité élémentaire

- La résistance R d'un fil électrique homogène de section constante est proportionnelle à la longueur L du fil, à la résistivité ρ du matériau et inversement proportionnelle à la section S du fil.
- Si un fil est purement résistif, la différence de potentiel U aux bornes du fil est proportionnelle à la résistance R de ce fil et à l'intensité I du courant qui le traverse. C'est la loi d'Ohm.
- Un sandwich Conducteur-Isolant-Conducteur réalise un condensateur. Sa capacité C augmente avec la surface S des armatures conductrices et diminue avec leur écartement. Elle varie selon les caractéristiques électriques du matériau isolant.
- La charge Q emmagasinée dans un condensateur est proportionnelle à la capacité C du condensateur et à la différence de potentiel U aux bornes du condensateur.
- La variation dQ/dt de la charge aux bornes du condensateur est l'intensité du courant de charge (ou de décharge) du condensateur.
- Si deux conducteurs sont branchés en série entre deux points, le courant doit passer dans les deux conducteurs. Les résistances s'ajoutent.
  - Dans le mécanisme du *pont diviseur* si deux résistances de valeurs R1 et R2 sont connectées en série entre deux points reliés à des potentiels Va et 0, le point situé entre les deux résistances est à un potentiel $V = Va \times R1/(R1 + R2)$.
  - Si deux conducteurs sont branchés en parallèle entre deux points, le courant passe en partie par un conducteur, en partie par l'autre, selon leurs résistances. Les conductances (inverse de résistances) s'ajoutent.
- Si un condensateur chargé, de capacité C, est mis en situation de se décharger à travers un conducteur de résistance R, il se décharge. La variation de tension est décrite par une exponentielle en $e^{-t/RC}$. Le temps de décharge est d'autant plus grand que R et C sont grands. Le phénomène de charge est symétrique.
- Une diode, constituée d'une zone dopée N et d'une zone dopée P, ne laisse passer le courant que dans un sens.

2.2 Le transistor à effet de champ M.O.S.

2.2.1 Description physique du principe du transistor à canal N

Observons la figure 7.1. Dans un substrat de silicium (variété monocristalline, faiblement dopée P) on délimite deux zones fortement dopées

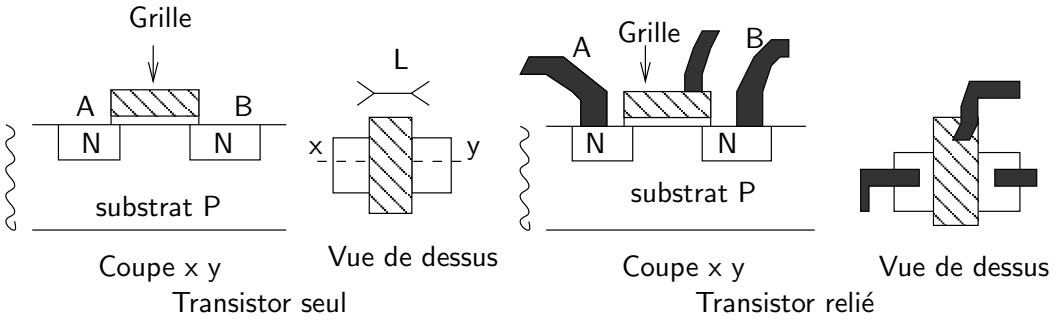


FIG. 7.1 – Coupe et vue de dessus d'un transistor seul ou relié

Négativement. Ces deux zones sont espacées d'une distance L. La zone faiblement dopée P est nommée *substrat*. Sur la zone rectangulaire entre les deux zones dopées, on fait croître du dioxyde de silicium : le verre (isolant). Au-dessus du verre on dépose du silicium (polycristallin) et on le dope aussi.

> **Remarque :** La réalité de fabrication est différente : en fait, le dopage du silicium monocristallin du substrat et celui du silicium polycristallin au-dessus de l'oxyde pourraient être simultanés : la couche de dioxyde de silicium bloque la diffusion.

On obtient ainsi deux sandwiches. L'un vertical :

Conducteur – Isolant – Semi-conducteur

et l'autre horizontal :

Semi-conducteur dopé – Semi-conducteur – Semi-conducteur dopé.

Le premier est à l'origine du nom Métal Oxyde Semi-conducteur. Sur la figure 7.1, les zones dopées du substrat sont notées A et B. On appelle *grille* la zone de silicium polycristallin dopé. L'isolant est sous la grille. Les deux zones A et B sont ici supposées rectangulaires pour faciliter le dessin. La distance L entre les deux zones est caractéristique d'une technologie de réalisation. Si le journal annonce la sortie d'un nouveau circuit en technologie 0,17 micron, cela donne la distance L pour les transistors.

2.2.2 Comportement électrique

Supposons que le substrat est relié à la masse et que les tensions sont telles que $V_{substrat} \leq V_A < V_B$. Nous appellerons B le drain et A la source.

Si la tension de grille est nulle, entre le drain et la source, se trouvent deux jonctions NP orientées en sens inverse l'une de l'autre. Or une jonction a pour propriété de ne conduire le courant que dans le sens N vers P. La jonction drain-substrat bloque donc le passage du courant entre le drain et la source : le transistor est bloqué.

Lorsqu'une tension positive est appliquée sur la grille, le champ électrique entre la grille et le substrat attire sous la grille et concentre en surface les électrons libres du substrat (et repousse les trous en profondeur). En sur-

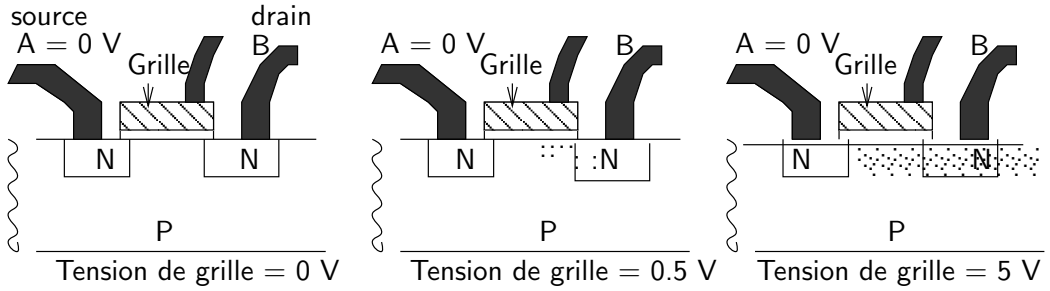


FIG. 7.2 – Formation du canal dans un transistor

face, *tout se passe* alors *comme s*'il existait sous la grille un canal drain-source de faible profondeur, artificiellement dopé négativement par l'accumulation d'électrons due au champ électrique grille-substrat. Ce canal est conducteur et un courant peut y circuler. L'intensité du courant est soumise à la loi d'Ohm : la *résistance* du canal entre source (A) et drain (B) est fonction de la longueur et de la section du canal mais aussi de la résistivité du semi-conducteur obtenu.

Cette résistivité diminue à mesure que la différence de potentiel entre la grille et le substrat augmente. Le transistor fonctionne donc comme une résistance commandée par la différence de potentiel grille-substrat.

Cet effet de conduction dû à un champ électrique a donné le nom de transistor à effet de champ.

Une modélisation plus fine du transistor met en évidence une limite du transistor : *la tension du drain et la source doit être inférieure à celle de la grille faute de quoi le canal* ne peut se former. Nous appellerons Vgs_{th} la différence de potentiel minimale entre grille et source nécessaire à la formation du canal.

La figure 7.2, dans laquelle les petits points représentent des électrons, suggère la formation du canal.

Par rapport au substrat la grille du transistor se comporte comme une capacité. Quand la capacité est chargée, elle est au potentiel d'alimentation, quand elle ne l'est pas, la grille est au potentiel de la masse.

2.2.3 Le transistor à canal P

Il est obtenu de façon assez symétrique du transistor à canal N. Le dopage est fait par du bore qui a 3 électrons sur la *couche externe*. Le dopage est *P*ositif : des trous sont apparus. Le substrat faiblement dopé N est relié au potentiel positif d'alimentation, typiquement 5 volts. Le canal se forme si le potentiel sur la grille est suffisamment plus petit que celui du substrat.

On remarque la difficulté de cohabitation sur un même substrat de silicium d'un transistor N avec un substrat P à la masse et d'un transistor P avec un substrat N relié à l'alimentation. C'est pourtant ce que l'on cherche à faire en technologie CMOS, où les deux types de transistors cohabitent. La technologie

de réalisation brièvement décrite au paragraphe 5.2 en tient compte.

2.2.4 Remarque finale à propos d'électricité

Le fonctionnement du transistor, N ou P, tel qu'il vient d'être décrit est très continu : une petite variation d'un des paramètres induit une petite variation de comportement. Le transistor ne passe pas brutalement de conducteur à non conducteur. Tout changement de l'épaisseur d'oxyde, de la longueur du canal du transistor, de la différence de potentiel entre la grille et le substrat ou entre les points A et B donne une variation de l'intensité du courant de façon continue.

La mise en équation des phénomènes physiques mis en jeu dans le transistor MOS est traitée dans les livres d'électronique (Par exemple [CDLS86]) et n'a pas sa place ici.

3. Phénomènes à l'échelle logique

Dans l'algèbre booléenne, les deux éléments significatifs sont codés 0 et 1. Avec deux symboles, interprétés comme des chiffres 0 et 1, la numération en base 2 permet de représenter les nombres. Les dispositifs à transistors ont un comportement continu : toute variation *infinitésimale* des entrées provoque une variation *faible* des sorties (courant, tension...). La question est de savoir comment représenter des informations numériques avec des dispositifs ainsi continus.

Il existe des calculateurs dits *analogiques*. Le principe est simple : le nombre 470 est représenté par la tension 4,7 volts, le nombre 32 est représenté par 0,32 volts. Un circuit additionneur est un dispositif à deux entrées, capable de délivrer sur la sortie la somme, ici 5,02 volts, des tensions. Ces machines sont difficiles à calibrer si l'on souhaite une précision dans les calculs de plus de 4 chiffres décimaux significatifs.

Par opposition aux calculateurs analogiques, les calculateurs les plus fréquents sont *numériques*, ou *digitaux*. Les nombres sont représentés par des vecteurs de booléens, ou vecteurs de *bits*.

3.1 L'abstraction logique

Les valeurs 0 et 1 d'un *bit* sont représentées par des tensions, respectivement nulle (0 volt ou masse) et la tension d'alimentation, standardisée à 5 volts (de plus en plus souvent 3,3 volts, voire moins, notamment dans les machines portables).

Les transistors sont fabriqués de telle façon qu'il existe une tension de seuil ("threshold" en anglais) V_{th} au-dessus de laquelle l'entrée d'un circuit interprétera le signal comme un 1, et comme un 0 au-dessous. La valeur nominale de V_{th} est choisie de manière à optimiser la tolérance aux bruits et parasites

Type de transistor	Tension de commande	Comportement
Canal N	Alimentation	Passant
	Masse	Bloqué
Canal P	Masse	Passant
	Alimentation	Bloqué

FIG. 7.3 – Comportement des transistors

électriques pouvant affecter le signal. Compte tenu des tolérances de fabrication sur la valeur de V_{th}, tous les circuits interpréteront une tension inférieure a 0,75 volts comme un 0 et supérieure à 4,5 volts comme un 1.

On parle de niveaux 0 logique et 1 logique, ou de *niveaux logiques* bas et haut. En logique négative le niveau haut correspond au 0 et le niveau bas au 1. Nous ne prendrons pas cette convention.

Etudions un comportement simplifié du transistor. Cette simplification consiste à faire comme si le canal du transistor était soit totalement bloqué soit passant, auquel cas il a une résitance R. Nous ne regardons les transistors que reliés soit à la masse, (le potentiel 0 volt), soit à un potentiel positif, la tension d'alimentation. En réalité les tensions électriques varient de façon continue, et parfois il y a des parasites.

Pour un transistor à canal N avec le substrat à la masse :

– Si la grille est à l'alimentation, le transistor est passant. S'il y a une différence de potentiel entre A et B, du courant circule entre A et B.
– Si la grille est à la masse, le transistor est bloqué. Même s'il y a une différence de potentiel entre A et B, aucun courant ne circule entre A et B.

Pour un transistor à canal P, avec le substrat à l'alimentation, le fonctionnement est inversé :

– Si la grille est à à la masse, le transistor est passant. S'il y a une différence de potentiel entre A et B, du courant circule entre A et B.
– Si la grille est à l'alimentation, le transistor est bloqué. Même s'il y a une différence de potentiel entre A et B, aucun courant ne circule entre A et B.

Ces différentes fonctions sont regroupées dans le tableau de la figure 7.3. Ce comportement simplifié fait abstraction de nombreux phénomènes. On parle d'*abstraction logique*.

3.2 Réalisation de la fonction logique la plus simple : l'inverseur

Les fonctions logiques peuvent être modélisées simplement par des fonctions booléennes. La réalisation matérielle de l'opération booléenne de complémentation s'appelle un *inverseur*. L'inverseur peut être un montage électrique ayant une entrée E et une sortie S (L'algèbre de Boole ne tient

évidemment pas compte de l'existence de l'alimentation et de la masse dans les montages électriques).

On fait abstraction des valeurs exactes des tensions électriques en disant : Si E = 0, alors S = 1 et si E = 1 , alors S = 0.

En réalité, comme on va le voir, si 0 <= E <= 0,75 volts, alors S = 5 volts et si 4,5 <= E <= 5 volts, alors S = 0 volt.

Cela nous donne les points extrêmes de fonctionnement d'un inverseur. Mais quel peut être le comportement souhaitable de l'inverseur entre ces deux extrêmes ? Pour répondre à cette question imaginons deux montages : l'un constitué de 4 inverseurs en série : la sortie de l'un est l'entrée du suivant. Dans l'autre les deux inverseurs sont rétrocouplés : la sortie de l'un est l'entrée de l'autre et réciproquement. Les schémas simplifiés correspondants sont donnés figure 7.4. L'inverseur y est représenté par une simple boîte avec une entrée e et une sortie s.

Dans le montage de 4 inverseurs en série, envoyons un signal d'entrée qui en fonction du temps passe de 0 à 5 volts (la représentation en escalier n'étant pas à prendre au pied de la lettre).

Examinons les sorties après 2, ou 4, inversions pour trois types d'inverseurs. Les trois sont candidats au titre du *meilleur* inverseur.

Pour les trois types, nommés Accroissement, Maintien, Diminution, nous donnons une courbe de transfert donnant la tension de sortie de l'inverseur en fonction de la tension d'entrée (Figure 7.5). Pour les trois types les valeurs extrêmes des tensions sont respectées, et il existe une tension médiane V_M pour laquelle la tension de sortie est égale à la tension d'entrée. Mais le comportement entre ces points est différent.

Pour les trois types nous donnons l'allure de la réponse du montage constitué de 2 ou de 4 inverseurs à l'entrée en escalier.

L'inverseur de type Accroissement accentue les différences entre les niveaux faibles et forts. C'est un amplificateur. Si il y a un faible parasite en entrée, le parasite n'est pas apparent en sortie.

A l'inverse l'inverseur de type Diminution diminue les différences entre niveaux faibles et forts. Dès que le signal d'entrée présente un parasite, le niveau de sortie risque d'être autour de V_M, ce qui n'est pas souhaitable.

De même, pour le montage constitué de deux inverseurs rétrocouplés, on comprend aisément que l'inverseur de type Accroissement donne un système stable. Soit une sortie vaut 1 et l'autre 0, soit le contraire, mais une stabilisation à un état intermédiaire est très improbable (quoique pas impossible). Avec un inverseur du type Diminution, on pourrait facilement obtenir un montage de deux inverseurs rétrocouplés se stabilisant avec des sorties à V_M.

C'est évidemment l'inverseur de type Accroissement qui est le plus intéressant pour la réalisation de fonctions booléennes.

L'électronique digitale étudie de façon précise COMMENT obtenir un montage ayant la bonne courbe de transfert. Elle permet aussi d'étudier une réalisation avec un point de basculement V_m proche de la moitié de la ten-

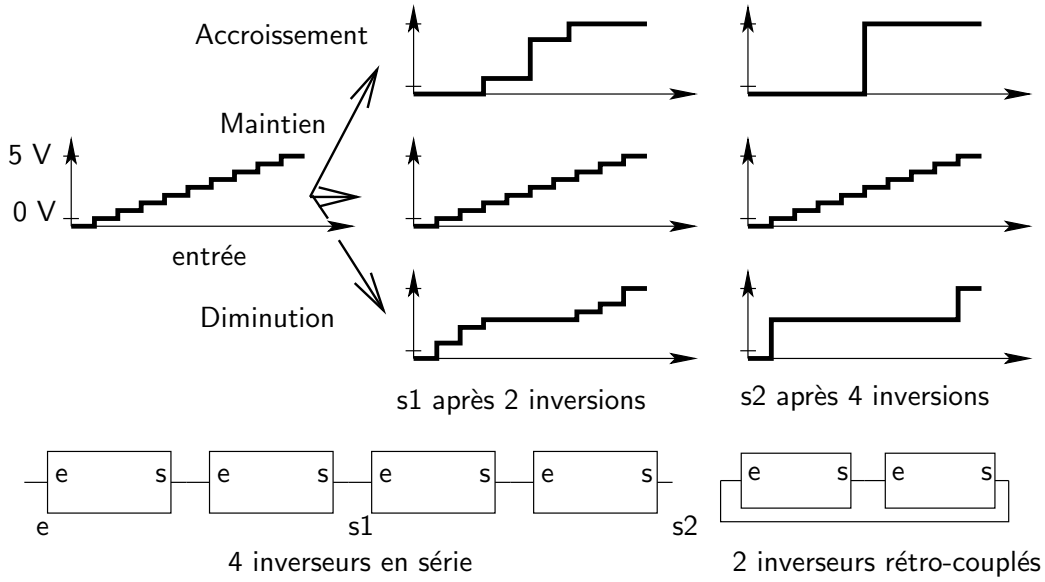


FIG. 7.4 – Comportement des candidats inverseurs après 2 ou 4 inversions

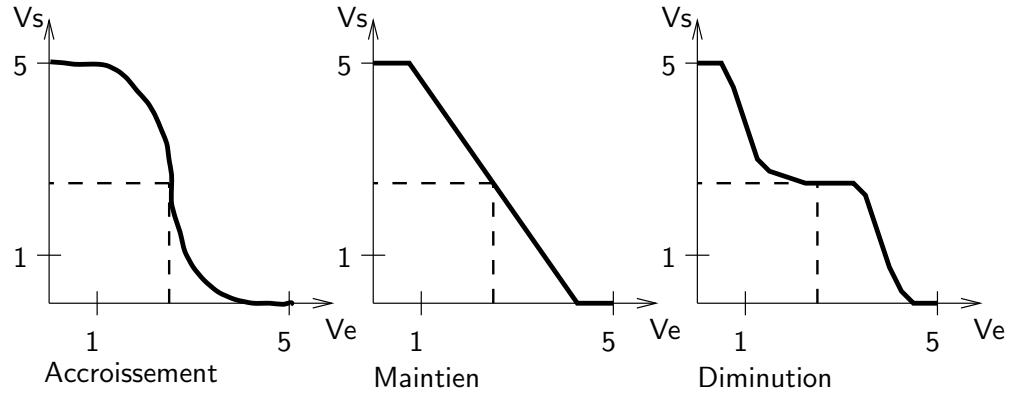


FIG. 7.5 – Courbes de transfert de trois candidats inverseurs

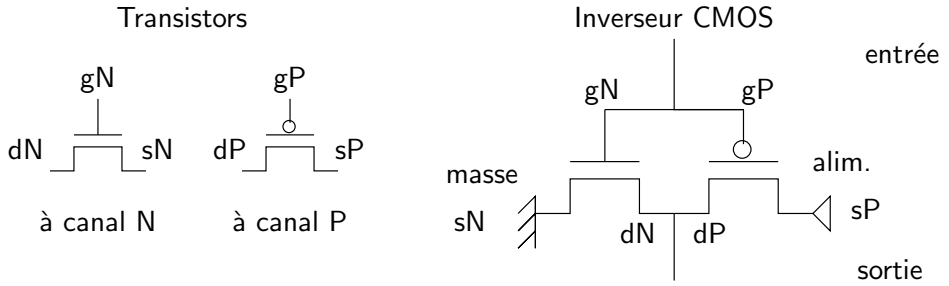


FIG. 7.6 – Schéma conventionnel des transistors MOS et de l'inverseur CMOS. d,g,s sont respectivement les Drains, Grilles et Sources. N et P désignant les transistors N et P.

Tension d'entrée Ve	Résistance du transistor N R_N	Résistance du transistor P R_P	Tension de sortie Vs
0 V <= Ve <= 0.75 V	infinie	R1	5 V
4.5 V <= Ve <= 5 V	R2	infinie	0 V

FIG. 7.7 – Réponse de l'inverseur CMOS

sion de référence, ce qui donne une bonne tolérance aux parasites ([CDLS86]).

Nous donnons ici *deux* solutions : la principale est la technologie à base de transistors MOS Complémentaires (Complementary MOS). La figure 7.6 donne la structure interne de l'inverseur. En technologie CMOS, l'inverseur est obtenu en connectant un transistor N et un transistor P en série entre la masse et l'alimentation. L'entrée est la tension de grille commune aux deux transistors et la sortie est le point intermédiaire entre les deux. Les substrats respectifs des deux transistors N et P sont à la masse et à l'alimentation.

Si l'on considère les différentes valeurs possibles pour la tension d'entrée Ve, on obtient le tableau de la figure 7.7, où R_N désigne la résistance du transistor à canal N, R_P désigne la résistance du transistor à canal P, Vs désigne la tension de sortie, égale, dans tous les cas, à $V_{\text{ref}} \times R_N/(R_P + R_N)$ ou $V_{\text{ref}} \times 1/(1 + R_P/R_N)$, où V_{ref} désigne la tension d'alimentation.

Une autre solution est de remplacer le transistor P par une résistance, mais la courbe de transfert est moins intéressante. Le principal avantage est une plus grande simplicité (Inverseur NMOS). On utilisera dans le livre certaines portes basées sur ce principe.

Il existe d'autres organisations d'inverseurs. Beaucoup des circuits de petite échelle d'intégration (quelques centaines de transistors par puce) sont dans une technique nommée Transistor Transistor Logic TTL. Mais la plupart des circuits de très grande échelle d'intégration sont aujourd'hui en CMOS.

3.3 Fonctionnements statique et dynamique de l'inverseur

L'analyse du fonctionnement *statique* de l'inverseur a été faite précédemment. Si l'entrée est stable à 1, la sortie est stable à 0. Si l'entrée est stable à 0, la sortie est stable à 1.

L'étude du comportement *dynamique* de l'inverseur concerne le comportement lors des changements de tension d'entrée. Elle doit prendre en compte *où* est connectée la sortie de cet inverseur. La réponse est simple : à des grilles de transistors, en entrée d'autres circuits logiques. Elles sont donc capacitives par rapport à la masse. On assimilera donc la sortie de l'inverseur à une capacité.

Que se passe-t-il lors du changement de la tension d'entrée ? Lors d'un front montant, où la tension d'entrée passe de 0 à 5 volts, la sortie doit passer de 5 à 0 volts. La capacité reliée à la sortie doit se décharger, vers la masse, à travers le transistor N. Symétriquement, lors d'un front descendant, la capacité de sortie doit se charger à travers le transistor P.

Cette charge ou cette décharge se fait en un certain temps. Ce temps constitue le temps de basculement, ou temps de réponse, de l'inverseur. Ce temps est couramment inférieur à la nanoseconde (10^{-9}s). Il dépend fortement de la valeur de la capacité. L'existence de ce temps de changement de la sortie de l'inverseur explique pourquoi tous les circuits logiques ont un temps de réponse.

4. Circuits logiques

Nous allons étudier dans la suite différents assemblages de transistors réalisant des fonctions booléennes. Ces assemblages seront classés en deux catégories :

- Les assemblages qui ne mémorisent pas l'information, que l'on nomme circuits combinatoires,
- Les assemblages qui mémorisent de l'information, que l'on nomme circuits séquentiels.

> *Les différences entre ces deux types de circuits sont difficiles à comprendre. On y revient dans la suite du livre. Chacune des deux familles fait l'objet d'un chapitre entier (Cf. Chapitres 8 et 10).*

4.1 Assemblages combinatoires

Les réalisations des fonctions booléennes s'appellent des *portes logiques*. La figure 7.9 donne la structure de différentes portes logiques. L'inverseur est une porte logique à une entrée et une sortie. Il est déjà connu (figure 7.9-a).

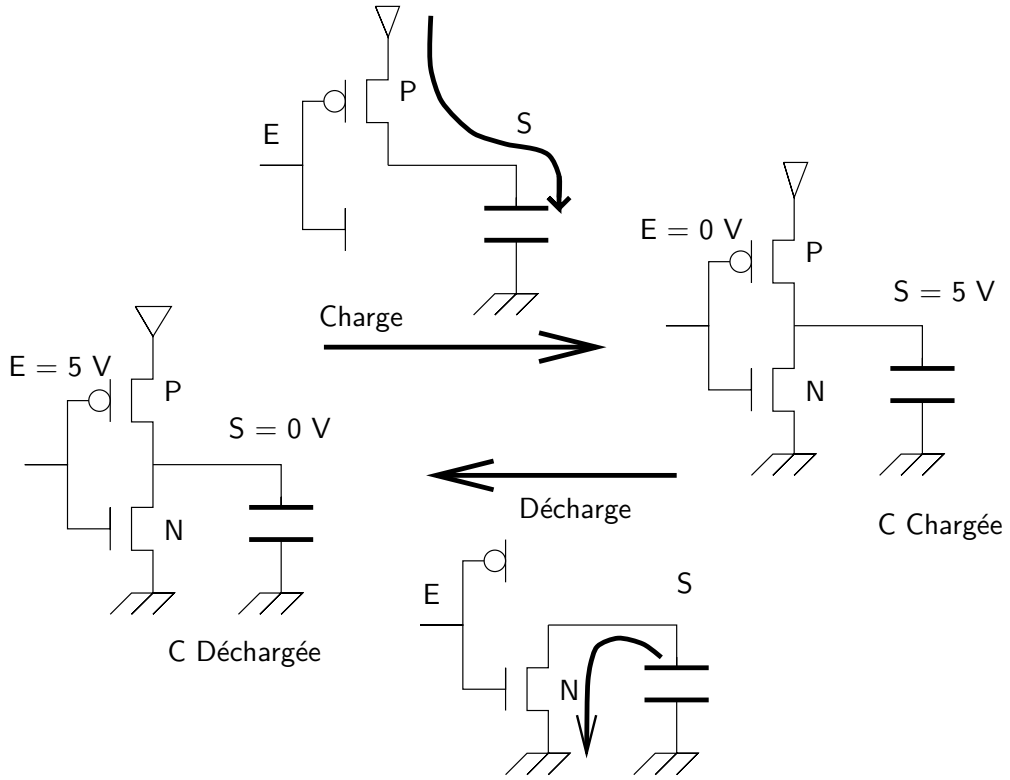


FIG. 7.8 – Décharge et charge de la capacité de sortie d'un inverseur

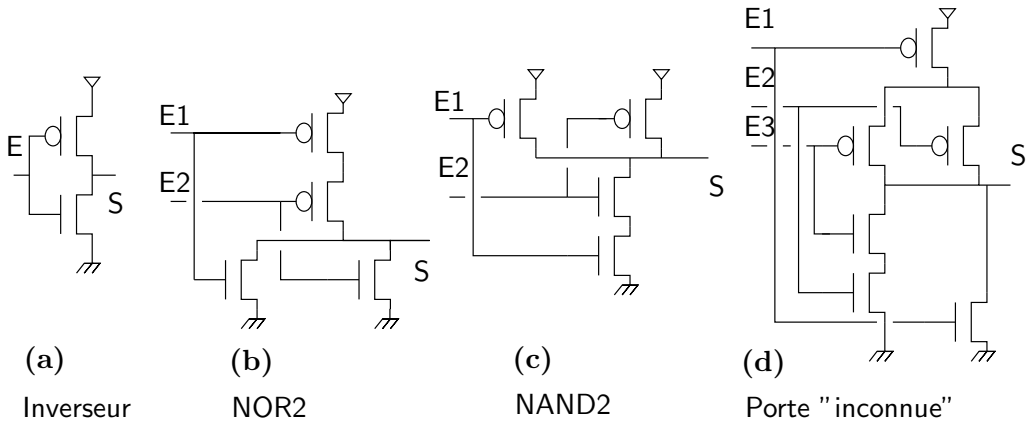


FIG. 7.9 – Structure interne de différentes portes de base

4.1.1 La porte NOR à deux entrées E1 et E2

Le montage comporte deux transistors N et deux transistors P. Les deux transistors N sont en parallèle entre la masse et la sortie, les deux transistors P sont en série entre l'alimentation et la sortie. Les grilles d'un transistor N et d'un transistor P sont connectées à l'entrée E1, les grilles de l'autre transistor N et de l'autre transistor P sont connectées à l'entrée E2. La sortie est donc reliée à la masse, via une résistance passante, si et seulement si une au moins des deux entrées est au niveau de l'alimentation. La fonction logique est un NOR à deux entrées. On le note parfois NOR2 (figure 7.9.-b).

4.1.2 La porte NAND à deux entrées E1 et E2

Le montage comporte deux transistors N et deux transistors P. Les deux transistors N sont en série entre la masse et la sortie, les deux transistors P sont en parallèle entre l'alimentation et la sortie. Les grilles d'un transistor N et d'un transistor P sont connectées à l'entrée E1, les grilles de l'autre transistor N et de l'autre transistor P sont connectées à l'entrée E2. La sortie est donc reliée à la masse, via une résistance passante, si et seulement si les deux entrées sont au niveau de l'alimentation. La fonction logique est un NAND à deux entrées. On le note souvent NAND2 (figure 7.9-c).

4.1.3 Les portes à plus de deux entrées

De façon très similaire au montage des portes NAND ou NOR à deux entrées, on obtient une porte NOR à trois entrées avec 3 transistors N en parallèle entre la masse et la sortie et 3 transistors P en série entre la sortie et l'alimentation. De même on obtient une porte NAND à trois entrées avec 3 transistors N en série entre la masse et la sortie et 3 transistors P en parallèle entre la sortie et l'alimentation.

Le procédé semble pouvoir continuer. N'oublions pas que les résistances des transistors passants en série s'ajoutent. Certaines techniques de réalisation vont imposer des contraintes d'origine électrique au nombre maximal d'entrées des portes.

4.1.4 Une porte *inconnue* à trois entrées

Le montage de la figure 7.9-d est particulier en ce sens qu'il ne réalise pas une fonction logique élémentaire comme le NAND ou le NOR. On vérifie aisément que S vaut 0 si et seulement si E1 vaut 1 ou si E2 et E3 valent 1. On a donc $S = \overline{E1 + E2.E3}$.

4.1.5 Deux inverseurs en parallèle : un petit et un gros

Si l'on connecte ensemble les sorties de deux inverseurs, que se passe-t-il ? Si les deux inverseurs ont la même entrée, donc la même sortie, il n'y a rien de spécial à dire, le comportement global est celui d'un seul inverseur.

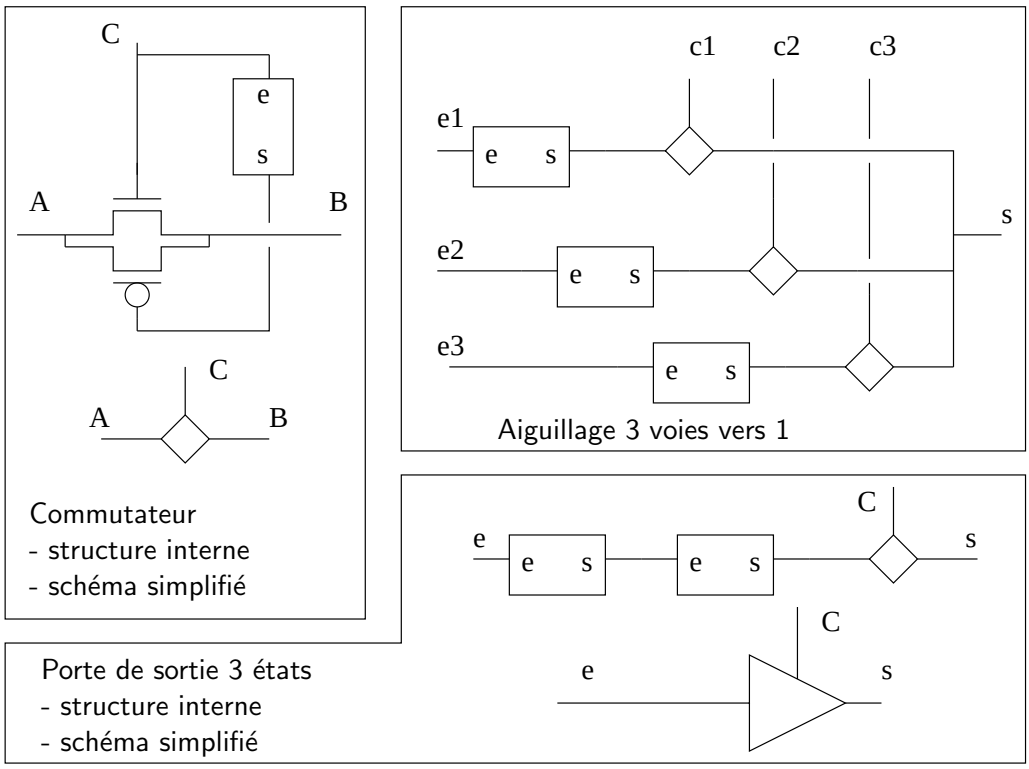


FIG. 7.10 – Commutateur et aiguillage à base de commutateurs

Si les deux inverseurs ont des entrées différentes, on se trouve avec deux montages de ponts diviseurs de résistances en parallèle. Si les deux inverseurs ont des résistances très proches, la tension de sortie est autour de 2,5 volts. Si l'un des deux inverseurs a des transistors dont les résistances sont très différentes de l'autre, un des deux inverseurs peut prendre l'avantage, la tension de sortie prenant des valeurs de 2 ou 3 volts, par exemple. Ce type de fonctionnement échappe évidemment au domaine des circuits combinatoires bien construits...

Dans certaines technologies le pont diviseur obtenu laisse passer beaucoup de courant, ce qui produit de l'énergie thermique. C'est le court-circuit.

4.1.6 Le commutateur ou interrupteur 3 états

La source d'un transistor MOS peut ne pas être connectée au même potentiel que le substrat. On obtient alors entre drain et source un demi-interrupteur électronique commandé par la tension de grille qui permet de réaliser des connexions intermittentes entre d'autres éléments. (Cf. Figure 7.10)

La polarité du signal de commande qui établit la connexion dépend du type de transistor (1 pour un transistor à canal N, 0 pour un transistor à canal P). Pour que les niveaux logiques 0 et 1 soient tous les deux bien transmis, on utilise une paire de transistors N et P. On a alors un interrupteur complet, ou

```
s = si     c1 et non c2 et non c3 alors non e1 sinon
    si non c1 et     c2 et non c3 alors non e2 sinon
    si non c1 et non c2 et     c3 alors non e3 sinon
    indéfini
```

FIG. 7.11 – Comportement de l'aiguillage

commutateur. Le commutateur est d'usage fréquent. Il est symbolisé par un simple carré incliné.

4.1.7 L'aiguillage

Observons la figure 7.10. Dans l'aiguillage 3 voies vers 1 réalisé à base de 3 commutateurs et de 3 inverseurs, on dit que la sortie s constitue une connexion de type *bus*. les complémentaires des trois signaux e1, e2, e3 peuvent être envoyés sur le bus. La sortie s est décrite Figure 7.11. Cette sortie est parfois indéfinie.

Le cas indéfini est complexe ; il y a deux sous-cas. Si l'on trouve deux ou trois sorties égales sur le bus, cela ne pose pas de problème, et s prend cette valeur. Si l'on trouve des sorties différentes, selon les forces respectives des inverseurs qui traitent e1, e2 et e3, s reçoit une valeur non *booléenne*. Ce montage ne doit donc s'utiliser qu'avec la garantie que seulement l'un parmi c1, c2, c3 vaut 1.

4.1.8 L'amplificateur 3 états

Observons la figure 7.12. L'ensemble inverseur (qui est aussi un amplificateur) suivi d'un commutateur est parfois réalisé en un seul circuit nommé amplificateur/inverseur 3 états. Le montage constitué de deux inverseurs suivis d'un commutateur reçoit le nom de porte de sortie 3 états. On peut dire aussi que c'est un inverseur suivi d'un amplificateur/inverseur. Ce montage est utilisé dans beaucoup de dispositifs comprenant des bus. Nous en utiliserons dans les chapitres relatifs aux entrées/sorties. On donne aussi une autre réalisation de la porte 3 états. Dans ces différents montages, les résistances des transistors quand ils sont passants doivent être ajustées soigneusement.

4.1.9 La gare de triage

Par généralisation des aiguillages, et utilisation de portes de sortie 3 états, le montage de la figure 7.13 permet de relier 4 points A, B, C et D.

Les liaisons entre A, B, C et D sont fonctions de signaux de commande de sortie aout, bout, cout et dout et de signaux de commande d'entrée ain, bin, cin et din. En fonctionnement normal, un seul des signaux de commande de sortie doit valoir 1. Plusieurs des signaux de commande d'entrée peuvent valoir 1. Par exemple quand aout et cin et din valent 1 et que tous les autres valent 0,

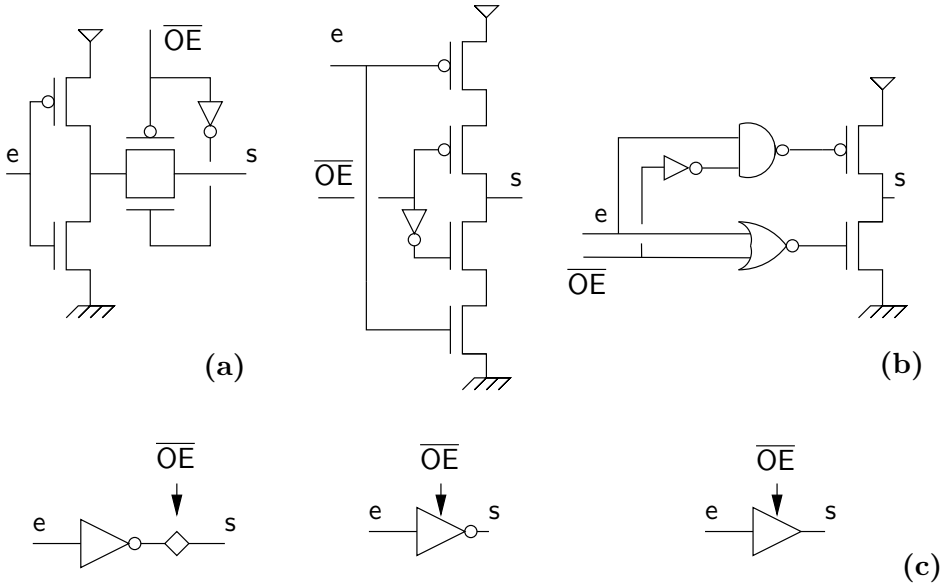


FIG. 7.12 – Deux réalisations de l'amplificateur/inverseur 3 états et leurs symboles. L'inverseur est représenté par un triangle avec un petit rond, les 2 portes sont un nand2 et un nor2. (a) schémas en transistors. (b) schémas en transistors et portes. (c) symboles logiques.

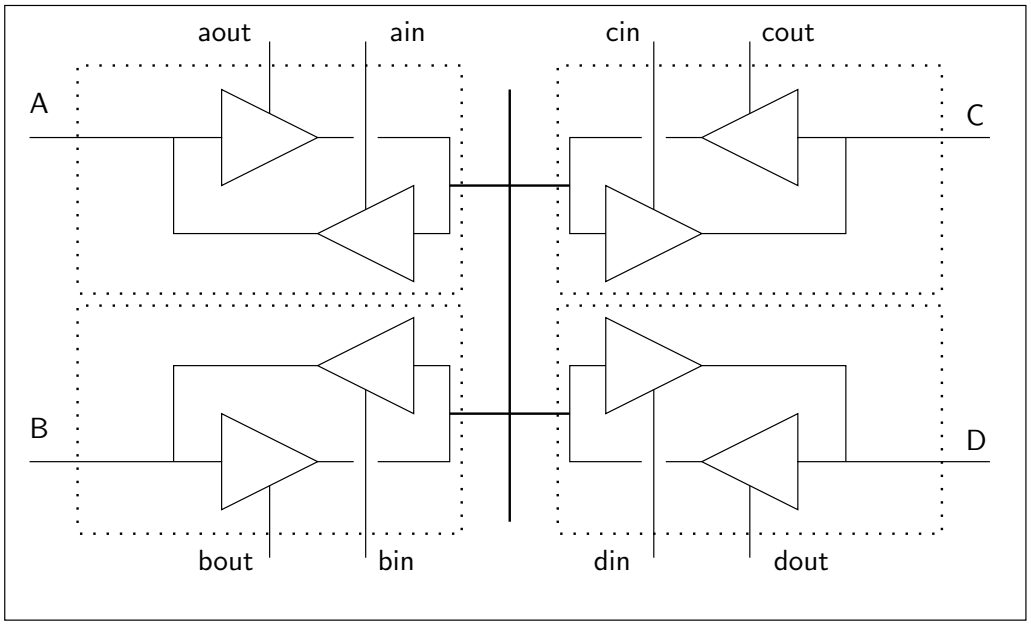


FIG. 7.13 – Ensemble d'aiguillages constituant une gare de triage.

la valeur de A est transmise sur C et D. Cette transmission est directionnelle. Ce type de structure est utilisée dans les interconnexions entre une mémoire et plusieurs processeurs par exemple (Cf. Chapitre 16). La liaison centrale (en gras sur la figure), par laquelle tout passe, est le bus du système.

4.2 Assemblages séquentiels

4.2.1 Introduction

La réalisation de dispositifs séquentiels permet de stocker l'information. Cela signifie piéger la valeur d'un signal électrique à un instant donné dans un dispositif où cette valeur restera stable.

Pour comprendre cette mécanique commençons par voir comment une valeur peut être mémorisée de façon stable. Pour cela examinons le rétrocouplage de deux inverseurs déjà rencontré pour l'étude de l'inverseur.

4.2.2 Le bistable

Observons la figure 7.14-a. Le montage de deux inverseurs, chacun ayant comme entrée la sortie de l'autre, a trois points de fonctionnement :

– l'entrée du premier est au niveau logique 1, sa sortie au niveau logique 0.
– à l'inverse, c'est le deuxième inverseur qui a l'entrée à 1 et la sortie à 0 (ces deux états sont stables, le montage s'appelle un *bistable* car il a deux points de fonctionnement stable).
– l'entrée et la sortie des deux inverseurs sont à 2,5 volts. Cet état est instable, le moindre parasite sur une des deux connexions est amplifiée et le système *tombe* dans un des deux états stables. C'est pour cela qu'on a choisi des inverseurs de type Accroissement (on parle souvent de méta-stabilité pour cet état).

Il reste un problème : ces deux états sont tellement stables qu'on ne voit pas comment y piéger une nouvelle valeur. Résoudre ce problème permet de réaliser une mémoire vive de 1 mot de 1 bit.

4.2.3 Les points mémoire de type mémoire vive

Le point mémoire élémentaire, telle qu'il est réalisé dans les boîtiers de mémoire vive statique, permet de piéger une valeur nouvelle dans un bistable (figure 7.14-b). Si la commande de forçage F sur les 2 transistors de part et d'autre du bistable est active et si aux extrémités du dispositif on présente une valeur V1 et son complément V2 à l'aide de circuits plus puissants que les deux inverseurs utilisés dans le montage bistable, ces deux valeurs restent en place quand la commande F sur les deux transistors redevient inactive.

C'est un point de mémorisation dans lequel on peut donc écrire. La lecture se fait simplement en récupérant les valeurs du bit mémorisé et de son

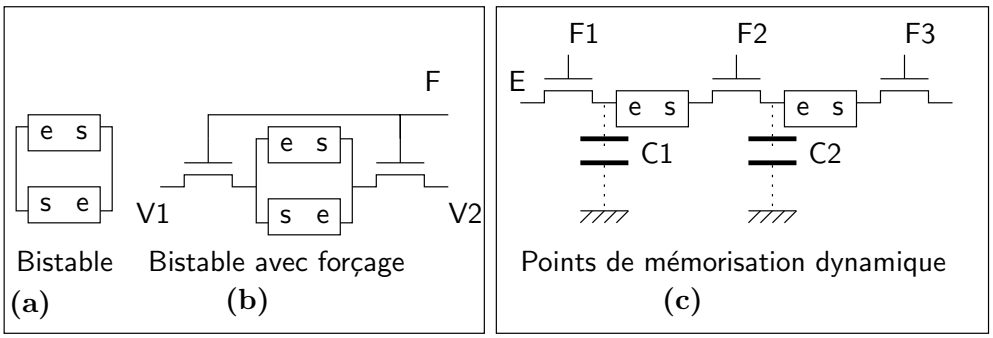


FIG. 7.14 – Les différents points mémoire

complémentaire en sortie des deux inverseurs. Le chapitre 9, consacré aux éléments de mémorisation, reprend ces points en détail.

4.2.4 Les points de mémorisation dynamiques

Il existe des points mémoire utilisant un principe tout à fait différent : c'est la capacité d'entrée d'un inverseur, ou d'une porte, qui piège la valeur (Figure 7.14-c).

La mémorisation a lieu pendant que les signaux F1 et F2 sont inactifs. Une nouvelle valeur est introduite dans la capacité du premier inverseur par la mise à 1 de F1. Cette mémorisation dans la capacité de grille d'un transistor ne peut être de longue durée. Elle est en revanche très intéressante car elle n'utilise pas de circuits en plus des fonctions combinatoires. Si l'on veut transférer l'information d'un point à un autre on utilise F2. Il convient dans ces points de mémorisation dynamique de ne pas avoir simultanément F1 et F2 actifs.

4.2.5 L'oscillateur

Le circuit oscillateur présenté figure 7.15-a oscille. Le montage d'un quartz de fréquence de résonnance donnée permet de rendre la fréquence d'oscillation constante. Ce montage fabrique un signal périodique, carré. Il est utilisé comme horloge. Dans la suite du livre nous utiliserons ces primitives de réinitialisation (Reset) et d'horloge pour les réalisations de circuits séquentiels.

4.2.6 Le monostable

Lors de l'appui sur le bouton Reset d'un ordinateur, une impulsion est générée, puis elle disparaît. Un montage comme celui de la figure 7.15-b assure cette fonctionnalité.

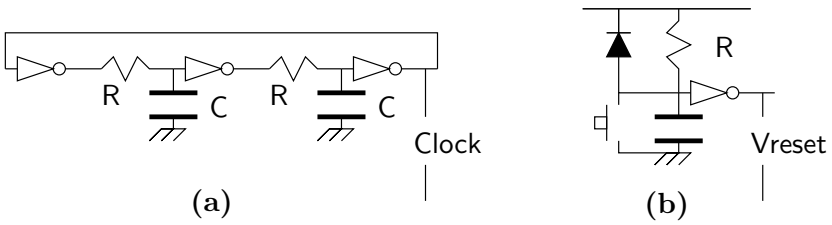


FIG. 7.15 – (a) Oscillateur délivrant une horloge. (b) Système de réinitialisation délivrant une tension de Reset.

5. Fabrication des dispositifs

Les dispositifs à base de transistors sont réalisés dans des chaînes de fabrication de haute technologie. Pour simplifier la présentation, supposons que le monde comporte deux catégories : les fabricants eux-mêmes et les clients. Si le client trouve dans les catalogues des fabricants le circuit dont il a besoin, déjà conçu, il l'achète et l'utilise. Sinon le client conçoit un circuit pour une application et le fait fabriquer, puis il intègre le circuit dans un assemblage, généralement logiciel et matériel. Cet assemblage peut être un ordinateur, un magnétoscope ...

Un circuit conçu spécialement pour une application s'appelle un A.S.I.C. (Application Specific Integrated Circuit). Le terme s'oppose aux circuits standard (mémoires, processeurs, ...). La fabrication est dans ses grandes lignes la même.

Les galettes de silicium de quelques décimètres de diamètre subissent une suite de traitements physico-chimiques destinés à fabriquer les transistors. Les différentes étapes sont simultanées : toutes les grilles des cent (et quelques) millions de transistors de la galette sont réalisées en même temps par un même dépôt de silicium polycristallin. Différents dépôts font appel au procédé de photogravure que nous allons présenter.

5.1 Le procédé de photogravure

Le procédé de photogravure (Cf. Figure 7.16) permet d'obtenir un motif complexe dans un matériau A à la surface d'un matériau B.

Pour cela on dépose du A partout à la surface du B. Puis on dépose par dessus une couche d'un produit photo-sensible. On pose au-dessus un masque partiellement opaque, partiellement transparent. Les zones opaques sont appelées les noirs, les autres les blancs. Il y a des milliards de motifs de chaque couleur, notre figure n'en comporte qu'un ! On illumine le tout par au-dessus (étape 1 de la figure 7.16). Le produit photo-sensible reçoit de la lumière en face des blancs du masque et n'en reçoit pas en face des noirs du masque. Un produit chimique permet d'attaquer le produit photo-sensible là où il a reçu de la lumière et seulement là (étape 2 de la figure 7.16).

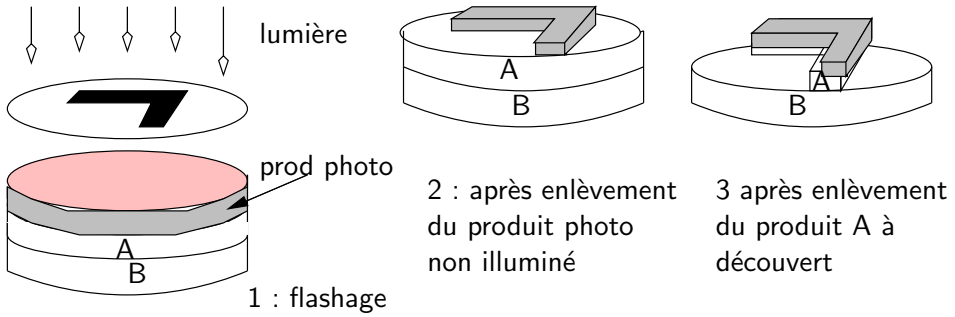


FIG. 7.16 – Trois étapes du procédé de photogravure

Après cette attaque, le matériau A est apparent en face des blancs du masque. Puis un deuxième produit chimique attaque le matériau A là où il est en surface (étape 3 de la figure 7.16). Le matériau B est alors sur le dessus en face des blancs du masque. Un troisième produit chimique attaque le produit photo-sensible là où il n'a pas été illuminé. On obtient ainsi une *forme* en matériau A à la surface d'un substrat constitué de matériau B. Cette forme est celle qui était dessinée sur le masque. La mise en oeuvre effective de ce processus de fabrication demande une haute technologie pour obtenir des dessins dont les tailles sont de l'ordre du dizième de micron.

A partir du schéma de connexions entre les transistors l'obtention des dessins des masques suppose de manipuler quelques centaines de millions de rectangles. Un outil de Conception Assistée par Ordinateur est évidemment indispensable. Les *vieillards* se souviennent avec émotion du *bon* temps où les circuits n'avaient que quelques centaines de transistors et où les crayons de couleurs et le papier quadrillé tenaient lieu d'assistance. Les schémas physiques étaient obtenus en découpant de l'autocollant noir, aux ciseaux, et en le collant sur du papier blanc. Un négatif d'une photo de la feuille de papier tenait lieu de masque.

5.2 Un procédé de fabrication : CMOS à grille silicium polycristallin et à deux niveaux de métal

La fabrication de circuits logiques organisés avec des transistors N et des transistors P nécessite de nombreuses opérations technologiques. Nous les envisageons ici à travers les dessins des masques impliqués. Les masques comportent les dessins des motifs nécessaires à la réalisation simultanée des quelques millions de transistors sur la galette de silicium d'une vingtaine de centimètres de diamètre. Nous ne dessinons ici qu'un inverseur.

La figure 7.17 montre par 6 dessins A, B, C, D, E et F les masques utilisés. Nos dessins sont simplifiés par rapport à la réalité. Sur les vues en coupe les vraies proportions ne sont pas respectées. La ligne de surface du substrat qui apparaît sur les coupes n'est pas si rectiligne. Il y a en réalité environ 15

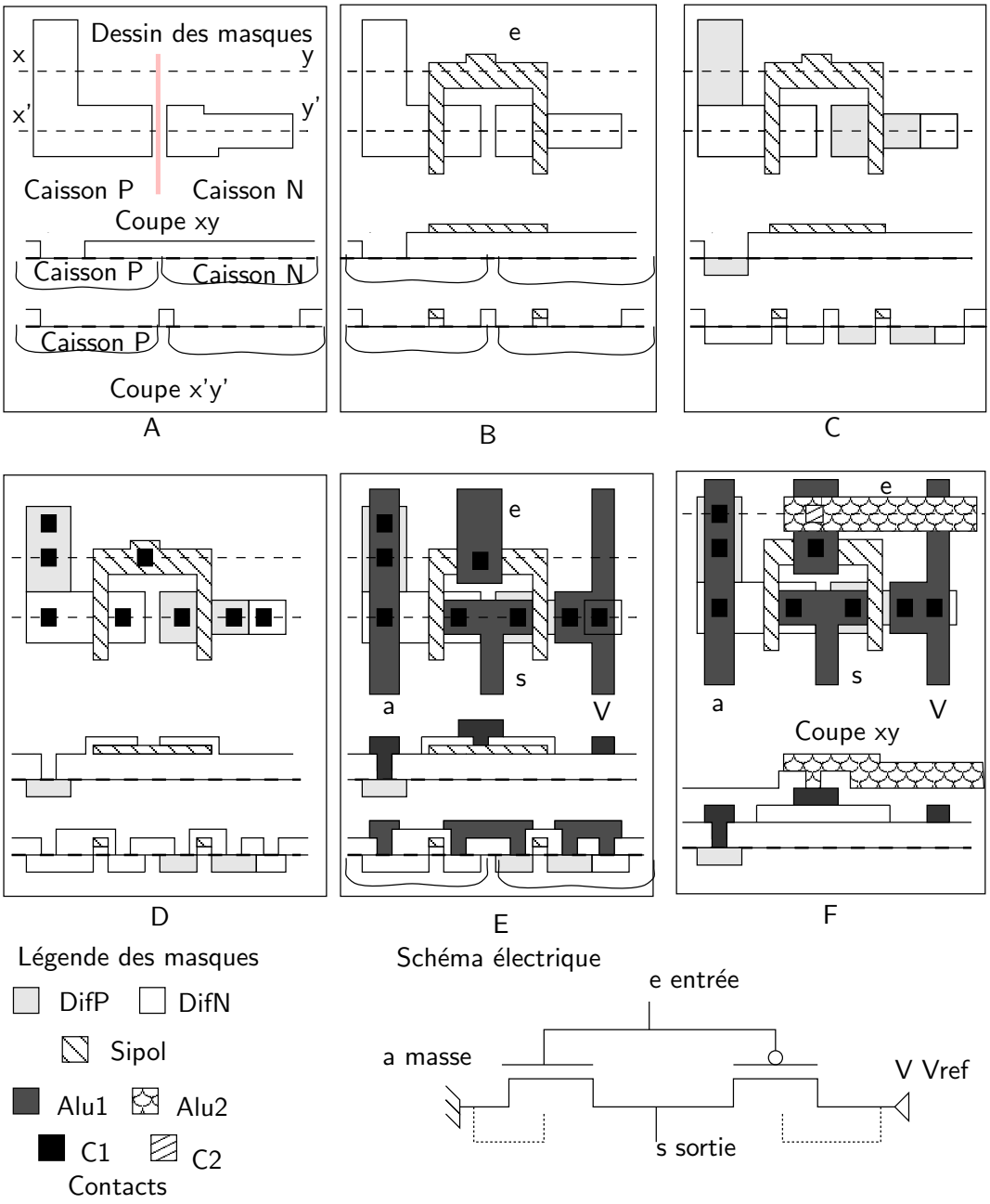


FIG. 7.17 – Les masques intervenant dans la technologie CMOS. DifP est le masque de la zone diffusée P, DifN de la zone diffusée N, Sipol est le masque du silicium polycristallin, Alu1 et Alu2 sont les deux niveaux d'aluminium, C1 et C2 sont les deux niveaux de contacts sous l'aluminium. Les masques sont cumulés dans les différentes parties de la figure. La vue en coupe après l'opération est selon la ligne xy ou la ligne x'y'. Dans les vues en coupe l'oxyde n'est pas coloré.

masques.

Décrivons les opérations principales :

1. La première consiste à doper, faiblement, un caisson P pour y réaliser des transistors N. Le caisson P existe sous toute la zone des transistors N. Il y a de même un caisson N pour les transistors P. Cette étape utilise un premier masque. Puis on délimite à l'aide d'un deuxième masque une zone active comprenant l'ensemble des zones qui seront diffusées et les transistors. Des points de contacts entre l'alimentation, ou la masse, et le subtrat, ou le caisson, sont aussi dans les zones actives. Ce sont les points de polarisation. A l'extérieur de la zone active se trouve un oxyde épais de silicium (ces deux masques caisson et zone active sont partie A).
2. Puis on délimite la zone de silicium polycristallin, qui est au-dessus d'une couche d'oxyde mince. Cette zone est l'entrée de l'inverseur. (partie B, où le caisson n'apparaît plus).
3. Puis deux masques marquent les zones diffusées N ou P. Cette diffusion ne passe pas à travers l'oxyde. Elle ne se fait pas, donc ni sous l'oxyde mince, laissant la place aux canaux des transistors, ni sous l'oxyde épais hors de la zone active (partie C).
4. Une nouvelle couche d'oxyde épais est ajoutée, dans laquelle on délimite des trous de contacts selon un nouveau masque (partie D).
5. Des connexions d'aluminium sont gravées entre différentes parties des différents transistors. L'alimentation, à droite sur la figure, relie le point de polarisation du substrat N, et un côté des transistors P. La masse, à gauche sur la figure, relie le point de polarisation du caisson P et un côté des transistors N. Une autre connexion d'aluminium, au centre, relie le transistor N, le transistor P et la sortie de l'inverseur. Des connexions d'aluminium peuvent aussi servir à relier des sorties d'inverseurs ou de portes à des entrées d'autres portes (partie E).
6. Si nécessaire, on introduit une nouvelle couche d'oxyde épais, percée de contacts, et une nouvelle couche d'aluminium (partie F). On peut trouver ainsi 3 ou 4 niveaux d'aluminium.

   Puis l'ensemble du circuit est recouvert d'un oxyde de protection.

5.3 Procédés simplifiés

La présentation qui a été faite au paragraphe précédent montre que la fonction du circuit provient de son dessin. La forme des connexions de grille et des zones dopées donne les transistors. La forme des connexions d'aluminium et l'emplacement des contacts donnent leurs liaisons. La figure 7.18 indique ce que peut être le masque du niveau d'aluminium d'un petit morceau de circuit (une centaine de transistors). La forme des connexions *est*, en fait, la fonction.

La conception, incluant notamment le dessin, et la fabrication des circuits intégrés digitaux sont des opérations complexes et onéreuses. On essaie dans

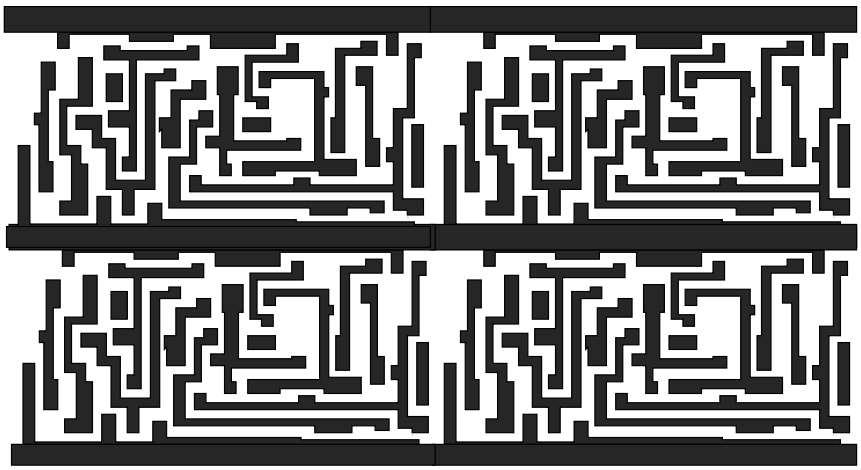


FIG. 7.18 – Masque du niveau d'aluminium d'un petit morceau de circuit (une centaine de transistors). On remarque une répétitivité du motif.

certains cas de les simplifier. Les diverses simplifications peuvent porter soit sur la fabrication proprement dite, soit sur le dessin du circuit. La simplification vise à apporter un gain soit dans la surface du circuit, soit dans sa vitesse de fonctionnement, soit dans la difficulté de conception, soit dans sa facilité à réaliser plusieurs fonctions. La surface est souvent un paramètre critique : plus le circuit est grand, plus il risque d'y avoir des défauts de fabrication, donc plus le rendement de fabrication est faible.

Des méthodes visant à simplifier la conception ou la fabrication sont exposées ci-après.

5.3.1 Simplification par la fabrication

Une des simplifications possibles est de ne pas fabriquer totalement le circuit pour une application donnée. En sortant de l'étape de fabrication, le circuit n'a pas encore de fonction. Cette fonction sera donnée par une étape de personnalisation finale du circuit. Cette personnalisation est plus ou moins définitive. Elle reçoit parfois le nom, plus ou moins impropre, de programmation.

Il y a quatre cas :

1. Aucune personnalisation, le circuit sort de la fabrication bon à l'emploi (mémoires vives, processeurs, ...).
2. Personnalisation par modification définitive : une étape de modification est appliquée au circuit. Certaines connexions sont établies, ou supprimées, pendant cette modification. Cette programmation se fait définitivement, par exemple en faisant claquer des fusibles (Programmable Logic Array, Programmable Logic Device). Cette programmation est en général faite par le client, chez lui. Une autre façon de faire assez semblable est de réaliser la personnalisation du circuit par un ensemble

de contacts et de connexions d'aluminium réalisés au moyen de masques spécifiques de l'application alors que les autres étapes ont été les mêmes pour des circuits de fonctionnalité différente (Gate arrays, sea of gates). Le fabricant réalise ainsi des ensembles de portes, non interconnectées, identiques pour différents clients. Chaque client donne au fabricant un schéma qui lui est propre pour le masque des contacts et celui de l'aluminium. Cette programmation, de mise en oeuvre lourde puisqu'il faut réaliser les dernières étapes, est faite chez le fabricant.

3. Personnalisation difficilement modifiable : certaines connexions sont établies, ou certaines informations sont stockées en mémoire, mais leur effaçage et remplacement est difficile (passage sous ultra violets, effaçage sous haute tension, etc.).
4. Personnalisation et effacement simple : une configuration est introduite dans le circuit à la façon dont un programme est introduit en mémoire. La modification est très simple (Field Programmable Gate Array). Les valeurs introduites pilotent des transistors utilisés en commutateur et établissent ou non des connexions. Ces circuits contiennent parfois aussi des petites mémoires vives de 16 mots de 1 bit permettant de tabuler certaines fonctions booléennes de 4 variables. Ces deux dernières personnalisations sont faites chez le client.

La forme de personnalisation chez le client est toujours plus ou moins la même : Le circuit est mis sur une boîte spéciale, branchée comme un simple périphérique d'un ordinateur individuel. Un logiciel, en général spécifique du type de circuit, établit à partir d'une description textuelle ou graphique de la fonction du circuit, la configuration qu'il faut introduire dans le circuit. La configuration est introduite dans le circuit sous conduite du logiciel. Dans le cas des FPGA, la configuration est stockée dans le circuit lui-même en mémoire vive.

5.3.2 Cas particulier des mémoires mortes

On trouve sous le nom de mémoire morte plusieurs cas de figures :

1. Mémoires où l'information stockée a été introduite par les masques de réalisation du circuit. La mémoire se comporte comme une tabulation matérielle de la fonction.
2. Mémoires où l'information est chargée après fabrication, de façon inaltérable (claquage de fusibles, par exemple).
3. des mémoires *flash*, vives, mais où le maintien d'information est assuré même en cas de coupure de l'alimentation électrique.
4. Mémoire morte effaçable plus ou moins souvent, plus ou moins commodément. Il existe des mémoires où l'on peut changer le contenu *seulement* un million de fois. C'est beaucoup pour une information *définitive*, mais c'est trop peu pour une mémoire vive.

5. Fausses mémoires mortes constituées en fait d'un boîtier intégrant une mémoire vive à faible consommation et une micropile.

5.3.3 Simplification par le dessin

On cherche dans de telles méthodes à avoir un dessin du circuit présentant une grande régularité.

Les simplifications du dessin des circuits consistent à reprendre des parties communes, à réutiliser certains blocs de dessins. Une façon simple est de dessiner à l'identique des tranches de circuits. Dans un microprocesseur 32 bits, il semble naturel que la partie physique qui traite le bit numéro 5 soit presque identique à celle qui traite le bit 23. On trouve régulièrement dans la presse des photos de processeurs. Sur les photos d'assez grande taille, on peut reconnaître de telles tranches. On a alors envie de dessiner très soigneusement la tranche pour qu'elle soit le plus petite possible et le plus *emboîtable* possible sur elle-même. Dessiner une tranche de microprocesseur est un problème qui ressemble un peu à dessiner un motif de papier peint avec raccord !

Une autre approche est de remarquer que les portes logiques constituant le circuit sont toutes plus ou moins identiques. On a alors comme *grain* de répétitivité non plus la *tranche* correspondant à une fonction, mais la simple porte logique. Toutes les portes de même type sont alors dessinées de façon identique. Il y a un patron pour les NAND à 2 entrées, un pour les NAND à 3 entrées, ...Toute fonction complexe utilisant une NAND2 utilise le même dessin de NAND2. Seules les connexions entre ces portes sont topologiquement différentes d'une fonction à l'autre. C'est le cas, par exemple, figure 7.18 où la même bascule est dessinée 4 fois.

6. Exercices

La "logique" voudrait qu'en permutant les positions des transistors N et P dans l'inverseur et la porte NOR, on obtienne respectivement la fonction identique (S=E) et une porte OR. Les transistors à canal N auront alors leur drain connecté à l'alimentation et leur source connectée à la sortie. En pratique le fonctionnement de ce type de porte n'est pas satisfaisant.

Pourquoi ?

Il faut raisonner de façon plus approfondie que 1 ou 0 logique et considérer les tensions $Valimentation$ - Vgs_{th}. Considérons le cas où l'entrée et la sortie de la porte sont au 1 logique. La tension de sortie devrait être égale à la tension d'alimentation. Sachant qu'une différence de potentiel minimale grille-source Vgs_{th} est indispensable à la formation du canal et la tension de grille atteint au mieux la tension d'alimentation, la tension de sortie ne pourra dépasser $Valimentation$ - Vgs_{th}. Le même raisonnement appliqué aux transistors à canal P et au 0 logique montre que la sortie ne peut descendre en dessous de Vgs_{th}. En résumé, les transistors MOS à canal N (respectivement P) ne

transmettent bien que les 0 (respectivement 1) logiques. Le cumul des pénalités Vgs_{th} rend ce montage inutilisable pour la construction de circuits à plusieurs étages de portes.

Pour réaliser une porte OR ou AND, on utilisera une porte NOR ou NAND suivie d'un inverseur.

Chapitre 8

Circuits combinatoires

Un circuit combinatoire est un dispositif matériel dont le comportement peut être décrit par une fonction booléenne générale, et toute fonction booléenne peut être réalisée par un circuit combinatoire. Un circuit combinatoire peut réaliser tout traitement si l'entrée et la sortie sont des informations codées par des vecteurs de booléens. Un circuit combinatoire convertit une information d'un code dans un autre, réalise une fonction arithmétique sur des nombres codés en binaire, etc.

Les entrées sont une nappe de fils. Une entrée, à un instant donné, est à un certain niveau logique Vrai ou Faux, c'est-à-dire à un niveau électrique. Ce niveau est susceptible de changer au cours du temps. Si une entrée est constante, la faire intervenir dans le calcul d'une fonction n'est pas une nécessité. Les sorties sont une nappe de fils. Elles sont aussi susceptibles d'évoluer au cours du temps. Il n'est pas nécessaire de réaliser un circuit combinatoire pour calculer une sortie constante.

Puisque nous nous limitons dans ce livre aux dispositifs électroniques, les circuits combinatoires sont alimentés en courant électrique mais l'alimentation et la masse ne sont pas considérées comme des entrées des circuits combinatoires.

Un circuit combinatoire est constitué d'un ensemble de portes logiques. Certaines ont été présentées au chapitre 7. Les entrées du circuit sont connectées à des entrées de portes. Les sorties du circuit combinatoire proviennent de sorties de portes. A l'intérieur du circuit il peut y avoir plusieurs circuits combinatoires ou portes successifs, les sorties des uns étant reliées aux entrées des autres.

Un circuit combinatoire est un être physique. Il occupe une certaine surface, consomme une certaine puissance électrique, puissance qui est dissipée sous forme thermique, il n'établit les valeurs correctes de ses sorties qu'un certain délai après le changement de ses entrées. Les concepteurs de circuits cherchent généralement à obtenir un circuit ayant la plus petite surface possible, donnant les délais de réponse les plus brefs possibles et consommant/dissipant le moins d'énergie possible. Ces trois critères participent au coût d'un circuit. Un autre

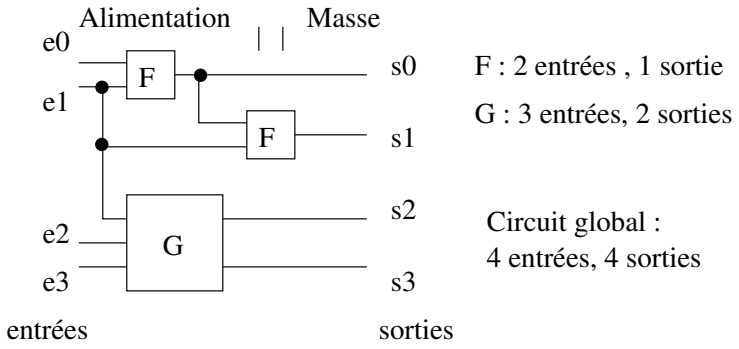


FIG. 8.1 – Un exemple de circuit combinatoire. Les carrés F et G sont des portes ou des circuits combinatoires.

critère de coût est la régularité du circuit, c'est-à-dire, indirectement, le temps nécessaire à sa conception et à son dessin. Plus un circuit comporte de fois un bloc répété, moins il est difficile de le concevoir. Voir par exemple la figure 7.18 du chapitre 7.

Dans ce chapitre nous donnons d'abord (paragraphe 1.) quelques éléments relatifs au comportement temporel des circuits combinatoires avant de définir précisément ces circuits. Cela permet de mieux situer la différence entre les circuits combinatoires et une autre classe de circuits qui fait l'objet du chapitre 10. Nous étudions ensuite (paragraphe 2.) en quoi consiste la conception de circuits combinatoires à partir de blocs physiques de base. Dans le paragraphe 3. nous insistons sur la ressemblance entre cette conception et la conception des algorithmes. Le paragraphe 4. présente une étude de cas. Certains des exemples retenus dans ce chapitre sont utiles dans plusieurs chapitres ultérieurs du livre.

1. Notion de circuit combinatoire

1.1 Comportement temporel d'un circuit combinatoire

L'origine du délai de réponse d'une porte (charge ou décharge de capacité) a été présenté au chapitre 7. Il est naturellement strictement positif. A l'heure où nous écrivons ce livre, il est couramment de l'ordre d'un dizième de nanoseconde, soit 10^{-10} s. Ce délai n'est pas constant, n'est pas une propriété de la porte elle-même. Il varie avec la valeur de la capacité à charger, la température de fonctionnement, etc. Les constructeurs donnent les valeurs maximales et minimales du délai de réponse des composants. Le délai de réponse d'un circuit combinatoire provient de l'accumulation des délais des différentes portes et interconnexions entre les entrées et les sorties. Par approximation, on considère souvent que les délais de portes cascadées s'ajoutent. Des valeurs transitoires peuvent apparaître. Mais au bout d'un certain temps les sorties sont stabi-

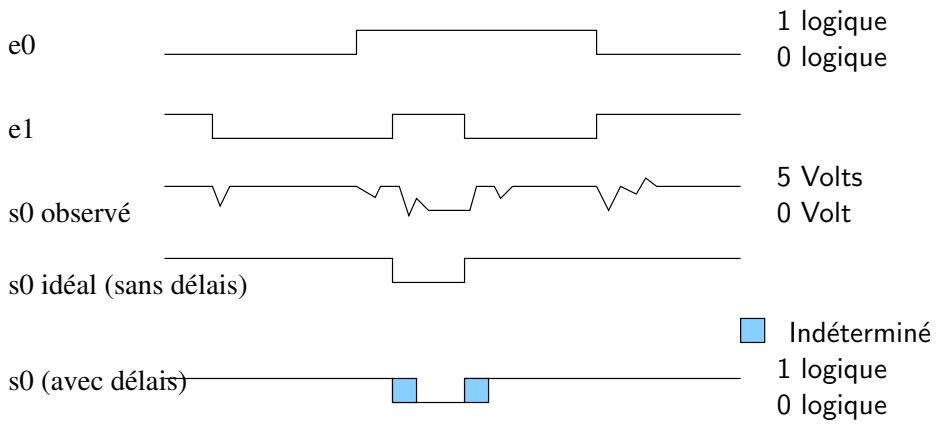


FIG. 8.2 – Comportement possible du circuit combinatoire donné en exemple

lisées. Dans un circuit combinatoire, une sortie ne peut se mettre à osciller indéfiniment. Un circuit électronique fabricant un tel signal oscillant est très utile mais n'est pas un circuit combinatoire. Par exemple le circuit présenté figure 8.1 peut donner le comportement décrit par la figure 8.2. Nous y distinguons des entrées idéales (par définition booléennes), des sorties telles qu'elles pourraient être observées (entre 0 et 5 Volts) et des sorties idéales (booléennes aussi). On trouve souvent une représentation avec des sorties indéterminées pendant le délai de réponse. Nous la faisons figurer aussi. C'est évidemment la situation la plus réaliste, mais elle n'est pas booléenne.

1.2 Caractérisation des circuits combinatoires

1.2.1 Caractérisation par le comportement

Un circuit combinatoire réalise une fonction. Cela veut dire qu'une certaine configuration des entrées donne *toujours* la même configuration des sorties. Examinons ce que signifie ce *toujours*. Si plusieurs configurations d'entrées sont appliquées successivement aux entrées du circuit combinatoire, on observe, après stabilisation, certaines configurations de sorties.

> Un circuit est *combinatoire* si :
>
> Pour tout couple (C1,C2) de configurations d'entrées, le circuit recevant la séquence temporelle C1, C2, C1 en entrée donne, après éventuelle stabilisation des valeurs, une séquence de sortie S1, S2, S1.

La configuration C1 donne *toujours* S1.

A l'inverse, si pour un circuit on peut trouver un couple de configurations d'entrées (C3, C4) tel que la séquence temporelle d'entrée C3, C4, C3 donne une séquence de sortie S3, S4, S5, avec $S3 \neq S5$, le circuit n'est pas combinatoire.

Intuitivement le circuit non combinatoire se *souvient* qu'il est passé par la configuration S4, cela change ses résultats ultérieurs. Le circuit combinatoire ne se souvient de rien. Les circuits combinatoires n'ont aucune fonction de mémorisation. On verra des circuits ayant une mémorisation dans des chapitres ultérieurs.

Il convient de préciser que si les configurations d'entrée C1 et C2 diffèrent de plusieurs bits, on suppose les changements de valeurs simultanés (comme les entrées e1 e0 de la figure 8.2).

Le circuit combinatoire réalise une fonction au sens mathématique du terme : chaque appel avec des valeurs identiques des paramètres délivre la même valeur. A l'inverse, la *fonction* `random` des calculettes n'est pas une fonction puisque différents appels ne donnent pas le même résultat.

1.2.2 Caractérisation par la structure

Considérons un assemblage de portes interconnectées comme un graphe. Les portes sont les noeuds, les connexions les arcs. Une orientation évidente des arcs est fournie par le sens sortie d'une porte vers l'entrée d'une autre. Si le graphe ainsi obtenu est sans cycle, le circuit est combinatoire.

> **Remarque :** Attention, la réciproque est fausse ! L'exercice E8.17 donne un circuit combinatoire comportant un cycle. Il est hors du sujet de ce livre de caractériser l'ensemble des circuits combinatoires avec un cycle.

1.3 Le principe de réalisation

Les objets de base utilisés dans les circuits combinatoires sont les portes logiques. Une technologie de réalisation étant choisie, il n'est pas nécessaire de garder la représentation en transistors des portes NOR ou NAND comme dans le chapitre 7. L'usage a consacré des symboles pour les portes. Les connexions entre ces portes sont représentées par des traits. Ces symboles sont représentés figure 8.3.

On trouve souvent des portes NAND et NOR sans le petit rond utilisées pour représenter des fonctions ET et OU.

1.3.1 Les circuits existants : inverseurs, portes NAND et NOR

Les inverseurs, les NAND et les NOR sont les portes élémentaires. L'inverseur est un NAND (ou un NOR) à une seule entrée. Pour des raisons électriques (trop grande résistance obtenue en mettant beaucoup de transistors en série, par exemple) le nombre d'entrées des portes est parfois limité. Ainsi pour la réalisation de carte imprimée à base de boîtiers de la famille technologique TTL (Transistor Transistor Logic) on peut disposer de portes NAND à 2, 3, 4, 8 ou 13 entrées. Pour réaliser une fonction NAND portant sur 6 variables, tout va bien puisque NAND(a, b, c, d, e, f) = NAND(a, b, c, d, e, f, f, f) et la porte

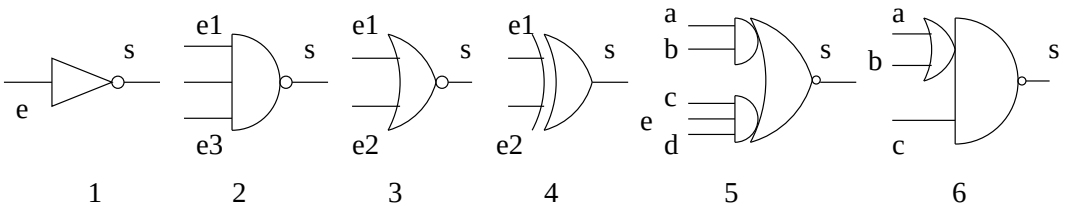


FIG. 8.3 – Représentation conventionnelle des portes logiques :
porte 1 : l'inverseur : s = not(e) ;
porte 2 : le NAND : s = nand(e1,e2,e3) ;
porte 3 : le NOR : s = nor(e1,e2) ;
porte 4 : le XOR : s = xor(e1,e2) ;
porte 5 : le ANDNOR : s = nor(and(a,b), and(c,d,e)) ;
porte 6 : le ORNAND : s = nand (or(a,b), c).

NAND à 8 entrées fait l'affaire. Mais pour une fonction à plus de 13 entrées c'est moins simple.

L'exercice E8.15 donne une idée de solution à ce problème. Dans d'autres technologies les portes NOR ne peuvent avoir que 2 ou 3 entrées et les portes NAND que 2, 3 ou 4. Dans certains cas, plusieurs technologies peuvent intervenir dans un même équipement matériel comportant plusieurs puces. Une puce peut ne contenir que des portes à au plus 4 entrées alors que la puce voisine a des portes à 20 ou 2000 entrées.

1.3.2 Assemblage systématique

Toute fonction booléenne peut s'exprimer sous forme de somme de produits de variables normales ou complémentées. Cette expression en somme de produits peut se traduire de façon systématique sous forme de combinaison de deux étages de NAND de variables normales ou complémentées. On rappelle que, par application directe des règles de De Morgan, si a, b, c, d, e, f sont des variables booléennes :

$a + b.c + d.e.f = \overline{\overline{a}.\overline{(b.c)}.\overline{(d.e.f)}}$

ou, en utilisant une notation préfixée pour le NAND,

$a + b.c + d.e.f = nand(not(a), nand(b, c), nand(e, f, g))$

De même, pour une expression en produit de sommes,

$a.(b + c).(d + e + f) = \overline{\overline{a} + \overline{(b + c)} + \overline{(d + e + f)}}$

ou, en utilisant une notation préfixée pour le NOR,

$a.(b + c).(d + e + f) = nor(not(a), nor(b, c), nor(e, f, g))$

On réalise un circuit dont le comportement est décrit par une fonction booléenne par un tel assemblage de portes NAND ou NOR et d'inverseurs (Cf. Figure 8.4). Cela donne des schémas logiques de circuits combinatoires dans lesquels il n'y a que deux ou trois étages de portes entre l'entrée et la sortie : un étage d'inverseurs pour certaines entrées, puis deux étages, soit de

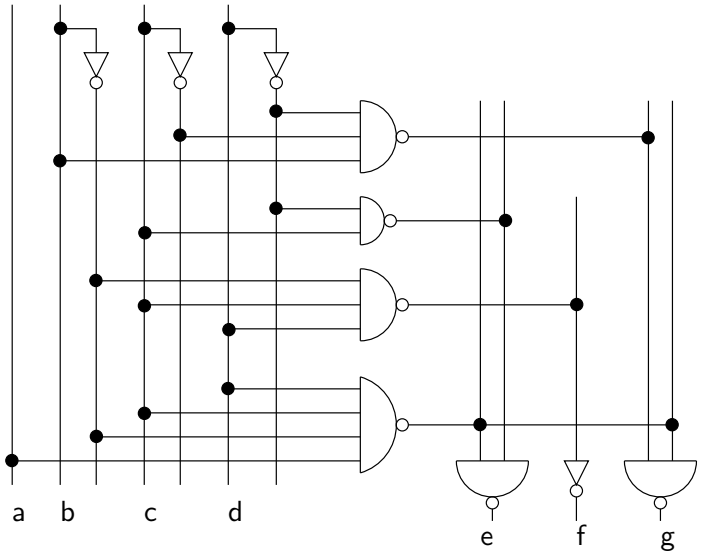


FIG. 8.4 – Exemple de réalisation de fonctions à base de portes NAND.
$g = a.\overline{b}.c.d + b.\overline{c}.\overline{d}$
$f = \overline{b}.c.d$
$e = a.\overline{b}.c.d + c.\overline{d}$

NAND soit de NOR. Ces circuits sont optimaux en terme de nombre d'étages.

Exemple E8.1 : Réalisation en NAND de la fonction majorité
Reprenons la table de vérité de l'addition de deux naturels (Cf. Paragraphe 2.2.2, chapitre 3) en nous limitant à la fonction majorité :

a	b	r_e	r_s $maj(a, b, r_e)$	a	b	r_e	r_s $maj(a, b, r_e)$
0	*0*	*0*	*0*	*1*	*0*	*0*	*0*
0	*0*	*1*	*0*	*1*	*0*	*1*	*1*
0	*1*	*0*	*0*	*1*	*1*	*0*	*1*
0	*1*	*1*	*1*	*1*	*1*	*1*	*1*

On obtient l'expression optimisée :

$$r_s = a.b + a.r_e + b.r_e$$
$$ou \quad r_s = nand(nand(a, b), (nand(a, r_e), nand(b, r_e))$$

Le schéma en portes NAND *du circuit combinatoire réalisant la fonction majorité est donné figure 8.5.*

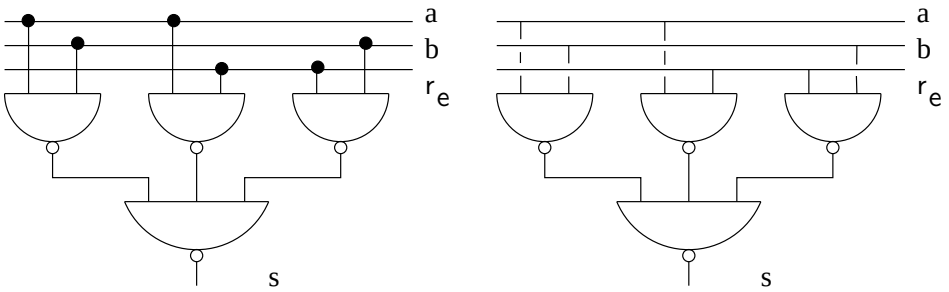


FIG. 8.5 – Réalisation de la fonction majorité en portes NAND. Sur le schéma, deux conventions usuelles sont présentées : avec l'une, les points noirs représentent des connexions entre un fil horizontal et un fil vertical ; avec l'autre, un trou dans le fil vertical permet de mettre en évidence la non-connexion.

1.4 Conception de circuits combinatoires

Etant donnée une fonction booléenne générale, concevoir un circuit combinatoire consiste à assembler des éléments logiques de base, choisis parmi une liste donnée, comme les portes par exemple, pour que le comportement global de l'ensemble soit décrit par la fonction booléenne voulue. De plus cet assemblage doit être d'un coût acceptable, voire minimal. Cette conception de circuits suppose deux niveaux de travail. Les noms de ces deux niveaux sont fluctuants mais nous choisissons ceux d'*algorithmique câblée* et de *synthèse logique*. On retrouvera cette distinction dans le chapitre 10. La conception peut se faire soit manuellement (rarement), soit, principalement, en utilisant des outils de Conception Assistée par Ordinateur. Il existe donc des outils de CAO de synthèse logique [1] ou d'algorithmique câblée. Il arrive aussi que les deux outils soient fusionnés en un seul.

1.4.1 Algorithmique câblée

Il arrive que l'expression de la fonction booléenne fasse appel à des objets extra-booléens. En particulier la donnée d'une fonction arithmétique et du codage binaire des nombres manipulés constituent une description d'une fonction booléenne. Par exemple *un multiplieur combinatoire de 2 nombres entiers codés sur 64 bits en complément à 2* est une description d'une fonction booléenne.

La conception conduit à une expression de la fonction globale comme une composition de sous-fonctions décrites en termes booléens et/ou arithmétiques. Cette décomposition n'est jamais unique et les aspects de coût doivent être pris en considération pour choisir la *bonne* solution. La possibilité de ré-utiliser des résultats intermédiaires doit être retenue pour économiser des éléments. Le

[1] Pour la beauté de son nom signalons un outil qui *construit les portes* à partir de la description de la fonction booléenne : BuildGates !

critère de régularité peut intervenir aussi comme critère de comparaison entre des solutions.

Cette recherche d'un assemblage d'éléments donnant un comportement attendu est très proche de l'algorithmique où l'on cherche à assembler les instructions pour obtenir un certain comportement.

Cet ensemble de techniques, que nous détaillons dans le paragraphe 3. sous le nom d'*algorithmique câblée*, est parfois nommée *conception logique*. Elle conduit à une description en terme de composition de fonctions booléennes de la fonction globale. Le terme *algorithmique câblée* peut sembler étrange. Il indique simplement que la composition de fonctions exprimées dans les algorithmes de ces traitements est un banal câblage. Utiliser les résultats d'une fonction comme entrées d'une autre fonction, c'est connecter les sorties du sous-circuit réalisant l'une aux entrées du sous-circuit réalisant l'autre.

1.4.2 Synthèse logique

On parle de *synthèse logique* pour décrire l'assemblage de portes physiques choisies parmi une liste donnée, à partir de la description de la fonction booléenne.

Il ne faut pas perdre de vue que la notion d'éléments logiques de base n'est pas absolue. Elle est relative à une liste donnée, une *bibliothèque* de circuits. De la même façon qu'en programmation, différents langages ou systèmes peuvent offrir des primitives plus ou loins avancées.

Un circuit combinatoire réalisant les fonctions majorité et $\oplus$ du chapitre 3 est un additionneur 1 bit. Il comporte en général deux portes XOR comme celles de l'exercice E8.14. La fonction majorité est connue aussi (exemple E8.1), mais rien n'interdit de considérer un additionneur 1 bit comme bloc de base. Il est d'ailleurs inclus dans beaucoup de bibliothèques. D'autres bibliothèques proposent un circuit de calcul rapide des retenues.

L'utilisation des outils de CAO de synthèse logique suppose la description de la fonction booléenne et celle de l'ensemble d'éléments de base selon un langage formel traitable par un programme. La conception manuelle repose plutôt sur l'utilisation d'un schéma et d'un catalogue des composants disponibles.

Entre la représentation de la fonction booléenne et celle de la structure de connexions entre éléments qui la réalise, il peut exister plusieurs représentations intermédiaires. Il faut que l'équivalence soit préservée dans ces différentes formes. Des techniques de synthèse logique peuvent s'appliquer plutôt sur la forme algébrique (remplacement d'une formule booléenne par une autre formule booléenne) ou plutôt au résultat physique (remplacement d'un élément de circuit par un autre, plus petit ou moins consommateur, par exemple).

Dans la suite nous présentons d'abord les cas simples, où la distance est faible entre l'expression algébrique et la réalisation. Pour cela nous présentons des exemples d'éléments logiques qui peuvent être considérés comme *de base*

et nous montrons les expressions algébriques qui y *collent* le mieux. Il s'agit bien de *synthèse logique*.

Nous étudions ensuite des circuits moins simples où la distance peut être grande entre l'expression de la fonction et la structure de la réalisation. Il s'agit bien alors d'une véritable *algorithmique câblée*.

Nous privilégions les solutions systématiques mais nous montrerons parfois quelques *astuces*.

2. Assemblage de blocs de base : synthèse logique

2.1 Décodeurs, encodeurs

2.1.1 Les circuits existants

Un circuit fabriquant en sortie les 2^N monômes canoniques correspondant à ses N entrées est appelé un *décodeur*. On en rencontre en particulier dans les mémoires où, à partir des N bits d'adresse, il faut émettre un des 2^N signaux de sélection d'un mot. Ainsi pour le décodeur, à partir d'un nombre codé en binaire, on obtient un seul 1 parmi une nappe de fils. C'est le fil dont le numéro est celui donné en entrée. Généralement il y a un AND entre une entrée supplémentaire de validation *val* et ce bit de sortie.

La fonction d'un *encodeur* est exactement symétrique. Si parmi une nappe de fils on est certain qu'un seul est à 1 à un instant donné, l'encodeur donne le numéro de ce fil. Si la garantie qu'un seul fil d'entrée est à 1 ne peut être établie, le circuit est différent. Il délivre alors le numéro du premier fil à 1. La notion de *premier* suppose un ordre sur les fils ; c'est soit l'ordre des numéros croissants, comme dans notre exemple, soit décroissants.

Les tables de vérité de la figure 8.6 caractérisent le décodeur, l'encodeur avec garantie qu'une seule entrée est à 1 (encodeur1), l'encodeur en général, sans cette garantie (encodeur2). Dans ce dernier on introduit une sortie a, vraie si aucune entrée n'est à 1.

On en déduit aisément les expressions logiques et les schémas correspondants. Par exemple, dans le décodeur : $s0 = \overline{e1}.\overline{e0}.$ val, dans l'encodeur1 : $s1 = e3 + e2$.

2.1.2 Synthèse systématique

L'exemple E8.8, paragraphe 3.3, montre l'utilisation de décodeurs. On en retrouvera dans le livre comme décodeurs d'adresse (Cf. Chapitres 9 et 15).

Décodeur		Encodeur1		Encodeur2	
Entrées val e1 e0	Sorties s3 s2 s1 s0	Entrées e3 e2 e1 e0	Sorties s1 s0	Entrées e3 e2 e1 e0	Sorties s1 s0 a
1 0 0	0 0 0 1	0 0 0 1	0 0	0 0 0 0	Φ Φ 1
1 0 1	0 0 1 0	0 0 1 0	0 1	0 0 0 1	0 0 0
1 1 0	0 1 0 0	0 1 0 0	1 0	0 0 1 x	0 1 0
1 1 1	1 0 0 0	1 0 0 0	1 1	0 1 xx	1 0 0
0 x x	0 0 0 0			1 xxx	1 1 0

FIG. 8.6 – Tables de vérité des encodeurs et décodeurs. Les x et les Φ indiquent une valeur non pertinente respectivement en entrée ou en sortie.

2.2 Programmable Logic Array

2.2.1 Les circuits existants

On trouve dans le commerce des circuits nommés PLA ou PLD (*Programmable Logic Arrays* ou *Programmable Logic Devices*). L'utilisateur peut facilement personnaliser ces circuits pour y réaliser des produits ou des sommes de produits. Dans ces circuits les nombres d'entrées, sorties et, s'il y a lieu, monômes, sont fixés. Par exemple un PLA donné a 12 entrées, 20 monômes et 8 sorties.

L'utilisateur claque des fusibles pour fixer : 1) quelle entrée (normale ou complémentée) fait partie de quel monôme ; c'est la partie AND du PLA. 2) quel monôme fait partie de quelle sortie ; c'est la partie OR du PLA.

Le claquage (la "*programmation*") se fait dans un petit dispositif électronique, connectable à un ordinateur personnel, et facilement commandable par un logiciel qui a reçu les équations logiques en entrée.

Des organisations proches des PLA peuvent être réalisées à base de portes dans les circuits non pré-existants.

2.2.2 Synthèse systématique

L'utilisateur cherche souvent à minimiser le nombre de monômes de la fonction ou des fonctions à réaliser. Soit parce que le nombre total de monômes est contraint par la technologie, soit pour économiser de la surface dans le circuit. Nous allons étudier un exemple de fonction réalisée sur un tel réseau programmable PLA permettant de faire des sommes de produits.

Dans un cas on a procédé à une minimisation de chacune des fonctions, indépendamment des autres, par des tableaux de Karnaugh. Dans l'autre cas, on a cherché une minimisation globale grâce à un outil de CAO.

Exemple E8.2 : Contrôle d'un afficheur 7 segments
L'exemple retenu est très classique. Nous l'avons déjà rencontré dans le chapitre 2. Un circuit combinatoire reçoit 4 entrées x_3, x_2, x_1, x_0 codant un naturel

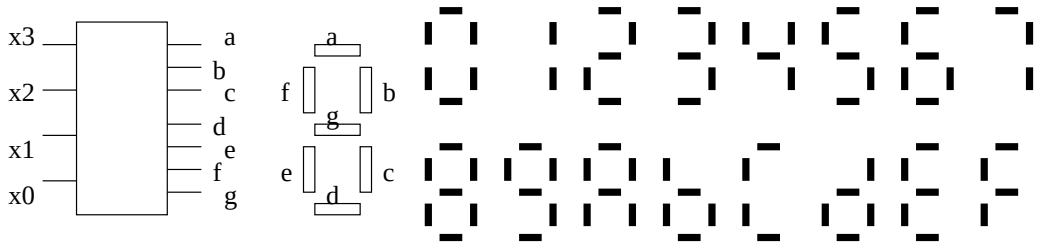


FIG. 8.7 – Affichage des nombres de 0 à 15 sur 7 segments

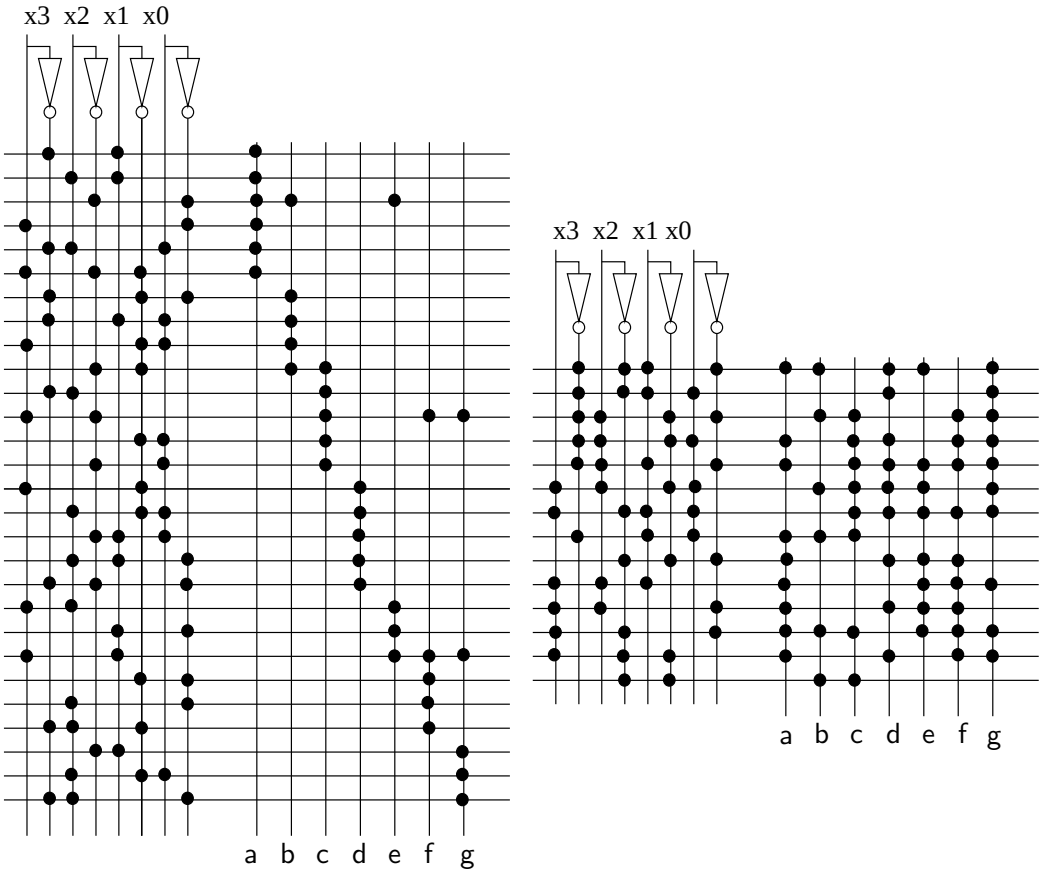


FIG. 8.8 – Description symbolique des PLA réalisant le codage pour un afficheur 7 segments. A gauche minimisation de chacune des fonctions, indépendamment les unes des autres, à droite, minimisation globale.

entre 0 et 15. Il délivre 7 sorties activant 7 segments d'un afficheur. Les 7 segments se nomment a, b, c, d, e, f et g. Ils sont disposés comme sur la figure 8.7. Les chiffres hexadécimaux sont affichés comme indiqué. La fonction du circuit est de transcoder entre le code binaire des nombres et le code en segments allumés et segments éteints.

On cherche à exprimer chacune des 7 fonctions booléennes a, ..., g en fonction de x_3, x_2, x_1, x_0. *Par exemple,*

$$a = \overline{x_3}.x_1 + x_2.x_1 + \overline{x_2}.\overline{x_0} + x_3.\overline{x_0} + \overline{x_3}.x_2.x_0 + x_3.\overline{x_2}.\overline{x_1}$$

Pour la solution globale on obtient 28 monômes différents. La partie gauche de la figure 8.8 donne les différentes fonctions. Chaque ligne figure un monôme. Pour chaque monôme, on représente par un point noir : quelles entrées il prend en compte (partie AND *du* PLA*) ; dans quelles sorties il figure (partie* OR *du* PLA*). Ainsi la troisième ligne représente le monôme* $\overline{x_2}.\overline{x_0}$. *Il est utilisé par les fonctions* a, b *et* e.

2.2.3 Minimisation locale, minimisation globale

Dans les réalisations à base de monômes, en portes ou en PLA, le concepteur cherche à minimiser le nombre total de monômes. La surface du PLA est en effet proportionnelle à ce nombre. Si cette recherche est faite indépendamment pour chacune des fonctions individuelles, il se peut que le résultat global soit moins bon que pour une minimisation globale. Les outils modernes de Conception Assistée par Ordinateur comportent de tels programmes de minimisation globale. Pour la solution du transcodeur de 7 segments, avec une telle minimisation globale, un outil développé par l'un des auteurs obtient 14 monômes différents. La partie droite de la figure 8.8 donne les différentes fonctions.

2.3 Mémoires mortes : une table de vérité câblée

2.3.1 Les circuits existants

Une *mémoire morte* de 256 mots de 16 bits réalise 16 fonctions combinatoires de 8 variables. En effet à partir d'une configuration des 8 entrées, habituellement interprétée comme une adresse, la mémoire morte délivre 16 bits. Une telle réalisation de fonction à base de ROM est parfois utilisée. Elle est optimale du point de vue du temps de conception. C'est une très bonne solution de paresseux ! La table de vérité suffit à décrire la réalisation. On peut aussi considérer cette solution comme un PLA avec *tous* les monômes canoniques. On trouve souvent dans les circuits programmables (FPGA) des petites ROM, inscriptibles par l'utilisateur, par programmation. Ces mémoires, qu'on ne peut plus appeler mortes, reçoivent le nom de Look-Up Tables (LUT).

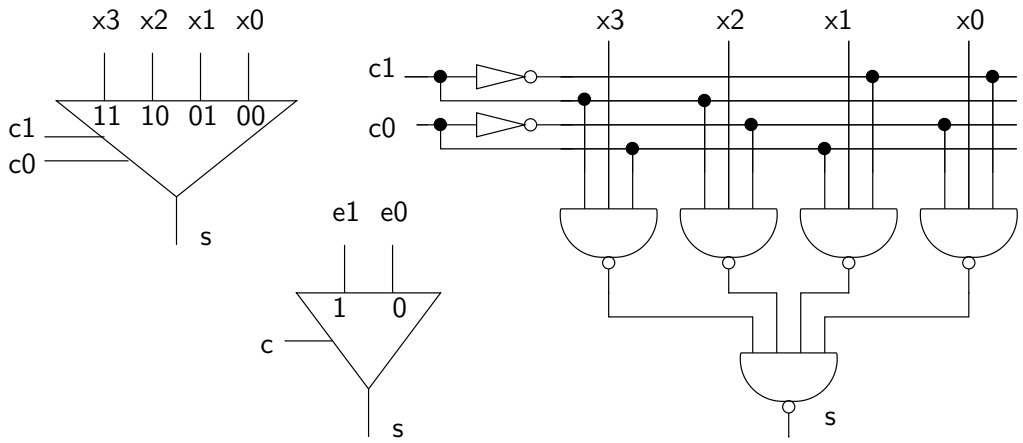


FIG. 8.9 – Représentation symbolique des multiplexeurs 4 voies vers 1 et 2 voies vers 1. Structure interne du multiplexeur 4 voies vers 1.

2.3.2 Synthèse systématique

Pour une fonction de 8 variables on forme les 256 monômes canoniques pour exprimer la fonction sous forme de somme de produits à partir de la table de vérité, et on réalise un circuit combinatoire en *collant* à cette expression.

2.4 Multiplexeurs

2.4.1 Les circuits existants

Un circuit combinatoire est d'usage fréquent : le *multiplexeur*. Il réalise la sélection parmi 2^N entrées de données. Celle des entrées sélectionnée est celle dont le numéro est donné sur les N bits de commande. Le nombre d'entrées de commande du multiplexeur est le logarithme à base 2 du nombre de bits de données.

Ainsi pour 2 bits de commande c1 et c0 et 4 bits de donnée x3, x2, x1, x0 la sortie s est décrite par l'équation logique :

$$s = c1.c0.x3 + c1.\overline{c0}.x2 + \overline{c1}.c0.x1 + \overline{c1}.\overline{c0}.x0.$$

Si les bits c1 c0 codent l'entier k, la sortie est égale à l'entrée x d'indice k. La réalisation interne s'en déduit aisément. Elle constitue dans ce cas un multiplexeur 4 voies vers 1. Le schéma conventionnel des multiplexeurs 2 voies vers 1 et 4 voies vers 1 sont donnés figure 8.9.

2.4.2 Synthèse systématique

Les multiplexeurs sont très pratiques pour synthétiser une fonction décrite par un graphe de décision binaire (BDD). Ils sont surtout une brique de base de l'algorithmique câblée où ils réalisent la primitive de choix. Ainsi de nombreux

outils de CAO partent d'une représentation des fonctions booléennes en BDD, notamment si la brique de base des circuits combinatoires est le multiplexeur.

> **Remarque :** Attention le multiplexeur est orienté. Ce n'est pas un aiguillage 4 voies vers 1 tel qu'il pourrait être réalisé avec 4 commutateurs (voir la figure 7.10 dans le chapitre 7).

Exemple E8.3 : Cascades de multiplexeurs
En utilisant 5 multiplexeurs 4 voies vers 1, on peut réaliser un multiplexeur 16 voies vers 1. Pour cela on utilise un premier étage de 4 multiplexeurs en parallèle recevant les mêmes 2 bits de commande, puis l'autre multiplexeur recevant les deux derniers bits de commande. Le lecteur est convié à examiner différentes possibilités de choix d'affectation des bits de commandes soit au premier étage de 4 multiplexeurs en parallèle soit au deuxième.

2.5 Portes complexes

2.5.1 Les circuits existants

Dans certains types de réalisations des portes un peu complexes sont utilisées ; par exemple les portes ORNAND et ANDNOR présentées figure 8.3.

2.5.2 Synthèse systématique

Le principal problème lié à l'utilisation de telles portes est que des procédés systématiques de synthèse ne sont pas toujours disponibles. On obtient facilement une expression en somme de produits, donc en NAND de NAND. Les portes plus complexes correspondant par exemple à $\overline{(abc + de + fg)}$ ne peuvent provenir que d'outils de Conception Assistée par Ordinateur. Les méthodes utilisées alors consistent à modifier les formes algébriques (ou autres comme les BDDs) des fonctions booléennes pour retrouver ou calquer (on parle de *mapping* en anglais) les motifs correspondant aux éléments de base disponibles.

3. Algorithmique câblée : conception logique

La conception logique a pour but de composer des fonctions booléennes, éventuellement assez complexes, pour réaliser une fonction booléenne plus générale. L'essentiel de la difficulté est la recherche de régularité.

3.1 La question de la régularité

Pour introduire la notion de régularité dans la conception, nous montrons ici les résultats de deux méthodes de conception sur un même exemple.

Exemple E8.4 : Conversion binaire vers DCB
Il s'agit d'une fonction booléenne à 9 entrées et 10 sorties. Sur la figure 8.10 les bits d'entrée et de sorties ne figurent que par leur numéro. Le circuit convertit l'écriture binaire d'un naturel de l'intervalle [1, 366] (donc sur 9 bits) vers son écriture en Décimal Codé en Binaire (donc sur 10 bits). Chaque chiffre de l'écriture décimale est codé en binaire, par exemple $285_{10} = 1\,0001\,1101_2 = 10\,1000\,0101_{DCB}$.

1) *Par l'étude de l'algorithme de conversion, nous connaissons une décomposition de la fonction du circuit en fonctions élémentaires. Il y a une forte* régularité *dans la façon dont se combinent ces fonctions élémentaires. Elle est liée à la régularité induite par l'algorithme de conversion. Cette décomposition donne la structure de la solution 2 de la figure 8.10. Tous les rectangles représentent la même fonction à 4 entrées et 4 sorties. Les deux rectangles marqués d'un point, blanc ou noir, ont une entrée de moins ou une entrée et une sortie de moins. Tous les blocs étant identiques et leur disposition étant régulière, le schéma topologique du circuit serait simple. La fonction étant une fonction arithmétique, pour obtenir le même circuit pour plus d'entrées et de sorties, il suffit d'étendre le schéma. On dénombre 6 niveaux de blocs entre les entrées et les sorties.*

2) *Nous avons, avec l'aide des auteurs d'un logiciel de synthèse logique, donné la table de vérité complète de ce circuit à l'outil. Cela représente un peu moins de 400 lignes de 10 bits. Elles peuvent être obtenues par un programme. Ce logiciel a travaillé* en aveugle *uniquement à partir de ces tables. Le logiciel cherchait à synthétiser à partir de fonctions à 3 ou 4 entrées. Il a essayé de minimiser le nombre total de blocs. Il a par ailleurs essayé de regrouper des fonctions qui utilisaient les mêmes variables ou les mêmes résultats intermédiaires. Il a de plus cherché à minimiser le nombre de niveaux logiques total entre les entrées et les sorties. Le résultat est celui de la partie 1 de la figure 8.10. Par exemple le bloc représenté en grisé reçoit les 4 entrées de numéro 7, 5, 3 et 2 et délivre 3 sorties, chacune étant utilisée dans deux blocs. On dénombre 4 niveaux de blocs entre les entrées et les sorties.*

Dans tous les circuits ayant un grand nombre d'entrées, le concepteur cherche une régularité permettant de simplifier le travail. Il est à noter que cette régularité se retrouve souvent dans la topologie effective de la réalisation du circuit. La plupart des circuits réalisant des fonctions arithmétiques, et d'autres, présentent de telles régularités. Nous allons les étudier. Cette partie suppose connus les éléments du chapitre 3 sur les représentations des grandeurs.
Dans la suite nous montrons 3 méthodes d'assemblage de sous-circuits.

– Dans la première, l'assemblage *itératif*, ou *linéaire*, la connaissance de la so-

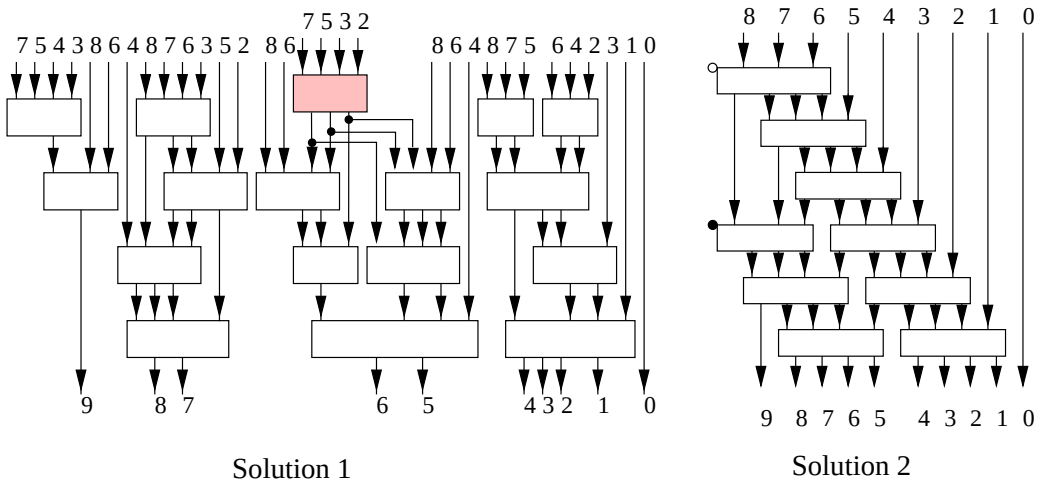


FIG. 8.10 – Deux solutions pour la réalisation d'une même fonction

lution pour le circuit travaillant sur $N-1$ bits permet de concevoir le circuit travaillant sur N bits. Cette structure est proche de la boucle d'itération. Le circuit global est obtenu en répétant N fois un circuit de base.

– Dans la deuxième, l'assemblage *récursif*, ou *arborescent*, la connaissance de la solution pour le circuit travaillant sur $N/2$ bits permet de concevoir le circuit travaillant sur N bits. Cette structure est proche de la structure d'arbre binaire. Parfois les deux réalisations de la fonction sur $N/2$ bits doivent être complétées pour permettre de réaliser la fonction sur N bits.
– La troisième méthode, *générale*, regroupe des blocs selon des règles de composition de fonctions quelconques.

Pour chacune de ces méthodes d'assemblage nous donnons un ou quelques exemples typiques. Un exemple simple, la fonction incrémentation, permet ensuite de comparer différentes méthodes dans une étude de cas.

3.2 Assemblages linéaires

3.2.1 Schéma en tranches, schéma en blocs

Représenter un circuit résultant d'un assemblage itératif peut se faire de deux façons représentées figure 8.11. C'est un exemple sans signification. Un circuit traite deux vecteurs de n bits a et b. Le traitement se compose de la mise en série de 3 fonctions. Dans la première fonction, symbolisée par un carré, une entrée externe X est prise en compte pour chaque bit. Dans la troisième fonction, symbolisée par un ovale, une information passe de bit en bit, à la façon d'une retenue.

On parle de *représentation en tranches* quand on fait apparaître toutes les cellules qui participent à la fonction globale. L'exemple réel de l'UAL (Cf. Exemple E8.10) utilise cette technique.

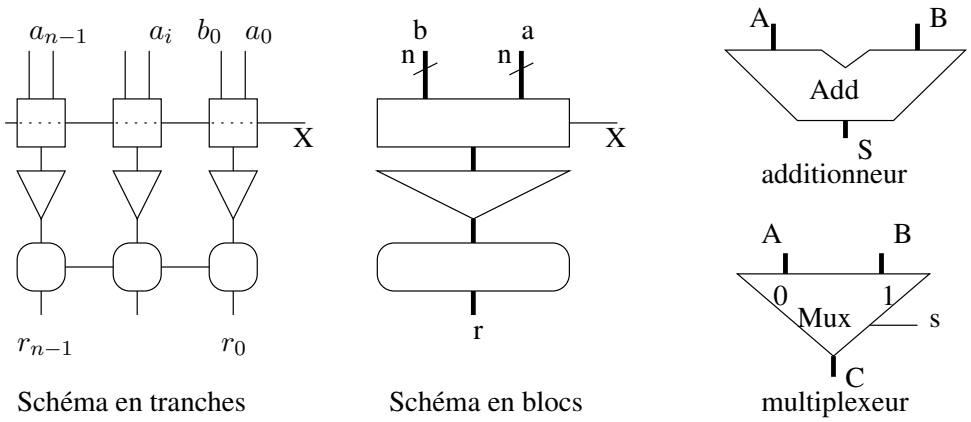


FIG. 8.11 – Représentation d'un circuit en tranches ou en blocs ; représentation conventionnelle de l'additionneur et du multiplexeur N bits

On parle de *représentation en blocs* quand on ne dessine que les fonctions sur N bits. On représente alors par un trait gras les bus, ou nappes de fils. L'exemple du circuit de calcul du quantième (Cf. Exemple E8.9) utilise cette représentation.

3.2.2 Exemples : addition et soustraction de naturels

Exemple E8.5 : L'additionneur N bits
La mise en cascade de N additionneurs 1 bit constitue un additionneur N bits. Il peut effectuer l'addition de deux naturels ou de deux relatifs codés sur N bits. La somme de deux naturels codés en binaire pur sur N bits est sur $N+1$ bits. Le schéma du circuit est donné figure 3.3 (chapitre 3).

Exemple E8.6 : L'additionneur/soustracteur N bits
On a vu au chapitre 3 que le calcul sur les vecteurs booléens donnant la représentation de la somme ou de la différence, codée en binaire pur ou en complément à 2, est le même. Puisque le circuit peut effectuer l'addition ou la soustraction, il dispose d'un bit de commande $Add/\overline{Sub}$. Ce fil vaut 0 si l'opération voulue est une soustraction, 1 si c'est une addition.

Le calcul de la somme A+B se fait en ajoutant A, B et 0. Le calcul de la différence A-B se fait en ajoutant A, le complémentaire booléen de B et 1.

On se sert du fil $Add/\overline{Sub}$ pour sélectionner l'opérande Q à ajouter à A. Pour chaque bit, on a $Q_i = Add.B_i + Sub.\overline{B_i}$.

De même, on fabrique le report entrant r_0, pour ajouter 0 ou 1, selon l'équation : r_0 = (si $Add/\overline{Sub}$ alors 0 sinon 1) = $\overline{Add/\overline{Sub}}$

Si l'opération est une addition, la retenue sortante C est le report sortant. Si l'opération est une soustraction, la retenue sortante C est le complémentaire

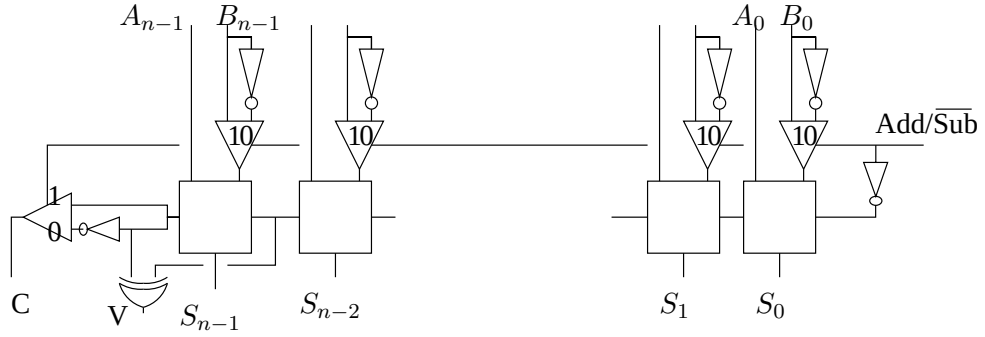


FIG. 8.12 – Additionneur-soustracteur N bits. Chaque carré est un additionneur 1 bit. Tous les multiplexeurs sont commandés par le même signal.

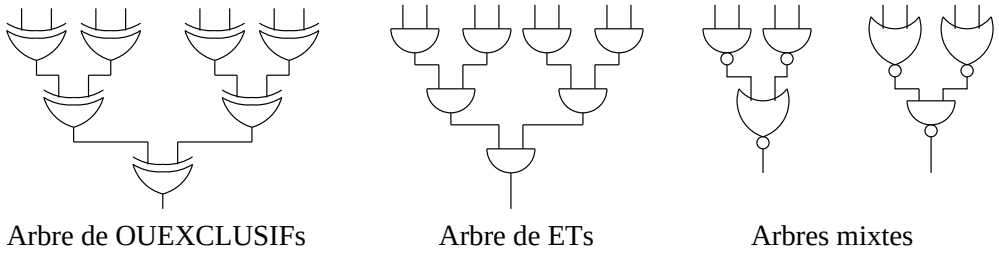


FIG. 8.13 – Arbres binaires de portes

de ce report sortant. Le bit d'oVerflow V est le XOR *des deux derniers reports. Le schéma, en tranches, du circuit est donné figure 8.12.*

3.3 Assemblages arborescents

Exemple E8.7 : Le XOR **généralisé**
On connaît la porte XOR *à 2 entrées. Cette fonction est la somme modulo 2 si l'on interprète les deux entrées comme entiers plutôt que comme booléens. Il est possible d'obtenir une somme modulo 2 de N entiers sur un bit (ou le* XOR *généralisé de N booléens) en utilisant l'associativité de cette fonction. Ce calcul est utilisé pour obtenir le bit de parité d'un mot qui vaut 1 si le mot a un nombre impair de 1. Ce genre de technique peut s'appliquer pour toute opération associative, par exemple le* AND *ou le* OR*. La figure 8.13 rappelle que des arbres de* NAND *et de* NOR *peuvent remplacer les* AND *ou les* OR*. Voir aussi l'exercice E8.15.*

Exemple E8.8 : Le décodeur N bits
Le décodeur est présenté paragraphe 2.1. Nous nous intéressons ici à sa réalisation interne. Nous supposerons que son nombre d'entrées N est une puissance de 2. Il fabrique 2^N sorties booléennes sur 2^N fils à partir de N

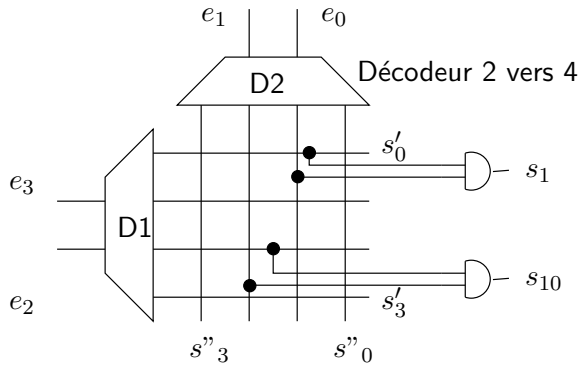


FIG. 8.14 – Décodeur à 4 entrées et 16 sorties, à partir de 2 décodeurs à 2 entrées et 4 sorties et de 16 portes AND.

entrées. Une seule des sorties est à 1. Il est très facile de décrire un tel circuit récursivement :

- *si N vaut 1, le circuit consiste en 1 seul inverseur. Les deux sorties sont l'entrée et l'entrée complémentée.*
- *si N est supérieur à 1, on dispose de deux décodeurs à $N/2$ entrées. Ils ont chacun $2^{N/2}$ sorties. En combinant 2 à 2 dans des portes* AND *à deux entrées les sorties des 2 décodeurs, on obtient le décodeur souhaité.*

Montrons le passage de 2 entrées à 4 par un exemple (Cf. Figure 8.14).
Un décodeur D1 à 2 entrées e_3 e_2 délivre les 4 sorties s'_3 s'_2 s'_1 s'_0.
Un décodeur D2 à 2 entrées e_1 e_0 délivre les 4 sorties s''_3 s''_2 s''_1 s''_0.
Les équations des sorties du décodeur à 4 entrées sont, pour p compris entre 0 et 15 :

$$s_p = s'_{p\ div\ 4}\ \text{AND}\ s''_{p\ modulo\ 4}$$

c'est-à-dire :

$$\begin{array}{rll} & s_{15} = s'_3\ \text{AND}\ s''_3 & s_{14} = s'_3\ \text{AND}\ s''_2 \\ \textit{jusqu'à} & s_1 = s'_0\ \text{AND}\ s''_1 & s_0 = s'_0\ \text{AND}\ s''_0 \end{array}$$

3.4 Assemblages généraux

L'expression d'une fonction booléenne très complexe comme composition de fonctions booléennes plus simples donne une organisation de circuits combinatoires. Il suffit de coller la structure du circuit sur la structure de la combinaison de fonctions. C'est le cas pour le circuit de calcul du quantième dans l'année présenté en exemple.

Dans les cas où une composition est connue, tout va bien. Si on ne sait pas exprimer la fonction booléenne complexe, il ne reste plus que la table de vérité

et sa traduction vers une somme de monômes. C'est le cas pour le calcul du nombre premier suivant présenté aussi.

Il y a peu de règles dans l'obtention de l'assemblage. C'est une branche de l'algorithmique, sans plus. Une propriété toutefois est à retenir : la sélection, exprimée dans les algorithmes par des structures *choix* est réalisée par des multiplexeurs. Souvent ce choix commute avec d'autres opérations et cette commutation peut être exploitée pour diminuer le coût d'un circuit. Cela suppose évidemment connus les coûts des multiplexeurs et autres blocs. Ceci est illustré dans le circuit d'Unité Arithmétique et Logique.

Exemple E8.9 : Le calcul du quantième dans l'année
Cet exemple a sa source dans [SFLM93]. Il a fait l'objet d'une vraie réalisation par un groupe d'étudiants de maîtrise d'informatique dans le cadre d'un projet européen de développement de l'enseignement de la microélectronique.

Un circuit reçoit le code binaire d'une date. Cette date est composée d'un numéro de jour dans le mois, codé sur 5 bits, d'un numéro de mois, codé sur 4 bits. L'année est limitée aux deux chiffres décimaux donnant l'année dans le siècle[2]*. Chacun de ces deux chiffres décimaux est codé en binaire, selon un code DCB.*

Le circuit délivre le code binaire du quantième de la date dans l'année. Ainsi le 3 mars est le $62^{ème}$ *jour de l'année les années non bissextiles et le* $63^{ème}$ *les années bissextiles.*

Concevoir ce circuit suppose de connaître une méthode de calcul. Ici on retient la suivante qui repose sur des fonctions très spécifiques de cette application :

- *Un premier circuit* bis *délivre 1 si l'année est bissextile, 0 sinon. Il s'agit de reconnaître un multiple de 4, à partir du code DCB.*
- *Un deuxième circuit* > 2 *délivre 1 si le numéro de mois est supérieur à 2.*
- *Un circuit* Déb *donne sur 9 bits le code binaire du quantième du premier jour du mois les années non bissextiles, à partir du code du mois. On fait aisément les 9 tables de vérité correspondant à cette fonction : 1→1 ; 2→32 ; 3→60,. . .,12→334.*
- *Un additionneur ajoute le numéro du jour, le numéro du premier du mois et 1 si l'année est bissextile et si le numéro de mois est supérieur à 2.*

On remarque que plusieurs fonctions sont Φ*-booléennes car des codes binaires d'entrées ne représentent pas des valeurs du domaine.*

Exemple E8.10 : L'unité arithmétique et logique
L'unité arithmétique et logique que nous étudions reçoit deux nappes de fils A et B. Elle délivre une nappe F.

[2]Encore un système informatique avec le bogue de l'an 2000 !

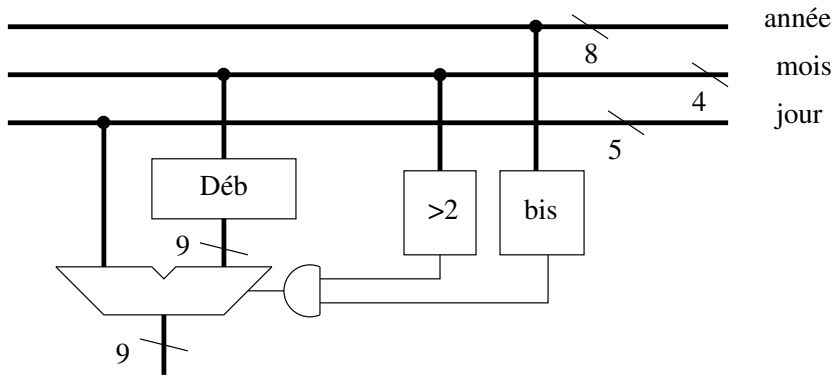


FIG. 8.15 – Circuit de calcul du quantième dans l'année

Opération souhaitée	Opération réalisée	retenue entrante	entrée d_i	entrée e_i	sortie f_i
A + B	A + B + 0	0	a_i	b_i	s_i
A - B	A + $\overline{B}$ + 1	1	a_i	$\overline{b_i}$	s_i
B div 2	B div 2 + 0 + 0	0	b_{i+1}	0	s_i
A AND B	A AND B	–	a_i	b_i	x_i

FIG. 8.16 – Opérations de l'UAL

Opération	mux 1	mux 2	mux 3	mux 4
A + B	b_i	c_i	a_i	s_i
A - B	$\overline{b_i}$	c_i	a_i	s_i
B div 2	–	0	b_{i+1}	s_i
A AND B	b_i	c_i	a_i	x_i

FIG. 8.17 – Commandes des multiplexeurs

Les nappes peuvent être interprétées comme des entiers ou des vecteurs de bits. L'UAL calcule, selon 2 bits de commande com1 com0*, la somme de A et B, la différence de A et B, le quotient de B par 2 ou, sans l'interprétation entière, le* AND *(bit à bit) des nappes A et B.*

L'UAL comporte un additionneur. En aiguillant les bonnes valeurs sur les entrées e_i, d_i et la retenue entrante de l'additionneur, on obtient les 3 résultats arithmétiques en sortie s_i de l'additionneur (Cf. Figure 8.16). En utilisant la sous-fonction AND *présente dans la fonction majorité de chaque tranche d'additionneur, on obtient la valeur $x_i = e_i$ AND d_i. Un dernier multiplexeur permet d'obtenir f_i égal soit à x_i soit à s_i.*

Les sélections des multiplexeurs 1, 2, 3 et 4 de la figure 8.18 peuvent être obtenues aisément (Cf. Figure 8.17). Il reste à exprimer les commandes de chaque multiplexeur en fonction de com1 com0*.*

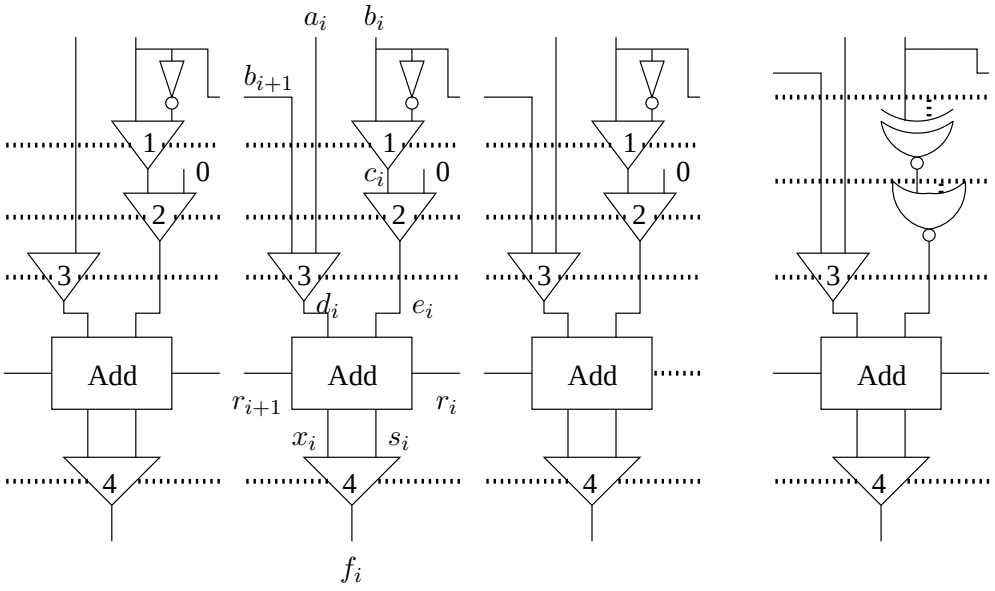


FIG. 8.18 – UAL. Les tranches représentées sont la tranche de plus fort poids, une tranche de rang intermédiaire et la tranche de poids faible. Le remplacement de certains multiplexeurs par des portes est fait dans la partie droite.

Exemple E8.11 : Le calcul du nombre premier suivant
Ceci est un exemple d'école. Un circuit reçoit le code binaire d'un naturel A codé sur N bits. Il délivre le code binaire du nombre premier immédiatement supérieur à A si A est premier, 0 sinon.

On ne connaît pas d'algorithme général donnant ce résultat. Pour réaliser le circuit il ne reste qu'une solution : pré-calculer la fonction sous forme de table, la coder en binaire sous forme de table de vérité et réaliser le circuit d'après les expressions booléennes.

Cela ne pose pas de problème pour les petites valeurs de N.

4. Etude de cas

Certains circuits peuvent donner lieu à différentes organisations car l'analyse de leur décomposition n'est pas unique. Nous montrons ici un exemple d'un tel circuit. Un exemple analogue est proposé en exercice E8.18.

Exemple E8.12 : Incrémenteur
Un incrémenteur est un circuit combinatoire qui incrémente le naturel présent en entrée. Les entrées sont une nappe de N fils $x_{N-1}, x_{N-2}, \ldots, x_1, x_0$. *Ces bits codent en binaire un naturel X. Les sorties sont une nappe de* $N+1$ *fils* $y_N, y_{N-1}, y_{N-2}, \ldots, y_1, y_0$. *Ces bits codent en binaire un naturel Y. Le circuit*

étudié doit être tel que $Y = X + 1$.

Introduisons les produits intermédiaires P_i *définis par :* $P_{-1} = 1$, $P_0 = x_0$, $P_1 = x_1.x_0$, $P_2 = x_2.x_1.x_0$, *et généralement* $P_j = \prod_{i=0}^{i=j} x_i$.

On obtient, pour tout k dans l'intervalle [0, N], $y_k = x_k \oplus P_{k-1}$ *ou, ce qui est équivalent,* $y_k = \overline{x_k} \oplus \overline{P_{k-1}}$.

La réalisation de l'incrémenteur suppose donc la réalisation des produits partiels P_i. *Si N est petit (3 ou 4), il est facile de former les* $\overline{P_i}$ *par des portes* NAND *par synthèse systématique. Plus généralement, examinons différentes solutions dans la fabrication des* P_i *avec des* NAND *et* NOR. *Les critères pris en compte sont le nombre total de portes, le nombre de niveaux logiques entre entrées et sorties et le nombre de portes (ou blocs) différentes à dessiner pour pouvoir les assembler et obtenir le dessin global du circuit.*

1) *Utilisation de portes* AND *à* $2,\ 3, \ldots, N-1$ *entrées (partie 1 de la figure 8.19) Le nombre de portes est de l'ordre de N. Le nombre de niveaux est optimal, il est de 1. La régularité est très mauvaise, chaque porte est différente des autres. Une solution consiste à dessiner une porte à N entrées et à n'en utiliser qu'une partie, mais ce n'est pas très économique. Les portes* AND *sont réalisées par des* NAND *suivies d'inverseurs. Pour N* grand *cette technique ne fonctionne que si l'on dispose de portes à nombre d'entrées quelconque. Ce n'est en général pas le cas.*

2) *Utilisation de portes* AND *à 2 entrées (partie 2 de la figure 8.19) Complexité : de l'ordre de* $2N$ *portes, de l'ordre de* $2N$ *niveaux de portes. Une seule cellule physique à dessiner (encadrée), est répétée N fois. Malheureusement la porte* AND *n'est pas une primitive physique en général ; elle est réalisée par un* NAND *suivi d'un inverseur. Cela conduit à chercher une autre solution.*

3) *Utilisation de portes* AND *à 2 entrées, alternativement réalisées par un* NAND *ou un* NOR *(partie 3 de la figure 8.19) Cette solution repose sur les égalités suivantes :*

$$u \oplus (v.w) = \overline{u} \ \oplus \ \mathsf{NAND}\ (v, w)$$
$$t \oplus (u.v.w) = t \ \oplus \ \mathsf{NOR}\ (\overline{u},\ \mathsf{NAND}\ (v, w))$$

Complexité : de l'ordre de $2N$ *portes, de l'ordre de N niveaux de portes. Une seule cellule physique à dessiner (encadrée). Elle comporte deux étages. Elle est répétée* $N/2$ *fois.*

4) *Décomposition récursive des produits partiels, à base de* AND *à 2 entrées (partie 4 de la figure 8.19) Voyons comment on passe du circuit à 8 entrées au circuit à 16 entrées. Supposons connu le circuit qui fabrique les* P_i *pour i allant de 0 à 7 à partir des* $x_7, \ldots, x_0$. *Dupliquons ce circuit et connectons-le aux entrées* $x_{15}, \ldots, x_8$. *On obtient des produits partiels* P'_i. $P'_8 = x_8$, $P'_9 = x_9.x_8$, *jusqu'à* $P'_{15} = x_{15}.\ldots.x_9.x_8$. *Il suffit d'un ensemble de portes* AND *à 2 entrées pour obtenir les* P_i *car, pour i de 8 à 15 :* $P_i = P'_i.P_7$

Le nombre de portes est de l'ordre de $2 \times N \times log_2 N$ *portes, le nombre de niveaux est de* $log_2 N$. *Chaque bloc est redessiné deux fois (encadré). Dans*

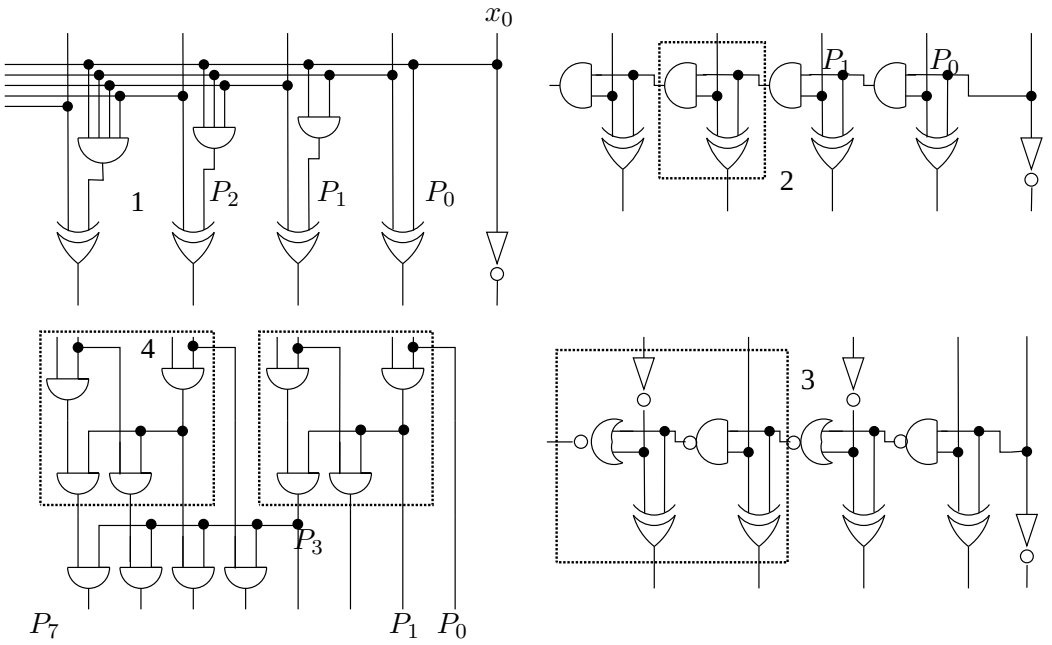


FIG. 8.19 – Différentes solutions pour la réalisation de l'incrémenteur. Les portes XOR ne sont pas dessinées dans la solution 4

chaque bloc, il faut dessiner le dernier étage de portes réalisant les AND *avec le dernier produit de l'étage précédent.*

Dans cette solution aussi il conviendrait de remplacer les cascades de AND *par des cascades de* NAND *et de* NOR*, alternativement. La mise au point de cette solution est un excellent exercice.*

5. Exercices

E8.13 : De toutes les couleurs
Reprendre les codes des couleurs dans le début du chapitre 3. Pour les couleurs codées dans les deux codes, concevoir le circuit combinatoire qui transcode dans un sens, dans l'autre.

E8.14 : Réalisation du XOR
Avec une porte $s = \overline{(ab + c)}$ et un NOR à 2 entrées, réaliser la fonction XOR. En se servant des schémas en transistor de la figure 7.9 du chapitre 7, compter les transistors de la réalisation du XOR.

Essayer d'autres expressions du XOR. Faire les schémas correspondants ; compter les transistors, comparer.

E8.15 : Des grandes portes avec des petites
En utilisant les règles de De Morgan (Cf. Chapitre 2), montrer que

$$\overline{\overline{(a+b)}.\overline{(c+d)}} = a+b+c+d$$

En déduire le schéma d'une fonction NOR à 8 entrées n'utilisant que des NAND ou NOR à 2 entrées. Penser à une organisation arborescente.

De même[3] donner le schéma d'une fonction NAND à 8 entrées n'utilisant que des NAND et NOR à 2 entrées.

Généralisation : donner la règle générale permettant de réaliser toute fonction AND, OR, NAND ou NOR à nombre quelconque d'entrées en se servant de NAND à au plus 4 entrées, de NOR à au plus 3 entrées et d'inverseurs.

E8.16 : Multiplieur
Reprendre la technique de multiplication des naturels dans le chapitre 3. Etudier le circuit de multiplication. Au lieu d'une itération en ligne, il faut penser à une itération en matrice. Il est aussi possible de donner une description récursive de la solution.

E8.17 : Un circuit combinatoire avec un cycle
Prendre un additionneur 1 bit. Reboucler le report sortant sur le report entrant. Se persuader que le circuit à deux entrées obtenu est un circuit combinatoire.

Indication : le report (sortant) est soit constant pour les entrées 00 et 11, soit égal au report entrant pour les entrées 01 et 10. On peut donc le reboucler sur le report entrant.

E8.18 : Un seul 1
Soit une nappe de N fils $x_{N-1}, x_{N-2}, \ldots, x_1, x_0$. Ce sont les entrées d'un circuit combinatoire C. La sortie S vaut 1 si et seulement si un seul des x_i vaut 1. Nous allons esquisser 5 solutions à ce problème.

- Idée 1 (fonctionne bien pour N petit) : faire la table de vérité de S, donner l'équation de S, en déduire le circuit.
- Idée 2 : concevoir un circuit C', n'ayant que $N-1$ entrées et deux sorties Z et T. Z vaut 1 si aucune des entrées ne vaut 1. T vaut 1 si une seule des entrées vaut 1. Concevoir un circuit C" qui, combiné avec C' donne un circuit ayant même comportement que C', mais N entrées. Construire C comme circuit itératif par mise en cascade de circuits C". Résoudre le cas particulier du premier étage.
- Idée 3 : supposer que N est une puissance de 2. Supposer que l'on sait faire un circuit C' à $N/2$ entrées. C' a deux sorties Z et T. Z vaut 1 si aucune des entrées ne vaut 1. T vaut 1 si une seule des entrées vaut 1. Concevoir un circuit C" qui combine les quatre sorties des deux circuits C' et délivre deux sorties Z et T. Construire C comme circuit récursif par mise en arbre de circuits C". Résoudre le cas particulier du premier étage.

[3] Après être allé au NOR, il faut qu'on pense à faire NAND (G. Brassens)

- Idée 4 : se persuader qu'un additionneur un bit donne le nombre de 1 parmi trois fils d'entrées. En utilisant un ensemble d'additionneurs un bits concevoir un circuit qui donne le nombre de 1 parmi une nappe de fils. Concevoir un circuit qui détecte si ce nombre de 1 est supérieur à un, ou nul. Simplifier le circuit qui calcule le nombre de 1 pour tenir compte du fait que dans cet exercice on n'a pas besoin du nombre de 1, seulement de savoir s'il est supérieur à un ou nul.
- Idée 5 : concevoir un circuit qui reçoit une nappe de N fils et délivre une nappe de N fils. Les deux nappes sont ordonnées (de droite à gauche, de haut en bas...). La notion de *premier* fait référence à cet ordre. Les sorties de ce circuit soit sont toutes à 0, soit sont toutes à 0 sauf une, celle dont le rang est le rang du premier 1 de la nappe d'entrée.

  Utiliser deux tels circuits pour la nappe des x_i, l'un pour un ordre, l'autre pour l'ordre inverse. Si il y a un seul 1, le premier 1 dans un sens est aussi le premier 1 dans l'autre sens.

Etudier les 5 solutions du point de vue du nombre de portes, du nombre d'étages de portes pour N valant 4, 16, 64 et 256. S'aider d'un outil de Conception Assistée par Ordinateur.

Chapitre 9

Eléments de mémorisation

Un ordinateur est muni de composants permettant de stocker les données et les programmes ; nous avons parlé du tableau MEM au chapitre 4 et nous reparlerons de mémoire dans les chapitres ultérieurs.

Des éléments de mémorisation sont aussi nécessaires pour réaliser des machines séquentielles telles que nous les avons définies au chapitre 5 (Cf. Chapitres 10, 11 et 14).

D'une façon générale, il s'agit de savoir comment réaliser la fonction d'affectation des langages de haut niveau : x $\longleftarrow$ f(a, b, c). La *mémorisation* peut avoir lieu chaque fois qu'une des variables a, b ou c est modifiée (comportement asynchrone) ou à des instants fixés par une entrée spécifique ne dépendant pas des autres entrées (comportement synchrone). Nous avons déjà parlé de ces aspects au chapitre 6 et dans ce livre nous nous limitons aux circuits synchrones. Un cas particulier de ce type d'affectation est : x $\longleftarrow$ f(x, e) où les deux instances du nom x correspondent aux valeurs de x sur une même nappe de fils, à des instants différents. Ce n'est pas une équation de point fixe. On la lit par exemple, comme dans un langage de programmation usuel, nouveau-x $\longleftarrow$ f (ancien-x, e).

Dans ce chapitre nous étudions les éléments de mémorisation permettant de résoudre les problèmes abordés ci-dessus sous les deux aspects :

- la vision externe, fonctionnelle, où nous voyons qu'un processeur connecté à de la mémoire peut *écrire une information dans la mémoire* ou *lire une information précédemment mémorisée* en envoyant des signaux de commande à cette mémoire. Une mémoire ne permet que les affectations de type x $\longleftarrow$ f(a, b, c) ; on ne peut pas lire et écrire à un même emplacement dans la mémoire d'un ordinateur dans le même instant.
- la vision interne, structurelle, où nous expliquons comment de la mémoire peut être fabriquée à partir d'éléments de mémorisation de base (nous partons du bistable étudié au chapitre 7). Nous montrons aussi quels éléments de mémorisation conviennent pour réaliser une affectation du type x $\longleftarrow$ f(x, e). Ces éléments sont aussi utilisés dans les dispositifs de traitement de l'information dont naturellement les processeurs (Cf. Chapitre 14).

Dans le paragraphe 1. nous présentons les composants élémentaires utilisés pour mémoriser de l'information et pour construire des circuits séquentiels. Nous présentons ensuite la notion de mémoire dans un ordinateur (paragraphe 2.) puis comment est construite une mémoire à partir de cellules ou points mémoires élémentaires (paragraphe 3.). Le paragraphe 4. présente des optimisations et des réalisations de mémoire particulières.

1. Points de mémorisation de bits : bascules et registres

Dans le chapitre 7, nous avons décrit le comportement électrique de certains points mémoire. Considérons une chaîne de $2k$ ($k \geq 1$) inverseurs, la sortie de l'un étant connectée à l'entrée du suivant. Si nous rebouclons la sortie de la chaîne d'inverseurs sur l'entrée, nous obtenons un circuit séquentiel à deux états stables, ou *bistable*. Nous nous intéressons au cas $k = 1$. Tel que (Cf. Figure 9.1-a), le bistable ne peut que fournir sa valeur à un autre composant, il n'est pas possible de le charger avec une valeur particulière ; il peut être lu, mais on ne peut y écrire.

Nous allons étudier deux réalisations permettant de forcer une valeur en entrée du circuit. La première consiste à remplacer les inverseurs par des portes NOR (ou NAND). On obtient ainsi un circuit avec deux entrées de commandes, appelé *bascule RS* (voir la figure 9.1-b pour le montage). La deuxième consiste à intercaler un multiplexeur entre les deux inverseurs ; ce montage est appelé *verrou* construit à partir du bistable (Cf. Figure 9.4-a).

Par essence, les points de mémorisation sont des circuits où l'une des sorties reboucle sur l'une des entrées. Cela conduit à des équations du type $\mathsf{x} = \mathsf{f}(\mathsf{x}, \mathsf{e})$, où les deux occurrences de x dénotent des valeurs de x à des instants différents. Pour distinguer une variable x à un instant et la même à l'instant suivant, nous écrivons x'. L'équation précédente devient : $\mathsf{x}' = \mathsf{f}(\mathsf{x}, \mathsf{e})$: la nouvelle valeur de x est fonction de son ancienne valeur et de e.

> *Comme nous l'avons précisé en introduction, dans ce livre nous nous limitons aux circuits synchrones. Lorsque nous utiliserons des éléments de mémorisation dans des assemblages complexes (Cf. Chapitres 10, 11 et 14), les valeurs des variables seront examinées à des instants définis par un signal en général périodique appelé horloge.*

1.1 Points de mémorisation élémentaires : bascule RS, verrou

1.1.1 Bascule RS

Une *bascule RS* possède deux entrées R (*Reset*) et S (*Set*) permettant de forcer l'état respectivement à 0 ou à 1, et deux sorties Q1 et Q2.

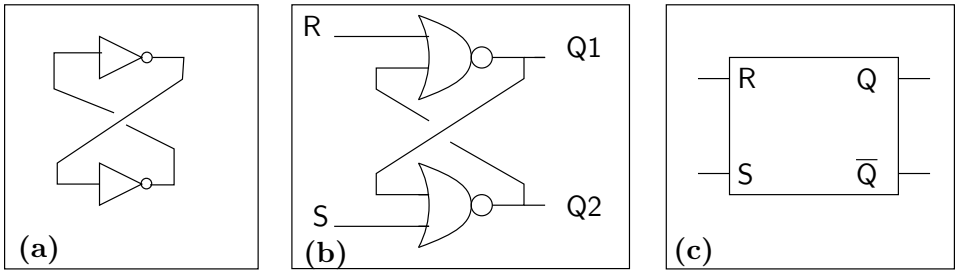


FIG. 9.1 – a) le bistable b) la bascule RS réalisée avec des portes NOR c) le symbole logique de la bascule RS

Nous allons détailler le fonctionnement d'une bascule RS réalisée à partir de portes NOR (voir pour le montage la figure 9.1-b) :

- Lorsque R et S sont stables à 0, la bascule est équivalente à un bistable. En effet, $\mathrm{NOR}(x, 0) = \overline{x}$.
- A partir de cet état stable, le passage de R à 1 fait passer Q1 à 0, après un délai de commutation de la porte NOR. Après un autre délai, Q2 passe à 1. Lorsque R repasse à 0, alors l'état reste stable.
- Par symétrie, à partir de l'état stable, le raisonnement est le même. Lorsque S passe à 1, Q2 passe à 0 et Q1 passe à 1, à ceci près que Q2 change avant Q1.
- Lorsque S (respectivement R) est à 1 et que R (respectivement S) passe à 1, les sorties Q1 et Q2 sont à 0. Cette situation n'évolue pas tant que les deux entrées restent stationnaires. Cela constitue le plus souvent une erreur d'initialisation.

Remarquons que lorsque R et S ne sont pas tous deux à 1, Q1 et Q2 sont complémentaires, ce qui justifie les noms habituels Q et $\overline{\mathrm{Q}}$.

La stabilisation des sorties ne peut avoir lieu exactement au même instant que le changement de l'entrée, à cause du temps de commutation de chaque porte. Il existe ainsi un délai de stabilisation de la bascule, délai faible et borné. Nous ne prenons pas en compte de façon chiffrée ce délai mais seulement son existence et notons que les valeurs en entrée et en sortie sont considérées à des instants successifs. Plus précisément, nous notons Q1, Q2 les valeurs de la bascule à un instant donné et Q1', Q2' les nouvelles valeurs, à un instant immédiatement ultérieur.

La table d'évolution des valeurs de la bascule est donnée ci-dessous, la première ligne se lisant : si les entrées S et R sont à 0, la sortie Q1 reste à la valeur qu'elle avait précédemment et donc Q1' = Q1. A partir de cette table, on obtient les équations données à côté.

Du schéma de la figure 9.1-b, on tirerait les équations : $Q1' = \overline{R + Q2}, Q2' = \overline{S + Q1}$. D'où $Q1' = \overline{R + \overline{S + Q1}} = \overline{R}.(S + Q1)$ et $Q2' = \overline{S + \overline{R + Q2}} = \overline{S}.(R + Q2)$.

S	R	Q1'	Q2'
0	0	Q1	Q2
1	0	1	0
0	1	0	1
1	1	0	0

$$\begin{aligned} Q1' &= Q1.\overline{S}.\overline{R} + S.\overline{R} \\ &= \overline{R}.(Q1.\overline{S} + S) \\ &= \overline{R}.(Q1 + S) \\ Q2' &= Q2.\overline{S}.\overline{R} + \overline{S}.R \\ &= \overline{S}.(Q2.\overline{R} + R) \\ &= \overline{S}.(Q2 + R) \end{aligned}$$

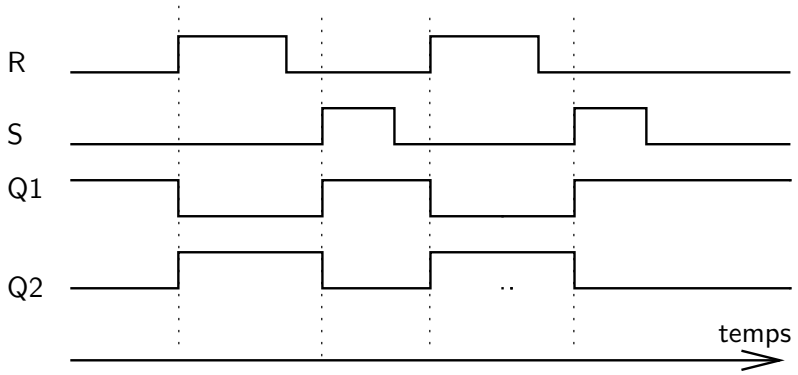


FIG. 9.2 – Chronogramme du comportement logique idéal d'une bascule RS. Les pointillés représentent les instants de changement de Q1 et Q2.

On remarque que si $R.S \neq 1$, $Q1 = \overline{Q2}$ ce qui justifie le schéma logique de la figure 9.1-c. En revanche, dans le cas où R et S valent 1, les sorties Q1 et Q2 ne sont pas complémentaires.

Le chronogramme de la figure 9.2 montre le comportement temporel logique de la bascule RS.

Avec la bascule RS à portes NOR, la remise à 0 est prioritaire sur la mise à 1 : en effet, $Q' = \overline{R}.(Q + S)$. Donc, si R vaut 1, la sortie Q passe à 0. Sinon, la sortie est conditionnée à la valeur de S : si S vaut 1, la sortie Q vaut 1 sinon 0.

La bascule RS peut être aussi réalisée avec des portes NAND. Les entrées de commande sont actives à 0 (lorsque R=S=1, la nouvelle valeur est égale à l'ancienne, si R vaut 0 et S vaut 1, Q passe à 0, si R vaut 1 et S vaut 0, Q passe à 1) et la mise à 1 est prioritaire sur la mise à 0.
La table d'évolution de la bascule et sa définition équationnelle sont alors :

S	R	Q'	$\overline{Q'}$
1	1	Q	$\overline{Q}$
1	0	0	1
0	1	1	0
0	0	1	1

$$\begin{aligned} Q' &= \overline{\overline{R.Q}.S} = R.Q + \overline{S} \\ \overline{Q'} &= \overline{\overline{S.\overline{Q}}.R} = S.\overline{Q} + \overline{R} \end{aligned}$$

1.1.2 Verrou

Un *verrou* (Cf. Figure 9.4) possède une entrée de donnée D (pour *Data*), qui est la valeur à mémoriser, et une entrée de commande En (pour *Enable*). Lorsque l'entrée En est active (En=1), le verrou est dit *transparent* et sa sortie Q est égale à la valeur de l'entrée D après un petit délai appelé *temps de traversée du verrou*. Lorsque En est à 0, le montage est équivalent à un bistable. La sortie Q est figée et sa valeur est celle de l'entrée D au moment du front descendant de En. La définition équationnelle du verrou D est : $Q' = En.D + \overline{En}.\ Q$. Le chronogramme de la figure 9.3 illustre ce comportement.

Le verrou peut être réalisé de plusieurs façons, par exemple à partir d'un bistable en intercalant un multiplexeur entre les deux inverseurs (Cf. Figure 9.4-a). L'équation déduite de la figure est : $Q' = En.D + \overline{En}.\overline{\overline{Q}}$. On retrouve là l'équation du verrou en notant que $\overline{\overline{Q}}$=Q.

Un autre montage peut être envisagé en rebouclant directement la sortie du multiplexeur sur son entrée. En effet, la réalisation d'un multiplexeur demande l'utilisation de portes, induisant ainsi un délai de commutation lorsqu'on effectue le rebouclage de la sortie sur l'entrée. Il est donc possible de ne pas intercaler d'inverseurs (en nombre pair) entre la sortie du multiplexeur et l'entrée. Dans la figure 9.4-d, on montre une réalisation de verrou à partir d'un multiplexeur, lui-même réalisé à partir de portes NAND. Remarquons que nous retrouvons cette réalisation à partir de l'équation : $Q' = En.D + \overline{En}.Q = \overline{\overline{En.D}.\overline{\overline{En}.Q}}$.

Nous proposons une dernière réalisation d'un verrou à partir d'une bascule RS. Nous l'obtenons en transformant l'équation précédente :

$$\begin{array}{rcl} Q' & = & \overline{\overline{En.D}.\overline{\overline{En}.Q}} \\ & = & \overline{\overline{D}.En + \overline{Q}.(\overline{En} + \overline{D})} \\ & = & \overline{En.\overline{D}.(Q + En.D)} \end{array} \quad = \quad \begin{array}{l} \overline{(\overline{En} + \overline{D})(En + \overline{Q})} \\ \overline{En.\overline{D} + \overline{Q}.\overline{En.D}} \end{array}$$

En rapprochant cette équation de celle de la bascule RS à portes NOR : $Q' = \overline{R}.(Q + S)$, avec $R = En.\overline{D}$ et $S = En.D$, on obtient la réalisation donnée dans la figure 9.4-b. Notons que, par construction, ce montage interdit $R = S = 1$.

1.2 Points de mémorisation pour les circuits séquentiels : bascule maître-esclave, bascule sensible au front

Nous avons dit précédemment que nous souhaitions réaliser des fonctions de mémorisation permettant des affectations de la forme $x \longleftarrow f(x, e)$ en nous limitant aux systèmes synchrones où la progression des valeurs de x est cadencée par un signal (généralement périodique) appelé *horloge*.

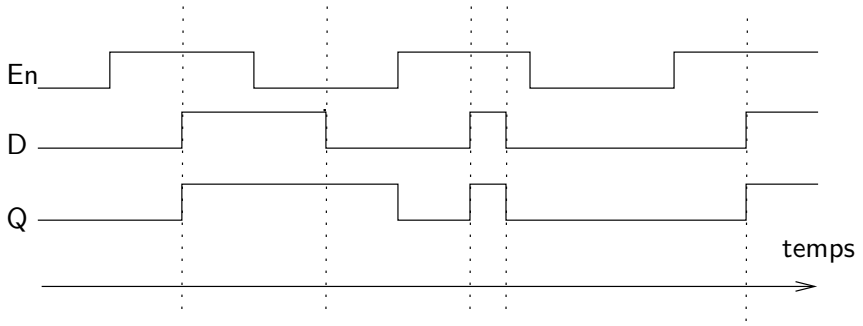


FIG. 9.3 – Chronogramme du comportement logique d'un verrou

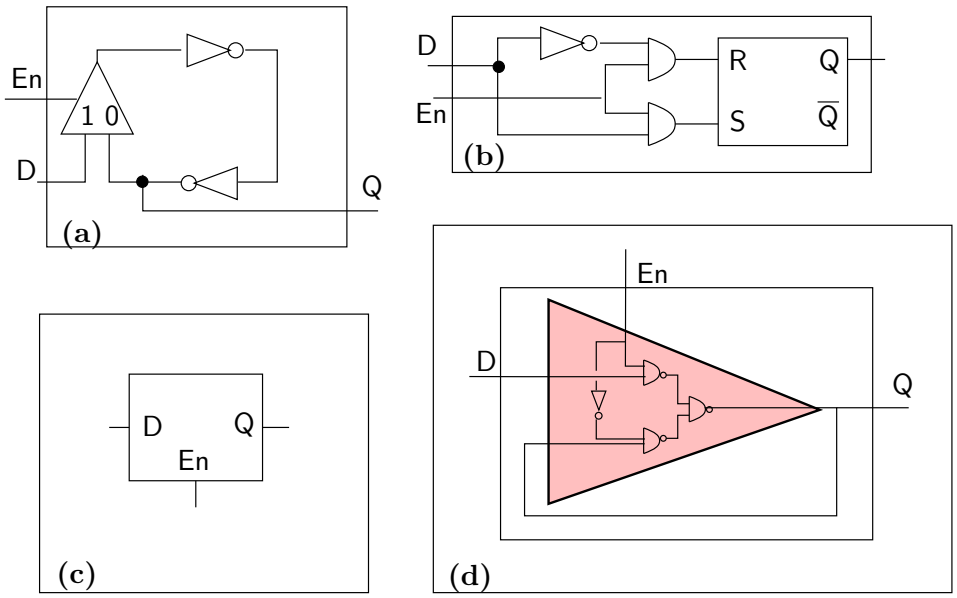


FIG. 9.4 – Trois réalisations d'un verrou de type D et son symbole logique. a) réalisation d'un verrou à partir d'un bistable et d'un multiplexeur, b) réalisation d'un verrou à partir d'une bascule RS, c) symbole logique d'un verrou, d) réalisation d'un verrou à partir d'un multiplexeur seul (en gris est représenté l'intérieur du multiplexeur).

Un verrou ne convient pas pour ce genre de réalisation car il ne permet pas de commander les instants où la mémorisation a lieu. Dans ce paragraphe, nous précisons pourquoi le verrou ne convient pas et nous montrons des solutions pour résoudre le problème.

1.2.1 Problème de rebouclage du verrou

Etudions le circuit dont l'équation est $x=\overline{x.e}$, réalisé avec un verrou, dont l'entrée d'activation est connectée au signal périodique En (par exemple l'horloge) et l'entrée D est reliée à la sortie d'une porte NAND à deux entrées. Cette porte NAND a pour entrée e et la sortie Q du verrou. Nous avons les équations suivantes :

$$Q' = En.D + \overline{En}.Q \qquad D = \overline{e.Q}$$

Pendant que En est à 1, si l'entrée e vaut 1, on a Q'=D et D=$\overline{Q}$. Si En reste à 1 pendant un temps supérieur au temps de traversée de la porte NAND, la sortie Q et l'entrée D peuvent passer successivement de 1 à 0 un nombre indéterminé de fois et donc fournir un résultat incohérent.

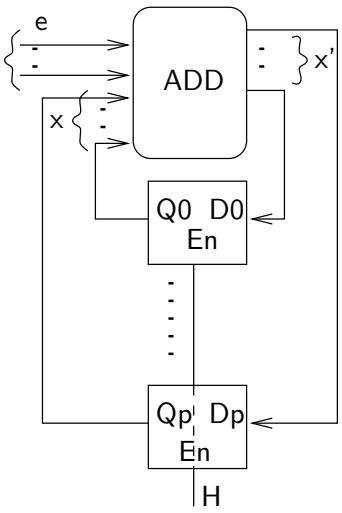
Illustrons ce comportement incohérent dans un circuit réel effectuant un calcul sur des nombres codés sur n bits. Supposons que l'on veuille réaliser $x' \longleftarrow x + e$ avec x et e entiers (Cf. Figure 9.5-a). Nous étudierons de façon systématique ces fonctions au chapitre 10. On veut que l'horloge H, connectée à l'entrée En des verrous, cadence les évolutions de x aux instants i_0, i_1, i_2, i_3, i_4, ...Notons x_0, x_1, x_2, x_3 et x_4 les valeurs successives de x. Les fils d'entrée e, eux, changent n'importe quand par rapport aux instants fixés par H. D'après les valeurs de e observées aux instants i_1, i_2, i_3 et i_4 (Cf. Figure 9.5-b), les valeurs de x à ces mêmes instants sont respectivement : $x_1 = x_0 + 1$, $x_2 = x_1 + 1$, $x_3 = x_2 + 3$ et $x_4 = x_3 + 7$.

Observons le bit de poids faible de x mémorisé dans le verrou d'entrée D0 et de sortie Q0. Il change à chaque addition puisqu'on ajoute des nombres impairs ; donc D0 = $\overline{Q0}$.

Si l'horloge H vaut 0, le verrou est stable, il ne se passe rien. Quand H vaut 1, le verrou est transparent et Q0 suit les variations de D0. Dans le circuit combinatoire qui fabrique D0, D0 = $\overline{Q0}$, et donc D0 passe alternativement de 1 à 0 et de 0 à 1. (Cf. Figure 9.5-c).

On ne peut pas contrôler combien de fois l'inverseur inverse pendant que H est à 1. Ceci peut donner une valeur quelconque lorsque H repasse à 0. Le verrou ne peut donc pas être à la base de réalisation du comportement $x \longleftarrow f(x, e)$.

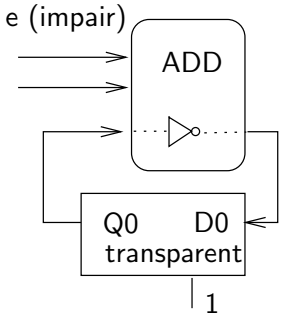
Réduire la durée de l'état haut de l'horloge pour éliminer le problème n'est pas réaliste. La solution consiste alors à construire une *bascule*, c'est-à-dire un dispositif pour lequel l'état transparent est limité à un très court instant au moment du front montant (ou descendant) de l'horloge.



(a) Réalisation de l'affectation x' ⟵ x + e
x est représenté sur p booléens
mémorisés dans p verrous

instants fixés par H	i_0	i_1	i_2	i_3	i_4
x	x_0	x_1	x_2	x_3	x_4
e	1	1	3	7	....

(b) valeurs de x et e aux instants fixés par H



(c) H=1 : verrou transparent

FIG. 9.5 – Problème posé par la nature transparente d'un verrou

1.2.2 Bascule de type maître-esclave

Une bascule de type *maître-esclave* est construite en connectant en série deux verrous commandés par des signaux complémentaires. Les figures 9.6 et 9.7 donnent respectivement le montage et un exemple de chronogramme.

Le premier verrou, appelé maître, mémorise l'entrée D lorsque En1, c'est-à-dire H, est à 1 : la sortie Q1 suit l'entrée D (D1=D). Pendant ce temps, la valeur mémorisée par le second verrou reste stable, puisque En2=0. Lorsque H prend la valeur 0, le contenu du premier verrou reste figé et est transféré dans le second verrou qui devient actif (En2=1) et mémorise donc la valeur précédemment stockée dans le premier verrou. Ainsi, la sortie Q reste stable pendant que le signal H est à 1 ou pendant que le signal H reste à 0. La sortie Q change lorsque le signal H passe de 1 à 0. Le temps pendant lequel H est à 1 doit être supérieur au *temps de traversée du verrou*.

1.2.3 Bascule D à front

Une bascule D à front a une entrée de donnée D, une entrée d'activation H et une sortie Q. La *bascule D à front montant* (respectivement descendant) est caractérisée par le fait que sa sortie Q est stable entre deux fronts montants (respectivement descendants) du signal connecté sur H, en général une horloge. La valeur de la sortie est celle de l'entrée D au moment du dernier front montant (respectivement descendant) de H. Il est donc nécessaire que l'entrée D soit stable pendant le front. Une bascule à front montant (respectivement des-

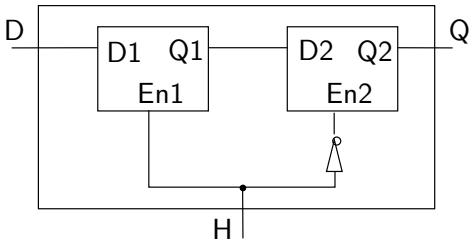


FIG. 9.6 – Bascule de type maître-esclave

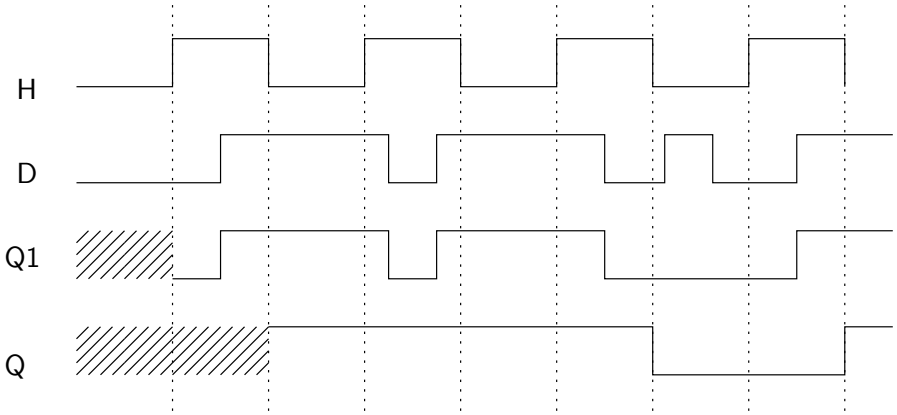


FIG. 9.7 – Chronogramme décrivant le comportement de la bascule maître-esclave. Avant le premier front montant de H, Q1 est indéterminé.

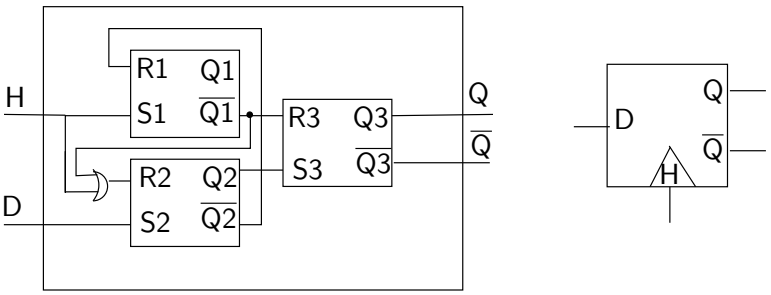


FIG. 9.8 – Une réalisation de la bascule de type D à front descendant et son symbole logique

cendant) peut être réalisée à partir de 3 bascules RS NAND (respectivement NOR).

Nous décrivons le comportement de la bascule à front descendant. La réalisation avec 3 bascules RS à portes NOR est donnée dans la figure 9.8. Il y a deux niveaux de bascules :

– la bascule en sortie RS3, dont les entrées sont pilotées par les sorties des bascules RS1 et RS2, et

– les deux bascules RS1 et RS2, dont les entrées sont pilotées par H et D.

Les équations de la bascule à front sont :

$$\begin{array}{lcll} S2 & = & D & (1) \\ \overline{Q1} & = & \overline{S1}.(\overline{Q1}+R1) & (3) \\ R3 & = & \overline{Q1} & (5) \\ S3 & = & Q2 & (7) \\ R1 & = & \overline{Q2} & (9) \\ \overline{Q3} & = & \overline{S3}(\overline{Q3}+R3) & (11) \end{array} \qquad \begin{array}{lcll} S1 & = & H & (2) \\ R2 & = & H+\overline{Q1} & (4) \\ Q2 & = & \overline{R2}(Q2+S2) & (6) \\ \overline{Q2} & = & \overline{S2}(\overline{Q2}+R2) & (8) \\ Q3 & = & \overline{R3}(Q3+S3) & (10) \end{array}$$

Nous allons montrer que la sortie ne change pas entre deux fronts descendants de H.

Nous faisons l'hypothèse que D reste stable pendant que H passe de 1 à 0.

Considérons l'état initial H=1, qui précède le front descendant de H. En appliquant les équations, nous obtenons les résultats partiels suivants : $S1 = 1, \overline{Q1} = 0, R2 = 1,$ $\boxed{R3 = 0}$, $Q2 = 0,$ $\boxed{S3 = 0}$. Donc, la bascule RS3 ne change pas d'état et la sortie Q3 est stable.

Supposons qu'à l'état initial, on ait en plus D=0. Il en résulte que S2=0, $\overline{Q2}$=1, R1=1. Lors du passage de H à 0, nous obtenons S1=0 d'après (2). Puisque R1=1 d'après (3), nous obtenons $\overline{Q1}$=1. D'où $\boxed{R3=1}$. Par ailleurs, le fait que R2=1 entraîne Q2=0 d'après (4), et donc $\boxed{S3=0}$. La sortie de la bascule Q3 est 0. Ensuite, tant que H reste à 0, $\overline{Q1} = 1$ et R2=1. Il s'ensuit que $\boxed{R3=1}$ et $\boxed{S3=0}$. La sortie reste à 0. Lorsque H repasse à 1, $\overline{Q1}$=0 d'après (2) et (3), et R2=1 d'après (4). Donc $\boxed{R3}$ passe à 0 et $\boxed{S3}$ reste à 0 : la sortie reste inchangée.

Si à l'état initial D=1, alors S2=1, $\overline{Q2}$=0, R1=0. Nous obtenons $\overline{Q1} = 0$. Comme R2=0 et S2=1, Q2=1. La bascule RS3 est forcée à 1. Par un raisonnement analogue au cas où D=0 à l'état initial, la sortie Q3 reste stable.

Nous avons montré que, si l'entrée D reste stable pendant que H passe de 1 à 0, la sortie Q3 reste stable jusqu'au front descendant suivant.

Le chronogramme de la figure 9.9 montre l'évolution de la sortie Q de la bascule à front descendant en fonction de l'entrée D.

Du point de vue de la réalisation, en technologie CMOS, la bascule à front utilise autant de transistors qu'une bascule maître-esclave c'est-à-dire deux fois plus qu'un verrou.

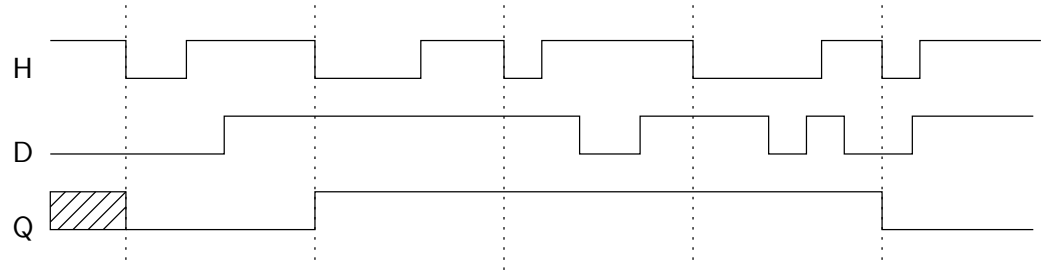


FIG. 9.9 – Chronogramme décrivant le comportement de la bascule à front descendant

1.2.4 Un exemple de circuit utilisant des bascules à front : le détecteur de passage

Le détecteur de passage, nécessaire lorque l'on veut repérer le passage d'un signal de 1 à 0, est un montage séquentiel qui échappe aux techniques de réalisation systématique que nous étudions au chapitre 10.

La figure 9.10 décrit un dispositif permettant la détection d'un passage. La sortie Q1 prend la valeur de l'entrée e aux fronts descendants de H et la sortie Q2 recopie Q1 aux fronts montants de H. Q1 est stable entre deux fronts descendants. La sortie Q2 prend ainsi la valeur de la sortie Q1 avec une demi-période de retard.

Supposons que les deux bascules soient initialement à 0. Si l'entrée e passe à 1, Q1 et S passent à 1. Après une demi-période, Q2 passe à son tour à 1 et S passe à 0. On obtient une impulsion de S d'une demi-période après chaque transition de 0 à 1 de l'entrée.

1.3 Autres commandes associées à une bascule

Pour certains circuits complexes, il est indispensable d'introduire une nouvelle entrée dite de *commande de chargement* sur les bascules. L'ensemble des bascules a l'entrée d'horloge connectée à l'entrée d'horloge du circuit et des groupes de bascules peuvent avoir l'entrée de chargement en commun.

En interne, le signal d'horloge est l'entrée d'horloge de la bascule (H) et le signal de chargement (Ch) commande un multiplexeur (Cf. Figure 9.11). Notons que dans certaines documentations l'entrée de chargement est appelée *enable* ce qui peut entraîner une certaine confusion avec le verrou.

On peut ajouter d'autres signaux, par exemple, pour l'initialisation Preset (ou Set) et Clear (ou Reset) forcent respectivement la valeur de la bascule à 1 et à 0. Dans le chapitre 10, on fera figurer sur les bascules les entrées de commande et d'initialisation. Selon la structure interne de la bascule, ces entrées sont considérées soit à un front (initialisation synchrone), soit dès qu'elles sont actives (initialisation asynchrone).

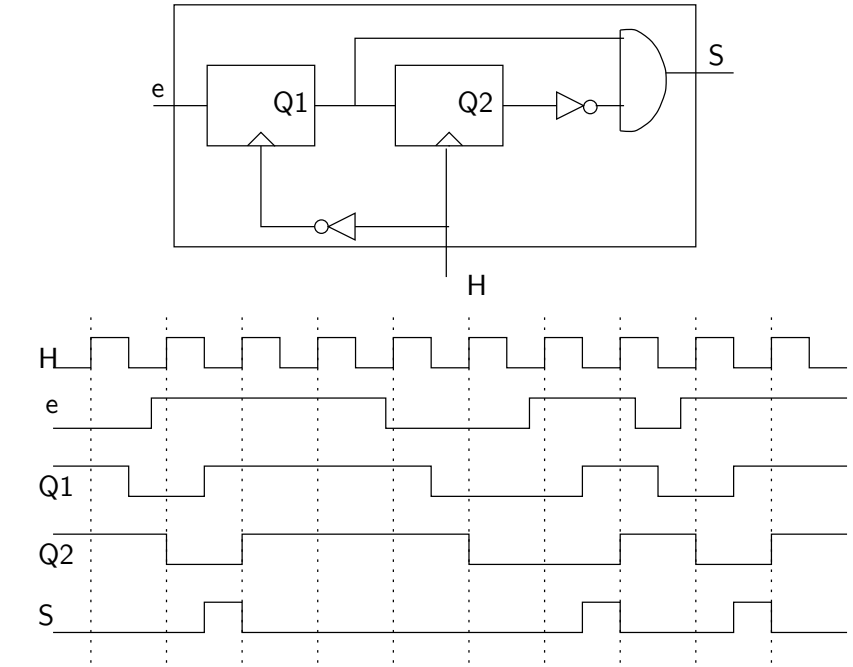


FIG. 9.10 – Détecteur de passage de 0 à 1

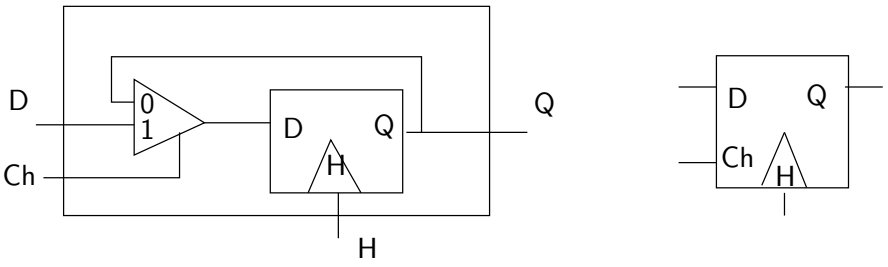


FIG. 9.11 – Une réalisation d'une bascule avec commande de chargement et son symbole logique

1.4 Notion de registre

Un verrou à n bits permet de stocker des informations codées sur plus d'un bit. On l'obtient par simple juxtaposition de verrous élémentaires commandés par le même signal de chargement.

Sur le même principe, en utilisant des bascules D, on obtient un *registre* à n bits.

La nappe des n booléens peut être interprétée comme un vecteur de bits mais aussi comme un nombre, un caractère, etc. (Cf. Chapitre 3).

> *Dans les chapitres 11 et 14, nous verrons l'utilisation de tels registres dans la réalisation des parties opératives. Dans la suite et en particulier dans ces chapitres, nous considérons systématiquement des registres fabriqués à partir de bascules à front et le plus souvent avec une commande de chargement.*

2. La mémoire : organisation matricielle des points de mémorisation

Tout ordinateur est doté de mémoires plus ou moins grandes à accès plus ou moins rapide. Ce paragraphe présente la mémoire du point de vue externe, celui de l'utilisateur. Nous expliquons ce que sont un *mot mémoire* et un *accès mémoire*, et nous donnons une idée des différents types de mémoire.

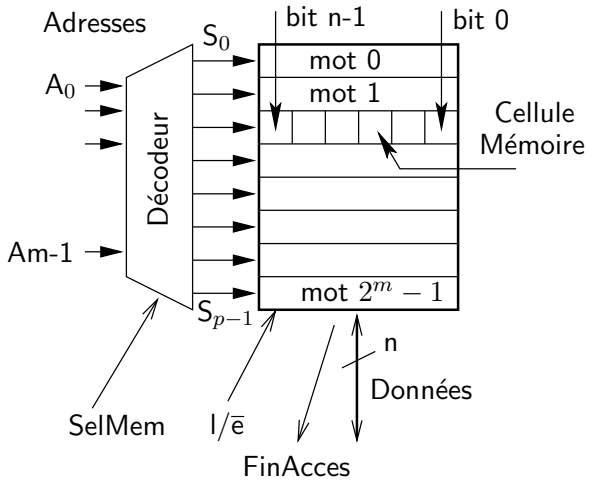
2.1 Notion de mémoire dans un ordinateur

Une *mémoire* est l'organisation d'un ensemble de points de mémorisation élémentaires en matrice à p lignes et n colonnes. On peut ainsi voir la mémoire comme l'assemblage de $n \times p$ bits. Mais on l'utilise comme un tableau de p éléments de taille n auxquels on accède par indice. Une ligne est appelée *mot de la mémoire* et on parle d'une mémoire de p mots de n bits.

A chaque mot, c'est-à-dire à l'ensemble des n points de mémorisation élémentaire qui le composent, est associé un fil dit de *sélection du mot*. La sélection d'un mot consiste ainsi à mettre à 1 le fil de sélection associé.

L'interface de la mémoire est composée de p fils de sélection $S_0, ..., S_{p-1}$. Lors d'un accès un seul des $S_0, ..., S_{p-1}$ doit valoir 1. De plus, un fil permet de préciser si l'accès souhaité est une lecture ou une écriture. Le signal spécifiant le sens de l'accès est noté $\mathsf{l}/\overline{\mathsf{e}}$ ($\mathsf{r}/\overline{\mathsf{w}}$ en version anglaise) ; s'il est à 1 il s'agit d'une lecture (read) et s'il est à 0 c'est une écriture (write). Une telle notation a déjà été vue au chapitre 8 pour le signal $\mathsf{Add}/\overline{\mathsf{Sub}}$.

Habituellement, le mot auquel le processeur accède est désigné par un numéro (compris entre 0 et $p-1$) appelé *adresse*. Si $p = 2^m$, l'adresse est codée sur m bits $(A_{m-1}, ..., A_0)$ et un décodeur associé à la mémoire réalise la fonction de calcul de l'unique fil de sélection valant 1 ; ainsi, si les m bits d'adresse $A_{m-1}, ..., A_0$ codent l'entier i $(0 \leq i \leq 2^m - 1)$, le fil de sélection de numéro i vaut 1 et tous les autres valent 0.



SelMem	l/ē	Opération
0	0	pas d'accès
0	1	pas d'accès
1	0	écriture
1	1	lecture

FIG. 9.12 – Mémoire de 2^m mots de n bits et signification des signaux SelMem et l/ē.

De plus, un accès à la mémoire est matérialisé par l'activation d'un signal de *sélection mémoire* que nous notons dans la suite SelMem.

La figure 9.12 représente une mémoire de 2^m mots de n bits ainsi que l'opération réalisée selon la valeur des signaux SelMem et l/ē.

> **Remarque :** Dans certains processeurs, les signaux de commande de la mémoire sont définis de façon différente avec, par exemple, deux signaux lec et ecr. L'accès à la mémoire en lecture est réalisé par la commande : lec ET $\overline{\text{ecr}}$ et l'accès en écriture par la commande : $\overline{\text{lec}}$ ET ecr ; $\overline{\text{lec}}$ ET $\overline{\text{ecr}}$ signifie qu'il n'y a pas d'accès à la mémoire, et lec ET ecr n'a aucun sens (et doit être évité).

La valeur à stocker dans la mémoire (cas d'une écriture) ou à extraire de celle-ci (cas d'une lecture) est appelée la *donnée* (de taille n).

Le processeur dialogue avec la mémoire via les signaux de contrôle SelMem, FinAcces et l/ē, et via le *bus mémoire* comprenant les adresses et les données. On parle de *bus d'adresses* et de *bus de données*.

2.2 Déroulement d'un accès mémoire par un processeur

Nous considérons ici un ordinateur composé d'un processeur et d'une mémoire vive, avec les hypothèses simplificatrices suivantes :

1. Les adresses sont sur m bits et les données sur n bits. Les adresses sont des adresses de mots de n bits et les accès mémoire sont limités aux seuls mots de n bits. Le cas général permettant l'accès à des sous-ensembles du mot mémoire est étudié dans le chapitre 15. L'accès à des sur-ensembles du mot mémoire, en mode *rafale*, est étudié dans le paragraphe 4.3 du présent chapitre.

2. La taille de mémoire physique et la capacité d'adressage du processeur sont identiques. En général, la capacité d'adressage du processeur est supérieure à la taille de la mémoire physique ; une même adresse risque alors de correspondre à plusieurs mots mémoire. Nous étudions cette situation au chapitre 15.

Le raccordement des signaux entre processeur et mémoire est très simple : le bus de données est connecté aux entrées et sorties des données de la mémoire, le bus d'adresse aux entrées de sélection de mot. Le bus de données est bidirectionnel alors que le bus d'adresses est monodirectionnel. L'entrée $\mathsf{l}/\overline{\mathsf{e}}$ de la mémoire est reliée au signal de même nom du processeur, et l'entrée d'activation de la mémoire SelMem au signal de demande d'accès à la mémoire du processeur AccesMem. La sortie FinAcces est reliée au signal du même nom du processeur.

1. Lors d'une écriture, le processeur 1) affiche sur le bus d'adresses le numéro de l'emplacement mémoire auquel il accède ; 2) affiche l'information à écrire sur le bus de données ; 3) met à 0 le signal $\mathsf{l}/\overline{\mathsf{e}}$; 4) met à 1 le signal AccesMem.

   A l'intérieur de la mémoire, le décodeur d'adresses sélectionne l'emplacement correspondant, active le dispositif d'écriture et désactive la sortie du circuit de lecture. Pour chaque bit du mot dont la nouvelle valeur diffère de l'ancienne, le bistable mémoire change d'état. Le délai maximal de commutation définit le *temps d'accès en écriture* de la mémoire. Le signal FinAcces est alors émis. A la fin de l'écriture, le processeur met à 0 le signal AccesMem.

2. Dans le cas d'une lecture, le processeur 1) affiche sur le bus d'adresses le numéro de l'emplacement mémoire auquel il accède ; 2) met à 1 le signal $\mathsf{l}/\overline{\mathsf{e}}$; 3) met à 1 le signal AccesMem.

   A l'intérieur de la mémoire, le décodeur d'adresse sélectionne l'emplacement correspondant, désactive le dispositif d'écriture et active la sortie du circuit de lecture. Après un certain délai, dont la borne supérieure est le *temps d'accès en lecture*, la valeur lue se stabilise sur le bus de données. Le signal FinAcces est alors émis. A la fin de la lecture, le processeur mémorise la valeur stabilisée sur le bus de données dans un registre (ou un verrou) interne et met à 0 le signal AccesMem.

Entre deux cycles d'accès mémoire, le signal AccesMem vaut 0 et les signaux d'adresses, de données et $\mathsf{l}/\overline{\mathsf{e}}$ ne sont pas significatifs.

Si un accès à la mémoire dure un seul cycle d'horloge du processeur et si le temps de cycle de la mémoire est inférieur ou égal à ce dernier, on peut simplifier le protocole de communication : la mémoire n'émet pas l'acquittement FinAcces pour signifier explicitement la fin d'un accès. Le processeur demande l'accès, signal émis sur sa propre horloge, et la lecture ou l'écriture sont supposées être effectives lors du prochain top d'horloge du processeur.

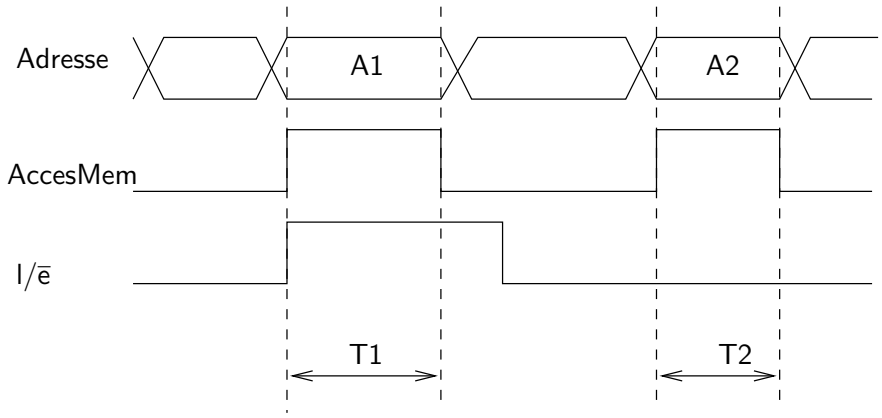


FIG. 9.13 – Chronogrammes décrivant l'accès à la mémoire. L'intervalle T1 correspond à la lecture du mot mémoire d'adresse A1 ; l'intervalle T2 correspond à l'écriture du mot mémoire d'adresse A2.

La figure 9.13 montre une évolution possible des différents signaux, données et adresses intervenant lors d'un accès à la mémoire par le processeur.

2.3 Typologie de mémoires

On peut donner une première classification de la mémoire en considérant l'ordre dans lequel le processeur accède aux données. La plupart des mémoires centrales offrent un *accès aléatoire* : les données peuvent être lues ou écrites à n'importe quel emplacement (en anglais *RAM* ou *Random Access Memory*). Le temps d'accès à une valeur est pratiquement indépendant de l'emplacement dans lequel elle est stockée.

Les bandes, cartouches et cassettes magnétiques sont au contraire d'excellents exemples de dispositifs à *accès séquentiel* (Cf. Chapitre 19). Pour accéder à une donnée située en fin de bande, il faut d'abord parcourir la totalité de la bande et des données qui précèdent. Le temps d'accès est proportionnel à l'éloignement de l'information sur la bande.

Les mémoires peuvent être classées selon leurs fonctionnalités. Une distinction est faite entre *ROM* (*Read Only Memory*) et RWM (*Read Write Memory*). Les premières étant accessibles en lecture seule, les secondes en lecture et écriture. Dans les RWM, les données sont mémorisées dans des *points mémoires statiques* (bascules) ou *dynamiques* (capacités). Dans la mesure où un point mémoire dynamique peut être réalisé avec moins de transistors, pour une même surface, une mémoire dynamique aura une plus grande capacité de stockage. En revanche, elle devra être rafraîchie régulièrement.

La structure des ROM est généralement basée sur un autre principe : l'information est codée dans la structure du circuit en ajoutant ou retranchant des transistors (Cf. Chapitre 7). La structure étant figée, la mémoire ne peut être modifiée. De plus, la déconnexion électrique du dispositif ne modifie pas

les données mémorisées.

Pour des raisons historiques, le sigle RAM est utilisé à la place de RWM (Read Write Memory).

3. Réalisation des mémoires statiques

3.1 Décomposition de la mémoire globale d'un ordinateur en boîtiers et barettes

Nous avons présenté la mémoire d'un ordinateur comme un tableau de 2^m mots de n bits (Cf. Paragraphe 2.). En pratique, on cherche à minimiser le nombre de broches. On va construire la mémoire à l'aide de plusieurs boîtiers pour obtenir la capacité voulue. On peut envisager deux stratégies :

- considérer un boîtier de capacité une colonne de 2^m mots de 1 bit et juxtaposer les boîtiers. Cela donne, par boîtier, un décodeur ayant m entrées d'adresse, une entrée SelMem, une entrée $\mathsf{l}/\overline{\mathsf{e}}$ et une sortie représentant le bit sélectionné. Le schéma est analogue à la figure 9.12, en considérant une mémoire de 2^m mots de 1 bit. En juxtaposant n boîtiers, partageant les mêmes entrées, on obtient une barette de capacité $2^m \times n$.
- intégrer les mots les plus longs possibles jusqu'à la taille n. Dans ce cas, nous considérons p boîtiers de 2^{k_i} mots de n bits, tels que $\Sigma_{i=1}^{p} 2^{k_i} = 2^m$. Nous obtenons ainsi une décomposition de la mémoire en tranches horizontales.

La première solution est plus souple et peut s'adapter à des processeurs de tailles différentes (16, 32 ou 64 bits). De plus, le nombre de broches est optimisé : il y a m broches d'adresses et n broches de données. Si on ajoute une broche de donnée, on passe d'une capacité de $2^m \times n$ à une capacité de $2^m \times (n+1)$. Si on ajoute une broche d'adresse, on passe d'une capacité de $2^m \times n$ à $2^{m+1} \times n$. Toutes les mémoires de grande capacité sont organisées suivant ce schéma. Dans la suite, on assimilera une barette de n boîtiers de 1 bit à un boîtier de n bits.

3.2 Réalisation physique

3.2.1 Principe d'une réalisation avec verrous

Nous donnons une réalisation interne d'un boîtier mémoire de 2^m mots de 1 bit à l'aide de verrous et de portes 3 états (Cf. Figure 9.14-a). Ce boîtier a $m+3$ entrées qui sont : l'adresse du mot (A_{m-1}, ...A_0), le bit de donnée Don, le signal SelMem, et le signal $\mathsf{l}/\overline{\mathsf{e}}$. Le boîtier comporte un décodeur qui sert à sélectionner le bon verrou : si A_{m-1}, ...A_0=i, mot_i=1. A l'aide du signal $\mathsf{l}/\overline{\mathsf{e}}$, on sélectionne le sens de transfert de la donnée : si ce signal est à 1, alors la valeur sur le fil de donnée Don est recopiée dans la bascule sélectionnée. Si ce

signal est à 0, la porte 3 états en sortie de bascule sélectionnée est activée et le contenu de la bascule est recopié sur le fil de donnée Don.

La réalisation de chaque bit met en jeu 2 sorties pour le décodeur, 1 verrou, 2 portes ET, 1 inverseur et une porte 3 états.

3.2.2 Cellule de mémoire statique

La figure 9.14-b montre une autre solution : la cellule mémoire à bistable et forçage par court-circuit.

Le processeur sélectionne la cellule de numéro i en activant le mot de ligne (mot_i), qui connecte via les deux transistors C1 et B1, les inverseurs aux colonnes v et $\overline{v}$ définissant la valeur d'un bit.

En *lecture* (SelMem vaut 1 et $\mathsf{l}/\overline{\mathsf{e}}$ vaut 1), la valeur stockée dans la cellule (côté gauche du bistable) et son complément (côté droit du bistable) apparaissent respectivement sur les colonnes v et $\overline{v}$ avec une dégradation des signaux logiques. Le comparateur analogique détecte la colonne sur laquelle la tension est la plus élevée et donne la valeur stockée dans la cellule. Cette valeur est envoyée en sortie (Don) de la mémoire via un amplificateur 3 états activé par le produit des signaux $\mathsf{l}/\overline{\mathsf{e}}$ et SelMem.

En *écriture* (SelMem vaut 1 et $\mathsf{l}/\overline{\mathsf{e}}$ vaut 0) on impose un zéro sur un des côtés du bistable en reliant une des colonnes v et $\overline{v}$ à la masse via un des deux transistors B2 ou C2. Le signal de commande du transistor B2 ou C2 est le produit du signal de sélection du boîtier (SelMem), du signal d'écriture ($\overline{\mathsf{l}/\overline{\mathsf{e}}}$) et du signal d'entrée (Don pour B2 ($\overline{v}$) et $\overline{\mathsf{Don}}$ pour C2 (v)).

- La colonne v est reliée à la masse si C2 est passant, ce qui est le cas lorsque Don vaut 0. Pendant ce temps, B2 est bloqué. Si de plus mot_i est à 1, le transistor C1 est passant et le côté gauche du bistable est forcé à la masse. Ceci installe un 1 du côté droit.
- La colonne $\overline{v}$ est reliée à la masse si B2 est passant, ce qui est le cas lorsque Don vaut 1. Si de plus mot_i est à 1, le transistor B1 est passant et le côté droit du bistable est forcé à la masse. Ceci installe un 1 du côté gauche.

3.2.3 Organisation de cellules de mémoire en matrice

On peut qualifier une mémoire en fonction de son *débit* : c'est le nombre de mots auxquels on accède par seconde. Considérons une mémoire de 2^{20} mots (20 bits d'adresse) de 1 bit organisée comme nous l'avons vu précédemment. On peut organiser cette mémoire, par exemple, comme une matrice (Cf. Figure 9.15) de 2048 (2^{11}) lignes de 512 (2^9) bits. La mémoire est ainsi constituée de lignes, une ligne étant sélectionnée grâce aux 11 bits de poids forts de l'adresse, et un étage de décodage des informations d'une colonne, la colonne étant sélectionnée grâce aux 9 bits de poids faibles de l'adresse. Le débit est alors amélioré puisque pendant le décodage des colonnes, il est possible de commencer le décodage d'une nouvelle ligne.

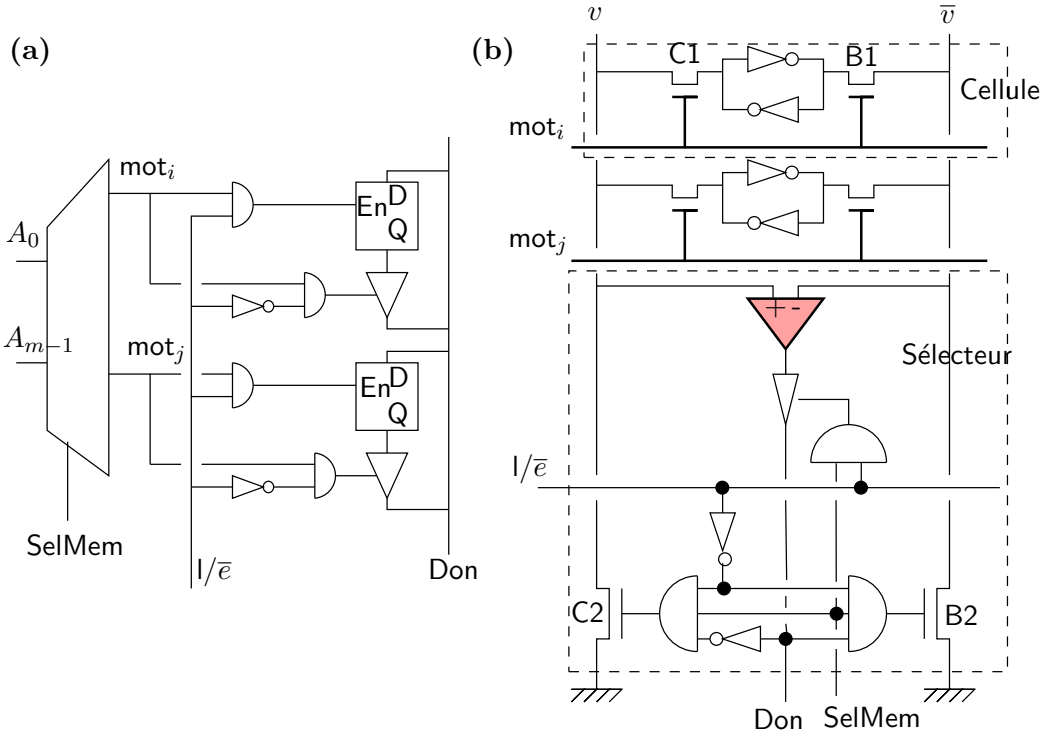


FIG. 9.14 – Deux réalisations de points de mémoire. a) à partir de verrous et de portes, b) à partir de bistables (le triangle en gris est un comparateur analogique).

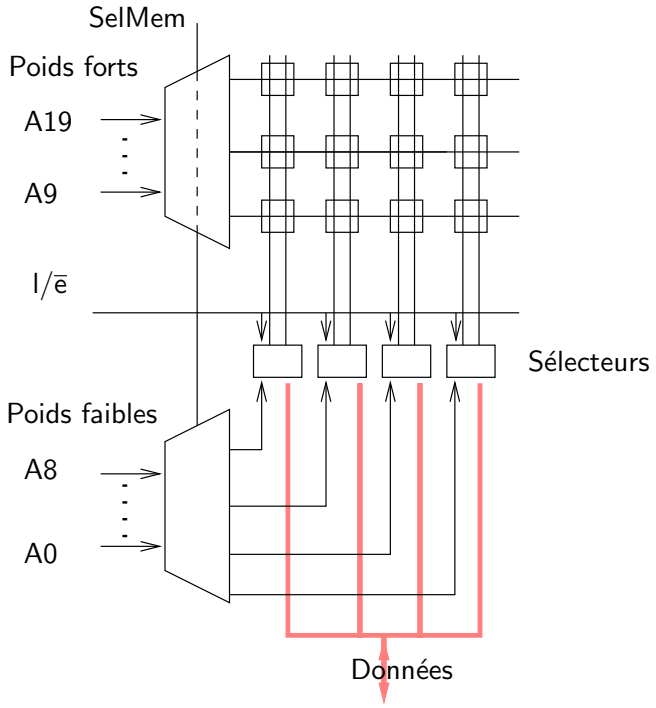


FIG. 9.15 – Matrice de Cellules construite à partir de bistables

Nous verrons dans le paragraphe 4. qu'une telle organisation permet aussi d'optimiser l'accès à des données appartenant à une même colonne.

4. Optimisations et techniques particulières

Il existe divers types de réalisation des mémoires. Nous en présentons quelques-unes et à la suite de l'observation du paragraphe 3.2.3, nous montrons quelques approches permettant d'améliorer encore le débit de la mémoire.

4.1 Multiplexage lignes/colonnes

Dans le cas d'une organisation de la mémoire telle que nous l'avons vue à la figure 9.15, on peut diminuer le nombre de broches de moitié. En effet, il ne sert à rien d'activer une colonne avant que la ligne ne soit sélectionnée.

L'idée est de réutiliser les broches servant à la sélection des lignes pour réaliser la sélection des colonnes. Il convient alors que le nombre de fils soit le même et on fabrique ainsi des matrices carrées de cellules.

Etant donné $m/2$ broches et une adresse codée sur m bits (m étant pair), les $m/2$ bits de poids forts codent une ligne et les $m/2$ bits de poids faibles une colonne. Le circuit reçoit les $m/2$ bits de poids forts, qui sont mémorisés et reliés au décodeur des lignes. Puis, pendant ce décodage, le circuit reçoit les $m/2$ bits de poids faibles qui sont reliés au décodeur des colonnes.

Ce circuit est réalisé à partir d'une matrice de cellules, en utilisant deux signaux supplémentaires RAS (Row Address Strobe) et CAS (Column Address Strobe). La ligne (respectivement la colonne) est sélectionnée au front descendant de RAS, i.e. $\overline{\mathsf{RAS}}$ (respectivement $\overline{\mathsf{CAS}}$). Le temps d'accès à une cellule mémoire est la somme du temps de sélection d'une ligne et du temps de sélection d'une colonne.

4.2 Mémoires dynamiques

Les mémoires dynamiques sont organisées en matrices tout comme les mémoires statiques. Dans une cellule, l'information y est codée sous forme de charge électrique stockée dans la capacité grille-source d'un transistor MOS.

La capacité de la cellule de mémoire dynamique se décharge lentement et l'information stockée disparaît avec le temps. Pour éviter cela, chaque ligne est périodiquement lue et réécrite en totalité. Ce processus, connu sous le nom de *rafraîchissement*, est effectué sur chaque ligne toutes les 2 à 4 ms. Dans une mémoire de 16Mbits (4096 lignes) de 50 ns de temps de cycle, le rafraîchissement représente de l'ordre d'un accès ligne par microseconde, ce qui consomme environ 5% du débit théorique de la mémoire. La cellule dynamique ne nécessite que deux transistors et un seul signal de colonne. Cela autorise la construction de mémoires de plus grande capacité.

Par ailleurs, les mémoires dynamiques sont dotées d'un registre interne de stockage de numéro de ligne (adresses de poids fort), ce qui permet d'économiser la moitié des broches d'adresse sur le boîtier au prix d'un dispositif externe de multiplexage (commun à tous les boîtiers).

L'accès mémoire se déroule en deux temps : le numéro de ligne est envoyé le premier et stocké dans un verrou interne. Le temps de décodage et l'établissement de la connexion entre la ligne sélectionnée et les signaux de colonne est mis à profit pour transmettre la deuxième partie de l'adresse (numéro de colonne) au boîtier. Notons que dans la salve d'accès à différentes colonnes d'une même ligne, l'étape de sélection et de connexion de la ligne aux colonnes peut être effectuée en une seule fois en début de salve. Cette optimisation est applicable à toute suite d'accès mémoire à des adresses ne différant que par les poids faibles, qui correspondent au numéro de colonne (Cf. Paragraphe 4.3). A partir de ce principe, certaines mémoires ont été conçues spécialement pour la réalisation de cartes vidéo (Cf. Paragraphe 4.5).

4.3 Mode rafale

Le multiplexage ligne/colonne permet d'économiser non seulement des broches mais aussi du temps : une fois l'accès à une ligne réalisé, l'accès à des colonnes dans cette ligne est rapide.

Si l'intervalle d'adresses auquelles on accède appartient à une même ligne, on accède à la première adresse par l'intermédiaire des poids forts, puis on accède à chaque colonne. Si on réalise N accès consécutifs à des éléments appartenant à la même ligne, le temps d'accès total est égal à : Temps d'accès ligne + Temps d'accès colonne * N.

On parle d'*accès en mode rafale*. Il existe de nombreuses manières de réaliser le mode rafale ; nous n'en donnons ici que quelques principes. Par exemple, pour le mode dit *quartet*, la mémoire est dotée d'un circuit interne qui compte modulo quatre ; à chaque impulsion de $\overline{\text{CAS}}$, le circuit incrémente le numéro de colonne modulo 4 : on accède à quatre informations consécutives. Le mode dit *page* permet de sélectionner n'importe quelle colonne dans une ligne : une page correspond à une ligne.

Nous étudions l'accès mémoire un peu plus précisément en considérant deux types d'interface entre la mémoire et le processeur : asynchrone et synchrone. Dans les deux cas, nous nous intéressons à la lecture de 4 données de la même ligne. Nous ne représentons pas le signal de lecture/écriture, il est échantillonné en même temps que l'adresse de colonne par CAS.

Lorsque l'interface est asynchrone RAS joue le rôle de SelMem et CAS permet la sélection des colonnes auxquelles on accède dans la rafale. Considérons les chronogrammes de la figure 9.16. Le signal $\overline{\text{RAS}}$ reste actif durant la sélection des colonnes d'une ligne.

Soit R-C une adresse composée d'un numéro de ligne R et d'un numéro de colonne C. Soient a, b, c et d les mots d'adresses R-C1, R-C2, R-C3 et R-C4.

FIG. 9.16 – Chronogrammes décrivant l'accès mémoire en mode rafale, pour une interface asynchrone

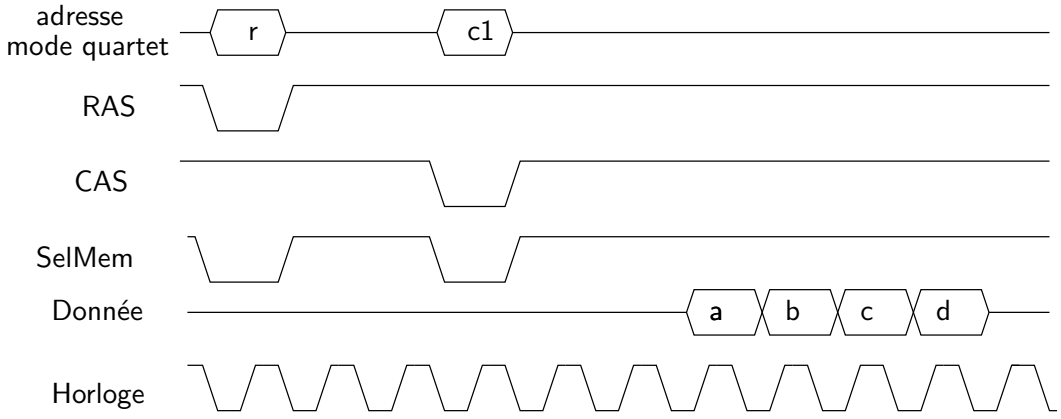


FIG. 9.17 – Chronogrammes décrivant l'accès mémoire en mode rafale, pour une interface synchrone

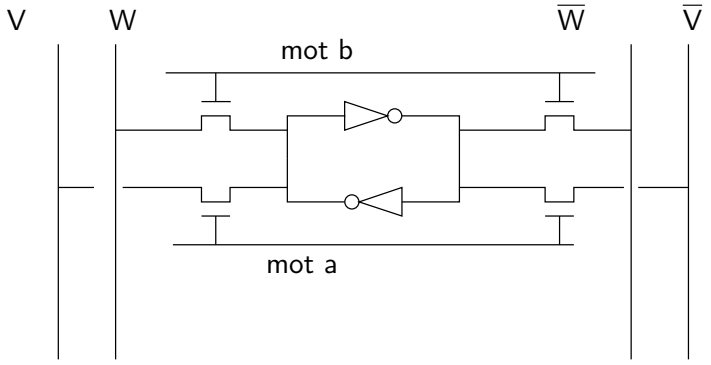


FIG. 9.18 – Schéma interne d'une cellule de mémoire à 2 accès simultanés

En mode page, le processeur envoie le numéro de ligne (R), puis celui de la colonne C1, celui de la colonne C2, celui de la colonne C3 et enfin celui de la colonne C4 (C1, C2, C3 et C4 pouvant être quelconques).

En mode quartet, le processeur envoie seulement le numéro de la première colonne C1 après celui de la ligne R. On accède à des emplacements consécutifs et c'est le circuit mémoire qui incrémente en interne le numéro de colonne : (C4 = C3 + 1, C3 = C2 + 1, C2 = C1 + 1).

Avec une interface synchrone (Cf. Figure 9.17), le processeur et la mémoire sont synchronisés sur la même horloge. L'accès mémoire se déroule un peu comme dans le mode quartet décrit précédemment à quelques différences près : il y a un signal SelMem en plus de RAS ; tout est cadencé par une horloge de bus H, dérivée de celle du processeur ; la longueur des rafales est définie en initialisant un registre de commande avant utilisation.

Lors de la commande d'initialisation de la mémoire, les nombres de cycles d'horloge entre RAS et CAS et entre CAS et la stabilisation des données sont définis en fonction du temps d'accès de la mémoire (qui est fixe), et de la période d'horloge qui peut varier avec la fréquence du bus. Par exemple, à 33 Mhz, la mémoire sera capable de fonctionner sans cycle d'attente entre RAS, CAS et la stabilisation des données. A 66 Mhz, on intercalera un cycle d'attente pour avoir le même temps d'accès.

4.4 Mémoires à plusieurs accès

Le principe est d'accéder simultanément à deux (ou plus dans le cas d'accès multiple) emplacements mémoire. On a autant de décodeurs, de signaux $\mathsf{l}/\overline{\mathsf{e}}$, de sélection de boîtiers SelMem et de bus de données, que d'accès. De plus, on rajoute un comparateur pour vérifier qu'il n'y a pas d'accès simultanés au même emplacement mémoire en écriture. Le schéma interne d'une cellule d'une telle mémoire est donné figure 9.18.

Les mémoires à n accès permettent de réaliser des bancs de n registres utilisés par exemple dans la partie opérative du processeur.

4.5 La mémoire vidéo

Le processeur est connecté à de la mémoire et l'ensemble processeur/mémoire est lui-même connecté à des périphériques permettant le dialogue avec le monde extérieur. L'écran et le clavier dont dispose tout utilisateur sont deux périphériques particuliers. Nous détaillons au chapitre 16 les aspects connexion et synchronisation, et au chapitre 17 les problèmes posés par la gestion de périphériques de plus en plus élaborés. Nous nous intéressons ici à l'un d'entre eux, posant des problèmes de gestion mémoire : l'écran graphique.

L'image affichée par un écran graphique est construite sous la forme d'une matrice à deux dimensions indiquant les points (ou pixels) de l'écran à allumer. Chaque élément de cette matrice, stocké dans une mémoire appelée *mémoire d'écran*, définit l'intensité et la couleur du pixel correspondant de l'écran.

L'intérieur du tube cathodique est recouvert d'une substance qui émet de la lumière lorsqu'elle est frappée par un faisceau d'électrons qui balaie la surface de l'écran, ligne après ligne. Le contenu de la mémoire d'écran est donc transmis séquentiellement au dispositif qui module l'intensité du faisceau. L'image générée est par nature fugitive et doit être rafraîchie (réaffichée) périodiquement, cinquante à cent fois par seconde.

La mémoire d'écran est une partie de la mémoire principale à laquelle on accède en écriture par le processeur pour modifier l'image affichée, et en lecture par le dispositif de rafraîchissement de l'écran. Dans certains cas, cette mémoire d'écran n'est accessible qu'à un processeur spécialisé, le processeur graphique.

En utilisant les techniques présentées dans les paragraphes précédents, le débit de la mémoire resterait insuffisant. Il faut organiser l'accès différemment. L'idée consiste à transférer un paquet de mémoire important vers le périphérique et à lui déléguer le travail consistant à calculer les pixels à afficher ; essentiellement, il s'agit de réaliser des décalages sur les données fournies au périphérique (le paquet de mémoire transféré).

On appelle *mémoire vidéo* une mémoire optimisée au niveau temps d'accès (matrice), et pourvue d'un registre interne sur lequel agit un circuit séquentiel permttant d'effectuer les décalages nécessaires à l'affichage des pixels aux instants fixés par les contraintes du balayage écran.

Chapitre 10

Circuits séquentiels

Un *circuit séquentiel* possède, comme un *circuit combinatoire* (Cf. Chapitre 8), un ensemble d'entrées et un ensemble de sorties. Un circuit séquentiel est un circuit dont les valeurs de sortie à l'instant présent dépendent de la séquence des valeurs d'entrée qu'il y a reçues depuis l'instant initial. Il se distingue ainsi d'un circuit combinatoire dans lequel les valeurs de sortie à l'instant présent dépendent seulement des valeurs d'entrée présentes à cet instant (après le délai de stabilisation dû à la traversée des portes logiques). Le circuit séquentiel possède une mémoire lui permettant de stocker des informations sur la succession des valeurs d'entrée. Ces informations constituent l'*état courant* du circuit séquentiel à un instant donné.

Un circuit séquentiel comporte ainsi des éléments de mémorisation (Cf. Chapitre 9) dotés d'une fonction permettant de fixer l'*état initial*. La valeur écrite dans ces éléments de mémorisation est fonction de celle qui y était à l'instant précédent : l'état suivant est une fonction de l'état courant et des valeurs d'entrée. Des circuits combinatoires permettent de calculer les sorties du circuit et l'évolution de son état.

Ce type de circuit permet de réaliser un comportement qui peut être décrit à l'aide d'*un automate d'états fini* (Cf. Chapitre 5) ou d'un *algorithme*. On peut parler de *machine algorithmique*.

Le nombre d'états, d'entrées et de sorties du circuit à concevoir sont très variables suivant la complexité de l'application. Cela détermine le choix de la méthode de conception. On retrouve les deux familles de solution évoquées au chapitre 8.

Dans le cas où l'algorithme peut être décrit de façon simple par un automate d'états fini le point de départ de la *synthèse* est le graphe explicite de l'*automate d'états fini*. La méthode de conception dépend du type de l'automate, de la bibliothèque de circuits combinatoires disponibles, et du type d'éléments de mémorisation utilisés ; nous nous limitons ici aux bascules de type D présentées au chapitre 9. On est proche ici de la synthèse logique.

Dans le cas plus général où la construction du graphe de l'automate correspondant à l'algorithme est impossible pour cause de trop grande complexité,

la conception du circuit se fait selon des procédés différents (Cf. Chapitre 11). On est proche ici de l'algorithmique câblée. Deux grands types d'*architectures* (organisations matérielles) des circuits séquentiels sont alors employés.

Dans l'un, la partie qui permet de stocker les variables de l'algorithme et de réaliser les calculs sur ces variables (*partie opérative*) est séparée de la partie commandant l'enchaînement de ces opérations (*partie contrôle*). Ces deux parties sont des circuits séquentiels.

Dans l'autre type d'architecture, les aspects de contrôle et de calcul sont mélangés. Ce sont les valeurs des variables (les données) qui contrôlent directement l'enchaînement des opérations sur celles-ci. On parle d'architecture à *flots de données* (*Data flow* en anglais). Des architectures à flots de données sont illustrées par des exemples dans ce chapitre. Le cas particulier des organisation à *pipeline* est introduit.

La méthode de synthèse basée sur une partie contrôle et une partie opérative est présentée en détail au chapitre 11.

Ces méthodes de conception de circuits sont aujourd'hui automatisées grâce à des outils de CAO de circuits. La réalisation se fait à partir des spécifications des algorithmes dans différents langages. Le plus courant, VHDL (devenu un standard) permet de décrire des spécifications de circuits séquentiels à différents niveaux : graphes d'automates d'états fini et algorithmes à base d'instructions de types divers (itératif, conditionnel ...).

Nous définissons dans le paragraphe 1. la notion de circuit séquentiel en précisant son architecture et en décrivant son comportement temporel. Dans le paragraphe 2. nous étudions en détail les méthodes de réalisation de circuits séquentiels à partir du graphe explicite d'un automate d'états fini (Cf. Chapitre 5). Nous détaillons ici deux types de synthèse : câblée et microprogrammée. Dans le paragraphe 3. nous décrivons deux exemples de réalisations par flots de données de circuits séquentiels à partir d'un algorithme. Nous donnons aussi une idée de la notion de pipeline.

1. Notion de circuit séquentiel

1.1 Caractérisation

Un circuit séquentiel mémorise des informations qui lui permettent de réagir à une séquence d'entrées. Les sorties à un instant donné ne dépendent plus seulement des entrées présentes à cet instant, mais aussi de la séquence des entrées qu'il y a reçues depuis un instant initial. Pour définir cet instant initial le circuit comporte une entrée particulière souvent appelée init.

Le changement des entrées peut être pris en compte soit à n'importe quel moment (celui où l'entrée change réellement), soit à des instants déterminés et réguliers dépendant d'une entrée particulière (*horloge* ou *clock*). Dans le premier cas on parle de circuits séquentiels *asynchrones*, dans le deuxième de

circuits séquentiels *synchrones*. La conception de circuits asynchrones, beaucoup plus délicate, continue d'être aujourd'hui utilisée pour des applications ayant des contraintes temporelles ou de consommation critiques. Pour des raisons pédagogiques, nous nous limitons dans ce livre à la conception de circuits de type synchrone.

Dans le cas synchrone, le circuit a besoin d'une entrée définissant les instants successifs de prise en compte des valeurs des entrées de données. C'est en général un signal régulier de période fixe. Le circuit est synchronisé sur cette horloge : son état évolue vers un nouvel état sur un des fronts (montant ou descendant) de l'horloge.

> **Remarque :** Dans la suite nous appelons *entrées* les entrées de données, les autres entrées sont désignées par leur nom spécifique (*init* et *clock*).

L'état courant de l'automate est mémorisé à l'aide de bascules sensibles au front (Cf. Chapitre 9) dont l'entrée d'activation est l'horloge. L'état suivant dépend de l'état courant et des entrées présentes à l'instant courant. Les sorties dépendent soit de l'état courant (modèle de Moore), soit de l'état courant et des entrées présentes à l'instant courant (modèle de Mealy). Les deux modèles sont présentés dans le chapitre 5.

1.2 Architecture générale

La figure 10.1 décrit l'architecture générale du circuit réalisant un automate dans le cas des deux modèles de Moore et de Mealy. Cette architecture peut être décomposée en 3 blocs aux fonctionnalités distinctes :

- Un bloc de bascules permet de mémoriser l'*état* courant de l'automate. Il donne ainsi en sortie la valeur de l'état courant et prend en entrée la valeur de l'état suivant. Ces bascules sont sensibles au front de l'entrée particulière clock : le passage de l'état courant au suivant est cadencé par cette entrée clock. Les bascules peuvent être initialisées à une valeur donnée (état initial) grâce à l'entrée init.
- Un bloc permet de calculer la *fonction de sortie* de l'automate. Si l'automate est de Mealy les sorties dépendent des entrées courantes et de l'état courant. Si l'automate est de Moore les sorties ne dépendent que de l'état courant.
- Un bloc permet de calculer la *fonction de transition* de l'automate : il donne l'état suivant à partir de l'état courant et des entrées courantes.

1.3 Comportement temporel

1.3.1 Echantillonnage des entrées et fréquence de l'horloge

Pour que l'automate fonctionne correctement il est indispensable que l'entrée des bascules soit stabilisée au moment du front d'activation du signal clock. Si ce n'est pas le cas la valeur de la sortie de ces bascules est indéterminée (Cf. Chapitre 9). Deux cas peuvent se présenter :

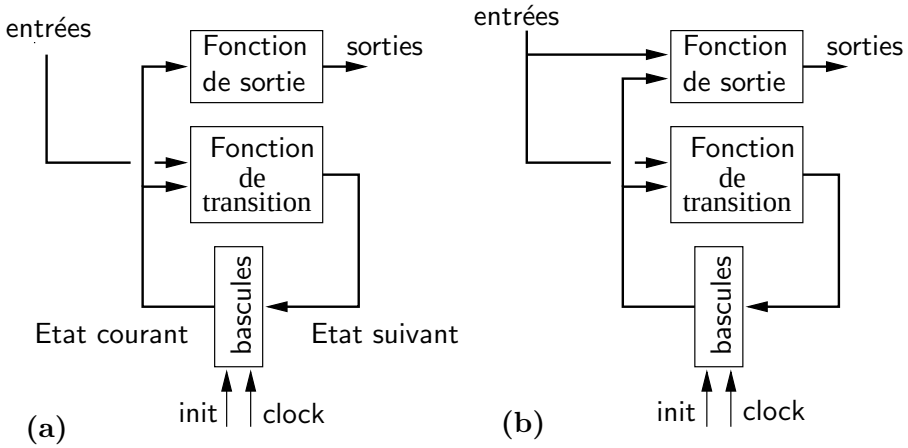


FIG. 10.1 – Architecture générale d'un circuit réalisant un automate d'états fini. a) Modèle de Moore ; b) modèle de Mealy.

– le système en amont (fournissant les entrées) ne connaît pas l'horloge, c'est le cas par exemple lorque les entrées proviennent de capteurs sur un monde extérieur. Nous avons vu dans le chapitre 9 comment mettre en place un mécanisme simple pour obtenir un échantillonnage à partir de l'horloge.
– le système en amont est déjà synchronisé sur la même horloge que l'automate. C'est le cas par exemple de composants d'un même ordinateur. Les différents composants ont en entrée la même horloge. On verra au paragraphe 1.3.3 comment réaliser la synchronisation entre deux systèmes de ce type.

Calcul de l'état suivant (Cf. Figure 10.2) Supposons ici que le front d'activation des bascules du circuit séquentiel soit le front montant de l'horloge. Soit t-états le délai nécessaire à la stabilisation des circuits combinatoires de calcul de l'état suivant. Nous avons vu au chapitre 8 que ce délai n'est pas nul. Soit i-entrées l'instant à partir duquel les entrées sont stables.

> **Remarque :** Le temps de stabilisation de la sortie des bascules n'est pas nul. On le néglige ici par rapport aux délais de stabilisation des circuits combinatoires.

Pour que l'automate puisse évoluer à chaque front montant de l'horloge, il faut que le résultat du calcul du nouvel état soit stable avant le prochain front montant de clock. Le chronogramme de la figure 10.2-a montre cette dépendance.

Etudions le cas simple pour lequel on sait échantillonner les entrées sur l'horloge, c'est-à-dire où les entrées changent toujours sur un front de l'horloge. Dans ce cas i-entrées correspond à un des deux fronts de l'horloge. On peut faire deux choix d'échantillonnage.

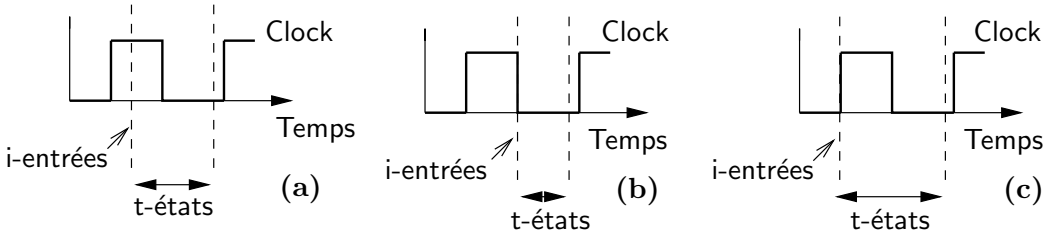


FIG. 10.2 – Chronogrammes d'échantillonnage des entrées d'un automate. a) Temps de calcul ; b) échantillonnage sur front descendant ; c) échantillonnage sur front montant.

Une première idée est d'échantillonner les entrées sur le front descendant de clock. Le circuit arrivant dans un nouvel état au front montant et les entrées sur le front descendant, les sorties des circuits combinatoires calculant l'état suivant ont alors une demi-période de l'horloge pour se stabiliser. Le chronogramme de la figure 10.2-b montre cette dépendance : la demi-période de clock doit être supérieure à t-états pour que les entrées des bascules soient stables au front montant.

Si l'on regarde de plus près, il s'avère que l'on peut anticiper cet échantillonnage et l'effectuer au même front que le changement d'état. Au même instant l'état suivant est mémorisé et les entrées sont fournies. En effet grâce au temps de stabilisation des sorties des circuits combinatoires t-états, l'arrivée des nouvelles valeurs des entrées n'est pas encore répercutée à l'entrée des bascules au moment du chargement du nouvel état. La période de clock doit être supérieure à t-états. Pour une valeur de t-états maximale donnée, on peut ainsi doubler par rapport à la première solution (Figure 10.2-b) la fréquence maximale à laquelle l'automate peut évoluer (si on prend des niveaux haut et bas de l'horloge de même durée). Le chronogramme de la figure 10.2-c montre cette évolution.

Calcul des sorties (Cf. Figure 10.3) Si la solution adoptée est l'échantillonnage des entrées sur le front d'activation des bascules, que l'on soit dans le cas d'un automate de Moore ou de Mealy le résultat est le même.

Soit t-sorties le délai de stabilisation des circuits combinatoires calculant les sorties. Les signaux de sorties seront stables un temps t-sorties après le front d'activation des bascules (Cf. Figure 10.3-a).

En revanche, dans le cas où l'échantillonnage ne se fait pas sur le front d'activation des bascules, les sorties d'un automate de Mealy auraient un comportement différent de celui d'un automate de Moore. Dans le cas de Moore, les sorties ne dépendant que de l'état courant, l'échantillonnage des entrées ne change en rien leur calcul.

Dans le cas du modèle de Mealy, les sorties dépendent des entrées et de l'état courant. Soit i-entrées l'instant à partir duquel les entrées sont stables.

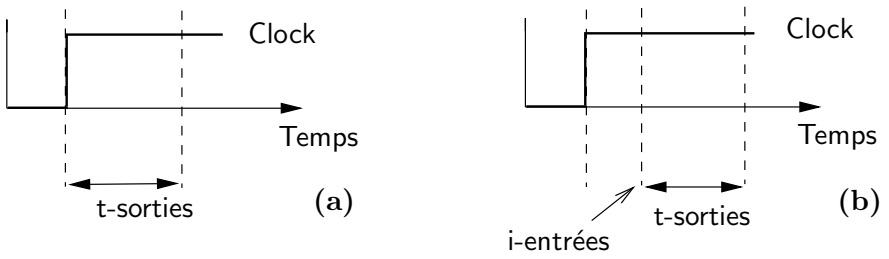


FIG. 10.3 – Chronogrammes des sorties d'un automate. a) Cas de Moore ; b) cas de Mealy.

Les sorties sont stables un temps t-sorties après i-entrées (Cf. Figure 10.3-b). Il apparaît alors des valeurs transitoires sur les sorties pendant des temps non négligeables avant leur stabilisation. Ces valeurs transitoires peuvent provoquer des évolutions non voulues pour un système aval (qui utiliserait les sorties ainsi produites).

1.3.2 Initialisation

Nous avons vu au chapitre 9 qu'il existe deux sortes d'initialisation des bascules. L'initialisation asynchrone est effectuée dès la présence de la valeur d'initialisation sur le signal correspondant. L'initialisation synchrone n'est effectuée qu'au moment du front d'activation de l'horloge.

Si l'initialisation n'est pas effectuée au moment du front montant de l'horloge l'automate peut passer dans un état indéterminé si les circuits combinatoires de calcul de l'état suivant n'ont pas le temps de se stabiliser entre l'instant d'initialisation et le prochain front montant de l'horloge. La solution synchrone est donc utilisée de préférence.

1.3.3 Synchronisation de deux réalisations

On a souvent besoin de réaliser deux automates 1 et 2, les sorties de l'automate 1 étant connectées aux les entrées de l'automate 2. Pour les raisons évoquées précédemment les horloges de ces deux automates ne peuvent pas être indépendantes. Supposons que le front d'activation de l'automate 1 soit le front montant et que ses entrées soient échantillonnées sur le front montant de son horloge clock1 (Cf. Figure 10.4). Supposons que le front d'activation de l'automate 2 soit aussi le front montant. Le temps de calcul de ses sorties n'est pas négligeable : elles sont stables un temps t-sorties1 après le front d'activation de l'horloge clock1. Le front montant de l'horloge de l'automate 2 doit avoir lieu après que ses entrées (les sorties de l'automate 1) sont devenues stables et que le calcul de son état suivant s'est stabilisé (t-états2).

Une première idée consiste à penser que les fronts d'activation des deux horloges doivent être décalés du temps de stabilisation nécessaire.

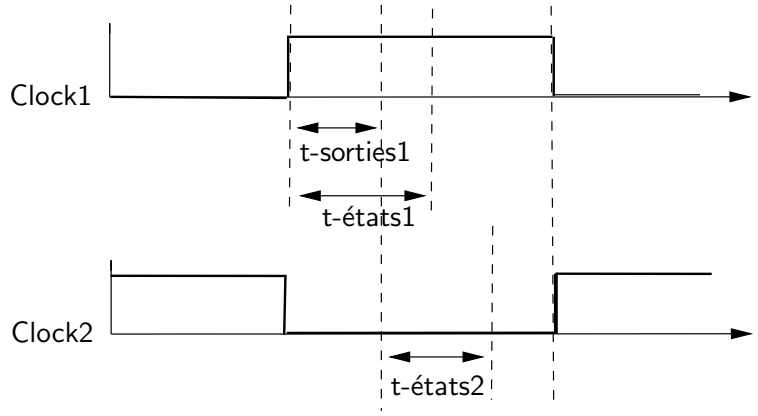


FIG. 10.4 – Chronogrammes de synchronisation de deux automates

Une façon simple pour réaliser ce décalage est de prendre pour clock2 le complément de clock1. Le décalage est alors d'une demi-période. Pour que les entrées des bascules des deux automates soient toujours stables au moment du front d'activation, la période minimale P de l'horloge doit alors vérifier :

- $P/2 >$ t-sorties1 + t-états2 pour que l'état de l'automate 2 soit stable au moment du front montant de clock2.
- $P >$ t-états1 pour que l'état de l'automate 1 soit stable au moment du front montant de clock1.

Dans ce cas le décalage de l'évolution de l'état des deux automates est d'une demi-période de l'horloge.

Il s'avère que l'on peut aussi prendre clock1 égale à clock2. Les entrées des bascules restent stables si l'on respecte les conditions suivantes sur la période P de l'horloge :

- $P >$ t-sorties1 + t-états2 pour que l'état de l'automate 2 soit stable au moment du front montant de clock.
- $P >$ t-états1 pour que l'état de l'automate 1 soit stable au moment du front montant de clock.

Automates en boucle Le cas particulier où les sorties de l'automate 2 sont les entrées de l'automate 1 est intéressant à étudier. Ce cas de figure se présente souvent et apparaît en particulier dans la réalisation d'algorithmes complexes étudiés au chapitre 11.

Dans le cas où les deux automates sont de type Mealy, nous obtenons la configuration de la figure 10.5. Cette architecture peut ne pas arriver dans un état stable puisque les entrées des blocs combinatoires C1 et C2 sont des sorties de ces mêmes blocs.

Il faut donc que l'un des deux automates soit de type Moore. Pour simplifier

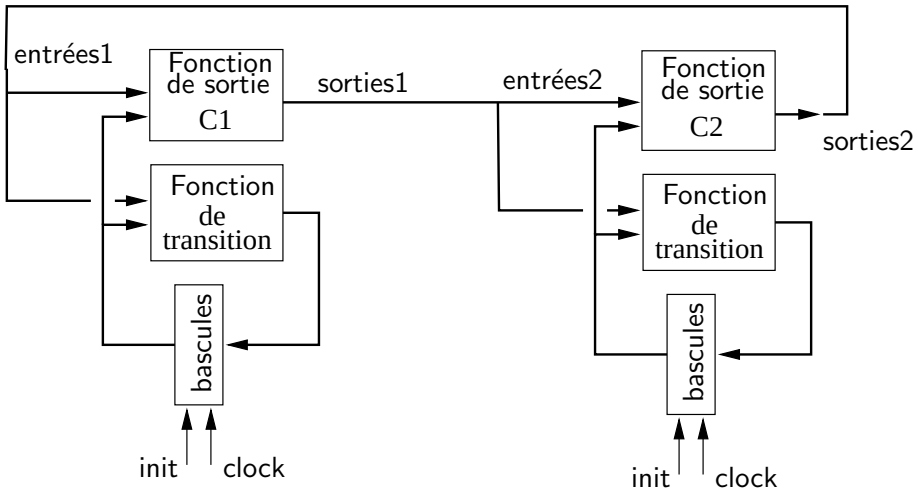


FIG. 10.5 – Cas de deux automates de Mealy en boucle

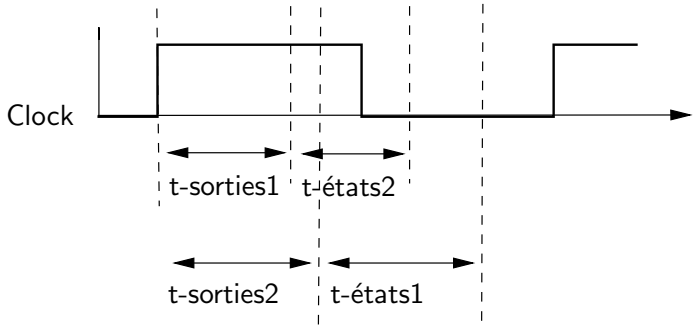


FIG. 10.6 – Chronogramme décrivant la synchronisation de deux automates en boucle

supposons que les deux soient de type Moore. Les deux automates peuvent alors évoluer à l'aide de la même horloge clock (chronogramme de la figure 10.6).

Nous obtenons alors les conditions suivantes sur la période P de l'horloge :

– $P >$ t-sorties1 + t-états2 pour que l'état de l'automate 2 soit stable au moment du front montant de clock.
– $P >$ t-sorties2 + t-états1 pour que l'état de l'automate 1 soit stable au moment du front montant de clock.

2. Synthèse des automates décrits par leur graphe

Le point de départ de cette synthèse est le graphe explicite d'un automate d'états fini. Nous nous limitons ici aux automates de type :

– synchrone (Cf. Chapitre 6) : les instants de changement des entrées sont

connus et synchronisés avec l'évolution de l'automate, le signal d'horloge permettant d'effectuer cette synchronisation. L'utilisation de ce type d'automate est largement répandue. La synthèse en est plus simple et donc d'un intérêt pédagogique important.

– réactif et déterministe (Cf. Chapitre 5) : d'un point de vue matériel, il est obligatoire que le circuit résultant évolue toujours en fonction de ses entrées vers un nouvel état déterminé et unique.

En ce qui concerne le choix du type de l'automate Moore ou Mealy, il n'y a pas de règle précise. On peut toujours décrire un automate de Moore équivalent à un automate de Mealy (Cf. Paragraphe 1.1.4, Chapitre 5). Suivant le système à spécifier une des deux formes peut être plus complexe que l'autre en nombre d'états et de transitions. Cependant pour des raisons de comportement temporel dans le cas de synchronisation d'automates (Cf. Paragraphe 1.3), le type Moore est en général plus utilisé.

Nous détaillons dans ce paragraphe la synthèse de deux types d'architecture. La première est dite *câblée* car la réalisation des fonctions de sorties et de transition est faite à l'aide de circuits combinatoires. On va *câbler les portes logiques* correspondantes. La réalisation de ces fonctions booléennes est effectuée suivant les techniques de synthèse de circuits combinatoires étudiées au chapitre 8.

Nous donnons ensuite les principes de réalisation d'un deuxième type d'architecture dite *microprogrammée*. Dans ce type de synthèse les fonctions de sortie et de transition sont en grande partie réalisées à l'aide d'une mémoire morte (ROM). Ce type de réalisation s'inspire des principes de programmation en langage d'assemblage (Cf. Chapitre 12).

2.1 Réalisation câblée

Les différentes étapes de ce type de synthèse sont expliquées et illustrées sur un exemple simple.

2.1.1 Un exemple : une commande de feu tricolore

On veut réaliser une commande d'un feu tricolore à plaque. Les informations d'entrées sont : voiture-présente (vp), voiture-absente (va). La sortie est la couleur du feu : Vert (V), Orange(O) ou Rouge (R).

Le comportement du système est le suivant (automate de Moore de la figure 10.7). Au départ le feu est rouge. Si le feu est rouge : si une voiture est présente, le feu passe au vert sinon le feu reste rouge. Si le feu est orange le feu passe au rouge. Si le feu est vert : si une voiture est présente, le feu reste au vert ; si une voiture est absente deux fois de suite, le feu passe au orange.

> **Remarque :** Le terme *deux fois de suite* implique que le temps est découpé en intervalles réguliers. Nous retrouvons ici les entrées d'un automate syn-

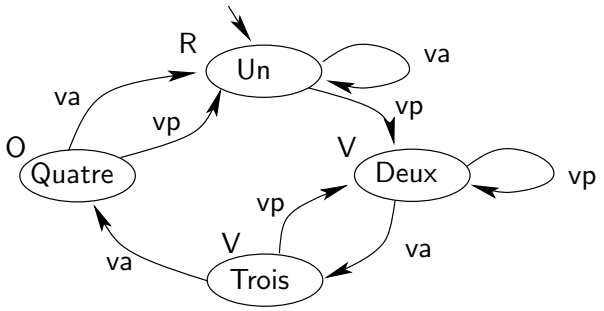


FIG. 10.7 – Automate décrivant un système de commande de feu tricolore

(a)

Vocabulaire	e
va	0
vp	1

(b)

Vocabulaire	s_1	s_2
R	0	0
O	0	1
V	1	0

(c)

Etat	q_1	q_2
Un	0	0
Deux	0	1
Trois	1	0
Quatre	1	1

FIG. 10.8 – Codage des entrées, des sorties et des états du système de commande de feu tricolore.

chrone échantillonnées sur une horloge qui découpe le temps de façon régulière. *deux fois de suite* signifie alors pendant deux périodes d'horloge successives.

2.1.2 Choix de l'élément de mémorisation

Pour pouvoir mémoriser l'état courant de l'automate nous avons à notre disposition les éléments de mémorisation élémentaires présentés au chapitre 9. Nous ne pouvons pas utiliser de verrous puisque la valeur de l'état suivant dépend de l'état courant. Nous utilisons donc des bascules de type D sensibles au front. Elles comportent un signal d'activation qui force le changement d'état et un signal d'initialisation (soit à 1, soit à 0 suivant le codage de l'état initial). Ce signal d'initialisation est de type synchrone.

2.1.3 Codage des entrées et des sorties

Les éléments des vocabulaires d'entrée et de sortie sont codés en binaire. Les codes inutilisés dans ce codage correspondent à des cas de valeur phi-booléenne dans les fonctions de transition et de sortie.

Reprenons l'exemple. Le vocabulaire d'entrée {va, vp} est codé à l'aide d'une variable booléenne e (Figure 10.8-a). Le vocabulaire de sortie {R, O, V} est codé à l'aide de deux variables booléennes s_1 et s_2 (Figure 10.8-b). Le code $s_1 = s_2 = 1$ ne correspond à aucune sortie.

2.1.4 Codage des états

L'état étant mémorisé dans des bascules, une bascule stockant une information binaire, les différentes valeurs de l'état doivent être codées en binaire. Les variables booléennes correspondant à ce codage sont appelées *variables d'états*. Différents types de codage peuvent être utilisés (Cf. Chapitre 3). Soit n le nombre d'états, le nombre minimum de bits permettant de réaliser le codage est $log_2(n)$. Un tel codage est appelé *compact*. Il permet une réalisation avec un nombre minimum de points de mémorisation. Un autre type de codage souvent utilisé est le codage *un parmi n*. Ce type de codage entraîne un nombre maximum de points de mémorisation mais l'élaboration du circuit réalisant l'automate peut se faire d'une manière spécifique que nous précisons dans le paragraphe 2.1.6.

L'automate de la figure 10.7 comporte 4 états. La figure 10.8-c donne un exemple de codage compact des états de cet automate à l'aide de deux variables d'états q_1, q_2.

Le choix du code $q_1 = 0$, $q_2 = 0$ pour l'état initial implique que l'initialisation des bascules à l'aide du signal init est une initialisation à zéro.

On peut choisir un autre code pour l'état initial. Dans ce cas les bascules doivent comporter les deux types d'initialisation (à 0 et à 1).

Il est à noter que les fonctions booléennes de sortie et de transition dépendent de ces codages et que le choix du codage influence fortement les caractéristiques du circuit (par exemple sa complexité en nombre de portes) et donc ses performances. Les outils de CAO réalisant cette synthèse prennent en compte ces critères lors du choix de ces codes.

2.1.5 Expression algébrique des fonctions de transition et de sortie

L'architecture générale du circuit (pour un automate de Moore) est donnée dans la figure 10.9. Soient $(d_1, d_2, \ldots d_n)$ les variables codant l'état suivant, $(q_1, q_2, \ldots q_n)$ les variables codant l'état à l'instant courant, $(s_1, s_2, \ldots s_m)$ les variables codant les sorties et $(e_1, e_2, \ldots e_t)$ les variables codant les entrées.

- Le bloc mémorisant l'état courant de l'automate est composé de n bascules D sensibles au front de l'horloge clock. Chacune a sur son entrée D un signal d_i et sa sortie Q donne un signal q_i.
- Le circuit combinatoire calculant la fonction de sortie réalise les fonctions booléennes définissant la valeur des s_i. Dans le cas d'un automate de Moore, s_i est une fonction booléenne des variables de l'état courant : $s_i = f_i(q_1, q_2, \ldots q_n)$. Dans le cas d'un automate de Mealy nous avons : $s_i = f_i(q_1, q_2, \ldots q_n, e_1, e_2, \ldots e_t)$.
- Le circuit combinatoire calculant l'état suivant réalise les fonctions booléennes : $d_i = g_i(q_1, q_2, \ldots q_n, e_1, e_2, \ldots e_t)$.

La fonction de transition de l'automate de la figure 10.7 peut être décrite sous forme de table. En utilisant les codes choisis précédemment, cette table

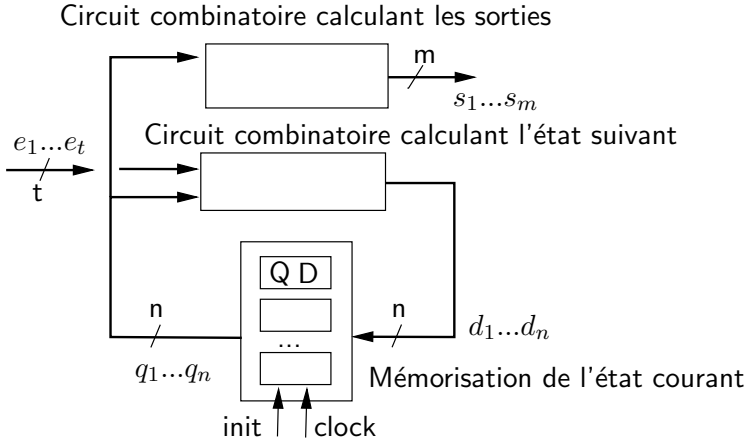


FIG. 10.9 – La structure d'un circuit réalisant un automate de façon câblée

donne les valeurs des variables codant l'état suivant (d_1, d_2) en fonction des variables codant l'état courant (q_1, q_2) et de l'entrée (e). Ce tableau correspond à la table de vérité des deux fonctions booléennes d_1, d_2. La figure 10.10 donne ces tables ainsi que celles correspondant aux sorties. La figure 10.11 décrit le circuit résultant réalisé à base de portes NAND et d'inverseurs.

2.1.6 Cas particulier du codage des états *un parmi n*

Ce codage consiste à coder n états sur n bits en représentant chaque état par un bit ; le code d'un état comporte alors un seul bit (parmi n) à 1. On peut procéder de la même façon que précédemment. Les fonctions de l'automate comportent alors beaucoup de phi-booléens puisqu'il y a un nombre important de codes inutilisés pour les états.

Toutefois on peut obtenir plus simplement une solution équivalente (et simplifiée) du circuit en se calquant directement sur le graphe de l'automate. La figure 10.13 montre la réalisation du circuit de commande de feu tricolore. Par exemple, la bascule 1, codant l'état 1, est chargée avec la valeur 1 si l'état courant est 1 et l'entrée $\bar{e}$ ou si l'état courant est 4. De façon plus générale, pour réaliser le circuit on applique les règles suivantes :

entrée	état courant	état suivant
va	Un	Un
vp	Un	Deux
va	Deux	Trois
vp	Deux	Deux
va	Trois	Quatre
vp	Trois	Deux
va	Quatre	Un
vp	Quatre	Un

e	q_1	q_2	d_1	d_2
0	0	0	0	0
1	0	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	0	1	1
1	1	0	0	1
0	1	1	0	0
1	1	1	0	0

état	sortie
Un	R
Deux	V
Trois	V
Quatre	O

q_1	q_2	s_1	s_2
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

FIG. 10.10 – Tables de définition des fonctions de transition et de sortie de l'automate de commande de feu tricolore

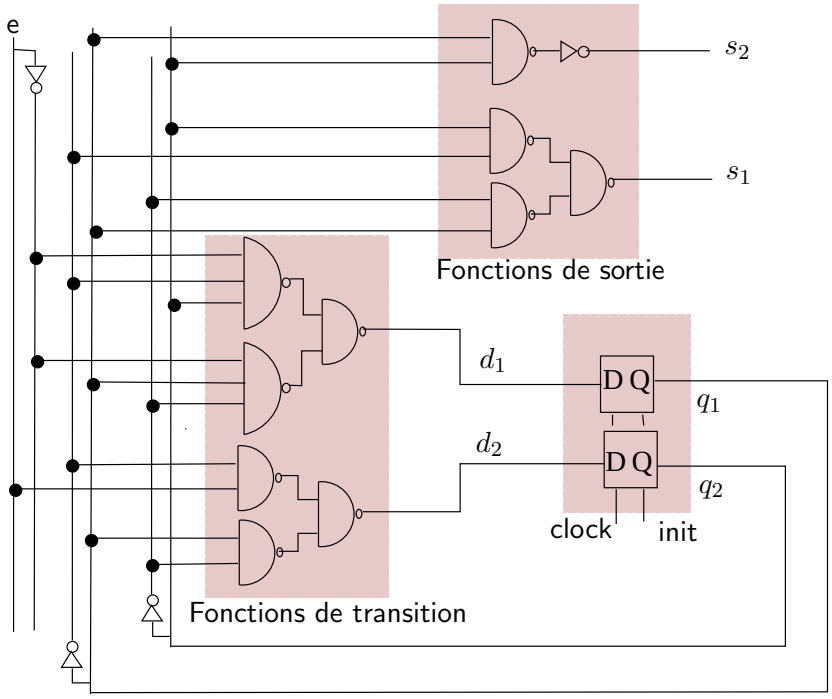


FIG. 10.11 – Synthèse de l'automate de la figure 10.7 à base de portes NAND et de bascules D. Les expressions booléennes correspondantes sont : $d_1 = \overline{q_1}.q_2.\overline{e} + q_1.\overline{q_2}.\overline{e}$ et $d_2 = \overline{q_1}.e + q_1.\overline{q_2}$; $s_1 = \overline{q_1}.q_2 + q_1.\overline{q_2}$ et $s_2 = q_1.q_2$.

Etat	q_1	q_2	q_3	q_4
Un	1	0	0	0
Deux	0	1	0	0
Trois	0	0	1	0
Quatre	0	0	0	1

FIG. 10.12 – Codage 1 parmi n des états du système de commande de feu tricolore

- On associe à chaque état un point de mémorisation (bascule D à front).
- Un arc sortant de l'état X portant l'entrée E est réalisé par une porte ET dont les entrées sont la sortie de la bascule correspondant à l'état X et l'entrée E. Si un arc sort de l'état X pour toute entrée du vocabulaire, cette porte E est inutile (comme l'arc de l'état 4 à l'état 1 dans l'exemple).
- Pour un ensemble d'arcs entrants dans un état Y les différentes sorties des portes ET correspondantes sont mises en entrée d'une porte OU dont la sortie est reliée à l'entrée de la bascule correspondant à l'état Y. Dans le cas où il n'y a qu'un seul arc entrant dans un état, cette porte OU est inutile (comme à l'entrée de l'état 3 ou 4).
- On réalise chaque sortie par une porte OU qui a comme entrées les sorties des bascules correspondant aux états où la sortie vaut 1.
- L'initialisation s'effectue en initialisant à 1 la bascule correspondant à l'état initial et à 0 les autres bascules.
- Au cas par cas une combinaison de portes ET-OU peut être remplacée par une combinaison NAND-NAND.

Pour l'exemple du feu tricolore, nous choisissons de coder l'état à l'aide des 4 variables q_1, q_2, q_3, q_4 (Figure 10.12). Le codage des entrées et des sorties reste inchangé. La figure 10.13 montre l'automate et le circuit résultant. Sur la figure la commande d'initalisation n'est pas représentée : la commande init initalise la bascule de numéro 1 à 1 et les autres à 0.

2.2 Réalisation microprogrammée

Dans ce type de synthèse les fonctions de transition et de sortie sont réalisées à l'aide d'une mémoire de type ROM. Chaque adresse de cette mémoire est le code d'un état de l'automate.

On parle de *microprogrammation* car on peut utiliser un langage de description textuel du contenu de la ROM comme on le ferait avec le langage d'assemblage pour du langage machine. Chaque ligne de la mémoire correspond à une *micro-instruction* du *microprogramme*.

Ce type de synthèse a beaucoup été utilisé pour la réalisation de gros automates demandant une mise au point importante. Elle permet en effet de modifier l'automate par simple reprogrammation d'une EPROM (Cf. Chapitre 9).

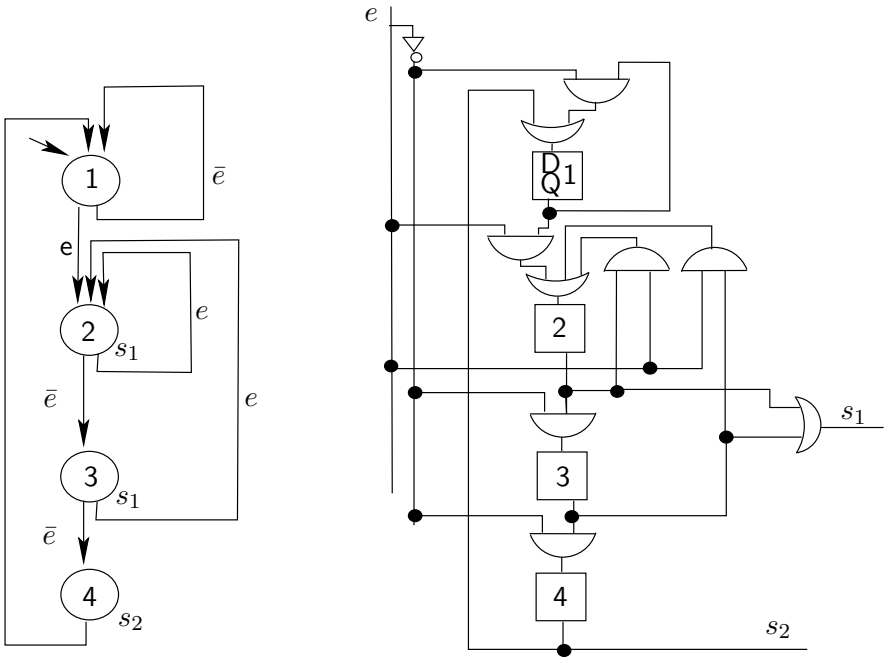


FIG. 10.13 – Un circuit réalisant un automate avec un codage un parmi n des états

Elle a été en particulier employée pour la conception de la partie contrôle de certains processeurs CISC comme le 68000. En effet, la partie contrôle d'un tel processeur est un automate de plusieurs centaines d'états et de sorties. Elle est aussi utilisée dans des circuits de type automate programmable.

2.2.1 Architecture générale d'une réalisation microprogrammée

Cette technique s'applique à la synthèse d'automates de type Moore. L'architecture générale du circuit est donnée dans la figure 10.14.

Contraintes sur l'automate Pour des raisons liées à l'architecture utilisée, des transformations préliminaires de l'automate peuvent être nécessaires : chaque état doit posséder au plus deux successeurs. On a vu au paragraphe 2.4 du chapitre 5 comment effectuer une telle transformation. Il est à remarquer que l'ajout d'états supplémentaires, s'il ne change pas le comportement de l'automate d'un point de vue fonctionnel, change son comportement temporel.

Les codes des états sont choisis de façon à ce que :

– quand un état x ne possède qu'un état successeur, le code de celui-ci est soit le code de x plus 1, soit un autre code choisi afin de limiter le nombre de codes.

– quand un état x possède 2 successeurs, le code de l'un des 2 est le code de x plus 1. L'autre est choisi de manière à minimiser le nombre de codes. Il se

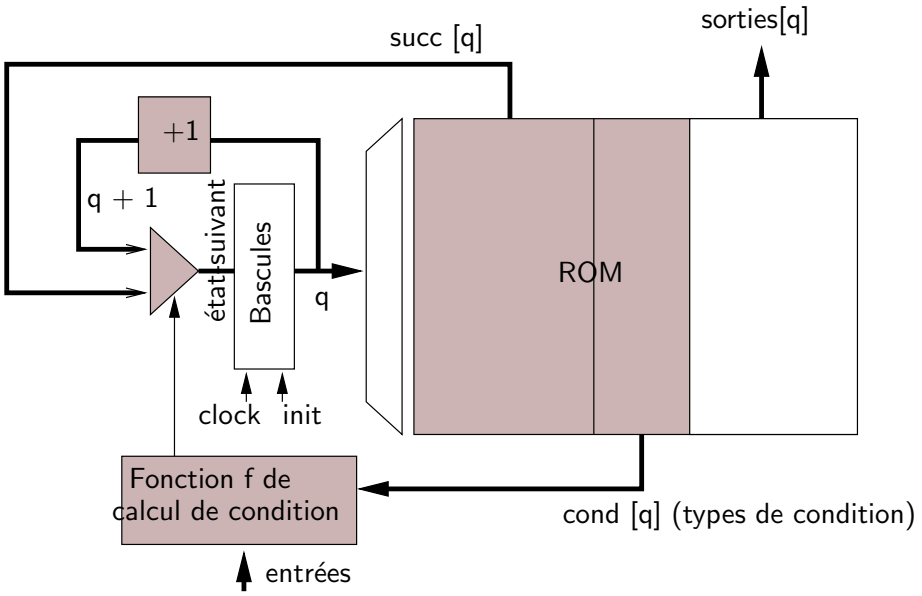


FIG. 10.14 – Architecture d'une réalisation microprogrammée d'un automate

peut que cela ne soit pas possible, on ajoute alors un état supplémentaire afin d'obtenir cette configuration.

Calcul de l'état suivant Le code de l'état courant est mémorisé dans des bascules et fournit une adresse de la mémoire.

Les primitives de calcul de l'état suivant sont de deux types : une incrémentation (circuit +1 sur la figure 10.14) ; la donnée dans la mémoire du code de l'état suivant, l'accès à ce code se faisant à l'aide du code de l'état courant (adresse d'une ligne de la mémoire).

Le fait que chaque état ait au maximum deux états successeurs permet de limiter la largeur de la mémoire. Quand le code de l'état suivant n'est pas le code de l'état courant plus 1, son code se trouve dans la mémoire.

Par ailleurs, il faut pouvoir spécifier quelle est la condition permettant de choisir l'état successeur (dépendant des entrées de l'automate) quand il y en a deux. Cette condition est aussi spécifiée dans la mémoire et correspond à certaines sorties de celle-ci. On parle de champ *condition*.

Soit q le code d'un état, soit succ[q] le code de l'état successeur (qui n'est pas q + 1) contenu dans la mémoire, soit cond[q] la condition à tester pour choisir l'état successeur suivant les entrées de l'automate. L'architecture microprogrammée comporte les circuits permettant de définir le code de l'état suivant comme suit :

état-suivant(q)= si f (cond[q], entrées) alors succ[q] sinon q+1.

Les codes sont choisis de façon à ce que :

– quand un état de code q ne possède qu'un état successeur, cond [q] spécifie soit la condition toujours vraie et état-suivant(q) = succ[q], soit toujours

fausse et état-suivant(q) = q + 1, quelles que soient les entrées de l'automate.
– quand un état q possède 2 successeurs, pour traiter les conditions d'entrées se trouvant sur les transitions de l'automate, un circuit combinatoire délivre la valeur de la fonction booléenne f calculant la condition à tester spécifiée dans la mémoire cond[q] à partir des entrées de l'automate. La sélection selon cette condition entre la sortie de l'incrémenteur et de la mémoire pourra se faire naturellement à partir d'un multiplexeur 2 voies vers 1.

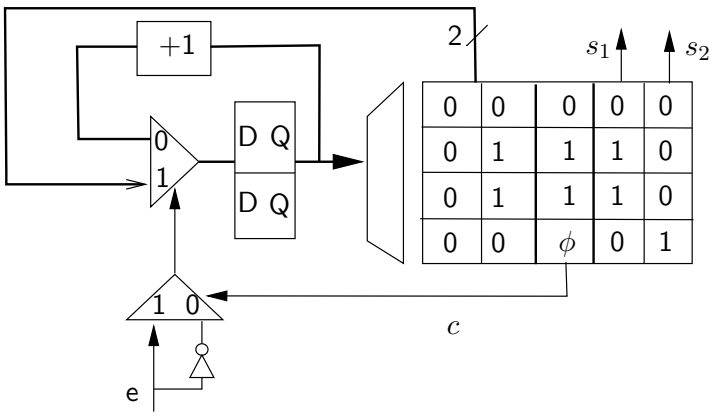
Le calcul de l'état suivant est donc réalisé à l'aide d'une partie de la mémoire, d'un incrémenteur et d'autres circuits combinatoires pour le calcul de la condition (en gris sur la figure 10.14).

Calcul des sorties Après avoir procédé comme précédemment au codage des sorties, pour chaque état, la valeur de chaque sortie est mémorisée une fois pour toute dans la ligne de la mémoire correspondant à l'état (noté sorties[q] sur la figure 10.14).

Optimisations On a limité ici le nombre d'états successeurs à 2 pour ne pas augmenter la largeur de la mémoire. Des techniques ont été développées pour définir plusieurs états dans la mémoire sans trop en augmenter la largeur. Elles consistent par exemple à ne donner dans la mémoire que les bits qui changent pour les adresses successives à partir d'une adresse de base. Ainsi par exemple 2 bits supplémentaires peuvent suffire pour résoudre les cas où un état possède 4 successeurs. D'autres techniques sont aussi souvent employées pour minimiser la largeur de la mémoire dépendant des sorties. Par exemple, on peut utiliser le champ contenant l'état successeur pour certaines sorties quand celui-ci n'est pas nécessaire.

On peut aussi remarquer que dans le cas très particulier de l'exemple de la commande de feu tricolore, repris ci-dessous, la colonne la plus à gauche pourrait être supprimée puisqu'elle ne comporte que des 0.

Microprogrammation Pour spécifier le contenu de la mémoire et permettre une mise au point aisée, les concepteurs définissent un langage particulier dont la syntaxe ressemble à celle d'un langage d'assemblage (Cf. Chapitre 12). Chaque ligne de la ROM est une micro-instruction du microprogramme. On retrouve ainsi des micro-instructions de branchement conditionnel pour les cas où un état possède deux successeurs. Les micro-instructions permettent aussi de spécifier des valeurs spécifiques pour les sorties. Elles peuvent alors faire référence, au niveau de la syntaxe, à des entités extérieures à l'automate, comme par exemple des registres ou une UAL dans le cas d'une partie contrôle d'un processeur (Cf. Chapitres 11 et 14).



0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	0	ϕ	0	1

FIG. 10.15 – Architecture d'une réalisation microprogrammée de l'exemple du feu tricolore

```
et1 :  bne   et1
et2 :  be    et2,  s1
       be    et2,  s1
       ba    et1,  s2
```

état	code	succ	état + 1	cond	code
1	00	00	01	$\overline{\text{e}}$	0
2	01	01	10	e	1
3	10	01	11	e	1
4	11	00	00	-	-

FIG. 10.16 – Microprogramme de l'exemple du feu tricolore

2.2.2 Exemple de la commande de feu tricolore

Chaque état possède au plus deux états successeurs. Le codage compact choisi précédemment vérifie la condition sur le code des états successeurs (q+1 ou succ[q]). Les différentes conditions à tester sont : $\overline{e}$ dans l'état Un ; e dans les états Deux et Trois ; *VRAI* dans l'état Quatre.

En utilisant un incrémenteur modulo 4 cette dernière condition est inutile. Un seul bit c suffit donc pour spécifier dans la ROM quelle est la condition à tester. Nous choisissons $c = 1$ pour la condition e et $c = 0$ pour la condition $\overline{e}$.

Pour les sorties, on choisit le même codage que précédemment. La ROM possède 4 lignes. Deux bits de la ROM sont donc utilisés pour les sorties, un pour le calcul de la condition, deux pour le codage de l'état suivant. La fonction f permettant de calculer la condition déterminant le choix de l'état suivant est réalisée à l'aide d'un multiplexeur 2 voies vers 1 (Cf. Figure 10.15).

L'initialisation se fait à l'aide du signal init qui initialise les bascules à 0 puisque le code de l'état initial est 00.

Le contenu de la mémoire dans le cas du feu tricolore pourrait être décrit par le microprogramme suivant de la figure 10.16.

`ba et1` est une micro-instruction de branchement inconditionnel à une étiquette `et1`. `bne` et `be` sont les micro-instructions de branchement sur les conditions $\overline{e}$ et e. Seules les sorties à 1 dans un état sont spécifiées dans la micro-instruction correspondante.

2.3 Un exemple détaillé : la machine à café

Nous reprenons l'exemple de la machine à café présenté dans le chapitre 5. La figure 10.17 rappelle son graphe de Moore. Nous étudions le codage des entrées/sorties du contrôleur et la synthèse d'un circuit séquentiel d'après la machine séquentielle qui décrit son comportement.

Exemple E10.1 : Machine à café (*suite de E5.2, p 105 et E6.3, p 133*) *Nous supposons ici que les entrées s_1, s_2 et s_5 venant des capteurs et que l'entrée f_s venant de la machine à café sont synchronisées sur le front d'une horloge* clock. *Nous allons étudier comment élaborer une réalisation câblée de cet automate.*

Le vocabulaire d'entrée de l'automate est $\{rien, s_1, s_2, s_5, f_s\}$. *rien* signifie : toutes les autres entrées sont fausses. Les combinaisons sur les 4 entrées s_1, s_2 , s_5 et f_s n'étant pas toutes possibles, 3 bits suffisent pour les coder. Les codages choisis sont donnés dans la figure 10.18. Les 3 codes sur e_1, e_2, e_3 non utilisés correspondent à des cas qui ne peuvent survenir.

D'autre part, vues les spécifications de l'automate, certaines de ces combinaisons ne peuvent pas survenir dans certains états. Ainsi on ne peut avoir ni s_1, ni s_2, ni s_5 dans les états 2F reçu et Trop perçu. On ne peut avoir f_s dans les autres états. La fonction de transition est phi-booléenne.

Nous procédons de même pour les sorties. Le vocabulaire de sorties de l'automate étant { CB, RCB, AUCUNE}, trois cas sont possibles. Les sorties sont codées sur 2 bits (Cf. Figure 10.18).

Il y a 4 états que nous codons sur deux bits q_1 et q_2 (Cf. Figure 10.18).

L'état est mémorisé dans des bascules D sensibles au front montant de l'horloge clock. La figure 10.19 donne la table de vérité des deux fonctions de transition d_1 et d_2. La dernière ligne résume tous les autres cas pour lesquels les deux fonctions sont à ϕ.

Si l'on effectue une minimisation de la forme polynômiale (Cf. Chapitre 2), on obtient les expressions : $d_1 = \overline{e_2}.q_1.q_2 + \overline{e_1}.\overline{e_2}.q_1 + e_3$ et $d_2 = \overline{e_1}.q_2 + e_1.\overline{e_2}.\overline{q_2} + e_3$. Pour les sorties on trouve : $sortie_1 = \overline{q_1}.\overline{q_2}$ et $sortie_2 = \overline{q_1}$.

De ces équations on peut aisément déduire la réalisation de cet automate en utilisant 2 bascules D et des portes ou un PLA (Cf. Chapitre 8).

3. Synthèse des circuits séquentiels par flots de données

Dans le cas où la spécification du système à réaliser est donnée sous forme d'algorithme manipulant des variables, la modélisation sous forme de graphe d'états fini devient rapidement impossible. En effet le nombre d'états peut devenir très grand. Il est proportionnel au nombre de valeurs possibles des variables de l'algorithme.

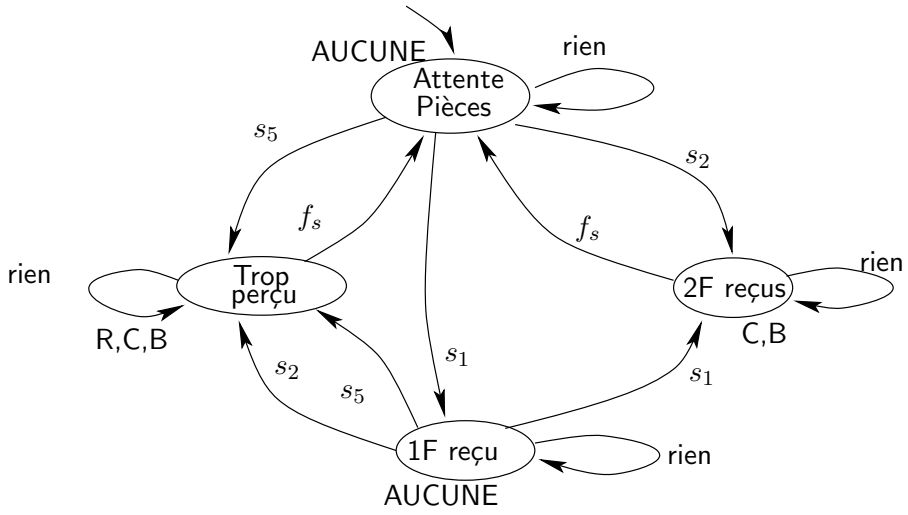


FIG. 10.17 – Graphe de Moore de l'automate de la machine à café

s_1	s_2	s_5	f_s	e_1	e_2	e_3
0	0	0	0	0	0	0
1	0	0	0	1	0	0
0	1	0	0	0	1	0
0	0	1	0	1	1	0
0	0	0	1	0	0	1

Vocabulaire de sorties	$sortie_1$	$sortie_2$
AUCUNE	0	0
R,C,B	1	1
C,B	0	1

$Etats$	q_1	q_2
Attentes Pièces	1	1
1F reçu	1	0
2F reçu	0	1
Trop perçu	0	0

FIG. 10.18 – Codage des entrées, des sorties et des états pour la synthèse de l'automate de contrôle de la machine à café

e_1	e_2	e_3	q_1	q_2	d_1	d_2
0	0	0	0	0	0	0
0	0	1	0	0	1	1
0	0	0	0	1	0	1
0	0	1	0	1	1	1
0	0	0	1	0	1	0
1	0	0	1	0	0	1
1	1	0	1	0	0	0

e_1	e_2	e_3	q_1	q_2	d_1	d_2
0	1	0	1	0	0	0
0	0	0	1	1	1	1
1	1	0	1	1	0	0
1	0	0	1	1	1	0
0	1	0	1	1	0	1
-	-	-	-	-	ϕ	ϕ

FIG. 10.19 – Tables de vérité des fonctions de transition de l'automate de contrôle de la machine à café

Par exemple l'algorithme : u $\longleftarrow$ 0 ; tant que vrai : u= (u+1) mod 2^n avec u représenté sur n bits peut être modélisé par un automate d'états fini dont le graphe comporterait 2^n états.

Une méthode systématique permet de réaliser le circuit sans passer par la définition de son graphe d'états. L'état du circuit séquentiel est défini par la valeur des variables apparaissant dans l'algorithme. Un registre est alors utilisé pour chacune de ces variables. Chaque calcul apparaissant dans l'algorithme est réalisé par les composants combinatoires nécessaires. L'enchaînement des calculs se fait au gré du *flot des données* à travers ces composants. Ainsi l'exemple ci-dessus se réalise évidemment avec un incrémenteur n bits et un registre n bits.

Dans la suite de ce paragraphe, nous illustrons cette méthode sur deux exemples. Puis nous donnons une idée de la notion de pipeline.

3.1 Circuit flot de données à une seule variable : la suite de Syracuse

Nous nous intéressons ici à l'étude d'un circuit qui délivre successivement les entiers composant une suite particulière (dite de Syracuse) définie par : si U_N est pair alors $U_{N+1} = U_N$ DIV 2 sinon $U_{N+1} = 3 * U_N + 1$, où DIV dénote la division entière. On peut décrire le calcul de cette suite par l'algorithme :

```
Lexique
   U0 : l'entier > 0 donné ; U : un entier > 0
Algorithme
   U ⟵ U0
   tantque vrai :
         si U MODULO 2 = 0
         alors U ⟵ U DIV 2
         sinon U ⟵ 3 * U + 1
```

Cette suite a la particularité de converger vers les trois valeurs 4, 2, 1 pour certaines valeurs de $U0$. Prenons comme hypothèse que U est borné quelle que soit la valeur de $U0$. Nous pouvons alors décrire cet algorithme à l'aide d'un automate d'états fini, mais cela serait fastidieux, l'automate comportant autant d'états que de valeurs possibles de U.

La figure 10.20 donne l'architecture d'un circuit flot de données réalisant cet algorithme.

Comme dans la synthèse câblée, l'état est mémorisé dans un ensemble de bascules qui contiennent la valeur de U. La boucle principale est réalisée autour de ce registre puisque U change de valeur à chaque itération. Une itération est effectuée pendant une période de l'horloge clock activant les bascules du registre. Nous sommes obligés de fixer une borne supérieure à U, en fixant le nombre de bascules de ce registre à n. Un circuit combinatoire calcule la valeur suivante U' de U.

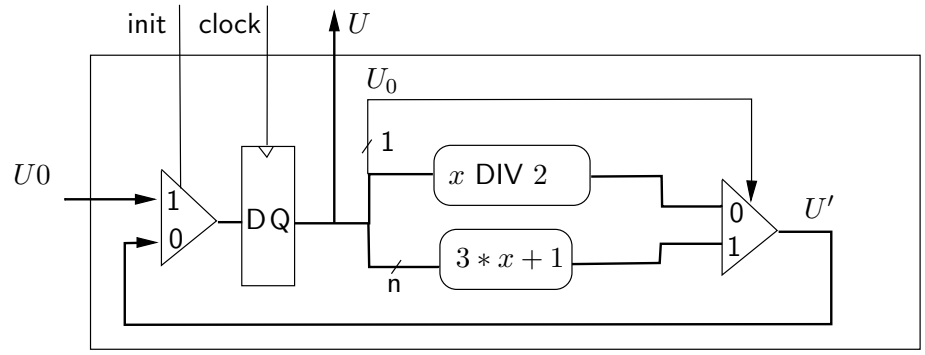


FIG. 10.20 – Calcul de la suite de Syracuse par un circuit à flot de données

L'action si ...alors ...sinon ... est réalisée en effectuant en parallèle les calculs : U DIV 2 et 3 * U + 1, puis en sélectionnant un des deux résultats à l'aide d'un multiplexeur. L'évaluation de l'expression de la condition ne demande pas de calcul puisqu'elle correspond au bit de poids faible de U (U_0).

Le composant calculant x DIV 2 est réalisé par un décalage vers les poids faibles avec introduction d'un 0. Le composant calculant 3 * x + 1 peut se réaliser à l'aide d'un additionneur en observant que 3 * x + 1 = 2 * x + x + 1, 2 * x se réalisant à l'aide d'un décalage vers les poids forts avec introduction d'un 0. L'opération d'incrémentation + 1 peut s'effectuer en forçant la retenue entrante de l'additionneur à 1.

L'initialisation de U à $U0$ peut se faire à l'aide d'un multiplexeur sélectionnant l'entrée du registre mémorisant U, la valeur de $U0$ ou de U'. Si l'entrée init est à 1 le circuit initialise U par $U0$, sinon il donne à U sa prochaine valeur U'.

La figure 10.21 donne l'architecture de ce circuit sous forme de tranches de 0 à n-1. Toutes les tranches sont identiques, sauf les tranches 0 et n-1. Chaque tranche implémente l'algorithme sur 1 bit et contient une bascule mémorisant le $i^{\text{ème}}$ bit de U. On retrouve les multiplexeurs de l'initialisation (en haut) et de l'action si alors sinon en bas. La division et multiplication par 2 se retrouvent dans le décalage des indices sur l'entrée de l'additionneur et du multiplexeur du bas.

3.2 Circuit flot de données à plusieurs variables : la racine carrée

L'algorithme de la figure 10.22 calcule la partie entière de la racine carrée de l'entier naturel x.

La première boucle de l'algorithme qui consiste à calculer la valeur de la plus petite puissance de 4 supérieure à x peut se faire à l'aide d'un circuit combinatoire. Le lecteur pourra s'intéresser au problème à titre d'exercice sur les circuits combinatoires.

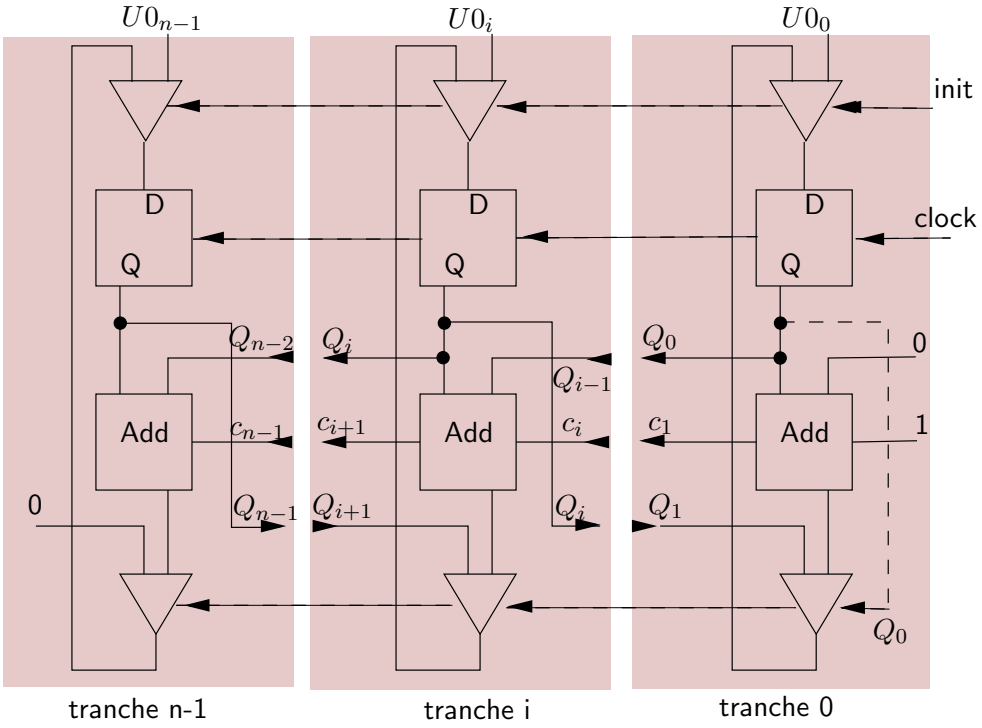


FIG. 10.21 – Architecture en tranche pour le calcul de la suite de Syracuse par un circuit à flot de données

```
Lexique
    x : l'entier ≥ 0 donné
    u, z, y, résultat : des entiers ≥ 0
Algorithme
    u ⟵ 1
    tantque u ≤ x : u ⟵ 4* u
    z ⟵ u ; y ⟵ x
    tantque u > 1 :
            u ⟵ u DIV 4
            z ⟵ z DIV 2 - u
            si z ≤ y :
                y ⟵ y - z
                z ⟵ z + 2 * u
    résultat ⟵ z DIV 2
```

FIG. 10.22 – Algorithme de calcul de la racine carrée [BB83]

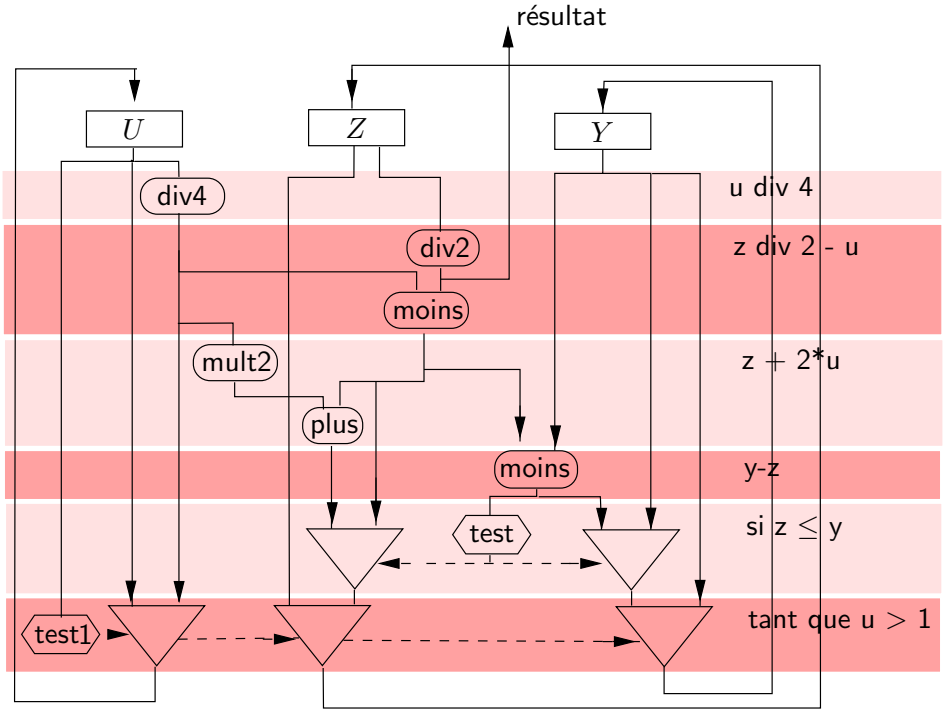


FIG. 10.23 – Calcul de la racine carrée par un circuit à flot de données

Nous nous intéressons ici à la deuxième boucle de l'algorithme. Le circuit d'ensemble est donné figure 10.23.

Les variables internes u, z et y modifiées dans cette boucle constituent l'état de l'automate. Elles sont mémorisées dans trois registres U, Z, et Y. La boucle correspond à la fonction de transition de l'automate. Un passage dans la boucle correspond à une transition de l'automate, donc à une période de l'horloge activant les registres. Par exemple la fonction de transition, restreinte à la partie U de l'état, est $NouvU =$ si $(U - 1 = 0)$ alors U sinon U DIV 4, où $NouvU$ dénote le nouvel état.

On ne s'intéresse pas ici au dialogue du circuit avec un hypothétique monde extérieur. Le résultat est présent tout au long du déroulement de l'algorithme mais il n'est valide que lorsque $U \leq 1$. La progression s'arrête quand U arrive à la valeur 1. Comme un automate ne s'arrête pas, on reproduit ce comportement en ne faisant plus changer les valeurs de U.

La boucle tant que $U > 1$ est réalisée à l'aide des 3 multiplexeurs du bas de la figure 10.23, chaque multiplexeur décidant du changement des variables U,Z et Y. Le prédicat $(U > 1)$ a besoin d'être calculé. Pour cela on peut faire une soustraction de 1 à U. On peut aussi, plus simplement, fabriquer le booléen $U > 1$ à l'aide d'une simple porte OU sur les tous les bits de U sauf le premier (composant test1 sur la figure 10.23).

Chaque calcul apparaissant dans la boucle est réalisé à l'aide d'un ou plusieurs composants apparaissant dans un étage de la figure 10.23. Le calcul de

Z DIV $2 - U$ nécessite un soustracteur (composant moins sur la figure). Le calcul $Z + 2 * U$ nécessite un additionneur (composant plus sur la figure). Le calcul $Y - Z$ nécessite un soustracteur (composant moins sur la figure). On néglige évidemment les multiplications ou divisions par une puissance de 2 (composants div2, mult2 et div4 sur la figure). Ce sont de simples décalages.

Le calcul du prédicat $Z \leq Y$ nécessite le calcul de $Z-Y$ ou $Y-Z$, on profite du calcul de $Y - Z$ pour l'obtenir à travers le composant test. La réalisation de l'instruction si $Z \leq Y$ est effectuée à l'aide de deux multiplexeurs commandés par la sortie du composant test et décidant du changement des valeurs de Z et Y par les résultats des calculs $Z + 2 * U$ et $Y - Z$.

3.3 Notion de pipeline

Le principe de *pipeline* est une optimisation particulière du principe de flot de donnée. Son utilisation est largement répandue dans la conception des microprocesseurs actuels. Le lecteur peut trouver une description plus détaillée dans [HP94].

> **Remarque :** Le terme de pipeline est un mot anglais. Il se prononce avec les diphtongues (a^I) /pa^{I}pla^{I}n/. Une traduction française, oléoduc, ligne de pipe, n'ayant qu'un rapport lointain avec ce dont nous parlons, nous gardons le terme "pipeline". On peut le prononcer à la française... On peut aussi risquer *octéoduc*.

L'organisation sous forme de pipeline d'un système séquentiel digital s'applique à des classes de fonctions particulières : il s'agit de systèmes qui délivrent un flot de sorties en fonction d'un flot d'entrées, en respectant un cadencement : la N$^{\text{ème}}$ sortie est l'image de la N$^{\text{ème}}$ entrée. On peut voir une analogie avec la file d'attente de certaines cafétérias ou cantines : les clients arrivent dans un certain ordre, prennent ou non entrée, dessert, boisson ou plat chaud dans l'ordre où ces plats sont présentés puis payent ; mais l'usage est que l'on ne se double pas. Par opposition on trouve des organisations de type buffet où chacun passe d'un comptoir à l'autre à sa guise. Si l'on prend peu de plats, on peut passer à la caisse avant un consommateur arrivé avant mais qui prend chaque plat. Evoquons les avantages et inconvénients des deux formules : attente dans le pipeline derrière l'indécis qui hésite entre frites et pommes sautées, bousculade dans le buffet asynchrone.

On suppose qu'à l'entrée les consommateurs arrivent à intervalles constants de durée T. Au premier plat présenté le premier client met moins de T à choisir. A l'instant T il passe alors au rayon du deuxième plat et un deuxième client prend sa place au premier rayon. A l'instant 2*T, le premier client passe au troisième rayon, le second client au second rayon et un troisième arrive dans le premier rayon. Le processus peut continuer. Si un client met plus de T à choisir ou s'il veut passer au rayon suivant plus vite que T, le système se met à mal fonctionner (Cf. "Les Temps Modernes" de Charlie Chaplin).

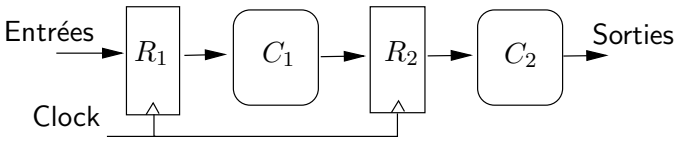


FIG. 10.24 – Un circuit à pipeline à trois niveaux

On a un fonctionnement du même type dans le système matériel décrit par la figure 10.24. A la date 1, un premier item d'entrée est chargé dans le registre d'entrée R_1. Le circuit combinatoire C_1 calcule une fonction f_1 à partir de la sortie de ce registre.

A la date 2, un deuxième item d'entrée est chargé dans le registre R_1. *Simultanément* R_2 est chargé avec la sortie du circuit C_1. Le circuit combinatoire C_1 calcule f_1 sur la sortie de R_1, C2 calcule f_2 sur la sortie de R_2.

La simultanéïté est nécessaire au bon fonctionnement du système. Evidemment la période T de l'horloge qui pilote les chargements de tous les registres doit être supérieure au maximum des délais des circuits combinatoires intervenant dans le pipeline. Là s'arrête l'analogie avec le restaurant self-service où l'on passe au rayon suivant de façon un peu asynchrone (dès que l'on a fini à un rayon et que le client précédent a libéré la place).

Le temps de réponse unitaire pour chaque traitement est le produit de la période par le nombre de tranches dans le pipeline. Il peut être supérieur à la somme des délais de chacun des circuits, voire très supérieur si le pipeline est mal équilibré.

4. Exercices

E10.2 : Compteur

On veut réaliser un compteur. Il délivre en sortie les entiers successifs de 0 à 7 (sur 3 bits). La sortie sur 3 fils (s_2, s_1, s_0) est incrémentée modulo 8 à chaque front montant d'une entrée incr. L'initialisation à 0 des sorties se fait à l'aide du signal init actif à 1. Donner l'automate de Moore correspondant à ces spécifications. Donner une réalisation câblée de cet automate.

Comment pourrait-on réaliser un circuit équivalent à l'aide d'un circuit combinatoire calculant une sortie S sur 3 bits égale à une entrée E (sur 3 bits) plus 1 ? Ce circuit a fait l'objet d'une étude de cas dans le chapitre 8.

Vérifiez en dessinant un chronogramme que le circuit de la figure 10.25 a le même comportement que les circuits précédents.

E10.3 : Automate reconnaisseur de séquence

Donner un codage binaire de trois informations a, b et c et construire une réalisation de l'automate étudié au chapitre 5 reconnaissant le langage régulier `a*b + c*`.

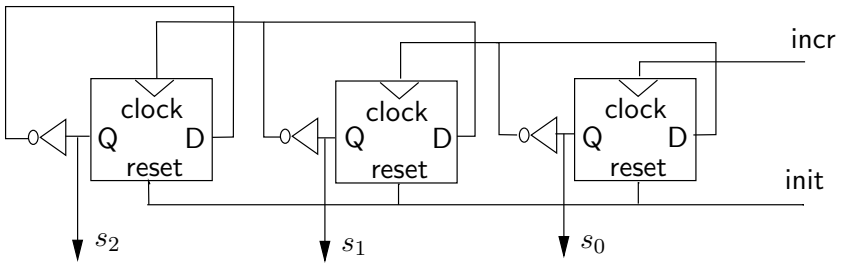


FIG. 10.25 – Circuit réalisant un compteur sur 3 bits

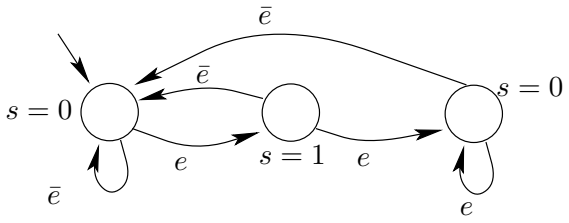


FIG. 10.26 – Graphe de Moore de l'automate correspondant au détecteur de front

E10.4 : Compteur/décompteur modulo 10

Un automate mémorise un entier naturel u de l'intervalle $[0, 9]$. Il a deux entrées plus, moins. L'évolution de l'état (c'est-à-dire de l'entier u) est définie ainsi :

```
u ⟵ 0 ;
tant que vrai :
        si plus alors u ⟵ (u+1) mod 10
        sinon
            si moins alors u ⟵ (u-1) mod 10
            sinon u ⟵ u
```

Donnez une réalisation en portes et bascules de ce compteur/décompteur modulo 10.

E10.5 : Détecteur de front

On veut réaliser l'automate du détecteur de front décrit dans le chapitre 9, paragraphe 1.2.4. On suppose que l'entrée e est synchronisée sur les fronts montants d'une horloge clock. La sortie s passe à 1 après chaque front montant de e et au front descendant de clock suivant. Elle doit rester à 1 jusqu'au prochain front montant de clock.

Vérifier que l'automate de la figure 10.26 correspond à ces spécifications. Faire la synthèse câblée de cet automate. Quelle doit être le signal que l'on doit mettre sur l'entrée d'activation des bascules de l'automate ? La sortie s de l'automate reste à 1 pendant une période de l'horloge, comment faire pour qu'elle reste à 1 seulement pendant la demi-période voulue ? Faire un chronogramme pour comprendre.

E10.6 : Machine à laver

On veut réaliser un contrôleur de machine à laver. La machine à laver possède 4 fils en entrée permettant d'effectuer des commandes sur les éléments de la machine : Lancer-Moteur-vitesse1, Lancer-Moteur-vitesse2, Entrée-Eau, Entrée-Lessive. La mise sous tension de 5 volts de ces fils correspond à :

- Lancer-Moteur-vitesse1 : active le moteur du tambour de la machine à une vitesse lente pendant une durée fixe T1 (permettant le lavage)
- Lancer-Moteur-vitesse2 : active le moteur du tambour de la machine à une vitesse rapide pendant une durée fixe T2 (permettant l'essorage)
- Entrée-Eau : permet l'arrivée de l'eau dans le tambour pendant une durée fixe T3 (active une pompe à eau)
- Entrée-Lessive : ouvre le conteneur de lessive (pour la mettre dans le tambour).

La machine à laver possède 1 fil Fin en sortie indiquant la fin d'une tâche lancée par l'une des 4 commandes précédentes. Ce fil passe à 5 Volts lorsque la tâche en cours se termine. Il repasse à 0 volts lors du lancement d'une nouvelle commande. Il est à 0 à l'initialisation.

On veut réaliser le circuit permettant de commander cette machine. Il a en entrée Fin, Init et Démarrer. Init est la commande d'initialisation à la mise sous tension. Démarrer passe à 5 volts un court instant lorsque l'utilisateur lance un programme de lavage. Il a en sortie les 4 commandes Lancer-Moteur-Vitesse1 (LMV1), Lancer-Moteur-Vitesse2 (LMV2), Entrée-Eau (EE), Entrée-Lessive (EL).

On veut offrir à l'utilisateur un seul programme de lavage qui correspond à la suite des étapes suivantes : un lavage : entrée de l'eau, de la lessive et lancement du tambour à vitesse lente ; un rinçage : entrée de l'eau et lancement du tambour à vitesse lente ; un essorage : lancement du tambour à vitesse rapide.

Pour les étapes à plusieurs tâches, les commandes correspondantes peuvent être effectuées simultanément. Fin passe alors à 1 à la fin de la tâche la plus longue. Donner le graphe de l'automate d'états fini qui correspond au contrôleur de la machine à laver. Faire une synthèse de cet automate en utilisant des bascules D sensibles au front montant et des portes NAND et inverseurs. Donner le dessin du circuit en faisant apparaître les entrées (Init, Démarrer, et Fin) et les sorties du circuit (LMV1, LMV2, EE, EL).

On veut maintenant offrir à l'utilisateur 2 programmes au choix. Pour cela on rajoute au contrôleur une entrée Prog spécifiant le programme de lavage à effectuer. Si Prog vaut 1 le programme de lavage est celui défini précédemment, si Prog vaut 0 le programme de lavage correspond seulement aux étapes de lavage et rinçage. Le contrôleur possède comme précédemment l'entrée Démarrage. Donner le graphe de l'automate correspondant à ce nouveau contrôleur et en réaliser une synthèse.

Chapitre 11

Conception de circuits séquentiels par séparation du contrôle et des opérations

Nous avons vu dans le chapitre 10 comment concevoir un circuit séquentiel correspondant à un automate d'états fini en partant de son graphe explicite. Ceci n'est faisable que lorsque le graphe de l'automate n'a pas trop d'états (une centaine). Au-delà le travail est complexe et fastidieux. C'est en particulier le cas lors de la réalisation de circuits correspondant à des algorithmes manipulant des variables entières. Dans ce cas d'autres techniques de conceptions de circuits existent. Nous avons vu dans le chapitre 10 à travers deux exemples une technique appelée *flot de données*. Nous nous intéressons ici à une technique différente permettant de décomposer le problème (et du coup le circuit correspondant) en deux parties distinctes : une *partie contrôle* et une *partie opérative*. La partie opérative est formée d'un ensemble de *registres* et d'*opérateurs* permettant de mémoriser les variables de l'algorithme et de réaliser les opérations apparaissant sur ces variables dans l'algorithme. Par contre l'enchaînement dans le temps des opérations est décidé par la partie contrôle. Par rapport au flot de données cette technique permet d'optimiser le nombre d'opérateurs nécessaires mais souvent au détriment du temps d'exécution.

Nous expliquons dans le paragraphe 1. le principe général d'une telle architecture. Nous décrivons dans le paragraphe 2. une partie opérative type, utilisable dans la plupart des cas. Nous donnons dans le paragraphe 3. les principes de la partie contrôle et étudions sa synchronisation temporelle avec la partie opérative. Nous appliquons ensuite ces principes généraux à deux exemples détaillés (paragraphe 4.).

Nous nous appuyons sur cette technique pour expliquer les principes de conception d'un processeur au chapitre 14.

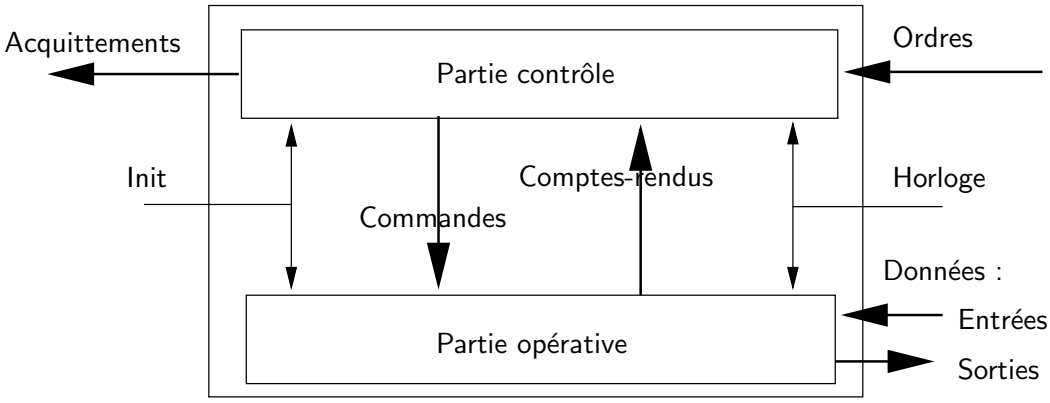


FIG. 11.1 – Principaux signaux d'une architecture PC/PO

1. Principe général

La *partie opérative* (ou PO) offre les ressources (Bus, registres, UAL ...) nécessaires à chaque opération sur les différentes variables apparaissant dans l'algorithme. Mais ce n'est pas elle qui décide de l'opération à effectuer à un instant donné. Elle envoie des signaux de comptes rendus sur ces calculs à la partie contrôle.

La *partie contrôle* (ou PC) gère l'enchaînement des calculs effectués sur les données au vu des comptes rendus de la PO. Elle génère l'activation des opérations à un instant donné, en envoyant des signaux de commandes à la partie opérative. Elle ne modifie pas directement les données. Elle traduit les primitives de contrôle apparaissant dans l'algorithme.

Les deux parties sont deux circuits séquentiels cadencés sur la même horloge. La figure 11.1 donne l'organisation des signaux entre la PC, la PO et le monde extérieur. Les communications entre les deux parties se font par les signaux de *commande* et de *compte-rendu*. A chaque front (montant par exemple) d'horloge :

- des valeurs de commandes sont envoyées à la PO par la PC pour sélectionner un calcul donné (par exemple : sélection des entrées de l'UAL, opération effectuée dans l'UAL, ...).
- des comptes-rendus du calcul effectué (par exemple les indicateurs arithmétiques d'une UAL) peuvent alors être renvoyés par la PO à la PC afin de lui permettre de prendre une décision pour le choix du calcul suivant.

Les connexions au monde extérieur tiennent compte de cette spécialisation :

- la PC ne reçoit que des ordres et ne délivre que des acquittements, signalant la fin de sa mission, par exemple.
- la PO ne reçoit que des données et ne délivre que des données. Les fils d'entrées et de sorties peuvent dans certains cas être les mêmes (bus bidirectionnel).

Nous avons vu au chapitre 5 comment obtenir une *machine séquentielle avec actions* à partir d'un algorithme. Chaque action apparaissant dans la machine séquentielle doit pouvoir être exécutée dans la PO. Les actions atomiques, ou *microactions*, auxquelles nous nous intéressons sont de type $R_i \longleftarrow R_j$ op R_k, $R_i \longleftarrow$ entrée ou sortie $\longleftarrow R_i$ où op est une opération et R_j un registre. On peut aussi avoir dans la PO type des microactions composées telle que sortie $\longleftarrow R_j \parallel R_i \longleftarrow R_j$ op R_k. La partie contrôle peut être décrite explicitement par le graphe de l'automate dans lequel on remplace les actions par un ensemble de sorties booléennes correspondant aux signaux de commande de la PO. Les entrées de cet automate sont les comptes-rendus des calculs effectués dans la PO. Le tableau 11.3 donne la correspondance entre microactions et sorties booléennes.

2. Notion de partie opérative type

Nous donnons Figure 11.2 la structure d'une partie opérative type permettant une construction méthodique et répondant à la plupart des besoins. Cette PO peut être dans la plupart des cas optimisée en termes de nombre de ressources (registres, bus, opérateurs) ou en termes de temps d'exécution. Ces critères sont souvent incompatibles.

Une partie opérative comprend des registres, un opérateur appelé UAL (Unité Arithmétique et Logique) et des bus.

> **Remarque :** Il est intéressant de noter qu'une partie opérative peut être décrite comme un automate d'états fini. Ses entrées sont les données entrantes, les commandes de chargement et d'initialisation des registres, les commandes d'opération. Son état est composé des valeurs contenues dans les différents registres. Ses sorties sont les données sortantes et les comptes-rendus à destination de la PC. Une utilisation systématique de cette description est faite dans l'exercice E14.4 du chapitre 14.

2.1 Registres et commandes de chargement

Les *registres* contiennent les valeurs des variables apparaissant dans l'algorithme. Un registre est un ensemble de bascules de même type partageant les mêmes commandes d'activation et d'initialisation (Cf. Chapitre 9, paragraphes 1.3 et 1.4). Les primitives matérielles sur un registre permettent d'y forcer une valeur présente sur la nappe de fils en entrée. Ceci est fait en connectant l'horloge générale du circuit à l'entrée d'horloge des bascules et en connectant l'entrée Enable des bascules à un signal de commande nommé signal de *chargement* de ce registre.

Sur la figure 11.2 chaque registre (R_i, $i = 1, ..., n$) est connecté à l'horloge générale et au signal de chargement noté ChRi. Ces signaux de chargement font partie des commandes envoyées par la PC.

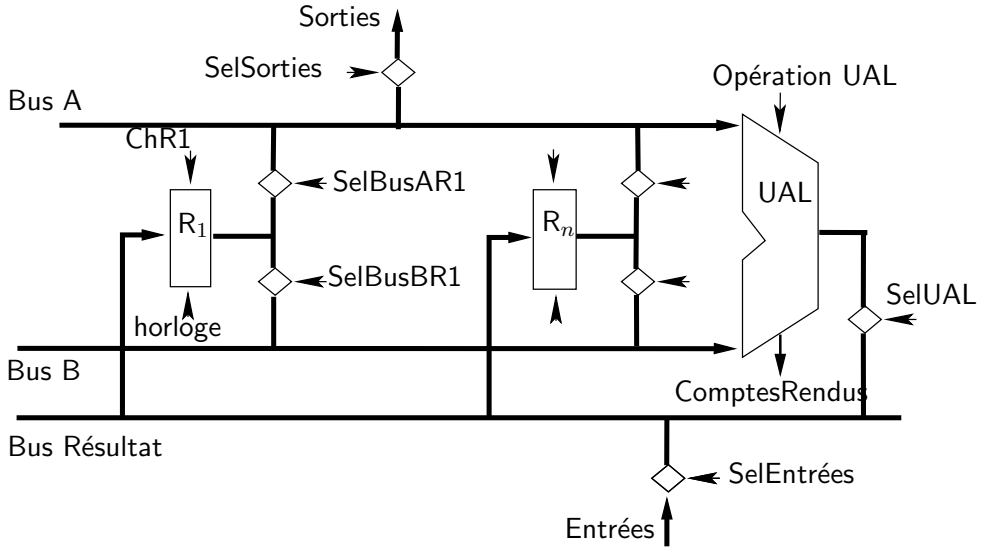


FIG. 11.2 – Une partie opérative type

microactions	commandes pendant le coup d'horloge
$R_i \longleftarrow R_j$ op R_k	$ChR_i = 1$; $ChR_{i'(i\neq i')} = 0$; $SelBusAR_j = 1$; $SelBusAR_{j'(j\neq j')} = 0$; $SelBusBR_k = 1$; $SelBusBR_{k'(k\neq k')} = 0$; $SelUAL = 1$; $SelEntrées = 0$; $SelSorties = 0$; commandes UAL pour l'opération op
$R_i \longleftarrow$ entrée	$ChR_i = 1$; $ChR_{i'(i\neq i')} = 0$; $SelBusAR_j = \varphi$; $SelBusBR_k = \varphi$; $SelUAL = 0$; $SelEntrées = 1$; $SelSorties = 0$; commandes UAL $= \varphi$
sortie $\longleftarrow R_i$	$ChR_i = 0$; $SelBusAR_i = 1$; $SelBusBR_k = \varphi$; $SelUAL = 0$; $SelEntrées = 0$; $SelSorties = 1$; commandes UAL $= \varphi$

FIG. 11.3 – Sorties correspondant aux microactions pour la P.O type de la figure 11.2

La sortie de chaque registre est connectée aux deux entrées de l'UAL mais une de ces deux connexions peut être inutile et donc supprimée. Il peut être intéressant d'initialiser les registres à l'aide du signal d'initialisation (à 1 ou à 0) des bascules plutôt que d'obtenir des valeurs initiales via le bus Entrées.

2.2 Opérateur et commande d'opération

La réalisation de l'opérateur (ou UAL) suppose d'identifier la liste des opérations nécessaires à l'exécution de l'algorithme. L'opérateur est un circuit combinatoire susceptible de réaliser ces différentes opérations selon des signaux de commande (OpérationUAL sur la figure 11.2). Si l'opérateur doit effectuer p opérations différentes, il est commandé par $\log_2(p)$ fils de commande. En plus du résultat de l'opération proprement dit, des sorties de l'opérateur peuvent être de type indicateurs arithmétiques, ressemblant aux classiques Z, N, C et V des mots d'état de processeurs (Cf. Chapitre 12). Rien n'empêche d'en utiliser d'autres si l'expression de l'algorithme utilise des primitives de test différentes. Ces sorties de l'opérateur forment les signaux de compte-rendu de la PO vers la PC (ComptesRendus sur la figure 11.2).

Il y a dans ce travail de conception d'un opérateur un aspect tout à fait particulier. L'expression d'un algorithme se fait à base de primitives supposées données ; ici le concepteur de machine algorithmique a le choix des primitives : ce sont celles réalisables par un circuit à un coût convenable. On peut par exemple décider d'utiliser un multiplieur combinatoire de nombres codés en virgule flottante sur 64 bits dans une machine algorithmique si on en a besoin. Cela permet de considérer la multiplication de réels comme une primitive. Si l'on ne veut pas utiliser un tel circuit, mais seulement un additionneur 8 bits, il faudra exprimer l'algorithme en n'utilisant que des additions de nombres codés sur 8 bits.

On peut utiliser plusieurs opérateurs effectuant chacun une opération donnée afin de pouvoir paralléliser certaines opérations et donc diminuer le temps d'exécution de l'algorithme. Toutefois ce genre d'optimisation augmente le nombre de connexions et d'opérateurs nécessaires. Si on le pousse à l'extrême, on retombe sur la solution flot de données présentée au paragraphe 3. du chapitre 10.

2.3 Liaisons, bus et multiplexeurs, commandes de sélection

Les liaisons entre les registres et l'opérateur se font par des liaisons nommées *bus*. Deux bus opérandes (A et B) permettent d'amener aux deux entrées de l'UAL le contenu d'un des registres. Les signaux SelBusARi et SelBusBRi permettent d'effectuer cette sélection. Un bus Résultat permet d'amener à l'entrée des n registres le résultat du calcul. Les signaux ChRi permettent d'effectuer le chargement du registre souhaité.

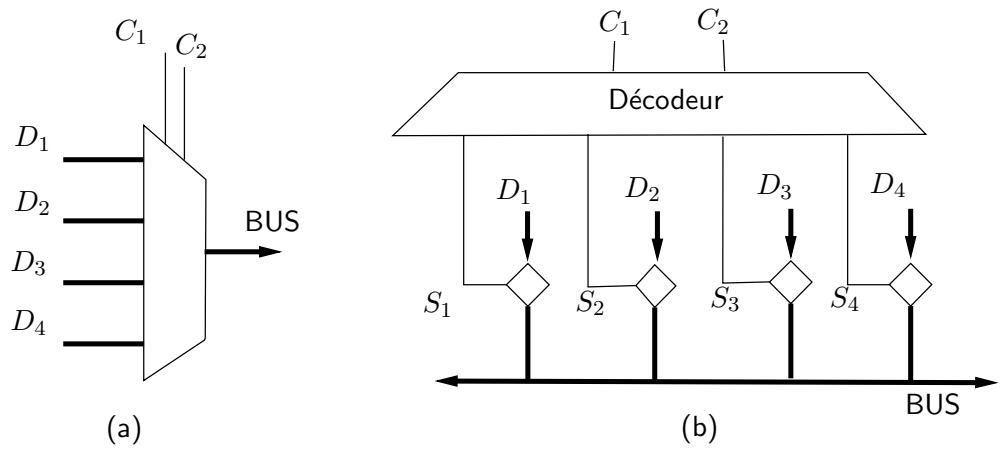


FIG. 11.4 – Liaisons de 4 registres à un bus. a) à base de multiplexeurs ; b) à base de portes trois états.

La liaison des registres à chaque bus est réalisée soit à l'aide de multiplexeurs, soit à l'aide de portes trois états entre les sorties des registres et le bus. Elle nécessite des signaux de commande permettant de déterminer ce choix. La figure 11.4 montre la réalisation d'un bus supportant 4 entrées D_1, D_2, D_3 et D_4 (qui sont par exemple les sorties de 4 registres) à l'aide de deux signaux de commande C_1 et C_2. Dans le cas d'une réalisation à base de portes trois états, le bus obtenu est à double sens (contrairement à l'autre cas). Cela peut être indispensable dans le cas d'entrées/sorties avec l'extérieur. Cette solution est la plus souvent utilisée.

Sur la figure 11.2, c'est la solution à base de portes trois états qui a été choisie. Le décodeur délivrant les signaux S_i (dans la figure 11.4) n'apparaît pas sur la figure 11.2. Ces signaux de sélections (S_i) peuvent être directement délivrés dans chaque état de la PC. Une autre solution consiste a réaliser le décodeur dans la PC ; le nombre de fils de commande entre la PC et la PO est alors fortement augmenté.

On peut diminuer le nombre de connexions en diminuant le nombre de bus au détriment du temps d'exécution de l'algorithme. On peut placer par exemple un registre tampon supplémentaire en sortie de l'UAL et connecter ce registre à un bus qui servira en même temps de bus résultat et de bus opérande.

2.4 Entrées/Sorties

Le bus Entrées permet de charger des valeurs depuis l'extérieur dans les registres. Les signaux SelEntrées et ChRi du registre concerné doivent alors être actifs et la valeur initiale présente sur le bus Entrées.

Le bus Sorties permet de délivrer à l'extérieur les résultats de l'algorithme. Il est donc connecté à un des deux bus de sorties des registres de la PO. La porte trois états activée par SelSorties n'est pas toujours nécessaire.

Dans certains cas, il peut être intéressant de posséder plusieurs nappes de fils de sorties. Dans ces cas-là les sorties de certains registres peuvent être, par exemple, directement des sorties du circuit.

Dans d'autres cas les entrées et les sorties peuvent être multiplexées sur les mêmes fils. On peut alors relier les bus Entrées et Sorties grâce à la présence de la porte trois états commandée par SelSorties sur la figure 11.2.

2.5 Relations entre microactions et commandes

On peut récapituler (Cf. Figure 11.3) l'ensemble des commandes nécessaires à l'exécution des 3 types de microactions sur la base d'une partie opérative comme celle de la figure 11.2. On verra dans la suite des primitives d'entrées/sorties plus riches permettant un protocole de poignée de mains.

2.6 Synchronisation du calcul et de l'affectation du résultat

Deux schémas simples de synchronisation peuvent être utilisés. Dans le premier schéma, tous les registres sont sensibles au (même) front d'horloge. A chaque coup d'horloge une microaction $R_i \longleftarrow R_j$ op R_k est effectuée et le résultat est chargé dans le registre concerné au prochain front de l'horloge.

Dans le deuxième schéma, on peut utiliser des registres de type verrou (Cf. Chapitre 9) ; on ajoute alors un registre tampon en sortie de l'opérateur. Ce registre est piloté par un signal de chargement actif sur le niveau haut de l'horloge. Les autres registres sont pilotés par un signal de chargement actif sur le niveau bas. Une microaction se décompose alors en deux phases, correspondant aux deux niveaux de l'horloge. Dans la première phase (haut) les opérandes sont aiguillés vers l'opérateur et le résultat du calcul est chargé dans le tampon. Dans la deuxième phase (bas), le résultat est chargé dans le registre concerné.

Dans la suite on fera abstraction de cette alternative de mise en oeuvre des actions atomiques.

3. Partie contrôle

Comme décrit au chapitre 5, nous pouvons à partir de l'algorithme obtenir une *machine séquentielle avec actions*. Pour des raisons de synchronisation avec la PO (Cf. Chapitre 10, paragraphe 1.3.3) le modèle de Moore est utilisé.

Cette *machine séquentielle avec actions* est ensuite transformée en automate d'états fini en remplaçant les actions apparaissant sur les états par l'affectation des valeurs correspondantes à ces actions, aux signaux de commande à destination de la PO. Cette étape est détaillée dans les études de cas traitées

au paragraphe 4. Cet automate peut être ensuite réalisé par du matériel suivant une des méthodes décrites dans le chapitre 10.

Le problème est de décider quels opérateurs et comptes-rendus de calculs sont disponibles dans la PO. Chaque calcul et affectation de variable correspondante effectué dans un état de l'automate doit être réalisable en un cycle d'horloge dans la PO. Chaque condition apparaissant sur les transitions de l'automate doit être un compte-rendu de l'opérateur utilisé disponible dans l'état précédent.

3.1 Entrées/sorties

La gestion des entrées et des sorties nécessite une synchronisation avec le monde extérieur. Le protocole de *poignée de mains* (Cf. Chapitre 6) peut être employé pour permettre le chargement ou la sortie de certains registres de la PO. Ce protocole de poignée de mains peut être adapté au cas par cas suivant l'environnement dans lequel on doit implanter le circuit à réaliser. Des signaux de synchronisation nécessaires à ces entrées/sorties sont ajoutés aux signaux de données. Ils sont reçus par la PC.

Dans le cas d'une entrée le circuit est le récepteur et le monde extérieur est l'émetteur, et inversement pour une sortie. On reprend le schéma de l'automate du récepteur et de l'émetteur dans une poignée de mains présenté dans le chapitre 6. On associe à l'automate de la PC deux états pour chaque acquisition d'entrée et pour chaque délivrance de sortie (Cf. Figure 11.5). Les entrées sont échantillonnées sur la même horloge que la PC comme nous l'avons vu dans le chapitre 10. Dans le cas d'une entrée, le signal PresE correspond au signal de présence d'une entrée venant de l'extérieur (émetteur prêt). Le signal EPrise correspond au signal de signification à l'extérieur de la prise en compte de l'entrée (récepteur non prêt). Bien entendu ce signal est à 0 dans tous les autres états de l'automate. Dans l'état Chargement de l'entrée les commandes à destination de la PO sont SelEntrées pour amener la valeur du bus extérieur à l'entrée des registres et ChRi pour charger le registre voulu.

Dans le cas d'une sortie, le signal PresS correspond au signal de présence d'une sortie pour l'extérieur (émetteur prêt). Ce signal est à 0 dans tous les autres états de l'automate. Le signal SPrise permet à l'extérieur de signaler au circuit qu'il a pris en compte la sortie (récepteur non prêt). Dans l'état Sortie présente les commandes à destination de la PO sont SelBusARi pour transmettre le registre voulu sur le bus A et SelSorties pour amener la valeur du bus A sur le bus de sortie.

Nous pouvons illustrer ce protocole à travers le dialogue d'un circuit avec une mémoire. Considérons par exemple un processeur effectuant des écritures (sorties) ou des lectures (entrées) en mémoire. Nous avons expliqué au paragraphe 2.2 du chapitre 9 comment se déroule un accès mémoire. Dans le cas où la mémoire est *lente* par rapport au processeur celui-ci doit attendre lors d'un accès en lecture ou en écriture que la mémoire lui signale la fin de l'accès.

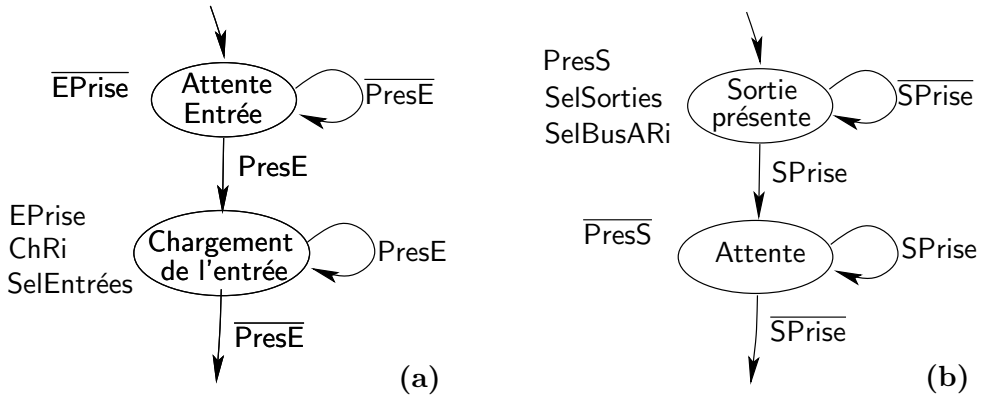


FIG. 11.5 – Gestion d'entrées/sorties : a) acquisition d'une entrée ; b) délivrance d'une sortie

La mémoire délivre alors un signal FinAccès lorsque, soit la donnée à lire est prête sur le bus, soit la donnée à écrire est effectivement écrite en mémoire. Le processeur délivre les signaux SelMem et $\mathsf{l}/\overline{\mathsf{e}}$ lors d'un accès à la mémoire. On reprend les automates de la figure 11.5. Pour l'écriture SPrise correspond à FinAccès, PresS à SelMem et $\overline{\mathsf{l}/\overline{\mathsf{e}}}$; pour la lecture PresE correspond à FinAccès, $\overline{\text{Eprise}}$ à SelMem et $\mathsf{l}/\overline{\mathsf{e}}$.

> *Nous retrouverons la connexion d'un processeur avec une mémoire dans l'étude de cas du paragraphe 4.3 et dans le chapitre 14. Nous nous placerons alors dans le cas simple et idéal où la mémoire est suffisamment rapide pour permettre un accès en un cycle d'horloge du processeur ; le signal* FinAccès *n'est alors plus utile, il est implicite.*
> *Nous retrouverons les aspects de synchronisation au chapitre 15, pour relier l'ensemble processeur/mémoire avec le monde extérieur.*

3.2 Synchronisation de la partie contrôle et de la partie opérative

Nous avons vu dans le chapitre 10, paragraphe 1.3.3, comment synchroniser les réalisations de deux automates. Dans le cas d'une réalisation PC/PO les sorties de la PO sont des entrées de la PC et inversement. Nous sommes donc en présence de deux automates en boucle.

Supposons que l'automate réalisant la PC évolue à chaque front montant d'une horloge H. Il faut qu'entre deux fronts montants de H, la PO effectue le calcul commandé et donne un compte-rendu pour permettre à la PC de calculer le prochain état. Regardons ce qui se passe dans les deux cas de synchronisation étudiés au paragraphe 1.3.3 du chapitre 10.

Dans le cas où les registres de la PO sont chargés au front montant (Cf. Figure 11.6), c'est le résultat du calcul effectué dans l'état précédent qui est chargé. La PO doit alors effectuer le calcul et émettre le compte-rendu de manière à laisser le temps à la PC de calculer l'état suivant avant le prochain

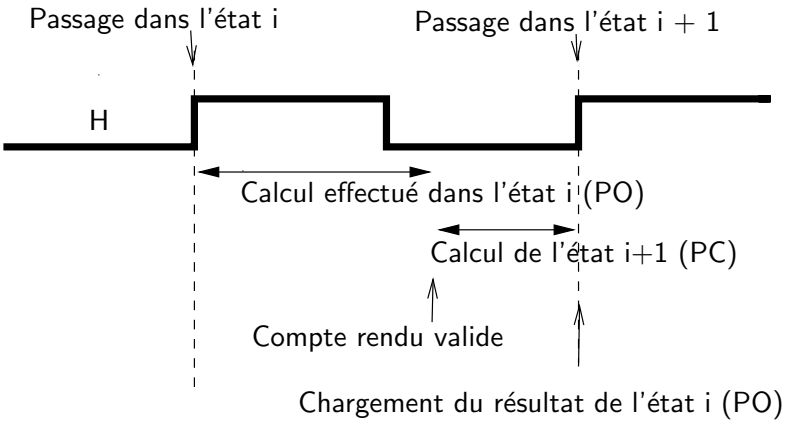


FIG. 11.6 – Chronogrammes montrant la synchronisation de la PC et de la PO avec chargement du registre résultat en fin de période

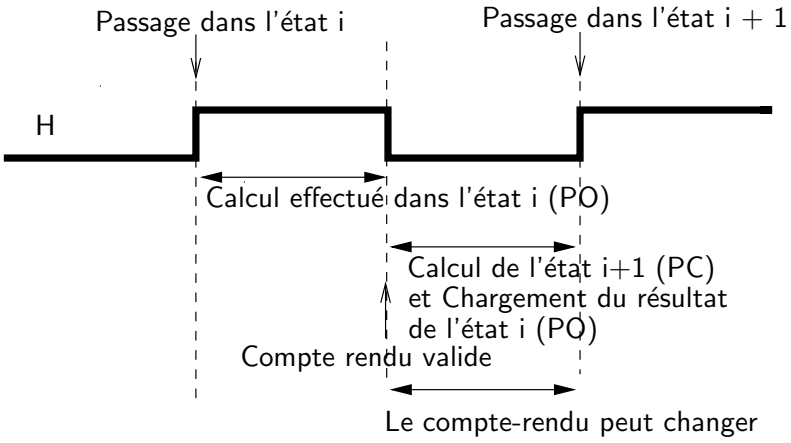


FIG. 11.7 – Chronogrammes montrant la synchronisation de la PC et de la PO avec chargement du registre résultat au milieu de la période

front montant de H. Dans ce cas-là, il n'est pas nécessaire de mémoriser ces comptes-rendus.

Dans le cas où le calcul est effectué dans la première phase (Cf. Figure 11.7) les comptes-rendus changent alors une deuxième fois dans la deuxième phase puisque certains des registres peuvent avoir changé après le front descendant de H. Il faut donc mémoriser ces comptes-rendus dans un registre au front descendant de H. L'émission des commandes de la PC et le calcul doivent donc être réalisés dans la première demi-période de H, le calcul de l'état suivant dans la deuxième. Cette solution est plus contraignante dans l'enchaînement des différentes opérations de la PC et de la PO.

4. Etudes de cas

4.1 Démarche de conception

Avant de traiter concrètement des exemples précis nous donnons l'ébauche d'une méthode de conception d'une architecture PC/PO. Les deux premières étapes apparaissant ci-après sont complètement liées et sont à effectuer simultanément.

- **Transformation de l'algorithme en une machine séquentielle avec actions :**

  Répertorier les actions et les prédicats apparaissant dans l'algorithme. Définir les opérations nécessaires aux calculs de ces prédicats. Définir les opérations et les affectations de variables pouvant être réalisables par un circuit disponible pour construire la PO.

  Décrire l'algorithme à l'aide d'une machine séquentielle avec actions utilisant ces variables et ces opérations. Les actions associées à chaque état doivent être réalisables dans la PO en un coup d'horloge.

- **Obtention de la PO :**

  Répertorier l'ensemble des variables apparaissant dans la machine séquentielle avec actions, chacune correspond à un registre de la PO.

  Répertorier l'ensemble des opérations apparaissant dans l'algorithme, y compris celles nécessaires aux calculs des prédicats. Construire une PO type (Cf. Parapraphe 2.) possédant un opérateur permettant de réaliser tous les calculs. On pourra éventuellement optimiser cette PO (au niveau temps de calcul) en multipliant les opérateurs et parallélisant les calculs par fusion d'états dans la machine séquentielle.

- **Obtention de l'automate correspondant à la PC :**

  Au vu de la PO et de la machine séquentielle avec actions obtenir l'automate d'états fini correspondant à la PC. Cet automate a comme entrées les fils de comptes-rendus (correspondant aux conditions apparaissant dans l'algorithme) sortant de l'opérateur de la PO et les signaux de synchronisation avec le monde extérieur. Il a comme sorties les fils de commande apparaissant sur les ressources de la PO et des signaux avec l'extérieur. Il faut donc définir pour chaque état la valeur des commandes à destination de la PO correspondant aux actions effectuées.

- **Synthèse de l'automate de contrôle :**

  Il reste à effectuer la synthèse de l'automate obtenu en utilisant une des méthodes données dans le chapitre 10.

- **Assemblage de la PC et de la PO**

```
Lexique
   m,n : des entiers ≥ 0 { m et n étant donnés}
   fin : le booléen Vrai
   j, k : des entiers ≥ 0
   Δ : un entier
Algorithme
   Tantque VRAI :
      Acquérir(m) ; Acquerir(n) ;
      k ⟵ 0 ; j ⟵ 0 ; Δ ⟵ −m ;
      tantque j ≤ m : {invariant : 0 ≤ j ≤ m et −2.m ≤ Δ ≤ 0 }
            Délivrer (j) ; Délivrer (k) ;
            j ⟵ j+1 ;
            Δ ⟵ Δ + 2.n ; {−2.m + 2.n ≤ Δ ≤ 2.n}
            si Δ ≥ 0
               k ⟵ k + 1 ;
               Δ ⟵ Δ − 2.m ; { après cela : −2.m ≤ Δ ≤ 0 }
      fin ⟵ VRAI ;
      Délivrer(fin) ;
```

FIG. 11.8 – Algorithme de Bresenham

4.2 Le traceur de segments

Le circuit que nous voulons concevoir doit délivrer les coordonnées des points d'un segment de droite sur un écran. Nous choisissons l'algorithme de Bresenham présenté au paragraphe 2.3 du chapitre 5. La figure 11.8 rappelle l'algorithme. Les actions Acquérir et Délivrer correspondent aux entrées et sorties. Pour que le circuit puisse fonctionner pour un nombre de segments illimité, nous ajoutons une boucle globale sans fin.

4.2.1 Obtention d'une machine séquentielle avec actions à partir de l'algorithme

On peut classer les actions à effectuer en deux catégories. Certaines correspondent à des entrées/sorties : Acquérir(m), Acquérir(n), Délivrer (j), Délivrer(k), Délivrer (fin) ; d'autres correspondent à des calculs : $j \longleftarrow 0$, $k \longleftarrow 0$, $\Delta \longleftarrow -m$, $j \longleftarrow j + 1$, $\Delta \longleftarrow \Delta + 2.n$, $k \longleftarrow k + 1$, $\Delta \longleftarrow \Delta - 2.m$.

Les initialisations de j et k à 0 se font directement à travers un signal d'initialisation sur les registres. On peut ainsi réunir les actions $j \longleftarrow 0$, $k \longleftarrow 0$, et $\Delta \longleftarrow -m$ dans le même état. On choisit d'utiliser un seul opérateur (Cf. Paragraphe 2.), les autres actions doivent donc se faire dans des états distincts.

Les prédicats à tester sont $j \leq m$ et $\Delta \geq 0$. On peut de façon équivalente calculer chaque prédicat ou son complémentaire ; par exemple on peut calculer $j>m$ ou $j \leq m$. Pour le calcul du prédicat $j > m$ on effectue $m - j$. Il est plus

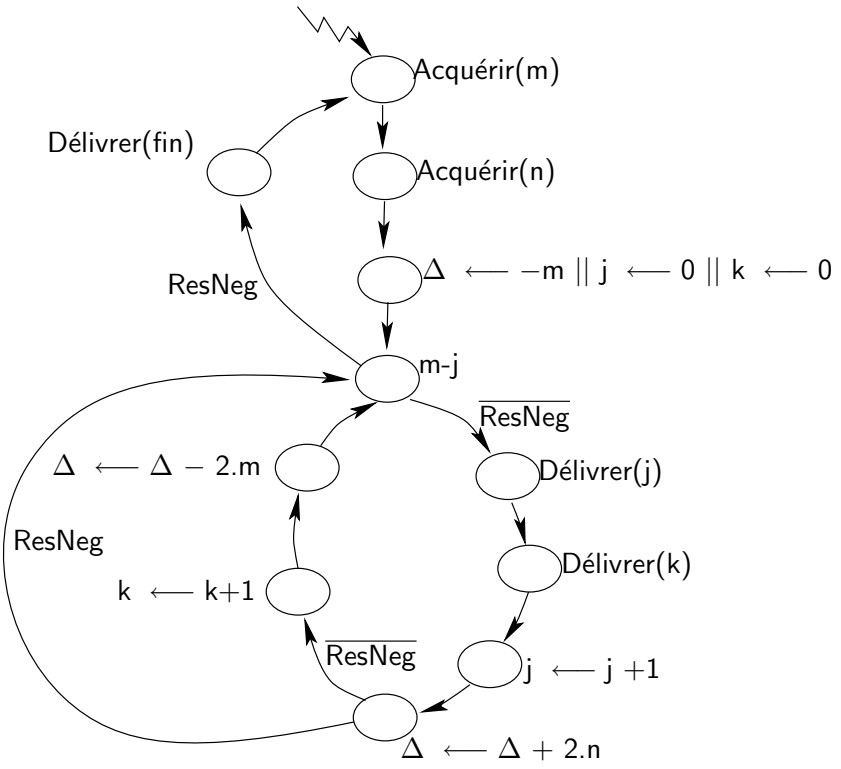


FIG. 11.9 – Une machine séquentielle avec actions réalisant l'algorithme de Bresenham

facile de réaliser une UAL qui donne un indicateur signifiant résultat négatif que résultat négatif ou nul. Pour le prédicat $\Delta \geq 0$, on a besoin du même indicateur.

Le calcul d'un prédicat peut nécessiter l'ajout d'un état. Par exemple ici le calcul de $j > m$ doit être effectué dans un état spécifique alors que le calcul de $\Delta \geq 0$ peut être effectué au moment de l'action $\Delta \longleftarrow \Delta + 2.n$.

On obtient la machine séquentielle avec actions de la figure 11.9.

4.2.2 Description de la partie opérative

La figure 11.11 donne la partie opérative. Le circuit comporte 5 registres nommés J, K, M, N et D contenant les valeurs des variables j, k, m, n et Δ.

Les registres J et K possédent un signal d'initialisation InitJK à 0 pour pouvoir effectuer les actions $j \longleftarrow 0$ et $k \longleftarrow 0$.

Les calculs à effectuer correspondent aux diverses actions énumérées précédemment : − M, K + 1, J + 1, D + 2.N, D − 2.M. Il faut ajouter les calculs correspondant aux prédicats : $j > m$ et $\Delta \geq 0$.

On utilise une PO type comme définie précédemment mais on minimise le nombre de connexions des registres aux bus. Ainsi les connexions $J \rightsquigarrow BusA$, $K \rightsquigarrow BusA$, $M \rightsquigarrow BusB$, $N \rightsquigarrow BusB$, $D \rightsquigarrow BusA$ suffisent.

Nous réalisons toutes les opérations avec un opérateur unique ; il doit

Op2	Op1	Op0	Opération
0	0	0	A + 1
0	0	1	B − A
0	1	0	A + 2*B

Op2	Op1	Op0	Opération
0	1	1	A − 2*B
1	0	0	− B

FIG. 11.10 – Signaux de commande de l'UAL de l'algorithme de Bresenham

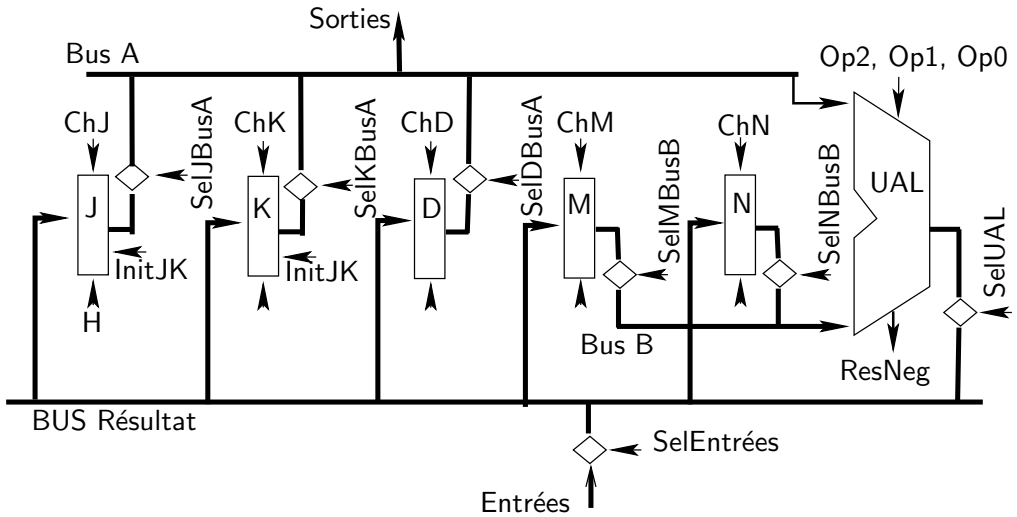


FIG. 11.11 – Partie opérative correspondant à l'algorithme de Bresenham

réaliser les opérations suivantes sur ses opérandes A et B : A+1, B−A, A+2*B, A−2*B et −B. Chacune de ces opérations doit être effectuée dans un état différent de la machine séquentielle avec actions.

La signification des trois signaux Op2 , Op1 , Op0 de commande des opérations de l'UAL est donnée dans le tableau de la figure 11.10.

Cette UAL peut être réalisée à l'aide d'un additionneur (Cf. Chapitre 8). Elle doit générer un bit de signe du résultat (ResNeg).

4.2.3 Hypothèses sur les entrées/sorties

Sans hypothèses sur l'environnement exact (écran, table traçante,..) il est impossible de décrire précisément la primitive d'affichage d'un pixel.

Nous convenons que M et N sont initialisés à partir d'un bus d'entrées (noté Entrées sur la figure 11.11), que J et K sont délivrés sur un bus de sorties (noté Sorties sur la figure 11.11) et que le signal Fin est délivré directement par un fil particulier issu de la PC (fil portant une valeur constante, on pourrait bien sûr l'éliminer).

Pour les entrées sur M et N et les sorties sur Fin, J et K nous utilisons le protocole de poignée de mains.

4.2.4 Définition de la partie contrôle par un automate d'états fini

La figure 11.12 donne la partie contrôle. Les états correspondant à Acquérir(N) et Acquérir(M) sont remplacés chacun par les deux états du récepteur dans le protocole poignée de mains. On introduit ainsi les signaux de contrôle venant de l'extérieur (PresN et PresM) et un même signal vers l'extérieur pour les deux cas EPrise. De même pour les sorties Fin, J et K avec les signaux SortieFin, SortieJ, SortieK et SPrise.

Nous supposons que les bascules utilisées dans la PO sont des bascules D sensibles au front montant, avec signal d'initialisation actif à 1.

Il est à remarquer que dans certains cas la valeur de signaux de commande de la PO n'a pas d'importance. Ces valeurs sont alors définies comme phi-booléennes pour les fonctions de sortie correspondantes.

La figure 11.13 donne le détail des sorties pour chaque état de la PC ; ce sont des fils de commande apparaissant sur la PO de la figure 11.11 et les fils destinés aux entrées/sorties avec l'extérieur.

4.2.5 Assemblage de la PC et de la PO

Le circuit de la figure 11.14 montre l'ensemble des signaux échangés lors de l'assemblage de la PC et de la PO ainsi que ceux permettant la gestion des entrées/sorties. Le seul compte-rendu de la PO vers la PC est ResNeg. La PC et la PO évoluent avec la même horloge H.

4.3 La machine à trier

Cette étude de cas montre l'utilisation d'une mémoire à côté d'une machine algorithmique. C'est à ce titre une bonne introduction au chapitre 14 qui porte sur la conception d'un processeur. Le mécanisme d'entrée/sortie utilisé ici est spécifique et différent de ceux explicités plus haut.

Soit une mémoire permettant de stocker M mots de N bits. On désire construire une machine permettant de réaliser le tri des éléments contenus dans cette mémoire. On considère que ces éléments sont des entiers naturels codés en base 2. La figure 11.15 donne l'algorithme du tri par insertion.

On suppose que M est initialisé au départ de l'algorithme. La mémoire comporte un bus donnée, un bus adresse et un signal de commande, noté $\mathsf{l}/\overline{\mathsf{e}}$, précisant si on réalise une opération de lecture ou d'écriture. Il faut donc que le circuit réalisant l'algorithme possède un bus de sortie correspondant à l'adresse en mémoire et un bus d'entrées/sorties correspondant à la donnée stockée en mémoire. On suppose que l'accès à la mémoire en lecture ou écriture peut se faire en un cycle d'horloge du circuit à concevoir. Il n'y a donc pas de signal FinAccès.

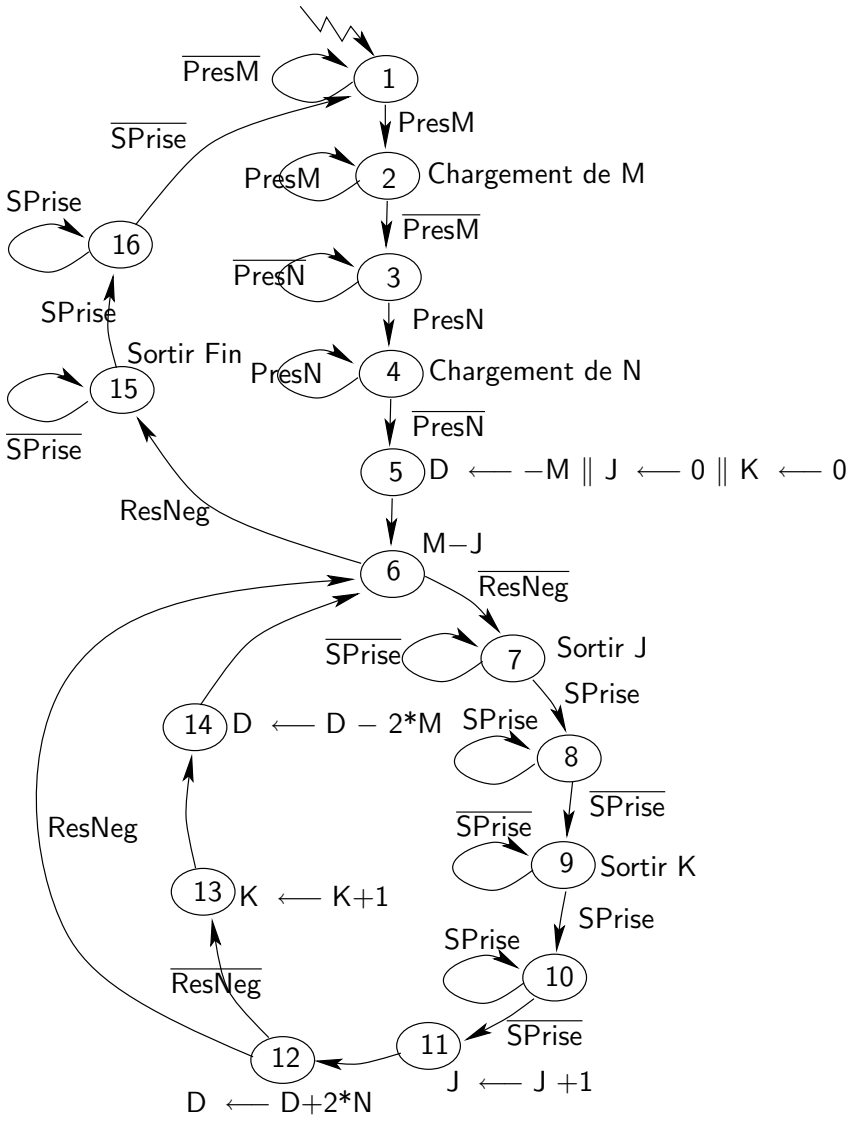


FIG. 11.12 – Automate d'états fini correspondant à l'algorithme de Bresenham

1	Attente : InitJK = ChJ = ChK = ChD = ChM = ChN = ϕ, (Op2, Op1, Op0) = (ϕ, ϕ, ϕ)
2	Chargement de M : ChM = SelEntrées = EPrise = 1, InitJK = ChJ = ChK = ChD = ChN = ϕ, (Op2, Op1, Op0) = (ϕ, ϕ, ϕ)
3	Attente : InitJK = ChJ = ChK = ChD = ChN = ϕ, (Op2, Op1, Op0) = (ϕ, ϕ, ϕ)
4	Chargement de N : ChN = SelEntrées = EPrise = 1, InitJK = ChJ = ChK = ChD = ϕ, (Op2, Op1, Op0) = (ϕ, ϕ, ϕ)
5	D $\longleftarrow -$M ; J $\longleftarrow$ 0 ; K $\longleftarrow$ 0 : ChD = InitJK = SelMBusB = SelUAL = 1, (Op2, Op1, Op0) = (1, 0, 0)
6	M $-$ J : SelJBusA = SelMBusB = 1, (Op2, Op1, Op0) = (0, 0, 1)
7	Sortir J : SortieJ = SelJBusA = 1, (Op2, Op1, Op0) = (ϕ, ϕ, ϕ)
8	Attente : (Op2, Op1, Op0) = (ϕ, ϕ, ϕ)
9	Sortir K : SortieK = SelKBusA = 1, (Op2, Op1, Op0) = (ϕ, ϕ, ϕ)
10	Attente : (Op2, Op1, Op0) = (ϕ, ϕ, ϕ)
11	J $\longleftarrow$ J+1 : ChJ = SelJBusA = SelUAL = 1, (Op2, Op1, Op0) = (0, 0, 0)
12	D $\longleftarrow$ D+2*N : ChD = SelDBusA = SelNBusB =SelUAL = 1, (Op2, Op1, Op0)=(0, 1, 0)
13	K $\longleftarrow$ K+1 : ChK = SelUAL = SelKBusA = 1, (Op2, Op1, Op0) =(0, 0, 0)
14	D $\longleftarrow$ D$-$2*M : ChD = SelDBusA = SelMBusB = SelUAL = 1, (Op2, Op1, Op0) = (0,1,1)
15	Sortir fin : SortieFin = 1, InitJK = ChJ = ChK = ChD = ChM = ChN = ϕ, (Op2, Op1, Op0) = (ϕ, ϕ, ϕ)
16	Attente : InitJK = ChJ = ChK = ChD = ChM = ChN = ϕ, (Op2, Op1, Op0) = (ϕ, ϕ, ϕ)

FIG. 11.13 – Sorties émises dans chaque état de la partie contrôle de la figure 11.12. Les signaux dont la valeur est 1 ou ϕ sont précisés et ceux pour lesquels elle vaut 0 sont omis.

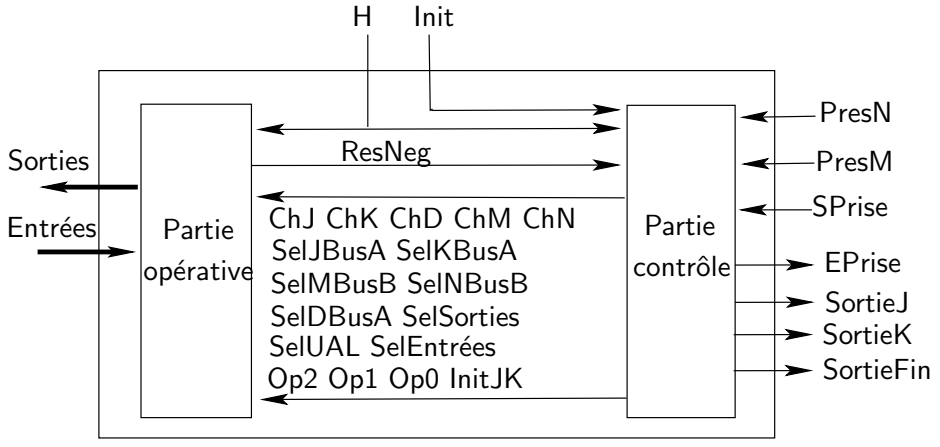


FIG. 11.14 – Vue générale du traceur de segments

```
Lexique
   M : un entier donné
   i, j, t : des entiers ≥ 0 ; trouvé, fin : des booléens
   mem : un tableau [0..M−1] d'entiers ≥ 0
Algorithme
   i ⟵ 1
   tantque i ≤ M−1 :
      j ⟵ i ; t ⟵ mem[i] ; trouvé ⟵ FAUX
      tantque (j > 0 et non trouvé) :
         si mem[j−1] > t alors mem [j] ⟵ mem[j−1] ; j ⟵ j−1
         sinon trouvé ⟵ VRAI
      mem [j] ⟵ t ; i ⟵ i + 1
   fin ⟵ VRAI ;
   Délivrer (fin)
```

FIG. 11.15 – Algorithme de la machine à trier

4.3.1 Obtention d'une machine séquentielle avec actions à partir de l'algorithme

Les actions à effectuer sont :
i ⟵ 1, j ⟵ i, t ⟵ mem[i], trouvé ⟵ FAUX,
mem[j] ⟵ mem[j−1], j ⟵ j−1, trouve ⟵ VRAI, mem[j] ⟵ t, i ⟵ i+1.

Nous convenons de coder FAUX par la valeur 0 et VRAI par la valeur 1. Etudions l'évaluation des prédicats :

- i ≤ M−1 : on peut calculer i − M et tester si le résultat est nul. En effet, i − M ≤ −1 ⟺ i − M < 0. Initialement i < M et i étant incrémenté de 1 à chaque itération on peut tester i = M.
- j > 0 et non (trouvé) : on peut calculer j et tester si le résultat est non nul, puis calculer trouvé et tester si le résultat est nul. Initialement j > 0 et j étant décrémenté de 1 à chaque itération on peut tester j = 0.
- mem[j−1] > t : on calcule mem[j−1] − t et on teste si le résultat est strictement positif. Pour ce calcul si l'on se restreint à un seul opérateur, il faut tout d'abord aller chercher en mémoire mem[j−1] et stocker la valeur dans une variable temporaire que nous appelons temp. On peut ensuite calculer temp − t. Le calcul se fait donc dans deux états successifs réalisant les actions temp ⟵ mem[j−1] puis évaluation du signe de temp − t.

L'affectation mem [j] ⟵ mem[j−1] nécessite deux accès à la mémoire. On ne peut donc la réaliser dans un seul état de la machine séquentielle. Il faut comme précédemment utiliser une variable, soit temp, permettant de stocker mem[j−1] avant de l'affecter à mem[j] ; à deux états successifs seront alors associées les actions temp ⟵ mem[j−1] puis mem[j] ⟵ temp. Comme l'affectation temp ⟵ mem[j−1] se fait juste après le test mem[j−1] > t, temp contient déjà la valeur de mem[j−1] ; il est donc inutile de réaliser cette affectation.

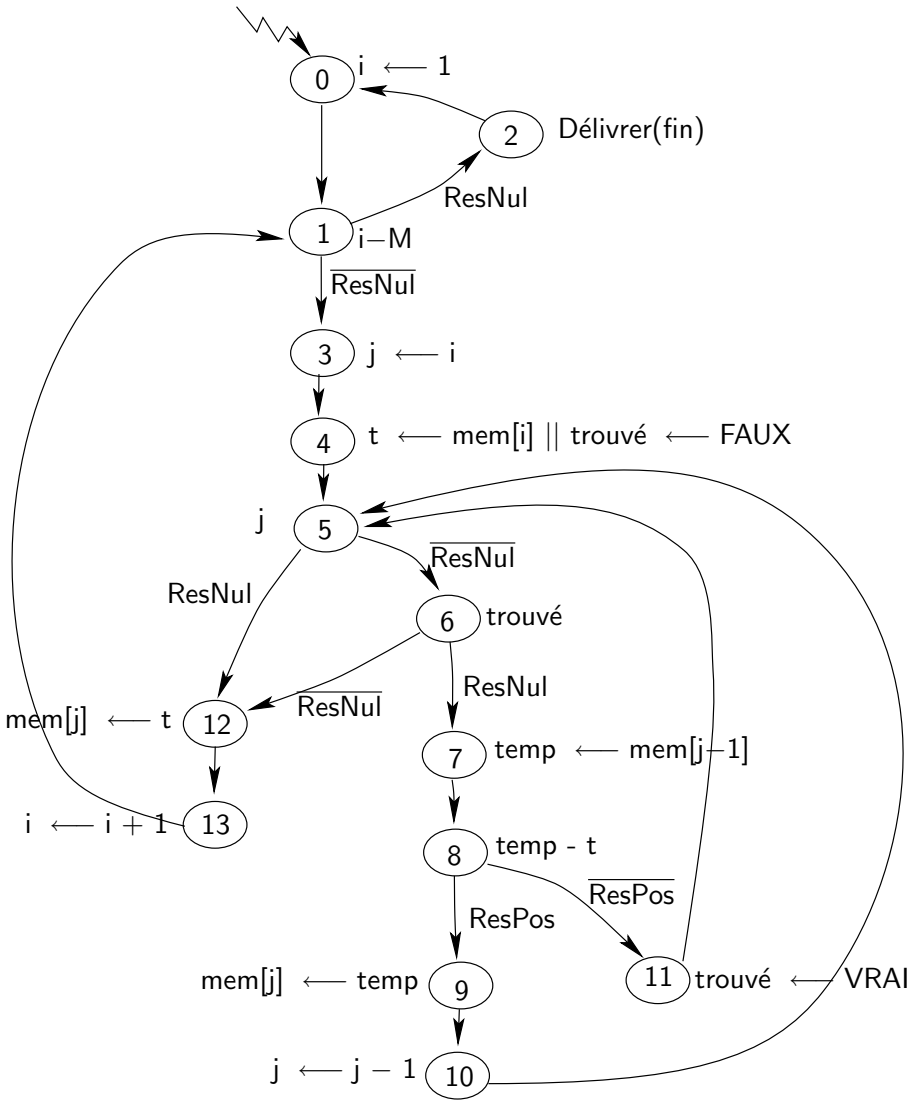


FIG. 11.16 – Machine séquentielle avec actions de la machine à trier

L'initialisation de trouvé à FAUX peut être faite dans le même état que l'affectation t ⟵ mem[i] en se servant de l'initialisation à 0 du registre trouvé.

Le calcul du prédicat j > 0 et non (trouvé) se fait dans deux états successifs (voir le mécanisme d'éclatement de conditions complexes dans le chapitre 5).

La machine séquentielle avec actions donnée dans la figure 11.16 tient compte de ces remarques.

4.3.2 Obtention de la PO

On part de la PO type décrite dans le paragraphe 2. La figure 11.17 décrit une PO qui permet toutes les actions apparaissant dans les différents états de la machine séquentielle avec actions donnée précédemment.

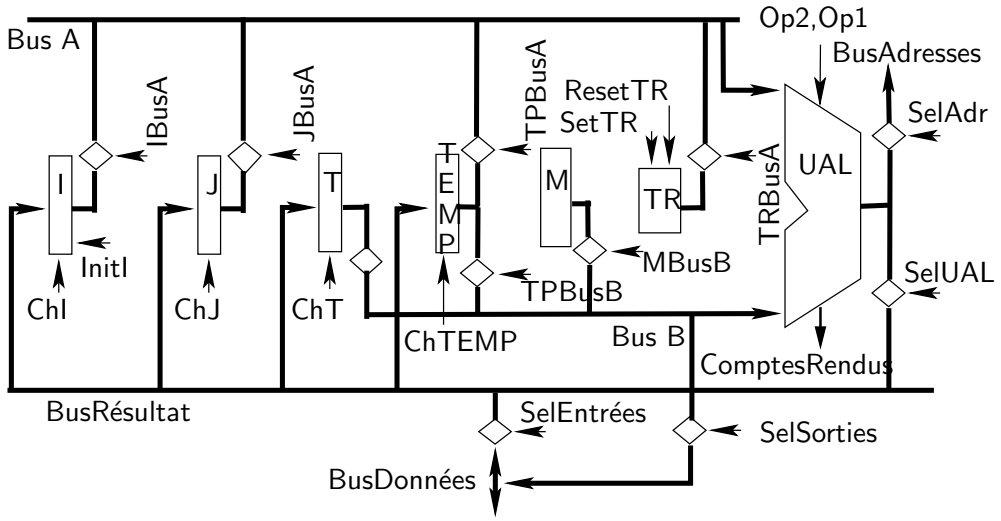


FIG. 11.17 – Partie opérative pour la machine à trier

Il y a 6 registres permettant de stocker les variables de l'algorithme : I, J, T, TR (pour trouvé), M et TEMP (pour la variable intermédiaire introduite). Le registre TR ne comporte qu'une seule bascule. Sa liaison sur n fils au bus A est complétée par des 0.

On se sert d'un opérateur unique à deux opérandes A et B qui doit permettre les calculs : A, A−B, A−1, A+1 et délivrer les comptes-rendus : résultat nul et résultat positif.

Pour réaliser I ⟵ 1, TR ⟵ 0 et TR ⟵ 1 on se sert des commandes d'initialisation des bascules des registres correspondants.

Les entrées/sorties sont effectuées via le bus adresses en sortie et le bus données en entrée et sortie.

On connecte I, J, TR et TEMP au bus A pour réaliser les actions J ⟵ I, J ⟵ J−1, I ⟵ I+1 et les calculs nécessaires à l'évaluation des prédicats I=M, J=0, TEMP>T et non (TR).

On connecte M et T au bus B pour le calcul des prédicats I=M et TEMP>T.

Pour les actions nécessitant un accès mémoire en écriture ou en lecture : T ⟵ mem[I], TEMP ⟵ mem[J−1], mem[J] ⟵ T, mem[J] ⟵ TEMP, on a le choix, soit de connecter le bus A ou B au bus adresses, soit de passer par l'UAL. On choisit la deuxième solution puisqu'il faut effectuer un calcul d'adresses pour l'action TEMP ⟵ mem[J−1].

On connecte le bus données au bus résultat pour les actions comportant une lecture en mémoire : T ⟵ mem[I] et TEMP ⟵ mem[J−1]. Pour les actions comportant une écriture en mémoire : mem[J] ⟵ T et mem[J] ⟵ TEMP, on transmet J sur le bus adresses en passant par le bus A et l'UAL, et on connecte T et TEMP au bus B puis le bus B au bus données.

En résumé, on doit pouvoir avoir sur le bus A le contenu des registres I, J, TR et TEMP. On doit pouvoir avoir sur le bus B le contenu des registres

3	j $\longleftarrow$ i : IBusA = ChJ = 1, (Op2, Op1) = (0, 1), ChTEMP = ChT = ResetTR = SetTR = ϕ
4	t $\longleftarrow$ mem[i] ; trouvé = FAUX : IBusA = ChT = ResetTR = l/$\overline{\text{e}}$ = SelMem = SelAdr = SelEntrees = 1, (Op2, Op1) = (0, 1), ChTEMP = ϕ
5	j : JBusA = 1, (Op2, Op1) = (0, 1)
7	temp $\longleftarrow$ mem[j−1] : JBusA = ChTEMP= l/$\overline{\text{e}}$ = SelMem = SelAdr = SelEntrées = 1, (Op2, Op1) = (1, 1)

FIG. 11.18 – Sorties pour quelques états de l'automate de la figure 11.16 ; sont précisés les signaux à 1 et à ϕ et omis ceux à 0.

M, T et TEMP. On doit pouvoir charger les registres I, J, TEMP et T par la valeur présente sur le bus résultat. On doit pouvoir amener la valeur présente sur le bus données sur le bus résultat. On doit enfin pouvoir amener la valeur présente sur le bus B sur le bus données.

4.3.3 Obtention de l'automate d'états fini de la partie contrôle

Le passage de la machine séquentielle à l'automate d'états fini se fait comme précédemment. On peut utiliser le protocole de poignée de mains pour l'action Délivrer (fin). On remplace l'état correspondant par les deux états de l'émetteur dans la poignée de mains (Cf. Paragraphe 3.1).

Nous donnons dans la figure 11.18, pour quelques états, les sorties de l'automate d'états fini correspondant à la machine séquentielle avec action de la figure 11.16. Nous utilisons les signaux apparaissant sur la PO de la figure 11.17. Pour les écritures et lectures en mémoire les signaux SelMem et l/$\overline{\text{e}}$ à destination de la mémoire sont des sorties supplémentaires de l'automate.

Les configurations 00, 01, 10 et 11 des deux signaux Op2 , Op1 qui commandent l'opération de l'UAL (OperationUAL) correspondent respectivement aux opérations : A + 1, A, A−B, A−1.

4.3.4 Optimisation

On peut simplifier ce circuit en n'utilisant plus de registre pour stocker le booléen trouvé. En effet on peut sortir directement de la boucle quand le test du prédicat mem[j-1] > t donne VRAI (voir la modification sur la machine séquentielle de la figure 11.19).

5. Exercices

E11.1 : La racine carrée
Il s'agit de construire un circuit permettant de calculer la partie entière de la racine carrée d'un entier naturel x. L'algorithme correspondant est donné au paragraphe 3.2 du chapitre 10. En suivant la méthode décrite dans ce chapitre

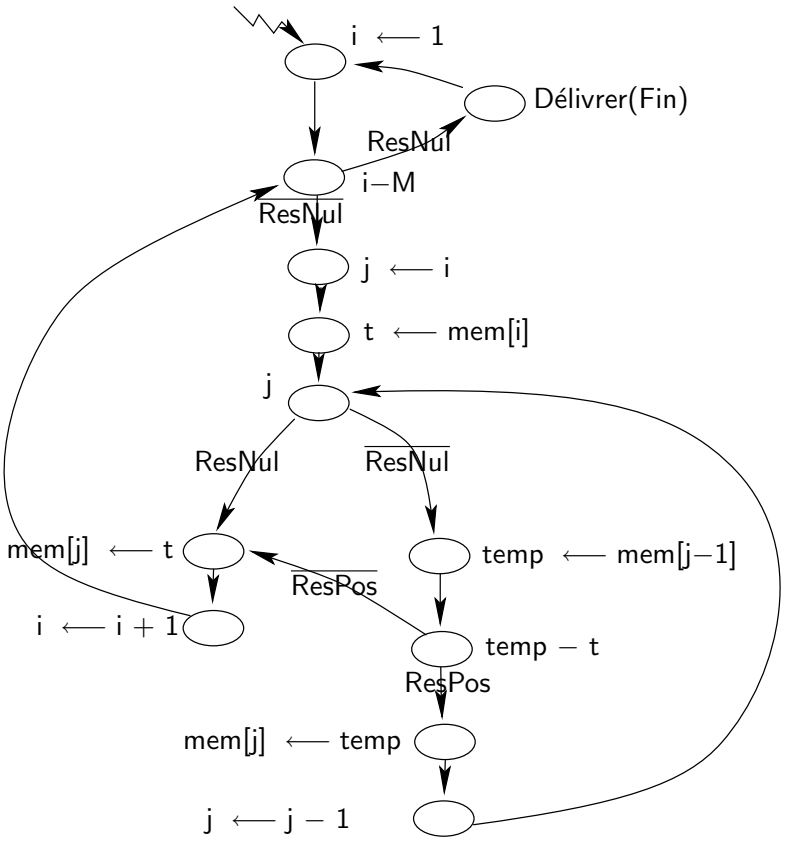


FIG. 11.19 – Machine séquentielle avec actions optimisée de la machine à trier

```
Lexique
   U : un entier > 0 ; fin : un booleen
Algorithme
   Tantque VRAI :
      Acquérir(U)
      tantque U > 1 :
         si U MODULO 2 = 0 alors U ⟵ U DIV 2
         sinon U ⟵ 3 * U + 1
         Délivrer (U)
      Délivrer(fin)
```

FIG. 11.20 – Algorithme de la suite de Syracuse

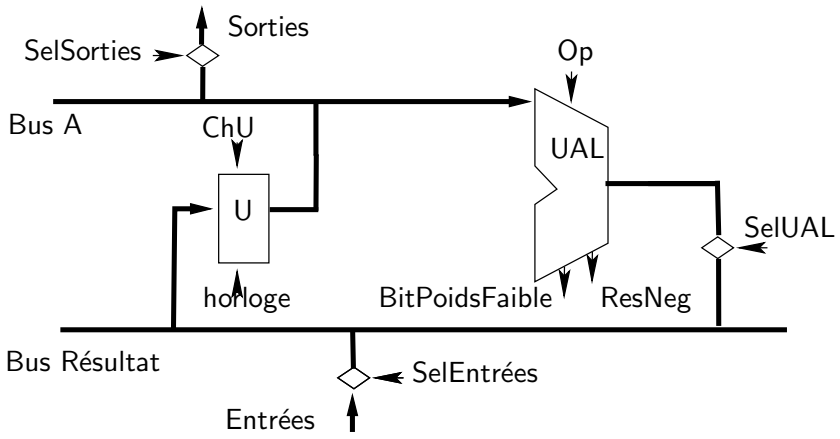


FIG. 11.21 – Une partie opérative pour la suite de Syracuse

donner une PO et une PC permettant de réaliser cet algorithme ; s'inspirer de la PO type donnée à la section 2.

E11.2 : La suite de Syracuse

La figure 11.20 donne l'algorithme correspondant au calcul de la suite de Syracuse. La figure 11.21 représente une PO issue de la PO type du paragraphe 2. permettant de réaliser cet algorithme. Donner la machine séquentielle avec actions correspondant à l'algorithme et utilisant cette PO. Les comptes-rendus de l'UAL sont le bit de poids faible et le signe du résultat. En déduire l'automate d'états fini correspondant. Faire une synthèse cablée de cet automate avec un codage compact. On choisira pour cela un codage pour les trois opérations de l'UAL (Op) : 1−X, X DIV 2 et 3*X+1. On effectuera les entrées/sorties à l'aide du protocole à poignée de mains via les bus Entrées et Sorties pour U.

Troisième partie

Techniques de l'algorithmique logicielle

Chapitre 12

Le langage machine et le langage d'assemblage

Nous avons étudié au chapitre 4 un ensemble de modèles des traitements qu'on peut vouloir faire effectuer par un dispositif informatique. Toutefois, ni les langages de programmation de haut niveau, ni les machines séquentielles, ne sont directement exploitables par un *ordinateur*, tel que nous l'avons défini en introduction.

Le seul langage compréhensible par un ordinateur est le *langage machine* de son processeur. Un programme en langage machine est une suite finie de bits, que le processeur *interprète*. Nous étudions au chapitre 14 les circuits séquentiels qui réalisent cette tâche d'interprétation. Un langage machine, bien que difficilement lisible par un être humain, possède une structure : il est possible d'y voir une suite de paquets de bits, chacun codant une *instruction* du processeur, c'est-à-dire une opération élémentaire réalisable par le processeur (par exemple additionner deux entiers codés sur un octet).

Concevoir un langage machine est une tâche indissociable de la conception du processeur. On doit choisir un ensemble d'instructions (on dit aussi : *jeu d'instructions*) de manière à fournir un modèle de calcul universel, mais ce n'est pas la seule contrainte. On peut avoir à assurer la compatibilité ascendante dans une famille de processeurs, c'est-à-dire à assurer que les programmes en langage machine qui fonctionnaient sur un processeur de la génération n sont *réutilisables* sur le processeur de la génération $n + 1$.

Pour définir complètement un langage machine, il faut ensuite choisir un *codage* de l'ensemble des instructions sur un certain nombre de bits.

Si l'on suppose donnés le processeur et son langage machine, on peut s'intéresser au problème de la programmation de ce dispositif par un être humain, qui ne saurait s'exprimer directement en termes de séquences de bits. Le *langage d'assemblage* est un langage textuel bâti sur le modèle du langage machine. Il a la même structure, c'est-à-dire les mêmes instructions, mais il se note par du texte.

Traduire le langage d'assemblage en langage machine suppose : une analyse

lexicale et syntaxique du texte, pour y repérer les structures ; la vérification des contraintes d'utilisation des opérateurs et opérandes ; un codage des structures en séquences de bits. Nous étudions les principes de cette traduction dans le présent chapitre. C'est par ailleurs la tâche principale de l'outil appelé *assembleur* des environnements de programmation. Nous verrons également au chapitre 18 les aspects de compilation séparée ou d'abstraction vis-à-vis de la position en mémoire à l'exécution. Nous verrons au chapitre 13 que le langage d'assemblage est lui-même une cible pour la traduction des langages de haut niveau.

Dans le paragraphe 1. nous exposons la démarche de conception d'un langage machine : choix du jeu d'instructions et codage, en prenant quelques exemples parmi les processeurs existants. Le paragraphe 2. décrit les caractéristiques d'un langage d'assemblage construit sur le langage machine. Les problèmes de la traduction du langage d'assemblage vers le langage machine correspondants sont étudiés au paragraphe 3. Enfin nous donnons au paragraphe 4. un exemple de programme, sous 5 formes : un algorithme du langage d'actions décrit au chapitre 4 ; un texte écrit dans un langage d'assemblage pour un processeur 68000 ; un texte d'un langage d'assemblage pour processeur SPARC *; un programme en langage machine 68000 ; un programme en langage machine* SPARC.

1. Le langage machine

1.1 Description générique de la machine

Pour définir la notion de langage machine, et proposer des critères de choix d'un ensemble d'instructions qui constitue un modèle de calcul universel, il faut tout d'abord définir précisément la *machine*.

Nous nous restreignons ici au modèle d'architecture de Von Neumann [BGN63] : une machine comporte une *unité de calcul* (qui réalise des opérations de base sur les entiers codés en binaire) et une *mémoire* qui contient des opérandes et des codes d'opérations (les instructions). Les instructions sont exécutées dans l'ordre où elles sont rangées en mémoire.

Nous distinguons par la suite :

– Une mémoire de grande taille, dont les éléments sont désignés par des numéros qu'on appelle des *adresses*. C'est la mémoire que nous avons manipulée en en donnant une abstraction par le tableau MEM au chapitre 4. Nous parlerons souvent de *mémoire principale*

– Une mémoire de plus petite taille dont les éléments, appelés *registres*, sont désignés par des noms ou par des numéros courts. Sur certaines machines on distingue des registres *données* et des registres *adresses*, avec des instructions travaillant sur l'une des catégories de registres seulement. Sur d'autres

machines, au contraire, les registres sont banalisés. Les machines les plus anciennes ne comportaient qu'un seul registre, dit *accumulateur*

> *Nous détaillons au chapitre 14 l'influence de ce genre de distinction sur ce qu'on appelle la* partie opérative *du processeur, c'est-à-dire la structure du circuit qui connecte l'unité de calcul aux bus et aux divers registres.*

La distinction entre ces deux types de mémoire permet de tenir compte, dans le choix d'un jeu d'instructions, de critères tels que le temps d'accès à la mémoire : les registres sont accessibles plus rapidement que la grande mémoire. Nous verrons au chapitre 15 que les registres sont en général situés physiquement dans le processeur, contrairement à la grande mémoire. D'autre part les adresses des octets dans la grande mémoire sont des entiers assez longs (typiquement 32 bits), qui soit ne peuvent pas apparaître tels quels dans le codage d'une instruction (Cf. Paragraphe 1.3), soit en ralentissemnt l'exécution ; en revanche, les adresses ou numéros des registres sont de petits entiers — 5 bits sur le processeur SPARC[1] par exemple — et peuvent donc apparaître comme désignation d'opérande dans le codage d'une instruction.

1.2 Eléments et structure du langage machine

Un programme écrit en langage machine est une suite finie de codages binaires d'*instructions* éventuellement paramétrées.

1.2.1 Instructions et codage

Une instruction élémentaire peut être par exemple un transfert mémoire vers mémoire, dont l'effet peut être décrit par une phrase du langage d'actions présenté au chapitre 4 : $\mathsf{MEM[a]} \longleftarrow_4 \mathsf{MEM[b]}$. Dans ce cas le codage complet de l'instruction comprend : les représentations en binaire pur des adresses a et b, codées sur un certain nombre de bits fixé ; le codage du type d'opération effectué : transfert mémoire vers mémoire de taille 4 octets. En langage machine 68000, cette instruction est codée sur $16 + 32 + 32$ bits. Les 16 premiers bits codent le type d'opération effectué, la taille du transfert et le fait que l'on doit trouver ensuite deux adresses de 32 bits ; les 32+32 bits suivants codent les deux adresses d'opérandes (voir paragraphe 1.3 pour un exposé des principes de codage des instructions).

L'ajout d'une constante à une case mémoire ($\mathsf{MEM[a]} \longleftarrow \mathsf{MEM[a]} + \mathsf{k}$) peut également constituer une instruction élémentaire dans un langage machine. Dans ce cas le codage complet de l'instruction comporte : la représentation en binaire pur de l'adresse a ; le codage du type d'opération effectué : incrémentation de case mémoire désignée par son adresse ; le codage binaire de la constante k.

[1] Dans ce chapitre, SPARC désigne le SPARC V8 qui a des mots de 32 bits, et non le SPARC V9, qui a des mots de 64 bits.

1.2.2 Notion de compteur programme

Sauf indication contraire, une suite d'instructions est lue *séquentiellement* par le processeur, qui *interprète* chacune d'entre elles, et passe à celle qui se trouve en mémoire à l'adresse suivante. Le processeur gère donc ce qu'on appelle le *compteur programme*, qui donne le numéro de la prochaine instruction à lire. Le compteur programme est incrémenté à chaque instruction.

Comme conséquence de l'interprétation d'une instruction, le processeur peut modifier le contenu de la mémoire ou d'un registre, ou commander une opération de l'unité de calcul. Il peut également modifier le numéro (l'adresse) de la prochaine instruction à lire. On parle alors de *rupture de séquence*, ou *branchement*. Une instruction de rupture de séquence doit comporter l'adresse de destination. Une rupture de séquence peut être inconditionnelle, auquel cas l'interprétation de l'instruction produit toujours le même effet ; elle peut être conditionnée, par exemple, par le signe du contenu d'une case mémoire, interprété comme un entier relatif ou comme le résultat d'un calcul. Dans ce cas la rupture effective de séquence, lors d'une exécution, dépend de l'état des données et donc de la mémoire à ce moment-là.

1.2.3 Désignation d'adresse et modes d'adressage

Nous traitons dans ce paragraphe de tout ce qui concerne les mécanismes de désignation d'emplacements en mémoire (mémoire principale ou registres), utilisables dans toute instruction de calcul si elle a des opérandes en mémoire, dans les instructions de transfert depuis ou vers la mémoire, et dans les instructions de rupture de séquence. Ces mécanismes sont appelés des *modes d'adressage*. La figure 12.1 en donne une vue synthétique.

Considérons tout d'abord le cas des instructions de calcul. Lorsqu'elles comportent un ou plusieurs opérandes en mémoire, le ou les paramètres correspondants de l'instruction doivent désigner une adresse d'octet en mémoire. Ce peut être fait en donnant directement dans l'instruction une *constante* entière positive qui est l'adresse voulue (c'est le cas dans l'exemple $\mathsf{MEM[a]} \longleftarrow_4 \mathsf{MEM[b]}$ évoqué ci-dessus). On parle alors d'adressage *absolu*. Ce peut être fait également en donnant comme paramètre un numéro de registre, dont le contenu, lors de l'exécution, donnera l'adresse mémoire ; il s'agit dans ce cas d'adressage *indirect* par registre.

Il en existe de nombreuses variantes : indirect par registre avec déplacement ; indirect par registre prédécrémenté (ou postincrémenté, ou préincrémenté, ou post-décrémenté) avec ou sans déplacement, etc. Le même raisonnement est valable pour les instructions de transfert depuis ou vers la mémoire.

Dans le cas des instructions de rupture de séquence, il faut indiquer d'une manière ou d'une autre à quelle adresse se trouve la prochaine instruction à exécuter, puisque ce n'est pas l'instruction suivante. Il existe deux sortes de branchements : les branchements *relatifs* à la position courante du compteur

programme — l'instruction spécifie alors un *déplacement* en avant ou en arrière par rapport au compteur programme — et les branchements *absolus* — l'instruction spécifie une nouvelle valeur du compteur programme, qui doit écraser l'ancienne. L'instruction doit donc spécifier un déplacement ou une adresse absolue. Dans les deux cas, on peut imaginer que cette donnée est fournie directement comme une constante dans l'instruction, ou indirectement dans un registre.

Le tableau 12.1 résume les diverses manières d'indiquer une adresse mémoire dans une instruction. Pour nommer les modes d'adressages, nous avons utilisé les termes les plus couramment employés. Le mode *indirect par registre avec index* est un adressage indirect par registre avec déplacement, le déplacement étant stocké dans un registre au lieu d'être donné par une constante codée dans l'instruction.

Noter que pour une même instruction, plusieurs modes d'adressage peuvent être possibles. Voir aussi le paragraphe 1.3 qui traite du codage des instructions, pour comprendre comment distinguer les différents cas.

1.2.4 Modèle de calcul universel, modèle de Von Neumann et langage machine type

La première contrainte à respecter dans la conception du jeu d'instructions d'un processeur est d'assurer un modèle de calcul universel. La compréhension complète de cette contrainte demande quelques connaissances en calculabilité, mais il est facile de se convaincre à l'aide d'exemples extrêmes, comme le codage des boucles, que le problème se pose effectivement.

Il est en effet assez facile de se convaincre qu'un langage de haut niveau qui ne disposerait pas de structures itératives, sous quelque forme que ce soit, ne permettrait pas de coder tout algorithme. Or les structures itératives correspondent aux ruptures de séquence dans le langage machine. Un langage machine qui ne permettrait pas ainsi de modifier l'adresse de la prochaine instruction à lire, de manière conditionnelle, serait donc incomplet.

La contrainte de fournir un modèle de calcul universel est réalisable avec un langage machine à une seule instruction complexe, du genre : accès mémoire en lecture avec adressage indirect, conditionnel. Programmer directement dans un tel langage serait déjà très difficile, et écrire un compilateur de langage de haut niveau — c'est-à-dire un algorithme capable de traduire tout texte de programme de haut niveau en suite de telles instructions — serait encore plus ardu.

Il ne suffit donc pas que le jeu d'opérations offertes garantisse un modèle de calcul universel. Le modèle de machine dit de Von Neumann repose sur les classes d'instructions suivantes : transfert de données (chargement depuis un registre, chargement depuis la mémoire, stockage dans la mémoire) ; branchements inconditionnels et conditionnels ; opérations arithmétiques et logiques. Le paragraphe 1.4 présente ces grandes classes d'instructions plus des instruc-

nom usuel	Information contenue dans le codage de l'instruction	valeur à utiliser (pour opérandes sources de calcul et sources de transferts)	adresse effective (pour résultats et cibles de transferts)	effet éventuel sur les opérandes
Pour les instructions de calcul et les transferts mémoire				
immédiat	une constante relative k	k	-	-
absolu	une constante naturelle k	MEM[k]	k	-
registre direct	un numéro n	contenu de Reg_n	Reg_n	-
indirect par registre	un numéro n	MEM[contenu de Reg_n]	contenu de Reg_n	-
indirect par registre avec index	deux numéros n et d	MEM[contenu de Reg_n + contenu de Reg_d]	contenu de Reg_n + contenu de Reg_d	-
indirect par registre avec déplacement	un numéro n, une constante relative d	MEM[contenu de $\text{Reg}_n + d$]	contenu de $\text{Reg}_n + d$	-
indirect par registre prédécrémenté	un numéro n	MEM[contenu de Reg_n - t]	contenu de Reg_n - t	$\text{Reg}_n \longleftarrow \text{Reg}_n$ - t
indirect par registre postdécrémenté	un numéro n	MEM[contenu de Reg_n]	contenu de Reg_n	$\text{Reg}_n \longleftarrow \text{Reg}_n$ - t
Pour les instructions de rupture de séquence				
relatif au compteur programme	une constante relative d	-	PC + d × N	-
absolu	une constante naturelle k	-	k	-

FIG. 12.1 – Modes d'adressage usuels. Noter que, pour les adressages indirects par registre avec pré (ou post) incrémentation (ou décrémentation), l'effet sur le registre d'indirection est de la forme : $\text{Reg}_n \longleftarrow \text{Reg}_n$ op t, où op est l'opération $+$ ou l'opération $-$, et t ne vaut pas nécessairement 1. t dépend de la taille de l'opération, c'est-à-dire de la taille des opérandes. Par exemple, si l'instruction travaille sur des mots de 32 bits, $t = 4$. Pour les adressages relatifs au compteur programme PC, la constante d donnée dans l'instruction peut éventuellement être multipliée par une constante N. Voir un exemple paragraphe 1.4.3.

tions spécifiques comme l'on en trouve dans toute machine.

On trouve parfois le terme d'architecture de Von Neumann, où la mémoire contient à la fois les données et les instructions, par opposition à l'architecture de Harvard, où deux mémoires spécifiques contiennent, l'une les données, l'autre les instructions. Les deux architectures ont la même puissance d'expression.

1.3 Codage des instructions

Nous avons donné au paragraphe 1.2.1 des exemples d'instructions, en indiquant brièvement quelles sont les informations à coder. Nous précisons ci-dessous l'ensemble des informations à coder, avant d'étudier les contraintes qui portent sur la structure du code. Le codage complet d'une instruction est obtenu par juxtaposition des codages des différentes informations qui la définissent ; on dit que le codage d'une instruction est structuré en *champs* (Cf. Figure 12.2) ; c'est aussi le terme utilisé pour les noms des différentes informations qui composent un type construit n-uplet (Cf. Chapitre 4).

1.3.1 Codage de la nature de l'opération

L'opération effectuée est prise parmi toutes les opérations de l'unité de calcul, les transferts entre mémoire et registres, les branchements, les instructions spécifiques éventuelles (Cf. Paragraphe 1.4 pour une liste détaillée d'instructions).

Si le jeu d'instructions comporte n instructions, le codage compact de la nature de l'opération demande $b = \lceil log_2 n \rceil$ bits (l'entier immédiatement supérieur à $log_2 n$). Il n'y a aucune raison pour que le nombre d'instructions d'une machine soit exactement une puissance de 2, et il existe donc toujours au moins une configuration d'un vecteur de b booléens qui ne correspond à aucune instruction. Cela justifie le cas d'erreur dans l'algorithme d'interprétation du langage machine du paragraphe 1.6.2 ci-dessous, et constitue l'une des causes d'*interruption logicielle* étudiées dans la partie VI.

On suppose que la *nature* d'une instruction comporte implicitement l'information sur le nombre d'opérandes. Même si l'on imagine un langage machine offrant par exemple une addition binaire et une addition ternaire, on considère que ce sont deux instructions différentes à compter dans les n instructions, et à coder globalement. Cela donne un codage plus compact que de séparer le codage de la nature de l'instruction et le codage d'un entier donnant le nombre d'opérandes (qui pour une grande majorité des opérations est toujours le même). De manière générale, le choix de la structure des informations qui constituent une instruction, c'est-à-dire le choix des champs, a une influence sur la compacité du codage.

1.3.2 Codage des modes d'adressage des paramètres

Si le langage machine est tel qu'une même opération accepte la désignation d'opérandes avec différents modes d'adressage, il faut coder le mode d'adressage de chacun des opérandes. Ce peut être réduit à 1 bit, comme dans le cas des opérations arithmétiques du SPARC : une instruction d'addition, par exemple, a toujours 3 paramètres, les emplacements des deux opérandes et l'emplacement du résultat. Le résultat et l'un des opérandes sont forcément dans des registres. Le deuxième opérande peut être dans un registre ou être une valeur immédiate. Pour distinguer ces deux cas, le codage du mode d'adressage de ce deuxième opérande comporte 1 bit. Dans le cas du 68000, en revanche, toutes les opérations de calcul acceptent des opérandes dans des registres, ou en mémoire avec des modes d'adressage variés. Le codage complet du mode d'adressage de chaque opérande comporte 3 bits.

1.3.3 Codage des informations mises en jeu dans le mode d'adressage

La nature de l'instruction, plus le mode d'adressage des paramètres, détermine entièrement quelle est la taille du reste du code (opérandes), et comment il faut l'interpréter. En reprenant la deuxième colonne du tableau 12.1, on obtient les différentes informations à coder : des entiers naturels (qui peuvent représenter des adresses mémoire absolues), des entiers relatifs (qui peuvent représenter des déplacements de branchements ou des opérandes d'opérations arithmétiques), des numéros de registres.

En général le nombre de registres est une puissance de 2, ce qui permet un codage compact et sans trous d'un numéro de registre. Sur le SPARC, les 32 registres sont codés sur 5 bits. Toute configuration d'un vecteur de 5 booléens correspond à un numéro de registre existant.

Les entiers sont codés (en binaire pur ou complément à 2) sur une taille prédéfinie (c'est toujours la même, elle est donc implicite et non codée). Le jeu d'instructions 68000 distingue une addition générale et une addition dite *rapide*, selon que l'un des opérandes est un entier long (32 bits) ou un entier court tenant sur un octet. Dans ce cas, la taille de l'entier paramètre est en fait codée dans la nature de l'instruction : il y a deux additions.

1.3.4 Structure du code

La figure 12.2 donne des exemples de structures de code. Le code d'une instruction complète comporte le codage de la nature de l'opération, le codage des modes d'adressage des opérandes (si nécessaire), le codage des informations effectives des opérandes (entiers, adresses, numéros de registres). Ces différents codages doivent être juxtaposés pour former le codage complet de l'instruction. Sur certaines familles de machines, les codes des différentes instructions peuvent avoir des tailles différentes. Lorsque la taille est variable, elle

est entièrement déterminée par la nature de l'instruction et les modes d'adressage des opérandes. Ces deux informations doivent être codées dans les premiers mots du code complet de l'instruction, puisqu'ils seront lus en premier. En 68000 par exemple, le premier mot de 16 bits contient ces deux informations, plus les informations relatives à l'un des opérandes, si elles tiennent dans la place restante.

Le choix du codage précis est guidé par des considérations matérielles, qui portent sur la structure du circuit qui constitue l'interface entre la partie opérative et la partie contrôle du processeur (Cf. Chapitre 14). Par exemple les portions de code qui correspondent à des numéros de registres sont situées au même endroit quelle que soit l'instruction, de manière à simplifier la partie opérative du processeur : les fils de sélection de l'ensemble des bascules qui réalise les registres sont toujours branchés sur les mêmes bits du registre instruction.

Cette contrainte, plus le fait que les valeurs immédiates sont nécessairement contiguës (c'est plus pratique pour la génération de langage machine depuis le langage d'assemblage, et cela évite des chevauchements de nappes de fils dans le circuit interprète), peut amener à couper le codage de la nature de l'instruction. Par exemple, sur le SPARC, la nature de l'instruction est codée sur les bits 31, 30, et 19 à 24. Entre ces deux champs on trouve 5 bits pour le numéro de registre destination.

1.4 Classes d'instructions usuelles

1.4.1 Instructions de calcul

En langage machine il n'y a pas de typage des données, mais les types sont implicites dans les opérations offertes. Certaines instructions ne sont utilisées que si le programmeur interprète la correspondance entre vecteurs de booléens et entiers selon un certain code (binaire pur, complément à 2, virgule flottante, décimal codé binaire, pixels...). Il existe des opérations qui ont un sens si on interprète les vecteurs de bits comme le codage d'entiers (`ADD`, *branchements sur codes de conditions entières*, voir ci-dessous), et il existe des opérations qui les interprètent comme des vecteurs de booléens (`AND`). En revanche il n'en existe pas qui les interprètent comme le code ASCII d'un caractère ; ce serait le cas s'il existait une instruction spécifique du langage machine capable de transformer un vecteur de bits représentant le caractère `'a'` en vecteur de bits représentant le caractère `'A'` ; cette opération est évidemment réalisable sur tout processeur, mais en passant par le codage des caractères par des entiers ou des vecteurs de booléens. Les seuls types sont donc les entiers et les vecteurs de booléens ; l'extension MMX [Int97] et VIS offrent de plus des opérations qui interprètent les vecteurs de 32 bits comme 4 sous-vecteurs de 8 bits.

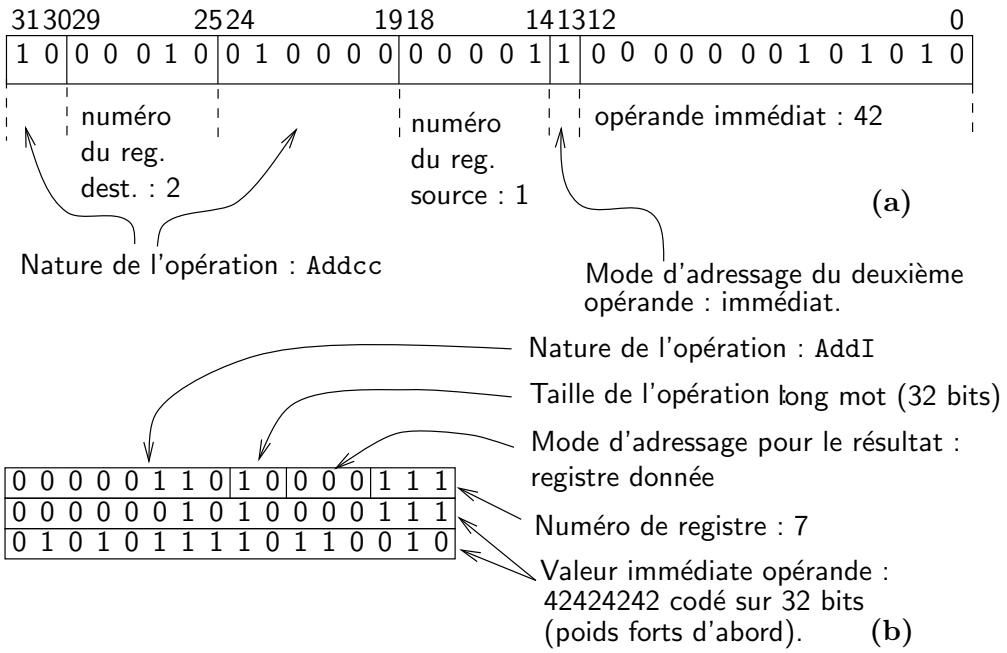


FIG. 12.2 – Structure du codage d'une instruction :
(a) codage de l'instruction `addcc %r1, 42, %r2` du SPARC ;
(b) codage de l'instruction `addi ♯42424242, D7` du 68000 (noter que $42424242_{10} = 028757B2_{16}$).

Opérations sur les entiers Tous les processeurs offrent les opérations d'addition et de soustraction d'entiers. La même instruction est utilisable pour les opérations sur les naturels codés en binaire bur et sur les entiers codés en complément à deux, grâce aux bonnes propriétés du codage en complément à deux (Cf. Chapitre 3). La différence d'interprétation du codage des entiers apparaît dans les instructions de branchement conditionnels (paragraphe 1.4.3).

Certains processeurs offrent également la multiplication et la division entière ; mais ce sont des instructions coûteuses en temps ou en surface du circuit interprète. Le SPARC offre une instruction `mulscc` effectivement exécutable en temps égal à celui d'une addition, mais cette instruction ne constitue qu'un pas de la multiplication 32 bits (la version 9 offre une vraie multiplication). Pour réaliser la multiplication de deux entiers 32 bits, il faut écrire 32 instructions `mulscc` en séquence. Le 68000 offre deux instructions `muls` et `mulu` de multiplication de deux entiers signés ou non de 16 bits, dont le résultat est sur 32 bits ; il offre de même deux instructions `divs` et `divu`.

Lorsque la multiplication et la division générale n'existent pas, on trouve toutefois les instructions de décalage arithmétique, qui permettent la division et la multiplication par des puissances de 2. (Cf. Chapitre 3, paragraphe 2.2.3).

Opérations sur les vecteurs de booléens Les opérations sur les vecteurs de booléens sont les extensions bit à bit des opérateurs booléens usuels et, ou, non, nand, etc. Elles n'existent pas nécessairement toutes. Par exemple le SPARC offre `AND` et `ANDN`, `OR` et `ORN`, `XOR` et `XORN`, mais pas de `NOT`. `ANDN` (resp. `ORN`) calcule la conjonction (resp. la disjonction), bit à bit, du premier opérande et de la négation bit à bit du second. Les propriétés de l'algèbre de Boole permettent de fabriquer les opérateurs manquants en termes des opérateurs disponibles.

Pour utiliser les opérations sur les vecteurs de booléens dans la compilation des opérations booléennes des langages de haut niveau, il faut inventer un bon codage des booléens du langage de haut niveau (Cf. Chapitre 4, paragraphe 2.2.1 et chapitre 13, paragraphe 1.2).

Opérations structurelles Les opérations structurelles manipulent les vecteurs de bits sans interprétation particulière de leur signification. Ce sont les décalages logiques, à droite ou à gauche. Un décalage à droite, combiné avec la conjonction booléenne bit à bit utilisée comme masquage, permet d'examiner individuellement tous les bits d'un vecteur de bits, ou d'extraire des sous-champs de longueur quelconque.

Par exemple, l'algorithme suivant permet de calculer le nombre de 1 dans un vecteur de bits. Il utilise une opération ET_bit_à_bit avec l'opérande 1 (c'est-à-dire le vecteur de bits qui n'a qu'un 1 en poids faible) pour tester le bit de poids faible de V. A chaque étape, le vecteur V est décalé d'un bit vers la droite.

```
lexique :
    V : un vecteur de bits ; Nb : un entier ≥ 0
algorithme
    Nb ⟵ 0
    tantque V ≠ 0
        si (V ET_bit_à_bit 1 = 1) alors Nb ⟵ Nb+1
        décaler V d'un bit vers la droite
```

Nombre d'opérandes Toutes les opérations examinées ci-dessus sont binaires : elles ont deux opérandes et un résultat. On pourrait donc penser que les instructions du langage machine doivent nécessairement désigner 3 emplacements mémoire ou registres. C'est le cas pour les machines dites *à 3 références* comme le SPARC, où l'on écrit par exemple `ADD g1, g2, g3` pour réaliser l'affectation `g3` ⟵ `g1+g2`. Une *référence* est la désignation d'un opérande ou du résultat, que ce soit un numéro de registre ou une adresse en mémoire. Mais il est possible de réduire le nombre de références, et l'on obtient ainsi des machines dites à 0, 1 ou 2 références (voir exercice E12.1).

Le 68000 est une machine à deux références. Les opérations sont de la forme `dest` ⟵ `source op dest`, et l'instruction contient donc la désignation de 2 emplacements seulement. L'un est utilisé à la fois en lecture et en écriture.

Si l'on veut maintenant ne désigner qu'un emplacement, il faut que le deuxième soit implicite. C'est le cas si l'on introduit un registre dit *accumulateur* noté `Acc`. Les instructions sont alors de la forme : `Acc` ⟵ `Acc + source`. Le registre accumulateur doit être chargé auparavant par un transfert mémoire-registre ou registre-registre.

Finalement, on peut imaginer une machine à 0 référence, où les positions des deux opérandes et du résultat sont implicites. On obtient, par exemple, une machine à pile. Toute opération dépile les deux valeurs de sommet de pile comme opérandes, et empile le résultat de l'opération. Une instruction est donc de la forme : `PILE[Sp+1]` ⟵ `PILE[Sp] + Pile[Sp+1] ; Sp` ⟵ `Sp+1`, si `Sp` pointe sur le dernier emplacement occupé et si la pile progresse en diminuant `Sp`.

1.4.2 Instructions de transfert entre mémoire et registres

Pour réaliser un transfert de ou vers la mémoire, il faut pouvoir désigner un emplacement en mémoire, c'est-à-dire fournir l'adresse de son premier octet, et sa taille en nombre d'octets.

Taille de transferts La taille n'est pas à proprement parler un *opérande* de l'instruction. Les tailles des transferts disponibles sont le plus souvent des *constantes*, en petit nombre. En général les tailles prévues vont de 1 octet (la plus petite unité adressable sur une machine) jusqu'à la taille du plus long mot manipulable par le jeu d'instructions, qui est aussi la taille des registres de données (Cf. Chapitres 9 et 15). Sur SPARC on trouve ainsi des transferts de 1, 2, ou 4 octets ; on trouve aussi un transfert *double* de 8 octets, qui concerne deux registres de numéros consécutifs.

Noter toutefois qu'il existe sur le VAX une instruction de copie de blocs mémoire, dont la taille est un vrai paramètre, donné par exemple par le contenu d'un registre, à l'exécution. Un telle instruction a un effet équivalent à celui d'une boucle de transferts de taille fixe. Elle peut avoir un intérêt si le processeur garantit une exécution plus rapide de la forme à instruction unique, par rapport à la forme avec boucle (transferts de taille fixe, comptage et branchements explicites).

Spécification complète des transferts Pour réaliser un transfert de ou vers un (ou plusieurs) registres, il faut désigner un registre (par son numéro) et éventuellement un sous-champ de ce registre.

Prenons l'exemple du jeu d'instructions SPARC. Il y a 32 registres de 32 bits. Les transferts mémoire sont de taille 1, 2, 4 ou 8 octets. Une instruction de transfert mémoire vers registre est de la forme : `LOAD t a r` où `t` est une constante parmi 1, 2, 4, 8 ; `a` est une adresse d'octet (voir tableau 12.1 pour l'obtention de cette adresse) ; `r` est un numéro de registre, dans l'intervalle $[0, 31]$. Pour un transfert de taille 4, l'opération est parfaitement spécifiée, la

source et la destination ayant la même taille. Pour des transferts de taille 1 ou 2, deux questions se posent : puisque la destination (un registre, de 4 octets) est plus grande que le transfert demandé, dans quelle portion du registre doit-on écrire ? Doit-on inscrire une valeur et, si oui, laquelle, dans la portion de registre inutilisée ?

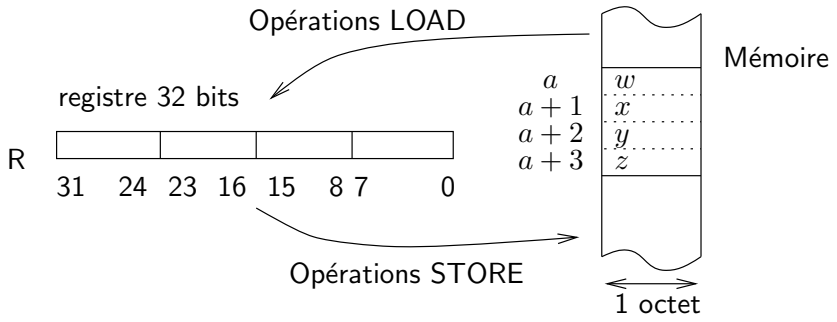
Sur le SPARC, les transferts ont toujours pour destination la portion de poids faible des registres. La portion de poids fort est complétée, soit par des zéros (on parle alors de transfert *non signé*), soit par une recopie du bit de poids fort de la portion transférée (on parle alors de transfert *signé*). Cette opération rappelle bien évidemment l'opération d'extension de format, tenant compte du codage en binaire pur ou complément à deux, que nous avons étudiée au chapitre 3, paragraphe 3.2.1.

Sur le 68000, la portion d'un registre qui n'est pas destination du transfert est laissée intacte.

Sur le SPARC, les instructions de transfert mémoire vers registres, de taille inférieure à 4, existent donc en deux versions : la version dite *signée*, et la version *non signée*. Ces différentes conventions sont illustrées et détaillées dans la figure 12.3 (cas du `LOAD` de SPARC). Les noms des instructions sont obtenus en combinant `LD` (pour `LOAD`) ou `ST` (pour `STORE`), le caractère signé ou non du transfert (`U` pour *unsigned*, rien pour *signed*), et un caractère spécifiant la taille : `B` pour *byte*, c'est-à-dire octet, `H` pour *half*, rien pour la taille d'un mot, `D` pour *double*. Dans le cas des transferts doubles, 2 registres `R` et `R'` interviennent. L'instruction n'en désigne qu'un ; le deuxième est implicite, c'est le registre de numéro suivant. De plus, le registre indiqué doit avoir un numéro pair. On retrouve là, pour l'ensemble des registres, l'équivalent des contraintes d'alignement en mémoire.

Lors d'un transfert registre vers mémoire (cas des `STORE` du SPARC, figure 12.3), des questions symétriques se posent. Si la taille du transfert est inférieure à celle du registre source, quelle portion du registre doit-on transférer ? D'autre part, si la taille du transfert est supérieure à l'octet, l'adresse en mémoire doit satisfaire les contraintes de cadrage en mémoire mentionnées au chapitre 4, paragraphe 2.3. Si l'adresse en mémoire est une constante, cette contrainte d'alignement est vérifiable statiquement, c'est-à-dire avant l'exécution, et le programme peut-être rejeté. En général, toutefois, l'adresse en mémoire n'est pas une constante ; c'est par exemple le contenu d'un registre, dans le cas de l'adressage indirect par registre. La contrainte d'alignement est alors vérifiée dynamiquement, c'est-à-dire au moment de l'exécution du programme par le processeur.

> *Nous verrons au chapitre 24 que ces erreurs d'alignement en mémoire lors des transferts font partie des* exceptions *ou* interruptions logicielles *prévues dans un processeur.*



transfert	taille	signé	inst.	effet
MEM → R	4	–	`LD`	R[7..0] ⟵ 1 MEM[a+3] ; R[15..8] ⟵ 1 MEM[a+2] ; R[23..16] ⟵ 1 MEM[a+1] ; R[31..24] ⟵ 1 MEM[a]
MEM → R	2	non	`LDUH`	R[7..0] ⟵ 1 MEM[a+1] R[15..8] ⟵ 1 MEM[a] ; R[31..16] ⟵ 2 ext16(0)
MEM → R	2	oui	`LDH`	R[7..0] ⟵ 1 MEM[a+1] R[15..8] ⟵ 1 MEM[a] ; R[31..16] ⟵ 2 ext16(R[15])
MEM → R	1	non	`LDUB`	R[7..0] ⟵ 1 MEM[a] ; R[31..8] ⟵ 3 ext24(0)
MEM → R	1	oui	`LDB`	R[7..0] ⟵ 1 MEM[a] ; R[31..8] ⟵ 3 ext24(R[7])
MEM → R, R'	8	–	`LDD`	idem `LD` plus : R'[7..0] ⟵ 1 MEM[a+7] ; R'[15..8] ⟵ 1 MEM[a+6] ; R'[23..16] ⟵ 1 MEM[a+5] ; R'[31..24] ⟵ 1 MEM[a+4]
R → MEM	4	–	`ST`	MEM[a] ⟵ 1 R[31..24] ; MEM[a+1] ⟵ 1 R[23..16] ; MEM[a+2] ⟵ 1 R[15..8] ; MEM[a+3] ⟵ 1 R[7..0]
R → MEM	2	–	`STH`	MEM[a] ⟵ 1 R[15..8] ; MEM[a+1] ⟵ 1 R[7..0]
R → MEM	1	–	`STB`	MEM[a] ⟵ 1 R[7..0]
R, R' → MEM	8	–	`STD`	idem `ST` plus : MEM[a+4] ⟵ 1 R'[31..24] ; MEM[a+5] ⟵ 1 R'[23..16] ; MEM[a+6] ⟵ 1 R'[15..8] ; MEM[a+7] ⟵ 1 R'[7..0]

FIG. 12.3 – Transferts registres vers mémoire et mémoire vers registres du SPARC. On note ext16(b) (resp. ext24(b)) le mot de 16 (resp. 24) bits obtenu en copiant 16 fois (resp. 24 fois) le booléen b.

1.4.3 Instructions de rupture de séquence

Pour définir complètement les instructions de rupture de séquence, il faut spécifier, d'une part l'instruction de destination, d'autre part la *condition* de branchement. Les branchements sont dits *relatifs* ou *absolus* selon la manière d'indiquer l'instruction de destination. Ils sont dits *inconditionnels* lorsque la condition est la constante booléenne vrai. Si la condition de branchement a la valeur faux il y a passage en séquence.

Branchements relatifs ou absolus Le branchement peut être *relatif* à la position courante du compteur programme — l'instruction spécifie alors un *déplacement d* en avant ou en arrière par rapport au compteur programme— ou *absolu* — l'instruction spécifie une nouvelle valeur v du compteur programme, qui doit écraser l'ancienne. L'effet d'une instruction de *branchement relatif* sur le compteur programme noté PC est de la forme : $\mathsf{PC} \longleftarrow \mathsf{PC} + d$. L'effet d'une instruction de branchement absolu, au contraire, est de la forme $\mathsf{PC} \longleftarrow v$.

Que l'on indique un déplacement ou une adresse de destination absolue, il faut choisir un mode d'adressage : par une constante immédiate, indirectement par un registre, avec ou sans déplacement, etc.

Noter que la notion de branchement relatif ou absolu est parfaitement indépendante du mode d'adressage direct ou indirect. On peut envisager toutes les combinaisons. Il existe ainsi sur le SPARC une instruction `jmpl` de branchement inconditionnel, absolu, indirect par registre avec déplacement : le codage de l'instruction contient deux numéros de registres $n1$ et $n2$ (ou bien un numéro de registre n et une constante relative d). L'effet sur le compteur programme est de la forme $\mathsf{PC} \longleftarrow \mathsf{Reg}_{n1} + \mathsf{Reg}_{n2}$ (ou bien $\mathsf{PC} \longleftarrow \mathsf{Reg}_n + d$). Il existe aussi une instruction `ba` de branchement inconditionnel, relatif, immédiat.

Donnée du déplacement En cas de branchement relatif, le déplacement est un nombre d'octets, spécifiant l'écart entre la valeur courante de PC et l'adresse de l'instruction de destination. Noter que dans la plupart des machines la valeur courante de PC est déjà sur l'instruction suivante (Cf. Chapitre 14). Le déplacement n'est pas un entier tout à fait quelconque : il est pair si le codage des instructions se fait sur un nombre pair d'octets. Sur le processeur SPARC, ce déplacement est même toujours multiple de 4, car toutes les instructions sont codées sur un format fixe de 4 octets (voir paragraphe 1.3). On peut alors profiter de cette information pour gagner de la place dans le codage des instructions de branchement : au lieu de coder le déplacement exact d, on code $d' = d/4$, ce qui économise 2 bits. L'effet de l'instruction est alors de la forme : $\mathsf{PC} \longleftarrow \mathsf{PC} + d' \times 4$ (voir tableau récapitulatif des modes d'adressages 12.1).

Expression d'une condition de branchement Dans un langage de programmation de haut niveau, les conditions des structures conditionnelles ou itératives sont des expressions booléennes quelconques qui font intervenir des

constantes, des noms de variables du lexique, des appels de fonctions, etc. (Cf. Chapitre 4, paragraphe 1.5).

Dans *une* instruction du langage machine, il paraît difficile de coder une condition *quelconque* faisant intervenir le contenu des registres ou de la mémoire et d'éventuels appels de fonctions.

Une solution consiste à utiliser les instructions de calcul du langage machine pour calculer la valeur booléenne de l'expression qui conditionne un branchement. On obtient ainsi, après un certain nombre d'étapes, *une* valeur booléenne, rangée par exemple dans un registre ou une partie de registre. Le branchement conditionnel peut ensuite être effectué d'après la valeur de ce registre.

On peut donc fabriquer un langage machine suffisant en ajoutant aux instructions de calcul, une unique instruction de branchement conditionnel de la forme `BV` n a. Cette instruction est un branchement *si condition vraie*, par exemple absolu, avec adressage absolu. L'effet sur le compteur programme PC est : si Reg_n = vrai alors PC $\longleftarrow a$ sinon PC $\longleftarrow$ PC+1.

Considérons le programme :

si (A+2*B $<$ 4 et C $\geq$ 0) alors ... sinon ...

On peut toujours le transformer en :

X : un booléen { *une nouvelle variable, non utilisée ailleurs* }

X $\longleftarrow$ A+2*B $<$ 4 et C $\geq$ 0

si X alors ... sinon ...

Cette transformation est aisément généralisable à toutes les structures conditionnelles ou itératives du langage d'actions. Elle permet de comprendre comment produire une séquence d'instructions du langage machine correspondante. Il suffit d'écrire tout d'abord une séquence d'instructions de calcul et/ou de transferts mémoire destinées à placer dans un registre, par exemple Reg_1, la valeur booléenne de la condition (A+2*B $<$ 4 et C $\geq$ 0). Suit immédiatement une instruction `BV 1` a, qui réalise un branchement d'après la valeur de Reg_1. (Pour une explication détaillée du codage des structures conditionnelles et itératives en langage machine, voir chapitre 13, paragraphes 1.3 et 1.4).

En réalité la plupart des processeurs offrent une méthode intermédiaire entre l'unique instruction de branchement conditionnel présentée ici et l'hypothétique instruction universelle contenant le codage d'une condition booléenne quelconque. Ces méthodes sont basées sur l'utilisation des *indicateurs arithmétiques* (ou *flags* en anglais) fournis par le processeur. Dans certains cas elles s'accompagnent de l'utilisation du *mot d'état* du processeur, qui permet de stocker temporairement la valeur de ces indicateurs.

Indicateurs arithmétiques et mot d'état L'idée est simple : lors de toute opération de calcul, l'unité arithmétique et logique du processeur produit des comptes-rendus sous la forme de 4 booléens dits *indicateurs arithmétiques*, qui peuvent être stockés dans une portion de registre interne spécialisé, appelé *mot d'état* du processeur. Noter que sur le SPARC, les instructions arithmétiques

existent en deux exemplaires : une version qui ne touche pas aux indicateurs, et une version qui les met à jour.

Ces 4 indicateurs sont : Z, qui est vrai si le résultat de l'opération est 0 ; C, qui est vrai si l'opération arithmétique a produit une retenue (C pour *Carry*) et qui, si l'on interprète les opérandes et le résultat comme des entiers naturels codés en binaire pur, signifie que le résultat n'est pas codable sur le même nombre de bits que les opérandes ; N, qui est le bit de poids fort du résultat (si ce résultat est interprété comme le codage en complément à 2 d'un entier relatif, si N vaut 1 alors le résultat est négatif) ; V, qui n'a de sens que si l'on interprète les opérandes et le résultat comme des entiers relatifs codés en complément à 2, et qui est vrai si le résultat n'est pas représentable sur le même nombre de bits que les opérandes (V pour *oVerflow*). Reprendre le chapitre 3 pour un exposé détaillé de la signification des divers indicateurs arithmétiques.

Si l'on considère un processeur qui travaille sur des nombres réels représentés en virgule flottante, il faut tenir compte d'autres indicateurs ; il existe pour la représentation en virgule flottante une notion de débordement pour des valeurs trop petites ou trop grandes, non représentables avec la précision disponible.

Expression des conditions de branchement à base d'indicateurs arithmétiques et de mot d'état Considérons le cas où les indicateurs arithmétiques sont stockés dans un registre après l'exécution de chaque opération arithmétique. On introduit alors des opérations de branchement d'après les valeurs de ces indicateurs (même idée que pour le branchement unique `BV` présenté plus haut, mais la condition peut utiliser 4 booléens au lieu d'un seul).

Sur des processeurs 8 bits comme le 6502, il y a 8 branchements, d'après la valeur vrai ou faux des 4 booléens.

Sur la plupart des processeurs actuels, il y a 16 branchements, selon des fonctions booléennes prédéfinies des indicateurs Z, N, C et V, correspondant aux tests de comparaison usuels entre deux entiers naturels ou relatifs. On trouve ainsi un branchement `BLE` (*Branch on Less or Equal*) dont la condition est Z ou (V et non N ou non V et N). Lorsqu'on a effectué une soustraction entre deux entiers A et B, les bits du registre d'état sont tels que cette condition est vraie si et seulement si $A \leq B$, en interprétant A et B comme des entiers relatifs codés en complément à 2, pour faire la comparaison. En effet, Z est vrai quand $A = B$, et la partie V et non N ou non V et N signifie que $A < B$, en tenant compte des cas où la soustraction déborde. Nous donnons les 16 fonctions booléennes usuelles au paragraphe 1.5. L'exercice E12.7 étudie la formule booléenne associée au branchement `BLE`.

Expression des conditions de branchement à base d'indicateurs arithmétiques sans mot d'état Sur certains processeurs, on peut trouver des instructions qui combinent un test et un branchement. Par exemple, le processeur MIPS R10000 fournit une instruction qui combine un test et un branchement conditionnel sur les entiers. La condition est soit l'égalité de deux registres, soit la comparaison d'un registre par rapport à zéro ($= 0$, < 0, > 0, ≤ 0). Considérons le fragment de programme :

A, B : des entiers
si A = B alors ... sinon ...

Le code produit est simple : il n'est pas nécessaire de calculer la valeur de la condition booléenne A = B avant d'effectuer un branchement, si les deux entiers sont dans des registres. On écrit une seule instruction de la forme BrEgal n1 n2 a dont l'effet est :
si $\text{Reg}_{n1} = \text{Reg}_{n2}$ alors PC $\longleftarrow$ a sinon PC $\longleftarrow$ PC+1.

Toutefois, pour coder si A $<$ B, il faut tout d'abord effectuer une soustraction, et en placer le résultat dans un registre explicitement manipulé par le programmeur.

Sauts à des sous-programmes Enfin tous les processeurs fournissent un moyen de transférer le contrôle à un *sous-programme*, avec sauvegarde de l'adresse de départ, pour reprendre le flot normal des instructions quand le sous-programme se termine.

Considérons le programme suivant, dans lequel `JSR` est une intruction de saut à un sous-programme, `RTS` est l'instruction de retour de sous-programme et `Inst-i` dénote une instruction de calcul quelconque (ne provoquant pas de rupture de séquence). `JSR` a comme opérande une étiquette qui désigne le sous-programme ; l'exécution de `JSR` provoque un branchement (donc une rupture de séquence) au sous-programme désigné avec sauvegarde de l'adresse qui suit l'instruction `JSR`. L'instruction `RTS` provoque un retour de sous-programme c'est-à-dire un branchement à l'adresse qui avait été précédemment sauvegardée. En cas de branchements successifs à des sous-programmes, les adresses de retour doivent être gérées en pile.

1	`Inst-1`	SP1	`Inst-5`	SP2	`Inst-7`
2	`JSR SP1`		`Inst-6`		`Inst-8`
3	`Inst-2`		`RTS`		`JSR SP1`
4	`Inst-3`				`Inst-9`
5	`JSR SP2`				`RTS`
6	`Inst-4`				

L'exécution de ce programme en terme des instructions `Inst-i` peut être décrite par la séquence : `Inst-1` (`Inst-5` `Inst-6`) `Inst-2` `Inst-3` (`Inst-7` `Inst-8` (`Inst-5` `Inst-6`) `Inst-9`) `Inst-4` dans laquelle nous avons utilisé une parenthèse ouvrante chaque fois qu'un appel à un sous-programme (instruction `JSR`) est exécuté et une parenthèse fermante lors du retour correspondant

(instruction `RTS`). Lors du premier appel (exécution de `JSR SP1`) l'adresse sauvegardée est 3 ; l'exécution de l'instruction `RTS` effectue le retour à cette adresse.

L'instruction de branchement avec sauvegarde de l'adresse de départ est généralement spécifique : il s'agit d'une instruction de branchement qui n'est pas ordinaire puisqu'il lui faut intégrer une sauvegarde. Or après le saut il est trop tard pour sauvegarder l'adresse d'où l'on vient ; avant le saut il n'est pas toujours très simple de la calculer. Selon les machines l'adresse sauvegardée est l'adresse qui suit le branchement, ou l'adresse du branchement elle-même ; le retour de sous-programme doit être cohérent avec ce choix.

Les instructions de branchement à des sous-programmes peuvent être absolues ou relatives, et l'adressage peut-être direct ou indirect par registre, avec ou sans déplacement. Sur le SPARC on dispose de deux instructions de branchement avec sauvegarde, qui peuvent donc être utilisées pour coder des sauts à des sous-programmes : `call` et `jmpl`. `call` est un branchement relatif à adressage direct, qui sauvegarde sa propre adresse dans un registre, toujours le même. `jmpl` est un branchement absolu à adressage indirect par registre avec déplacement, qui sauvegarde sa propre adresse dans un registre spécifié par l'instruction.

Nous détaillons au chapitre 13 l'utilisation des instructions de saut à des sous-programmes, et la structure de *pile* sous-jacente, pour le codage des actions et fonctions paramétrées des langages impératifs usuels, comme celui présenté au chapitre 4.

1.4.4 Combinaison test/instruction sans branchement

On trouve sur certains processeurs, par exemple MIPS R10000, ULTRASPARC, PentiumPro, des instructions de *transfert de registres conditionnels*. Une telle instruction est conditionnée par le résultat de l'opération précédente et permet d'éviter des branchements explicites. Nous empruntons à [SL96] l'exemple suivant :

A, B : des entiers
A ⟵ min (A, B)

Le code produit est de la forme :

{ *supposons A dans le registre r1 et B dans le registre r2* }
SUB r1, r2, r3 { *c'est-à-dire r3* ⟵ *r1* − *r2* }
MOV_COND_GT r3, r2, r1 { *c'est-à-dire : si r3* > 0 *alors r1* ⟵ *r2* }

La question intéressante avec ce type de langage machine est : comment écrire un algorithme de traduction des langages de haut niveau en langage machine qui profite au mieux de ces instructions sophistiquées ? Les compilateurs pour machines de ce type font appel à des techniques assez élaborées d'optimisation de code.

1.4.5 Instructions spéciales

La plupart des langages machines comportent, outre les instructions usuelles présentées jusque là, des instructions spécifiques imposées par des contraintes d'ordres assez divers.

• Par exemple, on trouve sur certaines machines de la famille x86 des instructions spécifiques `in` et `out` pour le contrôle des périphériques d'entrée/sortie. Nous verrons au chapitre 16 un exemple détaillé dans lequel la commande du coupleur d'entrée/sortie est complètement assimilable à une écriture en mémoire, et peut donc se faire par des instructions de transfert vers la mémoire déjà définies. Du point de vue du concepteur du langage machine, cela implique que les adresses d'accès à la mémoire, telles qu'elles sont produites par le processeur (qui les lit dans le codage des instructions) à destination des boîtiers mémoire (Cf. Chapitre 15) sont ensuite aiguillées vers de la mémoire véritable ou vers un circuit d'entrées/sorties, d'après leur *valeur*; typiquement, un intervalle d'adresses est réservé aux circuits d'entrées/sorties. Si les connexions du processeur à la mémoire et aux périphériques ne vérifient pas cette contrainte, le processeur doit émettre lui-même une indication d'aiguillage, et cela ne peut se faire que si l'instruction elle-même comporte l'information nécessaire. D'où l'existence d'instructions spécifiques `in` et `out`.

• Le processeur SPARC comporte une instruction très spécifique `sethi` (pour *SET High bits*) nécessaire à l'installation d'une valeur immédiate de 32 bits dans un registre. En effet, toutes les instructions SPARC sont codées sur 32 bits exactement, sans mot d'extension. Une valeur immédiate `v` de 32 bits ne peut donc pas tenir dans ce codage ; il n'y a pas d'instruction de la forme `set v, r`, où `v` est une valeur immédiate de 32 bits et `r` un numéro de registre. Comment réaliser une telle opération ? Le jeu d'instructions SPARC propose de réaliser l'affectation d'une constante 32 bits à un registre en deux instructions du langage machine : la première affecte les 22 bits de poids fort du registre ; la deuxième met à jour les 10 bits de poids faible sans modifier les 22 bits de poids fort (une instruction comme `add` ou `or` avec une valeur immédiate sur 13 bits convient). Pour faciliter l'écriture de tels couples d'instructions, le langage d'assemblage offre des macro-notations `%hi` et `%lo` qui permettent d'extraire respectivement les 22 bits de poids fort et les 10 bits de poids faible d'une constante. On écrit ainsi : `sethi %hi(0x0A08CF04), %r1` ; `or %r1, %lo(0x0A08CF04), %r1`. En langage machine, la valeur immédiate contenue dans l'instruction `or` est `0x304` et celle contenue dans l'instruction `sethi` est `0x028433`.

• On trouve également des instructions d'addition `ADDX` et de soustraction `SUBX` qui prennent en compte comme troisième opérande la retenue de l'opération précédente (présente dans le bit `C` du mot d'état). En enchaînant de telles instructions, on réalise l'addition d'entiers codés en complément à 2 ou en binaire pur sur plusieurs mots de 32 bits. L'exercice E12.6 détaille l'utilisation de cette instruction.

• Finalement, la plupart des langages machines comportent des instructions spécialement conçues pour faciliter la traduction des langages de haut niveau. Nous détaillons au chapitre 13 l'exemple des instructions `link` et `unlink` du 68000, ou `save` et `restore` du SPARC.

1.5 Description du langage machine par le lexique d'une machine séquentielle à actions

Pour un algorithme donné, pour comprendre le langage machine, on peut définir une machine séquentielle à actions (Cf. Chapitre 5) dont : 1) les actions sont les instructions de calcul disponibles du langage machine ; 2) les prédicats sont les conditions de branchement offertes par le langage machine ; 3) les transitions sont les branchements conditionnels ou inconditionnels et le passage en séquence du langage machine.

Ces machines séquentielles ont un lexique restreint caractérisé par : des branchements uniquement binaires, des opérations sur des vecteurs de bits de longueur fixe, pris dans des registres ou dans le tableau MEM représentant la mémoire ; des prédicats de branchement pris dans un ensemble prédéfini de formules booléennes à base d'indicateurs N, Z, C et V.

La figure 12.4 donne le lexique d'une telle machine séquentielle à actions. La figure 12.5 donne un exemple d'algorithme exprimé comme machine séquentielle à actions. Nous verrons au chapitre 13 comment obtenir facilement à partir de cette machine un programme en langage machine ou un texte du langage d'assemblage.

1.6 Algorithme d'interprétation du langage machine

Une autre manière de comprendre le langage machine est d'en donner un algorithme d'interprétation.

Lorsque l'on travaille sur un ordinateur dont le processeur a pour langage machine précisément le langage machine considéré, le programme en langage machine est directement interprété par le processeur, qui constitue une réalisation câblée de l'algorithme d'interprétation. Nous précisons cette notion au chapitre 14.

Lorsque le langage machine est *émulé* (Cf. Paragraphe 1.7.2) sur un ordinateur quelconque, l'algorithme d'interprétation est exprimé dans un langage de programmation comme C, compilé (c'est-à-dire traduit en langage machine de l'ordinateur hôte) puis exécuté sur cet ordinateur (voir chapitre 18 pour plus de détails sur la différence entre interprétation et compilation).

1.6.1 Un petit langage machine

Nous considérons un langage machine à 5 instructions. La nature de l'instruction est codée sur un octet. Il y a toujours un octet d'extension contenant

{ *Types mots de 8, 32 bits :* }
Vecteur8 : un tableau sur [0..7] de booléens
Vecteur32 : un tableau sur [0..31] de booléens
{ *Les 8 registres :* }
R : un tableau sur [0..7] de Vecteurs32
NumReg : le type entier sur 0..7
{ *La mémoire :* }
MEM : un tableau sur [0..tmem−1] de Vecteurs8
{ *Les indicateurs d'opérations arithmétiques* }
N, Z, C, V : des booléens
{ *Les prédicats :* }

fonctionA ⟶ un booléen	{ $fonctionA = vrai$ }
fonctionNev ⟶ un booléen	{ $fonctionNev = faux$ }
fonctionN ⟶ un booléen	{ $fonctionN = N$ }
fonctionZ ⟶ un booléen	{ $fonctionZ = Z$ }
fonctionC ⟶ un booléen	{ $fonctionC = C$ }
fonctionV ⟶ un booléen	{ $fonctionV = V$ }
fonctionNbar ⟶ un booléen	{ $fonctionN = non\ N$ }
fonctionVbar ⟶ un booléen	{ $fonctionVbar = non\ V$ }
fonctionCbar ⟶ un booléen	{ $fonctionCbar = non\ C$ }
fonctionZbar ⟶ un booléen	{ $fonctionZbar = non\ Z$ }
fonctionBGU ⟶ un booléen	{ $fonctionBGU = non\ (C\ ou\ Z)$ }
fonctionBGE ⟶ un booléen	{ $fonctionBGE = non\ (N\ ouexcl\ V)$ }
fonctionBG ⟶ un booléen	{ $fonctionBG = non\ (Z\ ou\ (N\ ouexcl\ V))$ }
fonctionBLEU ⟶ un booléen	{ $fonctionBLEU = C\ ou\ Z$ }
fonctionBL ⟶ un booléen	{ $fonctionBL = (N\ ouexcl\ V)$ }
fonctionBLE ⟶ un booléen	{ $fonctionBLE = Z\ ou\ (N\ ouexcl\ V)$ }

{ *Quelques actions :* }
SoustReg : une action (les données un, deux, trois : 3 NumReg)
lexique
X : un entier dans $[-2^{33-1}, 2^{33-1} - 1]$
algorithme
X ⟵ R[un] - R[deux] { *Cf. Chapitre 3, Paragraphes 2.2.5 et 3.2.3* }
si (R[un]$_{31}$ et non R[deux]$_{31}$ et non X$_{31}$) ou (non R[un]$_{31}$ et R[deux]$_{31}$ et X$_{31}$)
alors V ⟵ 1 sinon V ⟵ 0
si (non R[un]$_{31}$ et R[deux]$_{31}$) ou (X$_{31}$ et (non R[un]$_{31}$ ou R[deux]$_{31}$))
alors C ⟵ 1 sinon C ⟵ 0
Z ⟵ si X$_{31..0}$ = 0 alors 1 sinon 0
N ⟵ X$_{31}$
R[trois] ⟵ X$_{31..0}$

FIG. 12.4 – Le langage machine décrit par le lexique d'une machine séquentielle avec actions. On ne décrit ici que l'instruction soustraction avec mise à jour des indicateurs arithmétiques

```
{ Calcul du pgcd de deux entiers
A0 et B0 strictement positifs par
soustractions successives }
{ Posons A = A0 et B = B0 }
tantque A ≠ B
    si A > B alors
        A ⟵ A − B
    sinon
        B ⟵ B − A
{ propriété : A = B = pgcd(A0, B0) }
```

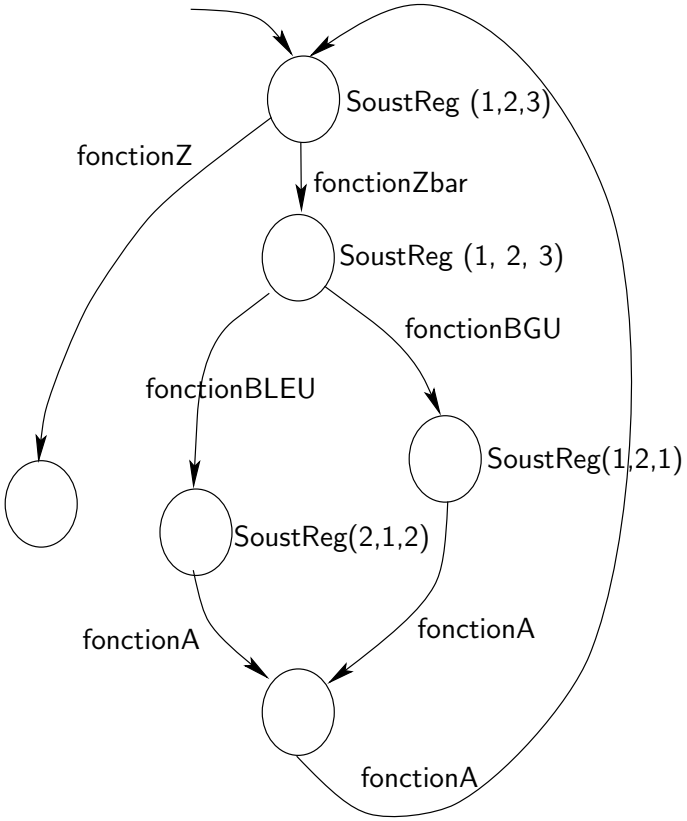


FIG. 12.5 – Traduction d'un algorithme en machine séquentielle à actions et lexique restreint : la machine peut bien sûr être simplifiée. Initialement, A0 et B0 sont respectivement dans les registres 1 et 2.

un numéro de registre. Les instructions sont : ADD (addition d'un registre et d'une valeur immédiate, donnée dans un octet d'extension), BT (branchement inconditionnel absolu, adressage direct par registre, dont le numéro est donné dans un octet d'extension), BZ (branchement si le résultat de la dernière opération de calcul était nul, relatif, direct par registre, numéro dans un octet d'extension), INCRR (incrémentation d'un registre, numéro dans un octet d'extension) et INCRM (incrémentation d'un mot mémoire, adressage indirect par registre, numéro dans un octet d'extension). Une machine proche de celle-ci est utilisée au chapitre 14 ; une autre plus complète au chapitre 22.

1.6.2 Interprétation

Pour décrire l'interprétation d'un programme écrit dans notre petit langage machine, nous supposons ici que le programme est installé en mémoire, à partir de l'adresse début. Le problème général de l'installation du codage binaire d'un programme en mémoire, ou *chargement*, est étudié au chapitre 18.

Chaque instruction est codée sur 1 octet, plus un ou deux octets d'extension. L'algorithme d'interprétation est donné figure 12.6.

Noter la complexité de certaines actions, par exemple celle nécessaire à l'interprétation d'une instruction d'incrémentation en mémoire, avec adressage indirect par registre : MEM[Reg[NumReg]] $\longleftarrow$ MEM[Reg[NumReg]] + 1. Nous verrons au chapitre 14 les contraintes qu'impose une telle action sur ce qu'on appelle la *partie opérative* du processeur, c'est-à-dire l'unité de calcul, les registres et les bus. Au niveau algorithmique, il est facile de remplacer cette action complexe par une action équivalente :

```
temp ⟵ MEM[Reg[NumReg]]
temp ⟵ temp + 1
MEM[Reg[NumReg]] ⟵ temp
```

qui fait apparaître un nouveau registre temp. Cette nouvelle forme permet de séparer l'accès en écriture de l'accès en lecture à la mémoire.

Noter également le cas d'erreur lorsque le code d'instruction lu ne correspond à aucune instruction valide (il est en effet possible que le codage de l'ensemble des instructions en vecteurs de n bits ne constitue pas une fonction surjective, comme nous l'avons signalé au paragraphe 1.3.1). Dans un programme d'interprétation du langage machine, on peut émettre un message d'erreur. Lorsque le langage machine est directement interprété par le processeur, cette erreur doit également être détectée, et signalée au programmeur. Nous y revenons au chapitre 24, paragraphe 1.2.

En toute rigueur, il faudrait prévoir une détection d'erreur lors de la lecture d'un numéro de registre dans un octet : NumReg $\longleftarrow$ MEM[PC], à moins de supposer que toute configuration d'un vecteur de 8 booléens correspond effectivement à un numéro de registre existant, c'est-à-dire que la machine comporte 256 registres.

```
lexique
    Vecteur8 : un tableau sur [0..7] de booléens
    Vecteur32 : un tableau sur [0..31] de booléens
    Reg : le tableau sur 0..31 de Vecteurs32                { Les 32 registres }
    NumReg : le type entier sur 0..7 ; adresse : le type entier sur 0..tmem−1
    MEM : un tableau sur [adresse] de Vecteurs8             { La mémoire }
    Z : un booléen                                { Indicateur d'opération arithmétique }
    ADD, BT, BZ, INCRR, INCRM : les constantes de type Vecteur8 : 0, 1, 2, 3, 4
    Inst : un Vecteur8 ; PC : une adresse
    début : une adresse ; taille : un entier > 0
        { MEM[début ... début+taille] contient les instructions }
algorithme
    PC ⟵ début
    tantque PC < début + taille
        Inst ⟵ MEM[PC] { premier octet } ; PC ⟵ PC + 1
        { Lire no registre, nécessaire pour toutes les instructions (1 octet) }
        NumReg ⟵ MEM[PC] ; PC ⟵ PC + 1
        { lecture autres octets selon l'instruction }
        selon Inst :
            Inst = ADD :
                { ajout de la valeur immédiate, troisième octet : }
                Reg[NumReg] ⟵ Reg[NumReg] + MEM[PC] ; PC ⟵ PC + 1
                si Reg[NumReg] = 0 alors Z ⟵ 1 sinon Z ⟵ 0
            Inst = BT : { NumReg est le numéro du registre d'indirection }
                PC ⟵ Reg[NumReg]
            Inst = BZ : { NumReg est le numéro du registre d'indirection }
                si Z alors { branchement effectif }
                    PC ⟵ PC + Reg[NumReg]
                sinon { PC ⟵ PC + 1 déjà effectué : passage en séquence }
            Inst = INCRR : { NumReg est le numéro du registre à incrémenter }
                Reg[NumReg] ⟵ Reg[NumReg] + 1
                si Reg[NumReg] = 0 alors Z ⟵ 1 sinon Z ⟵ 0
            Inst = INCRM : { NumReg est le numéro du registre d'indirection,
                             qui donne l'adresse de l'emplacement mémoire à
                             incrémenter }
                MEM[Reg[NumReg]] ⟵ MEM[Reg[NumReg]] + 1
                si MEM[Reg[NumReg]] = 0 alors Z ⟵ 1 sinon Z ⟵ 0
            sinon :
                { code d'instruction invalide. Cf. Chapitre 24 }
```

FIG. 12.6 – Algorithme d'interprétation du petit langage machine

1.7 Critères de choix d'un ensemble d'instructions

Nous résumons ci-dessous quelques-unes des contraintes à prendre en compte globalement pour le choix d'un jeu d'instructions. C'est un problème qui ne se pose qu'aux concepteurs de processeurs, mais l'utilisateur d'un processeur doit en être conscient.

1.7.1 Contraintes issues du coût du circuit interprète

Selon que l'on privilégie la variété des instructions offertes par le langage machine, ou bien le coût du circuit interprète (le processeur), on construit des jeux d'instructions assez différents.

L'exemple typique consiste à comparer un SPARC et un 68000. Le SPARC est une machine dite RISC (pour *Reduced Instruction Set Computer*). En réalité l'ensemble des opérations effectivement disponibles n'est pas si réduit que cela, mais les modes d'adressage sont rudimentaires : toute opération arithmétique se fait sur des registres, et il n'existe que deux instructions spécifiques LOAD et STORE pour réaliser des accès mémoire. Les instructions sont codables sur un format fixe, et leur interprétation est algorithmiquement simple. Le circuit interprète est donc simple, lui aussi. La simplicité, en termes de matériel, se traduit par la taille du circuit (en nombre de portes par exemple). Il reste donc de la place pour équiper le processeur d'un grand nombre de registres (peut être de l'ordre de 500, nous verrons au chapitre 13 comment utiliser ce *banc de registres* du SPARC pour rendre efficace le codage des procédures d'un langage d'actions). D'autre part le format fixe des instructions permet de les exécuter toutes dans le même temps.

Le 68000, en revanche, autorise généralement 8 modes d'adressage pour les opérations. Les instructions ont un format variable selon qu'elles font référence ou non à un opérande en mémoire, qui doit être désigné par son adresse. L'algorithme d'interprétation est plus complexe. Le processeur est *microprogrammé* (Cf. Chapitre 10) et les interprétations des instructions ont des durées différentes.

1.7.2 Problèmes de compatibilité

La définition d'un langage machine pour une nouvelle machine n'est jamais totalement libre. Une nouvelle machine est en général une étape dans une gamme de machines similaires plus anciennes, pour lesquelles il existe de nombreux programmes.

Lorsqu'un constructeur propose la machine de génération $n+1$, il doit assurer que les programmes qui fonctionnent sur les machines de génération n peuvent être réutilisés. Cette notion cache de nombreux niveaux. Elle peut signifier, simplement, que le constructeur fournit avec sa nouvelle machine un système et un compilateur pour chaque langage qui était disponible auparavant. Un utilisateur ayant programmé en C doit transmettre d'une machine

à l'autre les fichiers *source* de ses programmes, les compiler avec le nouveau compilateur, et les exécuter.

Malheureusement les utilisateurs très anciens ont parfois perdu les fichiers source de leurs programmes, ou bien, ce qui est le cas pour presque tous les logiciels commerciaux, ils n'avaient acheté le programme que sous sa forme exécutable. Ils ne disposent donc plus que du fichier objet, c'est-à-dire un programme en langage machine de génération n. Le constructeur doit alors garantir la *compatibilité ascendante* de ses machines, c'est-à-dire faire en sorte que le fichier objet de génération n soit interprétable sur la machine de génération $n+1$.

Il y a essentiellement deux solutions. Si les deux machines sont conceptuellement proches l'une de l'autre, le jeu d'instructions de la nouvelle machine est défini comme un sur-ensemble du jeu d'instructions de l'ancienne. Les anciens programmes sont exécutables directement sur la nouvelle machine ; ils n'utilisent qu'une partie du nouveau jeu d'instructions, et sont donc peut-être moins efficaces que ce qu'ils auraient pu être en profitant au mieux du nouveau jeu d'instructions. C'est le cas entre SPARC et ULTRASPARC.

Si les deux machines sont très différentes, le constructeur fournit un *émulateur* du langage machine n sur la machine $n+1$. Un émulateur est un programme, écrit dans un langage quelconque, par exemple C, et compilé sur la nouvelle machine, avec le nouveau compilateur C. Ce programme est un interprète du langage machine n. Le code objet des anciens programmes n'est donc plus directement interprété par un processeur, mais par un programme, lui-même compilé et exécuté sur un autre processeur.

C'est le cas des MACINTOSH : les processeurs 68000 et PowerPC sont très différents et il n'y a pas de compatibilité ascendante de leurs jeux d'instructions. Apple fournit donc un émulateur de 68000 parmi les programmes du logiciel de base fourni avec les machines à PowerPC.

1.7.3 Langage machine et traduction des langages de haut niveau, machines-langages

Le jeu d'instructions d'une machine peut comporter, outre les classes d'instructions usuelles du modèle Von Neumann, des instructions très spécifiques destinées à la traduction des programmes en langage de haut niveau.

On peut dire ainsi que le 68000 est une machine pour la compilation des langages à structures de blocs (Pascal, C, Ada, etc.). Le jeu d'instructions comporte les instructions `link` et `unlink` qui ne servent qu'à gérer l'allocation dynamique de mémoire pour les variables d'un tel type de langage (Cf. Chapitre 13).

De la même manière, le processeur SPARC offre les instructions `save` et `restore` qui permettent de décaler une *fenêtre* sur le banc de registres. Ce mécanisme permet l'allocation de mémoire pour le contexte local d'une procédure, et le passage de paramètres directement dans les registres, sans

accès mémoire.

De manière générale, la conception des processeurs est de plus en plus indissociable de la compilation des langages de haut niveau.

Si l'on pousse le raisonnement à l'extrême, le jeu d'instructions d'une machine peut être entièrement conçu pour l'exécution de programmes écrits dans un langage donné. On a ainsi construit des *machines LISP*, des *machines PROLOG*, etc. On entend parler également de *machines JAVA*. Dans ce dernier cas, de quoi s'agit-il exactement ? Les concepteurs du langage JAVA en ont fait la publicité avec des arguments de portabilité et de sécurité : JAVA est d'abord *compilé* en une forme intermédiaire (appelée *byte code*) qui est ensuite exécutable, par *interprétation*, sur tout ordinateur qui possède l'interprète adéquat. L'exécution sous forme d'interprétation est censément plus sûre que l'exécution par le processeur d'un programme en langage machine ; en effet, l'outil d'interprétation peut effectuer quelques vérifications sur les opérations qu'il effectue. Si maintenant on propose un processeur dont le langage machine est exactement le format intermédiaire produit par la première phase de compilation de Java, on obtient une machine Java.

En général, la question se pose de savoir s'il vaut mieux concevoir une machine dédiée à un langage particulier, et capable d'assurer une exécution efficace des programmes écrits dans ce langage, ou bien concevoir un bon compilateur de ce langage pour machine universelle.

2. Le langage d'assemblage

Le langage machine offre déjà tout ce qui est théoriquement nécessaire pour programmer n'importe quel algorithme. Il lui manque une notation lisible et manipulable par un être humain. Le *langage d'assemblage* est introduit à cet effet et offre donc :

- tout d'abord une notation textuelle aisément lisible du langage machine, c'est-à-dire : 1) une notation des opérations de la machine (les mnémoniques) et de leurs opérandes, 2) un ensemble de directives de réservation et d'initialisation de la mémoire
- la possibilité d'introduire des *commentaires*
- une notion de *zones* distinctes dans un programme : la zone des instructions (TEXT) et la zone de données (DATA, BSS), ainsi qu'une notation qui permet de repérer facilement les portions de programme appartenant à l'une ou l'autre de ces deux zones.
- un mécanisme de *nommage* des positions dans la zone des instructions ou dans la zone de données, qui permet de s'abstraire des valeurs explicites d'adresses mémoire.

Notons qu'il peut exister plusieurs langages d'assemblage pour le même langage machine. Les différences résident en général dans la notation des modes

d'adressage des opérandes d'instructions, ou dans les mots-clés utilisés pour les directives de réservation mémoire. Inversement, un constructeur offre souvent des langages d'assemblage aussi similaires que possible pour des machines équipées de processeurs différents. Ainsi la syntaxe des assembleurs SUN est-elle la même, en ce qui concerne la définition des zones de données, que le processeur soit un 68000 ou un SPARC.

2.1 Aspects de lexicographie et macros

Les langages d'assemblage usuels offrent la possibilité d'introduire des commentaires, délimités par exemple comme en C par : `/*` et `*/` ou par un caractère, par exemple '!' et la fin de ligne qui suit.

Ils offrent également la notation des constantes entières dans plusieurs bases (typiquement décimal, octal — préfixe 0 —, hexadécimal — préfixe 0x), ainsi que la notation des constantes entières du code ASCII directement sous forme de caractères : `'a'` représente l'entier noté `97` en décimal (voir chapitre 3 pour des détails sur le codage des caractères).

Enfin il est en général possible, soit dans l'assembleur, soit dans un outil situé en amont et appelé *macro-assembleur*, de définir des *constantes textuelles*. Ce mécanisme permet d'éviter une redondance d'écriture, qui mène souvent à des incohérences lors des modifications. On écrit ainsi en début de fichier une ligne de la forme `tailleMax=400`. Lors de l'assemblage, une première passe sur le texte du programme remplace toute occurrence de la chaîne de caractères `tailleMax` (sauf dans les commentaires) par la chaîne `400`. Ce mécanisme est assez limité, mais déjà très utile. Un outil comme `m4`, disponible dans les environnements UNIX, permet de définir des macros paramétrées ; il est ainsi possible de nommer une suite d'instructions qui apparaît souvent.

2.2 Structuration des programmes

Un programme en langage d'assemblage est destiné à être traduit en langage machine, puis placé en *mémoire vive* pour exécution par le processeur. Un programme comprend généralement des instructions et des données (respectivement l'algorithme et les descriptions de variables du lexique, dans la terminologie du langage d'actions présenté au chapitre 4). Le codage binaire du programme comporte donc le codage d'instructions, et le codage de données (des entiers, des caractères, des données structurées, etc.).

Or en langage machine, rien n'est typé. Le vecteur de booléens `00101010` peut être vu aussi bien comme : un entier naturel codé en binaire pur, le code ASCII d'un caractère, 2 champs du codage d'une instruction, la partie exposant du codage d'un réel, etc. Si la zone de mémoire sur laquelle on fait travailler le processeur contient des chaînes de bits qui correspondent au codage d'opérations valides, on peut exécuter ces opérations. Il n'y a pas de différence intrinsèque entre données et programmes.

Pour les besoins de la programmation, le langage d'assemblage fournit une notion de *zone* dans les programmes, et une notation correspondante. Les zones sont : TEXT pour les instructions ; DATA pour les données statiques, c'est-à-dire dont la valeur est donnée directement dans le texte du programme ; BSS pour les données seulement dynamiques. L'influence de la distinction entre les zones DATA et BSS est expliquée au chapitre 18.

Retenons simplement pour l'instant que la zone TEXT contient les instructions, et la zone DATA un ensemble de données initialisées que l'on désire placer en mémoire vive lors de l'exécution du programme. Ces données se comportent comme des variables globales d'un langage de programmation impératif : elles ont la durée de vie du programme.

2.3 Nommage des adresses ou étiquettes

Pour repérer les instructions dans la zone TEXT, ou les données dans la zone DATA, le langage d'assemblage fournit la notion d'*étiquette*. Ce mécanisme permet de faire abstraction des adresses absolues et des décalages exacts, dans le cas des branchements par exemple.

On écrit typiquement, dans un langage d'assemblage possible du processeur SPARC (`BNE`, `SUBcc` et `BA` sont des *mnémoniques*. voir paragraphe suivant) :

```
boucle:  SUBcc  r1, r2, r3
         BNE    fin
         ! n instructions ici
         BA     boucle
fin:
```

Les branchements conditionnels du SPARC sont relatifs, et `BNE fin` exprime donc le décalage des adresses entre la position de l'instruction elle-même et la cible du branchement, située à l'étiquette `fin`. En langage machine, le code de l'instruction `BNE fin` comprend un entier relatif qui donne le déplacement en nombre d'octets, divisé par 4, c'est-à-dire le déplacement en nombre d'instructions puisque toutes les instructions du SPARC sont codées sur 4 octets. Pour `BNE fin` on trouve un déplacement de $n+2$; pour `BA boucle`, on trouve $-(n+2)$.

La traduction des instructions de la forme `BNE fin` du langage d'assemblage en instructions du langage machine demande le calcul du décalage effectif. C'est un travail réalisé par l'assembleur, qui réalise une analyse lexicale du texte du programme, et associe à chaque étiquette une adresse relative au début du programme. Dans l'exemple ci-dessus, on associe 0 à `boucle` et $(n+3) \times 4$ à `fin`. C'est suffisant pour calculer les décalages signalés ci-dessus.

Lorsque les branchements sont absolus, l'utilisation d'une étiquette permet de manière similaire de faire abstraction de l'adresse absolue destination du branchement. Du point de vue du programmeur en langage d'assemblage, il n'est donc pas nécessaire de savoir si les branchements sont relatifs ou absolus.

En revanche l'assembleur doit maintenant traduire les étiquettes en adresses absolues, dans la mémoire vive de la machine, au moment de l'exécution. Pour cela il faut connaître l'adresse de base à laquelle le programme sera installé ; cette information n'est pas toujours disponible au moment de l'assemblage du programme. Il se pose alors un nouveau problème : comment produire un programme en langage machine indépendant de sa position d'installation en mémoire ? Un tel objet est appelé *code translatable*. Nous étudions sa production au chapitre 18, paragraphe 2.3.

Dans des ordinateurs simples où les programmes sont toujours installés à partir de la même adresse en mémoire, il n'est pas nécessaire de produire du code translatable.

2.4 Zone des instructions, mnémoniques et notation des opérandes

Observons Figure 12.7 un exemple, donné dans un langage d'assemblage possible pour processeur SPARC.

On associe à chaque instruction un *mnémonique*, c'est-à-dire un nom court et évocateur. Par exemple `LDUH` signifie *LoaD Unsigned Half*, c'est-à-dire chargement non signé, de taille demi-mot.

Une instruction tient en général sur une ligne, commençant par un mnémonique d'instruction, suivi de notations d'opérandes séparés par des virgules. Il semble que le terme *langage d'assemblage* vienne de là : il s'agit d'assembler ces différentes parties pour constituer une instruction.

Les paramètres des instructions peuvent être : des registres, des constantes, des désignations d'adresses en mémoire par l'intermédiaire de modes d'adressage plus ou moins sophistiqués.

Les registres sont en général nommés, par des noms dans lesquels apparaît une numérotation, comme `r1, r2, ...`. Il peut exister plusieurs noms pour un même registre physique : on parle d'alias. Lorsqu'il y a très peu de registres, les noms des registres peuvent être intégrés aux mnémoniques. On trouve par exemple dans un langage d'assemblage pour processeur 6502 les mnémoniques `LDA`, `LDY` et `LDX` qui correspondent au chargement des registres `A`, `Y` et `X`.

Les constantes entières sont notées en utilisant une des bases disponibles, ou même par l'intermédiaire du code ASCII (Cf. Paragraphe 2.1). L'utilisation des constantes textuelles permet d'écrire `ADD r1, MAX, r4` à la place de `ADD r1, 4, r4`.

Noter que l'instruction `ADD` du SPARC exige des registres comme premier et troisième opérande, mais autorise un registre ou une constante (suffisamment petite) comme deuxième opérande. La différence entre ces deux cas doit bien sûr être codée dans l'instruction du langage machine correspondante, puisque l'interprétation d'un des champs de l'instruction en dépend (valeur immédiate ou numéro de registre). Or le mnémonique est le même dans les deux cas. La distinction est donc faite par l'assembleur sur des critères lexicographiques :

```
MAX=4   ! définition d'une constante textuelle
        .text           ! début de la zone TEXT
ADDcc   r1, r2, r3
ADD     r1, MAX, r4
LDUH    [r1+4], r2
BNE     labas
CALL    fonction
JMPL    r2+r3, r0
```

FIG. 12.7 – Exemple de zone TEXT

```
VAL=250
ISE=-125
        .data           ! début de zone de données
XX :                    ! étiquette
        .long 0x15      ! 4 octets initialisés avec le vecteur de
                        ! bits décrit par 0x15 en hexadécimal,
                        ! repérables par l'étiquette XX
YY :    .half -12       ! 2 octets initialisés avec le codage
                        ! de -12 (nécessairement en complément à 2)
        .byte VAL       ! un octet initialisé avec le codage
                        ! de 250 (nécessairement en binaire pur)
        .byte ISE       ! un octet initialisé avec le codage
                        ! de -125 (nécessairement en complément à 2)
        .skip 12000     ! une zone contiguë de 12000 octets,
                        ! non initialisés.
        .asciz "toto"   ! 5 octets, initialisés avec les codes
                        ! ASCII des caractères 't', 'o', 't', 'o'
                        ! et un octet mis à zéro pour finir.
        .align 4        ! directive d'alignement
ZZ :    .long XX        ! 4 octets initialisés avec le codage
                        ! binaire de l'adresse absolue
                        ! représentée par l'étiquette XX.
```

FIG. 12.8 – Exemple de zone DATA

on peut toujours distinguer une chaîne de caractères qui constitue la notation d'une constante entière, d'une chaîne de caractères qui constitue un nom de registre, grâce à la lettre 'r' qui précède le numéro. Lorsqu'il risque d'y avoir confusion, ou même simplement pour des raisons de lisibilité, les constantes sont préfixées par le caractère '#'. On écrit ainsi dans un langage d'assemblage pour 68000 : `ADDI #4, D0`.

Les notations les plus compliquées viennent des modes d'adressage. Pour des raisons de lisibilité et de compréhension du programme, le langage d'assemblage propose en général une notation particulière pour les modes d'adressage qui supposent une indirection. Ce peuvent être des crochets ou des parenthèses. On écrit ainsi `LDUH [r1+4], r2` dans un langage d'assemblage pour SPARC, ou `move.l (A1), D1` dans un langage d'assemblage pour 68000 (voir exemple complet, figure 12.9). Les déplacements éventuels sont notés par des additions, comme dans `LDUH [r1+4], r2`.

Enfin un langage d'assemblage peut définir des *pseudo-instructions*. Par exemple, le jeu d'instruction SPARC étant limité, il n'existe pas d'instruction de comparaison de deux entiers (l'équivalent de l'instruction `CMP` du 68000). On utilise donc une soustraction de la forme `SUBcc r1, r2, r0` pour mettre à jour les indicateurs arithmétiques selon le résultat de la comparaison de `r1` et `r2` (`r0` est un registre spécial, dans lequel l'écriture n'a aucun effet. Voir exercice E12.2). L'assembleur permet d'écrire simplement `CMP r1, r2`, qui sera traduit en langage machine exactement comme `SUBcc r1, r2, r0`. `CMP` est une pseudo-instruction. Notons que certaines pseudo-instructions remplacent parfois plusieurs instructions.

2.5 Zone de données et directives de réservation mémoire

La zone de données comporte des *directives de réservation* d'emplacements mémoire, avec définition d'une valeur initiale à y placer avant de démarrer l'exécution du programme. Le codage de ces données en langage machine est simplement le codage binaire des constantes indiquées par le programmeur.

Observons l'exemple de la figure 12.8.

`.long`, `.half`, `.byte` et `.asciz` sont des directives de réservation mémoire avec initialisation. Elles sont suivies de la donnée d'une constante, sous des formes diverses : nom de constante textuelle (`.byte VAL`), notation d'entier en hexadécimal (`.long 0x15`), en décimal (`.half -12`), etc. La constante peut aussi être donnée sous la forme d'une étiquette (`.long XX`). Une étiquette représente une adresse, donc un entier positif. La seule difficulté provient du fait que l'adresse absolue associée à l'étiquette n'est pas connue avant le chargement en mémoire pour exécution. On retrouve ici pour les données la notion de *code translatable* déjà mentionnée au paragraphe 2.3.

Dans le cas de `.asciz`, la taille est déterminée par la constante chaîne de caractères associée. Ainsi `.asciz "toto"` réserve $5 = 4 + 1$ octets, dont les

4 premiers sont initialisés avec les codes ASCII des caractères `'t'`, `'o`, `'t'`, `'o'` et le dernier avec un caractère de code 0. On respecte ainsi la convention de représentation des chaînes de caractères utilisée en C, c'est-à-dire sous la forme d'une adresse de début, sachant que la chaîne est marquée par 0.

`.skip` est une directive de réservation mémoire sans initialisation. On la trouve plutôt en zone BSS que DATA, puisque la zone BSS permet de réserver de la mémoire non initialisée (Cf. Chapitre 18).

`.align` est une directive de cadrage mémoire nécessaire pour tenir compte des contraintes d'alignement mémoire dont nous avons parlé au paragraphe 2.3 du chapitre 4. La directive `.align 4` (resp. `.align 2`) tient compte de toutes les réservations mémoire effectuées auparavant, et ménage un espace perdu de la taille nécessaire pour atteindre la prochaine adresse multiple de 4 (resp. 2).

3. Traduction du langage d'assemblage en langage machine

La traduction du langage d'assemblage en langage machine est effectuée par l'outil appelé *assembleur* des environnements de programmation. Cet outil a essentiellement 3 fonctions :

- il réalise l'analyse lexicale et syntaxique du texte écrit en langage d'assemblage, diverses vérifications comme par exemple l'utilisation correcte des modes d'adressage et peut rejeter des programmes pour erreurs lexicales ou syntaxiques ; cette phase d'analyse peut s'accompagner de l'expansion des macro-notations et des définitions de constantes ;
- il réalise le codage en binaire des instructions et des données : transformation des mnémoniques d'opérations et des notations de modes d'adressage en codage des instructions, transformation des directives de réservation mémoire en codage binaire des données initialisées ;
- il réalise la traduction des étiquettes (ou symboles) en adresses absolues ou déplacements. En maintenant au cours du traitement un compteur associant à chaque instruction une adresse calculée par rapport au début du programme, il est facile de calculer les déplacements relatifs. Par contre les valeurs absolues ne peuvent être calculées tant que l'on ignore l'adresse à laquelle le programme sera implanté. Nous revenons sur ce point au chapitre 18 à propos de code translatable.

Nous montrons dans la section suivante à travers quelques exemples la traduction en langage machine SPARC et 68000.

4. Un exemple de programme

Considérons l'algorithme qui permet de déterminer le nombre de bits à 1 dans la représentation en binaire pur d'un entier :

Lexique :
  x : un entier ≥ 0 { *la donnée* }
  NombreDeUns : un entier ≥ 0 { *le résultat à calculer* }
Algorithme :
  NombreDeUns $\longleftarrow$ 0
  tantque x $\neq$ 0
    si x mod 2 $\neq$ 0 alors NombreDeUns $\longleftarrow$ NombreDeUns + 1
    x $\longleftarrow$ x div 2

Nous donnons ci-dessous des programmes correspondants dans des langages d'assemblage possibles pour les processeurs SPARC et 68000, et les programmes en langage machine correspondants. Noter qu'en langage d'assemblage les opérations arithmétiques notées x mod 2 et x div 2 sont remplacées par des manipulations explicites de vecteurs de booléens (décalage logique et masquage).

4.1 Programme pour 68000

La figure 12.9 donne un programme en langage d'assemblage pour processeur 68000.

Nous donnons ci-dessous le codage des instructions, accompagné de l'adresse de rangement, et du texte en langage d'assemblage obtenu par décodage dans un *désassembleur*.

```
0x0   43f9 00000028              lea     40,A1
0x6   2211                       move.l  (A1),D1
0x8   7000                       moveq   #0,D0
0xA   4a81                       tst.l   D1
0xC   6710                       beq     +16
0xE   2401                       move.l  D1,D2
0x10  0282 00000001              andi.l  #1,D2
0x16  6702                       beq     +2
0x18  5280                       addq.l  #1,D0
0x1A  e281                       asr.l   #1,D1
0x1C  60ec                       bra     -20
0x1E  23c0 0000002c              move.l  D0,44
0x24  4e75                       rts
```

Noter que l'on produit du code comme si le programme démarrait à l'adresse 0. Les données sont placées après la zone texte ; la zone `data` commence donc ici à l'adresse `0x28`. Le codage de la zone `data` est : `0000002a`, c'est-à-dire la valeur 42 en décimal codée sur 16 bits.

Lorsque le programme est *chargé* em mémoire centrale, les adresses sont *translatées* en fonction de l'adresse à laquelle le programme est rangé en mémoire (Cf. Chapitres 18 et 20).

Détaillons le codage de quelques instructions :

Exemple 1 : `43f9 00000028` `lea Donnee, A1`

```
! Correspondance des variables et des registres :
! x : D1
! NombreDeUns : D0
              .data           ! ZONE DE DONNEES INITIALISEES
Donnee :      .long 42        ! un mot de 4 octets contenant
                              ! le codage binaire de l'entier
                              ! noté 42 en decimal.
              .bss            ! ZONE DE DONNEES non INIT.
Resultat :    .skip 4         ! un mot de 4 octets
              .text           ! ZONE DE PROGRAMME
              .global _main   ! nécessaire (Cf. Chap. 18, §2.)
_main :                       ! le pt d'entrée s'appelle
                              ! nécessairement _main.
              lea Donnee,A1
              !transfert de la mémoire vers un registre
              !en adressage indirect par registre adresse.
              move.l (A1), D1
deb_calcul :  moveq #0,D0     ! NombreDeUns <-- 0
condition_tq :tst.l D1        ! tantque (x!=0)
              beq fin_tq      ! branchement lorsque cond. fausse
corps_tq :
si :
              move.l D1,D2
              andi.l #1,D2    ! si (x mod 2!= 0) alors
              beq fin_si
alors :       addq.l #1,D0    ! NombreDeUns<--NombreDeUns+1
fin_si :
              asr.l D1        ! x <-- x div 2
              bra condition_tq
fin_tq :
              move.l D0, Resultat
              ! transfert d'un registre vers la mémoire
              ! en adressage direct
fin_prog :    rts             ! fin de programme.
```

FIG. 12.9 – Nombre de 1 : programme en langage d'assemblage pour 68000

Il s'agit d'une instruction *load effective address* dans un registre.

`43f9` est le codage de la nature de l'instruction, du numéro de registre (A1) et du mode d'adressage utilisé pour l'autre opérande (c'est cela qui détermine s'il y a des mots d'extension à lire, et combien).

`00000028` : 2 *mots d'extension* de 16 bits chacun, donnant la valeur d'une adresse. Ici 0x28 en hexadécimal, c'est-à-dire 40 en décimal : c'est la taille en octets de la zone TEXT. Les données sont placées en mémoire après les instructions, et `Donnee` est le premier objet de la zone DATA.

Exemple 2 : `2211 move.l (A1), D1`

Tout est codé en 16 bits : la nature de l'opération, la taille des opérandes sur laquelles elle s'applique, la description des modes d'adressage des deux opérandes (direct ou indirect par registre, numéro de registre).

Exemple 3 : `6710 beq +16`

Tout est codé en 16 bits : la nature de l'opération (un branchement si Z), le déplacement de 16 octets, par rapport à l'instruction qui *suit* celle du branchement. L'instruction de branchement est à l'adresse 0xC : la suivante à l'adresse 0xE ; la destination à l'adresse 0x1E.

4.2 Programme pour SPARC

La figure 12.10 donne un programme en langage d'assemblage pour processeur SPARC.

Le codage des données (zone data) est : `0000 002a 0000 0000`. Le codage des instructions (zone text) donne une suite de mots de 32 bits, chacun codant une instruction.

```
9de3bfc0 21000000 a0142000 d2040000
90000000 80a24000 02800009 01000000
808a6001 02800003 01000000 90022001
93326001 10bffff8 01000000 21000000
a0142000 d0240000 81c7e008 81e80000
```

Exemple 1 : `21000000 a0142000 set Donnee, %l0`

Il s'agit d'une pseudo-instruction permettant l'affectation d'une valeur de 32 bits, l'adresse `Donnee` en zone `data`. Elle est remplacée par deux instructions `sethi %hi(Donnee), %l0` et `or %l0, %lo(Donnee), %l0` dont les codages respectifs sont : `21000000` et `a0142000`. La valeur associé à `Donnee` n'étant pas connue lors de la traduction, les champs correspondants sont pour l'instant mis à 0. Ils seront mis à jour lors du chargement du programme en mémoire centrale. L'information concernant `Donnee` est présente dans les informations de translation et la table des symboles (Cf. Chapitre 18).

Exemple 2 : `01000000 nop`

```
! Correspondance des variables et des registres :
! x : o1
! NombreDeUns : o0
              .data             ! ZONE DE DONNEES INITIALISEES
Donnee :      .long 42          ! un mot de 4 octets contenant
                                ! le codage binaire de l'entier
                                ! noté 42 en decimal.
              .bss              ! ZONE DE DONNEES non INIT.
Resultat :    .skip 4           ! un mot de 4 octets
                                ! non initialisé.
              .text             ! ZONE DE PROGRAMME
              .global _main     ! nécessaire (Cf. Chap. 18, §2.)
_main :       ! le pt d'entrée s'appelle
              ! nécessairement _main.
              save %o6, -64, %o6 ! voir codage des actions.
              set Donnee, %l0
              ! transfert de la mémoire vers un registre :
              ld [%l0], %o1
deb_calcul :  add %g0,%g0,%o0   ! NombreDeUns <-- 0
condition_tq :
              subcc %o1,%g0,%g0 ! tantque (x !=0)
              be fin_tq         ! branchement lorsque
              ! condition fausse
              nop
corps_tq :
si :
              andcc %o1,1,%g0   ! si (x mod 2 != 0) alors
              be fin_si
              nop
alors :
              add %o0,1,%o0     ! NombreDeUns<--NombreDeUns+1
fin_si :
              srl %o1,1,%o1     ! x <-- x div 2
              ba condition_tq
              nop
fin_tq :
              set Resultat, %l0
              ! transfert d'un registre vers la mémoire
              st %o0, [%l0]
fin_prog :
              ret               ! fin du programme
              restore
```

FIG. 12.10 – Nombre de 1 : programme en langage d'assemblage pour SPARC

L'instruction `nop` n'existe pas en SPARC. Elle est codée par une instruction ayant un effet nul : une addition ou une disjonction portant sur le registre spécial `%g0`.

Exemple 3 : `d2040000` `ld [%l0], %o1`

Cette instruction peut être codée complètement. Le mot de 32 bits se décompose en :

code op		reg d					code op						reg s 1					i
1	1	0	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0

inutilisé								reg s 2					
0	0	0	0	0	0	0	0	0	0	0	0	0	0

Exemple 4 : `02800009  be fin_tq`

L'opération est codée sur les bits 31 et 30 (resp. 0 et 0) et 24 à 22 (010). La condition equal est codée sur les bits 28 à 25 (0001). Le déplacement est codé sur les bits 21 à 0 ; la valeur est ici 9, ce qui correspond au nombre d'instruction de l'instruction de branchement jusqu'à l'étiquette `fin_tq`.

5. Exercices

E12.1 : Machines à 0, 1, 2 ou 3 références

Nous allons étudier l'écriture d'un algorithme simple dans un programme en langage d'assemblage de jeux d'instruction de différents types. Dans la suite de cet exercice nous notons `val` une valeur immédiate et `adr` une adresse. Nous utilisons les conventions d'écriture du langage d'assemblage similaire au SPARC décrite dans ce chapitre. `ope` représente le mnémonique d'une instruction parmi `add`, `mult`, `sub` et `div`, `OP` est l'opération arithmétique associée.

Traduire pour chacune des machines et langages d'assemblage associés décrits ci-après l'algorithme suivant, en convenant que A est à l'adresse `a`, ...et Y à l'adresse `y` :

Lexique :
A, B, C, D, E, F : des entiers { *des données* }
Y : un entier { *le résultat à calculer* }
Algorithme :
Y ⟵ (A + B + C) / (D * E * F - 3)

Même exercice avec le SPARC. Pour chaque type de machine, observer la taille du code obtenu en nombre d'instructions, estimer le nombre d'octets nécessaires au codage du programme.

1) Jeu d'instructions à 1 référence et 1 seul registre : la machine correspondant à ce type d'instruction possède un registre de calcul appelé accumulateur (noté `acc`) qui est toujours utilisé dans les instructions de calcul. L'accumulateur est un des deux opérandes et le résultat est forcément stocké dans celui-ci. L'ensemble des instructions ne possède qu'une référence :

instruction	signification
`ope val`	`acc ⟵ acc OP val`
`ope [adr]`	`acc ⟵ acc OP MEM[adr]`
`store adr`	`MEM[adr] ⟵ acc`
`load val`	`acc ⟵ val`
`load [adr]`	`acc ⟵ MEM[adr]`

2) Jeu d'instructions à 2 références et mode d'adressage restreint : on rajoute à la machine précédente d'autres registres (notés `Ri`). Chacun d'eux possède les mêmes fonctionnalités que `acc`. La destination du calcul est toujours un registre. Une des deux références (servant de source et destination) est en mode d'adressage registre direct.

instruction	signification
`ope Ri val`	`Ri` ⟵ `Ri OP val`
`ope Ri [adr]`	`Ri` ⟵ `Ri OP MEM[adr]`
`ope Ri Rj`	`Ri` ⟵ `Ri OP Rj`
`store Ri adr`	`MEM[adr]` ⟵ `Ri`
`load Ri val`	`Ri` ⟵ `val`
`load Ri [adr]`	`Ri` ⟵ `MEM[adr]`

Les processeurs fabriqués autour de l'année 1975 (famille 8080 d'INTEL et 6800 de MOTOROLA) utilisaient ces deux types d'instructions (1 et 2 références). Dans le cas de l'utilisation de l'accumulateur l'instruction est plus rapide.

3) Machine à 2 références et modes d'adressage variés : on ajoute un mode d'adressage indirect par registre. Une des deux références est forcément en mode d'adressage registre direct (mais pas forcément la destination). Les instructions `load` et `store` sont remplacées par une instruction unique `move`.

instruction	signification
Réf. destinations en mode d'adressage registre direct	
`ope val, Ri`	`Ri` ⟵ `val OP Ri`
`ope [adr], Ri`	`Ri` ⟵ `Ri OP MEM[adr]`
`ope [Rj], Ri`	`Ri` ⟵ `Ri OP MEM[Rj]`
Réf. destination en mode d'adressage absolu	
`ope Ri,[adr]`	`MEM[adr]` ⟵ `MEM[adr] ope Ri`
Réf. destination en mode d'adressage reg. indirect	
`ope Ri,[Rj]`	`MEM[Rj]` ⟵ `MEM[Rj] OP Ri`
Instruction `move`	
`move Rj, Ri`	`Ri` ⟵ `Rj`
`move val, Ri`	`Ri` ⟵ `val`
`move [adr], Ri`	`Ri` ⟵ `MEM[adr]`
`move Ri, [adr]`	`MEM[adr]` ⟵ `Ri`

Les processeurs de la famille 68000 de MOTOROLA (à patir de l'année 1980) a un jeu d'instructions de structure similaire avec un jeu de mode d'adressage encore plus large.

4) Machine à 0 référence : c'est une machine qui nécessite une pile. Les instructions d'opération se font toujours avec les deux opérandes qui sont au sommet de la pile (et qui sont alors enlevées de la pile), le résultat est stocké sur la pile. Ces instructions n'ont donc pas besoin de références. Il faut pouvoir stocker des valeurs sur la pile avant les calculs, on ajoute donc deux instructions particulières `push` et `pop`. Celles-ci ont une référence. On suppose ici que le pointeur de pile SP pointe sur le dernier emplacement occupé et que la pile progresse en diminuant SP.

instruction	signification
`ope`	`MEM[SP+1]` ⟵ `MEM[SP] OP MEM[SP+1]` ; `SP` ⟵ `SP+1`
`push val`	`SP` ⟵ `SP - 1` ; `MEM[SP]` ⟵ `val`
`push [adr]`	`SP` ⟵ `SP - 1` ; `MEM[SP]` ⟵ `MEM[adr]`
`pop [adr]`	`MEM[adr]` ⟵ `MEM[SP]` ; `SP` ⟵ `SP + 1`

E12.2 : Utilisation du registre `%g0` du SPARC
L'architecture du processeur SPARC introduit un objet manipulable comme un registre, mais qui possède des propriétés particulières : l'écriture dans ce registre, en général noté `%g0`, n'a pas d'effet, et la lecture de ce registre donne toujours 0. D'autre part le SPARC est une machine à trois références. Une instruction de calcul `op` a trois arguments `s1`, `s2` et `d`, et réalise l'affectation `d` ⟵ `op(s1, s2)`. Le dernier argument est toujours un registre, donc toute opération écrase un registre. Utiliser le registre `%g0` pour réaliser une comparaison et un test à 0 en termes de soustraction.

E12.3 : Transferts mémoire/registres en SPARC
Dans le jeu d'instructions du processeur SPARC on trouve des instructions de chargement d'une portion de registre (octet, B ou demi-mot, H) en mémoire signées (`LDB`, `LDH`) ou non signées (`LDUB`, `LDUH`). Pour l'instruction de rangement en mémoire (`STB`, `STH`), ce n'est pas le cas. Pourquoi ? Dans le processeur 68000 on n'a pas le problème. Pourquoi ?

E12.4 : Calcul d'un modulo
X étant un entier naturel, écrire une ou plusieurs instructions SPARC (ou de toute autre machine) permettant de calculer $X \bmod 256$ et généralement $X \bmod 2^n$.

E12.5 : Découpage d'un entier en octect
X étant un entier codé sur 32 bits (4 octets), O1, O2, O3 et O4 désignant 4 emplacements mémoire de 1 octet pas nécessairement contigus, écrire une séquence d'instructions en langage d'assemblage de n'importe quelle machine permettant de ranger les octets $X_{31..24}$, $X_{23..16}$, $X_{15..8}$ et $X_{7..0}$ respectivement aux adresses O1, O2, O3 et O4.

E12.6 : Addition double longueur
Se convaincre que `N, C, V` ont un sens après la suite d'instructions `ADDcc, r1, r2, r3` ; `ADDXcc r4, r5, r6`, par rapport aux entiers 64 bits codés respectivement dans les couples de registres (`r3, r6`), (`r2, r5`) et (`r1, r4`). En particulier comprendre que tout marche bien pour le complément à 2, bien qu'on ne transmette que `C` dans `ADDXcc`. `Z` n'a plus de sens, en revanche. Pour comprendre pourquoi, étudier le problème suivant : comment tester que la somme, réalisée en deux instructions, de deux entiers 64 bits occupant chacun 2 registres, est nulle ?

E12.7 : Expressions booléennes associées aux branchements usuels
A et B sont deux entiers représentés dans les 2 registres `%l0` et `%l1` du SPARC. Après l'exécution de l'instruction `subcc %l0, %l1, %g0` quelles doivent être les valeurs de `Z, N, C, V` pour que $A \leq B$ si $A, B \in \mathcal{Z}$? Même question pour que $A \leq B$ si $A, B \in \mathcal{N}$?

Noter que $A \leq B \Longleftrightarrow A < B$ ou $A = B$; pour $A < B$, envisager les 2 cas : $A - B$ est calculable ou l'opération provoque un débordement. Retrouver la formule : Z or (N xor V).

Etudier de même les formules associées à toutes les conditions de branchements.

E12.8 : Plus Grand Commun Diviseur
Donner en langage d'assemblage SPARC ou 68000 une séquence d'instructions permettant de calculer le pgcd de deux entiers. S'inspirer de la machine séquentielle à actions décrite dans le paragraphe 1.5 (Cf. Figure 12.5). Supposer que les valeurs initiales `A0` et `B0` sont dans deux registres. Pour traduire la boucle d'itération, s'inspirer de l'exemple traité dans le paragraphe 4. du présent chapitre.

E12.9 : Observation de code produit
On donne ci-dessous un programme en C et le programme en assembleur SPARC produit par le compilateur `gcc`. Analyser le code produit pour y retrouver les structures de contrôle de l'algorithme décrit en C.

```
main () {
   int  i ; int  T[10] ;
   i = 0 ;
   while (i < 10)
      { T[i] = 2*i + 1 ; i = i + 1 ; }
}
```

La structure de procédure du programme C donne les lignes 3, 25 et 26 (Cf. Chapitre 13, paragraphe 3.5).

```
 1  .text                          16      mov %o0,%o3
 2  main:                          17      sll %o3,1,%o4
 3      save %sp,-104,%sp          18      add %o4,1,%o3
 4      mov 0,%o0                  19      st %o3,[%o1+%o2]
 5  .LL2:                          20      add %o0,1,%o0
 6      cmp %o0,9                  21      b .LL2
 7      ble .LL4                   22      nop
 8      nop                        23  .LL3:
 9      b .LL3                     24  .LL1:
10      nop                        25      ret
11  .LL4:                          26      restore
12      mov %o0,%o2                27
13      sll %o2,2,%o1              28  .data
14      sethi %hi(T),%o3           29    T: .skip 40
15      or %o3,%lo(T),%o2          30
```

Chapitre 13

Traduction des langages à structure de blocs en langage d'assemblage

Nous nous intéressons ici au problème de la traduction d'un langage à structure de *blocs* en langage machine ou, de manière équivalente, en langage d'assemblage. Le petit langage d'actions présenté au chapitre 4 est un bon représentant de langage impératif à structure de blocs, qui sont ici les actions et les fonctions paramétrées, munies de lexiques locaux.

En étudiant la traduction de ce petit langage nous rencontrons les problèmes communs posés par la traduction d'un langage impératif comme Pascal, Ada, C, C++, Java ... La traduction des constructions de haut niveau de ces langages (objets de C++ ou Java, généricité d'Ada, etc.) ou la traduction des langages non impératifs, pose de nouveaux problèmes, qui dépassent le cadre de cet ouvrage. La plupart du temps, les langages dont la structure est très éloignée d'un style impératif comme celui présenté ici, sont traduits en langage d'assemblage en passant par une étape intermédiaire (un programme C par exemple).

Parmi toutes les méthodes à mettre en oeuvre pour traduire un programme du langage d'actions en langage machine, nous avons déjà étudié certains points :

- Au chapitre 4 nous avons montré comment représenter les types complexes en mémoire, et comment transformer les affectations à des objets complexes en suite de transferts mémoire de taille fixée.
- Indépendamment, nous avons montré au chapitre 5 comment traduire les structures de contrôle en états et transitions d'une machine séquentielle à actions, et comment définir le lexique de cette machine.
- Au chapitre 12 nous avons défini un *lexique restreint* pour machine séquentielle à actions, qui correspond à la structure d'un langage machine type : les tests doivent être uniquement binaires, et les prédicats restreints à un ensemble de fonctions prédéfinies sur des variables Z, N, C et V qui

modélisent les indicateurs arithmétiques. Les variables du lexique restreint représentent la mémoire (le tableau MEM) et les registres.

Comment utiliser ces transformations pour obtenir le programme en langage d'assemblage correspondant à un programme du langage d'actions, et que reste-t-il à faire ? Deux types de problèmes se posent.

Tout d'abord, il faut décrire la transformation de chaque bloc (action ou fonction) d'un programme du langage d'actions en machine séquentielle *à lexique restreint*, pour se rapprocher d'un langage machine. En particulier, puisque le lexique restreint ne propose que le tableau MEM et quelques variables représentant les registres, il faut décider, pour chacune des variables définies dans le bloc, d'un emplacement mémoire qui lui correspond. La difficulté de l'*installation* en mémoire du lexique vient de l'existence des lexiques locaux de fonctions et des actions éventuellement récursives. Il faut d'autre part traduire toutes les structures conditionnelles et itératives en n'utilisant que des branchements binaires, portant des prédicats prédéfinis sur Z, N, C et V.

Supposons maintenant qu'on a su obtenir, pour chaque bloc (action ou fonction) du langage d'actions, une machine séquentielle à lexique restreint. Il ne reste plus qu'à traduire ces machines séquentielles en *textes* de programmes en langage d'assemblage. Cela demande de disposer de manière séquentielle les codages des différents états, avec les branchements adéquats ; il faut finalement coder les *appels* de procédures ou fonctions en utilisant les instructions d'appel de sous-programmes offertes par le langage machine considéré.

> *Nous étudions tout d'abord dans le paragraphe 1. le cas des programmes à un seul bloc : installation du lexique en mémoire et obtention d'une machine séquentielle à lexique restreint, production du texte en langage d'assemblage. Le paragraphe 2. présente les problèmes spécifiques au codage des programmes à plusieurs blocs : problème du lexique local, des procédures ou fonctions récursives, passage de paramètres et contenu de la mémoire lors de l'exécution. En se basant sur les principes étudiés dans ces deux premiers paragraphes, il est possible de développer au paragraphe 3. des solutions globales pour deux types de langages machine : un langage machine à structure de pile explicite, type 68000 ; un langage machine à fenêtres de registres, type* SPARC.

1. Cas des programmes à un seul bloc

Nous avons étudié au chapitre 5 la traduction des programmes du langage d'actions en machines séquentielles avec actions générales. Il ne reste plus qu'une étape pour atteindre des machines séquentielles à lexique restreint : il faut installer toutes les variables dans la mémoire ou les registres (paragraphe 1.1), puis transformer les branchements généraux en branche-

ments binaires faisant intervenir des conditions prédéfinies sur les indicateurs arithmétiques (paragraphe 1.3).

A partir des machines séquentielles à lexique restreint, on obtient facilement des textes de programmes en langage d'assemblage (paragraphe 1.4).

1.1 Installation du lexique en mémoire

1.1.1 Le problème

Le langage d'actions présenté offre la notion de *lexique* : on déclare des variables en les *nommant*. La déclaration donne le type, et permet donc de connaître la taille nécessaire à la représentation en mémoire des valeurs de cette variable. Dans tout ce qui précède, nous avons étudié comment représenter en mémoire une donnée de type quelconque, en supposant qu'il y a de la place libre, en quantité suffisante, à partir d'une certaine adresse donnée.

> **Remarque :** Notons que nous ne parlons pas ici d'*allocation dynamique*, d'allocation de variables à la demande du programme par les actions Allouer et Libérer comme défini au paragraphe 4. du chapitre 4.

Nous ne nous sommes pas interrogés sur le mécanisme d'*allocation* de la mémoire, c'est-à-dire sur la manière de choisir une adresse pour chaque variable, en assurant que deux variables distinctes sont installées dans des portions disjointes de la mémoire globale. Plus précisément, cette contrainte vaut pour les variables dont les *périodes de vie* (on dit aussi *durée* de vie) ne sont pas disjointes (Cf. Paragraphes 1.1.2, 2.1.1 et 2.6).

Nous montrons ici comment remplacer systématiquement les variables par des zones du tableau MEM : il faut d'abord *choisir* la position de chaque variable, ce qui donne une adresse a ; on remplace ensuite toute occurrence du nom de variable dans le programme par MEM[a]. On peut alors oublier le lexique. On obtient ainsi de manière systématique l'algorithme qu'on aurait pu obtenir à la main en installant soi-même toutes les variables nécessaires dans le tableau MEM. Les programmes en deviennent bien sûr illisibles ; cette transformation est d'habitude le travail du compilateur, pas celui du programmeur. La transformation de programmes que nous proposons ici peut être vue comme la première étape de gestion de ce que l'on appelle couramment *système à l'exécution* dans les ouvrages traitant de compilation (voir par exemple [CGV80, WM94]).

1.1.2 Solution pour le lexique d'un programme à un seul bloc

Pour choisir la position des variables on procède de manière séquentielle, par exemple dans l'ordre du lexique, en ménageant des espaces perdus entre les variables pour satisfaire aux éventuelles contraintes d'alignement. La taille nécessaire pour l'installation de toutes les variables du lexique en mémoire est donc supérieure ou égale à la somme des tailles nécessaires pour les variables.

On parlera par la suite de *taille du lexique*. Attention, cette taille peut dépendre de l'ordre de déclaration des variables, qui a une influence sur la position et la taille des trous nécessaires aux contraintes d'alignement. Noter que, dans le cas d'un programme à un seul bloc, les variables ont toutes la même durée de vie, qui est celle du programme lui-même.

Dans l'exemple ci-dessous, la mémoire est un tableau d'octets, et on dispose des affectations de tailles 1, 2 et 4 :

N : l'entier 42
Entier32s : le type entier dans $[-2^{32-1}, 2^{32-1} - 1]$
Entier16ns : le type entier dans $[0, 2^{16} - 1]$
Structure : le type $<$ x : un Entier32s, y : un Entier16ns$>$.
c1, c2 : des caractères ; a : un Entier32s ; b : un Entier16ns
T : un tableau sur [0..N-1] de Structures

Les variables à installer en mémoire sont c1, c2, a, b et T. Commençons à une adresse A multiple de 4 pour c1 ; c2 peut être placé juste à côté, à l'adresse $A + 1$; a doit être placé un peu plus loin, à l'adresse $A + 4$; b peut être placé à l'adresse $A + 8$, sans perte de place ; T doit démarrer à l'adresse multiple de 4 qui suit $A + 8 + 2$, c'est-à-dire $A + 12$. Noter que 2 octets sont perdus entre b et T. Chaque élément de T occupe 8 octets. T[7].y est à l'adresse $A + 12 + (7 \times 8) + 4$. On traduit alors l'algorithme

c1 $\longleftarrow$ c2 ; T[7].y $\longleftarrow$ b

en :

delta_c1 : l'entier 0 ; delta_c2 : l'entier 1 ; delta_a : l'entier 4 ; delta_b : l'entier 8
delta_T : l'entier 12 ; delta_x : l'entier 0 ; delta_y : l'entier 4
taille_structure : l'entier 8
MEM[A+delta_c1] $\longleftarrow_1$ MEM[A+delta_c2]
MEM[A+delta_T+7*taille_structure + delta_y] $\longleftarrow_2$ MEM[A+delta_b]

1.2 Traduction des opérations de base sur les types de données

Nous avons introduit dans le langage d'actions du chapitre 4 des opérations sur les types de données de base : entiers, booléens, caractères. Pour les types structurés, la seule opération globale que nous ayons envisagée est l'affectation, et nous avons déjà montré comment l'exprimer en termes de transferts mémoire élémentaires (chapitre 4, paragraphe 3.).

Le lexique restreint qui représente un langage machine type ne propose que les opérations arithmétiques, logiques ou structurelles (Cf. Chapitre 12) disponibles dans un processeur usuel. Il faut donc exprimer toutes les opérations sur les types de base en termes de ces opérations élémentaires.

1.2.1 Cas des opérations sur les caractères

Toutes les opérations sur les caractères sont traduites en opérations arithmétiques ou logiques par l'intermédiaire du code ASCII. Nous signalions au chapitre 4, paragraphe 1.3.2, que le code ASCII est conçu de manière à faciliter cette traduction.

1.2.2 Cas des opérations booléennes

Il y a trois types d'utilisation des booléens : les opérations internes de l'algèbre de Boole étudiée au chapitre 2 (négation, conjonction, disjonction, etc.) ; les opérations de comparaison sur des types quelconques, qui *produisent* des booléens ; les conditions de structures conditionnelles ou itératives. Ces trois types d'utilisations interfèrent librement comme dans :

X, Y : des entiers ; B, B' : des booléens
B $\longleftarrow$ X $<$ Y et Y $\geq$ 0
si (non B ou B') alors ... sinon ...

Il faut choisir un codage des booléens compatible avec ces trois types d'utilisation, et susceptible d'être manipulé efficacement en langage machine. Typiquement, les processeurs offrent des opérations booléennes bit à bit (Cf. Chapitre 12, paragraphe 1.4.1) qu'on peut utiliser pour coder les opérations booléennes. En revanche il n'existe pas, en général, d'instruction de comparaison arithmétique de deux registres, avec résultat booléen dans un troisième, qui permettrait de coder simplement B $\longleftarrow$ X $<$ Y. Le résultat booléen des comparaisons est à recupérer dans les indicateurs arithmétiques. D'autre part, comme le mot d'état du processeur n'est en général pas accessible en lecture explicite, on doit coder B $\longleftarrow$ X $<$ Y comme on coderait :

si X $<$ Y alors B $\longleftarrow$ vrai sinon B $\longleftarrow$ faux

On retrouve donc le problème général de codage des structures conditionnelles.

1.2.3 Cas des entrées/sorties

Nous avons introduit dans le langage d'actions des actions Lire et Ecrire génériques (c'est-à-dire valables pour tous types). La manipulation détaillée des périphériques d'entrée/sortie ne fait pas partie, en général, du codage des programmes écrits en langage de haut niveau. Nous verrons en détail au chapitre 16 comment sont réalisées les communications entre le processeur et des périphériques comme le clavier et l'écran. Les programmes nécessaires, souvent écrits directement en assembleur, font partie du *logiciel de base* fourni avec un ordinateur ; nous définissons au chapitre 17 l'interface entre ces programmes de bas niveau rangés dans une *bibliothèque* et les programmes en langage d'assemblage produits par les compilateurs de langages de haut niveau. On trouve entre autres dans le logiciel de base usuel les primitives de manipulation des fichiers, telles que nous les définissons au chapitre 19.

Pour un compilateur, le codage d'une entrée/sortie est complètement simi-

laire à l'appel d'une procédure paramétrée. Simplement, cette procédure ayant été écrite directement en assembleur, ou produite indépendamment par un compilateur, il faut adopter les mêmes conventions de passages de paramètres et d'appel de sous-programme, dans les deux contextes. C'est d'ailleurs le problème général de la *compilation séparée* que nous étudions au chapitre 18.

Le seul travail du compilateur consiste à traduire une instruction de la forme Lire (X), où X est une variable de type quelconque, en un *ensemble* d'opérations de lecture élémentaires disponibles dans la bibliothèque. On ne peut pas supposer, en effet, que la bibliothèque d'entrées/sorties standard fournit une primitive de lecture pour tout type susceptible d'être défini dans un programme utilisateur.

La plupart des langages de programmation n'autorisent les primitives Lire et Ecrire que sur les types numériques, les caractères et les chaînes de caractères. Pour les types numériques cela suppose une convention de notation (décimal pour les entiers, norme IEEE... pour les flottants). Ainsi en Pascal est-il impossible de lire un booléen, défini comme un type énuméré, par manque de convention sur la notation des booléens. En Ada le compilateur traduit les lectures de booléens en lectures de chaînes de caractères, suivies de *conversions* des chaînes valides `true` et `false` en représentation interne des booléens.

Dans les exemples de ce chapitre, nous n'utilisons Lire et Ecrire que pour des entiers.

1.3 Traduction des conditions générales en conditions des machines restreintes

1.3.1 Codage de conditionnelles n-aires, type selon

La plupart des langages machine disposent de branchements uniquement *binaires*. Les structures conditionnelles de la forme selon du langage d'actions utilisé dans cet ouvrage, ou bien les structures `switch` de C, `case` de Pascal et Ada, sont donc traduites en séquences de conditionnelles binaires.

1.3.2 Codage de conditions booléennes complexes

Quel que soit l'ensemble des prédicats fonctions de N, Z, C et V disponibles dans le langage machine cible, il est toujours possible d'écrire dans le langage d'actions des conditions qui ne se traduisent pas par un seul prédicat.

On a donc le même problème pour coder : si $X < Y$ et $Y > 0$ grâce aux 16 prédicats usuels présentés au chapitre 12, figure 12.4, que pour coder si $X \leq Y$ grâce aux 8 prédicats du processeur 6502. La solution générale consiste à enchaîner des conditionnelles. On traduit donc

si $X < Y$ et $Y > 0$ alors A1 sinon A2

en :

si $X < Y$ alors (si $Y > 0$ alors A1 sinon A2) sinon A2.

```
A1
tantque C1
   A2
   si C2 alors A3 sinon A4
   A5
A6
```

```
A1 :..       ! codage de l'action A1
C1 :..       ! codage de la condition C1
   Bicc A6   ! branchement sur non C1
A2 :..       ! codage de l'action A2
C2 :..       ! codage de la condition C2
   Bicc A4   ! branchement sur non C2
A3 :..       ! codage de l'action A3
   ba A5     ! branchement inconditionnel
A4 :..       ! codage de l'action A4
A5 :..       ! codage de l'action A5
   ba C1     ! branchement inconditionnel
A6 :..       ! codage de l'action A6
```

FIG. 13.1 – Séquentialisation des codages en langage d'assemblage des états d'une machine séquentielle à actions et lexique restreint.

1.4 Traduction des machines séquentielles à lexique restreint en langage d'assemblage

Grâce aux transformations de programmes suggérées ci-dessus, on est capable d'obtenir un programme du langage d'actions dans lequel : 1) toutes les données sont en mémoire ou dans des registres ; 2) on n'a plus que des structures conditionnelles binaires (des si ... alors ... sinon ...) et des itérations : 3) toutes les conditions sont *élémentaires*, au sens où on peut en obtenir la valeur booléenne par une ou plusieurs instructions de calcul, suivies d'*une* consultation du mot d'état. Ainsi X+2*Y $<$ 4 est élémentaire, alors que X $<$ Y et Y $>$ 0 ne l'est pas. Cette forme de programme donne directement une machine séquentielle à lexique restreint, dans laquelle certains états correspondent au calcul des conditions élémentaires.

La technique consiste ensuite à coder chaque état par une séquence d'instructions du langage d'assemblage, à laquelle on associe une étiquette de début. Il ne reste plus qu'à disposer ces différentes séquences les unes à la suite des autres, en ajoutant les branchements nécessaires.

Cette technique est illustrée par l'exemple des figures 13.1 et 13.2.

2. Cas des programmes à plusieurs blocs

La difficulté de la traduction en langage d'assemblage des programmes à structure de blocs provient de deux aspects de ces programmes : les lexiques locaux et le passage de paramètres, dans le cas le plus général où les actions et fonctions introduites peuvent être récursives. Nous étudions les problèmes liés aux lexiques locaux et aux paramètres dans les paragraphes 2.1 et 2.2 ci-dessous, avant de les résoudre par une gestion de la mémoire en *pile*, aux

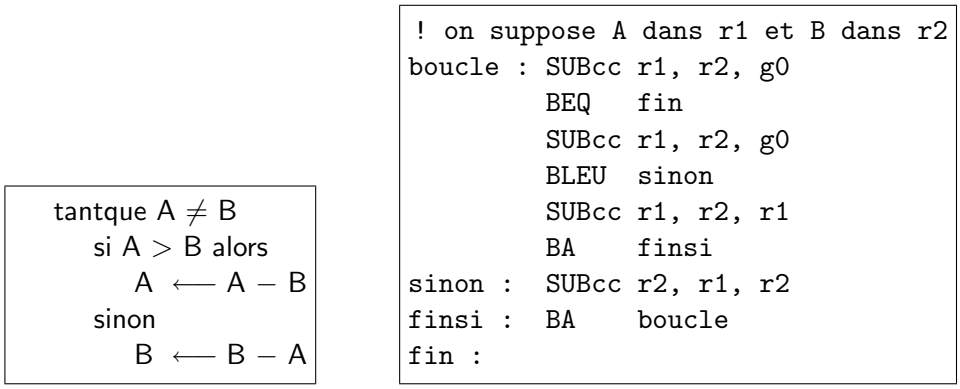


FIG. 13.2 – Séquentialisation des codages en langage d'assemblage des états d'une machine séquentielle à actions et lexique restreint : exemple du pgcd, Cf. Chapitre 12, figure 12.5. Le programme en langage d'assemblage est obtenu de manière systématique ; il peut ensuite être simplifié de plusieurs façons : suppression de la deuxième comparaison `SUBcc r1, r2, g0` ; remplacement du `BA finsi` par `BA boucle` directement.

paragraphes 2.3 et 2.4. Enfin le paragraphe 2.5 introduit le *lien dynamique* des compilateurs, et le paragraphe 2.6 résume l'occupation de la mémoire lors de l'exécution d'un ensemble de procédures et fonctions paramétrées.

Dans le cas où une procédure A appelle une procédure B, nous appelons *contexte appelant* l'ensemble des variables de A et *contexte appelé* l'ensemble des variables de B.

2.1 Installation en mémoire des lexiques locaux

2.1.1 Cas des lexiques locaux sans récursivité

Dans le cas d'un ensemble d'actions et fonctions sans récursivité (même croisée), on peut reprendre l'approche suivie précédemment pour le lexique d'un programme à un seul bloc : on installe en mémoire toutes les variables globales, puis toutes celles de la première fonction ou action, à la suite, puis toutes celles de la deuxième fonction ou action, etc. Si les lexiques locaux de deux actions ou fonctions A1 et A2 contiennent tous deux le nom i, avec éventuellement des types différents, alors ce nom correspond à deux emplacements distincts de la mémoire, éventuellement de tailles différentes. En traduisant l'algorithme de A1 on utilise l'un des emplacements, et en traduisant l'algorithme de A2 on utilise l'autre.

Si deux actions ou fonctions A1 et A2 sont telles que A1 n'appelle jamais A2 et A2 n'appelle jamais A1, on se convainc aisément que ces deux blocs ne seront jamais actifs en même temps : on dit que leurs *durées de vie* sont disjointes. Dans ce cas leurs lexiques locaux peuvent occuper la *même* portion de la mémoire.

2.1.2 Cas des lexiques locaux avec récursivité

Lorsque les actions et fonctions peuvent être récursives, il n'est plus possible d'appliquer la même méthode. Observons pour nous en convaincre le programme donné à titre d'exemple figure 13.3, qui utilise une action récursive de calcul de la suite de Fibonacci (définie par : $x_0 = x_1 = 1$ puis $\forall n \geq 2 \quad x_n = x_{n-1} + x_{n-2}$). La figure 13.4 illustre le déroulement du calcul si l'entier lu dans le programme principal est 4.

Pour obtenir le résultat correct il faut disposer d'autant d'exemplaires de f1 et f2 (notés f1', f1'', etc.) que d'appels imbriqués. Il faut donc *un espace mémoire* pour la variable locale f1 (resp. f2) de CalculFibo *pour chaque appel* de cette action. Or le nombre d'appels dépend de la variable d'entrée A ; sa valeur est obtenue par Lire et est donc par définition inconnue quand on est en train de modifier le texte du programme, indépendamment de toute exécution, comme c'est le cas ici. Il nous faut donc trouver un mécanisme permettant d'allouer une zone de mémoire à chaque entrée dans la procédure, zone qui devra être restituée avant la sortie de procédure.

2.2 Installation en mémoire des paramètres

Lorsqu'un programme comporte plusieurs blocs, il fait également usage de *paramètres* données ou résultats. Comme les lexiques locaux, le passage de paramètres demande de la mémoire. En effet, un passage de paramètres est un échange d'information entre le contexte *appelant* et le contexte *appelé*. Tout se passe comme si ces deux contextes avaient accès à un emplacement dans une portion de mémoire partagée gérée de manière très particulière : l'appelant peut y *écrire* une information à transmettre à l'appelé ; l'appelé y *lit* l'information. Le passage de paramètres nécessite donc une sorte de *boîte aux lettres*.

2.2.1 Conventions de passage des paramètres données ou résultats

Examinons tout d'abord comment se passe l'échange d'information entre *appelant* et *appelé*, lors d'un passage de paramètres, selon qu'il s'agit d'une *donnée* ou d'un *résultat*. Nous traitons ici le paramètre résultat avec le mode de passage par *référence*. Il existe d'autres modes de passage de paramètres dont les particularités sont expliquées dans les ouvrages traitant de compilation (voir par exemple [CGV80, WM94]) et dont nous ne parlons pas ici. Considérons l'algorithme de calcul de la somme de deux entiers, donné figure 13.5.

Le programme principal appelle CalculSomme en lui passant deux données dans les paramètres A et B, et en reçoit un résultat dans le paramètre R. Supposons pour l'instant que l'allocation mémoire soit résolue pour les paramètres (voir plus loin) et qu'on dispose de 3 emplacements mémoire d'adresses respectives aA, aB et aR pour le passage de paramètres. Par ailleurs les variables du lexique global sont placées aux adresses aX, aY et aZ. L'algorithme modifié est donné figure 13.6.

```
CalculFibo : une action
(la donnée n : un entier ≥ 0 ; le résultat f : un entier > 0)
    { état final : f = x_n }
    lexique local :
        f1, f2 : des entiers > 0
    algorithme :
        si n = 0 ou n = 1 alors f ⟵ 1
        sinon
            CalculFibo (n − 1, f1) ; CalculFibo (n − 2, f2)
            f ⟵ f1 + f2
lexique du programme principal :
    A, B : des entiers ≥ 0
algorithme du programme principal :
    Lire (A) ; CalculFibo (A, B) ; Ecrire (B)
```

FIG. 13.3 – Algorithme récursif de calcul de la suite de Fibonacci

```
1. CalculFibo (4, B)
        1.1 CalculFibo (3, f1)
                1.1.1 CalculFibo (2, f1')
                        1.1.1.1 CalculFibo (1, f1")
                                f1" ⟵ 1
                        1.1.1.2 CalculFibo (0, f2")
                                f2" ⟵ 1
                        1.1.1.3 f1' ⟵ f1" + f2"
                1.1.2 CalculFibo (1, f2')
                        f2' ⟵ 1
                1.1.3 f1 ⟵ f1'+ f2'
        1.2 CalculFibo (2, f2)
                1.2.1 CalculFibo (1, f1')
                        f1' ⟵ 1
                1.2.2 CalculFibo (0, f2')
                        f2' ⟵ 1
                1.2.3 f2 ⟵ f1' + f2'
        1.3 B ⟵ f1+f2
```

FIG. 13.4 – Déroulement du calcul de la suite de Fibonacci, pour l'entrée 4.
f1" ⟵ 1 ; f2" ⟵ 1 ; f1' ⟵ f1" + f2" { = *2* } ;
f2' ⟵ 1 ; f1 ⟵ f1'+ f2' { = *3* } ;
f1' ⟵ 1 ; f2' ⟵ 1 ; f2 ⟵ f1' + f2' { = *2* } ;
B ⟵ f1+f2 { = *5* } ;

```
CalculSomme (les données a, b : deux entiers ; le résultat r : un entier) :
        r ⟵ a+b
lexique du programme principal
    X, Y, Z : trois entiers
algorithme du programme principal
    Lire (X) ; Lire (Y)
    CalculSomme (X+1, Y−2, Z)
    Ecrire (Z)
```

FIG. 13.5 – Algorithme de calcul de la somme de deux entiers

```
CalculSomme :
    MEM[MEM[aR]] ⟵ MEM[aA] + MEM[aB]
algorithme du programme principal
    ... { traduction de Lire (X) ; Lire (Y) }
    MEM[aA] ⟵ 4 MEM[aX] + 1
    MEM[aB] ⟵ 4 MEM[aY] − 2
    MEM[aR] ⟵ 4 aZ
    CalculSomme
    ... { traduction de Ecrire (Z) }
```

FIG. 13.6 – Interprétation des natures de paramètres

Passage des données par *valeur* : Les valeurs des expressions qui constituent les paramètres données effectifs doivent être calculées puis placées en mémoire par l'appelant, à un endroit connu de l'appelé qui les lira. Ainsi on calcule la valeur de X+1, c'est-à-dire MEM[aX] + 1, et on la recopie dans MEM[aA].

Passage des résultats par *référence* : Pour les paramètres résultats, passer leur valeur n'a aucun intérêt. Par définition un *résultat* est produit par l'action appelée. Quand on écrit CalculSomme (X+1, Y−2, Z), on s'attend à ce que la valeur de la variable Z soit modifiée. Pour cela il faut que l'action appelée soit capable d'affecter une nouvelle valeur à la variable Z. Si l'action CalculSomme était toujours appelée depuis le programme principal, avec comme troisième paramètre la variable Z, il suffirait d'écrire dans le corps de l'action : MEM[aZ] ⟵ .... Mais elle peut être appelée à divers endroits du programme, avec des paramètres différents. Le corps de l'action doit donc être capable d'écrire dans la variable qu'on lui indique. Pour assurer ce fonctionnement, l'*adresse* de la variable qui constitue le paramètre résultat effectif est fournie à l'appelée ; elle est placée en mémoire par l'appelant, à un endroit connu de l'appelée qui la lira. On écrit donc : MEM[aR] ⟵ 4 aZ. L'appelée peut alors récupérer cette adresse dans MEM[aR], et s'en servir pour écrire dans la bonne variable, par *indirection* : MEM[MEM[aR]] ⟵ .... On retrouve

la contrainte énoncée au paragraphe 1.6 du chapitre 4 : les paramètres effectifs résultats doivent être des expressions qui pourraient figurer en partie gauche d'affectation, c'est-à-dire désigner des emplacements mémoire.

Si l'on définit la variable :

T : un tableau sur [0..N] de < y : un caractère ; x : un entier >

l'expression T[4].x peut-être utilisée comme paramètre résultat de CalculSomme. Dans ce cas le passage de paramètre consiste à écrire, dans le programme principal : MEM[aR] $\longleftarrow_4$ aT + 4*8 + 4, où aT désigne l'adresse de début du tableau T. Le corps de CalculSomme est inchangé.

2.2.2 Installation en mémoire des boîtes aux lettres

Une fois adoptées ces conventions de passage des paramètres données par valeur et résultat par adresse, il reste à étudier la manière d'obtenir aA, aB et aR : c'est le problème de l'allocation mémoire pour les paramètres.

Cas sans récursivité : Il faut prévoir une zone de mémoire pour chaque paramètre de F1 vers F2, pour chaque couple d'actions ou fonctions F1, F2 telles qu'un appel de F2 apparaît quelque part dans le corps de F1. Un examen du texte complet du programme, indépendamment de toute exécution, permet de repérer qui appelle qui, et la place nécessaire pour la liste de paramètres, dans chaque cas. On choisit alors les adresses dans MEM, par exemple à la suite de tous les emplacements alloués aux variables globales. Il suffit de se tenir à ces choix lorsque l'on transforme les algorithmes des appelants et des appelés, comme nous l'avons fait pour CalculSomme ci-dessus.

Cas avec récursivité : Dans le cas avec récursivité, on retombe sur le problème évoqué pour les variables des lexiques locaux : il faut un emplacement dédié aux paramètres, *pour chaque appel* de F1 qui appelle F2, à l'exécution. Le choix des adresses des paramètres ne peut donc se faire statiquement, c'est-à-dire au moment de la transformation du programme.

2.3 Allocation dynamique de mémoire pour les lexiques locaux

Il apparaît la nécessité de gérer *dynamiquement* l'association d'emplacements en mémoire aux variables des lexiques locaux. Une solution consiste à utiliser les primitives d'allocation dite *dynamique* de mémoire étudiées au chapitre 4, paragraphe 4.

Supposons que chaque appel de CalculFibo utilise Allouer et Libérer pour réserver momentanément la mémoire nécessaire à l'installation de ses variables locales f1 et f2. Notons taille_2_entiers le nombre d'octets nécessaires à ces deux entiers. Il nous faut encore une variable locale p qui donne l'adresse de la zone allouée. On suppose qu'il y a toujours de la place disponible. Cela donne le programme de la figure 13.7. Noter que l'allocation mémoire pour les

différentes variables du lexique local suit la même démarche que celle présentée au paragraphe 1.1.2. Une fois obtenue une adresse de base p par allocation dynamique, on dispose les variables les unes après les autres en ménageant des espaces éventuels pour tenir compte des contraintes d'alignement.

Cette transformation n'a pas encore permis d'éliminer le lexique local : on a remplacé les variables d'origine par une seule : p, mais il en faut toujours autant d'exemplaires que d'appels de CalculFibo.

Toutefois, on peut poursuivre la transformation du programme. En effet, on remarque que, vue la structure des appels d'actions et fonctions, la *dernière* zone allouée est toujours la *première* libérée (autrement dit, les appels sont *bien parenthésés* : on ne peut pas successivement *entrer dans A* ; *entrer dans B* ; *sortir de A*, *sortir de B*). On profite donc de cette situation très particulière pour ne pas utiliser un mécanisme général d'allocation et libération d'espaces mémoire, dont la complexité provient justement de la gestion des trous qui apparaissent si l'on ne libère pas toujours le dernier alloué.

On propose de réaliser l'allocation mémoire par les algorithmes de la figure 13.9. Le corps des actions est très court ; si l'on suppose de plus qu'il y a toujours de la place, on peut remplacer tout appel Allouer (p, n) par pp $\longleftarrow$ pp − n ; p $\longleftarrow$ pp et tout appel Libérer (p, n) par pp $\longleftarrow$ pp + n. On peut ensuite éliminer la variable locale p et ne garder que la variable globale pp. Cela donne l'algorithme de la figure 13.8. On dit que la mémoire est gérée *en pile*, pour rendre compte du fait que les allocations se font selon un ordre *dernier alloué/premier libéré* (Cf. Chapitre 4, paragraphe 5.).

2.4 Allocation dynamique de mémoire pour les paramètres

Pour l'allocation mémoire nécessaire aux paramètres, il faut prévoir de même des appels aux actions Allouer et Libérer. L'allocation de mémoire pour les paramètres a la même propriété que celle des variables locales : on libère toujours les derniers alloués. On applique donc la même simplification qui consiste à allouer et libérer de la mémoire grâce aux procédures de la figure 13.9. L'allocation de mémoire pour le passage de paramètres se fait dans l'*appelant*, juste avant l'appel ; la libération se fait juste après l'appel, toujours dans l'appelant.

En appliquant toutes les transformations étudiées jusque là (dont le mécanisme de passage de paramètre *résultat*), on obtient l'algorithme donné figure 13.10. La figure 13.11 illustre l'évolution du contenu de la mémoire lors de l'exécution du programme qui appelle CalculFibo(4, ...).

Noter que la variable pp est gérée grâce à des opérations parfaitement symétriques : pp $\longleftarrow$ pp − 8 (pour faire de la place aux variables locales) en entrant dans CalculFibo et pp $\longleftarrow$ pp + 8 en en sortant ; pp $\longleftarrow$ pp − 8 juste avant l'appel récursif de Calcul Fibo (pour tenir compte de la place occupée par les paramètres), pp $\longleftarrow$ pp + 8 juste après. Cette forme de code garantit l'invariant suivant : l'état de la mémoire (en particulier la position de pp) est

```
CalculFibo : une action (la donnée n : un entier ≥ 0 ; le résultat f : un entier > 0)
    p : un pointeur de caractère                          { variable LOCALE }
    { lexique local : f1 sera en MEM[p+0] et f2 en MEM[p+4]  }
    algorithme :
        Allouer (p, taille_2_entiers)
        si n = 0 ou n = 1 alors f ⟵ 1
        sinon
            CalculFibo (n − 1, MEM[p+0])
            CalculFibo (n − 2, MEM[p+4])
            f ⟵ MEM[p+0]+MEM[p+4]
        Libérer (p, taille_2_entiers)
lexique du programme principal :
    A, B : des entiers ≥ 0
algorithme du programme principal :
    Lire (A) ; CalculFibo (A, B) ; Ecrire (B)
```

FIG. 13.7 – Installation du lexique local de CalculFibo en mémoire

```
CalculFibo : une action (la donnée n : un entier ≥ 0 ; le résultat f : un entier > 0)
    algorithme :
        { Réservation de place pour les deux entiers du lexique local, qui rend
        disponibles les deux emplacements MEM[pp+0] et MEM[pp+4]. Pour
        simplifier, on ne fait pas le test de débordement (Cf. Figure 13.9) }
        pp ⟵ pp − taille_2_entiers
            { Corps de l'action proprement dite }
            si n = 0 ou n = 1 alors f ⟵ 1
            sinon
                CalculFibo (n − 1, MEM[pp+0])
                CalculFibo (n − 2, MEM[pp+4])
                f ⟵ MEM[pp+0]+MEM[pp+4]
        { Libération de la place occupée par le lexique local }
        pp ⟵ pp + taille_2_entiers

    lexique du programme principal :
        A, B : des entiers ≥ 0 ; pp : un pointeur

    algorithme du programme principal :
        Initialiser
        Lire (A)
        CalculFibo (A, B)
        Ecrire (B)
```

FIG. 13.8 – Simplification de l'allocation mémoire

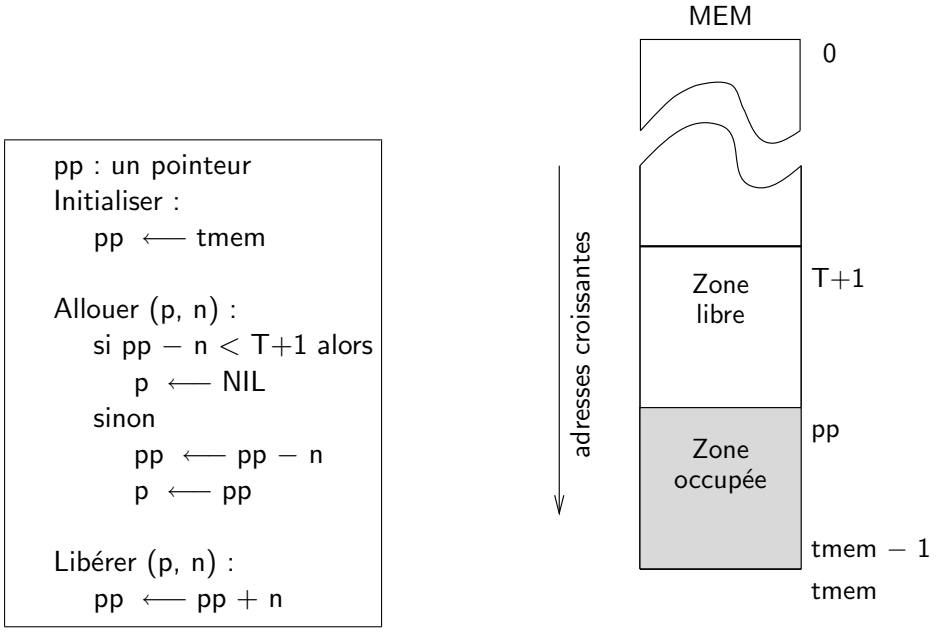


FIG. 13.9 – Gestion mémoire simplifiée : pour allouer une nouvelle zone de taille n, on déplace le pointeur pp de n octets vers les adresses décroissantes. La zone ainsi rendue disponible est entre les adresses pp incluse et pp+n exclue.

le même à l'entrée dans un bloc et à la sortie.

2.5 Repérage des objets locaux par rapport à la base de l'environnement : lien dynamique

Nous remarquons dans l'exemple CalculFibo (figure 13.10) que la position relative à pp des objets présents en mémoire est modifiée lorsque pp bouge. Ainsi, lors de l'entrée dans CalculFibo, les paramètres sont accessibles aux adresses MEM[pp+4] et MEM[pp+0]. Dès que l'on diminue pp de 8, de manière à ménager de la place pour les variables locales f1 et f2 de la procédure CalculFibo d'origine, le repérage des paramètres devient : MEM[pp+4+8] et MEM[pp+0+8]. Rien de bien difficile à calculer, mais le nouveau programme de CalculFibo devient vite illisible sans dessin.

Le problème vient du fait que les objets placés en mémoire (les variables locales et les paramètres) sont repérés par rapport à une unique adresse pp, elle-même destinée à évoluer.

L'idée du *lien dynamique* que l'on trouve en compilation est très simple : au lieu de repérer les objets locaux d'un bloc par rapport à l'adresse *du haut*, qui risque d'être modifiée, il suffit de les repérer par rapport à l'adresse *du bas*, qui ne bouge pas pendant toute la durée de vie du bloc. On introduit une adresse supplémentaire pb, dite *pointeur de base de l'environnement* (*frame pointer* en anglais), destinée à pointer sur la base de l'environnement du bloc

```
CalculFibo : une action
   { paramètres : on récupère la valeur de la donnée n en MEM[pp+4] ; on
   écrit la valeur du résultat f à l'adresse indiquée dans MEM[pp+0]. }
   pp ⟵ pp − 8
   { Attention, maintenant, n est en MEM[pp+4+8] et f en MEM[pp+0+8]
   Lexique local : f1 sera en MEM[pp+0] et f2 en MEM[pp+4] }

   si MEM[pp+4+8] = 0 ou MEM[pp+4+8] = 1 alors
      MEM[MEM[pp+0+8]] ⟵ 1
   sinon
      { Premier appel : (Point i) }
         MEM[pp−4] ⟵ 4 MEM[pp+4+8] − 1
         MEM[pp−8] ⟵ 4 p+0
         pp ⟵ pp − 8 { place des paramètres - (Point ii) }
         CalculFibo
         pp ⟵ pp + 8 { on ôte les paramètres }
      { Deuxième appel : }
         MEM[pp−4] ⟵ 4 MEM[pp+4+8] − 2
         MEM[pp−8] ⟵ 4 pp+4
         pp ⟵ pp − 8 { place des paramètres }
         CalculFibo
         pp ⟵ pp + 8 { on ôte les paramètres }
      MEM[MEM[pp+0+8]] ⟵ MEM[pp+0]+MEM[pp+4]
   pp ⟵ pp + 8

lexique du programme principal :
   pp : un pointeur { Unique variable globale restante }
algorithme du programme principal :
   Initialiser { Initialisation de la zone pile }
   pp ⟵ pp − 8 { Place nécessaire aux variables du lexique global }
   { A est en MEM[pp+0] et B en MEM[pp+4]. }
   { Traduction de l'appel Lire (A) : }
      MEM[pp−4] ⟵ 4 pp+0 { adresse de A empilée }
      pp ⟵ pp − 4 ; Lire ; pp ⟵ pp + 4 { valeur de A lue en pp+0 }
   { Appel de CalculFibo (A, B) : (point 1) }
      MEM[pp−4] ⟵ 4 MEM[pp+0] { paramètre donnée : valeur de A }
      MEM[pp−8] ⟵ 4 pp+4 { paramètre résultat : adresse de B }
      pp ⟵ pp − 8 { Allocation mémoire pour les paramètres }
      { (point 2) } CalculFibo
      pp ⟵ pp + 8 { Libération de la mémoire des paramètres }
   { Traduction de l'appel Ecrire (B) : }
      MEM[pp−4] ⟵ 4 MEM[pp+4] { valeur de B empilée }
      pp ⟵ pp − 4 ; Ecrire ; pp ⟵ pp + 4 { B écrit }
```

FIG. 13.10 – Elimination complète des lexiques et des paramètres dans CalculFibo (On a supposé taille_2_entiers = 8).

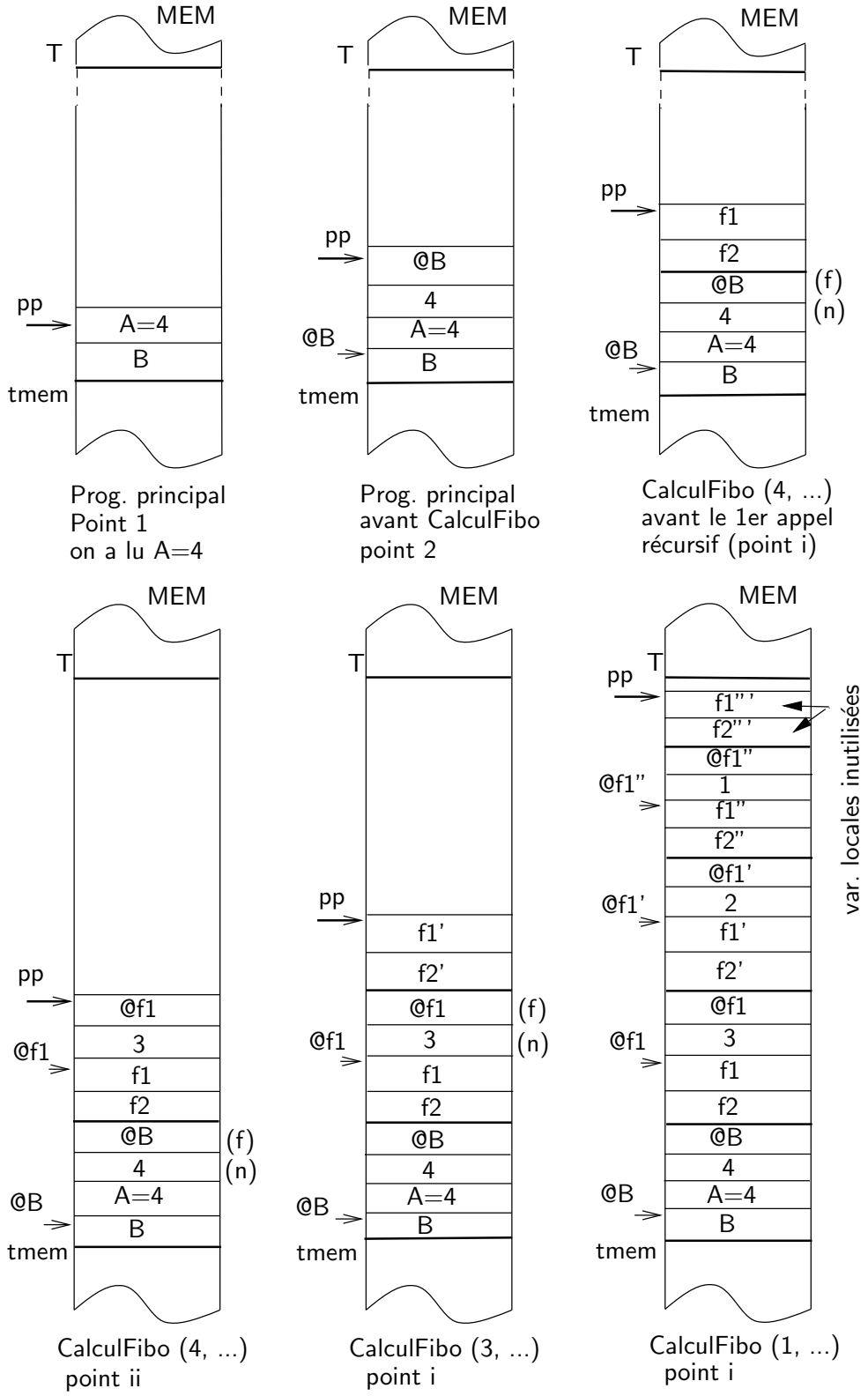


FIG. 13.11 – Contenu de la mémoire lors de l'exécution de CalculFibo

en cours d'exécution, c'est-à-dire juste sous les variables locales du bloc. Sur la figure 13.12-a, les pointeurs de base d'environnement sont figurés par des flèches en trait gras ; la notation @x est utilisée pour signifier l'adresse de x.

Lorsqu'on passe d'un bloc appelant à un bloc appelé, il suffit de placer la variable pb à la base de l'environnement du bloc appelé. Lorsque le bloc appelé se termine, il faut replacer pb à sa position antérieure, c'est-à-dire la base de l'environnement du bloc appelant. Contrairement à ce qui se passe pour le pointeur pp, il n'est pas toujours aisé de calculer l'ancienne position de pb de manière à déplacer pb par une action de la forme pb $\longleftarrow$ pb + k. On introduit donc un mécanisme de *sauvegarde* de pb, ce qui donne le schéma de la figure 13.12-b. Chacune des positions de pb pointe sur une case mémoire qui contient la sauvegarde de l'ancienne valeur (plus bas dans la pile). La suite des ces sauvegardes successives forme ce que l'on appelle le *chaînage dynamique*, ou *lien dynamique*.

> **Remarque :** Il existe également en compilation une notion de *lien statique*, à ne pas confondre avec le lien dynamique. Le lien dynamique chaîne entre eux les environnements de deux blocs A et B tels que A *appelle* B, à l'exécution. Le lien statique chaîne entre eux les environnements de deux blocs A et B tels que B *est défini dans* A, dans le texte du programme. Cela peut arriver en Pascal, Ada, ANSI C, par exemple, mais pas en C classique.

La figure 13.13 donne le programme de calcul de la suite de Fibonacci dans lequel on a introduit la manipulation de la variable pb.

2.6 Résumé de l'occupation de la mémoire et remarques

Nous avons déjà dit au chapitre 4, paragraphe 4., que la mémoire nécessaire aux données d'un programme est formée de deux zones disjointes : une zone nécessaire à la gestion des zones mémoire allouées et libérées dynamiquement à la demande du programme, qu'on appelle *tas*, et une zone nécessaire à la gestion des variables du lexique global et des lexiques locaux des procédures.

Nous venons de voir précédemment que cette deuxième zone comporte non seulement les variables et les paramètres mais aussi éventuellement des données de liaison entre appelant et appelé : sauvegarde du lien dynamique dans notre cas, adresse de retour de sous-programme dans le cas où l'instruction d'appel l'empile (cas du 68000, paragraphe 3.4.1), lien statique éventuellement. Cette zone est gérée comme une pile et on parle souvent de la *pile à l'exécution* pour la désigner.

Un certain nombre de variables peuvent aussi être stockées dans les registres du processeur. Il est alors nécessaire d'assurer que leur valeur ne peut pas être modifiée par un sous-programme appelé. La sauvegarde éventuelle de ces registres est aussi effectuée dans la zone pile (Cf. Paragraphe 3.1.3).

Noter que les accès à la zone *pile* sont des accès directs dans un tableau, par adresse et déplacement, du genre MEM[pp+4]. Dans une véritable pile les seules actions autorisées sont Empiler et Dépiler (Cf. Chapitre 4, paragraphe 5.)

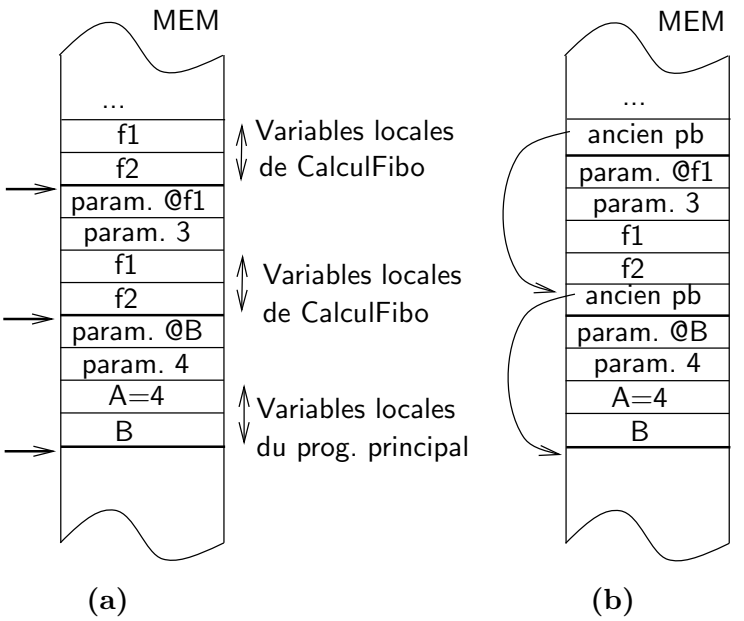


FIG. 13.12 – Pointeur de base d'environnement et lien dynamique. On reprend l'exemple de CalculFibo, figure 13.11.
(a) : position des variables locales du programme principal lors de deux appels de CalculFibo emboîtés ; les flèches indiquent la position de la base des 3 environnements ; la variable pb doit prendre successivement ces 3 valeurs lors du déroulement du programme.
(b) : insertion des sauvegardes de pb.

```
CalculFibo : une action
   pp ⟵ pp − 4 ; MEM[pp] ⟵ pb { sauvegarde de pb }
   pb ⟵ pp { installation du nouveau pb, l'ancien est en MEM[pb] }
   { paramètres : n est en MEM[pb+8] ; f est en MEM[pb+8]. }
   pp ⟵ pp − 8 { Allocation pour les variables locales }
   { Lexique local : f1 sera en MEM[pb−8] et f2 en MEM[pb−4] }
   si MEM[pb+8] = 0 ou MEM[pb+8] = 1 alors MEM[MEM[pb+4]] ⟵ 1
   sinon
      { Premier appel : }
         MEM[pp−4] ⟵ 4 MEM[pb+8] − 1
         MEM[pp−8] ⟵ 4 pb+4
         pp ⟵ pp − 8 ; CalculFibo ; pp ⟵ pp + 8
      { Deuxième appel : }
         MEM[pp−4] ⟵ 4 MEM[pb+8] − 2
         MEM[pp−8] ⟵ 4 pb+4
         pp ⟵ pp − 8 ; CalculFibo ; pp ⟵ pp + 8
      MEM[MEM[pb+4]] ⟵ MEM[pb−8]+MEM[pb−4]
   pp ⟵ pb { libération des variables locales }
   pb ⟵ MEM[pp] ; pp ⟵ pp+4 { restauration de l'ancien pb }


lexique du programme principal :
   pp, pb : des pointeurs { Uniques variables globales restantes }
algorithme du programme principal :
   Initialiser { Initialisation de la zone pile }
   pb ⟵ pp { Initialisation du pointeur de base d'environnement }
   pp ⟵ pp − 8 { Place nécessaire aux variables du lexique global }
   { A est en MEM[pb−8] et B en MEM[pb−4]. }
   { Traduction de l'appel Lire (A) : }
      MEM[pp−4] ⟵ 4 pb−8
      pp ⟵ pp − 4 ; Lire ; pp ⟵ pp + 4
   { Appel de CalculFibo }
      MEM[pp−4] ⟵ 4 MEM[pb−8]
      MEM[pp−8] ⟵ 4 pb−4
      pp ⟵ pp − 8 ; CalculFibo ; pp ⟵ pp + 8
   { Traduction de l'appel Ecrire (B) : }
      MEM[pp−4] ⟵ 4 MEM[pb−4]
      pp ⟵ pp − 4 ; Ecrire ; pp ⟵ pp + 4
```

FIG. 13.13 – Elimination complète des lexiques et des paramètres dans CalculFibo et introduction du pointeur de base d'environnement pb.

et ne préjugent pas de l'implantation de la pile, qui peut fort bien être une séquence chaînée.

Noter aussi que la gestion en pile de la mémoire permet une réutilisation maximale de la mémoire pour les lexiques locaux de deux blocs qui ne s'appellent pas l'un l'autre.

2.6.1 Problèmes d'initialisation

Nous verrons plus tard que, lors de l'exécution effective, le programme dit *principal* dans le texte d'un programme utilisateur n'est pas le premier bloc existant. Il y a toujours un appelant, ne fût-ce que l'interprète de commandes à partir duquel on lance l'exécution du programme (Cf. Chapitre 20).

Le codage du programme principal suit donc exactement les mêmes principes que le codage des actions et fonctions paramétrées. L'initialisation des pointeurs pp et pb n'apparaît pas dans le code des programmes utilisateurs. En produisant le codage du bloc qui correspond au programme principal, on suppose que ce bloc hérite, à l'exécution, d'une valeur pertinente de ces deux pointeurs, installée auparavant. Le codage du programme principal doit en contrepartie assurer que ce bloc se termine proprement comme une action ou fonction, par un retour au contexte appelant.

2.6.2 Allocation mémoire mixte statique/dynamique

L'allocation mémoire pour les variables, et la transformation de programme correspondante, produit des programmes dans lesquels apparaissent des accès mémoire de la forme MEM[b+k], où b est une adresse de base, et k est une constante calculée statiquement par le compilateur. En effet, dans le cas des procédures et fonctions récursives, il est impossible d'associer statiquement (c'est-à-dire pendant la compilation et indépendamment des exécutions) une adresse mémoire à chaque nom de variable apparaissant dans le lexique. Plusieurs adresses correspondent au même nom, et elles sont en nombre dépendant de l'exécution. Nous avons résolu cet aspect par une gestion de la mémoire en pile, et une allocation mixte statique/dynamique : les adresses de base sont dynamiques, elles dépendent de l'exécution ; les déplacements des différentes variables par rapport à l'adresse de base sont calculés statiquement, et sont indépendants de l'exécution.

La plupart des langages machine offrent des accès mémoire avec adressage indirect par registre et déplacement, qui sont utilisables directement pour coder nos programmes en langage d'assemblage. Il suffit que l'adresse de base soit rangée dans un registre. S'il n'existe pas d'adressage avec déplacement, l'adresse complète de la variable peut être calculée par une addition explicite avant l'accès mémoire.

En revanche le mode d'adressage *indirect* est indispensable. La gestion en pile de la mémoire n'est pas implémentable en langage machine sans adressage indirect pour les accès mémoire : un adressage direct signifie que toutes les

adresses sont calculables statiquement, et inscrites une fois pour toutes comme des constantes dans le programme en langage machine.

3. Traduction en langage d'assemblage : solutions globales

On intègre les solutions à tous les aspects en présentant deux classes de solutions : la classe des solutions à base de pile, du type de celle utilisée pour une machine 68000 ; la classe des solutions à base de fenêtres de registres, du type utilisé pour une machine SPARC.

3.1 Utilisation des registres et sauvegarde

3.1.1 Utilisation des registres pour les variables globales

Les registres de la machine pour laquelle on produit du code sont bien appropriés pour ranger les uniques variables globales qui subsistent après toutes les transformations de programme envisagées : les deux pointeurs, ou adresses, pp et pb. En effet les registres sont par définition accessibles dans tout contexte, et la rapidité d'accès (mémoire plus rapide que la grande mémoire, nombreuses instructions du langage machine travaillant directement sur des registres) est intéressante pour des variables qui sont manipulées très souvent.

3.1.2 Utilisation des registres pour les temporaires

D'autre part, lorsqu'on code un programme d'un langage de haut niveau, il est courant de faire apparaître des *variables temporaires* dans les calculs, qui ne correspondent à aucun nom explicitement défini par l'utilisateur dans le lexique. Pour un programme comme :

```
x, y, z : des entiers
x ⟵ 3*(y + 2*z) − 7*(x+y)
z ⟵ y
```

Il est impossible de calculer l'expression à affecter à x sans utiliser d'autres emplacements mémoire que ceux alloués aux noms x, y, z. Dans certains langages machine (comme celui du SPARC), il est même impossible de coder l'affectation z ⟵ y (z et y étant en mémoire) sans passer par un registre intermédiaire : il n'y a pas de transfert mémoire vers mémoire directement.

Où placer ces temporaires de calcul ? Pour des raisons de temps d'exécution, on aimerait les placer dans les registres de la machine. On peut toujours imaginer, toutefois, une expression suffisamment compliquée pour nécessiter plus de variables temporaires qu'il n'y a de registres disponibles. Dans ce cas les compilateurs placent les temporaires dans la pile.

3.1.3 Nécessité de sauvegarde des registres

Si les registres de la machine servent aux temporaires de calcul, il apparaît des cas comme :

```
lexique
    x, y, z : des entiers
    f : un entier ⟶ un entier                f(a) : 2 * a*a*a + 4 *a*a + 6*a + 1
algorithme
    x ⟵ f( 3*(y + 2*z) − 7*(x+y) )
```

Le programme principal et la fonction f doivent tout deux faire appel à des registres comme temporaires de calcul. S'ils utilisent les mêmes registres, il se peut que les temporaires du programme principal ne soient pas préservés de part et d'autre de l'appel de f.

Pour remédier à cet inconvénient, il faut sauvegarder les valeurs des registres utilisés comme temporaires de calcul, lorsqu'on passe d'un bloc à un autre. Il y a essentiellement deux classes de solutions : 1) *la sauvegarde dans l'appelant*, de tous les registres qu'il utilise et veut préserver de part et d'autre de l'appel d'un autre bloc ; 2) *la sauvegarde dans l'appelé*, de tous les registres dans lesquels il écrit. La deuxième solution a tendance à provoquer moins de sauvegardes : on ne sauvegarde que ce qui est effectivement modifié. Cela peut malgré tout être inutile : il est en effet possible qu'un appelant ne l'utilise pas.

La sauvegarde est réalisée dans la pile, par exemple juste au-dessus des variables locales. L'exemple détaillé au paragraphe 3.4.3 ci-dessous illustre la sauvegarde des registres dans l'appelé.

3.2 Appel de sous-programme : l'aller

Nous avons résolu les problèmes d'allocation mémoire pour les lexiques locaux et les paramètres. Dans nos algorithmes modifiés, il subsiste des appels réduits au nom de la procédure ou fonction appelée. Chaque bloc est codé en langage d'assemblage d'après les principes énoncés au paragraphe 1. Il reste à coder les appels de blocs par des instructions de branchement à des sous-programmes disponibles dans le langage machine considéré. Se pose alors le problème de l'*adresse* du bloc à atteindre.

Le cas le plus simple est celui des structures de blocs *statiques* : les procédures et fonctions sont toutes connues, elles ont des *noms*, et sont toujours appelées directement par l'intermédiaire de leur nom, dans le programme en langage de haut niveau.

On code chaque bloc P séparément (voir aussi aspects de *compilation séparée*, chapitre 18, paragraphe 2.), en lui associant une étiquette de début d'après le nom du bloc dans le langage de haut niveau, par exemple `_P`. Les appels de P sont codés par : `call _P`, si `call` est le nom de l'instruction de saut à un sous-programme (Cf. Chapitre 12, paragraphe 1.4.3).

Les textes obtenus par codage des différents blocs sont simplement juxta-

posés, dans un ordre quelconque. Il suffit de se rappeler quelle est l'étiquette qui correspond au bloc du programme principal. On l'appelle le *point d'entrée* du programme.

Un cas plus compliqué apparaît lorsque le langage de haut niveau qu'on utilise permet de définir par exemple un tableau de fonctions. Dans ce cas les fonctions ne sont plus nécessairement appelées par leur nom. On noterait par exemple, dans un langage d'actions étendu :

```
Tf : le tableau sur [1..10] de (un entier ⟶ un entier)
x : un entier sur [1..10] ; y, z : des entiers
Lire (x) ; Lire (y) ; z ⟵ Tf[x] (y)
```

Le programme en langage d'assemblage correspondant doit manipuler explicitement les adresses des fonctions, c'est-à-dire les adresses associées aux étiquettes de début des séquences d'instructions des fonctions. L'adresse à laquelle on doit brancher pour appeler la fonction Tf[x] doit être *calculée* dynamiquement, et le langage machine doit offrir un appel de sous-programme par adressage indirect. Dans la suite nous ne nous intéressons plus à ce cas.

3.3 Appel de sous-programme : le retour

Dans les programmes transformés du langage d'action obtenus au paragraphe précédent, nous écrivons tout simplement dans un bloc P : CalculFibo pour signifier qu'il faut exécuter le corps de cette procédure ; nous sous-entendons que, lorsque cette procédure appelée sera terminée, le cours de l'exécution reprendra dans P, juste après l'appel.

Le basculement de contexte à l'aller est simple, comme nous venons de le voir. Le basculement de contexte au retour, en revanche, doit être étudié de près.

Tout d'abord, il faut terminer le corps de la procédure appelée CalculFibo par un branchement explicite au point où l'on désire retourner. Et comme ce point dépend de la position de l'appel de CalculFibo dans le programme, c'est une adresse variable selon les appels. Le branchement de retour est donc nécessairement indirect, puisque le code d'une procédure est indépendant des endroits où elle est appelée.

En quelque sorte, l'adresse où il faudra retourner en fin de procédure est un *paramètre donnée* supplémentaire de toute procédure. Le raisonnement sur les paramètres de fonctions et procédures récursives que nous avons tenu au paragraphe 2. est encore valable.

L'adresse de retour doit donc être stockée quelque part (par exemple rangée dans la pile avec les autres paramètres données) par le bloc *appelant*, *avant* l'instruction de saut au sous-programme *appelé*. En effet, cette adresse est une valeur du compteur programme, et le saut à un sous-programme consiste justement à forcer une nouvelle valeur du compteur programme.

Fort heureusement les langages machines offrent toujours une instruction

de branchement à un sous-programme avec sauvegarde intégrée de l'adresse départ. Pour le retour, soit on trouve une instruction cohérente avec la sauvegarde lors de l'appel (cas du 68000), soit il faut utiliser une instruction de branchement en respectant les conventions de l'instruction d'appel (cas du SPARC).

Rappelons enfin que le programme principal doit se comporter comme les autres blocs ; on suppose que l'on y est arrivé par un mécanisme d'appel de sous-programme, et il se termine donc par un retour au contexte appelant.

> **Remarque :** Si l'on oublie l'instruction de retour à la fin d'un sous-programme, le processeur poursuit l'exécution en séquence c'est-à-dire en général dans le code d'un autre sous-programme.

3.4 Solutions à base de pile, type 68000

3.4.1 Gestion de la zone de pile en assembleur

Les modes d'adressage indirects par registre avec pré-décrémentation ou post-incrémentation sont particulièrement bien adaptés à la gestion en pile d'une portion de la mémoire. Il suffit de réserver un registre pour servir de pointeur de pile.

Le choix de ce registre n'est pas toujours entièrement libre. Par exemple, le jeu d'instructions du 68000 fournit l'instruction `jsr` de saut à un sous-programme, avec une sauvegarde automatique de l'adresse qui suit l'instruction `jsr` de la forme

$$\mathsf{RegA[7]} \longleftarrow \mathsf{RegA[7]} - 4\,;\ \mathsf{MEM[RegA[7]]} \longleftarrow \mathsf{PC}$$

On y reconnaît un mécanisme de gestion de pile avec RegA[7] (le registre d'adresse numéro 7) comme pointeur de pile, placé sur la dernière case occupée ; la pile croît en diminuant les adresses.

Il suffit donc de gérer le passage des paramètres et l'allocation de mémoire pour les lexiques locaux en adoptant cette convention imposée par le jeu d'instructions lui-même (et qu'on ne peut donc pas remettre en cause, puisque l'algorithme d'interprétation du langage machine est câblé).

Le retour de sous-programme adopte la même convention. L'instruction `rts` est un branchement inconditionnel, doublement indirect par registre avec postincrémentation. L'effet est :

$$\mathsf{PC} \longleftarrow \mathsf{MEM[RegA[7]]}\,;\ \mathsf{RegA[7]} \longleftarrow \mathsf{RegA[7]} + 4$$

Autrement dit `rts` trouve son adresse en sommet de pile, et la dépile.

3.4.2 Instructions `link` et `unlink`

Reprenons la séquence d'instructions nécessaires à l'installation de l'environnement local de la procédure CalculFibo (Cf. Figure 13.13) :

pp ⟵ pp - 4 ; MEM[pp] ⟵ pb ; pb ⟵ pp ; pp ⟵ pp − 8

Et la séquence d'instructions symétrique, à la fin de CalculFibo :

pp ⟵ pb ; pb ⟵ MEM[pp] ; pp ⟵ pp+4

En langage d'assemblage 68000, on écrit, en supposant que le pointeur pp est rangé dans A7 et le pointeur pb dans A6 :

```
link  A6, -8          ! séquence de début
....
unlink A6             ! séquence de fin
```

L'effet de ces instructions est exactement celui décrit plus haut. Noter que le choix de A6 comme pointeur de base d'environnement est libre, puisque c'est un paramètre explicite des instructions `link` et `unlink` ; c'est une convention des compilateurs. A7 en revanche est le pointeur de pile obligé.

3.4.3 Codage avec lien dynamique, variables locales dans la pile, temporaires dans des registres

Soit les actions A, B et C définies Figure 13.14. A est l'action principale, elle appelle B qui appelle C. Nous donnons figure 13.15 le codage de la procédure A dans un langage d'assemblage pour 68000. Rappelons que l'adresse de retour de sous-programme est sauvé dans la pile d'exécution par l'instruction `jsr` et que la sauvegarde du lien dynamique et la mise en place de l'environnement local du sous-programme sont assurées par l'instruction `link`. Par rapport au schéma de pile donné figure 13.12-b, la zone des données de liaison comporte non seulement la sauvegarde de pb mais aussi l'adresse de retour (en-dessous), ce qui donne la taille de deux adresses (8 octets) à cette zone.

3.5 Solutions à base de fenêtres de registres, type SPARC

Nous étudions ici les schémas de codage de sous-programmes suggérés par certaines architectures de processeurs, dites *à fenêtres de registres*. Pour préciser les choses, nous traitons l'exemple du SPARC. Comme nous l'avons signalé au chapitre 12, paragraphe 1.7.1, le processeur SPARC est équipé d'un *banc* de registres, qui sont à accès plus rapide que la grande mémoire. Ces nombreux registres sont utilisés pour optimiser les mécanismes de passages de paramètres et d'appels de sous-programmes. Les techniques étudiées plus haut, et mises en oeuvre directement dans le cas des langages machine type 68000, sont encore applicables. L'idée consiste à utiliser le banc de registres comme *un cache sur la pile*. Nous détaillons ci-dessous les mécanismes permettant de comprendre cette affirmation.

A : l'action { *sans paramètre, donc.* }
lexique : x, y : des entiers
algorithme : x ⟵ 1 ; B (3*x + 1, y) ; x ⟵ y
B : l'action (la donnée a : un entier, le résultat b : un entier)
lexique : z : un entier
algorithme : C (z) ; b ⟵ z + a
C : l'action (le résultat u : un entier)
lexique : i : un entier
algorithme :
i ⟵ 1 ; u ⟵ 1
tant que i ≤ 10
u ⟵ u + i ; i ⟵ i + 1

FIG. 13.14 – Procédures A, B, C

```
! Tous les idfs désignent des entiers de 4 octets
TailleEnvB = 4
! Accès à la variable locale, relatif à l'adresse de base :
deltaZ = 4
! Accès aux paramètres, relatifs à l'adresse de base :
deltaA = 12
deltaB = 8
   .text
B :link A6, -TailleEnvB
   ! l'action A a préparé les paramètres : donnée a <--> valeur 3*x+1
   ! et résultat b <--> adresse de y avant l'appel à B
   move.l A0, -(SP)            ! sauvegarde de 2 registres
   move.l D0, -(SP)
   ! appel de C :
   move.l A6, A0               ! calcul du param, adr de z dans A0
   sub.l deltaZ, A0
   move.l A0, -(SP)            ! empiler A0
   jsr C
   add.l ♯4, SP                ! libérer paramètre
   ! fin de l'appel de C
   ! b <-- z + a
   move.l (A6+deltaA), D0      ! D0 <-- a
   add.l (A6-deltaZ), D0       ! D0 <-- D0 + z
   move.l (A6+deltaB), A0      ! A0 <-- adresse b
   move.l D0, (A0)             ! b <-- D0
   move.l (SP)+, D0            ! restauration de 2 registres
   move.l (SP)+, A0
   unlk A6
   rts
```

FIG. 13.15 – Codage de l'action B en langage d'assemblage 68000

3.5.1 Le mécanisme de fenêtres de registres du SPARC

Le SPARC dispose d'un ensemble de registres géré en *fenêtres* : 8 registres dits *globaux* sont accessibles en permamence. Les autres registres sont accessibles par groupes de 24 registres dits *inputs*, *locals* et *outputs*. Le groupe, ou *fenêtre*, courant, est repéré par le registre `CWP` (pour *Current Window Pointer*).

Deux instructions spécifiques permettent de déplacer la fenêtre de registres en incrémentant ou décrémentant le registre `CWP` : `save` et `restore`. Le décalage est de 16 registres. Deux fenêtres consécutives coïncident donc sur 8 registres : les registres *ouputs* de la fenêtre courante avant l'exécution du `save` correspondent aux *inputs* de la fenêtre courante après l'exécution du `save`. Le `restore` a l'effet inverse. Le mécanisme des fenêtres de registres est illustré par la figure 13.16.

3.5.2 Schéma de codage idéal

Le mécanisme des fenêtres permet d'implémenter de manière efficace la notion de *contexte* des langages procéduraux. Schématiquement, lorsqu'une procédure en appelle une autre, la procédure appelée effectue un `save`, ce qui lui permet de travailler avec un nouveau contexte. La communication entre procédure appelante et procédure appelée (pour le passage des paramètres et la gestion de l'adresse de retour) est assurée par les 8 registres communs aux deux fenêtres, qui constituent la zone d'échange. En fin d'exécution, la procédure appelée effectue un `restore`, qui rétablit le contexte précédent, puis un saut à l'adresse de retour.

Notons que lors de l'appel de sous-programme l'adresse de l'instruction `call` est sauvegardée dans le registre `%o7`, d'où l'adresse `%o7+8` pour le retour (nous faisons suivre les instructions de saut par une instruction `nop` de façon à ne pas se préoccuper du *pipeline*).

Le cas idéal le plus simple qu'on puisse envisager est le cas où les procédures n'utilisent jamais plus de 7 paramètres d'entrée, ni plus de 8 variables locales, et où le nombre d'appels imbriqués ne dépasse pas le nombre de fenêtres physiquement disponibles. Dans ce cas le mécanisme des fenêtres de registres est suffisant. Il suffit de considérer l'effet de l'instruction `CALL` pour respecter la convention sur la mémorisation de l'adresse de retour : dans le registre `o7` de l'appelante, qui correspond au registre `i7` de l'appelée.

Le schéma de codage idéal est donné figure 13.17.

> **Remarque :** La séquence `restore ; jmpl %o7+8, %g0 ; nop` est souvent transformée en `jmpl %i7+8, %g0 ; restore`. En effet, l'adresse de retour est située dans le registre `%o7` de l'appelante, qui s'appelle `%i7` dans l'appelée. Elle s'appelle donc `%i7` *avant* le `restore`, et `%o7` *après* le `restore`. En profitant du délai de branchement, on peut réaliser le branchement avant la restauration du contexte.

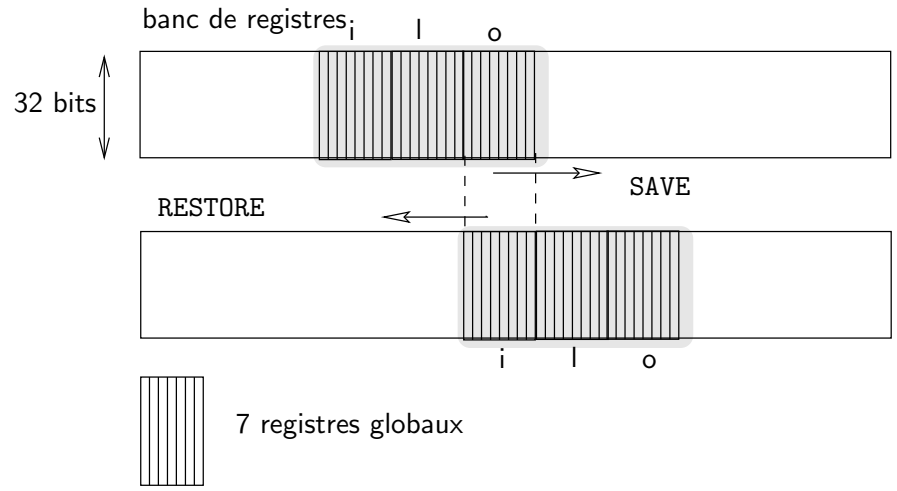


FIG. 13.16 – Mécanisme des fenêtres de registres

```
appelee:
    save
    ! utilisation des paramètres d'entrée %i0...%i6
    ! et des variables locales %l0..%l7
    restore
    jmpl %o7+8, %g0
    nop

appelante:
    ! place les paramètres dans %o0..%o6
    call appelee  ! sauvegarde de l'adresse de retour dans %o7
    nop
    ...
```

FIG. 13.17 – Schéma de codage idéal utilisant le mécanisme de fenêtres de registres

3.5.3 Problème : le banc de registres est fini

En cas de programme conséquent ou de procédure récursive, l'hypothèse sur le petit nombre d'appels imbriqués devient fausse. Pour que le programmeur puisse considérer le banc de registres comme virtuellement infini, il faut prévoir d'utiliser la mémoire en plus du banc de registres.

Une solution simple consiste à imaginer que, lorsque le banc de registres est plein, les procédures et fonctions se mettent à utiliser directement la mémoire gérée en pile, selon le schéma étudié pour 68000. Comme le codage pour banc de registres diffère du codage pour pile — ne serait-ce que dans l'accès aux variables locales et paramètres — chaque bloc devrait alors posséder deux codages différents, selon qu'il est appelé quand le banc de registres est plein ou non. C'est hors de question.

Il faut donc se débrouiller pour obtenir cette commutation du banc de registres vers la pile de manière *transparente* pour les programmes des divers blocs, qui doivent toujours avoir l'impression que le banc de registres est infini.

La mise en oeuvre complète de cette solution transparente, plus la prise en compte du fait que certains programmes ont plus de 8 variables locales ou plus de 8 paramètres, est délicate. Elle n'est pas réalisable de manière satisfaisante sans utiliser le mécanisme d'*interruption logicielle* qui sera présenté dans la partie VI.

Nous donnons ci-dessous quelques indices pour comprendre les schémas de codage de langages à structure de blocs en langage d'assemblage SPARC, tels qu'on peut les observer en lisant le code produit par un compilateur C par exemple.

3.5.4 Cas réaliste

Pour que le programmeur (ou le concepteur de compilateur) puisse considérer le banc de registres comme virtuellement infini, il faut prévoir un mécanisme de sauvegarde des fenêtres de registres, lorsqu'on effectue plus de `save` qu'il n'est physiquement possible d'en gérer.

Cette sauvegarde est assurée automatiquement, pendant l'interprétation de l'instruction `save` par le processeur, comme traitant d'une interruption interne déclenchée par le dépassement de capacité du banc de registres. Ce traitant d'interruption réalise une copie des registres `%i0..%i7` et `%l0..%l7` d'une fenêtre dans la pile. Il adopte des conventions sur la zone de pile utilisée, qui doivent être connues du programmeur.

L'idée est d'utiliser un registre comme pointeur de pile. C'est `%o6`. En langage d'assemblage `%sp` est d'ailleurs un synonyme de `%o6` (`sp` pour *Stack Pointer*). Toute fenêtre correspondant au contexte d'une procédure en cours d'exécution doit être telle que son registre `%o6` pointe sur une zone de 64 octets libres de la pile. Cet invariant est supposé vérifié au moment où la procédure principale de notre programme est appelée (il a été installé par l'appelant de cette procédure, l'interprète de commandes par exemple, Cf. Chapitre 20).

Pour que la propriété soit toujours vraie il suffit, lors de tout changement de contexte qui installe une nouvelle fenêtre, d'initialiser le registre `%o6` de la nouvelle fenêtre.

Or l'instruction `save` du SPARC se comporte comme une addition, qui interprète la désignation de ses opérandes dans la fenêtre de départ, et la désignation du résultat dans la fenêtre d'arrivée. Une instruction `save %o6, -64, %o6` permet donc d'initialiser le registre `%o6` de la nouvelle fenêtre d'après la valeur du registre `%o6` de l'ancienne : l'instruction décale le pointeur vers les adresses inférieures, réservant ainsi un espace de la pile de taille $4 * 16 = 64$ pour 16 registres de 4 octets. L'ancien pointeur de pile, qui s'appelait `%o6` dans le contexte de l'appelant, est toujours accessible. Il s'appelle `%i6` dans le contexte de l'appelé. `%fp`, pour *Frame Pointer*, est un synonyme de `%i6` en assembleur. Le programme de la figure 13.19 illustre ce mécanisme.

3.5.5 Variables locales dans la pile et paramètres en excès

Dans le cas où le nombre de registres ne suffit pas pour stocker les variables locales et passer les paramètres, il est possible d'utiliser la pile. On peut lors du `save` demander plus de place dans la pile que les 64 octets nécessaires à la sauvegarde des registres `%i0..%i7` et `%l0..%l7` par le traitant d'interruption gérant la demande d'une fenêtre alors qu'il n'en existe plus de libre.

La figure 13.20 illustre l'organisation de la pile et des fenêtres de registres dans ce cas. La seule contrainte qu'il faut respecter est de conserver la place *en haut* de zone allouée pour la sauvegarde éventuelle de la fenêtre courante.

Les variables locales sont alors rangées en bas de pile et on y accède via une adresse de la forme `%fp − d`, le déplacement `d` pouvant être calculé statiquement de la même façon que pour la solution à base de pile seulement.

Les paramètres sont rangées par l'appelant dans le haut de sa zone locale, juste sous la zone de sauvegarde pour le traitant d'interruption. Dans l'appelé on accède alors aux paramètres effectifs via une adresse de la forme `%sp + d'` avec `d'` $\geq$ `64`, `d'` étant lui aussi calculable statiquement.

4. Exercices

E13.1 : Observation de l'exécution d'une action récursive
Considérons l'algorithme de calcul de la suite de Fibonacci (Cf. Figure 13.3) et plus particuilièrement la traduction décrite dans le paragraphe 2.5 et la figure 13.10. Dessiner les différents états du tableau MEM et plus précisément la partie pile au cours de l'exécution de l'action calculFibo avec la valeur 4 pour la variable A.

E13.2 : Codage des fonctions
Reprendre la démarche décrite dans ce chapitre pour les fonctions. Les paramètres d'une fonction sont des données, pour lesquelles on peut procéder

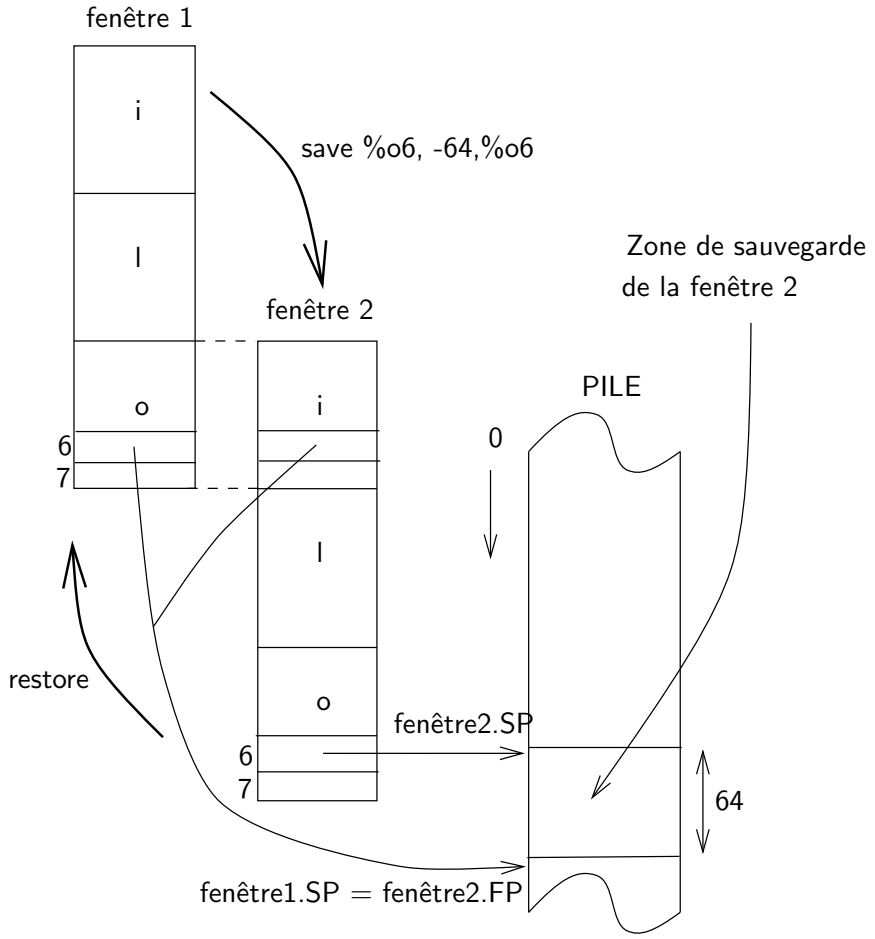


FIG. 13.18 – Utilisation des fenêtres de registres avec réservation de place dans la pile pour la sauvegarde des fenêtres. Noter que dans l'écriture de l'instruction `save %o6, -64, %o6`, le premier `%o6` désigne celui de l'ancienne fenêtre et le second celui de la nouvelle fenêtre.

```
appelee:
    save %o6, -64, %o6      ! ou save %sp, -64, %sp
    ! réserve une zone de 64=16*4 octets dans la pile, pour
    ! la sauvegarde des registres i et l de ce nouveau contexte.
    ! ... utilisation des paramètres d'entrée %i0...%i6
    ! et des variables locales %l0..%l7 ...
    ! retour et restauration de la fenêtre
    jmpl %i7+8, %g0
    restore
appelante:
    ...
    call appelee  ! sauvegarde de l'adresse de retour dans %o7
    nop
```

FIG. 13.19 – Programme nécessitant la sauvegarde des registres

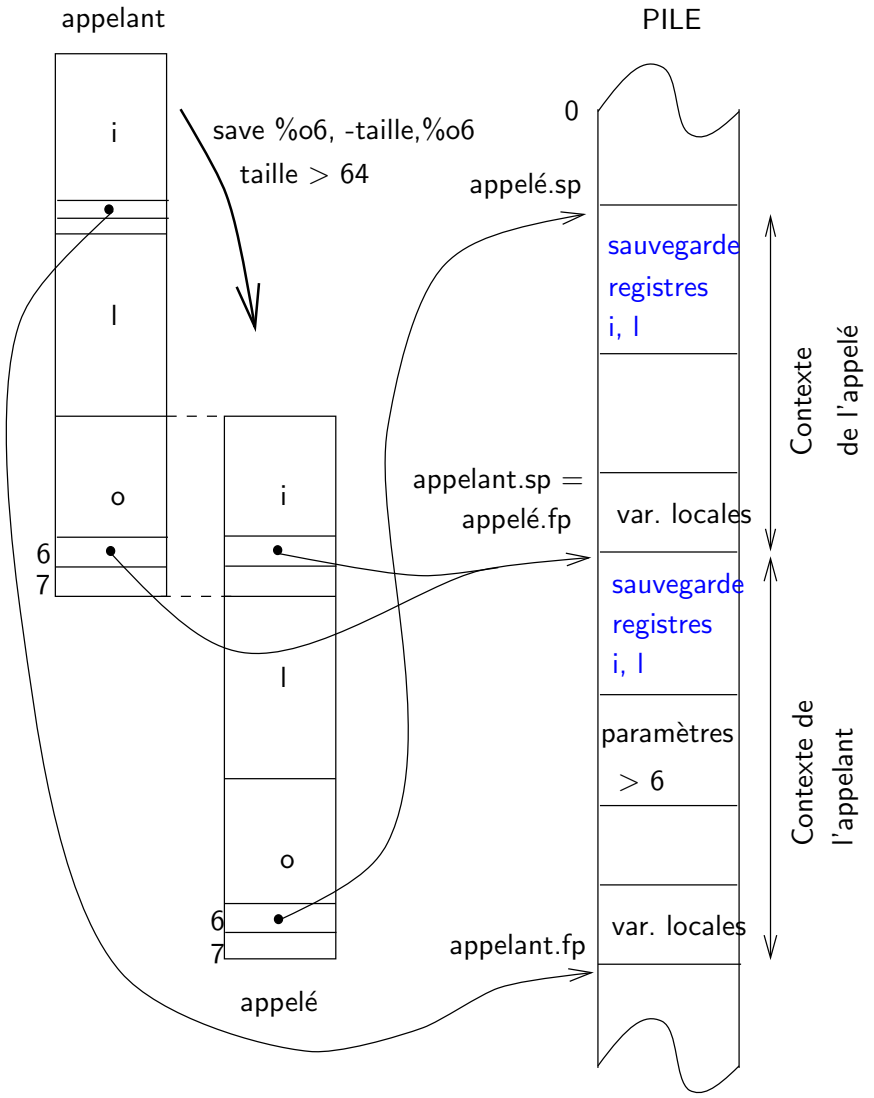


FIG. 13.20 – Variables locales et paramètres en excès dans la pile

comme dans le cas des actions : on passe leur valeur. Le *résultat* d'une fonction est calculé par l'appelé sans que l'appelant lui ait fourni l'adresse d'une de ses variables. Il faut donc choisir un mécanisme qui permet à la fonction appelée d'ecrire le résultat qu'elle calcule dans un emplacement mémoire connu du contexte appelant, qui pourra le lire. On peut examiner deux solutions : le résultat est stocké dans un registre ou dans la pile. Noter que selon le type du résultat de la fonction (par exemple s'il s'agit d'un produit de types ou d'un tableau) la solution de la pile est plus facile à mettre en oeuvre.

E13.3 : Nombre de '1' dans la représentation binaire d'un entier
Traduire dans le langage d'assemblage de votre choix l'algorithme présenté au chapitre 12, paragraphe 1.4.1, qui permet de compter les bits à 1 dans la représentation binaire d'un entier.

E13.4 : Traduction de boucles imbriquées
Traduire en langage d'assemblage l'algorithme suivant :

```
lexique
    i, j, x : des entiers ; N : l'entier ... { donné }
algorithme
    x ⟵ 0
    i parcourant 0 .. N
        j parcourant 0 .. N
            x ⟵ x + i * j
```

Commencer par exprimer chaque boucle à l'aide de la construction tant que comme suggéré au paragraphe 1.5 du chapitre 4.

E13.5 : Algorithme de Bresenham
Traduire dans le langage d'assemblage de votre choix l'algorithme présenté au chapitre 5, paragraphe 2.3 qui permet de calculer les coordonnées des points d'une droite dans un plan.

E13.6 : Suite de Syracuse
L'algorithme ci-dessous calcule les termes successifs de la suite de Syracuse. Traduire cet algorithme dans le langage d'assemblage de votre choix.

```
lexique :
    X : l'entier constant ...
algorithme :
    tant que X ≠ 1
        si X reste 2 = 1                                  { X est impair }
            X ⟵ 3 × X + 1
        sinon                                             { X est pair }
            X ⟵ X quotient 2
```

E13.7 : Suite de Fibonacci
Traduire complètement les algorithmes de l'exemple *suite de Fibonacci* (algorithme décrit dans la figure 13.3) dans l'assembleur de votre choix. Envisager des solutions avec ou sans gestion du lien dynamique, avec ou sans utilisation de fenêtres de registres.

E13.8 : Parcours d'un tableau d'entiers
Soit l'algorithme :

```
lexique
    N : l'entier ... { donné }
    T : un tableau sur [0..N−1] d'entiers
    S, i : des entiers
algorithme
{ calcul de la somme des éléments du tableau }
    S ⟵ T[0]
    i parcourant (1..N−1) : S ⟵ S + T[i]
```

Ecrire cet algorithme en langage d'assemblage. Envisager plusieurs solutions en utilisant diverses représentations pour les entiers (2, 4 ou 8 octets) et divers modes d'adressage pour les accès aux éléments du tableau.

E13.9 : Parcours d'un tableau de structures
Soit l'algorithme :

```
lexique
    N : l'entier ... { donné }
    ST : le type ¡c : un caractère ; m : un entier
    T : un tableau sur [0..N−1] de ST
    M, i : des entiers
algorithme
{ calcul du maximum des éléments du tableau }
    M ⟵ T[0].m
    i parcourant (1..N−1) :
        Si M < T[i] alors M ⟵ T[i].m
```

Proposer une représentation en mémoire du type ST et écrire en langage d'assemblage l'algorithme ci-dessus.

E13.10 : Parcours de matrice carrée et comparaison double longueur en complément à deux
Reprendre le problème E4.8 du chapitre 4, et proposer un programme en assembleur SPARC (description de la zone de données, suite d'instructions).

E13.11 : Observation de code
Observer le code produit par différents compilateurs pour différents programmes ; en général une option (-S pour le compilateur gcc sous UNIX) permet d'obtenir une version en langage d'assemblage du code. Retrouver l'implantation des variables globales, locales et des paramètres.

Quatrième partie

A la charnière du logiciel et du matériel...

Chapitre 14

Le processeur : l'interprète câblé du langage machine

Ce chapitre décrit la mise en oeuvre par un circuit de l'algorithme d'interprétation des instructions d'un processeur. Nous parlons aussi d'interprète du langage machine. Cet interprète est câblé : ce n'est pas un programme mais un circuit.

Etant donné le jeu d'instructions défini pour un processeur, la description de cet algorithme permet de comprendre comment est exécutée une instruction, donc un programme en langage machine. C'est ce point de vue simple et purement pédagogique que nous adoptons ici, la description des méthodes de conception d'un processeur dépassant le cadre de ce livre. Nous ne parlerons pas non plus de processeur à *flot de données* ou *pipeliné*; pour une description approfondie le lecteur peut consulter [HP94].

Un processeur peut être considéré comme une machine algorithmique (Cf. Chapitre 11) formée d'une partie opérative (une UAL, des bus, des éléments de mémorisation, etc.) et d'une partie contrôle. Le processeur est relié à une mémoire dans laquelle est stocké un programme en langage machine. La question de la liaison entre une machine algorithmique et une mémoire a été étudiée au chapitre 11.

Le processeur doit récupérer les instructions en langage machine dans la mémoire, dans l'ordre du programme, et les exécuter une par une. L'algorithme d'interprétation du langage machine consiste ainsi en une boucle infinie (mais nous verrons dans la partie VI, Chapitre 22, qu'il est possible d'*interrompre* cette boucle infinie) : lire l'instruction courante, la décoder, réaliser le travail correspondant et déterminer l'adresse de l'instruction suivante (Cf. Paragraphe 1.6 du chapitre 12). Si l'instruction à exécuter est par exemple add d0, d1, d2 le processeur doit faire en sorte que les contenus des registres `d0` et `d1` soient présents aux deux entrées de l'UAL, activer l'addition et envoyer le résultat dans le registre `d2`.

La compréhension de ce chapitre suppose connus les chapitres 8, 9, 10, 11 et 12. Après avoir expliqué les principes de l'interprétation des instruc-

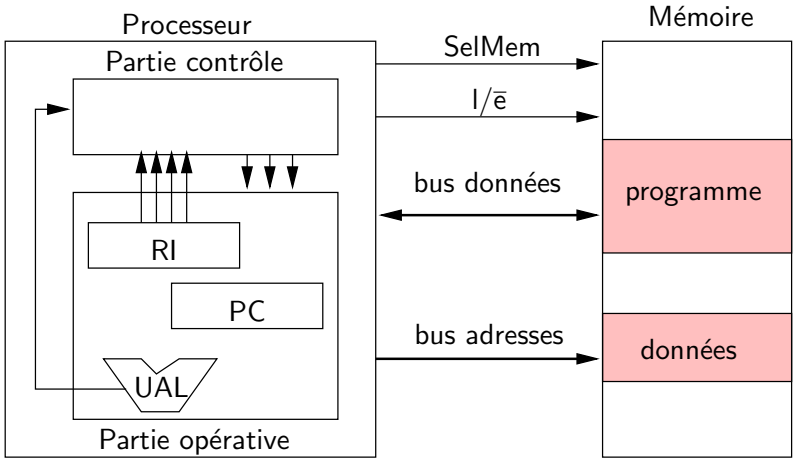


FIG. 14.1 – Organisation globale de l'ensemble processeur/mémoire. UAL désigne l'unité arithmétique et logique, PC le compteur de programme et RI le registre instruction.

tions du langage machine (paragraphe 1.), nous présentons un exemple extrêmement simple dont le jeu d'instruction est décrit dans le paragraphe 2. Nous donnons dans le paragraphe 3. une réalisation du processeur puis nous montrons comment l'améliorer (paragraphe 4.). Enfin, nous étudions les conséquences de quelques extensions du processeur (paragraphe 5.).

1. Les principes de réalisation

La figure 14.1 décrit la structure de l'ensemble processeur/mémoire. Le processeur est composé d'une *partie contrôle* et d'une *partie opérative* ; la première envoie des *commandes* à la seconde qui, elle, émet des *comptes-rendus*. La mémoire n'est ici reliée qu'à un processeur.

> *Les aspects précis de communication entre le processeur et la mémoire sont détaillés dans le chapitre 15. Les aspects de liaison avec le monde extérieur pour réaliser par exemple des entrées/sorties sont étudiés dans le chapitre 16.*

1.1 Relations du processeur avec la mémoire

Le processeur est relié à la mémoire par un bus adresses unidirectionnel et un bus données bidirectionnel.

Nous supposons ici que la lecture ou l'écriture s'effectue en un temps inférieur au temps de cycle d'horloge du processeur. L'accès à la mémoire est alors piloté par deux commandes : sélection de la mémoire SelMem et lecture ou écriture l/ē. Ainsi, pour écrire la valeur vvv dans la mémoire à l'adresse xxx le processeur doit mettre SelMem à 1 et l/ē à 0 en forçant la valeur vvv sur le bus données et la valeur xxx sur le bus adresses ; l'écriture est effective

au prochain front d'horloge du processeur. Pour lire la valeur contenue dans la mémoire à l'adresse xxx le processeur doit positionner les commandes SelMem et $\mathsf{I}/\overline{\mathsf{e}}$ en forçant la valeur xxx sur le bus adresses ; le contenu du mot mémoire est disponible sur le bus données à partir du prochain front d'horloge du processeur.

Le déroulement d'un accès mémoire dans le cas où la mémoire est plus lente a été décrit au paragraphe 2.2 du chapitre 9 et les aspects de synchronisation ont été détaillés au paragraphe 3.1 du chapitre 11.

La mémoire contient deux types d'informations : des instructions et des données. Une instruction machine comporte elle-même deux types d'informations : le code de l'instruction et la désignation de(s) opérande(s) de l'instruction. Selon les instructions et le format de leur codage ces informations peuvent être lues en un seul accès à la mémoire ou non. Lorsque plusieurs accès sont nécessaires, ils se déroulent nécessairement lors de cycles d'horloge différents et consécutifs, le code de l'instruction étant lu en premier. Un registre spécialisé appelé compteur programme (PC) repère le mot mémoire en cours de traitement. Il doit être mis à jour après chaque accès à une instruction en vue de l'accès suivant.

On peut envisager un autre type d'organisation dans laquelle la mémoire est organisée en deux parties distinctes : une mémoire pour les instructions et une mémoire pour les données. Ce type d'organisation nécessite deux bus adresses et deux bus données différents. Les principes généraux restent très proches ; nous n'en parlerons plus dans la suite.

1.2 Principes généraux de la partie opérative

La partie opérative d'un processeur doit être capable d'effectuer toutes les opérations et tous les transferts d'information nécessaires à l'exécution des instructions du langage machine.

Elle peut donc être très proche d'une partie opérative type, décrite au chapitre 11. Une particularité à signaler est l'existence d'un registre particulier (appelé *registre instruction* et noté RI), non manipulable par les instructions du langage machine et qui contient à chaque instant l'instruction en cours d'interprétation. Le contenu de ce registre sert de compte-rendu pour la partie contrôle.

L'UAL a deux types de sorties : une donnée qui est le résultat d'un calcul et les codes de conditions arithmétiques Zu, Nu, Cu, Vu. La donnée circule sur un bus et sera chargée dans un registre ou un mot mémoire ; les codes de conditions sont des informations de contrôle qui peuvent être testées par la partie contrôle : ce sont des entrées de la partie contrôle. Ces codes de conditions peuvent aussi être chargés dans 4 bits d'un *registre d'état*.

Notons qu'une adresse peut être une entrée de l'UAL lorsqu'un calcul est nécessaire sur une adresse ; par exemple, pour traiter un mode d'adressage indirect avec déplacement, il faut ajouter la valeur de l'adresse et le déplacement.

Les registres sont des éléments de mémorisation internes au processeur. Certains sont connus du programmeur et manipulables explicitement : un mnémonique leur est associé dans le langage d'assemblage. Ces registres peuvent contenir des données et/ou des adresses. Ils peuvent être classés en différentes catégories. Par exemple dans la famille des processeurs 68XXX, les registres sont typés en registres de données et registres d'adresses. Dans le SPARC, on trouve des registres globaux et des registres locaux, d'entrée et de sortie, les trois derniers étant organisés en fenêtres de registres.

D'autres registres, comme le compteur programme (PC), le pointeur de pile (SP), le mot d'état (SR) contenant les indicateurs Z, N, C, V, peuvent être manipulés au travers d'instructions spécialisées. Par exemple, les instructions de rupture de séquence ont un effet sur la valeur du registre PC. Les indicateurs Z, N, C, V du mot d'état contiennent les valeurs Zu, Nu, Cu, Vu calculées par l'UAL lors de la dernière instruction qui a mis à jour explicitement ces indicateurs. Dans certains processeurs, comme le SPARC par exemple, le jeu d'instructions comporte les instructions arithmétiques sous deux formes : `Addcc` et `Add`, addition avec ou sans mise à jour des indicateurs.

Toute action sur un registre provoque un changement d'état de la partie opérative. La partie opérative peut exécuter un certain nombre d'actions que nous appellerons *microactions* (Cf. Paragraphe 3.1). Une microaction est un ensemble de modifications simultanées de l'état de la partie opérative.

Rappelons que la partie opérative peut être vue comme un automate dont l'état est l'ensemble des valeurs contenues dans les registres. L'exercice E14.4 montre ce point de vue.

1.3 Principes généraux de la partie contrôle

La partie contrôle doit envoyer les commandes adéquates à la partie opérative, le processeur réalisant ainsi l'interprétation du langage machine. Le schéma général est le suivant : charger le registre d'instructions (RI) avec l'instruction courante (dont l'adresse est dans PC), décoder et exécuter cette instruction, puis préparer le compteur programme (PC) pour l'instruction suivante.

La partie contrôle d'un processeur est ainsi la réalisation matérielle d'un algorithme itératif qui peut être décrit par une machine séquentielle avec actions. Les sorties sont un ensemble d'ordres envoyés à la partie opérative ou à l'extérieur et les entrées sont des informations émanant de la partie opérative : valeur d'un code opération, valeur d'un code condition et indicateurs arithmétiques.

La machine séquentielle qui décrit l'algorithme de la partie contrôle est aussi appelée *automate (ou graphe) de contrôle* ou séquenceur.

Dans la suite nous développons un exemple simple pour lequel nous décrivons la partie contrôle tout d'abord de façon fonctionnelle par un algorithme itératif en considérant les ressources (registres) du processeur comme

des variables et la mémoire comme un tableau. Nous donnons ensuite une description sous forme de machine séquentielle avec actions aux états de laquelle on associe des microactions : ce sont les opérations effectivement réalisables par la partie opérative. L'objectif est de montrer comment on peut concevoir la partie opérative et la partie contrôle d'un processeur, étant donné le jeu d'instructions retenu.

2. Exemple : une machine à 5 instructions

Le processeur comporte un seul registre de données, directement visible par le programmeur, appelé `ACC` (pour accumulateur).

Les instructions sont décrites ci-dessous. On donne pour chacune une syntaxe de langage d'assemblage et l'effet de l'instruction.

`clr`	mise à zéro du registre `ACC`
`ld #vi`	chargement de la valeur immédiate `vi` dans `ACC`
`st [ad]`	chargement du mot mémoire d'adresse `ad` avec le contenu de `ACC`
`jmp ad`	saut à l'adresse `ad`
`add [ad]`	chargement de `ACC` avec la somme du contenu de `ACC` et du mot mémoire d'adresse `ad`.

Ce jeu d'instruction est extrêmement réduit, l'objectif étant de disposer de suffisamment d'éléments pour détailler la conception du processeur mais de se limiter de façon à garder une taille raisonnable à cette description. Remarquons que l'on pourrait rendre le jeu d'instructions plus symétrique en ajoutant une instruction de chargement absolu : `ld [ad]`.

La taille nécessaire au codage d'une adresse ou d'une donnée est 1 mot. Les instructions sont codées sur 1 ou 2 mots : le premier mot représente le codage de l'opération (`clr`, `ld`, `st`, `jmp`, `add`) ; le deuxième mot, s'il existe, contient une adresse ou bien une constante. Le codage des instructions est donné figure 14.2-a. Voici un exemple de programme écrit dans ce langage d'assemblage :

```
        ld #3
        st [8]
etiq:   add [8]
        jmp etiq
```

En supposant le programme chargé à partir de l'adresse 0, les adresses étant des adresses de mots, l'adresse associée au symbole `etiq` est 4. En supposant que la taille d'un mot est de 4 bits, la figure 14.2-b donne la représentation en mémoire du programme précédent après assemblage et chargement en mémoire à partir de l'adresse 0 (Cf. Chapitre 18 pour plus de détails).

Remarque : En fixant la taille d'un mot nous avons figé la taille maxi-

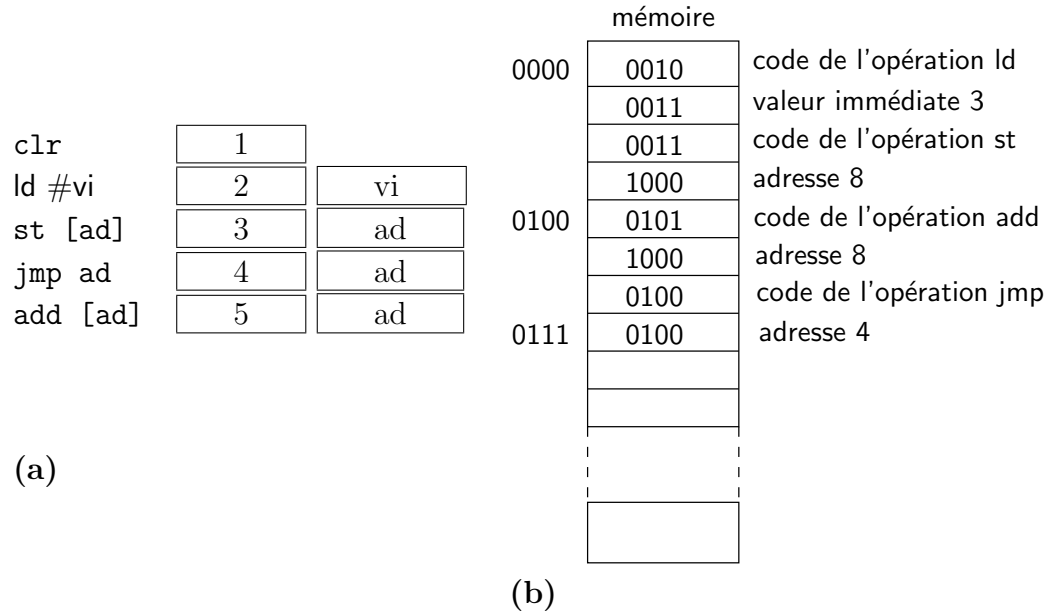


FIG. 14.2 – (a) Codage des instructions ; (b) représentation en mémoire d'un programme en langage machine

male de la mémoire puisqu'une adresse est codée sur un mot. La mémoire du processeur a ainsi une taille maximale de 16 mots.

Dans cet exemple, l'exécution du programme commence par l'interprétation de la première instruction, dont le code est en mémoire à l'adresse 0. Ce code étant celui de l'instruction `ld`, l'interprète lit une information supplémentaire dans le mot d'adresse 1. Cette valeur est alors chargée dans le registre `ACC`. Finalement, le compteur programme (`PC`) est modifié de façon à traiter l'instruction suivante.

Nous adoptons un point de vue fonctionnel, en considérant les ressources du processeur comme les variables d'un programme. L'algorithme d'interprétation des instructions est donné figure 14.3.

Nous montrons dans la suite comment réaliser cet algorithme par un circuit. Nous allons ainsi définir la partie opérative puis la partie contrôle qui la pilote.

3. Une réalisation du processeur

Pour chaque instruction du langage machine, nous commençons par nous poser les questions : *de quel matériel avons-nous besoin ?*, *comment organiser le flot des données pour cette instruction ?* Nous retrouvons là des questions très proches de celles résolues dans le chapitre 11. Les réponses à ces questions permettent de définir la partie opérative. Puis nous étudions la façon de réaliser les différentes étapes constituant l'exécution d'une instruction du langage machine ; nous définissons ainsi la partie contrôle.

```
lexique :
    entier4 : le type entiers représentés sur 4 bits
        { les opérations arithmétiques sont donc modulo 16 }
    PC, Acc : des entier4
    tailleMem : l'entier 16
    Mem : un tableau sur [0..tailleMem-1] d'entier4
algorithme d'interprétation des instructions :
    PC ⟵ 0
    tant que vrai
        selon Mem[PC]
            clr : Acc ⟵ 0 ; PC ⟵ PC + 1
            ld : Acc ⟵ Mem[PC + 1] ; PC ⟵ PC + 2
            st : Mem[Mem[PC + 1]] ⟵ Acc ; PC ⟵ PC + 2
            jmp : PC ⟵ Mem[PC + 1]
            add : Acc ⟵ Acc + Mem[Mem[PC + 1 ]] ; PC ⟵ PC + 2
```

FIG. 14.3 – Algorithme d'interprétation du langage machine

L'amélioration d'une réalisation matérielle s'exprime en termes de place occupée par le circuit et de temps de calcul. Nous donnons à la fin de ce paragraphe quelques éléments permettant d'améliorer le schéma obtenu selon ces critères et nous envisageons les conséquences d'une extension du jeu d'instructions.

3.1 Définition de la partie opérative

A partir du jeu d'instructions, on définit le flux des données nécessaire, les opérations que doit réaliser l'UAL, celles affectant le contenu des registres et les opérations concernant la mémoire. Cette étude permet petit à petit de construire la partie opérative et de préciser les opérations élémentaires qu'elle peut réaliser : les *microactions*.

Nous notons : $A \rightsquigarrow B$ le fait que le contenu de A doit pouvoir être transféré en B sans distinguer les cas où A ou B sont des bus ou des registres ; A $\Longleftarrow$ B op C une microaction qui permet de stocker dans A le résultat de l'opération op réalisée sur les opérandes B et C.

On arrive ainsi à la partie opérative décrite par la figure 14.4, la table 14.5 résumant l'ensemble des microactions ainsi que les transferts mis en jeu et les commandes associées pour la partie opérative.

Nous avons indiqué que le processeur est relié à la mémoire par le bus adresses (BusAd) et le bus données (BusDon). On dispose des transferts : BusDon $\rightsquigarrow$ Mem[BusAd] (écriture mémoire) et Mem[BusAd] $\rightsquigarrow$ BusDon (lecture mémoire).

Tout d'abord, il faut assurer le transfert de l'instruction courante (repérée par PC) de la mémoire vers le registre instruction (RI) de la partie opérative.

Dans notre exemple, une instruction est formée du code de l'opération à réaliser, plus éventuellement une valeur ou une adresse. Le code opération, les valeurs et les adresses sont tous codés sur 1 mot. Nous choisissons de découper le registre RI en deux registres RI1 et RI2, le premier contenant le code opération et le second l'information additionnelle (valeur immédiate ou adresse). D'où les transferts PC $\rightsquigarrow$ BusAd, BusDon $\rightsquigarrow$ RI1 (respectivement BusDon $\rightsquigarrow$ RI2), couplés avec l'opération de lecture de la mémoire. Le résumé des microactions associées se trouve dans les lignes 1 et 2 de la table 14.5.

Pour l'instruction `clr`, il faut pouvoir forcer la valeur 0 dans le registre ACC : cette opération peut être réalisée par une commande de remise à zéro du registre (ligne 3 de la table 14.5).

L'instruction `ld #vi` nécessite un transfert de la valeur immédiate `vi` dans l'accumulateur. La valeur `vi` est stockée dans le registre RI2 ; d'où le transfert RI2 $\rightsquigarrow$ ACC (ligne 4 de la table 14.5).

L'instruction `st [ad]` nécessite un transfert de la valeur de ACC vers la mémoire ; ce transfert a comme intermédiaire le bus données. Le transfert a lieu à l'adresse qui a été stockée dans RI2. D'où : RI2 $\rightsquigarrow$ BusAd et ACC $\rightsquigarrow$ BusDon, transferts couplés avec l'opération d'écriture mémoire (ligne 5 de la table 14.5).

L'instruction `jmp ad` nécessite un transfert de la valeur `ad` dans le registre PC. L'information `ad` étant dans RI2 : RI2 $\rightsquigarrow$ PC (ligne 6 de la table 14.5).

L'instruction `add [ad]` nécessite un transfert des valeurs de ACC et du mot mémoire d'adresse `ad` (stockée dans RI2) vers les deux entrées de l'UAL, et un transfert de la sortie de l'UAL vers l'accumulateur. La valeur provenant de la mémoire passe par le bus données d'où : ACC $\rightsquigarrow$ UAL, RI2 $\rightsquigarrow$ BusAd, BusDon $\rightsquigarrow$ UAL, UAL $\rightsquigarrow$ ACC et l'opération de lecture mémoire. Evidemment l'UAL doit disposer d'une commande d'addition de deux valeurs (ligne 7 de la table 14.5).

De plus, pour assurer le passage au mot suivant, il faut pouvoir incrémenter le compteur de programme, d'où : PC $\rightsquigarrow$ UAL et UAL $\rightsquigarrow$ PC, l'UAL disposant d'une commande d'incrémentation d'une de ses entrées (ligne 8 de la table 14.5).

Enfin il faut pouvoir initialiser le compteur de programme avec l'adresse de la première instruction (0 dans notre exemple) : d'où une commande de remise à zéro du registre PC (ligne 9 de la table 14.5).

3.2 Description de l'automate de contrôle

L'exécution d'une microaction est provoquée par l'activation des commandes correspondantes et c'est à la partie contrôle que revient le rôle d'activer ces commandes au bon moment.

L'automate de contrôle du processeur est donné dans la figure 14.6. Les entrées de cet automate sont des informations en provenance de la partie opérative : conditions portant sur le code opération courant c'est-à-dire le contenu du registre RI1.

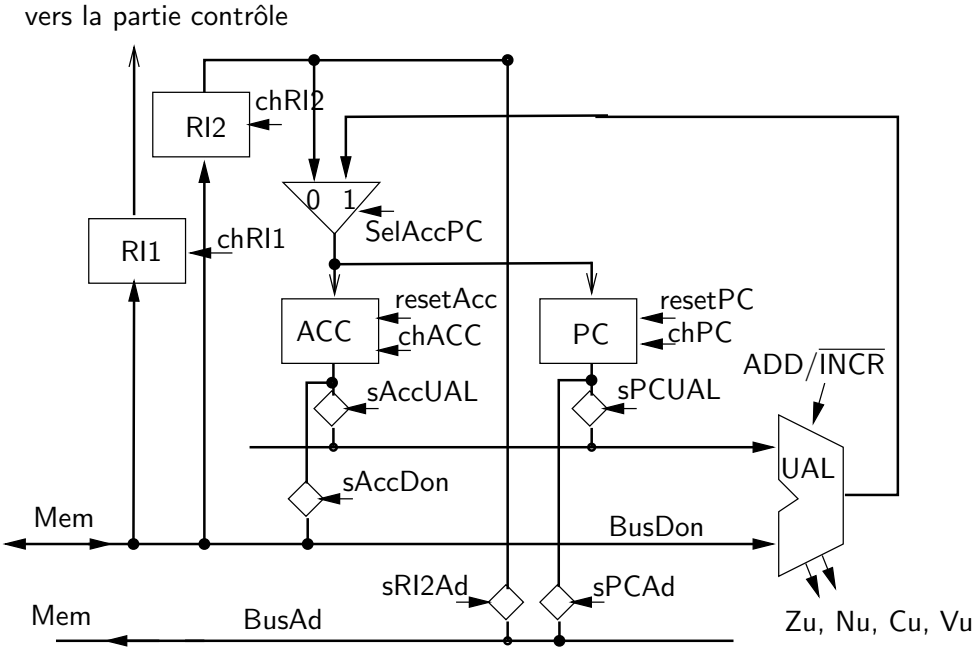


FIG. 14.4 – Une partie opérative possible pour le processeur

	microactions	transferts utilisés	commandes pour la P.O.
1	RI1 $\Longleftarrow$ Mem[PC]	PC $\rightsquigarrow$ BusAd Mem[BusAd] $\rightsquigarrow$ BusDon BusDon $\rightsquigarrow$ RI1	sPCAd, SelMem, I/$\overline{\text{e}}$ chRI1
2	RI2 $\Longleftarrow$ Mem[PC]	PC $\rightsquigarrow$ BusAd Mem[BusAd] $\rightsquigarrow$ BusDon BusDon $\rightsquigarrow$ RI2	sPCAd, SelMem, I/$\overline{\text{e}}$ chRI2
3	Acc $\Longleftarrow$ 0		resetAcc
4	Acc $\Longleftarrow$ RI2	RI2 $\rightsquigarrow$ Acc	$\overline{\text{SelAccPC}}$, chAcc
5	Mem[RI2] $\Longleftarrow$ Acc	RI2 $\rightsquigarrow$ BusAd Acc $\rightsquigarrow$ BusDon BusDon $\rightsquigarrow$ Mem[BusAd]	sRI2Ad, sAccDon, SelMem, $\overline{\text{I/e}}$
6	PC $\Longleftarrow$ RI2	RI2 $\rightsquigarrow$ PC	$\overline{\text{SelAccPC}}$, chPC
7	Acc $\Longleftarrow$ Acc + Mem[RI2]	Acc $\rightsquigarrow$ UAL(1) RI2 $\rightsquigarrow$ BusAd Mem[BusAd] $\rightsquigarrow$ BusDon BusDon $\rightsquigarrow$ UAL(2) UAL $\rightsquigarrow$ Acc	sAccUAL, sRI2Ad, SelMem, I/$\overline{\text{e}}$, add/$\overline{\text{incr}}$, SelAccPC chAcc
8	PC $\Longleftarrow$ PC + 1	PC $\rightsquigarrow$ UAL(1) UAL $\rightsquigarrow$ PC	sPCUAL, $\overline{\text{add/incr}}$, SelAccPC, chPC
9	PC $\Longleftarrow$ 0		resetPC

FIG. 14.5 – Commandes et transferts associés aux microactions. Les notations UAL(1) et UAL(2) désignent l'entrée 1 et l'entrée 2 de l'UAL.

Une première version consisterait à effectuer le test de chacun des codes conditions. Après avoir lu le code de l'instruction, nous obtiendrions un choix à 5 cas (`clr`, `ld`, `st`, `jmp` et `add`). En remarquant que 4 instructions demandent la lecture du mot suivant (Etats E2, E4) nous pouvons regrouper les traitements. C'est ainsi que nous organisons d'emblée les tests en 2 cas : instruction `clr` ou non.

L'automate décrit dans la figure 14.6 est la traduction de l'algorithme du paragraphe 2. en paramétrant certains traitements. On note que pour toute instruction sauf `jmp` le compteur de programme doit être incrémenté (Etat E9) afin que la partie contrôle puisse passer à l'instruction suivante ; dans le cas de l'instruction `jmp`, le compteur de programme est chargé avec l'adresse de la cible du branchement (Etat E6).

3.3 Réalisation matérielle de la partie contrôle

Maintenant que l'automate de contrôle a été décrit en terme des microactions et des valeurs du code opération courant, nous allons détailler sa réalisation matérielle.

Les entrées de l'automate sont des informations en provenance de la partie opérative, c'est-à-dire les 4 bits du registre RI1 ; notons ri_3, ri_2, ri_1, ri_0 le contenu de RI1, ri_0 étant le bit de poids faible. La transition de l'état E1 vers l'état E3 est conditionnée par : $\overline{ri_3}.\overline{ri_2}.\overline{ri_1}.ri_0$ car le code de l'instruction `clr` est 0001. Pour un jeu d'instructions plus complet, les entrées de la partie contrôle peuvent être plus nombreuses.

Les sorties de l'automate sont les commandes de la partie opérative. On peut les représenter par une valuation du vecteur booléen : (SelMem, I/$\overline{\text{e}}$, resetPC, resetAcc, chRI1, chRI2, chAcc, chPC, add/$\overline{\text{incr}}$, SelAccPC, sAccUAL, sPCUAL, sAccDon, sRI2Ad, sPCAd). A l'état E1 est associé le vecteur de sortie : (1, 1, 0, 0, 1, 0, 0, 0, φ, φ, φ, φ, φ, 0, 1).

L'exercice E14.5 propose de réaliser la synthèse complète de cet automate.

Cet automate est cadencé par une horloge dont la période correspond au temps nécessaire à l'exécution de la microaction la plus longue ; ce temps est appelé *temps de cycle*. Quand on lit qu'un processeur a une horloge à 500 Megahertz, on peut penser que le coeur du processeur a un temps de cycle de 2 nanosecondes. Dans l'exemple, la microaction la plus longue est Acc $\Longleftarrow$ Acc + Mem[RI2], qui comporte une addition et un accès mémoire alors que les autres microactions ne comportent que l'un des deux.

4. Critique et amélioration de la solution

Après avoir produit une solution, il est judicieux de se demander si elle peut être améliorée. Il faut alors savoir selon quels critères physiques : fréquence de fonctionnement, surface du circuit, puissance électrique dissipée, etc.

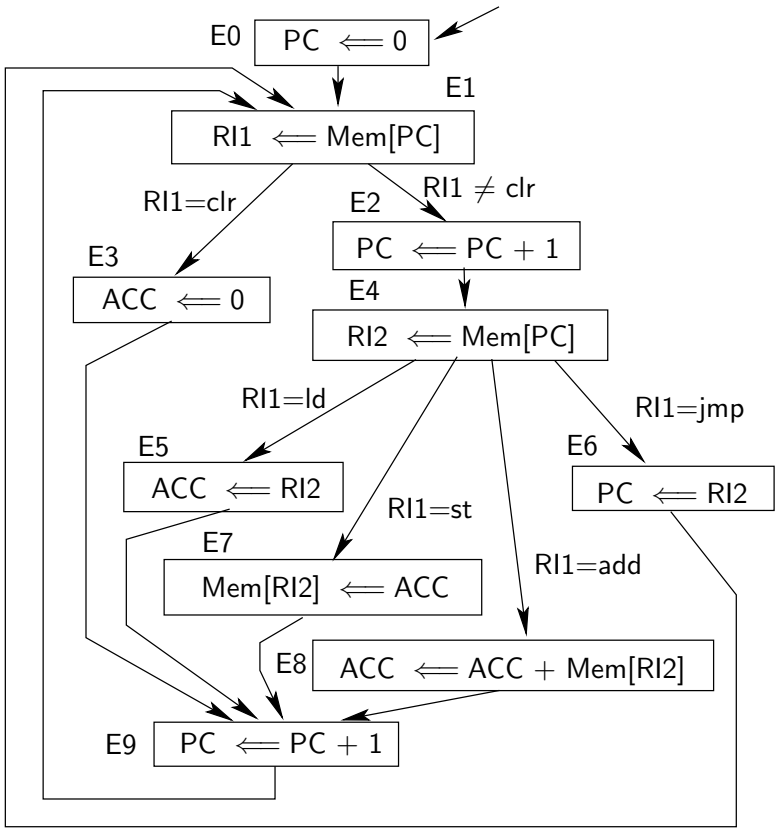


FIG. 14.6 – Un premier automate de contrôle pour le processeur

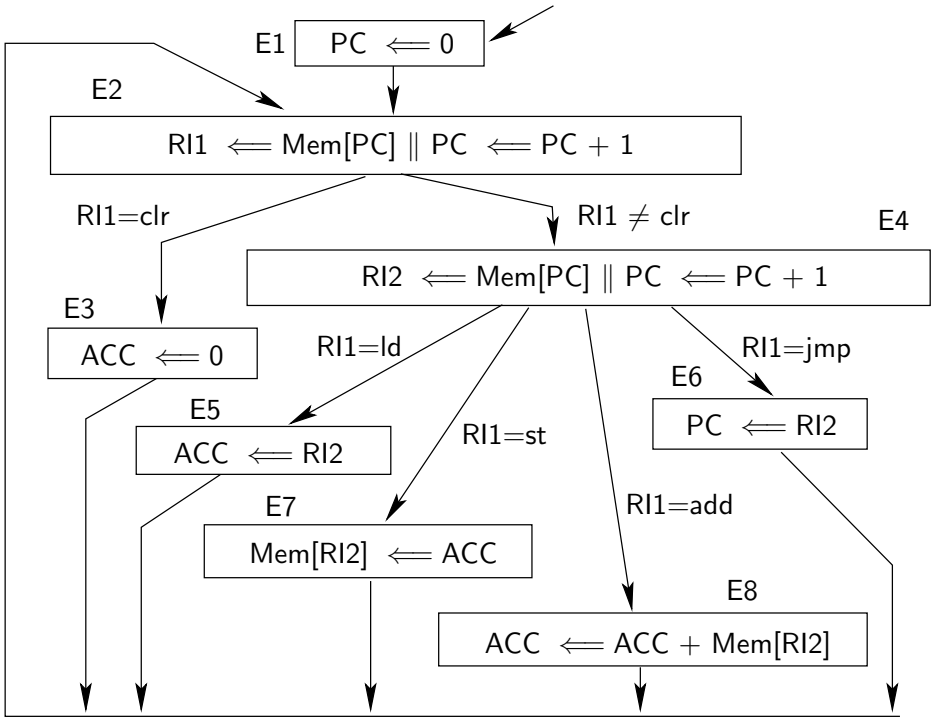


FIG. 14.7 – Un autre automate de contrôle pour le processeur

En général, on essaie de rendre le processeur le plus rapide possible en diminuant le temps d'exécution des instructions machine. Le temps d'exécution d'une instruction donnée est égal à $N \times C$ où N est le nombre d'états nécessaires à l'exécution de l'instruction et C est le temps de cycle. On peut donc diminuer soit le temps de cycle, soit le nombre d'états nécessaires à l'exécution d'une instruction. Dans notre exemple, l'exécution de l'instruction `add`, par exemple, prend 5 cycles d'horloge.

On peut aussi chercher à économiser le matériel : utiliser le moins possible de registres ou de bus dans la partie opérative.

Une bonne réalisation est souvent le résultat d'un compromis entre tous ces aspects. Nous donnons ici quelques idées d'améliorations sur l'exemple précédent.

4.1 Décomposition des microactions

Pour diminuer le temps de cycle, on peut éviter de grouper un accès à la mémoire et une opération UAL dans la même microaction. Dans notre exemple, on pourrait imaginer un registre tampon T connecté au bus données et à l'entrée de l'UAL. L'état E8 pourrait alors être divisé en 2 étapes correspondant aux microactions T $\Longleftarrow$ Mem[RI2] et ACC $\Longleftarrow$ ACC + T. Le temps de cycle correspondrait alors au temps d'accès à la mémoire. Cela ne fait pas gagner de temps pour l'exécution de l'instruction d'addition mais les autres instructions sont, elles, exécutées plus rapidement.

On peut, par ailleurs, se demander si le registre RI2 est bien nécessaire, c'est-à-dire se poser la question : peut-on faire transiter directement la valeur ou l'adresse associée à un code opération vers le point où elle est nécessaire ? Dans le cas de l'instruction `ld` (respectivement `jmp`) la microaction Acc $\Longleftarrow$ Mem[PC] (respectivement PC $\Longleftarrow$ Mem[PC]) convient. Pour cela, il faut connecter le bus données directement aux multiplexeurs d'entrée de chacun des registres Acc et PC, ce qui est parfaitement possible. En revanche, dans le cas des instructions `st` et `add`, il est indispensable de disposer d'un registre intermédiaire pour stocker l'adresse qui est utilisée pour l'accès mémoire, en écriture pour `st` et en lecture pour `add`. Cette modification permettrait donc de gagner un état lors de l'exécution des instructions `ld` et `jmp` et seulement dans ces deux cas.

4.2 Parallélisation de microactions

Une autre façon d'améliorer l'efficacité du processeur consiste à effectuer en parallèle certains traitements, quitte à ajouter des opérateurs de base ou des registres dans la partie opérative. Par exemple, pour la machine précédente il est possible de charger un mot de la mémoire dans RI1 et d'incrémenter en parallèle PC afin qu'il soit prêt pour la lecture suivante, les microactions RI1 $\Longleftarrow$ Mem[PC] et PC $\Longleftarrow$ PC + 1 n'utilisant pas le même matériel. Etant

données deux microactions A1 et A2, A1 ‖ A2 dénote leur activation en parallèle, c'est-à-dire l'activation de l'ensemble des commandes associées à l'une et à l'autre dans le même cycle d'horloge.

La figure 14.7 décrit un nouvel automate de contrôle pour le processeur. L'incrémentation du compteur de programme est exécutée parallèlement à la lecture d'un mot mémoire puisque le matériel (la partie opérative) le permet. Cela ne pose pas de problème pour le traitement de l'instruction `jmp` car la valeur ainsi stockée dans le registre PC est écrasée par l'adresse adéquate ultérieurement. Le temps d'exécution de l'instruction `add` est maintenant de 3 cycles d'horloge. De façon générale, le temps d'exécution de toutes les instructions machine a diminué car le nombre d'états traversés est plus petit.

Ce genre d'optimisation doit être fait avec précaution. Supposons que la mémoire associée au processeur soit lente, qu'un accès mémoire dure plus d'un cycle d'horloge ; la mémoire émet alors un signal fin-accès lorsque la donnée lue est disponible sur le bus données. La microaction de lecture RI1 $\Longleftarrow \overline{\text{Mem[PC]}}$ est alors réalisée dans un état comportant une boucle sur le signal $\overline{\text{fin-accès}}$ et le contrôleur passe à l'état suivant sur l'entrée fin-accès. La mise en parallèle de cette microaction avec PC $\Longleftarrow$ PC + 1 peut être incorrecte ; le compteur programme peut avoir été incrémenté avant que la lecture ne soit réalisée et Mem[PC] peut alors correspondre à un mot suivant celui auquel on devait accéder.

4.3 Paramétrisation des commandes

Une autre amélioration consiste à essayer de minimiser le nombre d'états de la partie contrôle. Ceci rend la réalisation de la partie contrôle plus compacte et peut aussi améliorer l'efficacité du processeur.

La *paramétrisation* consiste à ajouter une partie de matériel à la frontière de la partie contrôle et de la partie opérative, ce qui permet de regrouper des traitements dans la partie contrôle.

Par exemple, dans la partie contrôle de la figure 14.7, les états E5 et E6 se ressemblent beaucoup. Nous pouvons les rassembler en un seul état comportant la microaction ACCouPC $\Longleftarrow$ RI2 dont la commande associée est chACCouPC. Il suffit alors d'ajouter entre la partie contrôle et la partie opérative le circuit combinatoire donné dans la figure 14.8. Lorsque la partie contrôle active la commande chACCouPC la commande de chargement du registre adéquat est activée selon la valeur du code opération, contenue dans RI1.

L'exemple précédent est simpliste. En réalité, il existe nombre de cas où la paramétrisation fait gagner un grand nombre d'états. Nous en montrons une utilisation dans le paragraphe suivant.

Le matériel ajouté peut constituer une partie importante du processeur ; par exemple, dans un 68000, la partie contrôle, la partie opérative et la partie servant à la paramétrisation constituent chacune de l'ordre d'un tiers du matériel.

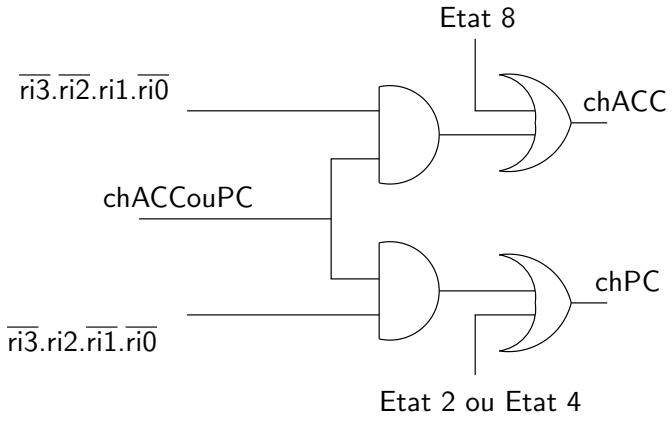


FIG. 14.8 – Production des commandes chACC et chPC selon la commande chACCouPC, le code opération et l'état de l'automate de contrôle. chACCouPC est émise par la partie contrôle, chACC et chPC sont reçues par la partie opérative. chACC est activée lorsque l'automate de contrôle est dans l'état 8 ou lorsque chACCouPC est vraie alors que l'instruction courante est `ld` (codée par 2 en décimal). chPC est activée lorsque l'automate de contrôle est dans un des états 2 ou 4 ou lorsque chACCouPC est vraie alors que l'instruction courante est `jmp` (codée 4 en décimal).

5. Extensions du processeur

Dans ce paragraphe, nous étudions des extensions de deux types pour le processeur : augmentation du nombre de registres utilisables par le programmeur et extension du jeu d'instructions.

5.1 Augmentation du nombre de registres utilisateur

Imaginons que notre processeur ait 16 registres de données `ACC0`, `ACC1`, ..., `ACC15` au lieu d'un seul accumulateur.

Les instructions `clr`, `ld`, `st` et `add` ont alors un paramètre supplémentaire. L'instruction `jmp` reste inchangée. La syntaxe de ces instructions peut être :

`clr ACCi`	mise à zéro du registre `ACCi`
`ld #vi, ACCi`	chargement de la valeur `vi` dans le registre `ACCi`
`st ACCi, [ad]`	stockage du contenu de `ACCi` à l'adresse ad en mémoire
`add [ad], ACCi`	stockage de la somme du contenu de la mémoire d'adresse ad et du contenu de `ACCi` dans `ACCi`.

Le codage des instructions `ld`, `st` et `add` demande 3 mots (toujours de 4 bits) : un pour le code opération, un pour la valeur immédiate ou l'adresse et un troisième pour le numéro du registre. Le codage de l'instruction `clr` en demande 2 (Cf. Figure 14.9).

La figure 14.10 décrit une nouvelle partie opérative pour le processeur

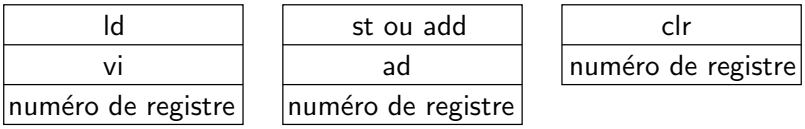


FIG. 14.9 – Codage des instructions pour une extension de la machine

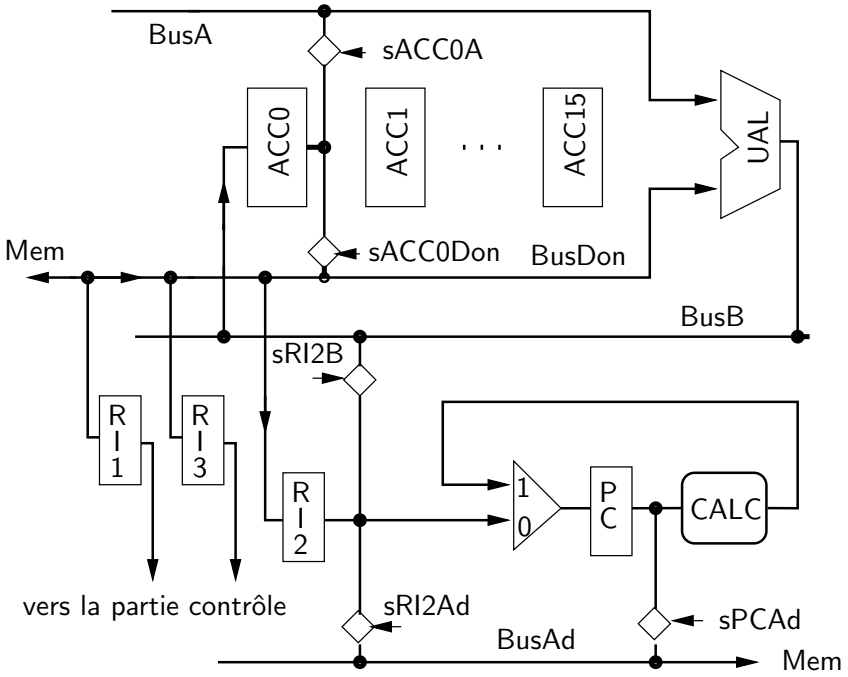


FIG. 14.10 – Partie opérative complétée pour le processeur étendu

étendu. On y trouve évidemment les 16 registres `ACCi`, $i = 0, .., 15$, un registre supplémentaire RI3 qui permet de stocker le numéro du registre lu en mémoire lors du traitement d'une instruction `clr`, `ld`, `st` ou `add`. Les bits de ce registre sont de nouvelles entrées pour la partie contrôle.

Cette partie opérative comporte deux parties de calcul : une sur les données et une sur les adresses. On pourrait utiliser l'unité de calcul sur les adresses pour gérer, par exemple, la mise à jour d'un pointeur de pile.

Sans utiliser de technique de paramétrisation, la partie contrôle comporterait un état par registre pour chacune des instructions `clr`, `ld`, `st` et `add`. Par exemple, pour l'instruction `clr`, on aurait les microactions : $\mathsf{ACC}_0 \Longleftarrow 0$, $\mathsf{ACC}_1 \Longleftarrow 0$, ..., $\mathsf{ACC}_{15} \Longleftarrow 0$. L'automate a alors beaucoup d'états et le circuit à synthétiser est complexe et volumineux.

Lors de l'exécution des instructions `clr`, `ld`, `st` et `add`, le registre RI3 contient la valeur `i`, numéro du registre `ACCi` concerné. Les 16 états correspondant à la mise à zéro des 16 registres peuvent être remplacés par un seul état comportant la microaction $\mathsf{ACC}_{RI3} \Longleftarrow 0$ qui signifie : le registre ACC dont le

microaction	commandes
$ACC_{RI3} \Longleftarrow 0$	resetACC
$ACC_{RI3} \Longleftarrow RI2$	sRI2B, chACC
$Mem[RI2] \Longleftarrow ACC_{RI3}$	sACCDon, sRI2Ad, SelMem, Ecr
$ACC_{RI3} \Longleftarrow ACC_{RI3} + Mem[RI2]$	sACCA, sRI2Ad, SelMem, Lec, chACC

FIG. 14.11 – Commandes associées aux nouvelles microactions

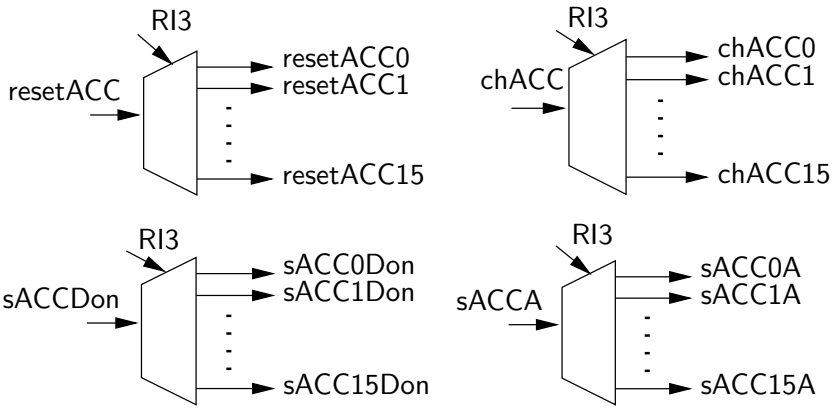


FIG. 14.12 – Réalisation de commandes paramétrées

numéro est la valeur contenue dans RI3 est remis à zéro. De même, les 16 états correspondant au chargement des 16 registres peuvent être remplacés par un état avec la microaction : $ACC_{RI3} \Longleftarrow RI2$. Pour l'instruction `st`, on va définir la microaction : $Mem[RI2] \Longleftarrow ACC_{RI3}$ et pour l'instruction `add` la microaction : $ACC_{RI3} \Longleftarrow ACC_{RI3} + Mem[RI2]$. La table de la figure 14.11 donne les commandes associées à chacune de ces nouvelles microactions.

Il faut ajouter le matériel nécessaire pour élaborer les commandes resetACC0, ..., resetACC15, chACC0, ..., chACC15, sACCDon0, ...sACCDon15, sACC0A, ..., sACC15A à partir des commandes resetACC, chACC, sACCDon, sACCA et du contenu de RI3. La figure 14.12 décrit ces circuits réalisés avec un décodeur.

La description de la partie contrôle paramétrée est ainsi la même que pour un seul registre ACC.

5.2 Extension du jeu d'instructions

Des extensions simples comme l'ajout d'opérations (soustraction, conjonction logique, incrémentation d'un registre) sont aisément réalisées en compliquant l'UAL. La commande de l'UAL peut alors être réalisée directement à partir d'un sous-ensemble du registre instruction. Il faut toutefois faire attention aux états où l'UAL est utilisée pour des calculs à usage interne, comme par exemple : $PC \Longleftarrow PC + 1$.

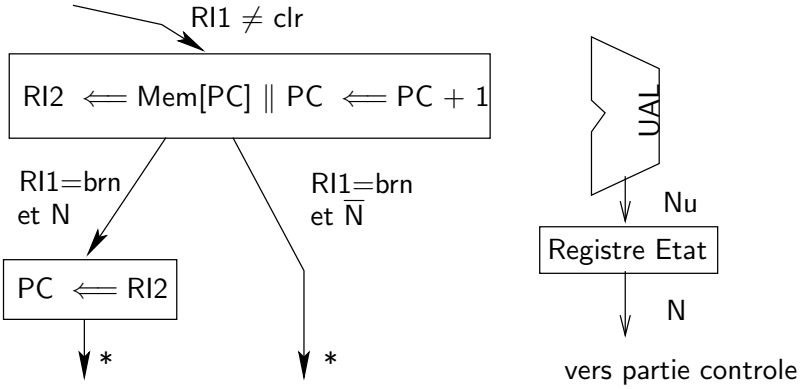


FIG. 14.13 – Extrait de la partie contrôle pour le traitement de l'instruction `brn`. Les flèches notées * ont pour cible l'acquisition de l'instruction suivante, c'est-à-dire l'état E2 de la figure 14.7.

Supposons que nous voulions maintenant ajouter des branchements conditionnels. Par exemple, on ajoute l'instruction `brn ad` dont l'effet est : si le résultat de l'opération précédente est négatif alors effectuer un branchement à l'adresse `ad` sinon passer à l'instruction suivante. Ce genre d'instruction utilise les codes de conditions calculés par l'UAL lors de l'exécution des opérations ; ici, il s'agit de l'indicateur de résultat négatif (N) que l'on peut supposer chargé lors de toute instruction arithmétique. En revanche, le bit N n'est pas chargé lors d'un calcul de type PC ⟸ PC + 1.

L'UAL est complétée avec un registre à 1 bit. Ce bit est souvent stocké dans le registre d'état. Le contenu de ce registre est accessible en permanence par la partie contrôle. La figure 14.13 décrit la portion de partie contrôle traitant l'instruction `brn`.

Les exercices E14.1, E14.2 et E14.3 suggèrent des modifications plus complètes pour le processeur étudié ci-dessus.

6. Exercices

E14.1 : Branchement conditionnel général

On veut ajouter au langage machine du processeur étudié dans ce chapitre une instruction `BRcc` où `cc` représente l'un des codes de conditions classiques de tout processeur. Etudier les conséquences de cette modification sur la partie opérative et la partie contrôle du processeur. On peut considérer un codage des différentes conditions `cc` sur 4 bits $b1$, $b2$, $b3$, $b4$ (ou prendre le codage d'un processeur existant). Réaliser, en particulier, le circuit combinatoire qui reçoit en entrée les bits $b1$, $b2$, $b3$, $b4$ du code opération et les 4 bits Z, N, C, V et délivre un bit `Br` disant si la condition donne effectivement lieu à un branchement ou non. Le bit `Br` est exploité par la partie contrôle pour

établir la nouvelle valeur du compteur programme. C'est encore une forme de paramétrisation, sur les comptes-rendus plutôt que sur les commandes.

E14.2 : Mode d'adressage relatif
Dans le langage machine du processeur étudié dans le présent chapitre, les branchements sont absolus, l'adresse cible du branchement est donnée dans l'instruction. On veut maintenant introduire des branchement relatifs, c'est-à-dire pour lesquels l'adresse cible du branchement est égale à la somme de PC et d'une valeur constante donnée dans l'instruction. On peut modifier la partie opérative de la figure 14.10 et remplacer le circuit CALC par un additionneur ou bien utiliser l'UAL pour réaliser l'addition. Etudier les modifications de la partie opérative et décrire la partie contrôle du processeur.

E14.3 : Appel de sous-programme
Ajouter dans le langage machine des instructions d'appel et de retour de sous-programme (ajouter les laisons entre PC et le bus données, ajouter un pointeur de pile). Une instruction d'appel de sous-programme demande la sauvegarde de l'adresse de retour (Cf. Paragraphe 1.4.3 du chapitre 12). Un processeur intégrant entre autres cette possibilité est décrit dans le chapitre 22.

E14.4 : Partie opérative vue comme un automate
Nous avons signalé au paragraphe 1.2 que la partie opérative peut être vue comme un automate. L'objectif de cet exercice est de préciser la définition de cet automate. Une telle approche pourrait être utilisée dans un langage de description de matériel.

La partie opérative du processeur (Cf. Paragraphe 3.1) est un automate à 15 fils d'entrée : chRI1, chRI2, SelAccPC, resetAcc, chACC, sAccUAL, sAccDon, resetPC, chPC, sPCUAL, sRI2Ad, sPCAd, ADD/$\overline{\text{INCR}}$, Lire. La commande Lire correspond à la conjonction : l/$\overline{\text{e}}$ et SelMem. La commande $\overline{\text{l/e}}$ et SelMem (Ecrire) ne modifie pas l'état du processeur, mais seulement celui de la mémoire. Elle n'est pas une entrée de la partie opérative.

15 fils d'entrées donnent 2^{15} entrées possibles. On ne peut donc pas décrire les transitions selon toutes les valeurs des entrées. On écrira : si chRI1 alors ... pour parler en fait des 2^{14} entrées pour lesquelles chRI1 vaut 1.

Définissons maintenant l'ensemble des états de l'automate. Il y a 4 registres de 4 bits : RI1, RI2, Acc et PC. Ces 16 bits définissent 2^{16} états. On va décrire le nouvel état en fonction de l'ancien de façon symbolique, en utilisant un algorithme. On peut en effet donner une description fonctionnelle de la fonction de transition à l'aide d'expressions conditionnelles définissant la valeur de chaque registre selon sa valeur précédente et les entrées courantes. On introduit des variables intermédiaires pour faciliter l'écriture (Cf. Figure 14.14).

Poursuivre la description fonctionnelle de cette partie opérative.

E14.5 : Synthèse d'une partie contrôle
L'objectif de cet exercice est la synthèse de la partie contrôle du processeur

```
< RI1, RI2, Acc, PC > : état de la partie opérative
< nouvelRI1, nouvelRI2, nouvelAcc, nouveauPC > :
    nouvel état de la partie opérative
sortieUAL, BusDon, BusAd : variables intermédiaires
{ Une partie de la fonction de transition de la partie opérative }
nouvelRI1 = si chRI1 alors BusDon sinon RI1
nouvelRI2 = si chRI2 alors BusDon sinon RI2
nouvelAcc = si resetAcc alors 0
              sinon si chACC alors
                    si selAccPC alors sortieUAL sinon RI2
                    sinon Acc
BusDon = si sAccDon alors Acc
           sinon si Lire alors Mem[BusAd] sinon non défini
```

FIG. 14.14 – Description symbolique d'une fonction de transition

état				inst	entrées (RInst)				nouvel état			
e_3	e_2	e_1	e_0		ri_3	ri_2	ri_1	ri_0	ne_3	ne_2	ne_1	ne_0
0	0	0	0		x	x	x	x	0	0	0	1
0	0	0	1	`clr`	0	0	0	1	0	0	1	1
0	0	0	1	autre	$\neq$0	0	0	1	0	0	1	0
0	0	1	0		x	x	x	x	0	1	0	0
0	0	1	1		x	x	x	x	1	0	0	1
0	1	0	0	`ld`	0	0	1	0	0	1	0	1
0	1	0	0	`st`	0	0	1	1	0	1	1	1
0	1	0	0	`jmp`	0	1	0	0	0	1	1	0

FIG. 14.15 – fonction de transition

donnée dans la figure 14.6 selon la technique étudiée au chapitre 10. L'automate a 10 états que l'on peut coder sur 4 bits e_3, e_2, e_1, e_0 ; l'état Ei est représenté par le codage binaire de l'entier i.

La table de la figure 14.15 donne une partie de la *fonction de transition*. Terminer cette table et réaliser la synthèse de cette fonction combinatoire avec des portes, ou un PLA (Cf. Chapitre 8). Remarquer au passage que l'automate de contrôle révèle une sous-spécification : rien n'est prévu si le code de l'instruction dans RI1 n'est pas un code valide. En fait, dans le cas de code invalide, lors du décodage de l'instruction une interruption est générée par le processeur (Cf. Chapitre 22).

La partie contrôle a 16 fils de sorties : SelMem, I/$\overline{\text{e}}$, resetPC, resetAcc, chRI1, chRI2, chAcc, chPC, add/$\overline{\text{incr}}$, SelAccPC, sAccUAL, sPCUAL, sAccDon, sRI2Ad, sPCAd (Cf. Paragraphe 3.2). La partie contrôle étant décrite par un automate de Moore, les sorties dépendent uniquement de l'état. Nous donnons dans la figure 14.16 une partie de la fonction de sortie de l'automate. Compléter cette

état				chRI1	SelAccPC	chAcc	ADD/$\overline{\text{INCR}}$	...
e_3	e_2	e_1	e_0					
0	0	0	1	1	ϕ	0	ϕ	...
0	0	1	0	0	0	0	0(incr)	...
1	0	0	0	0	1	1	1(add)	...

FIG. 14.16 – fonction de sortie

nom de l'instruction	assembleur	sémantique
addition	`ADD S1, S2, Rd`	Rd $\longleftarrow$ S1 + S2
soustraction	`SUB S1, S2, Rd`	Rd $\longleftarrow$ S1 - S2
soustraction bis	`SUBR S1, S2, Rd`	Rd $\longleftarrow$ S2 - S1
conjonction	`AND S1, S2, Rd`	Rd $\longleftarrow$ S1 $\wedge$ S2
disjonction	`OR S1, S2, Rd`	Rd $\longleftarrow$ S1 $\vee$ S2
ou exclusif	`XOR S1, S2, Rd`	Rd $\longleftarrow$ S1 $\oplus$ S2
charg. d'un registre	`LOAD Rx, S2, Rd`	Rd $\longleftarrow$ Mem[Rx+S2]
stockage en mémoire	`STORE Rx, S2, Rs`	Mem[Rx+S2] $\longleftarrow$ Rs
branchement indexé	`JMP COND, S2, Rx`	PC $\longleftarrow$ Rx + S2 si COND
branchement relatif	`JMPR COND, Y`	PC $\longleftarrow$ PC + Y si COND
charg. bit poids forts	`LDHI Rd, Y`	Rd_{31-13} $\longleftarrow$ Y, Rd_{12-0} $\longleftarrow$ 0

FIG. 14.17 – Instructions d'un processeur inspiré du SPARC

table et synthétiser la fonction combinatoire avec des portes ou un PLA.

En utilisant des bascules D pour représenter l'état de l'automate, dessiner le circuit synthétisant la partie contrôle toute entière.

E14.6 : Interprète d'un langage machine type SPARC

Ce problème a pour but de décrire l'interprète du langage machine d'un processeur imaginaire inspiré du processeur SPARC.

Les adresses et les données sont sur 32 bits. Le processeur comporte 31 registres (notés R1, ..., R31) de 32 bits chacun et un registre spécial noté R0 contenant 0 en opérande source et non modifiable.

La table 14.17 décrit le jeu d'instructions du processeur et la figure 14.18 précise le codage des différentes instructions. S1, Rx, Rd, Rs désignent des registres : un des registres Ri, i=0, ..., 31. S2 désigne un registre ou une valeur immédiate (sur 13 bits). Y représente une valeur immédiate sur 19 bits. On peut ajouter les instructions `ADDcc`, `SUBcc`, `SUBRcc`, `ANDcc`, `ORcc` et `XORcc` qui ont le même effet que `ADD`, `SUB`, `SUBR`, `AND`, `OR` et `XOR` avec mise à jour des codes de conditions. Toute opération mettant en jeu des valeurs codées sur moins de 32 bits (13 pour S2, 19 pour Y) doit prendre en compte l'extension du signe.

La figure 14.19 décrit la partie opérative. Cette partie opérative comporte 3 bus internes. Les entrées de l'UAL sont connectées aux bus Bus1 et Bus2 et sa sortie au bus BusRes. Le circuit combinatoire ext-sign extrait du registre instruction RI la valeur immédiate et l'étend sur 32 bits afin de l'envoyer sur

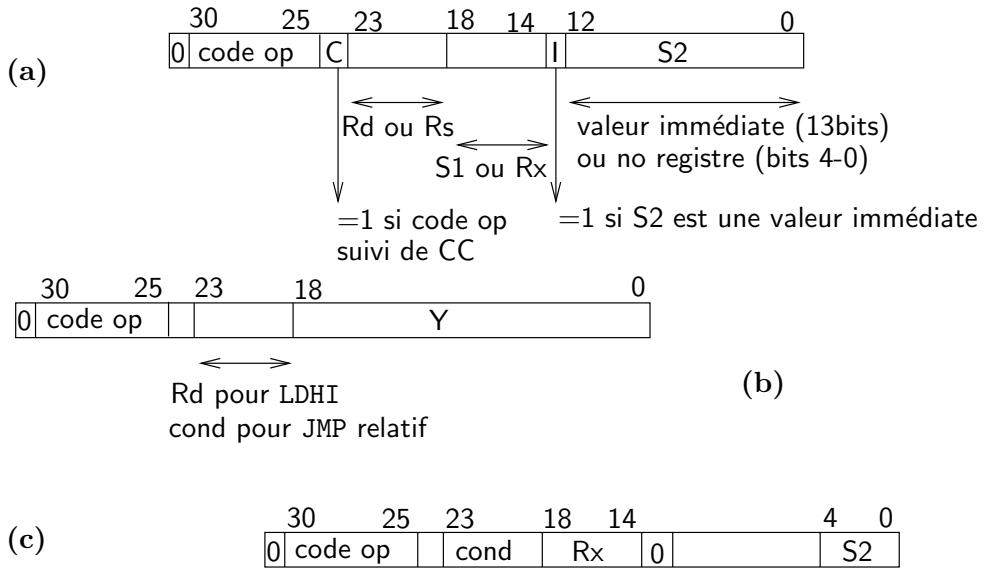


FIG. 14.18 – Codage des instructions d'un processeur inspiré du SPARC : a) instructions `ADD, ADDcc, SUB, SUBcc, SUBRm SUBRcc, AND, OR, XOR, LOAD, STORE`; b) instructions `LDHI` et `JMPR`; c) instruction `JMP`.

Bus2. Le circuit combinatoire DEC effectue un décalage d'une valeur circulant sur Bus2; elle est utile pour l'exécution de l'instruction LDHI. Le compteur programme PC peut être incrémenté, ou chargé à partir du résultat d'un calcul (via BusRes). La valeur circulant sur BusAd peut être le résultat d'un calcul réalisé par l'UAL, une valeur stockée précédemment dans le registre interne T ou le contenu de PC. Voici quelques pistes à explorer :

1. Ajouter sur la partie opérative les commandes nécessaires.
2. Décrire la partie contrôle du processeur sous forme d'une machine séquentielle avec actions. Pour chaque microaction utilisée, vérifier qu'elle est effectivement exécutable par la partie opérative fournie. Pour cela, donner en détail l'ensemble des commandes qui lui sont associées.
3. Dessiner les circuits ext-sign, DEC et COND.
4. Décrire le circuit permettant de commander les registres : accès aux bus et chargement des registres.
5. Choisir un codage pour les opérations et pour les conditions arithmétiques, et réaliser le séquenceur.

E14.7 : Interprète d'un langage machine type 68000

Ce problème a pour but de décrire l'interprète du langage machine d'un processeur imaginaire inspiré du processeur 68000. La principale différence avec le problème précédent vient de l'existence d'un nombre important de modes d'adressage pour ce processeur.

Le programmeur dispose d'un ensemble de registres notés Di, i=0, ..., max. Le registre Dmax joue un rôle particulier, celui de *pointeur de pile* (aussi noté SP). La partie opérative comporte deux bus internes BusSource et BusRésultat

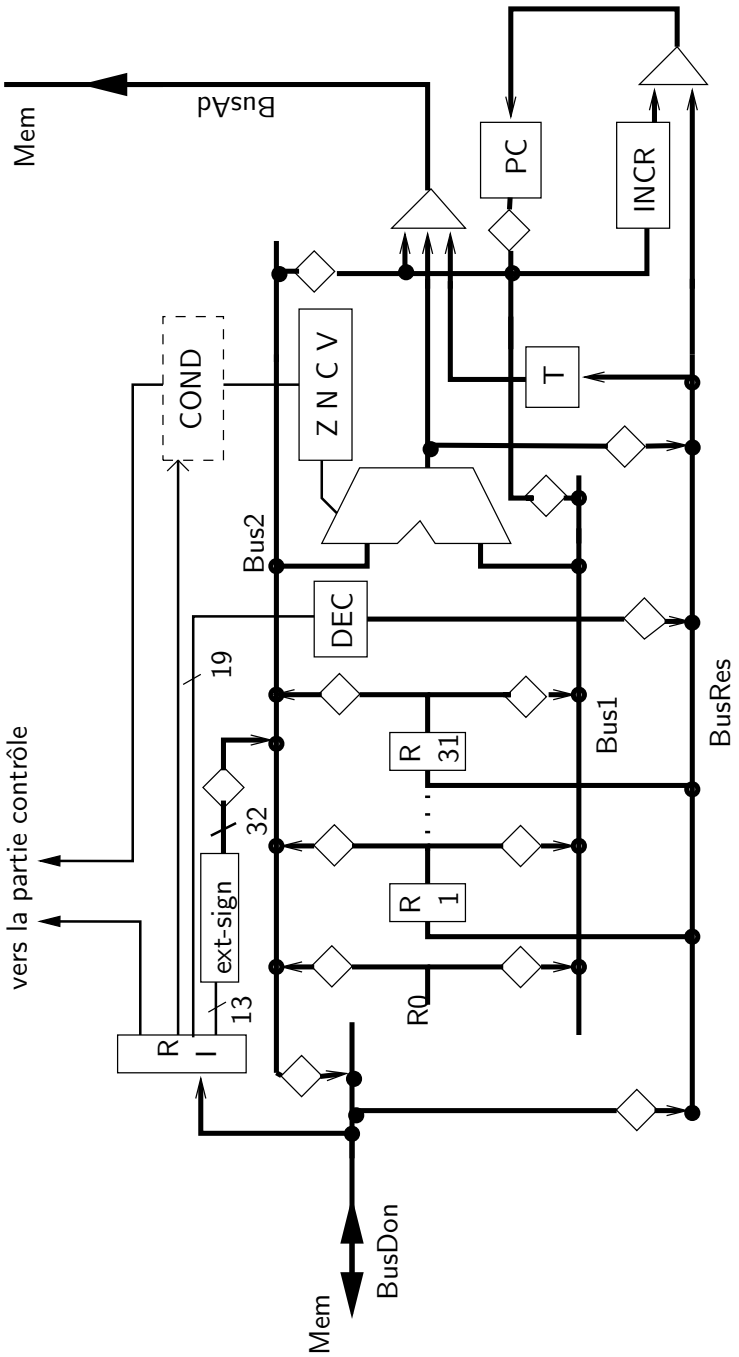


FIG. 14.19 – Partie opérative d'un processeur inspiré du SPARC

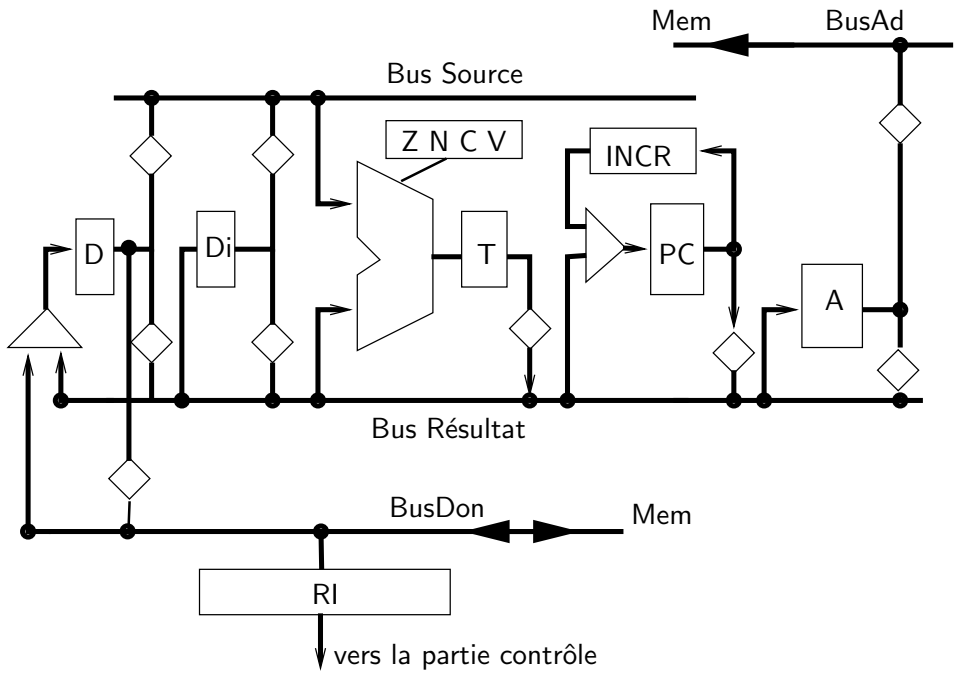


FIG. 14.20 – Partie opérative d'un processeur inspiré du 68000

et trois registres internes (non visibles par le programmeur) : A, D et T (Cf. Figure 14.20). T est un registre tampon servant à stocker le résultat d'un calcul en sortie de l'UAL, les deux autres (A, D) vont permettre de faciliter l'interprétation des instructions. Les registres et les bus ont 32 bits.

Une instruction peut être codée sur 1 ou 2 mots de 32 bits. Le deuxième mot éventuel est appelé *mot d'extension*. Le format de codage du premier mot d'une instruction est le suivant :

code opération	nRd	ModeAdr	nRs

C'est une machine à deux références (Cf. Chapitre 12) ; nRd est le numéro d'un registre qui constitue le premier opérande. Les deux informations ModeAdr et nRs désignent le deuxième opérande, ModeAdr décrivant le mode d'adressage de cet opérande et nRs le numéro du registre concerné. Le résultat d'une opération binaire est stocké dans le registre de numéro nRd.

Les modes d'adressage considérés sont : registre direct (noté nRs), registre indirect (noté nRs@), registre indirect avec prédécrémentation (noté nRs@-), registre indirect avec postincrémentation (noté nRs@+), immédiat (noté #imm), relatif et absolu. Les modes d'adressage immédiat, relatif et absolu nécessitent un mot d'extension ; pour l'adressage immédiat, il s'agit d'une valeur, pour l'adressage relatif d'un déplacement par rapport à la valeur courante du compteur de programme et pour l'adressage absolu d'une adresse. Pour une description de la sémantique des modes d'adressage, voir le chapitre 12.

Les instructions considérées sont l'addition (`add`), la soustraction (`sub`), le et logique (`and`), le chargement d'une adresse (`lea`), le chargement d'un registre

		def : mode d'adressage								
		MaJ	000	001	010	011	100	101	110	111
	abc	ZNCV	nRs	nRs@-	nRs@	nRs@+	rel		#imm	abs
lea	000			X	X	X	X			X
jmp	001			X	X	X	X			X
jsr	010			X	X	X	X			X
store	011			X	X	X	X			X
add	100	X	X	X	X	X	X		X	X
sub	101	X	X	X	X	X	X		X	X
and	110	X	X	X	X	X	X		X	X
load	111	X	X	X	X	X	X		X	X

FIG. 14.21 – Modes d'adressage autorisés selon les instructions

(`load`), le stockage d'un registre en mémoire (store), le branchement (`jmp`) et le branchement à un sous-programme (`bsr`).

La table 14.21 donne un codage des instructions, le code opération étant codé sur trois bits (nommés a, b, c), et un codage pour le mode d'adressage lui aussi sur trois bits (nommés d, e, f). Une croix dans une case indique que le mode d'adressage est autorisé pour l'instruction. Le tableau indique aussi les instructions pour lesquelles il y a une mise à jour du mot d'état (bits Z, N, C, V). Voici quelques pistes à explorer :

1. Etudier le déroulement de l'instruction add pour chacun des modes d'adressage. Constater les redondances et remarquer que l'interprétation peut être faite en deux phases : tout d'abord le calcul de l'adresse du deuxième opérande (s'il s'agit d'une valeur, on la stockera dans D, et si c'est une adresse, dans A), puis le traitement de l'opération proprement dite.
2. Etudier l'interprétation de l'instruction de branchement à un sous-programme (jsr) ; l'adresse de retour est stockée dans la pile, le registre pointeur de pile étant Dmax. Nous n'avons pas fourni d'instruction de retour ; comment peut-on la programmer avec les instructions données ?
3. Ajouter sur la partie opérative les commandes nécessaires.
4. Décrire la partie contrôle du processeur sous forme d'une machine séquentielle avec actions. Pour chaque microaction utilisée, vérifier qu'elle est effectivement exécutable par la partie opérative fournie ; pour cela, donner en détail l'ensemble des commandes qui lui sont associées.
5. Proposer une réalisation microprogrammée (Cf. Chapitre 10) de l'automate précédent.
6. Décrire les circuits de calcul des commandes d'accès aux bus et de chargement des registres.

Cinquième partie

Architecture d'un système matériel et logiciel simple

Un système matériel et logiciel simple

Dans ce qui précède, nous avons étudié ce qu'est un processeur. Du point de vue matériel il s'agit d'un assemblage de circuits combinatoires et séquentiels dont le rôle est l'interprétation d'un jeu d'instructions particulier. Le jeu d'instructions fourni permet à un utilisateur d'écrire un programme dans un langage de bas niveau : le *langage machine*.

Pour obtenir un ordinateur, il manque encore un certain nombre de composants matériels et logiciels. La partie V décrit le minimum nécessaire pour obtenir un ordinateur simple, mono-utilisateur. L'ordinateur ainsi élaboré ne sera pas un ordinateur réel, notre propos étant plutôt de donner les principes permettant de comprendre le rôle et la place de chaque composant. Nous verrons dans la partie VI comment étendre ce système simple à un ordinateur multitâches, donc éventuellement multi-usagers.

Processeur/mémoire et entrées/sorties

Le chapitre 15 montre comment connecter le processeur et de la mémoire centrale (ou principale). Il s'agit de mémoire à semiconducteurs, à accès aléatoire. Cette mémoire est organisée en plusieurs composants dont certains sont des mémoires mortes programmables (EEPROM) ; d'autres sont nécessairement des mémoires vives. Cette mémoire sert à l'utilisateur : le programme en cours d'exécution et les données associées sont stockés en mémoire vive. Les informations et le code nécessaire au démarrage de l'ordinateur sont stockés en mémoire morte.

Pour réaliser des programmes dont la valeur ne se réduit pas à une constante, l'ensemble processeur/mémoire doit être ouvert vers l'extérieur. Nous verrons dans le chapitre 16 que la communication avec le monde extérieur comporte des aspects de câblage (connexion physique de périphériques d'entrées/sorties) mais qu'il faut surtout gérer des problèmes de synchronisation (Cf. Chapitre 6). Nous reprendrons la description des entrées/sorties dans la partie VI et verrons des améliorations utilisant le mécanisme des interruptions.

Les périphériques d'entrées/sorties sont très divers. On peut distinguer deux

types de fonctionnalités : dialogue avec l'environnement et gestion de mémoire secondaire.

– Certains périphériques permettant un dialogue avec un utilisateur humain (clavier/écran, souris et manettes diverses, scanner, tables traçantes, cartes son, imprimantes, etc.), les circuits de communication avec un environnement industriel (gestion d'automatismes, de conduite de procédés, de robots, capteurs et actionneurs en tous genres) et enfin les circuits servant à établir des communications entre ordinateurs (gestion des réseaux locaux et liaisons grande distance, cartes réseau, modems).

– La mémoire secondaire est une mémoire non volatile, de grande capacité et en général de faible coût (par rapport à la capacité). Les périphériques de gestion de mémoire secondaire sont les disques et disquettes (accès aléatoire), et les systèmes d'archivage et de sauvegarde (bandes, accès séquentiel).

Notion de système d'exploitation

Nous avons à ce stade un squelette d'ordinateur mais celui-ci est inutilisable sans logiciel permettant de lui faire exécuter les travaux auxquels il est destiné. Ce logiciel constitue ce que l'on appelle le *système d'exploitation*. Une partie réside en mémoire centrale et la plus grande partie est stockée en mémoire secondaire. Ce logiciel doit être adapté d'une famille d'ordinateurs à l'autre et entre les générations successives d'ordinateurs d'une même famille. Un système d'exploitation (dont l'essentiel n'est pas écrit en langage d'assemblage d'une famille de processeurs particulière, mais dans un langage de haut niveau) peut fonctionner sur des plate-formes matérielles très diverses. Par exemple, on trouve le système UNIX sur des PC, des stations de travail ou le CRAY. Inversement, il existe souvent plusieurs choix possibles de systèmes d'exploitation pour un même ordinateur ; sur un PC, on peut installer le système UNIX (version LINUX, ou version XINU) ou WINDOWS ou WINDOWS NT.

On trouve 3 sortes de logiciel dans un ordinateur :

– Des bibliothèques chargées de la gestion des principales ressources de l'ordinateur, dont les périphériques, la mémoire et les fichiers. Elles constituent le coeur du système d'exploitation (ce que l'on appelle parfois le logiciel de base). Elles définissent des interfaces standardisées offrant les mêmes fonctionnalités que le matériel physique, mais sous une forme normalisée. Ceci s'applique en particulier à la manipulation des périphériques.

– Diverses bibliothèques qui ne manipulent pas de ressources particulières et que les programmeurs peuvent décider d'utiliser ou non pour développer leurs applications (calcul mathématique, graphique). Elles donnent des fonctionnalités de plus haut niveau sur le matériel existant pour décharger le programmeur d'une partie du travail commun à de nombreuses applica-

tions (par exemple, la plupart des sytèmes fournissent des bibliothèques de gestion des chaînes de caractères, de tri, etc.).

– Les applications qui sont des programmes exécutables. Certaines ne font pas à proprement parler partie du système d'exploitation mais sont livrées presque systématiquement avec. Elles permettent de développer d'autres applications (éditeurs de texte, compilateurs et assembleurs, éditeurs de liens, débogueurs) ou d'observer l'activité du système (comme par exemple regarder les files d'attente des imprimantes).

Les bibliothèques qui constituent le système d'exploitation ont un statut particulier : les applications ont besoin de services similaires et ces bibliothèques interagissent avec la gestion des ressources de l'ordinateur dont nous verrons dans la partie VI qu'elles peuvent être partagées entre plusieurs utilisateurs et plusieurs applications. Par conséquent, on essaie de faire de ces bibliothèques un point de passage obligé, protégé et contrôlé pour accéder aux ressources. De plus, elles sont utilisées par toutes les applications et donc résidentes en mémoire principale (en EEPROM ou chargées lors du démarrage du système).

Les couches supérieures du système d'exploitation sont généralement stockées sur disque, ce qui permet de changer facilement de version ou de système d'une part et d'installer éventuellement plusieurs systèmes d'exploitation différents sur des disques (ou des partitions de disques) distincts et choisir celui que l'on veut lancer au moment du démarrage. Elles sont chargées en mémoire lors de la phase de démarrage.

Notion de pilote de périphérique

Nous ne nous intéressons dans cette partie qu'à la couche basse du système d'exploitation, c'est-à-dire au minimum nécessaire à la gestion des périphériques d'entrées/sorties, à la sauvegarde des programmes (et des données) en mémoire secondaire et à l'exécution des programmes.

En ce qui concerne les périphériques, de nombreux détails peuvent changer d'un ordinateur à l'autre, ou même durant la vie d'une même machine : souris à 2 ou 3 boutons, taille du (des) disque(s), adresses et formats des registres des coupleurs, claviers à la norme française AZERTY ou anglo-saxonne QWERTY, etc.

Lors du démarrage de l'ordinateur et de l'initialisation de la bibliothèque de gestion des périphériques, cette dernière a besoin de connaître les caractéristiques exactes de l'ensemble des périphériques présents qu'elle a à gérer. Les ordinateurs sont généralement équipés d'EEPROM qui permettent de stocker ces informations de manière non volatile, ainsi qu'un petit programme permettant de les consulter et de les mettre à jour : le gestionnaire de configuration. Le système d'exploitation stocké sur le disque peut contenir ses propres pilotes de périphériques. Mais il doit en exister au moins une version rudimen-

taire en EEPROM de mémoire centrale pour les périphériques indispensables lors du démarrage, typiquement clavier écran, disque, disquette ou CD.

Les périphériques étant compliqués et offrant plusieurs fonctionnalités, certaines partagées entre plusieurs périphériques, leur gestion est regroupée au sein d'un partie du système qui s'appelle le pilote de périphérique. Ce sera l'objet du chapitre 17.

Système de gestion de fichiers et interface de commande

Une autre partie du système utilisant les primitives du pilote offre des services de plus haut niveau comme la gestion d'informations structurées : le système de gestion des fichiers (chapitre 19).

Par ailleurs, un programme en langage d'assemblage ne peut s'exécuter qu'après une phase de traduction en langage machine et éventuellement de liaison avec d'autres programmes ou avec des bibliothèques. Nous décrivons dans le chapitre 18 les différentes étapes de la vie d'un programme.

La couche interface entre l'utilisateur et tous ces composants est l'interprète du langage de commande. C'est l'objet du chapitre 20. Nous y décrivons en particulier le chargement d'un programme en mémoire centrale et son lancement. Ensuite, c'est le processeur qui interprète les instructions du programme comme nous l'avons décrit dans le chapitre 14.

Chapitre 15

Relations entre un processeur et de la mémoire

Dans une version minimale, un ordinateur est composé d'un *processeur* (Cf. Chapitre 14) et d'une *mémoire* (Cf. Chapitre 9). Le processeur produit des informations à stocker dans la mémoire ou bien récupère des informations précédemment rangées en mémoire.

Nous avons vu au chapitre 14 qu'un processeur peut être considéré comme une machine algorithmique, assemblage d'une partie opérative et d'une partie contrôle.

Nous avons expliqué au chapitre 9 comment se déroule un accès mémoire et précisé au chapitre 11 les aspects de synchronisation lors de la connexion d'une machine algorithmique à une mémoire. Au chapitre 14 nous nous sommes placés dans l'hypothèse simplificatrice d'une mémoire rapide.

Ce chapitre apporte des informations complémentaires concernant la réalisation de la connexion entre le processeur et la mémoire. Par ailleurs, nous présentons les différents aspects du décodage d'adresse permettant de gérer le fait que la mémoire soit organisée en différents morceaux, et que l'on puisse la plupart du temps accéder à des sous-ensembles du mot mémoire.

> *Le paragraphe 1. est consacré à différents aspects concernant la connexion du processeur et de la mémoire. Le paragraphe 2. montre les conséquences de l'organisation de la mémoire en plusieurs unités physiques, ou boîtiers. Enfin dans le paragraphe 3. nous montrons comment gérer des accès à des données logiques de tailles différentes (Cf. Chapitre 4) alors que les accès se font à des données physiques de tailles identiques.*

1. Le bus mémoire

Nous avons vu au chapitre 9 que le *bus mémoire* est constitué du bus de données et du bus d'adresse. Le bus de données est un ensemble de fils D_{n-1} à D_0 via lesquels transitent les valeurs échangées par le processeur et

la mémoire. Ce bus est bidirectionnel (transfert dans les deux sens). Le bus d'adresse (unidirectionnel) est un ensemble de fils A_{m-1} à A_0 en sortie du processeur via lesquels ce dernier indique à la mémoire le numéro (l'adresse) du mot auquel il accède.

Dans l'hypothèse où le temps de cycle de la mémoire est inférieur ou égal au cycle d'horloge du processeur, les seuls signaux AccèsMem et $\mathsf{l}/\overline{\mathsf{e}}$ suffisent pour gérer le protocole de communication. Nous nous limitons à cette situation dans ce chapitre pour nous concentrer sur les aspects connexion, organisation en différents boîtiers et accès à des sous-ensembles du mot mémoire.

Notons toutefois que cette hypothèse est assez rarement vérifiée. Le temps d'accès peut dépendre de la zone mémoire à laquelle on accède (ROM, RAM ou entrées/sorties). Les processeurs gèrent donc un protocole complet (inspiré du *protocole poignée de mains* décrit dans le chapitre 6) et la durée d'un accès peut être étendue d'une ou plusieurs périodes d'horloge via un signal d'acquittement piloté par la mémoire. Le nom du signal varie avec les familles de processeurs (Data Transfert Ack pour le 68000, Memory Hold pour le SPARC, Wait ou ready pour d'autres processeurs 8 ou 16 bits ...).

1.1 Connexion simple

Nous avons dit au chapitre 9 que la connexion des fils était simple mais en réalité il ne suffit pas de relier directement les fils de même nature.

Les adresses et les données représentent un grand nombre de sorties du processeur (64 pour un processeur à 32 bits). La puissance dissipée totale et le courant qu'est capable de débiter un circuit intégré sont limités. Cela limite le courant pour chaque fil de sortie. Chaque sortie ne peut donc piloter qu'un nombre réduit d'entrées, alors qu'elle peut être connectée à de nombreux boîtiers de mémoire, d'où la nécessité d'une amplification externe.

L'amplification des signaux unidirectionnels tels que les adresses ne pose pas de problème particulier : la sortie des amplificateurs externes peut rester active en permanence.

Le bus de données, bidirectionnel, implique au contraire le recours à des amplificateurs à sorties 3 états montés tête-bêche. Lors d'une lecture, la commande 3 états dans le sens mémoire/processeur doit être activée et celle dans le sens processeur/mémoire doit être au contraire inhibée ; et réciproquement lors d'une écriture. Il suffit donc de commander la validation des sorties processeur et mémoire respectivement par le signal $\mathsf{l}/\overline{\mathsf{e}}$ et son complément.

La figure 15.1 illustre ces connexions ; seul 1 fil de la nappe des fils du bus données (respectivement bus adresses) est représenté, à savoir Di (respectivement Aj).

Sur la figure, on voit un signal supplémentaire DeconnexionProcesseur dont nous parlons au paragraphe 1.3.

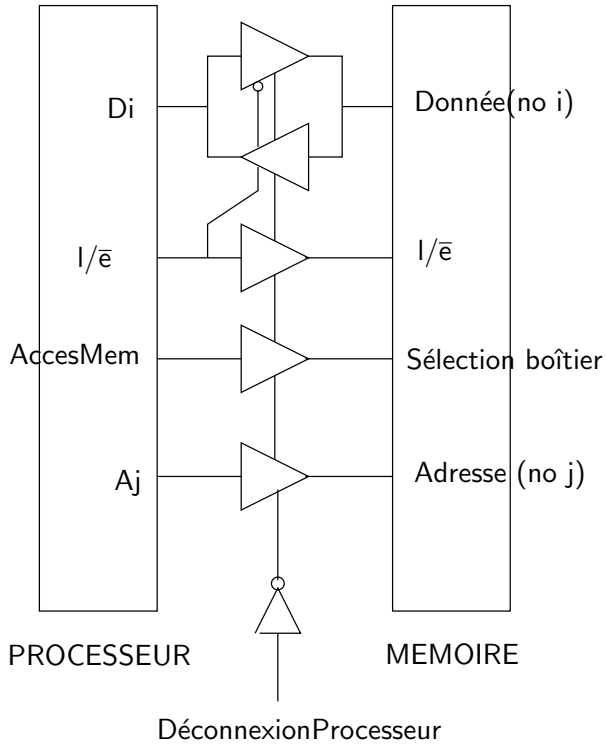


FIG. 15.1 – Connexion processeur/mémoire. Le signal DéconnexionProcesseur est expliqué au paragraphe 1.3

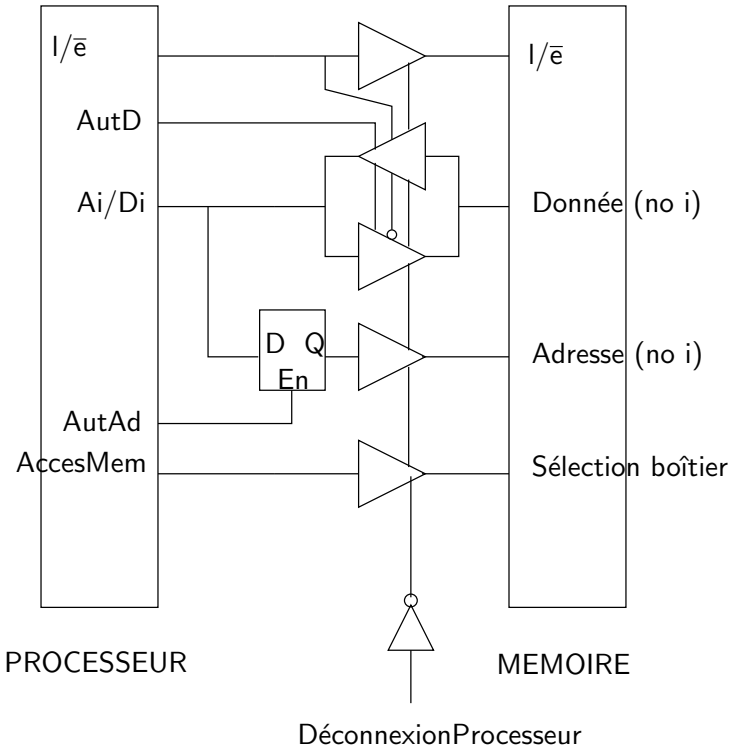


FIG. 15.2 – Multiplexage des bus adresses et données

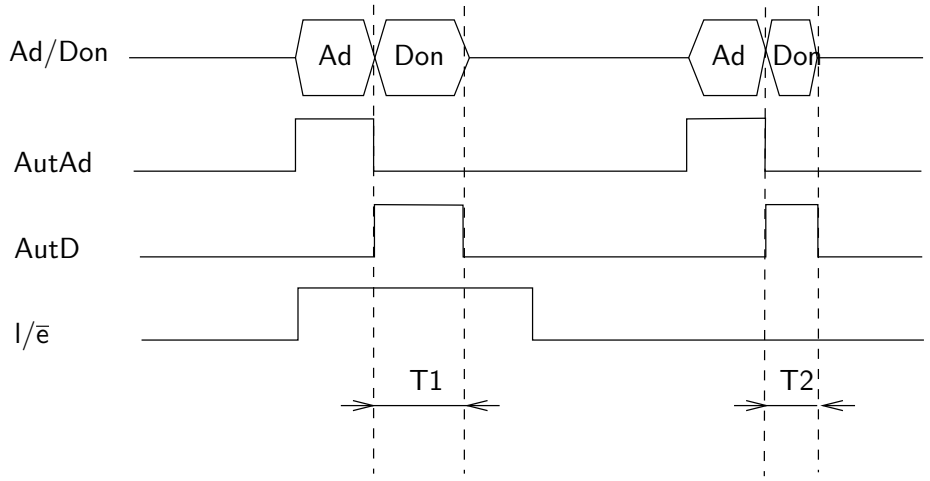


FIG. 15.3 – Chronogrammes décrivant l'accès à la mémoire dans le cas d'un bus multiplexé : l'intervalle T1 correspond à une lecture mémoire et l'intervalle T2 à une écriture mémoire.

1.2 Multiplexage du bus mémoire

Pour des nécessités d'amplification des signaux, et pour économiser le nombre de broches sur les boîtiers, les processeurs sont parfois dotés d'un bus mémoire multiplexé. Le principe consiste à faire transiter successivement les adresses puis les données via les mêmes broches.

On trouve généralement deux sorties supplémentaires servant à l'*échantillonnage des données* (AutD) et à l'*échantillonnage des adresses* (AutAd) (*data strobe* et *address strobe* en anglais) signalant les instants auxquels les adresses et les données transitent sur le bus.

Sur la figure 15.2 qui montre les connexions, on note la présence d'un *verrou* avant la commande 3 états connectée à la mémoire.

Un cycle de lecture se déroule en deux phases. Tout d'abord, le processeur émet l'adresse, accompagnée de son signal de validation AutAd. Cette adresse est mémorisée dans le verrou commandé par le signal AutAd. Durant cette phase du cycle, la sortie de données de l'amplificateur mémoire vers processeur doit être désactivée. Puis dans une deuxième phase, la lecture se déroule comme dans le cas d'un bus non multiplexé, à ceci près que le transfert de la donnée est conditionné par le signal d'échantillonnage des données AutD.

Un cycle d'écriture se déroule de façon similaire à un cycle de lecture.

La commande des amplificateurs externes associés au bus données est modifiée en conséquence : une solution simple consiste à utiliser le signal AutD comme condition supplémentaire de validation de ces amplificateurs. L'amplificateur dans le sens mémoire vers processeur sera ainsi activé par la condition I/$\bar{e}$ ET AutD. La figure 15.3 décrit l'évolution des différents signaux.

1.3 Déconnexion du bus mémoire

Dans un ordinateur, le processeur n'est pas le seul composant qui réalise des accès à la mémoire.

Nous verrons au chapitre 16 consacré aux circuits d'entrées/sorties que certains d'entre eux ont vocation à accéder à la mémoire sans passer par le processeur (optimisation appelée *accès direct à la mémoire*).

Plus généralement, dans le cas où plusieurs processeurs se partagent la même mémoire (par exemple un processeur général et un processeur graphique), le bus mémoire doit être partagé, d'où la nécessité d'un *arbitrage de bus*. Nous ne détaillons pas dans ce livre la façon de réaliser des arbitres de bus. Nous montrons seulement comment déconnecter le processeur du bus mémoire ce qui est un préalable à toute possibilité de partage de ce bus.

Pour les signaux unidirectionnels d'adresses, $\mathsf{I}/\overline{\mathsf{e}}$ et AccèsMem, il suffit d'ajouter une commande aux amplificateurs 3 états : un signal de déconnexion du processeur : DéconnexionProcesseur sur les figures 15.1 et 15.2.

En ce qui concerne les signaux bidirectionnels de données, les amplificateurs 3 états sont déjà présents, il suffit de rajouter le signal DéconnexionProcesseur comme condition supplémentaire d'activation des sorties.

2. Utilisation de plusieurs circuits de mémoire

Il arrive que la capacité mémoire d'un boîtier soit inférieure à la capacité souhaitée. En général, on veut même que la quantité de mémoire associée à un processeur soit modifiable ; il est courant de rajouter des boîtiers de RAM dans un ordinateur. Par ailleurs un ordinateur est doté de boîtiers de mémoire vive et de mémoire morte. Aussi pour fabriquer une mémoire on utilise plusieurs boîtiers. Dans un premier temps, considérons que les boîtiers sont tous de la même taille.

Par ailleurs les processeurs sont conçus avec un bus adresses d'une certaine taille α qui lui permet potentiellement d'adresser 2^α mots différents. Nous allons fabriquer une mémoire de taille β (avec un certain nombre de boîtiers) et $\beta \leq 2^\alpha$.

Ce paragraphe montre comment alors associer une adresse à chaque mot physique de la mémoire, problème appelé *décodage d'adresses*.

Nous considérons tout d'abord le cas où $\beta = 2^\alpha$ puis nous étudions le cas $\beta < 2^\alpha$ et enfin nous parlons d'extension mémoire.

2.1 Décodage externe et sélection des mémoires

Le problème peut se poser dans les termes suivants : comment former une mémoire de 2^m mots avec 2^k boîtiers mémoires (numérotés de 0 à $2^k - 1$), de 2^{m-k} mots chacune ? Chacun de ces boîtiers a ses $m - k$ bits d'adresse et son propre signal de sélection.

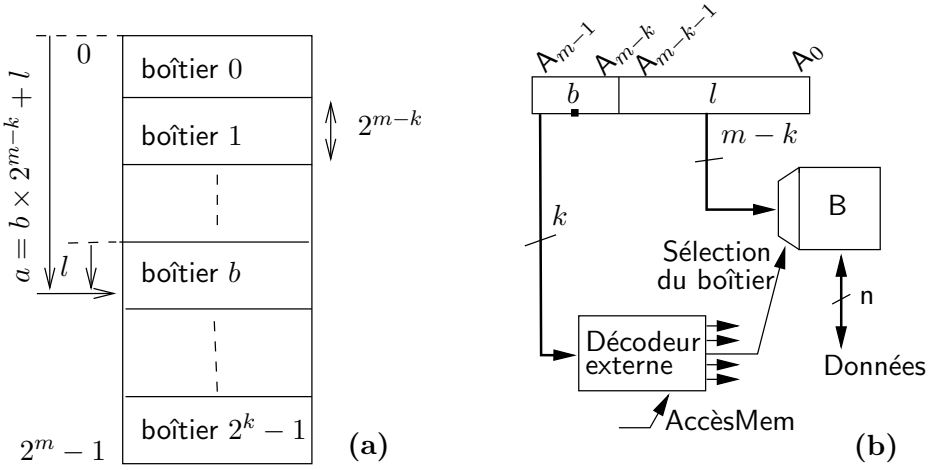


FIG. 15.4 – (a) Mémoire de 2^m mots organisée avec 2^k boîtiers de 2^{m-k} mots chacun. (b) Décodage d'adresse en cascade. B est l'un des 2^k boîtiers de taille 2^{m-k}.

Soit a une adresse de mot sur m bits. Soient respectivement b et l (codés sur k et $m-k$ bits) le quotient et le reste de la division de a par 2^{m-k} : $a = b \times 2^{m-k} + l$ (le mot adresse en haut de la figure 15.4-b).

Le principe consiste à stocker le mot d'adresse a dans le mot numéro l du boîtier ou de la barette de numéro b (Cf. Figure 15.4-a).

Le raccordement est le suivant : le bus de données, le signal $\mathsf{l/\overline{e}}$ et les signaux d'adresses A_{m-k-1} à A_0 sont reliés aux signaux correspondants des 2^k circuits. Les signaux d'adresses A_{m-1} à A_{m-k} sont reliés aux entrées de sélection d'un décodeur externe commandé par le signal AccèsMem et dont chaque sortie pilote le signal de *sélection de boîtier* d'un circuit mémoire. Le décodeur d'adresse sur m bits, initialement interne aux boîtiers, est ici remplacé par la mise en cascade du décodeur externe sur k bits et (dans chaque circuit mémoire) d'un décodeur interne sur $m-k$ bits (Cf. Figure 15.4-b).

Cette organisation regroupe dans le même boîtier les mots d'adresses consécutives de poids forts identiques. Lorsque l'on ne veut utiliser que x boîtiers ($x < 2^k$), la solution que nous venons de décrire présente l'avantage de permettre la création d'une mémoire contiguë de $x \times 2^{m-k}$ mots.

Il existe des variantes de décodage dans lesquelles la sélection des boîtiers exploite d'autres bits d'adresses que ceux de poids forts. A partir de la décomposition $a = l' \times 2^k + b'$, par exemple, nous pourrions connecter les l' signaux de sélection de mots des boîtiers aux signaux d'adresses A_{m-1} à A_k du bus d'adresses et les b' signaux de poids faible A_{k-1} à A_0 au décodeur externe pour sélectionner des boîtiers. Contrairement à la précédente, cette dernière organisation disperse les mots d'adresses consécutives dans des boîtiers différents. Elle ne supporte pas l'absence d'un boîtier, les mots manquants créant des trous disséminés dans toute la zone mémoire. Elle est à la base de certaines techniques d'optimisation du débit de la mémoire (bancs de mémoire).

2.2 Décodage partiel et synonymie d'adresses

Il se peut que le nombre m de signaux d'adresses du processeur excède largement le nombre p de bits nécessaires mis à sa disposition pour adresser la mémoire physique.

Une adresse de mot émise par le processeur se décompose maintenant de la façon suivante : $a = e \times 2^{m-p} + b \times 2^{p-k} + l$. En partant des poids faibles : l occupe les bits 0 à $k-1$, b occupe les bits k à $p-1$, e occupe les bits p à $m-1$.

Le concepteur doit définir quelle plage d'adresses du processeur parmi les 2^p possibles il attribue à la mémoire (généralement de 0 à 2^{p-1}, soit $e = 0$).

Une première possibilité est d'introduire un nouvel étage de décodage définissant AccèsMem. Dans le cas où $e = 0$, AccèsMem $= \overline{A_{m-1}}, ...\overline{A_p}$ et erreur $= \overline{\text{AccèsMem}}$. Ainsi, AccèsMem traverse un nouveau décodeur avant d'atteindre la commande du décodeur externe du schéma précédent, ce nouveau décodeur étant commandé par les bits d'adresses de poids forts restés inutilisés. On parle alors de décodage complet.

Lors d'un cycle d'accès en lecture en dehors de la plage d'adresses dévolue à la mémoire, aucune sortie n'imposera de niveau électrique sur les signaux de données du bus. Il en résulte que les bits peuvent prendre une valeur quelconque (qui peut dépendre entre autres de la technologie des amplificateurs de bus utilisés). Un tel accès constitue une erreur de programmation. Le programmeur *ne peut faire aucune hypothèse sur le résultat d'une telle lecture.* En pratique, on obtiendrait le plus souvent un mot dont tous les bits sont à 1.

Lorsqu'il détecte un cycle d'accès en dehors de la plage d'adresses de la mémoire, le circuit de décodage d'adresses peut se contenter de n'activer aucun boîtier. Il est toutefois préférable de renvoyer un signal d'erreur au processeur qui provoquera un déroutement de type erreur de bus (Cf. Chapitre 22), signal qui sera répercuté au niveau de l'utilisateur (le classique *bus error*).

Une autre possibilité consiste à conserver le schéma précédent sans tenir compte des $m - p$ signaux d'adresses de poids forts : on parle de décodage partiel. Tout mot de la mémoire physique possède dans ce cas 2^{m-p} adresses équivalentes ou synonymes : son adresse officielle a et toutes les autres adresses de la forme $(a \pm i \times 2^{m-p})$ modulo 2^m avec i entier.

2.3 Supports de boîtiers multitaille et extension mémoire

Les fabricants d'ordinateurs prévoient généralement un certain nombre de connecteurs dans lesquels on peut enficher des boîtiers de mémoires pour en augmenter la capacité (extension de mémoire). La fiche technique précise alors la quantité de mémoire déjà installée dans l'appareil et la taille maximale qu'il est possible d'atteindre en équipant tous les connecteurs de boîtiers.

Il est intéressant de prévoir des connecteurs pouvant accepter aussi bien les

boîtiers disponibles sur le marché au moment de la conception que les futurs boîtiers de capacité double ou quadruple (et plus) que l'utilisateur pourra se procurer ultérieurement.

Soit 2^k le nombre de connecteurs. Soit 2^p la taille de la plus petite barette supportée et 2^g celle de la plus grande. La technique consiste à décoder les signaux A_p à A_{p+k-1} pour générer les signaux de sélection de boîtiers. Les autres signaux A_0 à A_{p-1} et A_{p+k} à A_{g-1} sont disponibles sur les connecteurs et reliés aux entrées de sélection de mots des boîtiers.

2.4 Spécialisation de zones mémoires

Le décodage prend en compte d'autres éléments que la seule adresse.

Certains jeux d'instructions distinguent plusieurs espaces d'adressage. Au moins un des espaces est toujours destiné aux accès mémoire ordinaires via les instructions normales (`load/store` ou `move`) d'accès à la mémoire. Le ou les autres espaces sont destinés à des usages spécifiques et nécessitent l'usage d'instructions spéciales pour y accéder.

Des sorties additionnelles du processeur indiquent le numéro de l'espace utilisé. Du point de vue du décodage, on peut considérer ce numéro comme des bits de poids fort de l'adresse à décoder.

A titre d'exemple, les processeurs de la famille INTEL distinguent un espace mémoire ordinaire (instruction `move`) et un espace dédié aux seules entrées/sorties (instructions `in` et `out`, Cf. Chapitre 12, paragraphe 1.4.5). Une sortie M/IO du processeur indique à quel espace le cycle d'accès s'adresse.

De plus, le processeur délivre vers l'extérieur des signaux donnant des informations sur son état interne ou sur la nature de l'instruction en cours d'exécution. Le décodeur doit émettre un signal d'erreur lorsqu'il détecte un accès à la mémoire non conforme aux informations fournies par le processeur. Ce signal se traduit par l'envoi au processeur d'une requête d'interruption (Cf. Chapitre 22).

Par exemple, il est possible au concepteur de la carte de protéger en écriture certaines zones de mémoire. Lors d'une demande d'accès en écriture à une telle zone, le décodeur d'adresse détecte que l'adresse ne fait pas partie des plages mémoires autorisées en écriture et émet donc un signal d'erreur.

Pour obtenir par exemple l'expression du signal de sélection d'une mémoire morte, il suffit de prendre le signal que l'on aurait utilisé pour une mémoire vive et d'en faire le produit avec le signal $\mathsf{l}/\overline{\mathsf{e}}$. Le concepteur de la carte peut installer un décodeur qui détecte les accès en écriture en mémoire morte et génère une erreur.

Par ailleurs, nous verrons dans le chapitre 24 qu'il faut implanter des mécanismes de protection lorsque différentes entités utilisent le processeur et la mémoire. Certaines parties de mémoire sont ainsi réservées à certains types d'utilisateurs et l'accès par d'autres provoque une erreur detectée au niveau du décodeur d'adresses. On distingue la notion d'accessibilité en *mode super-*

viseur ou en *mode utilisateur* ; lors de tout accès à la mémoire le processeur spécifie le mode d'accès courant : ce mode est une entrée supplémentaire pour le décodeur d'adresses. Dans ce cas aussi l'accès erroné à une zone reservée à un certain mode doit être détecté par le décodeur qui émet alors un signal d'erreur. L'exercice E15.5 illustre la prise en compte d'un tel cas.

3. Accès à des données de tailles différentes

L'unité adressable de la majorité des processeurs est l'octet : les adresses utilisées pour l'accès à la mémoire sont des adresses d'octet. Mais les processeurs sont également capables d'accéder à des multiples de l'octet allant jusqu'à la taille du bus données du processeur.

Par exemple, dans la famille des processeurs 68XXX, l'instruction en langage d'assemblage comporte la taille de la donnée manipulée ; ainsi, les instructions `move.b D1,D2`, `move.w D1,D2` et `move.l D1,D2` signifient respectivement le transfert d'un octet, d'un mot de 16 bits ou d'un mot long de 32 bits du registre `D1` vers le registre `D2`. Dans le SPARC les données sont sur 32 bits sauf pour certaines instructions ; par exemple, l'instruction `ld` permet le chargement dans un registre d'un mot mémoire 32 bits, mais on peut aussi lire un octet (respectivement un demi-mot de 16 bits), signé ou non, avec une des instructions : `ldsb` ou `ldub` (respectivement `ldsh`, `lduh`).

Considérons par exemple un processeur 32 bits capable d'accéder à des octets, des demi-mots de 16 bits et des mots de 32 bits. Puisque les adresses sont des adresses d'octet, la logique voudrait que le processeur soit doté d'une mémoire d'un octet de large. Pour transférer un mot de 32 bits d'adresse A, il suffirait d'enchaîner quatre accès mémoire aux adresses consécutives A, $A+1$, $A+2$ et $A+3$. Cette solution présente l'inconvénient d'être lente. Le processeur est donc doté de quatre mémoires juxtaposées, fournissant chacune un octet du mot de 32 bits, que le processeur peut lire en un seul cycle d'accès mémoire. En revanche, lorsque le processeur effectue un accès à un octet, il suffit de ne sélectionner qu'une seule de ces mémoires. Cette organisation pose des problèmes :

1. d'alignement et de *décodage d'adresse* pour sélectionner les quatre mémoires d'octet.
2. de cadrage des données sur le bus de données et dans les registres. On pourrait vouloir cadrer la donnée de différentes façons (vers les poids forts, vers les poids faibles, au milieu...) ; il faudrait alors plusieurs instructions de chargement ou de lecture d'un registre. En fait, un seul type de cadrage suffit ; en effet, on peut toujours réaliser les autres avec des instructions de décalages, plus générales. Le principe retenu est de cadrer les données de taille inférieure à 32 bits en poids faible des registres et de recopier le bit de signe (ou des 0 si on veut interpréter la donnée comme non signée) dans les bits de poids fort des registres.

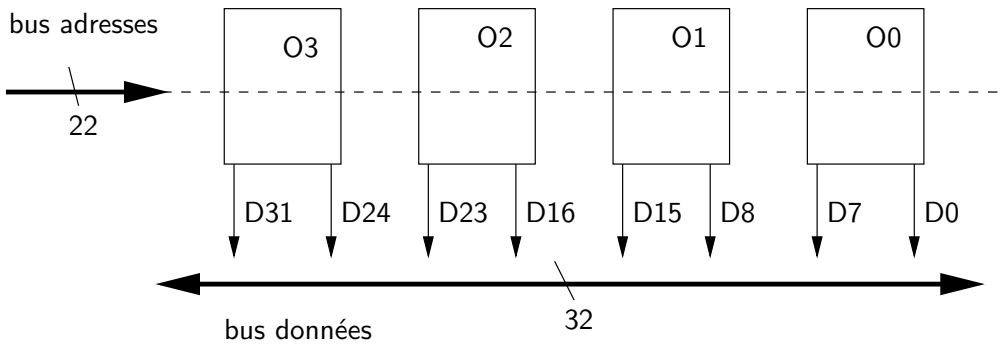


FIG. 15.5 – Mémoire organisée avec 4 boîtiers de 1 Mo

Si l'on veut accéder à des données de taille supérieure à celle du bus données, il faut réaliser plusieurs accès mémoire, le programmeur organisant lui-même le stockage de ces données en mémoire.

3.1 Etude de cas : décodage d'adresses

Dans ce paragrapge, nous présentons une étude de cas illustrant le décodage d'adresses pour un processeur ayant un bus de données sur 32 bits et un bus d'adresses sur 22 bits.

3.1.1 Organisation matérielle

Nous disposons de 4 boîtiers mémoire de 1Mo (1 Mega-octets), c'est-à-dire de 2^{20} octets. En effet, $2^{20} = 2^{10} * 2^{10} = 1024 * 1024 \approx 10^6$.

Pour pouvoir accéder à un mot mémoire de 32 bits en 1 cycle, on prend chacun des 4 octets dans un des boîtiers. La figure 15.5 donne une première idée de cette organisation.

Puisqu'il ne peut y avoir qu'une adresse à la fois sur le bus adresses, les quatre octets auxquels le processeur accède sont à la même adresse, chacun dans son boîtier. En conséquence l'adresse d'un mot de 32 bits doit être un multiple de 4. Supposons, en effet, que l'on veuille accéder à 4 octets consécutifs à partir d'un octet du boîtier O2 ; il faudrait prendre les 3 premiers octets respectivement dans O2, O1 et O0 et le quatrième dans O3, mais à l'adresse suivante d'où une adresse différente pour ce dernier octet ...

Nous utilisons les 2 bits de poids faibles de l'adresse émise par le processeur pour distinguer les boîtiers.

> **Remarque :** L'association entre les mémoires et les octets du bus dépend de la convention utilisée : gros ou petit boutiste (Cf. Chapitre 4). Nous supposons ici que la convention est gros-boutiste.

L'adresse d'un octet de O3 est de la forme $4 * X$ ($A1A0 = 00$), l'adresse d'un octet de O2 de la forme $4 * X + 1$ ($A1A0 = 01$), l'adresse d'un octet de

O1 de la forme $4*X+2$ ($A1A0 = 10$) et l'adresse d'un octet de O0 de la forme $4*X+3$ ($A1A0 = 11$). Les autres bits de l'adresse (A21, ..., A2) désignent une adresse dans un boîtier.

Le bus de données se décompose à présent en quatre octets. Les mémoires d'octet sont connectées chacune à un octet du bus de données.

Ainsi, étant donnée une adresse $4k+i$ émise par le processeur, la valeur k codée sur les vingt bits de poids fort reliés aux décodeurs internes des mémoires représente un numéro d'octet dans son boîtier et la valeur i codée sur les deux bits de poids faible est un numéro de boîtier où trouver l'octet d'adresse $4k+i$.

En émettant k en poids fort du bus d'adresse, le processeur peut accéder simultanément et en un seul cycle mémoire à : l'octet d'adresse $4k$ via D_{24} à D_{31} du bus de données, l'octet d'adresse $4k+1$ via D_{16} à D_{23}, l'octet d'adresse $4k+2$ via D_8 à D_{15} et l'octet d'adresse $4k+3$ via D_0 à D_7.

Par construction, les octets d'adresses $4k+x$ et $4(k\pm1)+y$, avec ($0 \leq x \leq 3$ et $0 \leq y \leq 3$), ne sont pas accessibles dans un même cycle mémoire.

La combinaison $4k+2$ et $4k+3$ correspond au transfert d'un demi-mot de seize bits d'adresse $4k+2$. En revanche, un demi-mot de seize bits d'adresse $4k+3$, composé des octets d'adresses $4k+3$ et $4(k+1)+0$ n'est pas accessible en un seul cycle mémoire.

Nous retrouvons là l'origine des règles d'alignement exposées au chapitre 4 imposant des adresses de demi-mot de seize bits paires, des adresses de mots de 32 bits multiples de 4 et ainsi de suite (notons cependant que le matériel serait capable de transférer en un cycle des demi-mots d'adresses $4k+1$).

> **Remarque :** Les versions 32 bits de certaine familles de processeurs (INTEL et MOTOROLA) héritent de logiciels développés pour leurs prédécesseurs travaillant sur 8 ou 16 bits et pour lesquels les contraintes d'alignement ne se posaient pas. C'est pourquoi ils tolèrent les adresses non alignées au prix de deux accès mémoire par transfert d'objet non aligné et d'une complexité matérielle accrue. Sur les processeurs modernes, l'accès à des données à des adresses non alignées déclenche une erreur.

3.1.2 Le décodeur d'adresses

Lors d'un accès mémoire, le processeur établit les signaux suivants :

- l'adresse A21, ..., A0.
- la taille de la donnée ; cette information provient du décodage de l'instruction en cours d'exécution. Le chargement ou le rangement se fait sur 1 octet, 1 demi-mot de 16 bits ou 1 mot de 32 bits ; ces 3 cas peuvent être codés sur 2 bits.
- les signaux AccèsMem et $\mathsf{l}/\overline{\mathsf{e}}$.

Le décodeur d'adresses doit produire les signaux de sélection de chacun des boîtiers : SelO3, SelO2, SelO1, SelO0.

Nous avons vu au paragraphe précédent que le boîtier de numéro i, $i \in \{0,1,2,3\}$ contient le mot d'adresse $4k+i$. De plus le décodeur d'adresses

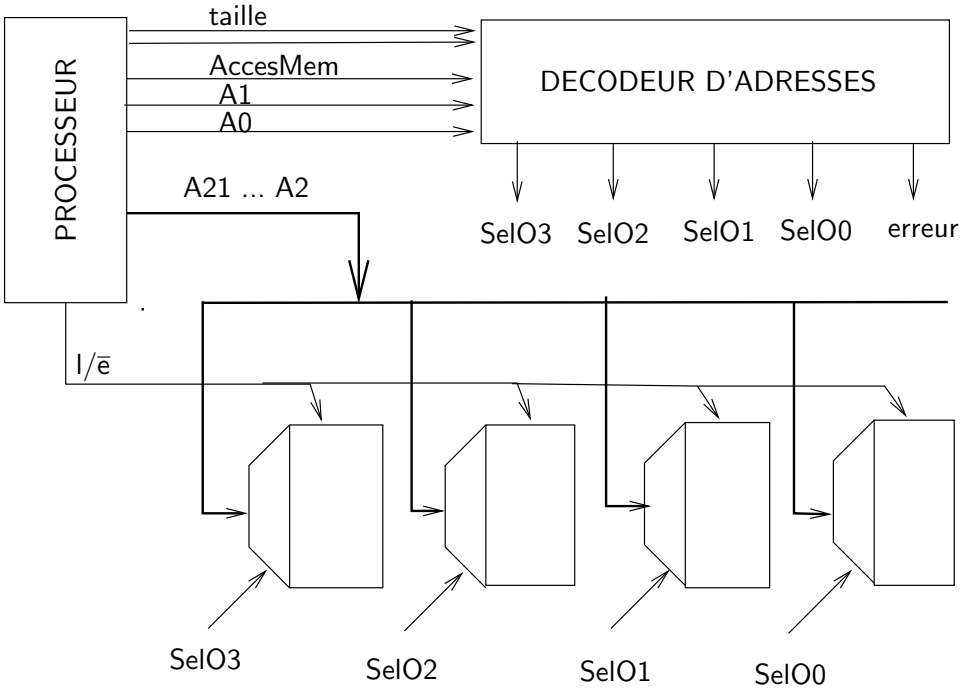


FIG. 15.6 – Décodage d'adresse dans le cas d'accès à des sous-ensembles du mot mémoire

interne de chaque boîtier reçoit l'adresse sur 20 bits de l'octet auquel le processeur accède.

La figure 15.6 décrit cette organisation. Le tableau 15.7 donne la table de vérité de la fonction de décodage d'adresses. Notons la présence d'un signal erreur émis par le décodeur ; ce signal correspond à une demande d'accès à une adresse invalide, il pourra être associé à une interruption (Cf. Chapitre 22).

Les exercices E15.4 et E15.5 poursuivent cette étude de cas.

Nous avons dit que la taille de la mémoire peut varier dans la vie d'un ordinateur, les constructeurs prévoyant en effet la possibilité de rajouter des boîtiers mémoire. Le décodeur d'adresses doit avoir été prévu pour ce faire et c'est lors de l'initialisation de l'ordinateur qu'un programme détecte la quantité de mémoire réellement présente.

3.2 Etude de cas : gestion du bus données

Le problème qu'il reste à résoudre est le cadrage des données de taille inférieure à la taille du bus données sur celui-ci. Nous traitons un exemple simplifié de façon à ne pas écrire des tables de vérité trop complexes.

Considérons un processeur ayant un bus adresses sur m bits (adresse = $A_{m-1}, ..., A_0$), un bus données sur 16 bits et une mémoire formée de deux boîtiers de 2^{m-1} octets (Cf. Figure 15.8).

Soit X l'entier représenté en binaire par les bits $m-1$ à 1 du bus adresses. La

A1 A0	taille	SelO3	SelO2	SelO1	SelO0	erreur
0 0	octet	1	0	0	0	0
0 1	octet	0	1	0	0	0
1 0	octet	0	0	1	0	0
1 1	octet	0	0	0	1	0
0 0	16 bits	1	1	0	0	0
0 1	16 bits	0	0	0	0	1
1 0	16 bits	0	0	1	1	0
1 1	16 bits	0	0	0	0	1
0 0	32 bits	1	1	1	1	0
0 1	32 bits	0	0	0	0	1
1 0	32 bits	0	0	0	0	1
1 1	32 bits	0	0	0	0	1

FIG. 15.7 – Fonction de décodage d'adresses

mémoire notée $2X$ stocke les octets d'adresses paires (adresse $= A_{m-1}, ..., A_1, 0$) et la mémoire notée $2X + 1$ stocke les octets d'adresses impaires (adresse $= A_{m-1}, ..., A_1, 1$).

La mémoire "$2X$" est connectée à l'octet de poids fort du bus données ($D_{15}, ..., D_8$) et la mémoire "$2X + 1$" est connectée à l'octet de poids faible du bus données ($D_7, ..., D_0$).

Le processeur indique au dispositif de décodage d'adresses la taille de l'information à laquelle il accède (octet ou mot de 16 bits) et le bit de poids faible d'adresse (A_0) indique s'il s'agit d'une adresse paire ou impaire.

Le décodeur d'adresses produit les signaux de sélection des boîtiers mémoire : Sel2X et Sel2X+1.

Lorsqu'un accès à un mot de 16 bits avec une adresse paire est demandé, il n'y a aucun problème : un octet de chacun des boîtiers étant envoyé (ou récupéré) sur le bus données.

Lorsque le processeur veut écrire un octet en mémoire le problème est simple. Le programmeur sait à quelle adresse il écrit et est responsable de l'organisation de ses données en mémoire. Il lui suffit donc de préciser la taille de la donnée à écrire. En général des instructions sont prévues pour cela dans les processeurs ; par exemple, dans le SPARC (Cf. Chapitre 12, figure 12.3), l'écriture de 32 bits dans la mémoire est réalisée par l'instruction `ST` et l'écriture d'un octet par une instruction différente (`STB`, store byte).

En revanche, pour transférer un octet de mémoire vers un registre R (lecture d'un octet en mémoire), il faut savoir quelle partie du registre est affectée et que vaut le reste du registre. Le problème est ainsi de sélectionner la bonne partie du bus données.

Pour traiter ce problème, un circuit C est ajouté au processeur : la figure 15.9 montre sa position et la table 15.10 décrit la fonction qu'il réalise.

Les octets étant cadrés dans les poids faibles du registre R, si un octet de

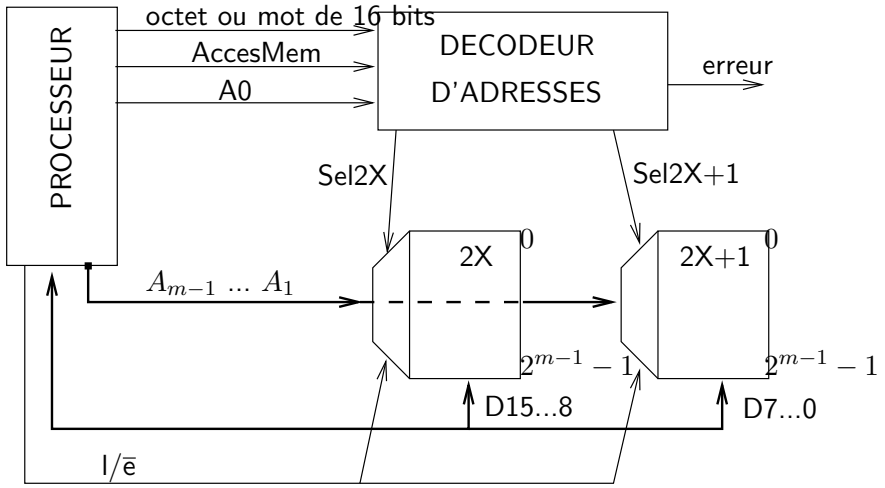


FIG. 15.8 – Mémoire accessible par octets ou mots de 16 bits

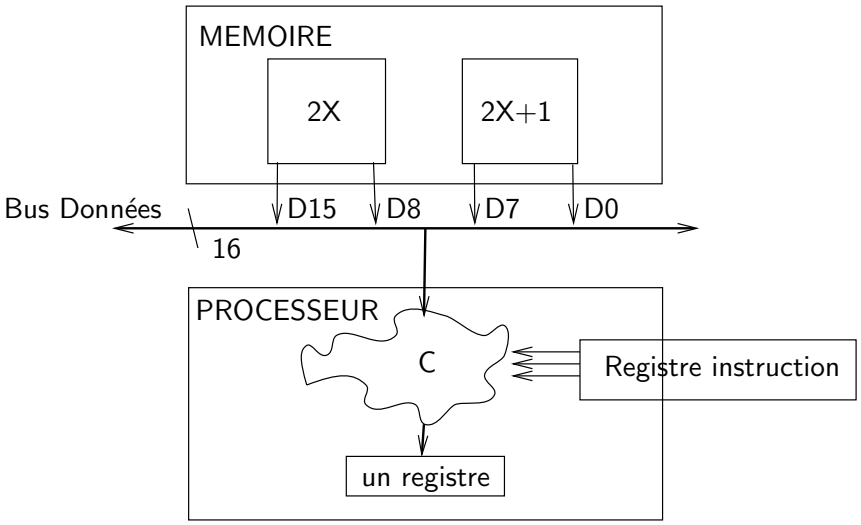


FIG. 15.9 – Recadrage des données lors de lecture de sous-multiples du mot mémoire

Type Accès	Adresse	Type Instruction	R15 ... R8	R7 ... R0
16	paire	-	D15 ... D8	D7 ... D0
16	impaire	-	———	———
8	paire	non signé	0 ... 0	D15 ... D8
8	impaire	non signé	0 ... 0	D7 ... D0
8	paire	signé	D15 ... D15	D15 ... D8
8	impaire	signé	D7 ... D7	D7 ... D0

FIG. 15.10 – Cadrage des données dans un registre lors d'une opération de lecture mémoire

la mémoire "$2X + 1$" est transféré vers le registre R, il faut étendre la valeur représentée en remplissant les 8 bits de poids fort du registre R avec des 0 ou le bit de signe de la valeur, selon qu'il s'agit d'un chargement non signé ou signé (Cf. Chapitre 3). S'il s'agit d'un octet de la mémoire "$2X$", il doit être placé dans les poids faibles de R, les poids forts étant traités comme précédemment.

Le circuit C reçoit en entrée la taille de la donnée (octet ou mot de 16 bits), la parité de l'adresse (bit A_0) et le type d'instruction (signée ou non). Par exemple, dans le processeur SPARC on trouve une instruction de lecture d'un octet signé `LDSB` ou non signé `LDUB`. Le tableau 15.10 précise quels fils du bus données sont envoyés vers chaque bit du registre R ; certaines lignes ne sont pas précisées : lorsqu'il y a une demande d'accès d'un mot de 16 bits à une adresse impaire, le décodeur d'adresses envoie le signal d'erreur, et la valeur calculée par le circuit C n'a aucune importance.

4. Exercices

E15.1 Faire le schéma détaillé en portes du circuit C de la figure 15.9 en prenant les codages de taille dans la documentation d'un vrai processeur.
E15.2 Comment peut être étendu le mécanisme décrit au paragraphe 3.2 pour traiter des données de 32, 16 et 8 bits ?
E15.3 Chercher, dans des documentations techniques de processeurs, s'ils ont ou non une contrainte d'alignement des mots de 2^n octets sur frontière multiple de 2^n. En profiter pour regarder quelle convention a été adoptée : gros-boutiste ou petit-boutiste (Cf. Chapitre 4, paragraphe 2.2.4).

E15.4 : Une mémoire simple (Cet exercice fait suite à l'étude de cas du paragraphe 3.1)
Le processeur a toujours un bus données de 32 bits mais un bus d'adresses de 24 bits. Nous disposons de 16 boîtiers de 1Mo. Décrire l'organisation de la mémoire et le décodage d'adresses afin de disposer d'une mémoire de $4 * 2^{20}$ mots de 32 bits, sachant que l'on veut pouvoir accéder à des octets, à des mots de 16 bits ou à des mots de 32 bits.

E15.5 : Une mémoire générale
On veut gérer une mémoire dans laquelle l'accès à des octets, des mots de 16 ou 32 bits est possible. On dispose des boîtiers suivants :

- 4 boîtiers de ROM de 256Ko accessibles en mode superviseur,
- 4 boîtiers de RAM de 256Ko accessibles en mode superviseur,
- 8 boîtiers de RAM de 1Mo accessibles en mode superviseur et utilisateur,
- 1 boîtier servant aux entrées/sorties de 3 mots de 8 bits, accessibles en mode superviseur, le premier mot pouvant être lu et/ou écrit, le deuxième ne pouvant être que lu et le troisième uniquement écrit,
- 1 boîtier servant aux entrées/sorties de 3 mots de 16 bits, accessibles en

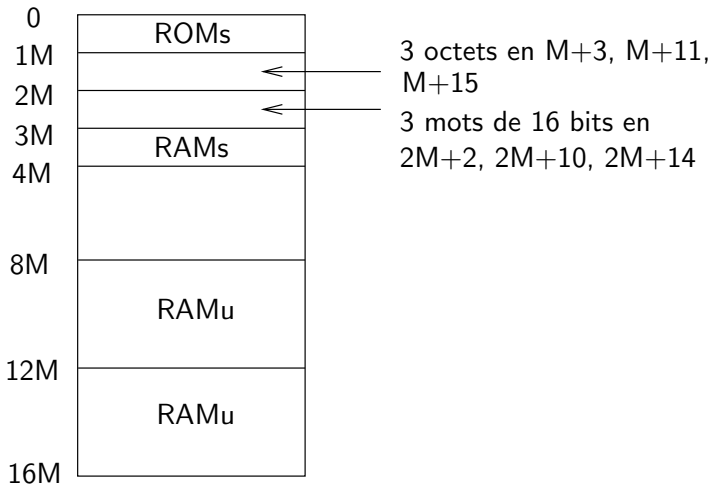


FIG. 15.11 – Organisation d'une mémoire générale

mode superviseur, le premier mot pouvant être lu et/ou écrit, le deuxième ne pouvant être que lu et le troisième uniquement écrit.

Remarque : Un boîtier de 256K est un boîtier 2^{18} octets, $2^{18} = 2^8 * 2^{10} = 256 * 1024 \approx 256 * 10^3$.

La figure 15.11 décrit l'implantation de chacun de ces boîtiers, ou mots pour la mémoire servant aux entrées/sorties. Pour cette dernière, pour l'instant, on peut considérer que ce sont des mots mémoire avec seulement des contraintes de lecture et/ou d'écriture ; on verra au chapitre 16 comment les utiliser.

Remarque : Les adresses d'implantation sur la figure 15.11 sont données en M (méga-octets) pour alléger le dessin. Notons que $1M = 10^6 = 10^3 * 10^3 \approx 2^{10} * 2^{10} = 2^{20}$.

Le processeur a des données sur 32 bits et des adresses sur 32 bits. Il émet les signaux : AccèsMem, l/$\overline{\text{e}}$, et le type d'accès (octet, 16 bits ou 32 bits). De plus, il envoie un signal s/$\overline{\text{u}}$ indiquant s'il est en mode superviseur ou utilisateur. Nous verrons au chapitre 22 que le processeur possède différents modes d'exécution mais pour l'instant on se contentera d'émettre une erreur d'accès mémoire dans le cas où un accès à une zone superviseur est demandée alors que le processeur n'est pas en mode superviseur.

Faire un schéma précis de l'organisation de cette mémoire et décrire le décodeur d'adresses. La fonction de décodage d'adresses doit calculer les signaux d'accès à chacun des boîtiers utilisés et le signal d'erreur d'accès mémoire.

Chapitre 16

Circuits d'entrées/sorties

Nous avons vu au chapitre 15 comment se passent les transferts entre le processeur et la mémoire. Nous voulons maintenant enrichir notre ordinateur minimal en offrant des moyens de communication entre l'ensemble processeur/mémoire et le monde extérieur. Le monde extérieur peut être un autre ordinateur ou un ensemble d'organes périphériques tels que clavier, lecteur de disquettes, imprimante, écran, capteurs, etc.

Lorsqu'il y a communication d'information de l'ensemble processeur/mémoire en direction du monde extérieur, on parle d'une *sortie*, et lorsque l'échange a lieu depuis le monde extérieur vers l'ensemble processeur/mémoire on parle d'une *entrée*.

Les circuits d'entrées/sorties assurent la gestion des échanges entre le processeur et les périphériques, et plus particulièrement gèrent la synchronisation entre ces dispositifs qui ont des vitesses de fonctionnement différentes.

Le paragraphe 1. présente la notion d'entrées/sorties et précise ce que l'on appelle un circuit d'entrées/sorties. Les aspects de synchronisation mis en jeu lors d'une communication entre le processeur et des organes périphériques sont abordés dans le paragraphe 2. Dans le paragraphe 3. nous montrons comment connecter matériellement des organes périphériques à l'ensemble processeur/mémoire. La programmation de sorties et d'entrées élémentaires ainsi que l'interface matérielle nécessaire sont présentées dans les paragraphes 4. et 5. Dans le paragraphe 6. nous nous intéressons à l'enchaînement d'entrées/sorties et présentons des moyens d'optimiser des tranferts de blocs : notions d'accès direct à la mémoire (DMA), de canal et de processeur d'entrées/sorties.

1. Notion d'entrées/sorties

Pour fixer les idées, considérons une configuration simple avec deux organes périphériques : un clavier pour les entrées et un afficheur sept segments (Cf. Exemple E8.2) pour les sorties (Cf. Figure 16.1).

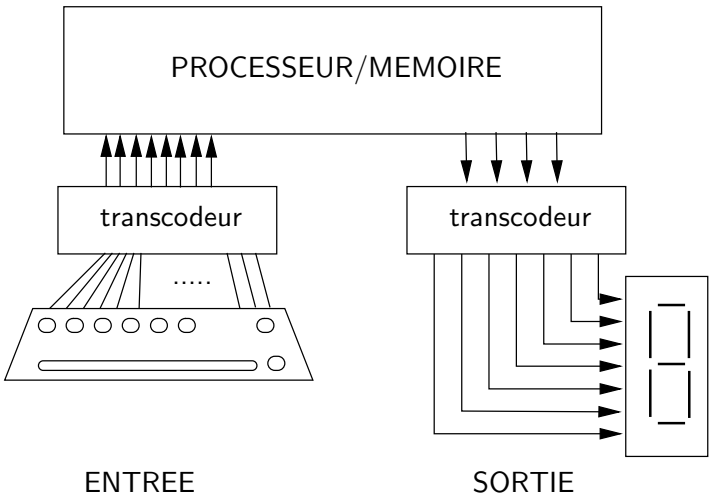


FIG. 16.1 – Entrée clavier et sortie afficheur sept segments

La frappe d'une touche du clavier provoque l'envoi de signaux à un transcodeur qui fournit un codage de cette valeur sur 7 bits (code ASCII, Cf. Chapitre 3). Cette information doit alors être traitée par l'ensemble processeur/mémoire. Deux types de questions se posent : où cette information est-elle stockée ? Comment gérer le flux des caractères frappés à un rythme complètement libre par l'utilisateur ?

Imaginons maintenant un programme qui calcule des valeurs entières sur 4 bits à délivrer à l'utilisateur, c'est-à-dire à afficher sur l'afficheur sept segments. On peut supposer que ces valeurs sont stockées dans la mémoire associée au processeur. Un transcodeur fabrique les signaux permettant d'allumer les bons segments de l'afficheur afin de donner la représentation en hexadécimal de l'entier en question. Là aussi le même type de questions se pose : comment la mémoire est-elle reliée à ce transcodeur ? A quel rythme est-il possible (souhaitable) d'envoyer les informations ?

Que ce soit pour une entrée ou une sortie, on voit qu'il y a deux aspects à prendre en compte : la réalisation de la connexion physique (matérielle) entre le périphérique et l'ensemble processeur/mémoire et la gestion de la synchronisation des échanges. Il est ainsi nécessaire d'intercaler entre tout périphérique et l'ensemble processeur/mémoire un circuit de commande que l'on appelle *circuit d'entrées/sorties* ou *coupleur de périphérique*.

Dans la suite du chapitre, nous commençons par préciser les aspects liés à la synchronisation des échanges, puis nous présentons les aspects matériels de connexion (quels fils et circuits faut-il ajouter et où ?). Enfin nous montrons comment programmer une entrée ou une sortie, c'est-à-dire comment utiliser les circuits d'entrées/sorties depuis un programme en langage d'assemblage.

2. Synchronisation entre le processeur et un périphérique

Les problèmes qui se posent ici sont inhérents à toute communication : perte d'une partie de l'information qui doit être échangée ; répétition d'une même information déjà échangée (Cf. Chapitre 6). Avant d'étudier précisément le type de protocole qu'il est nécessaire d'appliquer, nous allons discuter du niveau auquel cette synchronisation intervient.

2.1 A quel niveau se passe la synchronisation ?

Les échanges entre l'ensemble processeur/mémoire et le monde extérieur peuvent être décomposés en deux niveaux : échanges entre processeur/mémoire et périphérique et échanges entre le périphérique et le monde extérieur.

Lors de la lecture d'un caractère, l'ensemble processeur/mémoire dialogue avec le périphérique clavier mais au-delà du clavier il y a un utilisateur. Cet utilisateur tape des caractères à la vitesse qui lui convient, c'est-à-dire de façon complètement arbitraire et non contrôlable. Le système informatique doit gérer la synchronisation entre le processeur et le périphérique car on ne peut rien contrôler entre l'utilisateur et le périphérique.

Envisageons maintenant un cas où la vitesse imposée n'est pas celle d'un utilisateur mais celle d'un organe mécanique. Par exemple, prenons le cas d'une imprimante. Là, on connaît la vitesse à laquelle l'imprimante peut afficher des caractères et donc assurer une synchronisation de façon à ne pas en perdre. Il peut toutefois survenir des problèmes à des instants non prévisibles comme par exemple l'absence de papier : le problème est en général traité au niveau du système qui gère une liste des fichiers en attente d'impression et qui n'envoie une information à l'imprimante que si celle-ci peut la traiter.

Dans le cas général, il convient donc de tenir compte des erreurs et anomalies inhérentes à toute interaction avec l'environnement extérieur.

Nous avons vu au chapitre 15 comment gérer les échanges entre la mémoire et le processeur. La connexion d'organes périphériques ne peut pas s'en inspirer directement ; en effet, les entrées/sorties présentent des particularités qui les distinguent des accès mémoire :

- la vitesse des processeurs et des mémoires à semiconducteurs est supérieure de plusieurs ordres de grandeur à celle des périphériques mettant en jeu des dispositifs mécaniques. A titre d'illustration une imprimante à impact (à marguerite, matricielle à aiguille, etc.) atteignant la dizaine de milliers de caractères imprimés à la seconde représenterait déjà une prouesse mécanique alors que tous les processeurs récents dépassent la centaine de millions d'instructions par seconde.
- le processeur ne peut décider seul des instants auxquels les échanges seront effectués : il ne peut par exemple deviner à quel moment l'utilisateur va appuyer sur une touche du clavier. Au mieux, il est possible de lisser

les problèmes en imaginant des mécanismes de mémorisation permettant d'accumuler les caractères frappés au clavier en attendant que le processeur les traite. Dans la pratique, il n'y a pas réellement de problème car la vitesse des processeurs est nettement supérieure au temps de réaction d'un utilisateur.

2.2 Synchronisation par poignée de mains

Une entrée (ou une sortie) met en jeu deux entités : un émetteur et un récepteur. Dans le cas d'une entrée l'émetteur est le périphérique et le récepteur est le processeur. Dans le cas d'une sortie c'est l'inverse. Il faut mettre en oeuvre un protocole d'échange qui permette au récepteur de détecter l'arrivée des informations à consommer et assurer un contrôle de flux, autrement dit éviter que l'émetteur ne soumette des informations plus vite que le récepteur ne peut les traiter ou que le récepteur ne traite plusieurs fois la même information.

Par exemple, considérons la sortie de caractères sur une imprimante ; le protocole d'échange doit assurer que le processeur n'émettra pas un nouveau caractère si le précédent n'a pas encore été imprimé et que l'imprimante n'imprime pas plusieurs fois le même caractère.

Dans le cas général, l'échange entre un processeur et un périphérique peut être régi par le protocole de dialogue dit *poignée de mains* présenté au chapitre 6.

La mise en oeuvre de ce protocole réclame la gestion de signaux de synchronisation disant si l'information à échanger est parvenue ou non au récepteur et si le récepteur a traité ou non l'information qu'il a reçue.

Dans un programme d'entrée ou de sortie la valeur de ces signaux de synchronisation est testée et tant que le signal attendu n'a pas la bonne valeur il faut le tester à nouveau jusqu'à ce qu'il devienne correct. On parle d'*attente active* car le processeur qui exécute ce test ne peut rien faire d'autre pendant ce temps-là, il est mobilisé pendant toute la durée du transfert.

> *Nous verrons au chapitre 24, paragraphe 3. que d'autres solutions (utilisant la notion d'interruption) plus efficaces et plus réalistes sont en fait mises en oeuvre dans les systèmes multitâches.*

3. Connexion d'organes périphériques

3.1 Notion de coupleur

Le circuit nécessaire à la communication s'appelle *coupleur*. Le processeur perçoit le système d'entrées-sorties comme un ensemble d'emplacements mémoire reliés au monde extérieur. L'usage a consacré le terme de *registres du coupleur* pour ces emplacements.

Cela signifie que certaines adresses de la mémoire sont réservées à l'usage

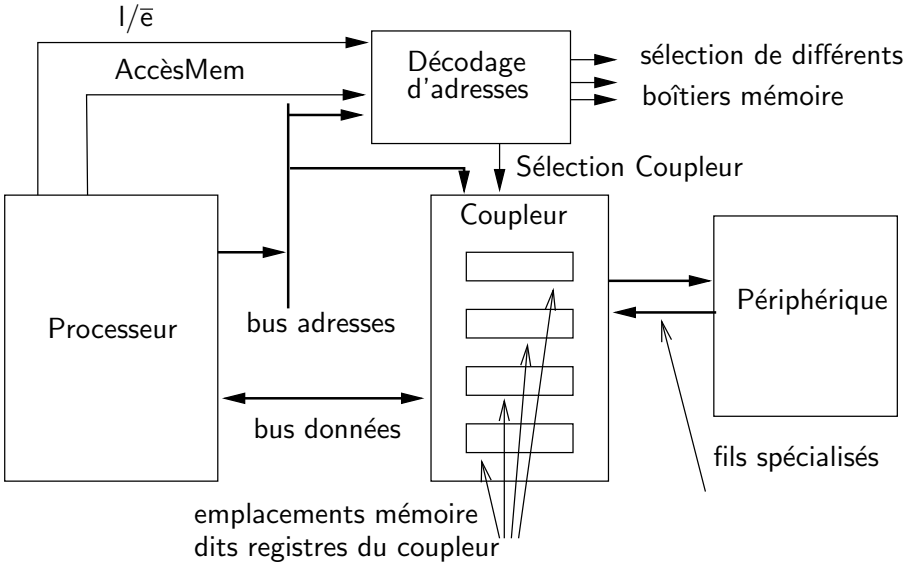


FIG. 16.2 – Connexion d'un processeur et d'un périphérique via un coupleur

des entrées/sorties. C'est le *décodage d'adresses* qui prend en compte cette nouvelle contrainte.

Les données transitant par le bus données (Cf. Chapitres 9 et 15) il faut qu'une connexion physique soit réalisée entre le périphérique et le bus données via le coupleur. La figure 16.2 donne une première idée des éléments intervenants dans cette mise en oeuvre.

Pour réaliser une entrée il faut que des fils provenant de l'unité périphérique soient connectés au bus de données via des portes trois états ou des éléments de mémorisation. Pour réaliser une sortie il faut connecter le bus de données au périphérique via des éléments de mémorisation.

Le montage n'est pas nécessairement symétrique pour les entrées et les sorties. En effet, lors d'une sortie, la valeur émise par le processeur apparaît fugitivement sur le bus données pendant le cycle d'écriture et doit donc être mémorisée. Par contre, en entrée, il peut suffire d'échantillonner la valeur au moment de la lecture sans besoin de mémorisation. En effet, il est raisonnable de faire l'hypothèse qu'en entrée les valeurs sont stables : le périphérique est supposé maintenir les données pendant l'intervalle de temps spécifié par le protocole de synchronisation. Nous nous plaçons dans ce cas de figure pour la suite du chapitre.

3.2 Connexion de périphériques à des éléments de mémorisation

Pour décrire les connexions physiques des fils, nous commençons par rappeler comment une cellule mémoire est connectée au bus données. Nous

considérons une cellule de 1 bit (représenté par un verrou) connectée au i$^{\text{ème}}$ fil du bus de données Di. Le schéma 16.3 montre la structure du circuit d'échange d'information entre ce bit et le processeur. AccèsMem et $\text{l}/\overline{\text{e}}$ sont les signaux de sélection et de lecture/écriture de la mémoire.

Nous voulons maintenant faire communiquer l'ensemble processeur/mémoire avec le monde extérieur. Considérons, par exemple, un monde extérieur très simple dans lequel le processeur doit pouvoir envoyer (écrire) une valeur sur des lampes, ou bien récupérer (lire) une valeur sur des interrupteurs.

Reprenons le schéma 16.3 et supprimons la connexion entre la sortie Q du verrou et la porte 3 états donnant l'accès au fil Di du bus données. Le montage (Cf. Figure 16.4) crée *un port d'entrée* et *un port de sortie* reliés sur l'exemple respectivement à un interrupteur et une lampe.

Le montage décrit permet la création d'*une* entrée élémentaire et d'*une* sortie élémentaire. Le port d'entrée et celui de sortie peuvent occuper la même adresse, comme ici, ou des adresses différentes.

L'écriture sur un port de sortie ou la lecture sur un port d'entrée mettent en jeu un mécanisme de sélection d'adresse analogue à celui que nous avons décrit au chapitre 15. Pour une écriture (respectivement une lecture) sur un port de sortie (respectivement d'entrée), le décodeur d'adresses reçoit une demande d'accès à la mémoire (AccèsMem=1) accompagnée d'une adresse correspondant à l'un des circuits consacrés aux entrées/sorties. Il active alors le signal de sélection de ce circuit : SelCoupleur.

Le coupleur doit lui-même comporter un décodeur d'adresses. Il fabrique les signaux de commande de chargement des bascules associées aux ports de sortie et les signaux de commande des portes 3 états associées aux ports d'entrée, à partir de SelCoupleur, du signal $\text{l}/\overline{\text{e}}$ émis par le processeur et de l'adresse. Sur la figure 16.4, une seule cellule mémoire est représentée ; nous n'avons donc pas tenu compte de l'adresse. La figure 16.5 décrit l'organisation d'un coupleur à deux ports d'entrée et deux ports de sortie. Nous pouvons y observer la place des décodeurs.

Dans la suite de ce chapitre, nous ne représenterons plus les différents décodeurs mais dessinerons le processeur, le décodeur d'adresses et le décodeur du coupleur comme un ensemble.

4. Programmation d'une sortie

4.1 Interface simplifiée

Nous traitons l'exemple de l'impression d'un texte (suite de caractères) sur une imprimante. Le coupleur permet de traiter trois informations :

– une donnée sur 8 bits, le caractère, qui est une sortie pour le processeur et une entrée pour l'imprimante,

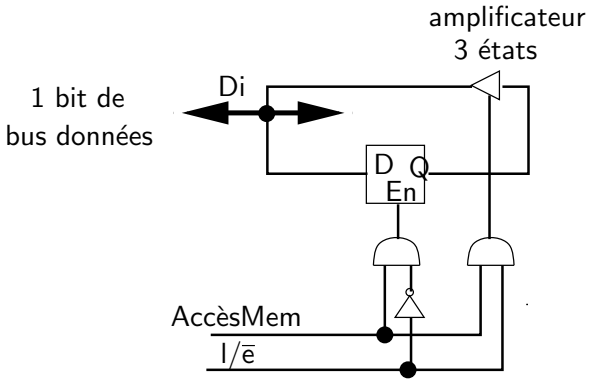


FIG. 16.3 – Echange entre le processeur et la mémoire sur un bit : lorsque le signal AccèsMem vaut 1 et le signal l/ē vaut 0 la bascule mémorise la valeur présente sur le bus et lorsque AccèsMem et l/ē valent tous deux 1 la valeur mémorisée dans la bascule est présente sur le bus.

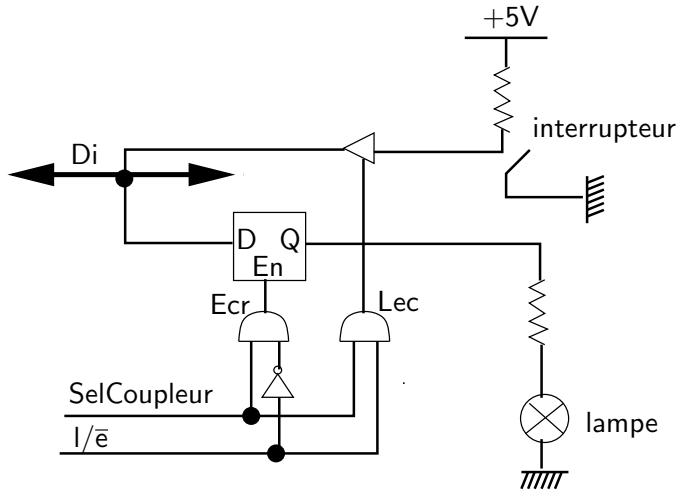


FIG. 16.4 – Echange entre le processeur et l'extérieur sur un bit : le signal électrique commandé par l'interrupteur apparaît sur le fil Di du bus données lors d'un cycle de lecture (SelCoupleur=1 et l/ē=1). Si l'interrupteur est fermé le processeur lira un 0, s'il est ouvert il lira un 1. Par ailleurs la valeur émise sur le fil Di par le processeur lors d'une écriture (SelCoupleur=1 et l/ē=0) est mémorisée par le verrou ; si cette valeur est 1 alors la lampe s'allume, si c'est 0 la lampe s'éteint.

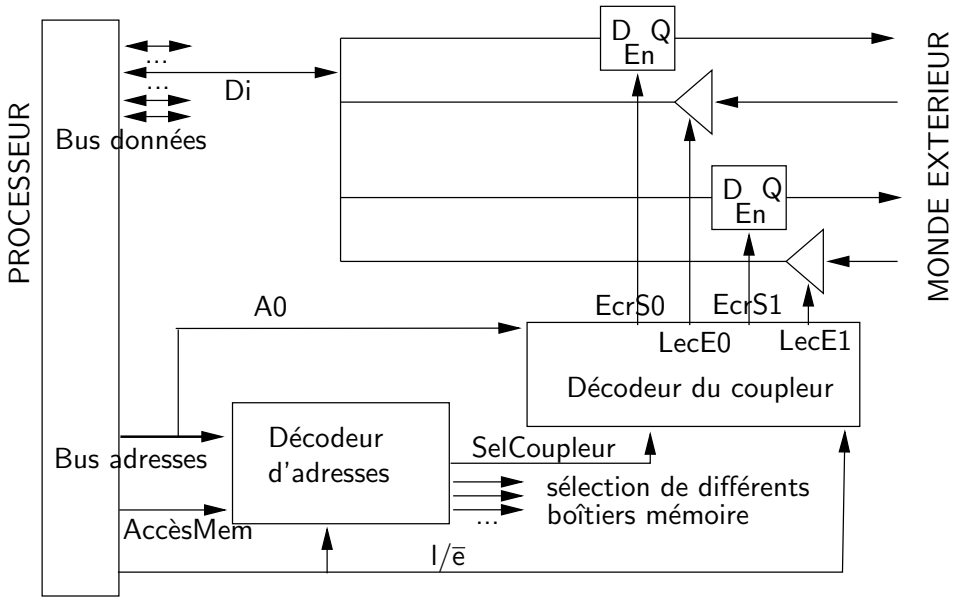


FIG. 16.5 – Coupleur à deux ports d'entrées et deux ports de sortie : le décodeur d'adresses active le signal SelCoupleur lorsqu'il reconnaît l'adresse qui lui est fournie comme une adresse du circuit d'entrées/sorties. En supposant que le bit d'adresse A0 permet de faire la distinction entre les deux mots du coupleur, le décodeur du coupleur fabrique les signaux de commande des 2 ports de sortie EcrS0 et EcrS1 et des 2 ports d'entrée LecE0 et LecE1, à partie des signaux SelCoupleur, I/ē et A0.

– un booléen processeur prêt qui signifie qu'un nouveau caractère est disponible et doit être imprimé,
– un booléen imprimante prête, qui signifie que l'imprimante est prête à traiter un caractère. Après une impression il signifie que le caractère précédent a été imprimé.

Le coupleur est composé d'un registre de données RegD et d'un registre de commande RegC auxquels on accède en écriture, et d'un registre d'état RegE auquel on accède en lecture. Vus du processeur RegD est à l'adresse RD et contient la donnée, RegC est à l'adresse RC et contient un seul bit significatif : processeur prêt ; RegE est à l'adresse RE et contient un seul bit significatif imprimante prête.

Posons comme convention que le signal processeur prêt est actif lorque le registre RegC vaut 1 et que le signal imprimante prête est actif lorsque le signal RegE vaut 1.

La figure 16.6 décrit cette organisation matérielle et la figure 16.7 donne les programmes d'initialisation du coupleur et d'impression d'un caractère stocké à l'adresse `car`. Ils sont écrits en langage d'assemblage 68000 (Cf. Chapitre 12 pour un exemple de syntaxe de ce langage d'assemblage). L'exécution de ces

programmes met en oeuvre l'algorithme de l'émetteur (voir le protocole poignée de mains dans le chapitre 6).

La boucle d'attente sur l'étiquette att-pret, correspond à l'état où le processeur attend que le récepteur soit libre ; lorsque c'est le cas, le processeur peut lui envoyer une valeur. La boucle d'attente sur l'étiquette att-traite correspond à l'intervalle de temps pendant lequel le processeur attend que le récepteur ait traité la donnée envoyée. Si le processeur n'attend pas, il risque d'écraser le caractère envoyé. La remise à zéro du registre RegC correspond à la désactivation du signal `processeur prêt`.

4.2 Gestion d'erreur

Dans ce paragraphe nous examinons comment prendre en compte les erreurs provenant d'un périphérique. Nous poursuivons avec l'exemple de l'imprimante en considérant le problème de l'absence de papier : l'imprimante ne doit plus recevoir de caractères.

Pour gérer le problème, il faut que le processeur puisse en être informé ; pour cela le registre d'état RegE va être complété par un nouveau signal erreur papier. Pour fixer les idées nous supposons que ce signal est connecté au bit 1 du bus données (Cf. Figure 16.8). Le programme d'impression d'un caractère, modifié pour prendre en compte la gestion de l'erreur, est donné dans la figure 16.9.

En général, les informations contenues dans le registre d'état d'un coupleur permettent d'effectuer les tests liés aux aspects de synchronisation de l'échange et de gérer les différents types d'erreurs liés au fonctionnement du périphérique associé.

4.3 Interface optimisée

Nous étudions dans ce paragraphe une version optimisée du programme de sortie d'un caractère sur une imprimante. On va chercher à réduire le nombre d'accès mémoire faits par le processeur pour réaliser le transfert d'une donnée. Il faudra ajouter un peu de matériel mais celui-ci n'est pas vraiment coûteux. Nous repartons de la version de base ne traitant pas d'erreurs.

Tout d'abord, remarquons que lors d'une sortie, l'activation de processeur prêt va toujours de pair avec l'écriture dans le registre de données. De plus, hors initialisation, sa désactivation suit toujours le front descendant du signal imprimante prête.

Le registre de commande RegC peut être remplacé par une bascule RS mise à un par le signal d'écriture dans le registre de données et remise à zéro par le signal $\overline{\text{imprimante prête}}$. Lors de l'initialisation, la bascule doit être mise à 0. La bascule remplace le registre de commande et sa sortie se substitue au signal processeur prêt (Cf. Figure 16.10).

D'autre part, pour pouvoir soumettre un nouveau caractère, il faut à la fois que l'imprimante soit prête (imprimante prête actif) et que le précédent

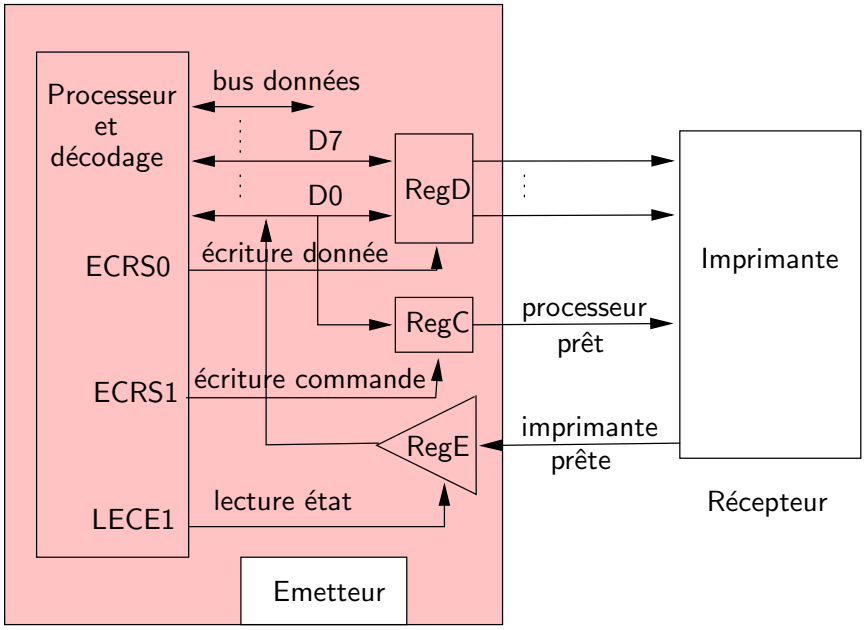


FIG. 16.6 – Exemple de coupleur d'imprimante. L'émetteur est constitué du processeur, du coupleur et des décodeurs, le récepteur est l'imprimante. Cette organisation nécessite 2 ports de sortie (S0 pour les données, S1 pour la commande) et un port d'entrée pour l'état (E1).

```
IMP-PRETE = 1
PROC-PRET = 1
NON-PROC-PRET = 0
          .data
car :     .ascii 'A'                  ! code du caractère A
          .text
Init :    moveq #NON-PROC-PRET, RC !le processeur n'a rien à émettre

ImpCar :  ! attendre que le périphérique soit prêt
att-pret :move.b RE, D1               ! D1 est un reg. donnée libre
          andi.b #IMP-PRETE, D1
          beq att-pret                ! l'imprimante n'est pas prête
          ! périphérique prêt : envoyer la valeur à imprimer
          move.b car, RD
          moveq #PROC-PRET, RC
att-trt : move.b RE, D1
          andi.b #IMP-PRETE, D1
          bne att-trt                 ! le périphérique traite
          ! le caractère a été imprimé
          moveq #NON-PROC-PRET, RC
          rts
```

FIG. 16.7 – Programme de sortie simple

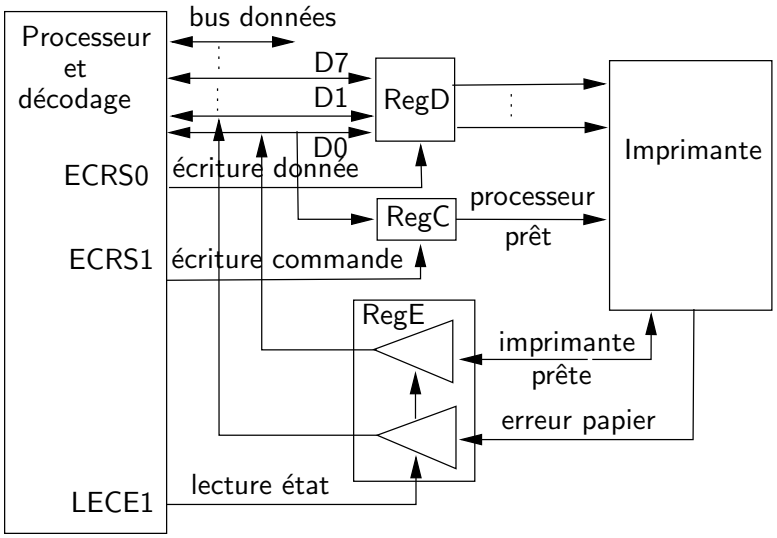


FIG. 16.8 – Coupleur d'imprimante avec gestion de l'erreur : absence de papier

```
IMP-PRETE = 1 ; PROC-PRET = 1
NON-PROC-PRET = 0
ERREUR = 2   ! bit 1 du registre RE : 2^1
             .data
car :        .ascii 'A
             .text
ImpCar :     ! A la fin du traitement D1 contient 0 si tout
             ! s'est bien passé et 1 s'il y a eu une erreur
att-pret :   move.b RE, D1
             andi.b #ERREUR, D1
             bne pb-papier                ! plus de papier
             move.b RE, D1
             andi.b #IMP-PRETE, D1
             beq att-pret
             move.b car, RD
             moveq #PROC-PRET, RC
att-traite :move.b RE, D1
             andi.b #ERREUR, D1
             bne pb-papier
             move.b RE, D1
             andi.b #IMP-PRETE, D1
             bne att-traite
OK :         moveq #0, D1
             bra fin
pb-papier :  moveq #1, D1
fin :        moveq #NON-PROC-PRET, RC
             rts
```

FIG. 16.9 – Programme de sortie gérant un type d'erreur

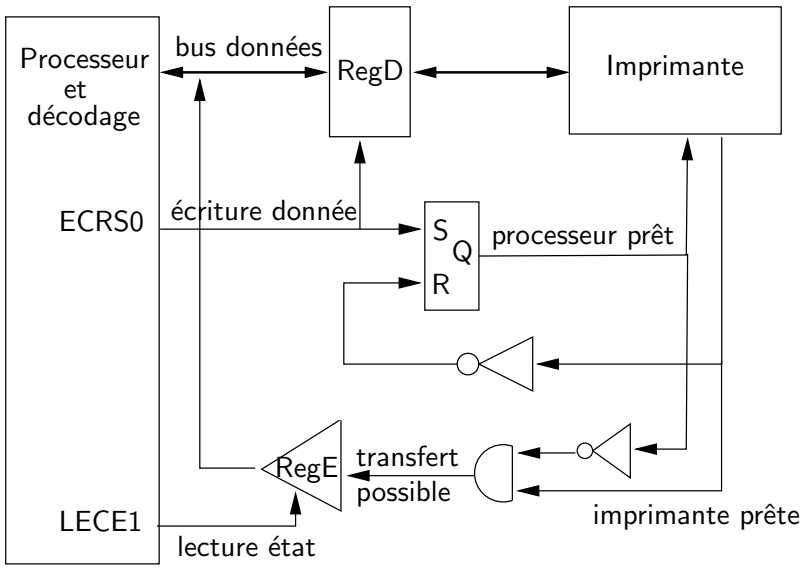


FIG. 16.10 – Exemple de coupleur d'imprimante optimisé

caractère ait été récupéré par l'imprimante, ce qui est détectable par le fait que processeur prêt soit inactif. Si cette nouvelle double condition (transfert possible sur la figure 16.10) est testée à la place de la simple condition imprimante prête, la boucle d'attente qui suit l'émission d'un caractère peut être supprimée.

Le processeur n'a plus alors qu'à tester transfert possible et écrire le caractère à transférer dans le registre de données. Le signal processeur prêt est automatiquement mis à jour par la bascule. La présence de la bascule RS remplace les accès au registre de commande.

La figure 16.10 décrit le matériel nécessaire à cette nouvelle interface et le programme d'impression d'un caractère sur l'imprimante dans ce nouveau contexte est décrit dans la figure 16.11 .

> **Remarque :** Nous verrons, au paragraphe 6. de ce chapitre et au paragraphe 3. du chapitre 24, deux méthodes pour éliminer l'attente active sur `att-pret` exécutée par le processeur.

5. Programmation d'une entrée

5.1 Interface simplifiée

Nous considérons l'exemple de la lecture d'un caractère au clavier.

Le coupleur du clavier comporte trois informations : la donnée (adresse RD) et le booléen clavier prêt (adresse RE) accessibles en lecture, et le booléen processeur prêt (adresse RC) accessible en écriture. clavier prêt signifie qu'un caractère a été frappé sur le clavier. processeur prêt signifie que le processeur

```
TRANSFERT-POSSIBLE = 1
          .data
car :     .ascii 'A
          .text
ImpCar :
att-pret :move.b RE, D1
          andi.b #TRANSFERT-POSSIBLE, D1
          beq att-pret
          move.b car, RD
          ! la bascule RS passe a 1, l'imprimante sait qu'elle
          ! doit prendre un caractere
          rts
```

FIG. 16.11 – Programme de sortie optimisé

est prêt à traiter un caractère. Après une lecture antérieure cela signifie que le caractère précédemment envoyé a été récupéré.

Les figures 16.12 et 16.13 décrivent respectivement l'organisation matérielle et les programmes d'initialisation du coupleur et de lecture d'un caractère.

5.2 Interface optimisée

L'idée consiste, comme dans le cas de la sortie, à remplacer le registre de commande par une bascule RS (Cf. Figure 16.14) et à supprimer la boucle d'attente après la récupération du caractère lu. Pour cela, constatons que la lecture du caractère envoyé par le clavier doit faire passer le signal processeur prêt à zéro (entrée R de la bascule). Lorsque le signal clavier prêt devient inactif, la bascule est alors remise à 1. La bascule doit être initialisée à 1. D'autre part, pour pouvoir lire un nouveau caractère, il faut que le clavier en ait soumis un (clavier prêt actif) et que le processeur ne l'ait pas déjà lu (processeur prêt actif) ; cette double condition constitue le nouveau booléen : transfert possible.

Les figures 16.14 et 16.15 décrivent respectivement le matériel nécessaire et le programme optimisé de lecture d'un caratère au clavier.

6. Optimisation des entrées/sorties groupées

Nous nous intéressons maintenant à une situation très classique consistant à enchaîner plusieurs entrées/sorties.

Par exemple, pour effectuer la sortie des éléments d'un tableau de n caractères (zone de mémoire de n octets consécutifs), on peut insérer le programme de sortie d'un caractère dans une boucle de parcours du tableau. Cette solution n'est pas très efficace car pour chaque caractère, une attente va avoir lieu.

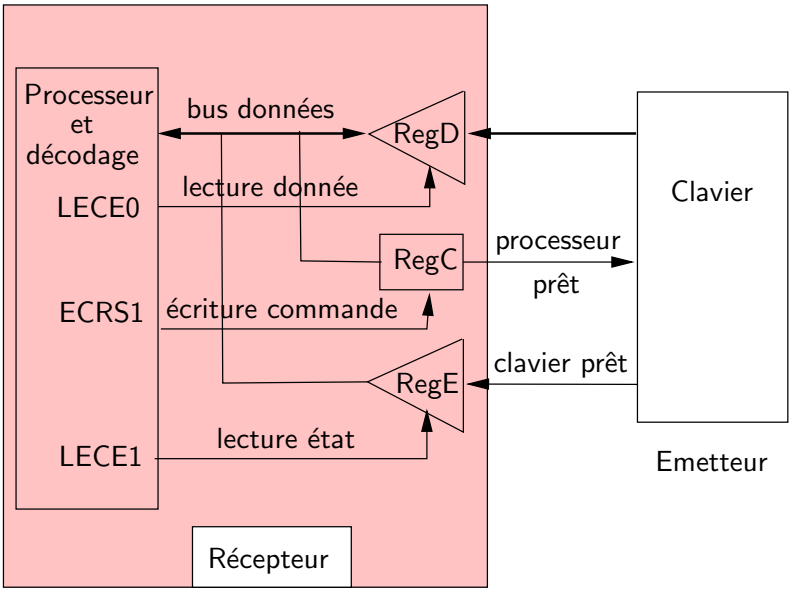


FIG. 16.12 – Exemple de coupleur de clavier. L'émetteur est le clavier, le récepteur est constitué du processeur, du coupleur et des décodeurs. Cette organisation nécessite 2 ports d'entrées (E0 pour la donnée, E1 pour l'état) et un port de sortie pour la commande (S1).

```
CLAVIER-PRET = 1
PROC-PRET = 1
NON-PROC-PRET = 0
            .data
            ! un octet initialisé à 0 pour stocker le caractère lu
car :       .byte 0
            .text
Init :      ! le processeur est prêt à recevoir
            moveq ♯PROC-PRET, RC

LireCar :
att-clavier move.b RE, D1 ! attendre périphérique prêt
            andi.b ♯CLAVIER-PRET, D1
            beq att-clavier             ! le clavier n'a rien envoyé
            ! le périphérique est prêt : récupérer le caractère
            move.b RD, car
            moveq ♯NON-PROC-PRET, RC
att-traite :move.b RE, D1 ! attendre le traitement
            andi.b ♯CLAVIER-PRET, D1
            bne att-traite
            ! le caractere a été lu
            moveq ♯PROC-PRET, RC
            rts
```

FIG. 16.13 – Programme d'entrée simple

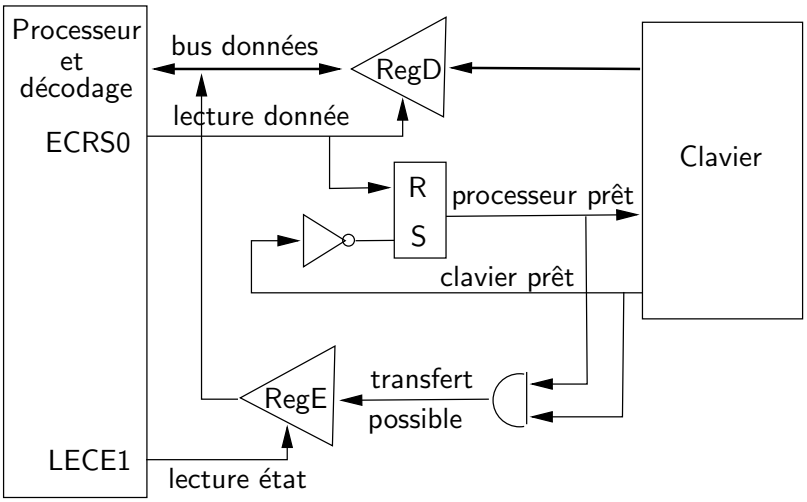


FIG. 16.14 – Exemple de coupleur de clavier optimisé

```
TRANSFERT-POSSIBLE = 1
            .data
car :       .byte 0
            .text
LireCar :
att-clavier move.b RE, D1
            andi.b ♯TRANSFERT-POSSIBLE, D1
            beq att-clavier
            move.b RD, car
            rts
```

FIG. 16.15 – Programme d'entrée optimisé

Il est possible de décharger le processeur d'une partie de ce travail en ajoutant un circuit qui s'en chargera. Le circuit effectuant les entrées/sorties est un automate câblé dont la réalisation est peu coûteuse : un registre contenant une adresse, un registre contenant le nombre d'éléments à transférer, un incrémenteur, un décrémenteur et quelques bascules pour l'automate de commande.

Cette technique est appelée *accès direct à la mémoire* (en anglais *Direct Memory Access*, *DMA*), l'interface accédant directement à la mémoire.

Dans la suite, nous présentons les aspects d'organisation matérielle d'un système comportant un processeur, de la mémoire, un circuit d'entrées/sorties et un contrôleur d'accès direct à la mémoire ; puis nous décrivons le déroulement d'une sortie. Enfin nous évoquons l'évolution des systèmes d'entrées/sorties.

6.1 Accès direct à la mémoire et partage de bus

Cette technique permet de réaliser par matériel le transfert impliqué par une entrée ou une sortie. De plus, elle libère le processeur pendant l'attente de la disponibilité du périphérique. Ainsi, le processeur peut récupérer, pour effectuer des calculs, le temps qu'il utilisait précédemment à exécuter une attente active.

La réalisation de l'accès direct à la mémoire par un contrôleur indépendant du processeur pose un certain nombre de problèmes que nous examinons ci-dessous. La figure 16.16 décrit l'organisation du dispositif.

Il faut gérer l'accès à la mémoire (adresses, données, signaux d'accès mémoire et lecture/écriture) à la fois par le processeur et par le contrôleur d'accès direct à la mémoire (CDMA). Le principe est de connecter alternativement les bus au processeur et au CDMA, via des amplificateurs à sortie trois états. Le CDMA accède à la mémoire en la pilotant directement à la place du processeur, temporairement déconnecté du bus (Cf. Paragraphe 1.3 du chapitre 15 pour la réalisation matérielle de cette déconnexion). La gestion du bus adresses demande deux signaux de dialogue entre le processeur et le CDMA : une demande de libération du bus émise par le CDMA (demande bus) et l'autorisation correspondante émise par le processeur (libère bus).

Le processeur doit pouvoir autoriser ou non le coupleur à émettre des requêtes de transfert en direction du CDMA. Pour cela, on utilise un booléen autorisation requête dma dans le registre de commande du coupleur. Ce booléen est mis à jour par le processeur lorsqu'il initie un transfert.

Le coupleur doit pouvoir signaler au CDMA qu'il faut faire un transfert lorsque le périphérique est disponible. Cette information est matérialisée par le signal requête transfert qui est un et logique entre tranfert possible (Cf. Paragraphe 4.3 et figure 16.10) et autorisation requête dma.

Le CDMA doit pouvoir accéder directement à la donnée du coupleur sans passer par le décodage d'adresses standard. De façon plus précise, le CDMA

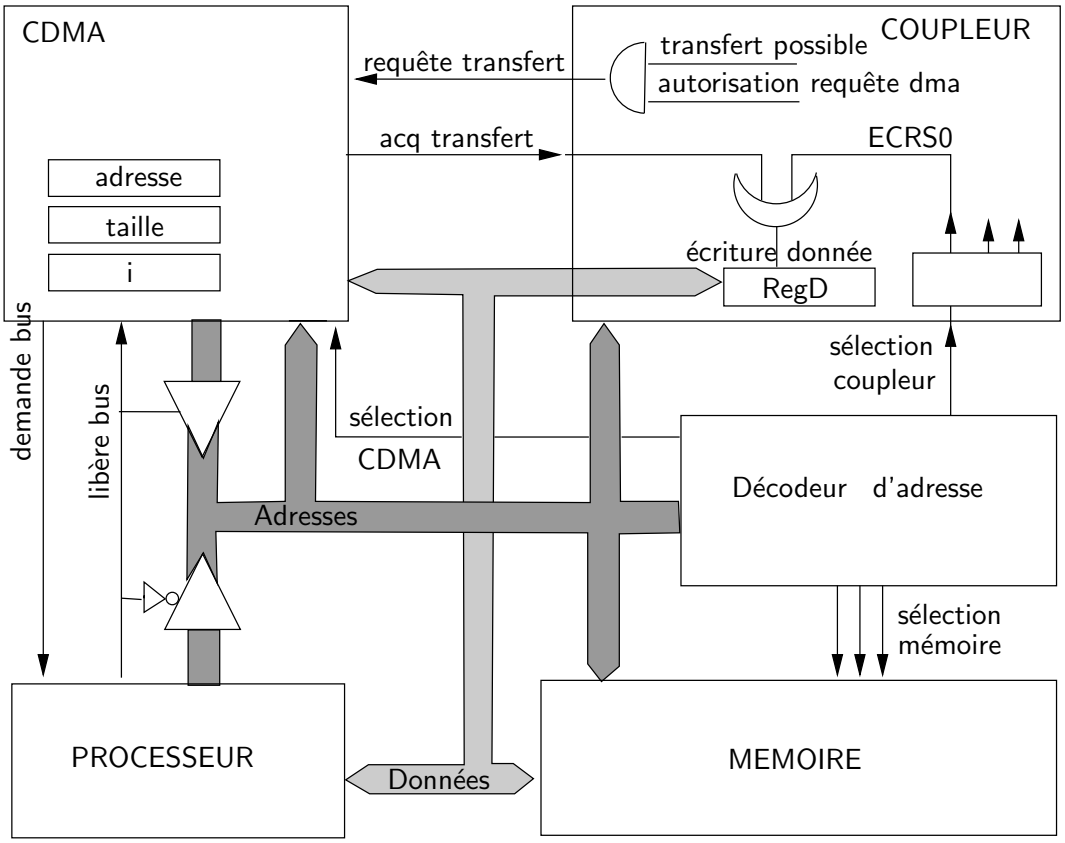


FIG. 16.16 – Accès à la mémoire avec DMA : on observe les 5 composants : processeur, contrôleur de DMA, coupleur, mémoire et dispositif de décodage d'adresse qui sélectionne les boîtiers. Le CDMA comporte des registres décrivant la zone de mémoire à transférer (adresse et taille) et un registre index (i). Via deux amplificateurs 3 états, le bus d'adresses est piloté soit par le processeur, soit par le CDMA. Le bus d'adresses est une entrée pour le CDMA, le coupleur et la mémoire : il sert à sélectionner le registre du CDMA ou du coupleur ou le mot mémoire lors d'un accès initié par le processeur. Le bus d'adresses est une sortie pour les deux maîtres : le processeur et le CDMA ; il sert alors à sélectionner le registre du coupleur ou le mot mémoire.

soit pouvoir sélectionner simultanément la mémoire en lecture et le coupleur en écriture. Ceci suppose l'ajout d'un signal acq transfert. La sélection effective du registre de données du coupleur est alors le ou logique entre acq transfert et le signal de sélection normal (ECRS0 sur la figure 16.10).

> **Remarque :** Dans le cas d'une entrée, le CDMA accèdera simultanément au coupleur en lecture et à la mémoire en écriture.

6.2 Déroulement d'une sortie avec accès direct à la mémoire

Nous pouvons maintenant décrire plus précisément le déroulement d'une sortie avec CDMA. Nous considérons l'exemple de la sortie d'un tableau `tab` de `n` octets :

1. Le processeur écrit l'adresse `tab` et la taille `n` du tableau à transférer dans les registres `adresse` et `taille` du CDMA. Ceci provoque l'initialisation du registre `i` (compteur d'octets transférés) du CDMA.
2. Le processeur autorise le coupleur à émettre une requête lorsqu'il est prêt : il écrit dans le registre de commande du coupleur pour mettre le booléen autorisation requête dma à vrai.
3. Le processeur vaque à d'autres occupations.
4. Lorsque le périphérique a terminé le travail qu'il effectuait précédemment, il devient prêt et le coupleur active le signal requête transfert.
5. Le CDMA active le signal demande bus.
6. Le processeur termine l'éventuel accès mémoire en cours et active libère bus pour indiquer que le bus est libre. Le processeur est alors déconnecté du bus.
7. Le CDMA émet l'adresse de l'octet courant, accède à la mémoire en lecture et active simultanément acq transfert pour écrire la donnée dans le coupleur. Pendant ce temps, l'exécution de l'instruction courante par le processeur peut se poursuivre jusqu'à ce qu'un accès mémoire soit nécessaire.
8. Le périphérique reçoit le caractère à traiter et désactive requête transfert. Le CDMA désactive à son tour demande bus et incrémente le registre `i`. Le processeur se connecte à nouveau au bus en désactivant libère bus et poursuit son travail jusqu'au prochain transfert de caractère.

La politique de partage des bus mémoire peut être plus ou moins sophistiquée. Elle peut être rudimentaire et pré-établie (par exemple accès par le processeur durant la demi-période haute d'une horloge, et par le CDMA pendant l'autre demi-période). Elle peut être confiée à un maître privilégié (tel que le processeur), qui décide seul des instants auxquels il va céder les bus.

Les bus peuvent être partagés entre un nombre quelconque de maîtres, selon des stratégies d'allocation élaborées telles que des priorités fixes ou tournantes. L'attribution des bus est alors gérée par un circuit d'arbitrage, trop sophistiqué pour être détaillé dans cet ouvrage.

6.3 Canaux et processeurs d'entrées/sorties

L'étape suivante est celle du *canal d'entrées/sorties* qui s'est surtout développé dans les grands systèmes de gestion transactionnelle connectés à de nombreux disques et terminaux.

Il s'agit d'une unité d'accès direct à la mémoire capable d'enchaîner automatiquement des transferts de blocs dont les paramètres (adresse et taille du tableau, périphérique et sens du transfert) sont stockés en mémoire par le processeur d'une part, et d'entrelacer des entrées et sorties avec plusieurs périphériques d'autre part. La suite d'ordres d'entrées/sorties que le canal lit en mémoire et exécute est quelquefois appelée programme canal.

Le canal peut être muni de la possibilité d'effectuer des itérations et d'effectuer ou non certaines entrées/sorties selon certaines conditions. Cette extension du canal aboutit à la notion de *processeur d'entrées/sorties* doté de tout un jeu d'instructions de comparaison et de branchement en plus des instructions d'entrée et de sortie de blocs de caractères. Citons à titre d'exemple le processeur 8089 dans la famille 8086 d'INTEL.

A partir de là, l'évolution du système d'entrées/sorties peut emprunter plusieurs directions. On peut disposer de (co)processeurs d'entrées/sorties dédiés chacun à un type de périphérique (processeur graphique, processeur de signaux sonores, etc.). On peut choisir de dupliquer le processeur de calcul, l'un des deux processeurs se substituant au processeur d'entrées/sorties. On obtient alors un multiprocesseur à mémoire commune.

Actuellement, la plupart des terminaux graphiques sont des *périphériques* dits *intelligents* qui sont dotés d'une certaine puissance de traitement, en particulier pour les traitements graphiques.

7. Exercices

E16.1 : Circuit d'entrées/sorties
Récupérer la documentation d'un circuit d'entrées/sorties (par exemple RS232, PIA). Repérer les registres du coupleur. Retrouver l'implantation des signaux évoqués dans ce chapitre ; en général, ils sont représentés par certains bits des registres du coupleur. Etudier les types de problèmes gérés par le circuit. Ecrire les programmes d'entrées/sorties.

E16.2 : Contrôleur d'accès direct à la mémoire
Etudier la structure interne d'un circuit contrôleur de DMA.

Chapitre 17

Pilotes de périphériques

En introduction de la partie V nous avons structuré le système d'exploitation en 2 parties :

- Une partie basse fortement dépendante des caractéristiques du matériel et fournissant des fonctionnalités très proches de celui-ci mais sous une forme normalisée. Il s'agit des bibliothèques de gestion des périphériques, appelées *pilotes de périphériques*. C'est l'objet de ce chapitre.
- Une partie haute utilisant les primitives de la précédente pour offrir des services de plus haut niveau sémantique, en l'occurrence le SGF (Cf. Chapitre 19), ou le chargeur/lanceur (Cf. Chapitre 20).

L'intérêt d'une couche intermédiaire entre la programmation de très bas niveau des entrées/sorties décrite au chapitre 16, et les couches supérieures du système, repose essentiellement sur deux aspects.

La diversité des caractéristiques physiques des périphériques de même nature, par exemple les disques, conduit à en faire abstraction pour définir une interface normalisée, sur laquelle s'appuient les programmes du système de gestion de fichiers.

D'autre part, dans un système simple, l'utilisateur dialogue avec les applications via le clavier et l'écran physique de l'ordinateur. Il est commode de donner aux applications l'illusion qu'il en est encore de même lorsque l'utilisateur est par exemple connecté à distance depuis un autre ordinateur, via le réseau. On peut avoir recours à des pilotes de périphériques *virtuels*, qui simulent l'existence d'un périphérique fictif du type escompté par les applications. Le pilote d'écran virtuel se contentera de retransmettre les requêtes à l'ordinateur distant où elles seront traitées par le pilote de clavier et d'écran local. Le principe est le même avec les systèmes de fenêtrage.

Après avoir présenté la structure d'un pilote de périphérique (paragraphe 1.), nous montrons par l'exemple comment définir une couche pilote de périphérique. Nous étudions en détail un pilote de périphérique de type caractère : le clavier (paragraphe 2.) ; un pilote de périphérique de type bloc : le disque (paragraphe 3.). Les fonctions introduites dans le pilote de disque sont réutilisées au chapitre 19. Le paragraphe 4. évoque

la complexité des périphériques actuels, et décrit brièvement un pilote d'écran graphique.

1. Structure d'un pilote de périphérique

1.1 Interface d'un pilote

Un pilote est constitué de structures de données et d'un ensemble de procédures ou fonctions qui sont autant de services utilisables par le système d'exploitation ou éventuellement les programmes d'application. Les structures de données décrivent les caractéristiques du périphérique et de son coupleur, son état et les variables internes du pilote. L'interface comporte en général les fonctions suivantes :

- Une procédure de lecture et/ou d'écriture, selon la nature du périphérique, d'une suite d'octets.
- Des procédures dites d'ouverture et de fermeture, appelées respectivement avant et après une suite d'accès en lecture ou en écriture. Par exemple, on ouvre un fichier avant d'en lire ou d'en modifier le contenu ; on doit démarrer et arrêter le moteur du lecteur de disquette.
- Une procédure d'initialisation utilisée lors du démarrage, suite à la mise sous tension, ou lors de la réinitialisation du système.
- Une fonction de contrôle permettant de consulter ou de modifier les paramètres de fonctionnement du pilote et du périphérique.
- un ensemble de routines particulières appelées traitants d'interruption que nous étudions aux chapitres 22 et 24.

1.2 Identification des périphériques et de leur pilote

Les adresses des routines des pilotes sont regroupées dans une table de branchement à deux dimensions, indicée d'une part par le type de périphérique et d'autre part par la fonction demandée.

Il peut exister plusieurs exemplaires d'un même type de périphérique, repérés par un numéro d'exemplaire et gérés par le même pilote. Dans ce cas la structure de données du pilote devient un tableau à autant d'entrées que d'unités connectables à l'ordinateur.

Chaque périphérique peut, par exemple, être identifié par son numéro de type et son numéro d'exemplaire, appelés numéros de périphériques respectivement majeur et mineur dans la terminologie du système UNIX. Les exemplaires de disques de même type peuvent par exemple différer par leur taille (2,4 ou 9 Go). Des périphériques de même nature peuvent avoir des caractéristiques suffisamment différentes pour être considérés comme des types différents et gérés par des pilotes distincts. On pourra par exemple trouver un pilote de disques à la norme de raccordement IDE et un pilote de disques de type SCSI.

2. Pilote pour un clavier

Le clavier est un périphérique de dialogue. Les échanges entre le clavier et l'ordinateur sont typiquement caractères par caractères.

2.1 Description d'un clavier et de son coupleur

2.1.1 Vision externe

Un clavier est un ensemble de touches munies de cabochons indiquant la fonction (caractère) associée à la touche. Chaque touche est un bouton poussoir dont le contact est fermé lorsque la touche est appuyée et ouvert lorsque la touche est relâchée.

Nous supposons pour simplifer l'exposé que le contact est exempt de rebond à l'ouverture et à la fermeture. Le lecteur est invité à consulter [AL78, Zak80] pour une présentation plus détaillée des techniques d'interfaçage des claviers.

Chaque touche est repérée par un numéro indiquant sa position physique dans le clavier. Chaque touche peut avoir plusieurs sens selon l'état de diverses touches modificatrices (majuscule, contrôle, etc) au moment où elle est enfoncée.

Nous considérons à titre d'exemple un clavier de 64 touches ordinaires plus une touche de majuscule.

2.1.2 Interface du clavier physique

Nous supposons que la lecture du coupleur de clavier retourne une structure formée des champs suivants : un booléen de présence indiquant si une touche est enfoncée au moment de la lecture, un ou plusieurs booléens indiquant l'état des touches modificatrices, un entier donnant la position physique de la touche enfoncée.

Il est souvent commode de considérer la juxtaposition de la position de la touche et de l'état des modificateurs comme un numéro de touche global dans un clavier virtuel dont chaque touche ne serait associée qu'à un seul caractère. A chaque touche physique ordinaire correspondent autant de touches virtuelles que de combinaisons possibles d'état des modificateurs au moment où la touche est enfoncée. Le clavier de notre exemple possède 128 touches virtuelles : 64 touches $\times$ 2 états possibles de la touche majuscule.

2.1.3 Vision interne

En pratique, les claviers sont organisés sous forme matricielle pour obtenir une réalisation plus économique et plus compacte (moins de fils). Chaque touche ordinaire est placée à l'intersection d'une ligne et d'une colonne de la matrice. Pour tester l'état d'une touche, il suffit d'envoyer un 0 sur sa ligne : si la touche est appuyée, le 0 apparaîtra sur sa colonne. Le coupleur devient

```
TOUCHEPRESENTE : l'entier 0x80 {  Bit 7 du coupleur  }
BITSTOUCHE : l'entier 0x7F { pour récupérer le numéro de la touche  }
ADRCLAVIER : l'entier ... { adresse du coupleur clavier }

ToucheAppuyée :  ⟶ un entier
   { Retourne un entier < 0 si aucune touche n'est enfoncée ; retourne un
   entier ≥ 0 dans le cas contraire, et c'est le code émis par la lecture du
   coupleur de clavier }
   c, t : des caractères
   c ⟵ 1 Mem [ADRCLAVIER]
   t affect c ET BITSTOUCHE { ET bit à bit pour masquage }
   si c ET TOUCHEPRESENTE = 0
      { aucune touche appuyée }
      t ⟵ -1
   ToucheAppuyée : t
```

FIG. 17.1 – Fonction de détection de touche appuyée

alors un petit circuit séquentiel qui balaie les lignes à tour de rôle et mémorise la première touche appuyée rencontrée.

Dans la suite, pour fixer les idées, nous considérons que la lecture du coupleur de clavier retourne la position de la touche sur les bits 0 à 5 du bus de données, l'état de la touche majuscule sur le bit 6 et la présence d'une touche, en bit 7.

Nous définissons ainsi la fonction ToucheAppuyée qui donne le numéro de touche appuyée par l'utilisateur (Figure 17.1). Attendre l'enfoncement d'une touche t s'écrira :

répéter t ⟵ ToucheAppuyée() jusqu'à t ≥ 0

De même, attendre le relâchement de la touche t s'écrira :

répéter tt ⟵ ToucheAppuyée() jusqu'à tt ≠ t

2.2 Fonctionnalités du pilote de clavier

2.2.1 Traduction des positions en codes ASCII

Le pilote de clavier pourrait se limiter à une simple fonction retournant le code lu sur le coupleur. Toutefois, à l'exception de certains jeux, les applications ne s'intéressent généralement pas à la position physique de la touche, mais au caractère qui lui est associé. Or la disposition des caractères sur les touches dépend de la langue à laquelle le clavier est destiné.

A titre d'exemple, les types de clavier alphanumériques sont souvent définis par les six premiers caractères de la deuxième rangée : QWERTY (version anglo-saxonne), AZERTY (version francisée), etc.

Une première fonction du pilote est donc de convertir le numéro de touche

en code ASCII en fonction de la topologie du clavier, que l'on peut par exemple décrire par un tableau (indicé par le numéro global de touche) :

NumVersAscii : un entier ⟶ un caractère
{ *t étant le numéro rendu par la lecture du coupleur de clavier, accompagné de l'information concernant la touche modificatrice, NumVersAscii (t) est le caractère associé à t* }

La correspondance n'est pas biunivoque : certaines touches retournent le même caractère quels que soient les modificateurs, comme par exemple la barre d'espace, la touche de retour/fin de ligne ou la touche de suppression de caractère.

2.2.2 Problèmes d'échantillonnage

Les applications souhaitent généralement récupérer un (et un seul) exemplaire du caractère par frappe de touche. Chaque frappe correspond pourtant à deux événements physiques : l'appui d'une touche et son relâchement.

La procédure d'acquisition d'un caractère ne peut donc se limiter à la seule détection d'une touche enfoncée. Elle doit également attendre le relâchement de la touche : le relâchement est le seul événement permettant d'échantillonner correctement une suite de caractères identiques. Sans cette précaution, une touche pourrait être échantillonnée plusieurs fois par frappe.

Le pilote peut éventuellement offrir une fonction de répétition automatique. Le principe consiste à mesurer le délai durant lequel la touche reste enfoncée. Chaque accroissement de ce délai d'un temps égal à la période de répétition est assimilé à un relâchement et donne lieu à la réémission du caractère. Notre pilote simplifié ne gère pas cette fonctionnalité.

2.2.3 Mode interactif et mode ligne

La primitive de base offerte par le pilote est la lecture au clavier d'une suite de caractères, les paramètres passés par l'application étant le nombre n de caractères attendus et l'adresse t du tableau de caractères à remplir.

Le pilote peut se contenter de lire les n caractères et de les passer à l'application sans traitement particulier. Ce mode de fonctionnement est adapté aux applications interactives telles que les éditeurs de texte, qui gèrent elles-mêmes la mise à jour de l'écran et réagissent à chaque frappe de caractère (n étant le plus souvent égal à 1). Nous dirons dans ce cas que le pilote fonctionne en mode *interactif*.

Toutefois de nombreuses applications ne lisent pas des caractères isolés, mais des lignes. Une ligne est une suite de caractères terminée par un caractère de fin de ligne. Le mode ligne est par exemple bien adapté au fonctionnement d'un interprète de commande textuel simple (Cf. Chapitre 20). Il est même imposé par certains systèmes de gestion transactionnelle dotés de nombreux

terminaux distants. Dans ce contexte, il est important de minimiser le traffic sur les lignes entre les terminaux et l'ordinateur central ainsi que de décharger ce dernier des tâches subalternes. La saisie et l'édition des lignes sont alors gérées en local par le terminal qui ne s'adresse au système central que lorsqu'il dispose d'une ligne complète. Ceci revient à intégrer une partie du pilote dans le matériel du terminal.

L'application ne connaît généralement pas à l'avance la longueur de la ligne qui est retournée par la routine de lecture du pilote. Pendant l'acquisition d'une ligne, les caractères saisis sont affichés en écho à l'écran et déposés dans un tampon de ligne. La routine de lecture retourne un résultat lorsque le tampon contient un caractère de fin de ligne. Durant la saisie, l'utilisateur peut effectuer diverses corrections, telles que supprimer le dernier caractère de la ligne en appuyant sur la touche d'effacement.

Lors de l'appel de la routine de lecture du pilote, le tampon de ligne peut contenir une chaîne de ℓ caractères terminée par une fin de ligne, ou être vide. Dans ce dernier cas, le pilote attend les caractères saisis par l'utilisateur et les recopie dans le tampon, jusqu'à ce que ce dernier contienne une fin de ligne.

Le paramètre n permet de limiter le nombre de caractères transférés par appel du pilote. Si $n \geq \ell$, la ligne est transférée en entier et le tampon de ligne est vidé. Si $n<\ell$, seuls les n premiers caractères de la ligne sont consommés et retirés du tampon de ligne. Le reste de la ligne sera consommé lors d'appels ultérieurs du pilote.

Il existe une taille de ligne maximale, de l'ordre de la centaine de caractères. Lorsque le tampon de ligne ne contient plus qu'une case libre, le pilote refuse tous les caractères excepté la fin de ligne. L'écho ignore les caractères refusés et les remplace par le pseudo-caractère "sonnerie" que le pilote d'écran traduit en un signal sonore ou en un bref clignotement de l'écran pour avertir l'utilisateur du problème.

Le paramètre n est généralement égal à la taille maximale de la ligne, ce qui garantit aux applications de lire une ligne complète à chaque appel du pilote.

2.2.4 Mode avec ou sans echo

Par défaut, la ligne en cours de saisie apparaît à l'écran, ce qui permet à l'utilisateur de détecter et de corriger d'éventuelles fautes de frappe. Il existe cependant des cas de figures justifiant la saisie d'une ligne sans écho à l'écran pour éviter qu'une personne indiscrète ou indélicate ne lise la saisie par dessus l'épaule de l'utilisateur. L'exemple typique de cette situation est la saisie d'un mot de passe.

2.3 Programmation du pilote de clavier

Le fonctionnement de notre pilote est régi par deux variables booléennes : mode ligne ou interactif, et mode avec ou sans écho à l'écran.

```
{ Données du pilote }
ModeLigne : un booléen
ModeEcho : un booléen
MaxLigne : un entier > 0
Ligne : un tableau sur 0..MaxLigne − 1 de caractères { le tampon }
tailleligne : un entier ≥ 0
DebLigne : un entier sur 0..MaxLigne − 1 { pointeur }

{ Initialisation du pilote de clavier }
InitClavier : une action
    ModeLigne ⟵ vrai
    ModeEcho ⟵ vrai
    tailleligne ⟵ 0
```

FIG. 17.2 – Programmation du pilote de clavier - Variables et initialisation

La fonction de contrôle du pilote permet de fixer le mode de fonctionnement et de consulter la taille maximale de ligne.

On pourrait également prévoir la possibilité d'accéder au tableau de correspondance entre numéro de touche et code ASCII du caractère associé, pour changer la signification de certaines touches ; par exemple pour réaffecter des touches de fonctions qui n'ont pas de signification prédéfinie à des caractères accentués manquant sur le clavier.

Les programmes sont donnés Figures 17.2, 17.3 et 17.4. Outre les variables globales du pilote. la fonction de lecture d'une ligne fait appel à l'écriture d'un caractère à l'écran lorsqu'elle doit appliquer l'écho.

3. Pilote pour un disque

Un disque est un périphérique de stockage de type bloc, c'est-à-dire que les échanges se font par ensembles de plusieurs octets : les *secteurs*.

3.1 Types de supports magnétiques

Les périphériques de stockage magnétique utilisent un support dont la surface est enduite d'une fine pellicule de matériau magnétisable, qui défile sous un électroaimant : la tête de lecture/écriture. La trajectoire de cette tête par rapport à la surface du support est appelée *piste magnétique*.

L'information est transférée en série, bit après bit, pendant que la piste défile à vitesse constante sous la tête. Elle est représentée sur le support par une succession d'inversions de polarité du champ magnétique, que l'électroaimant détecte (lecture) ou impose (écriture).

Les disques durs, comme leur nom l'indique, utilisent un plateau circulaire

```
{ Quelques types et constantes }
Fonction : le type (MODELIGNE, MODEECHO, TAILLEMAX)
Auxiliaire : le type entier
CompteRendu : le type entier
FONCTION_INCONNUE : le CompteRendu -1
PARAM_INCORRECT : le CompteRendu -2
OK : le CompteRendu 0

ContrôleClavier : une Fonction, un Auxiliaire ⟶ un CompteRendu
    { ContrôleClavier (f, a) permet de fixer des paramètres du pilote ou de les
    interroger, selon la valeur du paramètre f. Elle fournit un compte-rendu,
    qui est soit un code d'erreur (valeur négative), soit une valeur demandée
    lorsqu'on l'utilise pour interroger les données internes du pilote. }
    lexique
        code : un CompteRendu
    ContrôleClavier (f, a) :
        code ⟵ OK
        selon f :
            f = MODELIGNE :
                selon a :
                    a = 0 : ModeLigne ⟵ faux
                    a = 1 : ModeLigne ⟵ vrai
                    sinon : code ⟵ PARAM_INCORRECT
            f = MODEECHO :
                selon a :
                    a = 0 : ModeEcho ⟵ faux
                    a = 1 : ModeEcho ⟵ vrai
                    sinon : code ⟵ PARAM_INCORRECT
            f = TAILLEMAX :
                code ⟵ MaxLigne
            sinon : code ⟵ FONCTION_INCONNUE


EcrireEcran : l'action (la donnée : un caractère)
{ affichage à l'écran, à la position courante du curseur, du caractère donné en
paramètre }
```

FIG. 17.3 – Programmation du pilote de clavier - Modification et consultation de l'état du pilote de clavier : écriture à l'écran.

```
LectureClavier : une action ( la donnée MaxCar : un entier > 0,
        le résultat Chaîne : un tableau sur [0..MaxCar-1] de caractères,
        le résultat NbCar : un entier)
{ MaxCar est le nombre de caractères à lire, Chaîne est le tableau à remplir et
NbCar est le nombre de caractères effectivement lus }
lexique : c : un caractère ; t, tt : des entiers ; termine : un booléen
algorithme
  si n ≤ 0 alors NbCar ⟵ −1
  sinon
    si non ModeLigne { Lecture des MaxCar caractères demandés }
      i parcourant 0..MaxCar − 1
        répéter t ⟵ ToucheAppuyée() jusqu'à t ≥ 0
        c ⟵ NumVersAscii (t) { Conversion en caractère }
        si ModeEcho alors EcrireEcran (c)
        répéter tt ⟵ ToucheAppuyée() jusqu'à tt ≠ t
    sinon { Mode ligne }
      si tailleligne ≠ 0
        { tampon non vide. on consomme la ligne à partir du début. }
        NbCar ⟵ Min (TailleLigne, MaxCar)
        i parcourant 0 .. NbCar − 1 : Chaîne[i] ⟵ Ligne[DebLigne+i]
        { le reste sera consommé lors du prochain appel }
        DebLigne ⟵ DebLigne + NbCar ; TailleLigne ⟵ TailleLigne − NbCar
      sinon { tailleligne = 0, tampon vide - Saisie d'une ligne }
      DebLigne ⟵ 0 ; Terminé ⟵ faux
      tantque non Terminé
        répéter t ⟵ ToucheAppuyée() jusqu'à t ≥ 0
        selon t :
          t = EFFACEMENT
            { ôter le dernier caractère du tampon, s'il existe. }
            si tailleligne >0
              tailleligne ⟵ tailleligne − 1 ; NbCar ⟵ NbCar - 1
              si ModeEcho : EcrireEcran (t)
          t = FINENTREE
            si ModeEcho : EcrireEcran (t)
            Ligne[tailleligne] ⟵ NumVersAscii(t)
            tailleligne ⟵ tailleligne + 1 ; NbCar ⟵ NbCar + 1
            Terminé ⟵ vrai
          sinon { garder une place pour la fin de ligne }
            si tailleligne ≥ MaxLigne − 1 alors EcrireEcran (SONNERIE)
            sinon
              si ModeEcho alors EcrireEcran (NumVersAscii(t))
              Ligne[tailleligne] ⟵ NumVersAscii(t)
              tailleligne ⟵ tailleligne + 1 ; NbCar ⟵ NbCar + 1
        répéter tt ⟵ ToucheAppuyée() jusqu'à tt ≠ t
```

FIG. 17.4 – Programmation du pilote de clavier - Fonction de lecture d'une ligne

rigide animé d'un mouvement de rotation uniforme. La tête mobile flotte à la surface du disque (l'altitude de vol de la tête est de l'ordre du micron) et se déplace radialement pour accéder aux différentes pistes, circulaires et concentriques.

Les disquettes sont un support mince et souple. Pour éviter une usure inutile, les têtes ne sont plaquées sur la surface du média magnétique que durant les accès à la disquette. Après quelques secondes d'inactivité, les têtes sont écartées du support et la rotation de ce dernier est stoppée.

Il existe un ordre de grandeur de différence entre les performances (débit et temps d'accès) des disquettes et celles des disques durs, l'écart étant encore plus important en ce qui concerne la capacité.

3.2 Description d'un disque dur

Un disque dur comporte des *pistes* circulaires et concentriques. Les pistes sont découpées en arcs de cercles appelés *secteurs*.

3.2.1 Notion d'unité de transfert et secteurs

Le secteur correspond à l'unité de transfert entre le disque et la mémoire, de même que l'octet est généralement l'unité d'échange entre la mémoire et le processeur.

Les secteurs sont généralement de 256 ou 512 octets ; il existe également des formats de disquettes avec des secteurs de 128 octets.

Les données stockées le long d'une piste sont séparées par des intervalles permettant d'absorber les petites fluctuations de vitesse de rotation.

L'accès individuel aux octets sur le disque consommerait trop de place pour les intervalles de séparation. Considérons à titre d'illustration une très faible variation de vitesse de rotation (0,1%) et de très courte durée (1% de la durée d'un tour) du disque de 18 Go dont les caractéristiques sont détaillées au paragraphe 3.2.4. Un centième de tour correspond à 10 Kbits et une fluctuation de 0,1% représente 10 bits, soit 1,25 octet. Les intervalles entre 2 octets pour absorber une telle fluctuation représenteraient déjà plus de la moitié de la longueur des pistes.

La modification d'un octet d'un secteur directement sur le disque n'étant pas réaliste, les données sont lues et écrites par secteurs complets : le secteur est lu en entier, l'octet est modifié dans la copie du secteur en mémoire, et le secteur est réécrit sur le disque.

3.2.2 Nombre de secteurs par piste

Nous supposons pour simplifier l'exposé que le nombre de secteurs par piste est constant. Ceci signifie que la fréquence de transfert des informations à la tête de lecture/écriture est constante et que la densité d'enregistrement

maximale autorisée par le support n'est atteinte que pour la piste intérieure (la plus courte).

La fréquence de transfert pourrait être adaptée à la longueur des pistes pour exploiter au mieux le support, les pistes les plus externes ayant plus de secteurs. Le prix à payer est une électronique de lecture/écriture plus sophistiquée et une légère complication des algorithmes de localisation des données sur le disque, le numéro de piste ne pouvant plus être obtenu par simple division.

3.2.3 Plateaux, cylindres et temps d'accès

Pour augmenter la capacité des disques de manière économique, on utilise les deux faces des plateaux. On monte également un ensemble de d plateaux sur le même axe de rotation. Les $2d$ têtes sont portées par un bras unique, elles se déplacent solidairement, et se partagent à tour de rôle l'unique électronique de lecture/écriture : on ne peut accéder qu'à une face de plateau à la fois.

L'ensemble des pistes accessibles dans une position donnée du bras portant les têtes est appelé *cylindre*. Un cylindre contient $2d$ pistes (une par tête). Le temps d'accès piste à piste est le délai nécessaire pour déplacer les têtes d'un cylindre à un cylindre adjacent. Le temps d'accès piste est le délai nécessaire pour amener les têtes à un cylindre donné. On peut en définir la valeur maximale (trajet entre les 2 cylindres extrêmes) et une valeur moyenne en supposant une répartition équiprobable des cylindres de départ et d'arrivée sur le disque.

Le temps d'accès secteur est le temps de rotation nécessaire pour amener le secteur voulu sous la tête. Sa borne supérieure est la durée d'une rotation complète et la moyenne le temps d'un demi-tour.

3.2.4 Caractéristiques des disques durs et performances

L'évolution technologique des disques améliore trois caractéristiques :
- la densité linéaire d'enregistrement le long des pistes, d'où une augmentation de la capacité de stockage (par piste),
- la densité radiale d'enregistrement (autrement dit l'écart entre deux pistes), ce qui à capacité égale réduit l'encombrement du disque et le débattement des têtes et donc le temps d'accès aux pistes,
- la fréquence de fonctionnement de l'ensemble tête et électronique de lecture/écriture, ce qui permet d'augmenter le débit et la vitesse de rotation, par réduction du temps d'accès secteur.

Le diamètre courant des disques est successivement passé de huit pouces à cinq pouce un quart puis à trois pouces et demi (standard actuel, soit environ neuf centimètres). L'étape suivante la plus probable est deux pouces et demi.

Voici les principales caractéristiques d'un disque dur de 1998 : six plateaux de trois pouces et demi tournant à 10000 tours/minute, 6996 cylindres (pistes par plateau), 35566480 secteurs de 512 octets chacun, soit une capacité totale de 18 Go, une fréquence de transfert de 152 à 211 Mbits/s, soit environ 1Mbit/tr, un temps d'accès secteur de 2,99 ms, et un temps d'accès piste à piste de 0,9 ms (temps accès piste moyen = 6 ms, maximal = 13 ms).

3.3 Structure des informations sur un disque dur

Nous supposons par convention que le cylindre de numéro 0 correspond à la piste la plus externe. Dans la gestion du mouvement des têtes, nous assimilerons piste et cylindre.

3.3.1 Structure d'une piste

Une piste a un début. Le rayon marquant le début d'une piste est repéré par un index. Dans le cas des disquettes, cet index est un simple trou dans le support, détecté par une fourche optoélectronique.

Le contenu d'une piste est une suite d'autant d'enregistrements que de secteurs. Chaque enregistrement se compose d'une en-tête et d'un bloc de données (le contenu du secteur). Lors d'une écriture, seul le bloc de données est écrit sur le disque, l'en-tête étant utilisée pour repérer le début du secteur. L'écriture des en-têtes est effectuée une fois pour toutes lors d'une opération d'initialisation appelée *formatage physique* du disque.

Une en-tête contiendra vraisemblablement le numéro de piste et le numéro de secteur dans la piste. On pourrait en principe se contenter d'une simple marque de début d'enregistrement. La piste peut être connue en comptant les déplacements de la tête depuis la piste 0, mais le mécanisme qui déplace la tête doit être parfaitement précis et fiable. De même, le numéro de secteur pourrait être déterminé en comptant les débuts d'enregistrement depuis le début de piste.

3.3.2 Protection contre les erreurs de lecture/écriture

Les informations stockées sont munies d'octets dits de CRC qui permettent de détecter d'éventuelles erreurs de recopie des données. Cette technique de détection d'erreur, qui ne permet pas de les corriger, est appelée contrôle de redondance cyclique.

Les intervalles avant et après les données sont remplis par des motifs binaires prédéfinis tels qu'une suite de bits à 1 (sur laquelle l'enregistrement précédent peut déborder légèrement) suivie d'une séquence plus courte de bits à 0 qui permet de détecter la fin de l'intervalle.

La valeur de CRC est calculée à partir de celle des données transférées. Elle est ajoutée aux données lors de l'écriture et, lors d'une lecture, comparée aux octets de CRC présents sur le disque. Toute différence indique bien entendu une erreur.

3.4 Fonctionnalités du pilote de disque

3.4.1 Interface entre le disque et le contrôleur de disque

Les principaux signaux de commande envoyés au disque par le contrôleur de disque sont les suivants : 1) sens du transfert des données : lecture ou écriture,

si l'électronique de pilotage de la tête ne fait pas partie du contrôleur ; 2) signal donnée à écrire et signal d'échantillonnage ; 3)sélection de la tête de lecture/écriture à utiliser : face inférieure ou supérieure du plateau et numéro de plateau ; 4) déplacement de la tête d'une piste ; 5) sens du déplacement de la tête ; 6) remise à 0.

Les principaux signaux reçus par le contrôleur sont : 1) signal donnée lue et signal d'échantillonnage ; 2) signal de présence en piste 0 ; 3) index/début de piste ; 4) disque prêt ; 5) erreur, comme par exemple, déplacement de tête au-delà des pistes extrêmes.

Pour une disquette, on trouve de plus une commande de rotation du moteur et de chargement/déchargement des têtes ainsi qu'un signal d'entrée indiquant une éventuelle protection contre l'écriture.

L'initialisation à la mise sous tension consiste essentiellement à envoyer un signal de remise à 0 à l'unité de disque, attendre que la vitesse de rotation soit stabilisée (signal prêt du disque), à ramener la tête sur la piste 0 et à remettre à 0 la variable piste courante du pilote.

Les paramètres décrivant le disque (nombre de plateaux, taille d'un secteur, nombre de secteurs par piste, nombre de pistes) sont également initialisés à partir de l'EEPROM décrivant la configuration. La taille de bloc sera initialisée à sa valeur par défaut (par exemple 1 secteur).

3.4.2 Interface du pilote de disque vers les couches supérieures

Vu du système de gestion de fichiers (SGF), le disque présenté par le pilote est une suite de blocs numérotés ; c'est une structure linéaire. Un bloc n'est pas nécessairement réduit au secteur physique. En général, un bloc est une suite de secteurs, et le nombre de secteurs par bloc est une puissance de 2.

Nous appelons *adresse physique* le numéro de bloc. Le nombre de secteurs (≥ 1) par bloc est une information du pilote.

Vu de la couche supérieure, il est indifférent que l'unité d'accès (le bloc) offerte par le pilote de disque soit effectivement un secteur. Il suffit de connaître la taille du bloc en nombre d'octets.

Les procédures de lecture et d'écriture du pilote permettent au SGF de transférer une suite de blocs d'adresses physiques consécutives entre le disque et un tableau ou tampon en mémoire.

3.4.3 Correspondance entre les adresses physiques de blocs et les numéros de secteurs et de pistes

Les procédures de lecture et d'écriture de bloc offertes aux couches supérieures font appel aux procédures de lecture et d'écriture d'un secteur fournies par le contrôleur de disque.

La première étape consiste à convertir le numéro de bloc en numéro global S de secteur. Il suffit pour cela de le multiplier par la taille T d'un bloc exprimée en nombre de secteurs.

Le numéro global de secteur doit alors être décomposé en un numéro de piste (ou plus exactement de cylindre), un numéro de tête et un numéro de secteur dans la piste sélectionnée. Le pilote contrôle au passage que le numéro global de secteur appartient à l'intervalle légal de numéros de secteurs correspondant à la capacité du disque.

En supposant que toutes les pistes ont le même nombre de secteurs, il suffit de diviser S par le nombre de secteurs par cylindre. Le quotient de la division donne le numéro de piste. Le reste est à son tour divisé par le nombre de secteurs par piste. Le quotient donne le numéro de tête et le reste, le numéro local de secteur dans la piste.

Cette organisation réduit le déplacement des têtes et les temps d'accès en groupant les secteurs et les blocs de numéros consécutifs sur le même cylindre ou sur des cylindres adjacents.

Le problème revient alors à effectuer une suite de copies entre le secteur de numéro global $S + i$ et le tampon d'adresse A à l'adresse A + i*TailleSecteur avec $0 \leq i \leq T$.

3.5 Programmation des fonctions du pilote

Les coupleurs ont évolué avec les générations de circuits et intégré une part croissante de la gestion des disques. Les coupleurs rudimentaires sont devenus des contrôleurs de disques qui déchargent le processeur de l'essentiel du travail (formatage physique des disques, sérialisation des octets de données, calcul de CRC et accès aux pistes et aux secteurs). Ce sont des circuits complexes dont la documentation technique est souvent aussi volumineuse (plusieurs dizaines de pages) que celle des processeurs qui les utilisent.

Il est donc hors de question de présenter ici un contrôleur de disque : la description de la norme de raccordement SCSI peut occuper à elle seule un livre entier.

Nous nous contenterons donc de donner les grandes lignes de l'algorithme de lecture et d'écriture d'un secteur, sans préciser la répartition des rôles entre le logiciel du pilote et le matériel du contrôleur.

3.5.1 Communication avec le coupleur

On peut s'attendre en pratique à ce que le pilote se contente de transmettre au contrôleur la nature de l'opération à réaliser : formatage, accès à une piste, retour à la piste 0, lecture ou écriture d'un secteur et les paramètres correspondants, d'attendre la fin de l'opération en testant le registre d'état du contrôleur et de gérer la reprise du processus en cas d'erreur.

Dans le cas général, le secteur appartient à une piste différente de la piste courante survolée par la tête. La différence entre les deux pistes est calculée et convertie en autant d'impulsions de déplacement du signal déplacement piste dans la direction correspondante. La fréquence des impulsions est fonction du

temps d'accès piste du disque. La variable piste courante est mise à jour. La tête est ensuite sélectionnée en lecture pour consulter les en-têtes d'enregistrement.

Le numéro de piste éventuellement contenu dans la première en-tête passant sous la tête après le déplacement est comparé avec la valeur de la variable piste courante. Un désaccord indique une erreur de calibrage du compteur de piste. Le remède consiste à ramener la tête sur la piste 0, à remettre à 0 la variable piste courante et à recommencer le processus depuis le début.

L'étape suivante consiste à attendre le passage des en-têtes qui défilent sous la tête et d'en comparer le numéro de secteur avec celui du secteur recherché. Si cette information est absente de l'en-tête, il suffit d'attendre le passage du début de piste et de compter les en-têtes à partir de celui-ci.

La détection de la bonne en-tête précède immédiatement le passage du bloc de données du secteur sous la tête. La tête est commutée en écriture si nécessaire, et le transfert commence. Les données sont transférées bit par bit à la cadence imposée par la rotation du disque.

La sérialisation de chaque octet est effectuée par un registre à décalage du contrôleur de disque, le processeur se contentant de déposer ou de récupérer l'octet dans le registre de données du contrôleur.

La valeur de CRC est calculée pendant le transfert, une erreur pouvant être éventuellement détectée. L'écriture d'un secteur peut être suivie d'une relecture de vérification au tour de piste suivant.

En lecture, l'erreur éventuelle peut être due à une petite erreur de positionnement de la tête ou dans le cas d'une disquette à une poussière sur le média. L'erreur peut être transitoire et corrigée en déplaçant la tête puis en la ramenant à nouveau sur la piste pour une nouvelle tentative d'accès. Au-delà d'une dizaine de tentatives infructueuses, l'erreur peut être considérée comme fatale, et la donnée irrécupérable.

Si un nouveau cycle écriture-lecture donne à nouveau une erreur, le secteur (voire toute la piste) est probablement défaillant et devra être marqué comme tel et retiré de la liste de secteurs utilisables.

Le cadencement des accès aux secteurs est défini par la rotation du disque et la fréquence de transfert des octets s'impose à l'ensemble processeur/mémoire. Si la cadence de transfert n'est pas scrupuleusement respectée, un ou plusieurs octets seront perdus et le transfert se terminera par une erreur.

L'exécution d'une boucle de transfert par le processeur peut s'avérer trop lente pour le débit du disque. Par exemple, une fréquence de transfert approximative de 160 Mbits/s représente 20 Mo/s, soit 50 ns par octet ; c'est à peu près le temps de cycle d'une mémoire centrale.

Pour augmenter le débit, on transfère les données par mots de 32 ou 64 bits plutôt qu'octet par octet moyennant les contraintes d'alignement d'adresses des tampons, à exploiter les accès en mode rafale (Cf. Chapitre 9), et à confier la boucle de transfert à une unité d'accès direct à la mémoire (Cf. Chapitre 16).

3.5.2 Fonctions offertes par le pilote

Nous avons vu au paragraphe 3.4.2 que le disque est organisé en blocs, chaque bloc étant une suite de secteurs. Un bloc est défini par un numéro et sa taille. Le pilote offre aux couches supérieures des fonctions d'accès à un bloc :

TailleBloc : l'entier ... { *Taille d'un bloc en nombre d'octets* }
NbBlocs : l'entier ... { *Nombre de blocs du disque* }
AdPhysique : un entier sur 0 .. NbBlocs − 1
Bloc : un tableau sur [0 .. TailleBloc − 1] d'octets

LireBloc : une action (NoB : une AdPhysique, Tampon : un Bloc)
{ *lecture du bloc de numéro NoB dans le tableau Tampon* }

EcrireBloc : une action (NoB : une AdPhysique, Tampon : un Bloc)
{ *Ecriture du tableau Tampon dans le bloc de numéro NoB* }

D'autre part les fonctions de contrôle du pilote pourraient être les suivantes : accès aux paramètres : taille d'un secteur, du disque, nombre de secteurs par bloc ; formatage physique du disque ; etc. Certaines fonctions sont spécifiques des unités à support amovible : marche/arrêt rotation, chargement/déchargement des têtes, détection de protection contre l'écriture, éjection du média (disquette, CDROM).

4. Pour aller plus loin...

Les périphériques ont évolué en prenant directement en charge une part croissante du travail de gestion des entrées/sorties assuré initialement par le processeur. L'interface matérielle de raccordement et la complexité de la programmation des entrées/sorties varient énormément selon le degré de sophistication du périphérique raccordé.

Sous le même nom et pour les mêmes fonctions, on trouve aujourd'hui des dispositifs mécaniques dotés d'une interface électronique rudimentaire à laquelle le processeur donne des ordres très élémentaires, tels que : déplacer la tête de l'imprimante d'un dixième de millimètre à droite, et d'autres disposant en interne de véritables petits ordinateurs de gestion capable d'interpréter des requêtes de niveau sémantique élevé, telles que : tracer un cercle et peindre l'intérieur en jaune. On parle dans ce dernier cas de *périphériques intelligents*.

Au coeur de nombreux périphériques on rencontre une puce électronique intégrant tous les ingrédients (processeur, mémoire vive et mémoire morte) d'un petit ordinateur. L'éventail de périphériques concernés est très large, des systèmes de disques jusqu'aux ensembles clavier/souris.

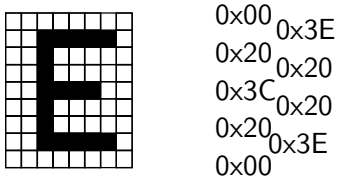


FIG. 17.5 – Représentation de la lettre "E" dans un carré de 9 par 8 pixels

4.1 Pilote pour un écran graphique

Un écran graphique est un écran cathodique constitué par une matrice de points. Dans un écran noir et blanc, à chaque point est associée une information booléenne signifiant si le point est allumé ou éteint. Pour un écran couleur, à chaque point est associé le codage de sa couleur (sur 8, 16, 24 ou 32 bits). L'ensemble des informations définissant la valeur de chaque point est stockée dans une mémoire appelée mémoire d'écran. La gestion d'une image se fait ainsi par lecture et écriture dans la mémoire d'écran. Nous avons parlé au paragraphe 4.5 du chapitre 9 de la gestion optimisée de cette mémoire.

Pour afficher un caractère, le processeur doit passer de son code ASCII à sa représentation matricielle, à recopier dans la mémoire d'écran. La figure 17.5 illustre la représentation d'un E majuscule dans un carré de neuf par huit pixels de côté.

La forme matricielle a l'inconvénient d'être volumineuse (neufs octets par caractère dans cet exemple) mais elle permet de mélanger du texte et des dessins. Lorsque l'écran n'affiche que du texte, la conversion peut être effectuée à la volée par le dispositif de rafraîchissement. Cela réduit la taille de la mémoire d'écran qui ne contient plus alors que les codes des caractères affichés, la table de conversion ASCII vers la forme matricielle étant figée dans une mémoire morte. On parle d'*écran alphanumérique*.

La connexion de l'écran est illustrée figure 17.6. La mémoire principale, la mémoire écran et les autres interfaces d'entrées/sorties sont reliées par le bus mémoire. En bas de la figure se trouvent les maîtres qui se partagent l'accès à la mémoire d'écran : le processeur et l'unité de rafraîchissement.

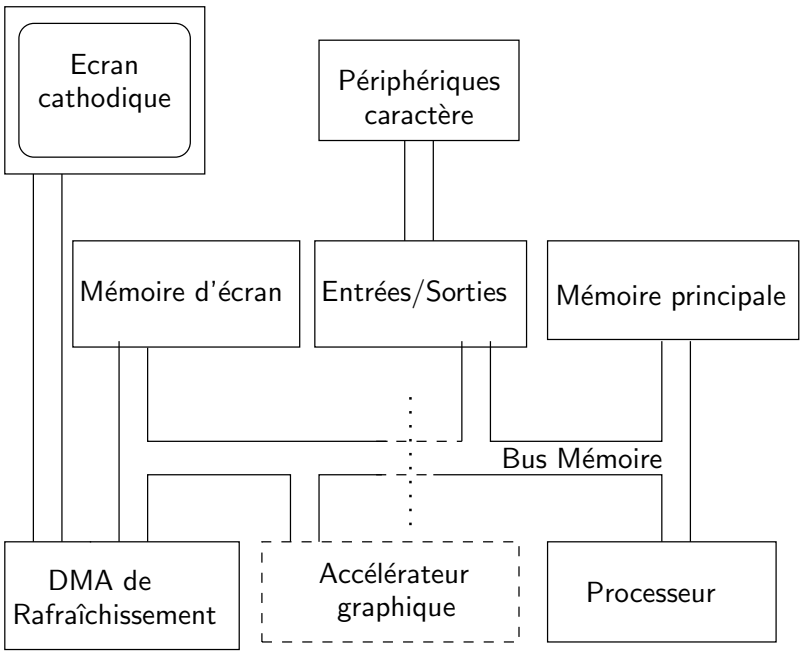


FIG. 17.6 – connexion d'un écran avec ses mémoires

Pour un affichage plus performant, les primitives grahiques les plus courantes (dont le tracé de segments de droites et de figures géométriques) peuvent être câblées (Cf. Chapitre 11) et déléguées à un circuit accélérateur. Le bus peut également être scindé en deux parties (césure en pointillé) par des connexions de type trois états, pour autoriser des accès simultanés de l'accélérateur graphique à la mémoire d'écran et du processeur à la mémoire principale ou aux autres entrées/sorties.

Le processeur principal qui génère l'information à afficher peut donc adresser de la même manière la mémoire d'écran et la mémoire principale.

A l'opposé, l'ensemble de la figure 17.6 peut constituer un terminal graphique intelligent. Le dialogue avec le terminal sera en réalité un échange entre deux ordinateurs : le système informatique central qui décide d'afficher quelque chose et l'ordinateur interne dédié du terminal. Ces deux ordinateurs se considèreront l'un l'autre comme des périphériques ordinaires de sortie et d'entrée, transférant des caractères.

Le processeur du terminal exécute une boucle infinie consistant à attendre un ordre d'affichage venant de l'ordinateur principal et à effectuer ensuite l'opération correspondante : dessiner un rectangle, allumer tel pixel, effacer tout l'écran, etc. Le processeur principal se contente alors d'envoyer les quelques octets représentant l'ordre graphique correspondant.

La syntaxe de commande des périphériques de sortie graphique peut être très élaborée : la complexité et la puissance d'expression du langage POSTSCRIPT, reconnu par de nombreuses imprimantes, sont celles des langages de programmation.

Chapitre 18

Vie des programmes

Dans ce chapitre, nous examinons toutes les questions relatives à la *vie d'un programme*, depuis sa création sous forme de fichier texte dans un langage particulier, jusqu'à son exécution par le processeur d'une machine donnée.

La notion d'exécution d'un programme recouvre deux techniques différentes, dont nous verrons qu'elles ne sont finalement que deux formes de la même approche : l'*interprétation* et la *compilation*.

En ce qui concerne l'interprétation, nous n'avons vu pour l'instant que le cas du processeur qui *interprète* le langage machine. Il s'agit d'une réalisation matérielle — câblée — de l'algorithme d'interprétation du langage machine présenté au chapitre 12. Cette idée d'écrire un algorithme pour interpréter les constructions d'un langage de programmation est utilisée par les environnements d'exécution de tous les langages dits *interprétés*, comme TCL, PERL, SCHEME, etc. Ces algorithmes d'interprétation sont alors simplement programmés au lieu d'être câblés.

En ce qui concerne la compilation, l'exécution d'un programme consiste en une interprétation, par le processeur, d'un programme en langage machine obtenu d'après le texte du programme par des étapes de traduction plus ou moins nombreuses et sophistiquées. Nous avons défini au chapitre 4 les structures principales des langages de programmation impératifs et étudié, au chapitre 13, la traduction de ce type de langage en langage d'assemblage. Ces techniques constituent le coeur des techniques de compilation des langages de haut niveau.

La notion de vie d'un programme tient compte également du fait qu'un programme est rarement définitif dès sa première écriture : les étapes d'écriture, traduction, exécution et correction peuvent être répétées de nombreuses fois, et il convient de réduire le temps nécessaire à un tel cycle en introduisant la possibilité de traiter séparément les différents fichiers qui composent un même programme. Nous avons toujours considéré jusque là qu'un programme est constitué d'un texte unique, traité de manière globale par les outils de traduction successifs. La réalité est plus complexe, et nous détaillons ici les notions de *compilation séparée* et de code translatable. Dans le paragraphe 3.,

nous présentons l'*édition de liens* et nous décrivons précisément le format d'un fichier objet translatable.

Enfin nous abordons la notion de *code translatable* et le problème de sa production systématique depuis un programme en langage d'assemblage. Il s'agit de produire un programme en langage machine sans préjuger de l'adresse absolue à laquelle le programme sera placé en mémoire vive pour exécution.

> *Dans le paragraphe 1. nous définissons précisément les termes* interprétation *et* compilation*, en donnant des exemples de langages exécutés selon l'une ou l'autre technique. Dans le paragraphe 2. nous rappelons les étapes de traduction des langages de haut niveau vers un langage machine, et nous introduisons les notions de compilation séparée, code translatable et édition de liens.*

1. Interprétation et compilation

Pour étudier les deux principes d'exécution de programmes, par *interprétation* ou par *compilation*, nous utilisons un petit langage impératif très simple. Ce langage est toutefois plus riche que les langages machine dont nous avons vu l'interprétation aux chapitres 12 et 14. Il comporte en particulier des structures itératives et conditionnelles.

1.1 L : un langage impératif simple

Le langage n'offre que le type entier relatif, d'une seule taille. Il y a trois noms de variables prédéfinis : `"X"`, `"Y"` et `"Z"`, et aucun moyen d'en déclarer de nouvelles. La portée et la durée de vie de ces 3 variables sont globales (le langage n'a pas de structure de blocs). On dispose de trois opérations binaires notées `"+"`, `"*"` et `"−"` avec le sens usuel. On peut utiliser dans les opérations des constantes entières positives.

Comme actions élémentaires, le langage dispose de primitives d'entrée/sortie à un seul paramètre, et de l'affectation. Les compositions d'actions sont la séquence, la boucle `while` et la structure conditionnelle `if-then-else`. Les conditions booléennes du `while` sont toujours de la forme *variable* $\neq$ *0*. Celles du `if` sont de la forme *variable* $>$ *0*. Les boucles ne sont pas imbriquées, les structures conditionnelles non plus. En revanche on peut trouver une boucle dans une structure conditionnelle ou vice-versa.

La séquence d'actions est implicite : chaque action occupe une ligne. On a droit à des lignes de commentaires, dont le premier mot est `rem`. Les entrées/sorties se notent : `Read ...` et `Write ...`, où les pointillés doivent être remplacés par le nom d'une des variables prédéfinies. La structure `while` comporte un marqueur de fin : le mot-clé `endwhile` tout seul sur sa ligne. De même la structure conditionnelle comporte une ligne `endif` (voir l'exemple de la figure 18.1 pour la notation des conditions. L'affectation est notée par une

```
read X
read Y
Z <-- X - Y
while Z
rem signifie :
rem tant que Z non nul
    if Z
    rem signifie Z > 0
    then
        X <-- X - Y
    else
        Y <-- Y - X
    endif
    Z <-- X - Y
endwhile
write X
```

```
! X dans l0, Y dans l1, Z dans l2
              call read ; nop
              add g0, o0, l0
              call read ; nop
              add g0, o0, l1
while :       subcc l0, l1, l2
              be endwhile ; nop
              ble else ; nop
              subcc l0, l1, l0
              ba endif ; nop
else :        subcc l1, l0, l1
endif :       ba while ; nop
endwhile :    add g0, l0, o0
              call write ; nop
```

FIG. 18.1 – (a) Exemple de programme L (b) Programme SPARC correspondant.

instruction de la forme : ... ⟵ `expr`, où les pointillés doivent être remplacés par le nom d'une des variables prédéfinies, et où l'expression `expr` est formée d'un seul opérateur, appliqué à des opérandes qui sont soit des noms de variables, soit des notations de constantes entières positives en décimal.

La figure 18.1 donne un exemple de texte du langage L et un programme en langage d'assemblage SPARC correspondant.

1.2 Exécution par compilation

Si l'on utilise la technique de compilation, on doit traduire le texte d'un programme en langage machine d'un processeur dont on dispose. Supposons que l'on compile notre langage simple vers du langage machine SPARC. Le compilateur réalise l'analyse lexicale et syntaxique du texte du programme (découpage en mots et vérification de la conformité des phrases à l'ordre imposé), puis traduit les structures de haut niveau en branchements. On obtient un programme du type décrit figure 18.1-(b). Ce programme est ensuite traduit en langage machine SPARC par l'outil d'assemblage. Le résultat est stocké dans un fichier objet (Cf. Chapitre 19) qui est ensuite chargé en mémoire vive pour exécution (Cf. Chapitre 20), et lancé, c'est-à-dire interprété directement par le processeur de la machine (Cf. Chapitre 14).

Ecrire un compilateur, c'est-à-dire le programme qui réalise les phases d'analyse et de traduction, est une tâche très bien étudiée maintenant, au moins pour les langages à structure classique comme celui que nous avons étudié au chapitre 4. On trouvera dans [CGV80, WM94] un exposé complet des techniques de compilation.

1.3 Exécution par interprétation

1.3.1 Solution de base

Nous donnons figures 18.2 et 18.3 l'algorithme d'interprétation du langage L. Pour programmer en langage L sur une machine à processeur M, il faut programmer l'algorithme d'interprétation, par exemple dans un langage de haut niveau comme ADA, puis compiler ce programme pour le langage machine du processeur M, le charger et le lancer. A l'exécution, ce programme d'interprétation travaille sur un programme du langage L, pris dans un fichier ou tapé directement au clavier par le programmeur. Pendant cette exécution, le texte de programme en langage L est traité comme une *donnée* par le programme interprète, alors qu'il est perçu comme un *programme* par l'utilisateur humain qui l'a écrit. On voit ici que la distinction entre programmes et données n'est pas intrinsèque.

Nous avons omis dans l'algorithme d'interprétation la phase de lecture du fichier texte du programme. Cette phase de lecture, dans un fichier ou au clavier, est supposée effectuée complètement avant que ne commence l'exécution. Elle fournit le programme sous la forme d'un tableau de lignes, chaque ligne étant découpée en mots (il y a au plus 5 mots sur une ligne dans la syntaxe du langage que nous étudions ; un commentaire peut être considéré comme ayant 2 mots). Les textes comportant des lignes de plus de 5 mots ont été rejetés. On suppose que le programme lu tient dans le tableau de MAXLIGNE lignes.

Par exemple, la lecture de la ligne `X <-- X - Y` du programme d'exemple donné ci-dessus fournit : Prog[10] = < "X", "<--", "X", "-", "Y" > .

Une telle phase de lecture s'apparente à la phase d'analyse lexicale et syntaxique dans un compilateur. Noter toutefois que le travail est perdu d'une exécution à l'autre. L'algorithme fourni figure 18.3 est la phase d'exécution proprement dite, par parcours du tableau de lignes.

1.3.2 Prise en compte des boucles imbriquées

Dans le langage L présenté ci-dessus, nous avons supposé que les structures itératives ne sont pas imbriquées. Cette hypothèse justifie l'algorithme très simple d'interprétation des structures itératives, pour lequel une seule adresse de début de boucle DebBoucle suffit. Si les structures itératives peuvent être imbriquées à un niveau quelconque, il faut prévoir une *pile* d'adresses de retour. D'autre part la recherche du mot-clé `"endwhile"` (lorsque la condition de boucle devient fausse) est plus compliquée. Il faut en effet parcourir les lignes du texte en comptant les `while` et en décomptant les `endwhile`.

1.3.3 Prétraitements divers

La lecture du fichier et le stockage dans un tableau des lignes découpées en mots constitue déjà un traitement préalable à l'exécution. Le découpage en mots des lignes qui constituent le corps d'une boucle est effectué une seule fois.

Un autre prétraitement intéressant consiste à associer à chaque instruction `"while"` le numéro de la ligne du `"endwhile"` correspondant. Cela évite la boucle tantque Mot1 de Prog[CP] $\neq$ "endwhile" : CP $\longleftarrow$ CP + 1 de l'algorithme d'interprétation.

On pourrait bien sûr éliminer les commentaires dès la phase de lecture du fichier, et imaginer de nombreux autres prétraitements, qui évitent de répéter du travail lors des multiples exécutions d'une ou plusieurs instructions du programme.

1.4 Définitions et exemples

1.4.1 Compilation

On appelle *compilation* un mécanisme d'exécution de programmes dans lequel les analyses lexicale, syntaxique et de typage, ainsi que la transformation du programme en un langage de plus bas niveau, sont effectuées par des prétraitements, avec résultats intermédiaires stockés dans des *fichiers persistants*. En compilation, les fichiers produits contiennent toute l'information nécessaire à l'exécution du programme d'origine. On peut exécuter un programme si l'on a perdu le fichier source, ou même si on ne l'a jamais eu.

En général on réserve aussi ce mot au cas où la forme finale produite par l'outil dit *de compilation* est un langage machine destiné à être exécuté par le processeur correspondant.

Les langages PASCAL, ADA, C, C++ sont habituellement compilés ; rien n'empêche toutefois de programmer pour ces langages des algorithmes d'interprétation.

1.4.2 Interprétation et programmation incrémentale

On appelle *interprétation* un mécanisme d'exécution de programmes dans lequel on repart du texte source à chaque exécution (et donc il ne faut surtout pas le perdre !) ; il y a éventuellement des prétraitements effectués sur le texte du programme avant exécution, et des formes intermédiaires stockées en mémoire vive pendant l'exécution.

On confond souvent le fait que le langage soit interprété avec le fait que l'environnement de programmation autorise la *programmation incrémentale*. Dans un environnement de programmation `scheme` ou `lisp`, par exemple, on ajoute des fonctions de manière interactive avant de les appeler. De même, les langages de commandes étudiés au chapitre 20 sont prévus pour la programmation incrémentale, le langage POSTSCRIPT également (l'outil d'affichage GHOSTVIEW est basé sur l'interprète GHOSTSCRIPT qui permet de programmer directement en langage POSTSCRIPT et d'observer le résultat).

La programmation incrémentale implique l'exécution par interprétation, mais l'inverse est faux.

Les langages SCHEME, LISP, TCL, POSTSCRIPT, HTML, ML, PROLOG sont

lexique

MAXLIGNE : l'entier 100
numligne : le type entier sur 1..MAXLIGNE
Texte : le type séquence de caractères
{ *On suppose l'existence d'opérations manipulant des textes, comme l'égalité notée =, la différence notée ≠, etc. De plus on note les constantes texte avec des guillemets.* }
Ligne : le type < Mot1, Mot2, Mot3, Mot4, Mot5 : des Textes >
Prog : un tableau sur [1..MAXLIGNE] de Lignes
M1, M2, M3, M4, M5 : des Textes
CP : un entier sur 1..MAXLIGNE+1
{ *Le compteur programme, c'est-à-dire le numéro de la ligne de l'instruction en cours d'interprétation.* }
DebBoucle : un entier sur 1..MAXLIGNE
vX, vY, vZ : des entiers
tmp1, tmp2 : des entiers ; cond : un booléen
N : le numéro de la dernière ligne du texte lu

ValeurDeNom : un Texte ⟶ un entier
{ *ValeurDeNom (t) est définie pour un texte t parmi les noms de variables autorisées "X", "Y" ou "Z", et donne la valeur de la variable correspondante* }
ValeurDeNom (t) :
selon t :
t = "X" : vX
t = "Y" : vY
t = "Z" : vZ

AffectParNom : une action (la donnée t : un Texte, la donnée a : un entier)
{ *pour les textes t parmi les noms de variables autorisées "X", "Y" ou "Z", AffectParNom (t, a) affecte à la variable correspondante (vX, vY ou vZ) la valeur a* }
AffectParNom (t, a) :
selon t :
t = "X" : vX ⟵ a
t = "Y" : vY ⟵ a
t = "Z" : vZ ⟵ a

ValeurDeNombre : un Texte ⟶ un entier ≥ 0
{ *ValeurDeNombre (t) est l'entier noté t en décimal.* }

FIG. 18.2 – Lexique de l'algorithme d'interprétation

```
CP ⟵ 1
tantque CP ≠ N+1
   M1 ⟵ Mot1 de Prog[CP]
   selon M1 :
      M1 = "rem" : CP ⟵ CP + 1
      M1 = "read" :
         Lire (tmp) ; AffectParNom (Mot2 de Prog[CP], tmp) ; CP ⟵ CP + 1
      M1 = "write" :
         Ecrire (ValeurDeNom (Mot2 de Prog[CP])) ; CP ⟵ CP + 1
      M1 = "X" ou M1 = "Y" ou M1 = "Z" :
         M3 ⟵ Mot3 de Prog[CP] ; M4 ⟵ Mot4 de Prog[CP]
         M5 ⟵ Mot5 de Prog[CP]
         tmp1 ⟵ selon M3
            M3 = "X" ou M3 = "Y" ou M3 = "Z" : ValeurDeNom (M3)
            sinon ValeurDeNombre (M3)
         tmp2 ⟵ selon M5
            M5 = "X" ou M5 = "Y" ou M5 = "Z" : ValeurDeNom (M5)
            sinon ValeurDeNombre (M5)
         tmp ⟵ selon M4
            M4 = "+" : tmp1 + tmp2
            M4 = "*" : tmp1 * tmp2
            M4 = "−" : tmp1 − tmp2
         AffectParNom (M1, tmp) ; CP ⟵ CP + 1
      M1 = "while" :
         tmp ⟵ ValeurDeNom (Mot2 de Prog[CP])
         cond ⟵ (tmp ≠ 0)
         si cond alors
            DebBoucle ⟵ CP ; CP ⟵ CP + 1
         sinon
            tantque Mot1 de Prog[CP] ≠ "endwhile" : CP ⟵ CP + 1
            CP ⟵ CP + 1
            { on est sur la ligne qui suit la ligne du "endwhile" }
      M1 = "endwhile" : CP ⟵ DebBoucle
      M1 = "if" :
         cond ⟵ (ValeurDeNom (Mot2 de Prog[CP])) > 0
         si cond alors CP ⟵ CP + 1
         sinon
            tantque Mot1 de Prog[CP] ≠ "else" : CP ⟵ CP + 1
            CP ⟵ CP + 1
      M1 = "then" : CP ⟵ CP + 1
      M1 = "else" :
         tantque Mot1 de Prog[CP] ≠ "endif" : CP ⟵ CP + 1
      M1 = "endif" : CP ⟵ CP + 1
      sinon : Ecrire ("Erreur : instruction inconnue :", M1)
```

FIG. 18.3 – Algorithme d'interprétation

habituellement interprétés. Pour certains d'entre eux il existe également un compilateur, qui permet d'accélérer les exécutions. C'est le cas par exemple de ML ou SCHEME.

1.4.3 Compilation dynamique

Dans le cas de l'interprétation, on peut imaginer un prétraitement qui consisterait à produire à partir du texte de programme l'équivalent en assembleur ou en langage machine. Tant que cette forme intermédiaire n'est pas stockée dans un fichier persistant, on peut considérer qu'il s'agit toujours d'un mécanisme d'interprétation. On trouve parfois le nom de *compilation dynamique* pour parler de ces situations.

1.4.4 Emulation

Nous avons vu au chapitre 12 la notion de compatibilité de familles de processeurs. Si les deux machines sont très différentes, le constructeur fournit un *émulateur* du langage machine n sur la machine $n + 1$. Un émulateur est un programme, écrit dans un langage quelconque, par exemple C, et compilé sur la nouvelle machine, avec le nouveau compilateur C. Ce programme est un interprète du langage machine n. Le code objet des anciens programmes n'est donc plus directement interprété par un processeur, mais par un programme, lui-même compilé et exécuté sur un autre processeur.

C'est le cas des MACINTOSH : APPLE fournit un émulateur de 68000 parmi les programmes du logiciel de base fourni avec les machines à POWERPC.

1.4.5 Code intermédiaire

Pour certains langages de haut niveau, il est difficile de dire si l'exécution est assurée par un mécanisme d'interprétation ou de compilation. En effet, le programme est d'abord compilé dans un code intermédiaire stocké dans un fichier, lequel est ensuite interprété. C'est le cas du PASCAL UCSD compilé en *P-code*, de PROLOG et JAVA.

Si l'on construit une machine dont le langage machine est exactement le code intermédiaire, on dispose d'un mécanisme d'exécution par compilation, au sens défini plus haut. Sinon le code intermédiaire doit être interprété par programme.

2. Compilation séparée et code translatable

Nous revenons sur le mécanisme d'exécution par compilation, pour préciser les problèmes à résoudre dans le cas réaliste où les programmes ne sont pas traités globalement par les outils de traduction. L'exposé est basé sur un

```
/* FICHIER main.c */
#include <stdio.h>
#include "fact.h"
void main ()
{
  long R ; short n ;

  printf("Donnez un entier : ") ;
  scanf ("%hd", &n) ;
  R = Fact (n) ;
  printf("Fact(%d)=%d", n, R) ;
}
```

```
/* FICHIER fact.h */
extern long Fact (short) ;
```

```
/* FICHIER fact.c */
#include "fact.h"
long Fact(short x)
{
  if (x==0)
    return 1 ;
  else
    return x * Fact (x-1) ;
}
```

FIG. 18.4 – (a) Factorielle en C

exemple très simple écrit en C ; le lecteur familier d'un langage de ce type transposera facilement le discours dans le langage de son choix.

2.1 Un exemple en C

Nous donnons figure 18.4 un exemple de programme C décomposé en trois fichiers : `main.c` qui contient le programme principal, lequel fait appel à une fonction `Fact` non définie dans ce fichier-là ; `fact.c` qui contient la définition complète de la fonction `Fact` (profil et corps) ; `fact.h` qui contient le profil de la fonction `Fact`. Ce fichier dit *d'interface* est inclus dans le fichier du programme principal, qui peut ainsi être compilé indépendamment du fichier qui contient le corps de la fonction `Fact`.

Le fichier d'interface est également inclus dans le fichier qui contient la définition complète de la fonction `Fact` ; cette redondance de définitions permet de faire vérifier au compilateur la conformité entre la version de `Fact` du fichier `fact.c` et la version publiée dans `fact.h` à l'usage d'autres fichiers utilisateurs comme `main.c`. Noter toutefois que l'inclusion de `fact.h` dans `fact.c` n'est pas obligatoire ; c'est une précaution du programmeur, pour éviter des erreurs dues au mécanisme très rudimentaire qui sert de support à la programmation modulaire en C. Un langage comme ADA offre en revanche un support complètement contrôlé.

Les paragraphes suivants détaillent la structure de l'exemple.

2.2 Notion de compilation séparée

La *compilation séparée* consiste à réaliser la compilation d'un programme, en traitant *séparément* différentes portions de ce source qui, par conséquent, peuvent même être rangées dans des fichiers différents.

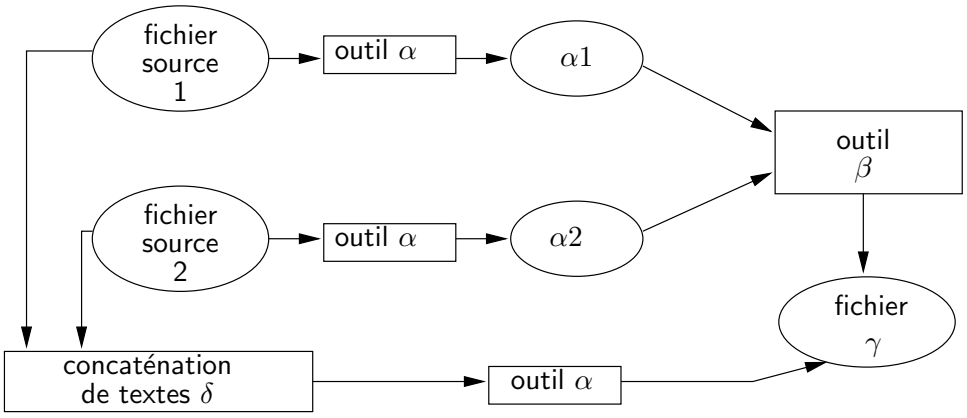


FIG. 18.5 – Schéma général de compilation séparée

La figure 18.5 donne le schéma général de la compilation séparée, dans le cas de deux fichiers source. On étend facilement au cas de n fichiers source. Si le programme est constitué de deux fichiers source 1 et 2, la compilation séparée fournit des outils α et β tels que le diagramme commute :

$$\gamma = \beta(\alpha(1), \alpha(2)) = \alpha(\delta(1, 2))$$

Autrement dit, on obtient le même résultat γ en compilant séparément les deux fichiers 1 et 2 grâce à l'outil α et en rassemblant les résultats grâce à l'outil β, qu'en compilant grâce à l'outil α le fichier obtenu par simple concaténation des fichiers source 1 et 2.

Pour comprendre ce schéma de principe et les contraintes qui portent sur la définition des outils α, β ou sur la structure du format des fichiers $\alpha 1, \alpha 2$, il faut se poser 3 questions, dont les réponses sont liées :

- Etant donné un programme donné dans un seul fichier source, comment *séparer* ce programme en plusieurs fichiers source distincts, de telle sorte qu'ils puissent être traités indépendamment l'un de l'autre par l'outil α ?
- Quel type d'information doit-on trouver dans le format des fichiers $\alpha 1, \alpha 2$, pour qu'il soit possible de définir l'outil β ?
- Que gagne-t-on à mettre en oeuvre un tel mécanisme, nécessairement plus compliqué que la compilation centralisée ?

Nous répondons ci-dessous aux trois questions, sans supposer tout de suite que le format produit par l'outil α correspond à du langage machine. Cela permet de comprendre la compilation séparée indépendamment de la production de code translatable. En effet les deux problèmes sont conceptuellement indépendants, même s'ils sont en général traités conjointement dans les environnements de programmation usuels.

L'examen des trois questions ci-dessus conduit également à étudier la notion de *portée* des noms dans un langage de programmation.

2.2.1 Séparation d'un programme et notion d'interface

La possibilité de séparation d'un programme en plusieurs fichiers dépend du langage source dans lequel les programmes sont écrits, et du format des fichiers $\alpha 1, \alpha 2$.

Si la transformation α se réduit à éliminer les commentaires, par exemple, il suffit de découper le programme sans couper les commentaires, et le schéma de compilation séparée fonctionne : β se contente de concaténer les textes $\alpha 1, \alpha 2$.

Toutefois il n'est pas intéressant de développer un tel mécanisme simplement pour réaliser l'élimination des commentaires de manière séparée (voir aussi la réponse à la question "Que gagne-ton ?").

Supposons donc que le format des fichiers $\alpha 1, \alpha 2$ soit plutôt du langage d'assemblage (ou du langage machine éventuellement assorti d'informations supplémentaires, voir réponse à la question suivante), et considérons un langage source du type de celui étudié au chapitre 4.

Il paraît peu probable que la compilation séparée soit réalisable si l'on coupe un fichier source au milieu du corps d'une fonction ou d'une procédure : le principe de génération de code pour les langages à structure de blocs étudié au chapitre 13 produit un prologue et un épilogue très symétriques pour chaque bloc, et il est donc nécessaire de disposer de ce bloc complètement en une seule fois.

On imagine facilement, en revanche, que le programme puisse être coupé entre deux procédures ou fonctions ; nous avons montré au chapitre 13, paragraphe 3.2, comment produire du code indépendamment pour les différents blocs. Il suffit ensuite de concaténer les textes en langages d'assemblage (ou les programmes en langage machine) obtenus pour les différents blocs.

En réalité la compilation d'un bloc n'est pas possible de manière complètement indépendante des autres portions. Pour générer le code d'un appel de procédure, il est nécessaire de connaître le *profil* exact de cette procédure, pour réaliser correctement le passage de paramètres en tenant compte de la représentation en mémoire des types des paramètres formels et du mode de passage des paramètres.

Toutefois il n'est pas nécessaire de connaître le *corps* de la procédure appelée. Cette distinction entre profil et corps de fonction conduit à la notion d'*interface* de portion de programme.

Sur l'exemple du calcul de la fonction factorielle donné en C ci-dessus, l'interface `fact.h` reprend la ligne de déclaration de la fonction `Fact`. Ce fichier est inclus dans le programme principal `main.c` : cela suffit pour savoir compiler l'appel de `Fact`.

Le profil d'une fonction est l'information nécessaire et suffisante à donner aux utilisateurs éventuels de cette fonction. La même distinction entre définition complète d'un objet du programme, et information réduite nécessaire aux utilisateurs, peut être étudiée pour d'autres classes d'objets dans les langages de programmation : constantes, variables, types, etc.

2.2.2 Rassemblement de codes

Supposons que l'on a produit du code séparément pour deux portions de programme source. Il faut maintenant savoir rassembler les différentes portions de code pour constituer le code du programme global.

Reprenons l'exemple du découpage du programme entre les procédures et fonctions pour un langage dans lequel les programmes sont des collections de fonctions. C'est le cas de C ; le programme *principal* n'est pas intrinsèquement différent des autres fonctions du fichier, et doit simplement s'appeler `main`. Cela signifie simplement que le choix du *point d'entrée* dans le code produit à partir du programme C se fait par convention sur un nom de fonction réservé. En Pascal, en revanche, la syntaxe du langage demande d'expliciter le programme principal, dans un bloc `begin...end` non nommé. Le code produit pour ce bloc a la même structure que celui produit pour une procédure ordinaire, et c'est le point d'entrée du programme.

La situation typique de compilation séparée correspond au cas d'une fonction définie dans le fichier source 1, et utilisée dans le fichier source 2, comme la fonction `Fact` de l'exemple en C. L'inclusion du fichier `fact.h` donnant le profil de la fonction `Fact`, dans le fichier `main.c`, a servi dans la phase α, pour générer correctement la séquence d'appel à la fonction.

Supposons que l'on a obtenu deux fichiers en langage d'assemblage qui contiennent : l'un une instruction du genre `call Fact`, l'autre une portion de code à l'étiquette `Fact :` .

Pour obtenir le code du programme global, il suffit de concaténer les deux fichiers en langage d'assemblage obtenus. Puisque ce ne sont que deux collections de fonctions, le fichier concaténé représente l'union des deux collections, et l'ordre n'a pas d'importance.

Un problème peut toutefois survenir lorsque les deux fichiers comportent chacun une définition d'une étiquette `L :`. En effet, il est incorrect, dans un texte en langage d'assemblage, de définir deux fois la même étiquette.

Ces considérations sur les *noms* des étiquettes (qui correspondent aux noms des objets du programme en langage de haut niveau) nous amènent à définir la notion de *portée* des noms dans les langages de programmation.

2.2.3 Notion de portée des noms

Le conflit de noms éventuel rencontré lors de la fusion de deux fichiers produit par l'outil α suppose que les étiquettes définies dans les fichiers $\alpha 1, \alpha 2$ sont visibles partout.

Certaines d'entre elles proviennent de noms d'objets du programme source, en particulier les étiquettes qui repèrent le début du code des fonctions et procédures. D'autres ont été créées de toutes pièces pour coder les structures conditionnelles et itératives (Cf. Chapitre 13, Figure 13.1).

Pour ces dernières, il est particulièrement ennuyeux qu'elles soient visibles partout. En effet, la compilation séparée d'un programme risque fort de pro-

duire les mêmes noms pour le codage des structures de contrôle, et ces noms seront identifiés par l'outil de fusion β.

Pour les étiquettes provenant de noms d'objets du programme source, cela peut paraître moins contraignant : il est à la charge du programmeur de ne pas définir deux fois la même fonction dans deux fichiers différents du même programme. Toutefois, si l'on imagine un programme vraiment très grand, écrit par une équipe de 50 programmeurs, il est fort probable que deux d'entre eux auront écrit une fonction `max` pour des besoins locaux. Dans un langage comme ADA, C ANSI, PASCAL, on peut cacher la définition de la fonction `max` dans une autre fonction, et le problème est réduit : pour ces fonctions *locales*, le même mécanisme de portée que pour les variables locales de procédures s'applique. Le problème subsiste pour les fonctions principales, qui ne sont incluses dans aucune autre.

Ces problèmes trouvent une solution dans la structure de *modules* des langages de programmation, implémentée de manière plus ou moins propre et contrôlée selon les langages.

L'idée de base, que l'on retrouve dans le mécanisme rudimentaire de définition de portée par fichier en C, consiste à permettre un regroupement d'objets du langage dans un *module*. A l'intérieur d'un module, des objets sont définis et localement utilisables. Pour être visibles de l'extérieur (dans les autres modules), ils doivent être exportés. Il devient donc possible de définir deux fonctions `max` dans deux modules différents, du moment qu'on ne les exporte pas. L'interface d'un module récapitule les objets définis dans ce module et utilisables à l'extérieur.

En C, la notion de module correspond à la structure de fichier. Ce n'est pas le cas en ADA par exemple, où un même fichier peut contenir plusieurs modules ou `packages`. En C, tout objet défini dans un fichier est par défaut exporté. Pour cacher sa définition à l'extérieur, il faut préfixer sa déclaration par le mot-clé `static`. En langage d'assemblage, c'est souvent l'inverse : toute étiquette définie est locale au fichier, sauf exportation explicite. Dans les langages d'assemblage que nous avons utilisés dans cet ouvrage, l'exportation explicite se fait par une directive `.global` (voir par exemple l'exemple donné Figure 12.9 du chapitre 12).

Ce mécanisme rudimentaire de masquage des noms ne saurait être qualifié de support à la programmation modulaire. Un inconvénient majeur est l'impossibilité de partager un nom entre deux fichiers (ou modules) d'un programme, sans le partager également avec tous les autres : la directive d'exportation est tous azimuts.

2.2.4 Avantages de la compilation séparée

Récapitulons les aspects étudiés ci-dessus, pour répondre à la troisième question : que gagne-t-on à mettre en oeuvre un mécanisme de compilation séparée ?

– On gagne du temps de compilation. Attention, cet argument n'est valable qu'en raison des multiples compilations que subit un même programme au cours de sa vie. Pour une compilation unique, le schéma de compilation séparée ne peut pas être plus rapide qu'un schéma de compilation global, puisqu'il gère des informations supplémentaires (voir détails ci-dessous). Il est donc plus précis de dire que l'on gagne du temps sur un ensemble de n compilations : si l'on modifie le fichier 1 sans toucher au fichier 2, il suffit de recompiler le fichier 1 et de réaliser l'étape β pour obtenir le programme exécutable à jour. Pour que l'argument tienne, il faut également que la durée d'une étape β soit très inférieure à celle d'une étape α.
– Le schéma de compilation séparée permet aussi de développer des programmes complexes qui utilisent certaines portions de programmes directement sous forme compilée. On y gagne la notion de *bibliothèque*, distribuable sous forme compilée seulement. Ainsi tout système UNIX est fourni avec de nombreuses bibliothèques de fonctions utilisables dans des programmes C, dont les fonctions d'entrées/sorties `printf` et `scanf`, ou encore les fonctions mathématiques `cos`, `sin`, etc. Une bibliothèque de fonctions est composée de deux fichiers : un fichier objet qui contient le code des procédures et fonctions, et un fichier texte dit d'interface, qui donne les profils de ces procédures et fonctions. La figure 18.6 donne un exemple obtenu par la commande `man cos` sur un système UNIX standard.
– Le schéma de compilation séparée autorise également la programmation *multilangages* : si l'on dispose, pour deux langages différents L1 et L2, des traducteurs α, on peut rassembler les fichiers produits et obtenir un programme exécutable global, dont certaines parties étaient à l'origine écrites en L1, et d'autres en L2. Cela implique toutefois de fortes contraintes sur la structure de L1 et L2, qui doivent être compilables dans le même format. Nous utilisons beaucoup dans les parties V et VI la programmation multilangages C et langage d'assemblage.
– Enfin le schéma proposé permet de définir une notion de portée des noms d'objets par fichier (même si ce n'est pas à proprement parler une bonne idée). Le diagramme de la figure 18.5 ne commute donc pas réellement : si l'on définit deux fonctions de même nom dans les fichiers source 1 et 2, le chemin par concaténation de textes relève une erreur ; en revanche, le chemin par compilation séparée ne relève pas d'erreur, à condition que les deux fonctions soient cachées dans leurs fichiers respectifs, c'est-à-dire non exportées.

2.3 Traduction des étiquettes en adresses et notion de code translatable

La notion de code translatable et son intégration dans le schéma général de compilation séparée présenté ci-dessus suppose que l'on fixe le format des

Mathematical Library cos(3M)

NAME

cos - cosine function

SYNOPSIS

```
cc [ flag ... ] file ... -lm [ library ... ]
#include <math.h>
double cos(double x) ;
```

DESCRIPTION

The `cos()` function computes the cosine of x, measured in radians.

RETURN VALUES

Upon successful completion, `cos()` returns the cosine of x.
If x is NaN or +Inf, NaN is returned.

FIG. 18.6 – Fonction `cos` de la bibliothèque mathématique. Le paragraphe SYNOPSYS indique successivement : la commande compilation et édition de liens à utiliser pour spécifier l'utilisation de la bibliothèque `-lm` ; la ligne d'inclusion du fichier d'interface, à placer dans tout fichier utilisateur ; le profil de la fonction `cos`.

fichiers produits par l'outil α. Il s'agit de langage machine, c'est-à-dire d'un format obtenu après traduction des *étiquettes* ou *symboles* du langage d'assemblage en *adresses*.

Le choix de la nature des informations présentes dans un fichier objet translatable est le résultat d'un compromis entre deux contraintes : 1) la phase α doit effectuer le maximum de travail, et la phase β le minimum, pour que l'argument de gain de temps tienne ; 2) si l'on veut pouvoir placer le programme n'importe où en mémoire, il est impossible de réaliser, dans α, la totalité des transformations qui vont du fichier source au code binaire directement exécutable.

Nous examinons le problème de la traduction des étiquettes en adresses, et la notion d'utilisation relative ou absolue d'un symbole, sur un exemple en langage d'assemblage pour processeur SPARC (Figure 18.7).

2.3.1 Utilisation relative d'un symbole

Considérons le cas de l'instruction `ba debut`. Il s'agit d'une utilisation *relative* du symbole `debut` (Cf. Chapitre 12, paragraphe 1.4.3). En ce qui concerne la traduction des symboles en adresses, cela signifie simplement que le symbole `debut` n'est utilisé, dans l'instruction `ba debut`, que pour s'abstraire d'un calcul explicite de la distance entre l'instruction de branchement et sa cible. L'assembleur compte les instructions et remplace `ba debut` par `ba -4` (on compte en nombre d'instructions).

La notion d'utilisation relative de symbole est une propriété intrinsèque

```
        .data
D :     .long 42 ! une donnée de 32 bits initialisée à la valeur 42
        .text
debut :
        sethi %hi (D), %r1      ! couple d'instructions destiné
        or %r1, %lo(D), %r1     ! à ranger la valeur sur 32 bits de
                                ! l'adresse représentée par D dans r1
        ld [%r1], %r2           ! chargement depuis la mémoire
                                ! du mot d'adresse r1
        ! ici r2 doit contenir 42.
        nop
        ba debut                ! branchement inconditionnel
        nop
```

FIG. 18.7 – Utilisation relative ou absolue d'un symbole

d'une instruction du langage machine considéré. Le processeur SPARC a deux instructions relatives : les branchements et les appels de procédures `call`. Toutes les autres instructions utilisant des symboles en font une utilisation absolue.

Le processeur 68000 a une instruction de branchement relatif, similaire à celle du SPARC. En revanche l'instruction `jsr` de saut à un sous-programme est absolue.

2.3.2 Utilisation absolue d'un symbole

Le cas des instructions `sethi %hi (D), %r1` et `or %r1, %lo(D), %r1` est plus compliqué. Toutes deux sont des utilisations *absolues* du symbole `D`. De même l'instruction `add %r1, X, %r2` est une utilisation absolue de `X`. Attention, il n'y a pas d'indirection implicite, et il s'agit bien d'ajouter au registre `r1` la valeur d'adresse associée au symbole `X` ; toutefois, cette valeur est nécessairement tronquée à 13 bits puisque c'est la taille du champ valeur immédiate dans le codage de l'instruction `add` du SPARC.

Du point de vue de la traduction des symboles en adresses, cela signifie que le symbole `D` doit être remplacé par l'adresse à laquelle se trouvera la donnée 42, en mémoire vive, lors de l'exécution du programme.

Cette adresse ne peut être connue à l'assemblage que si le chargeur (Cf. Chapitre 20) décide de toujours installer les programmes à exécuter à une adresse fixe A connue de l'assembleur ; dans ce cas, l'assembleur peut remplacer le symbole `D` par une valeur de la forme $A+d$, où d représente la position relative de la donnée `D` dans le code produit pour ce programme. Ici, en supposant que les données sont placées après les instructions, $d = 6 \times 4$ octets, puisqu'il y a 6 instructions codées sur 4 octets avant la donnée repérée par `D`.

Toutefois, dans les systèmes multitâches, le chargeur gère la mémoire de

```
A     0000001100??????????????????????sethi %hi (D), %r1
A+4   10000010000010000011??????????????or %r1, %lo(D), %r1
A+8   11000100000000000100000000000000   ld [%r1], %r2
A+12  10000000000000000000000000000000   nop (add %g0, %g0, %g0)
A+16  00010000101111111111111111111100   ba debut (ba -4)
A+20  10000000000000000000000000000000   nop (add %g0, %g0, %g0)
A+24  00000000000000000000000000101010   .long 42
```

FIG. 18.8 – Contenu de la mémoire à l'exécution

manière dispersée, et l'adresse de chargement d'un programme donné n'est connue qu'au dernier moment. Dans le programme en langage machine produit par l'outil α et stocké dans un fichier persistant, le symbole `D` n'a donc *pas encore* été remplacé par l'adresse absolue qui lui correspond, et qui d'ailleurs dépend de l'exécution.

2.3.3 Contenu de la mémoire à l'exécution

La figure 18.8 montre ce que doit être le contenu de la mémoire vive lors de l'exécution du programme de la figure 18.7. Chaque instruction est codée sur 32 bits. Les données sont supposées placées en mémoire après les instructions : on trouve en dernière ligne le codage en binaire de la valeur 42_{10}, sur 32 bits. La colonne de gauche donne les adresses, en supposant que le programme a été installé en mémoire à l'adresse `A`. L'instruction `ba debut` a été entièrement codée, le champ déplacement contient le codage en complément à 2, sur 22 bits, de la valeur -4.

La question intéressante est : que contiennent les deux champs de bits notés par des `?` ? L'instruction `sethi` comporte un champ de 22 bits qui contient une valeur immédiate. `sethi %hi(D), %r1` signifie qu'il faut utiliser les 22 bits de poids fort (*high*) de la valeur d'adresse représentée par `D` comme valeur immédiate. Or $D = A + 24$, donc $\texttt{hi}(D) = \texttt{hi}(A + 24)$. De même, `or %r1, %lo(D), %r1` signifie qu'il faut placer dans le champ de 13 bits réservé à une valeur immédiate, les 10 bits de poids faible (*low*) de la valeur d'adresse représentée par `D`, ce qui vaut : $\texttt{lo}\ (A + 24)$.

Ces deux valeurs dépendent de l'adresse de chargement A, et les deux champs représentés par des `?` ne peuvent donc pas être remplis correctement tant que l'on ne connaît pas A.

2.3.4 Fichier objet translatable et algorithme de translation d'adresses

Le fichier objet translatable est une étape intermédiaire entre le programme en assembleur donné figure 18.7 et le contenu de la mémoire vive lors de

l'exécution de ce programme. La phase qui va du programme en assembleur au fichier translatable est appelée *assemblage*, elle laisse des trous dans le codage binaire des instructions et des données. La phase qui va du fichier translatable à l'image mémoire du programme est appelée *chargement/lancement* (Cf. Chapitre 20). Outre les aspects système d'allocation de mémoire, le mécanisme de chargement/lancement applique l'*algorithme de translation d'adresses* que nous définissons ici, et qui a pour but de remplir les trous.

Pour permettre à la procédure de chargement/lancement de compléter les trous, le fichier translatable doit contenir des informations additionnelles appelées *données de translation*, qui décrivent *où* se trouvent les instructions ou les données incomplètes (les trous) et *comment* les compléter le moment venu.

Pour chaque trou, le fichier objet translatable fournit une *donnée de translation*. Intuitivement, pour l'exemple étudié plus haut, l'information nécessaire est la suivante : il existe un trou de 22 bits cadré à droite dans le mot d'adresse $A + 0$, à remplir avec les 22 bits de poids forts de la valeur $(A + 24)$, notés $hi22(A + 24)$. Il existe un trou de 13 bits cadré à droite dans le mot d'adresse $A + 4$, à remplir avec les 10 bits de poids faibles de la valeur $(A + 24)$, notés $lo10(A + 24)$.

Une donnée de translation comporte donc les informations suivantes : 1) la position du trou à remplir et sa taille, donnée de manière relative au début du programme ; 2) le *mode de calcul* de la valeur à utiliser : c'est une *fonction* de A, dans laquelle apparaît une constante (ici 24) connue à l'assemblage (c'est le décalage de la position du symbole `D` par rapport au début du programme). Le mode de calcul est donc composé d'une constante K et d'une expression dans laquelle faire intervenir la constante K et A.

Les données de translation sont bien sûr codées sur un format fixe fini, et il est hors de question de coder des modes de calcul (des expressions) quelconques, comme par exemple $hi((A + 24) * 42)$. En réalité, l'ensemble des expressions nécessaires est entièrement défini par le jeu d'instructions de la machine considérée. Par exemple, pour SPARC, les expressions nécessaires pour compléter tous les trous possibles dans les instructions et les données sont au nombre de 32. Il suffit de 5 bits pour les coder. Ce codage est une convention système, connue de tous les outils qui produisent des fichiers objets translatables, et de la procédure de chargement/lancement qui les interprète.

Nous donnons Figure 18.9 une ébauche d'algorithme de translation d'adresses pour SPARC, en faisant apparaître les deux expressions nécessaires dans l'exemple de la figure 18.7 et la donnée de translation nécessaire au codage de l'instruction `add %r1, X, %r1`.

lexique
ModeDeCalcul : le type (reloc_hi22, reloc_lo10, reloc13, ...)
{ *Type énuméré représentant les expressions de calcul. Les noms sont les noms effectifs utilisés dans les systèmes à base de* SPARC. *En anglais, translation se dit* relocation }
DonnéeTranslation : le type <
position : un entier ≥ 0, mode : un ModeDeCalcul,
const : un entier >
{ *La taille du trou est implicitement codée dans le mode de calcul ; la position du trou est donnée en adresses d'instructions (donc c'est un multiple de 4) car le codage des instructions* SPARC *est tel que les trous sont toujours cadrés à droite.* }
D : une DonnéeTranslation
Masque22pF : l'entier $((2^{22} - 1) \times 2^{10})$
{ *22 bits à 1 en poids Forts, 10 bits à 0 en poids faibles. Pour les détails de construction, revoir le paragraphe 4. du chapitre 3, à propos du lien entre l'arithmétique et les booléens.* }
Masque10pf : l'entier $2^{10} - 1$
Masque13pf : l'entier $2^{13} - 1$
{ *10 (ou 13) bits à 1 en poids faibles, 22 (ou 19) bits à 0 en poids Forts* }

algorithme
{ *Le fichier objet translatable a été copié en mémoire à partir de l'adresse A, instructions d'abord, données ensuite, comme sur la figure 18.8. On parcourt les données de translation du fichier objet.* }
D parcourant les données de translation du fichier :
selon D.mode :
{ *On modifie un mot de 32 bits dans la mémoire, à l'adresse A+D.position, en superposant un autre mot de 32 bits, grâce à une opération OR bit à bit. Voir chapitre 12, paragraphe 1.4.1* }
D.mode = reloc_hi22 :
MEM [A + D.position] $\longleftarrow_4$
MEM [A + D.position] OR (((A + D.const) ET Masque22pF) / 2^{10})
D.mode = reloc_lo10 :
MEM [A + D.position] $\longleftarrow_4$
MEM[A + D.position] OR ((A + D.const) ET Masque10pf)
D.mode = reloc_13 :
MEM [A + D.position] $\longleftarrow_4$
MEM [A + D.position] OR ((A + D.const) ET Masque13pf
....

FIG. 18.9 – Algorithme de translation d'adresses pour SPARC.

3. Format des fichiers objets translatables et édition de liens

3.1 Edition de liens, définition

L'édition de liens consiste à prendre un ensemble de fichiers translatables et à tenter de les rassembler pour en faire un fichier unique exécutable. Ce n'est pas exactement la spécification de l'outil β de la figure 18.5. L'outil β réalise en fait la *fusion* de deux fichiers objet translatables, et donne un fichier du même type. Une erreur peut survenir lors de la fusion en cas de double définition d'un symbole.

L'édition de liens peut être vue comme un mécanisme de fusion n-aire (ou bien binaire, et on fusionne les fichiers deux par deux), suivie d'une étape de vérification : pour que l'ensemble des fichiers fournis constitue effectivement un programme exécutable, il ne doit plus y avoir de symbole indéfini (un appel d'une fonction qui n'est définie nulle part, par exemple). Noter que lorsqu'on utilise une fonction en bibliothèque, la commande d'édition de liens spécifie la bibliothèque, c'est-à-dire que cette bibliothèque est prise en compte dans l'ensemble des fichiers à rassembler.

L'édition de liens de n fichiers $F_1, ..., F_n$ peut être réalisée par l'algorithme suivant (même si ce n'est pas très réaliste) :

$F \longleftarrow \beta\ (F_1,\ F_2)$
$F \longleftarrow \beta\ (F,\ F_3)$
...
$F \longleftarrow \beta\ (F,\ F_n)$
{ *Des erreurs de double définition peuvent survenir lors des fusions* }
Si le fichier F contient des symboles indéfinis alors
    ERREUR
sinon
    Transformer F en fichier exécutable
    { *ce n'est pas tout à fait le même format que les fichiers objet translatables, voir détails ci-dessous.* }

On peut considérer que tout fichier soumis à la procédure de chargement/lancement est *complet*, c'est-à-dire qu'il ne comporte plus de symboles indéfinis.

3.2 Format d'un fichier objet translatable

Comme signalé plus haut, un fichier objet translatable est constitué essentiellement de langage machine. Nous avons défini au paragraphe précédent les besoins en informations supplémentaires relatives à la translation d'adresses.

Il nous faut ici compléter ces informations par la *table des symboles*. En effet, en étudiant l'outil β au paragraphe 2.2.2, nous avons supposé que le

format produit par α est du langage d'assemblage, dans lequel on retrouve facilement les noms des fonctions sous forme de symboles (ou étiquettes). La table des symboles sert à établir le lien entre les noms du programme d'origine et des adresses dans le programme en langage machine correspondant. Les noms sont indispensables à la fusion de deux fichiers en langage machine ; c'est le seul moyen de mettre en correspondance une instruction issue de l'appel d'une procédure F avec l'étiquette qui marque le début du code de F. C'est aussi dans la table des symboles qu'on trouve l'information relative à la portée des noms, nécessaire lors de la fusion.

Nous détaillons ci-dessous la structure d'un fichier objet. Nous nous inspirons des fichiers objet SOLARIS, mais l'exposé vaut pour la plupart des systèmes. Un fichier objet est composé de sections dont les formats diffèrent. On y trouve au début une en-tête puis, dans un ordre fixé mais quelconque, les zones TEXT et DATA, les zones de translation TEXT et DATA, la table des symboles et la table des chaînes. Nous détaillons ces diverses zones ci-dessous.

3.2.1 En-tête

Un fichier objet translatable commence par une en-tête qui constitue en quelque sorte la *carte* du fichier. On y trouve en particulier l'indication sur la taille de toutes les autres sections, qui permet d'y accéder directement par des décalages (Cf. Chapitre 19).

On y trouve aussi la taille de la zone BSS du programme d'origine. La section BSS des programmes en langage d'assemblage est analogue à la section DATA, mais on ne fait qu'y demander la réservation d'une certaine zone mémoire, sans déclarer de valeur initiale. La seule information nécessaire dans le fichier objet est donc la taille de cette zone, alors que pour la zone DATA il faut stocker le codage de toutes les valeurs initiales. Au moment du chargement/lancement, l'allocation de mémoire est faite en tenant compte des besoins de la zone BSS.

Enfin l'en-tête indique le *point d'entrée* du programme, c'est-à-dire où se trouve l'instruction correspondant au début du programme principal, parmi toutes les instructions de la zone TEXT. Le point d'entrée est donné comme un décalage par rapport au début de la zone TEXT.

3.2.2 Zones TEXT et DATA

La zone TEXT contient le codage binaire des instructions du programme. Elle comporte éventuellement des trous, comme déjà vu sur l'exemple du paragraphe 2.3.

La zone DATA contient le codage binaire des valeurs initiales spécifiées dans le programme en assembleur. Elle peut également comporter des trous. Dans les exemples que nous avons vus jusque là, il n'y a jamais d'*utilisation* de symbole en zone DATA, et donc pas de problème d'utilisation absolue. La syntaxe de l'assembleur autorise pourtant des déclarations de zones DATA de

la forme :

```
X:   .long 42
Y:   .long 212
     .long X       ! utilisation absolue du symbole X
     .long Y - X   ! utilisation relative des symboles Y et X
```

A l'exécution, le mot de 32 bits situé en troisième position dans la zone des données initialisées contiendra l'adresse effective correspondant au symbole `X`. C'est une utilisation absolue de `X`, et l'assembleur ne peut produire le mot de 32 bits correct.

En revanche, l'expression `Y - X` est une utilisation relative des deux symboles `X` et `Y` : la différence des adresses représentées par ces deux symboles ne dépend pas de l'adresse de chargement du programme, et peut être calculée lors de l'assemblage ; elle vaut 4. Une telle déclaration en zone DATA utilise les symboles de la même manière que l'instruction de branchement `ba debut` de l'exemple étudié plus haut.

3.2.3 Table des symboles et table des chaînes

Pour conserver l'information relative aux *noms* des symboles, le fichier comporte une table des chaînes et une table des symboles.

La table des chaînes est un tableau de caractères où sont rangées, les unes après les autres et séparées par des 0, toutes les chaînes de caractères qui sont des noms d'étiquettes (et non pas les chaînes de caractères qui sont des données du programme, déclarées par des directives `.asciz "machaine"` dans la zone DATA, et dont le code ASCII est présent dans la zone DATA du fichier objet). Isoler la table des chaînes permet de ne stocker qu'une fois une chaîne lorsqu'elle est le nom de plusieurs symboles. Nous expliquons au paragraphe 3.3 comment un fichier objet translatable peut contenir plusieurs symboles de même nom.

La table des symboles est une collection de n-uplets décrivant chacun un symbole, rangés dans un tableau indicé de 1 à NombreSymboles (l'ordre n'est pas pertinent, toutefois). Chaque n-uplet est codé sur un format fixe (de l'ordre de 12 octets). Le type correspondant est donné figure 18.10.

L'ensemble Zone et Portée permet de déterminer exactement le statut du symbole considéré (Figure 18.11).

3.2.4 Données de translation TEXT et DATA, généralisation

Le fichier objet contient deux zones distinctes pour les données de translation relatives aux instructions, et pour celles relatives aux données. La zone BSS n'étant pas initialisée, il n'y a pas de trous dans son codage, et donc pas de données de translations associées.

Au paragraphe 2.3.4 nous avons donné la structure des données de trans-

ZoneDef : le type < zTEXT, zDATA, zBSS, zNONDEF >
Symbole : le type <
Nom : un entier > 0,
{ *c'est un indice dans la table des chaînes. Le nom du symbole est la chaîne comprise entre cet indice inclus et le prochain 0.* }
Zone : un ZoneDef,
{ *zone où apparaît la définition de l'étiquette dans le programme en langage d'assemblage, ou valeur zNONDEF si l'étiquette n'est pas définie dans le fichier considéré, mais seulement utilisée, par exemple dans une instruction* `call`. }
Portée : un booléen, { *voir détails ci-dessous.* }
Valeur : un entier ≥ 0
{ *L'écart d'adresses entre le début de la zone TEXT et la position où est définie l'étiquette. Dans l'exemple de la figure 18.7, 24 est la valeur du symbole* `D`. *Lorsque Zone=zNONDEF, Valeur est non pertinent.* }
>

FIG. 18.10 – Type d'un élément de la table des symboles

Portée $\longrightarrow$ Zone $\downarrow$	vrai	faux
Zone $\neq$ zNONDEF	(1) Le symbole est défini localement, et exporté	(2) Le symbole est défini localement, non exporté
Zone $=$ zNONDEF	(3) Le symbole n'est pas défini localement, il peut être pris dans un autre fichier (importé)	Le symbole n'est pas défini localement, il ne peut pas être pris dans un autre fichier. Ce cas n'existe pas normalement.

FIG. 18.11 – Portée et zone de définition des symboles

Modedecalcul : le type (reloc_hi22, reloc_lo10, reloc13, ...)
DonnéeTranslation : le type <
position : un entier ≥ 0, mode : un ModeDeCalcul,
numsymb : un entier sur 1..NombreSymboles
{ *numéro du symbole qui apparaît dans l'instruction à trou. on peut consulter la table des symboles pour savoir si le symbole est défini et dans ce cas consulter sa valeur, qui correspond à la constante du type présenté figure 18.9.* }
>

FIG. 18.12 – Type d'une donnée de translation générale

lation dans le cas des symboles connus à l'assemblage, c'est-à-dire définis dans le fichier.

Les utilisations de symboles indéfinis sont une autre source de trous dans le code généré par l'assembleur. Ainsi une instruction de branchement relatif `ba labas` produit un mot de 32 bits dont les 22 bits de poids faibles forment un trou, lorsque le symbole `labas` n'est pas défini localement.

Pour repérer ce trou, et décrire comment le compléter plus tard, on utilise également une donnée de translation. Le type d'une donnée de translation générale est donné figure 18.12.

3.3 Mise en oeuvre de l'édition de liens

Nous étudions le problème de l'édition de liens en ébauchant un algorithme de fusion de deux fichiers objets translatables, et rendant un nouveau fichier objet translatable. Les éditeurs de liens réels travaillent sur n fichiers simultanément, mais cela ne fait que compliquer l'algorithme.

3.3.1 Structure du fichier fusionné

Considérons trois fichiers objets 1, 2, et 3, tels que 3 soit la fusion des fichiers 1 et 2. Notons respectivement Hi, Ti, Di, TTi, TDi, TSi, TCi l'en-tête, la zone TEXT, la zone DATA, la zone de translation TEXT, la zone de translation DATA, la table des symboles et la table des chaînes du fichier numéro i.

Le fichier 3 est en gros la concaténation, zone par zone, des fichiers 1 et 2. Plus précisément :

- T3 est la concaténation de T1 et T2 ; certains trous sont éventuellement remplis. Le fichier fusionné comporte toutes les instructions du fichier 1, et toutes celles du fichier 2. L'ordre n'a pas d'importance, mais il faut décider où se trouve le point d'entrée. L'outil de fusion peut prendre en paramètre le nom d'une étiquette qui désigne le point d'entrée global.
- De même, D3 est la concaténation de D1 et D2 ; certains trous sont éventuellement remplis. Le fichier fusionné comporte toutes les données du fichier 1 et toutes celles du fichier 2. Il n'y a pas de problème de point d'entrée.
- TT3 est basé sur l'union des données de translation TEXT du fichier 1 (TT1) et de celles du fichier 2 (TT2). Toutefois certaines données de translation peuvent être utilisées pendant la fusion pour compléter certains trous en zone T3, et disparaissent donc.
- La situation est similaire pour TD3 par rapport à TD1, TD2.
- La table des symboles TS3 est obtenue d'après TS1 et TS2 de la manière suivante : les noms de symboles qui n'apparaissent que dans l'un des fichiers sont conservés tels que ; pour un nom de symbole qui apparaît dans les deux fichiers, il faut considérer, pour chaque symbole, les 3 cas de la figure 18.11.

D'autre part un fichier objet peut comporter plusieurs symboles de même nom. L'analyse détaillée de tous les cas possibles donne la structure de l'algorithme de fusion.

– La table des chaînes TC3 est une véritable *union* des deux tables de chaînes TC1 et TC2, en ne conservant qu'un seul exemplaire dans le cas où une même chaîne apparaît dans les deux fichiers.
– Enfin l'en-tête H3 récapitule les tailles des différentes zones, et donne le point d'entrée du fichier fusionné.

3.3.2 Que se passe-t-il lors de la fusion ?

Effets de décalages La fusion des deux fichiers a tout d'abord des effets de décalage, puisque les symboles sont numérotés indépendamment dans les deux fichiers, et leurs valeurs calculées indépendamment :

– L'union des tables de symboles produit une nouvelle numérotation globale des symboles, qu'il faut reporter dans les données de translation qui y font référence.
– La concaténation des zones TEXT (resp. DATA) a pour effet de déplacer les étiquettes du deuxième fichier par rapport au début de l'ensemble. Il faut donc modifier la valeur des symboles du deuxième fichier. Aux symboles définis en zone T2, on ajoute la taille de T1 ; aux symboles définis en zone D2 on ajoute la taille de T1, plus la taille de D1, etc. Dans le fichier 3, les valeurs des symboles sont des décalages par rapport au début de la concaténation T1 T2 D1 D2.
– La concaténation des zones TEXT (resp. DATA) a aussi pour effet de déplacer les trous par rapport au début de l'ensemble. Il faut donc mettre à jour les positions de trous dans les données de translation. Cela peut se faire globalement au début.

Identification de symboles Un cas intéressant de fusion survient lorsqu'un trou peut-être rempli. Cela se produit par exemple si le fichier 1 comporte une instruction `call F` avec `F` non défini, et le fichier 2 une définition de `F` exportée. On a donc un symbole de nom `F` dans chacun des fichiers. D'autre part l'instruction `call F` est incomplète dans le fichier 1, et il y a dans TT1 une donnée de translation qui décrit ce trou.

En considérant les portées relatives de ces deux symboles, la fusion détermine qu'il s'agit en fait du *même symbole*, et réalise l'identification. Cela consiste à ne garder qu'un élément dans la table de symboles globale. D'autre part, puisque l'instruction `call F` fait une utilisation *relative* du symbole `F`, et que celui-ci est connu dans le fichier fusionné, l'instruction `call F` incomplète qui provenait du fichier 1 peut maintenant être complétée. La donnée de translation du fichier 1, plus les informations sur le symbole données par TS2 suffisent pour cela.

Noter que si la même situation se présente, mais avec une instruction utilisant `F` de manière absolue dans le fichier 1, il y a bien identification des symboles, mais il subsiste une donnée de translation pour cette instruction, et elle ne sera complétée qu'au chargement.

Il y a également identification de symboles lorsque par exemple les deux fichiers importent la même fonction, qui n'est définie dans aucun d'eux. Elle le sera peut-être dans un troisième fichier, qui sera fusionné avec ceux-là plus tard.

Plusieurs symboles de même nom dans un fichier Les fichiers obtenus par assemblage d'un programme en langage d'assemblage ont exactement un symbole par nom. Pour bien comprendre l'algorithme de fusion, qui peut s'appliquer incrémentalement, il faut remarquer qu'un fichier objet obtenu par fusion peut contenir deux symboles de *même* nom, qui diffèrent par leurs autres attributs (zone de définition, valeur). Ce phénomène est dû au mécanisme de masquage des noms par fichier : si l'on fusionne deux fichiers objet contenant chacun un symbole de nom i défini et non exporté, le fichier global comporte deux symboles de nom i. Notons toutefois que si plusieurs symboles ont le même nom, ce sont nécessairement des symboles définis localement et non exportés. Dans les données de translation du fichier, les symboles sont référencés par numéro, pas par leur nom.

De manière générale, pour un nom de symbole i donné, un fichier objet peut contenir : un nombre quelconque de symboles de nom i définis localement et non exportés (cas 2 de la figure 18.11) ; au plus un symbole de nom i *visible*, c'est-à-dire défini localement et exporté (cas 1), ou bien non défini et importé (cas 3).

3.3.3 Ebauche d'algorithme

Nous donnons ci-dessous la structure principale de l'algorithme de fusion, qui est entièrement guidé par l'union des tables de symboles : on commence par recopier entièrement TC1 et TS1 dans TC et TS, puis l'on examine les symboles de TS2 un par un.

```
recopier TS1 dans TS ; recopier TC1 dans TC
{ Parcours de la table des symboles du fichier 2 }
i2 parcourant 1..Taille(TS2) :
    n2 ⟵ TS2[i2].Nom
    N ⟵ TC2[n2] { la chaîne de TC2 commençant à l'ind. n2 }
    Z2 ⟵ TS2[i2].Zone ; P2 ⟵ TS2[i2].Portée ; V2 ⟵ TS2[i2].Valeur
```

La figure 18.13 détaille l'analyse par cas selon les portées respectives des symboles communs aux deux fichiers.

si N n'apparaît pas parmi les chaînes de TC1
{ *Cas simple, on ajoute la chaîne N à TC (à la suite, soit n l'indice), et le symbole TS2[i] à TS, à la suite, en le renumérotant. Il conserve sa portée et sa zone de définition. Son nom vaut maintenant n, indice dans la table de chaînes globale. Sa valeur est décalée ; il faut aussi reporter son nouveau numéro dans TT et TD (Cf. Paragraphe 3.3.2).* }
sinon
{ *N apparaît parmi les chaînes de TC1. Il y a dans TS1* **un ou plusieurs** *symboles portant ce nom N. C'est le cas intéressant. Par examen de P2, Z2 et des portées et zones de définition de ces symboles, on détermine que faire de TS2[i].* }
si non P2 { *symbole défini et caché dans le fichier 2* }
{ *Aucune identification ne peut avoir lieu avec les symboles portant le même nom en provenance du fichier 1. On ajoute le symbole TS2[i], en le renumérotant et en décalant sa valeur. Il faut aussi reporter son nouveau numéro dans TT et TD.* }
sinon { *c'est-à-dire symbole visible* }
{ *Aucune identification ne peut avoir lieu entre le symbole TS2[i] et les symboles de même nom cachés dans le fichier 1. Il faut donc déterminer s'il existe un symbole visible (défini et exporté ou bien importé) de même nom dans le fichier 1, que l'on notera S1.* }
s'il n'existe pas S1
{ *on ajoute le symbole TS2[i], en le renumérotant et en décalant sa valeur. Il faut aussi reporter son nouveau numéro dans TT et TD.* }
s'il existe S1 = < n1, Z1, vrai, V1 >
{ *Il existe deux symboles de même nom visibles dans les deux fichiers. Tout dépend maintenant du fait qu'ils sont définis ou non.* }
selon Z1, Z2 :
Z1 = zNONDEF et Z2 = zNONDEF :
{ *Il y a identification des symboles. Par exemple les deux fichiers importent une fonction qui sera définie dans un autre, non encore fusionné. Le symbole reste non défini, importé. Il n'a pas de valeur. Il est renuméroté par rapport au numéro qu'il avait dans le fichier 2, et il faut reporter le nouveau numéro dans TT, TD.* }
Z1 ≠ zNONDEF et Z2 ≠ zNONDEF :
ERREUR : double définition
(Z1 = zNONDEF et Z2 ≠ zNONDEF) ou
(Z1 ≠ zNONDEF et Z2 = zNONDEF) :
{ *C'est LE cas intéressant de fusion. L'un des fichiers importe un symbole, qui se trouve être défini et exporté par l'autre. Il y a identification. Le symbole devient défini exporté. Sa valeur est calculée d'après la valeur qu'il a dans le fichier où il est défini. On reporte son nouveau numéro dans les données de translation. De plus, les utilisations* **relatives** *du symbole dans le fichier qui l'importe peuvent être résolues, et cela supprime des données de translation.* }

FIG. 18.13 – Structure de l'algorithme de fusion

Chapitre 19

Système de gestion de fichiers

Dans ce chapitre nous nous intéressons au problème de la gestion des informations qui doivent être placées en mémoire *secondaire*. Nous avons vu dans le chapitre 16 comment connecter un périphérique d'entrée/sortie à un ordinateur, et dans le chapitre 17 comment réaliser l'interface entre ce périphérique et les programmes du système d'exploitation. Les périphériques de mémoire secondaire sont les disques, les bandes magnétiques, etc., que l'on peut lire et écrire un nombre "infini" de fois.

La mémoire secondaire est utilisée pour le stockage de données dont la durée de vie doit être supérieure à la durée de vie des programmes qui les manipulent, et même éventuellement supérieure à la durée de vie du système informatique qui les a stockées. Dans le premier cas on trouve par exemple les fichiers source, objet et exécutable d'un programme de l'utilisateur (voir chapitre 18). Dans le deuxième cas il peut s'agir des fichiers d'une base de données, qui doivent être conservés et transmis d'un environnement informatique (machine, système d'exploitation) à un autre.

Mise à part l'exigence sur leur durée de vie, les données que l'on peut vouloir stocker sur des supports permanents ne sont pas de nature intrinsèquement différente de celle des données manipulées par les programmes. Nous avons étudié au chapitre 4 un petit langage de description des *structures de données* usuelles, et comment coder et installer dans les éléments du tableau MEM tout type de donnée structurée. Les éléments du tableau représentent les plus petits blocs de la mémoire accessibles par le processeur (en général des octets, voir chapitre 15). Les supports de mémoire secondaire imposent également des contraintes physiques sur la taille du plus petit bloc accessible (d'un ordre de grandeur différent toutefois : 256 octets pour un disque par exemple, voir chapitre 17).

L'analogie s'arrête là. En effet, abstraire la mémoire vive d'une machine par un *tableau* est légitime, puisque la mémoire vive offre l'accès *direct* par une adresse. En revanche certains supports de mémoire secondaire n'offrent pas l'accès direct : c'est le cas des bandes magnétiques.

Lorsqu'on s'intéresse à l'installation des données dans les blocs accessibles

d'une mémoire secondaire, il faut tenir compte des accès que l'on désire réaliser sur ces données : si l'on a besoin de l'accès direct, un support à accès séquentiel ne convient pas. Chacun sait qu'il est plus facile d'écouter le troisième mouvement d'une symphonie sur un disque compact que sur une cassette audio.

Il existe une deuxième différence essentielle entre les données d'un programme (qui sont stockées quelque part en mémoire vive pendant l'exécution du programme) et les données stockées sur un support de mémoire secondaire. Il s'agit du mécanisme d'accès aux informations par leur *nom*, autrement dit du lien entre un *nom externe* connu de l'utilisateur et une *adresse* en mémoire.

Dans le cas des programmes, les informations sont rangées dans des variables *nommées* par l'utilisateur ; la correspondance entre le nom et l'adresse dans le tableau MEM est calculée par le compilateur, et prise en compte lors de la fabrication du fichier exécutable, dans lequel on peut oublier les noms (sauf si l'on désire effectuer du débogage symbolique, voir chapitre 18).

Dans le cas des données présentes sur un support de mémoire secondaire, le nom externe est un *nom de fichier*, dans la terminologie usuelle. La correspondance entre ce nom externe et les adresses des données du fichier sur le support est établie par exemple lors de la création d'un fichier, par le logiciel qui s'occupe de la gestion du support. Cette correspondance est une information dont la durée de vie doit être au moins égale à celle des données considérées : elle doit donc être stockée sur le support lui-même. A n'importe quel autre "endroit" (mémoire vive de la machine, que ce soit dans un programme système ou dans un programme utilisateur), elle aurait une durée de vie inférieure à celle des données sur le support de mémoire.

On appelle *système de fichiers* l'ensemble des données stockées sur un support de mémoire secondaire (disque, bande, ...). Ces données comprennent bien sûr les données de l'utilisateur, mais aussi des informations qu'il n'a pas à connaître, sur l'organisation de ce support de mémoire : correspondance entre noms externes et adresses, où reste-t-il de la place libre ?, etc.

On appelle *système de gestion de fichiers* (SGF dans la suite) l'ensemble des programmes responsables de l'installation d'un système de fichiers sur un support de mémoire secondaire. Le SGF réalise l'interface entre l'utilisateur, qui peut désigner ses données par des noms de fichiers par exemple, et le logiciel pilote de périphérique qui réalise les lectures/écritures effectives sur le support de mémoire.

Notons que ces deux notions correspondent aussi bien à la gestion des fichiers utilisateurs sur un système mono ou multi-utilisateurs ordinaire, qu'à la manipulation des fichiers de stockage d'un SGBD (Système de Gestion de Bases de Données). L'organisation des informations sur les supports secondaires est toutefois plus compliquée dans le cas des SGBD, pour lesquels les contraintes de temps d'accès sont primordiales (et le volume des données tel que l'on ne peut pas systématiquement recopier les données en mémoire vive avant de les traiter). Nous étudierons dans ce chapitre le cas d'un système de fichiers

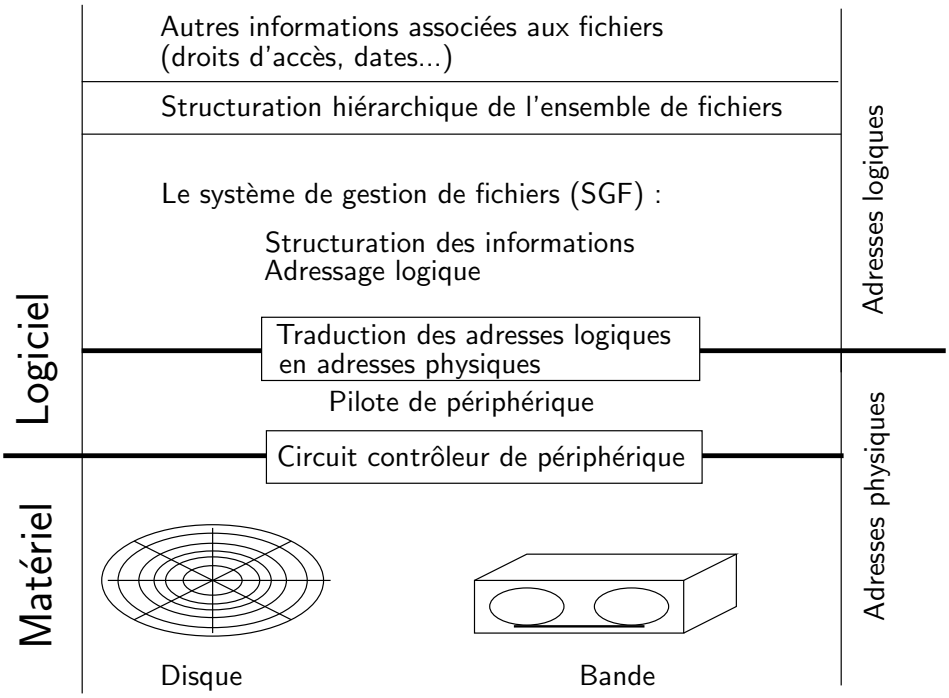


FIG. 19.1 – Situation générale du système de gestion de fichiers

utilisateurs ordinaire.

Nous précisons tout d'abord dans le paragraphe 1. la position du système de gestion de fichiers, entre le matériel spécifique (disques, bandes et leurs contrôleurs, Cf. Chapitre 16) et la vision des informations que peut avoir l'utilisateur à travers un interprète de commandes textuel (Cf. Chapitre 20). Le paragraphe 2. rappelle la notion d'adresse physique héritée du pilote de périphérique (Cf. Chapitre 17) et définit les notions de fichier et d'adresse logique. Le paragraphe 2.3 étudie la fonction principale d'un système de gestion de fichiers, c'est-à-dire l'implantation des fichiers sur les supports physiques. Le paragraphe 4. décrit comment associer aux fichiers des informations comme les noms externes ou la date de création. Finalement, nous étudions dans le paragraphe 5. quelques fonctions de base d'un système de gestion de fichiers, comme le formatage d'un disque ou la création de fichier.

1. Situation du système de gestion de fichiers

La figure 19.1 illustre la situation du système de gestion de fichiers. Nous détaillons ci-dessous la nature des informations manipulées aux divers niveaux ainsi que les primitives disponibles aux divers niveaux.

Le logiciel pilote de périphérique associé au lecteur de disque ou de bande

magnétique (Cf. Chapitre 17) fournit vers les couches d'abstraction supérieures une interface qui permet de manipuler des *blocs* — ou *unités d'accès* — numérotés séquentiellement. Dans la suite de ce chapitre, nous appellerons *adresse physique* le mécanisme de désignation non ambiguë d'une unité d'accès, c'est-à-dire le numéro de bloc. Rappelons que le logiciel pilote de périphérique traduit cette numérotation séquentielle en couples (numéro de secteur, numéro de piste) pour les disques (l'unité d'accès peut même être constituée de plusieurs secteurs).

Tout en haut du schéma on trouve la couche supérieure visible par l'utilisateur. Les informations manipulées sont des *noms* de fichiers (qui reflètent éventuellement la structure hiérarchique de l'ensemble des fichiers). On peut trouver également d'autres informations associées à un fichier, comme les droits d'accès des différents utilisateurs, la date de création, le nom du programme qui l'a créé, etc.

Le SGF est la couche intermédiaire : il assure la correspondance entre les noms de fichiers et la position des données de ces fichiers sur les blocs du support de mémoire.

Lorsque l'utilisateur tape `editer /users/machin/toto.c`, le programme `editer` fait appel aux fonctions du SGF pour retrouver le lien entre ce nom externe et la position des données sur le support de mémoire, et aux fonctions offertes par le logiciel pilote de périphérique pour réaliser la lecture effective de ces données.

Nous détaillons au paragraphe 5. les principales fonctions du SGF accessibles aux programmes de l'utilisateur.

2. Structure des données et influence sur l'implantation

2.1 Notion de fichier

Les données à stocker sont regroupées dans des *fichiers*. C'est l'unité de base à laquelle on peut associer un nom externe connu de l'utilisateur (à travers un interprète de commandes, ou par programme). Chaque fichier a une certaine *structure* interne, qui correspond au *type* des données qui y sont stockées.

2.2 Relation entre la structure des données d'un fichier et le type d'accès nécessaire

2.2.1 Structure des données

Nous avons déjà rencontré au chapitre 18 les fichiers source, objet et exécutable d'un programme. Le fichier source est un *texte*, c'est-à-dire une suite de caractères. On peut le considérer comme une suite de bits, mais les

opérations usuelles que l'on effectue sur un fichier source (édition, impression, ...) l'interprètent comme une suite de caractères (c'est-à-dire une suite d'octets). De même, le fichier objet peut être considéré comme une suite de bits, ou comme une suite d'octets. Les images ou les contenus de bases de données constituent d'autres types de fichiers.

Les fichiers texte sont intrinsèquement séquentiels, puisque le type Texte manipulé par les programmes est défini comme une séquence de caractères (l'ordre des éléments fait partie du type). Inversement les fichiers de bases de données relationnelles correspondent au type de données ensemble de n-uplets. Un fichier de personnes peut ainsi être constitué de triplets formés d'un nom, d'une date de naissance et d'un nom de ville. Il n'y a pas de structure séquentielle attachée à ce type de données. Les n-uplets peuvent être désignés de manière non ambiguë par une *clé*. Dans la théorie de l'algèbre relationnelle, la clé — ou *identifiant* — d'une relation est un sous-ensemble des champs tel que, pour une valeur des champs de la clé, le n-uplet est unique. Dans le fichier de personnes donné en exemple, on peut décider d'utiliser le nom comme clé.

On appelle *adresse logique* le mécanisme de désignation non ambiguë d'un enregistrement du fichier. Si le fichier comporte une structure séquentielle, c'est en général un numéro par rapport au début ; dans le cas contraire, on utilise la notion d'identifiant fournie par la structuration des données : une valeur de la clé permet de désigner un n-uplet sans ambiguïté.

2.2.2 Types d'accès

Le type d'accès nécessaire aux données d'un fichier dépend de la manière dont on interprète ces données.

Tout fichier peut être considéré comme une suite de bits. Lorsqu'on utilise une commande de dump pour afficher le contenu d'un fichier quelconque, le fichier est interprété, au choix, comme une suite d'octets, de mots de 16 ou 32 bits, ... Les outils spécifiques connaissent la structure et l'utilisent ; des outils génériques peuvent l'ignorer et ne voir qu'une suite de bits.

Un outil comme `dump` ne réalise que des accès séquentiels (lecture dans l'ordre, pas de suppression ni insertion au milieu), et a donc peu d'exigences sur la manière dont est implanté le fichier. En revanche, tout outil qui connaît la structure des données du fichier peut nécessiter un accès direct. Pour la lecture, on peut vouloir par exemple ne lire que le caractère numéro 4200 dans un fichier texte ; ou seulement la section des instructions dans un fichier objet ; ou encore seulement une portion d'une image dans un fichier JPEG, ... Pour l'écriture, on peut avoir besoin d'insérer une ligne dans un texte, de supprimer une section d'un fichier objet, etc.

Noter qu'on peut vouloir réaliser des accès directs à un fichier texte, bien que le type des données soit intrinsèquement séquentiel. Inversement, la sauvegarde d'un disque de base de donnée sur une bande magnétique ne réalise que des accès séquentiels aux données, bien que leur type soit intrinsèquement

non séquentiel.

Reprenons l'analogie avec les disques et cassettes audio : il est plus facile d'écouter la douzième chanson sur un disque compact que sur une cassette audio, mais, lorsqu'on recopie une cassette sur une autre, on n'a besoin de l'accès direct ni sur la source, ni sur la copie.

Dernier point important : pourquoi se préoccuper d'accès direct aux données stockées sur un support de mémoire secondaire ? On pourrait en effet imaginer une manière fort simple de manipuler les fichiers : on commence toujours par le recopier entièrement en mémoire vive (et cela nécessite une lecture séquentielle), puis on effectue tout type de modification, puis on le recopie sur le support de mémoire secondaire (et cela constitue une écriture séquentielle). Dans ce cas les seuls types d'accès nécessaires sont séquentiels.

Cette approche est applicable à l'édition de petits fichiers texte. Pour d'autres types d'application c'est hors de question : tri de gros fichiers, accès à une base de données, ...

Pour assurer des accès directs parmi les données d'un fichier, il faut donc prévoir l'implantation en conséquence.

2.3 Influence du type d'accès nécessaire sur l'implantation

Le type de manipulation envisagée des fichiers peut nécessiter un accès direct ou non. Les adresses logiques sont des numéros, ou des désignations par le contenu. Les supports physiques sont à accès direct ou séquentiel. Il y a donc de nombreux cas à considérer, que nous regroupons en deux classes ci-dessous.

2.3.1 Accès par numéro

C'est le cas réaliste d'un système de gestion des fichiers utilisateurs. Quel que soit le type des données de ces fichiers, ils sont considérés comme des suites d'octets, et les adresses logiques sont des décalages par rapport au début en nombre d'octets.

Accès direct sur support séquentiel C'est un cas peu réaliste. Sur tout support séquentiel on dispose d'une opération de retour au début (rembobinage d'une bande magnétique), mais passer d'un élément d'adresse n à un élément d'adresse m en rembobinant, puis en avançant de m, ne peut pas avoir des performances comparables à celles d'un véritable accès direct.

Accès séquentiel sur support séquentiel Ceci constitue le cas typique de la sauvegarde, pendant laquelle les fichiers sont considérés comme séquentiels. En effet, aucune insertion ou suppression d'élément du fichier ne sera effectuée sur le support à accès séquentiel. Les accès au support séquentiel sont séquentiels, lors des sauvegardes et restaurations.

Accès direct sur support à accès direct Le fichier peut être considéré comme une séquence de n-uplets plus ou moins complexes, numérotés par les adresses logiques.

On trouve dans tout bon ouvrage d'algorithmique une discussion sur les mérites respectifs de la représentation des séquences dans des tableaux ou des séquences chaînées. Dans un tableau, l'ordre des éléments de la séquence est implicite : c'est l'ordre des indices ; dans une séquence chaînée, l'ordre des éléments est représenté de manière explicite : chaque élément "pointe" sur son successeur (et/ou sur son prédécesseur). Notons que les éléments de la séquence sont de toute façon dans le tableau MEM : la différence entre les deux approches est donc entre une implantation *contiguë* et une implantation *dispersée*. .

Avec la solution tableau on occupe le minimum de mémoire ; avec la solution chaînée, on ajoute un "pointeur" par élément (c'est-à-dire une adresse mémoire, de l'ordre de 32 bits). La comparaison sur le temps nécessaire pour une opération d'insertion tourne en revanche à l'avantage de la solution chaînée : il suffit de raccrocher deux ou trois pointeurs et le tour est joué ; dans un tableau il faut ménager une place en *décalant* des éléments, c'est-à-dire en les recopiant d'une case dans une autre. Comme d'habitude en informatique, les gains en place se font au détriment du temps, et vice-versa. Notons également qu'avec une implantation contiguë on dispose de l'accès direct à un élément par son numéro ; on perd cette propriété avec une implantation dispersée.

Imaginons le cas d'une séquence de bits, stockée en utilisant une solution mixte contigüe/dispersée : on peut chaîner entre eux des octets, auquel cas le rapport entre informations utiles et informations de chaînage est assez mauvais : un pointeur occupe de l'ordre de 4 octets, et il en faut 1 par octet utile. En choisissant la taille des blocs chaînés, on règle le rapport entre informations utiles et informations de chaînage.

Appliquons le raisonnement au cas de l'implantation des fichiers.

Dans une implantation contiguë, les emplacements de deux enregistrements consécutifs quelconques du fichier sont eux-mêmes consécutifs à l'intérieur d'une même unité d'accès, ou situés dans deux unités d'accès de numéros consécutifs. La structure de séquence du fichier est représentée grâce à la séquence des adresses physiques. Une insertion ou suppression d'enregistrement dans le fichier demande un décalage des éléments présents, ce qui peut être assez coûteux.

Dans une implantation dispersée, deux enregistrements consécutifs du fichier peuvent être placés à des positions quelconques sur le disque. Pour reconstituer la structure séquentielle, on peut chaîner les éléments entre eux, ou utiliser une table d'implantation (voir paragraphe 3. ci-dessous). L'insertion ou la suppression d'un élément demande alors seulement la réorganisation du chaînage ou de la table d'implantation, mais jamais le déplacement des enregistrements du fichier sur le disque. La solution par chaînage est coûteuse en taille (une adresse de *suivant* pour chaque élément du fichier), et impose un

accès séquentiel au fichier.

2.3.2 Accès direct par le contenu

Ce cas nécessite des organisations spécifiques selon le type d'accès direct nécessaire.

Dans les systèmes de gestion de bases de données (SGBD) par exemple, le mécanisme d'adressage logique est induit par une description de haut niveau de l'ensemble des informations gérées (un ensemble de schémas de relations par exemple, avec définition des *clés* de relations). L'implantation des données sur le disque est réalisé par les couches basses des SGBD, qui constituent des systèmes de gestion de fichiers spécifiques.

Ainsi certains SGBD proposent-ils, dans le langage de description de la structure des informations, un mécanisme qui permet au concepteur de la base de données d'exprimer des contraintes sur la position relative des informations sur le disque. Le mécanisme de *cluster* du logiciel ORACLE permet ainsi de déclarer une association entre des champs X et Y, appartenant respectivement à des relations R et S, si l'on sait que les requêtes d'interrogation comporteront souvent un produit de la forme R (X=Y)* S en algèbre relationnelle (c'est-à-dire un produit cartésien des relations R et S, suivi d'une sélection des n-uplets qui portent la même valeur dans les champs X et Y). Le SGBD tient compte au mieux de cette association, et tente d'installer à des positions proches sur le disque les n-uplets de la relation R qui correspondent à une valeur X0 du champ X, et les n-uplets de la relation S qui correspondent à cette même valeur dans le champ Y. L'idée est que, si on lit sur le disque un n-uplet de la relation R portant une valeur X0 du champ X, on aura sûrement lu dans le même bloc les n-uplets correspondants de la relation S.

Même si le SGBD n'offre pas au concepteur de la base de données un tel moyen de contrôler la position des informations, l'implantation des données est un problème crucial dans la réalisation d'un SGBD, et elle est nécessairement spécifique. L'utilisation d'un système de gestion de fichiers conçu pour gérer l'ensemble des fichiers utilisateurs réduirait considérablement les performances.

3. Implantation dispersée sur un disque

Nous étudions ici un exemple d'implantation dispersée d'un ensemble de fichiers sur un disque, par un mécanisme de table d'implantation à plusieurs niveaux d'indirection. Le paragraphe 3.3 décrit l'organisation des fichiers dans le système de gestion de fichiers d'UNIX.

Dans une implantation dispersée, l'idée est de partitionner le fichier en morceaux, qui seront disposés sur le disque indépendamment les uns des autres ; toutefois un morceau sera placé sur des unités d'accès consécutives. Ainsi le repérage d'un morceau se réduit à *une* adresse physique, connaissant la taille des morceaux.

La taille de ces morceaux est appelée *unité d'allocation*. C'est un multiple de la taille de l'unité d'accès définie pour le disque (en général une puissance de 2). Le choix de l'unité d'allocation permet de contrôler le degré de dispersion des informations du fichier. L'un des cas extrêmes consiste à définir l'unité d'allocation égale à l'unité d'accès : les morceaux du fichier sont de taille minimale, et les informations sont donc dispersées au maximum. L'autre cas extrême consiste à choisir une unité d'allocation très grande, supérieure à la taille des fichiers. Dans ce cas les informations sont dispersées au minimum, c'est-à-dire pas du tout : on retrouve l'implantation contiguë.

L'unité d'allocation est un paramètre fourni par l'utilisateur au moment de l'installation d'un système de fichiers sur un disque (au *formatage logique*, voir paragraphe 5.1). Cette information doit être conservée, puisqu'elle est indispensable pour l'interprétation des données sur le disque ; elle doit avoir une durée de vie égale ou supérieure à celle des données présentes sur le disque, et doit donc être inscrite sur le disque lui-même (voir au paragraphe 3.7 la synthèse des informations présentes sur un disque).

3.1 Cas d'un seul fichier de petite taille

Considérons tout d'abord le cas simple d'un seul fichier de petite taille. On introduit une table d'implantation de taille T, qui donne T adresses physiques de début de blocs de taille égale à l'unité d'allocation.

Cette table d'implantation doit elle-même être stockée sur disque, de manière contiguë, et à une position connue. Elle peut occuper une ou plusieurs unités d'accès au début du disque par exemple (Cf. Figure 19.2).

La taille du fichier est limitée à $T \times U_a$ octets, où U_a représente l'unité d'allocation.

3.2 Cas d'un seul fichier de taille moyenne

Lorsque le fichier est de taille plus importante, la table d'allocation unique ne suffit plus. Les blocs du fichier ont toujours une taille égale à l'unité d'allocation, mais ils ne sont pas tous accessibles directement. On introduit un mécanisme à deux étages : les T premières cases de la table donnent toujours des adresses physiques de début de blocs du fichier ; une case supplémentaire donne l'adresse physique d'une seconde table, qui de nouveau donne T' adresses physiques de début de blocs du fichier (Cf. Figure 19.3).

La taille d'un fichier moyen est ainsi bornée par $(T + T') \times U_a$. Notons que l'accès aux T premiers blocs est plus rapide que l'accès aux blocs suivants, puisqu'il ne nécessite pas d'indirection.

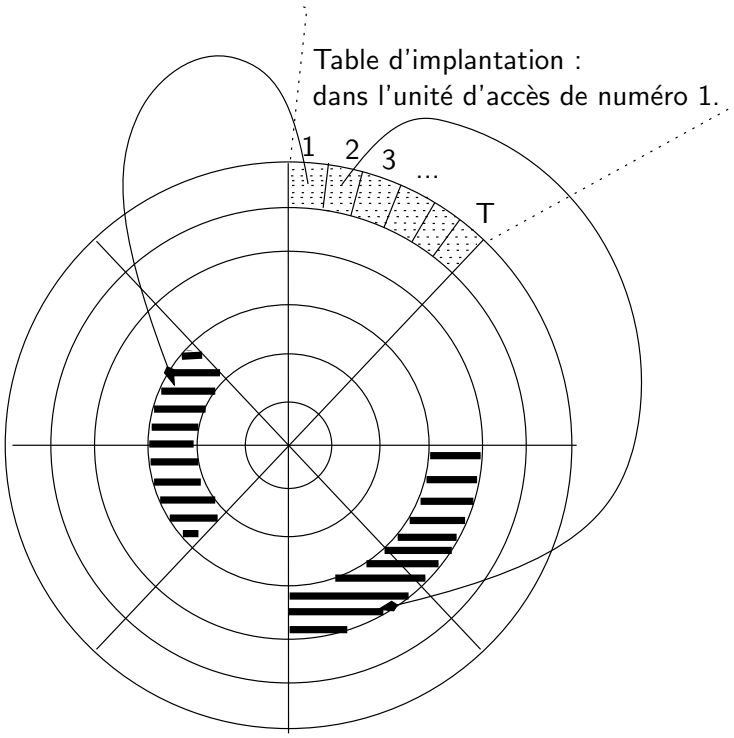


FIG. 19.2 – Implantation dispersée d'un seul fichier de petite taille : l'unité d'allocation est égale à deux unités d'accès, et la table d'implantation est située sur une unité d'accès. Elle donne T adresses physiques d'unités d'accès, qu'il faut interpréter comme les adresses de début des blocs du fichier, chacun de taille égale à l'unité d'allocation.

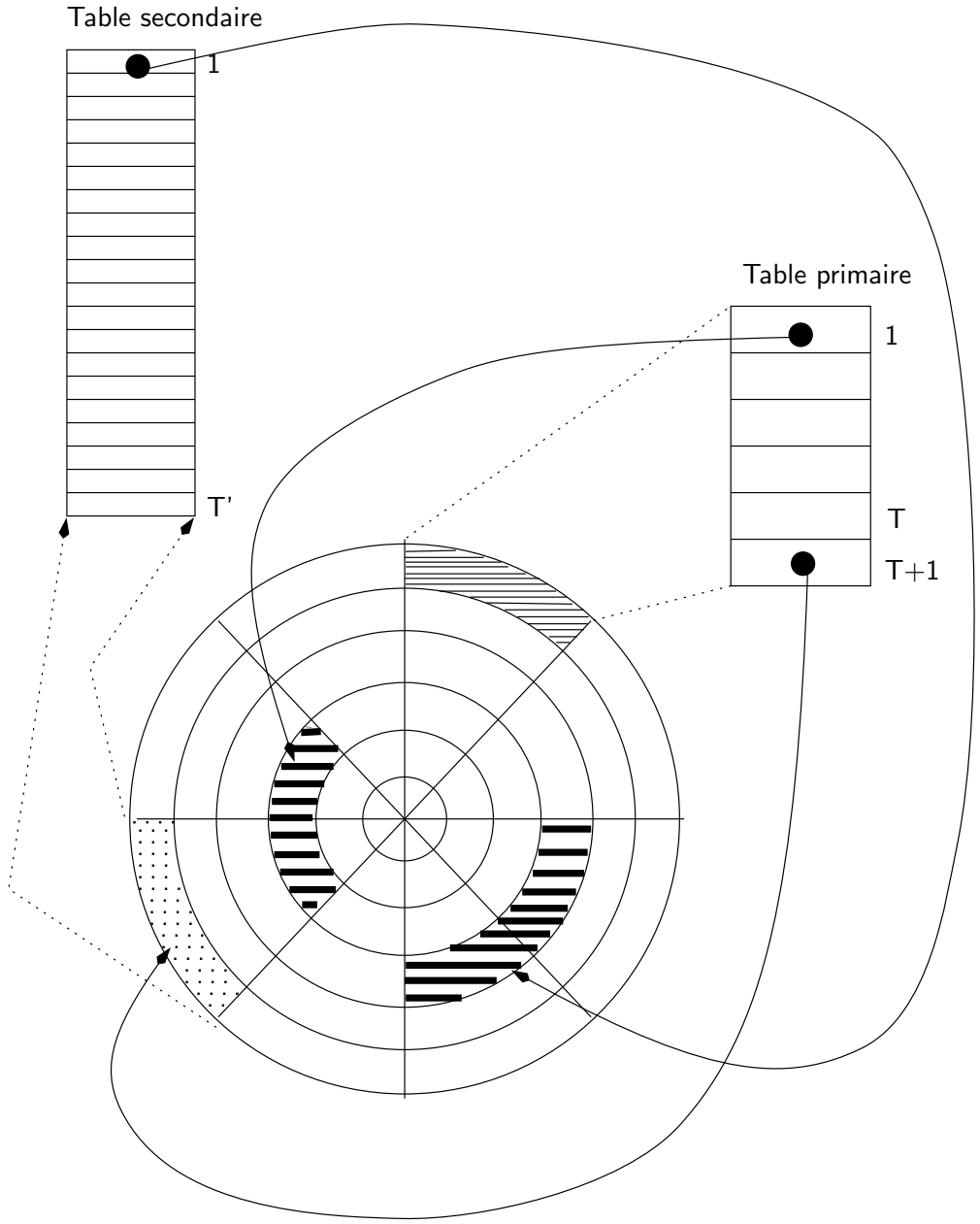


FIG. 19.3 – Implantation dispersée d'un seul fichier de taille moyenne : l'unité d'allocation correspond à deux unité d'accès. La table d'implantation primaire est située sur une unité d'accès. Elle donne tout d'abord T adresses physiques d'unités d'accès, qu'il faut interpréter comme les adresses de début de blocs du fichier. Elle donne également, dans sa case d'indice $T+1$, l'adresse physique d'une unité d'accès qui contient une table secondaire. Cette nouvelle table donne T' adresses physiques d'unités d'accès, qu'il faut interpréter comme les adresses de début des blocs du fichier. Tous les blocs du fichier sont de taille égale à l'unité d'allocation

3.3 Cas d'un seul fichier très grand

Lorsque le fichier est vraiment très grand, on reproduit le raisonnement ci-dessus et on obtient un mécanisme à trois étages. Les T premières cases de la table donnent toujours des adresses physiques de début de blocs du fichier ; une case supplémentaire d'indice $T+1$ donne l'adresse physique d'une seconde table, qui de nouveau donne T' adresses physiques de début de blocs du fichier (comme dans le cas du fichier de taille moyenne) ; une autre case, d'indice $T+2$, donne l'adresse d'une table de tables ; une dernière case, d'indice $T+3$, donne l'adresse d'une table de tables de tables d'adresses physiques de début de blocs du fichier.

La taille d'un grand fichier est ainsi bornée par $(T+T'+T'^2+T'^3) \times U_a$. Si l'on choisit $U_a = 512$ octets, $T = 10$ pour la table primaire, et $T' = 128$ pour les suivantes, la taille de fichiers peut atteindre $(10+128+128^2+128^3) \times 512$ octets, soit de l'ordre d'un gigaoctets.

L'accès aux T premiers blocs est privilégié, puisqu'il est direct ; les T' blocs suivants sont accessibles grâce à une indirection ; les T'^2 suivants grâce à deux indirections ; les T'^3 suivants grâce à trois indirections.

3.4 Cas de plusieurs fichiers

Tant que l'on considère le cas d'un seul fichier, la table d'implantation primaire peut être placée à une position fixe sur le disque. Si l'on veut maintenant stocker plusieurs fichiers selon la même méthode, chacun doit disposer de sa table d'implantation, elle-même stockée sur le disque. Il faut donc établir une correspondance entre le fichier et l'adresse physique de sa table d'implantation.

A cet effet, on réserve sur le disque une zone dite *des descripteurs de fichiers*, de taille fixe, et permettant le stockage de n tables d'implantation. Le nombre de fichiers stockables est ainsi borné par n, une fois pour toutes (lors du formatage du disque). Un fichier est désigné par un numéro, et l'adresse physique de son descripteur est obtenue comme un décalage par rapport à l'adresse physique 0.

Le nombre de descripteurs prévu au formatage est donc le nombre maximum de fichiers représentables sur le disque. Selon la nature de l'ensemble des fichiers à représenter, ce paramètre peut être choisi plus ou moins grand.

3.5 Allocation de blocs

Les différents fichiers présents sur un disque et représentés de manière dispersée occupent des blocs de taille égale à l'unité d'allocation, répartis de manière quelconque. Lors d'une modification de l'ensemble des fichiers (création, supression, augmentation...), il faut pouvoir : 1) déterminer l'adresse physique d'un bloc *libre* ; 2) déclarer qu'un bloc n'est plus utilisé, de manière à permettre son utilisation ultérieure.

C'est un problème général de gestion de mémoire. Il y a essentiellement 2 techniques. La première consiste à maintenir une table de marquage des blocs occupés. La deuxième consiste à maintenir une liste chaînée de blocs libres.

3.6 Redondance des informations

Tout fichier est découpé en blocs répartis de manière quelconque sur le disque, la structure du fichier est donc entièrement reflétée par les tables d'implantation, elles-mêmes accessibles par un unique point d'entrée situé à une position fixe : le descripteur de fichier. Si l'on endommage ce descripteur, tout est perdu. D'autres informations présentes sur le disque sont particulièrement critiques : l'unité d'allocation utilisée pour tous les fichiers du disque ; la table de marquage des blocs libres (ou la liste chaînée de blocs libres) ; etc.

Bien qu'il soit impossible de garantir complètement la préservation des informations du disque en toutes circonstances, certaines informations particulièrement critiques sont regroupées et répliquées. En profitant de la structure d'un disque en *cylindres* (Cf. Chapitre 17), on dispose le paquet des informations critiques à plusieurs emplacements d'adresses physiques prédéfinies, disposés en spirale verticale : ainsi deux exemplaires situés sur des plateaux différents correspondent à des positions différentes du bras. Si un problème physique provoque l'atterrissage des têtes de lecture/écriture, il existe toujours des exemplaires à l'abri.

3.7 Résumé de l'occupation du disque

En tenant compte des diverses considérations énoncées jusque là, l'ensemble des informations présentes sur un disque est donc constitué de :

- Plusieurs exemplaires du groupe des informations critiques (unité d'allocation, nombre de blocs de cette taille disponibles sur le disque, nombre maximum de fichiers). Les informations du groupe sont stockées de manière contiguë, et l'ensemble ne tient pas nécessairement dans un bloc, tel qu'il est défini pour la gestion des fichiers. La taille du groupe est prédéfinie (elle n'a pas à être inscrite sur le disque, ce qui repousserait le problème). Les adresses physiques de début des positions de recopie du groupe sont également prédéfinies.
- Également à une position fixe prédéfinie sur le disque, mais en un seul exemplaire, on trouve un marquage des descripteurs de fichiers occupés
- Une zone contiguë du disque est réservée aux descripteurs de fichiers. Cette zone commence à une adresse prédéfinie, et les différents descripteurs sont accessibles par des décalages. La zone des descripteurs est de taille fixe. La taille d'un descripteur est fixe et prédéfinie.
- Une autre zone contiguë du disque est réservée à la table de marquage des blocs occupés. Il faut 1 bit par bloc du disque utilisable pour les fichiers. La table de marquage est d'autant plus volumineuse que l'unité d'allocation est petite. A la limite, il faut 1 bit par unité d'accès physique.

– Tout l'espace restant, vu comme un ensemble de blocs de taille égale à l'unité d'allocation, est disponible pour le stockage dispersé des fichiers.

4. Noms externes et autres informations attachées aux fichiers

4.1 Désignation de fichiers par des noms externes

Pour l'instant les différents fichiers représentés de manière dispersée sur un disque sont identifiables par un entier de l'intervalle $[1, n]$, où n est le nombre maximum de descripteurs fixé au formatage.

L'une des fonctions du système de gestion de fichiers est de permettre à l'utilisateur une désignation des fichiers par des *noms externes*. Le SGF doit donc gérer une association nom/numéro, c'est-à-dire un ensemble de couples comportant une chaîne de caractères (de taille bornée) et un entier. Cet ensemble de couples constitue le fichier *catalogue* du disque, et doit correspondre à un numéro de descripteur fixe, par exemple 1.

L'accès à un fichier de nom externe X comporte ainsi 3 étapes :

– Lecture du descripteur numéro 1 (adresse physique fixe), qui donne la table d'implantation primaire du fichier catalogue.

– Parcours des blocs du fichier catalogue, pour chercher un couple <X,i>. Notons ici que le nom de fichier se comporte comme une *clé* de la relation entre noms externes et numéros de descripteurs, c'est-à-dire qu'à un nom de fichier donné ne peut correspondre qu'un descripteur.

– Lecture du descripteur numéro i. On est ramené au cas des fichiers désignés directement par des numéros.

> **Remarque :** L'introduction de la relation entre noms externes et numéros de descripteur, représentée dans le fichier catalogue, permet d'associer plusieurs noms externes au même numéro de descripteur. C'est le mécanisme de lien dit *physique* offert par de nombreux systèmes d'exploitation, qui permet le partage d'informations stockées de manière unique, et visibles sous des noms externes différents. Noter toutefois que, dans ce cas, la *suppression* de fichier peut se réduire à la suppression d'*une* association nom/numéro de descripteur. Les blocs du fichier et son descripteur ne peuvent être effectivement libérés que lorsqu'aucun couple du catalogue n'y fait plus référence. Pour éviter un parcours exhaustif du catalogue qui permettrait de s'assurer de cette propriété lors de la suppression d'un couple, on maintient en général un *compteur de références* dans le descripteur : la suppression effective du fichier n'est réalisée que lorsque ce compteur vaut 1 ; dans les autres cas le compteur est décrémenté.

4.2 Structure hiérachique de l'ensemble de fichiers

La plupart des systèmes de gestion de fichiers actuels gèrent des arborescences de catalogues. Un catalogue peut contenir deux types de fichiers : des catalogues sur les noeuds internes et les fichiers ordinaires sur les feuilles de l'arbre.

Le nom externe complet d'un fichier décrit le chemin depuis la racine de l'arbre vers la feuille considérée. Dans la syntaxe de la plupart des interprètes de commandes UNIX, `/users/info1/arthur/tp.c` est le nom complet d'un fichier situé au niveau 4 sous la racine. Les noms intermédiaires `users`, `info1`, `arthur` correspondent à des sous-arbres, qu'on appelle *répertoires* (*directory* en anglais) du SGF.

L'accès à ce fichier suit les étapes :

- Lecture du descripteur numéro 1, qui donne la table d'implantation du fichier catalogue de la racine.
- Parcours des unités d'allocation de ce fichier, en recherchant un couple $<$ `users`, $i >$.
- Lecture du descripteur numéro i, qui donne la table d'implantation du fichier catalogue de la racine du sous-arbre `users`.
- Parcours des unités d'allocation de ce fichier, en recherchant un couple $<$ `info1`, $j >$.
- Lecture du descripteur numéro j, qui donne accès au catalogue du sous-arbre `users/info1` et recherche d'un couple $<$ `arthur`, $k >$
- Finalement, lecture du descripteur numéro k, qui donne accès au catalogue du sous-arbre `users/info1/arthur`, et recherche d'un couple $<$ `tp.c`, $\ell >$
- ℓ est le numéro de descripteur du fichier cherché.

4.3 Autres informations attachées aux fichiers

Outre le mécanisme de désignation par nom externe, la plupart des systèmes d'exploitation permettent d'associer aux fichiers des informations comme : la date de création (voire la date de dernière modification, la date de dernière consultation, etc.) ; le propriétaire (dans un contexte multi-utilisateurs) et les droits d'accès en lecture et écriture des utilisateurs non propriétaires, etc.

Ces informations font partie du descripteur de fichier.

5. Etude de quelques fonctions du système de gestion de fichiers

5.1 Formatage logique

Rappelons que nous avons défini au chapitre 17 la notion de *formatage physique* d'un disque, qui détermine les pistes et secteurs, et établit la correspondance entre l'adressage par un couple (numéro de piste, numéro de secteur dans la piste), et l'adressage global par numéro d'unité d'accès.

Une fois cette structuration du disque en unités d'accès effectuée, le SGF est responsable du formatage *logique*, qui permet de voir le disque comme un ensemble de blocs de taille égale à l'unité d'allocation.

Les paramètres du formatage logique sont : l'unité d'allocation et le nombre maximum de fichiers représentables. L'opération de formatage installe les multiples exemplaires du groupe des informations critiques, et les valeurs initiales de la table de marquage des descripteurs occupés et de la table de marquage des blocs occupés. Elle construit également le fichier catalogue de la racine du système de fichiers, qui est initialement vide. Le descripteur de ce fichier est le descripteur de numéro 0.

5.2 Gestion d'une information répertoire courant

Une fonctionnalité usuelle du système de gestion de fichiers consiste à fournir une notion de répertoire courant. Nous verrons au chapitre 20 que cette fonction est indispensable à la réalisation de la notion de répertoire courant des interprètes de commandes.

Les fonctions offertes peuvent être décrites par :

RepCour : une séquence de caractères
SetRepCour : une action (la donnée : une séquence de caractères)
  SetRepCour(R) : RepCour ⟵ R
GetRepCour : ⟶ une séquence de caractères
  GetRepCour : RepCour

Il est intéressant de se poser la question de la durée de vie de l'information RepCour. En effet, elle est nécessairement supérieure à celle des appels des primitives offertes par le SGF, dont SetRepCour et GetRepCour, mais aussi toutes les opérations de lecture/écriture décrites au paragraphe 5.4 ci-dessous.

Pour tous les accès au système de gestion de fichiers paramétrés par un nom externe, on peut convenir que les les noms de fichiers qui ne commencent pas par un caractère slash sont des noms *relatifs* au répertoire courant. Pour obtenir des noms absolus, il suffit de les préfixer par le contenu de la variable RepCour. Cela suppose évidemment que le contenu de la variable RepCour est un nom absolu de répertoire, décrivant un chemin depuis la racine de l'arbre des fichiers jusqu'à un noeud catalogue.

Noter que, dans le cas général d'un système multitâches, le répertoire courant est une notion locale à un exemplaire d'interprète de commandes (Cf. Chapitre 20) en cours d'exécution, et il peut y en avoir plusieurs en parallèle. Dans ce cas, une seule variable de répertoire courant dans le SGF ne suffit plus, et il y en a en réalité une par *processus* (Cf. Chapitre 23).

5.3 Création et suppression de fichier

A chaque fichier est associé un ensemble d'informations, permettant d'accéder à son contenu, regroupées dans une structure appelée *descripteur physique* du fichier ou *noeud d'information*, ce terme faisant référence aux *i-nodes* du système UNIX. On y trouve en particulier la taille du fichier et sa table d'implantation primaire.

NoeudInfo : le type <
    type : (catalogue, fichier-ordinaire) { *type du fichier* }
    taille : un entier ; { *taille du fichier en nombre d'octets* }
    tab-imp : un tableau sur [0..99] d'AdPhysiques
    { *potentiellement : adresses de début de 100 blocs du fichier, chacun de la taille de l'unité d'allocation UAlloc.* }
>

Les octets formant le contenu du fichier sont stockés sur les unités d'accès d'adresses physiques :
`tab-imp[0]+0`, `tab-imp[0]+1`, ..., `tab-imp[0]+UAlloc −1`,
`tab-imp[1]+0`, `tab-imp[1]+1`, ..., `tab-imp[1]+UAlloc −1`,
...

Le nombre d'unités d'accès effectivement occupées est déterminé par la taille du fichier en octets et la taille de l'unité d'accès en octets.

L'ensemble des fichiers présents sur le disque est représenté dans le *répertoire* où un fichier est identifié par son nom et son accès indiqué par le numéro de son NoeudInfo. Une entrée de répertoire peut être décrite par :

EntreeRep : le type (nom : un texte ; NoNoeudInfo : un entier)

Le répertoire est lui-même un fichier. Les NoeudInfo sont rangés de façon contiguë au début du disque, en commençant par celui du répertoire.

La *création* d'un fichier demande l'allocation d'une entrée dans le catalogue du répertoire courant et la mise à jour de cette entrée avec le nom du fichier et un numéro de noeud d'information libre. L'ajout au fichier répertoire est une écriture de fichier, décrite ci-dessous.

La *suppression* demande la libération de toutes les unités d'accès spécifiées dans la table d'implantation du descripteur et la libération de l'entrée correspondante dans le catalogue. La suppression d'une entrée de catalogue est une modification du fichier catalogue.

5.4 Ouverture, lecture et écriture de fichiers

Avant tout accès en lecture ou en écriture à un fichier il est nécessaire de réaliser l'opération d'*ouverture* du fichier. Au niveau utilisateur un fichier est

désigné par un nom ; l'ouverture consiste à donner accès à toutes les informations stockées dans le fichier.

Ensuite on utilise les actions de lecture et d'une action d'écriture d'une unité d'accès fournies par le pilote de périphérique associé au disque (Cf. Paragraphe 3. du chapitre 17).

5.4.1 Description d'un fichier pour les utilisateurs

Les programmes utilisateurs du SGF manipulent les fichiers par l'intermédiaire d'une structure de données qui contient toutes les informations permettant d'accéder au fichier ; outre les informations concernant l'accès physique, tirées du descripteur physique (ou noeud d'information) du fichier, il est nécessaire si l'on veut réaliser un accès séquentiel, de mémoriser la position atteinte lors du dernier accès, le mode d'utilisation autorisé, etc. Cette structure de données est installée par le système de gestion de fichier en mémoire lors de l'ouverture du fichier et recopiée sur le disque lors de la fermeture. On obtient :

```
Descripteur : le type <
        type : (catalogue, fichier-ordinaire) { type du fichier }
        taille : un entier ; { taille du fichier en nombre d'octets }
        tab-imp : un tableau sur [0..99] d'AdPhysiques
        { potentiellement : adresses de début de 100 blocs du fichier, chacun de la
        taille de l'unité d'allocation UAlloc. }
        offset : un entier ;
            { pointeur courant dans le fichier : c'est un décalage par rapport au
            début, en nombre d'octets. }
            mode : (lect, ecr) ; { accès en lecture ou en écriture }
            { Autres informations comme par exemple les droits d'accès à un utili-
            sateur ou à un autre }
        >
```

5.4.2 Ouverture d'un fichier

La fonction d'ouverture d'un fichier a pour rôle de fournir aux programmes utilisateurs un accès au fichier identifié par son nom.

Il s'agit de chercher dans le catalogue le nom du fichier. Pour cela il faut lire le noeud d'information du fichier catalogue, puis parcourir les blocs du catalogue (via sa table d'implantation) à la recherche de la chaîne de caractères décrivant le nom du fichier.

Lorsque l'on a trouvé le nom du fichier on récupère le numéro de son noeud d'information. Les noeuds d'information sont stockés à une adresse fixe du disque ; on calcule alors l'adresse physique du noeud d'information du fichier.

L'étape suivante consiste à lire ce noeud d'information et à l'utiliser pour mettre à jour le descripteur du fichier. Le contenu du noeud d'information est

recopié et les informations non liées à l'implantation du fichier sur disque sont initialisées.

5.4.3 Lecture et écriture de fichier

La lecture demande au préalable une ouverture du fichier en mode lecture. L'utilisateur dispose alors du descripteur dans lequel il trouve à la fois les informations d'accès physique et la taille du fichier. Lors d'une lecture séquentielle, les blocs, dont on trouve les adresses physiques de début dans la table d'implantation, sont lus les uns après les autres dans un tampon en mémoire. Le déplacement par rapport au début du fichier doit être maintenu à jour.

Une opération d'écriture est similaire. Le déplacement par rapport au début étant égal au nombre d'octets du fichier, l'écriture séquentielle a lieu à la suite des octets déjà mémorisés dans le fichier.

Un éditeur de texte qui réalise à la fois des opérations de lecture et d'écriture travaille dans un tampon mémoire contenant tout ou une partie du fichier. Au lancement le fichier est ouvert puis lu dans le tampon, une opération explicite permet à l'utilisateur de mémoriser le contenu de son fichier, c'est-à-dire de l'écrire sur disque.

5.5 Sauvegarde, restauration, reconstitution des informations d'un disque

La sauvegarde des informations d'un disque (sur une bande magnétique par exemple) peut prendre deux formes, qui correspondent aux commandes `dump` et `tar` des systèmes de la famille UNIX. Si l'on veut sauvegarder un sous-ensemble donné de fichiers, il est nécessaire d'accéder au disque en suivant les informations fournies par les noms des fichiers et les tables d'implantation de leurs descripteurs. L'accès est alors similaire à une série de lectures.

Si, en revanche, on veut sauvegarder entièrement le disque, il suffit d'oublier momentanément la structure imposée par le formatage logique, et de recopier séquentiellement tous les secteurs du disque sur une bande, dans l'ordre des adresses physiques. C'est beaucoup plus rapide que la sauvegarde fichier par fichier, qui réalise finalement le même travail, mais en compliquant les accès. C'est la commande `dump`.

Finalement il existe en général une fonction du SGF qui permet de profiter de la redondance d'information sur le disque pour tenter quelques réparations, lorsqu'un problème physique endommage un ou plusieurs secteurs. C'est la commande `fsck` (pour *file system check*) des systèmes de la famille UNIX.

5.6 Désignation des périphériques à travers le SGF

Dans le système UNIX les périphériques sont nommés de la même façon que s'il s'agissait de fichiers : `/dev/...`. La convention de nommage et le type

associé à ce genre de fichier permet de les distinguer des autres.

La procédure d'ouverture de fichier analyse le nom et se branche sur la procédure décrite plus haut s'il s'agit d'un fichier standard, ou sur une procédure spéciale d'accès au périphérique. Les procédures de lecture/écriture font de même.

Chapitre 20

Démarrage du système, langage de commandes et interprète

Nous venons de construire un système logiciel et matériel simple, en connectant un processeur et de la mémoire (chapitre 15), en assurant la connexion de cet ensemble au monde extérieur grâce aux circuits d'entrées/sorties (chapitres 16 et 17), et en construisant le système de gestion de fichiers (chapitre 19) pour le stockage des informations à longue durée de vie. Que manque-t-il encore pour en faire un ordinateur, au sens défini chapitre 1 ? Il manque un protocole de dialogue entre cette machine et un utilisateur humain. Sans intervention d'un utilisateur humain qui commande des calculs de la machine, écrit de nouveaux programmes, les compile et les exécute, etc., le système matériel et logiciel que nous avons décrit, aussi sophistiqué soit-il, ne permet d'exécuter des programmes que s'ils sont inscrits en mémoire morte.

Le dialogue entre l'utilisateur humain et le système matériel suppose encore une fois la définition d'un *langage* précis et non ambigu, que nous appellerons *langage de commandes*. Ce langage est nécessairement *interprété*, puisqu'il est saisi de manière interactive par l'utilisateur humain.

Le programme interprète du langage de commandes (ou, plus simplement, l'*interprète de commandes*) est actif depuis le lancement du système jusqu'à son arrêt ; il lit au clavier une ligne de commande, analyse ce texte pour déterminer quelle commande il représente, avec quels paramètres. Dans un interprète textuel du système UNIX, on écrit par exemple : `more toto.c` ; `more` est le nom d'un programme résidant quelque part dans le système de gestion de fichiers, et `toto.c` est le paramètre de la commande ; ici c'est également un nom de fichier. L'interprète de commandes doit accéder au SGF pour lire le fichier exécutable du programme `more`, et l'installer en mémoire pour exécution en lui passant les paramètres indiqués dans la ligne de commandes. Lorsque le programme `more` se termine, on retourne dans l'interprète de commandes, qui est prêt à lire une nouvelle ligne.

Pour terminer la présentation de l'architecture logicielle et matérielle d'un système simple, il nous reste à décrire comment ce système simple *démarre*.

Nous commençons, paragraphe 1., en décrivant le démarrage du système. Le paragraphe 2. définit exactement le mécanisme de base nécessaire à tout interprète de commandes, le chargeur/lanceur *de programmes ; ce mécanisme est également utilisé au chapitre 23 pour la création de processus. Le paragraphe 3. donne le programme type d'un interprète de commandes textuel. Nous terminons, au paragraphe 4., en évoquant les langages de commandes graphiques (ou icôniques) et les langages de commandes à structures de contrôle.*

1. Démarrage du système

Nous considérons un système simple qui comporte de la mémoire morte, de la mémoire vive et un disque dur.

Au démarrage du système, c'est-à-dire à la mise sous tension du dispositif matériel et logiciel, on peut supposer que la réalisation matérielle charge la valeur 0 dans le compteur programme PC. Le processeur commence donc à interpréter le contenu de la mémoire à l'adresse 0. Cette adresse correspond à de la mémoire morte, qui contient l'*amorce* du système.

Cette amorce est constituée du code de la procédure de démarrage ; d'un pilote de disque rudimentaire (c'est-à-dire un ensemble de procédures d'accès au disque) et d'un embryon de système de gestion de fichiers capable de retrouver sur le disque une image du système d'exploitation complet et de la recopier en mémoire vive.

Le système complet comporte le système de gestion de fichiers complet (décrit au chapitre 19), les pilotes de périphériques complets (décrits au chapitre 17) dont le pilote du disque, le code de la procédure de chargement/lancement de programmes, que nous étudions en détail au paragraphe 2. ci-dessous.

1.1 Première étape du démarrage du système

La procédure de démarrage réalise les étapes suivantes :

- Appel aux fonctions de l'embryon de SGF (qui appellent elles-mêmes les fonctions du pilote de disque rudimentaire), pour localiser et lire sur le disque l'image du système d'exploitation complet.
- Recopie de ce système en mémoire vive, à une adresse prédéfinie.
- Installation de la structure de données en mémoire vive, nécessaire à la vie du système (zone libre pour les programmes, zone occupée par le système, adresses qui délimitent ces zones).
- Initialisation du registre SP du processeur utilisé comme pointeur de pile.

La figure 20.1-(a) donne le contenu de la mémoire et du registre pointeur de pile juste après le démarrage du système, c'est-à-dire après le déroulement du code d'amorçage décrit ci-dessus.

La structure de mémoire décrite ici n'est pas nécessairement implantée telle quelle dans les systèmes réels. Toutefois les informations manipulées et les arguments avancés pour choisir cette structure sont réalistes. On peut considérer que les systèmes réels ne sont que des variantes de la structure décrite ici.

A l'une des extrémités de la mémoire, on trouve rassemblés des programmes et des données qui doivent avoir la durée de vie du système : le programme et les données du système de gestion de fichiers, des pilotes de périphériques, la procédure de chargement/lancement de programmes pris dans des fichiers.

Parmi les données du système on trouve les trois adresses DébutMEM, FinMEM et DébutMEMlibre qui délimitent les portions disponibles de la mémoire : la zone entre DébutMEM et DébutMEMlibre est occupée par les données et programme système, et n'est donc pas disponible pour les programmes utilisateur. La zone entre DébutMEMlibre et FinMEM est disponible pour les programmes utilisateur. La base de la pile est en FinMEM, et elle progresse vers la zone système, jusqu'à la borne DébutMEMlibre.

Le registre pointeur de pile du processeur est initialisé à la valeur FinMEM.

1.2 Deuxième étape du démarrage

Une fois cette structure de mémoire installée, il ne reste plus qu'à lancer un programme interprète d'un langage de commandes, qui sera actif durant toute la durée de vie du système, et fera appel à la procédure de chargement/lancement de programme pour exécuter les programmes requis par l'utilisateur à la ligne de commandes. Pour installer le programme interprète de commandes lui-même, la procédure de démarrage utilise également la procédure de chargement/lancement.

1.3 Fin du système

Nous verrons au paragraphe 3. que l'une des commandes de tout langage de commandes est terminer, qui provoque la terminaison du programme d'interprétation, et donc le retour dans la procédure de démarrage du système. Celle-ci peut alors appeler des procédures du système de gestion de fichiers qui permettent de placer le système de fichiers dans un état cohérent ; on peut alors éteindre la machine.

2. Mécanisme de base : le chargeur/lanceur

Au fil des chapitres 4, 5, 12, 13 et 18 nous avons étudié les transformations successives d'un fichier texte de programme en langage de haut niveau, jusqu'au fichier exécutable qui lui correspond. Nous avons, au chapitre 18, anticipé sur le *chargement* du programme en mémoire, en étudiant les algorithmes de translation d'adresses.

Nous étudions ici le mécanisme complet de chargement et lancement de programme résidant dans un fichier du SGF, en précisant en particulier comment et où se fait l'allocation effective de mémoire. Noter que l'éditeur de textes, le compilateur et l'assembleur qui ont servi à produire le programme exécutable à charger sont eux-mêmes des programmes résidant dans le SGF sous forme de code exécutable, et ont dû, en leur temps, être chargés en mémoire pour exécution.

Toute la difficulté réside dans le fait que la procédure de chargement/lancement ne se comporte pas tout à fait comme une procédure standard, dont nous avons étudié le codage au chapitre 13. Pour comprendre la suite, il est toutefois nécessaire d'avoir bien assimilé le principe du codage des blocs imbriqués à l'aide d'une pile.

Supposons qu'un programme P veuille lancer un programme Q. Très schématiquement, le fonctionnement est le suivant :

1) La procédure active du programme P appelle la procédure de chargement/lancement avec comme paramètres : le nom du fichier dans lequel se trouve le code compilé du programme Q à lancer et les paramètres éventuels nécessaires au programme Q lors de son lancement.

2) La procédure de chargement/lancement *alloue* une zone de mémoire nécessaire pour l'installation en mémoire vive du fichier objet de Q (zones TEXT, DATA et BSS), installe le programme et réalise la translation d'adresses (Cf. Chapitre 18, Figure 18.9). Attention, cette zone de mémoire est nécessaire pour l'installation du code du programme lui-même, et n'a rien à voir avec la zone de pile nécessaire pour les variables locales de ses procédures lors de son exécution.

3) La procédure de chargement/lancement libère la place qu'elle occupe dans la pile et *se transforme* en la procédure principale du programme lancé. Nous détaillons ce mécanisme au paragraphe 2.3. Lorsque le programme lancé Q se termine, le contrôle revient donc directement dans le programme P, sans repasser par le contexte intermédiaire de la procédure de chargement/lancement.

Nous précisons ci-dessous l'interface de la procédure de chargement/lancement, et son déroulement.

2.1 Paramètres et résultat de la procédure de chargement/lancement

2.1.1 Les données

Dans le cas général, la procédure de chargement/lancement a pour paramètres données : une chaîne de caractères qui donne le nom du fichier qui contient le programme à charger ; une suite de chaînes de caractères qui constituent les paramètres à donner à ce programme.

La première utilisation de la procédure de chargement/lancement est faite

par la procédure de démarrage pour installer l'interprète de commandes. Dans ce cas le paramètre nom de fichier est le nom de l'interprète de commandes standard (qui peut être une donnée inscrite dans le code de démarrage).

Toutes les utilisations ultérieures sont le fait de l'interprète de commandes lui-même. Dans ce cas le nom du programme à charger et les paramètres qui doivent lui être transmis sont donnés par l'utilisateur dans sa ligne de commandes. Par exemple, dans le langage de commandes `csh` des systèmes UNIX, la commande `ls -l` doit provoquer le chargement en mémoire du code exécutable de `ls`, en lui transmettant le paramètre additionnel `-l`. (`ls` est la commande utilisée pour afficher la liste des noms de fichiers du répertoire courant, et `-l` est une directive de présentation de cette liste). Noter que les paramètres sont lus au clavier par l'interprète, sous forme de chaînes de caractères. Le programme interprète n'a aucun moyen de décoder ces chaînes de caractères pour y retrouver par exemple des nombres. Le programme chargé reçoit donc des chaînes, et les décode lui-même.

Cela permet de comprendre le profil de la fonction `main` des programmes écrits en C, qui joue le rôle de programme principal :

```
int main (int argc, char *argv[])
```

où `argc` est le nombre de mots apparaissant sur la ligne de commandes (y compris le nom de la commande elle-même), et `argv` est un tableau de chaînes de caractères qui contient les différents mots de la ligne de commande.

2.1.2 Le résultat

Le résultat de la procédure de chargement/lancement est un code de retour, c'est-à-dire une information qui tient dans un registre du processeur.

Les valeurs possibles du code de retour de la procédure de chargement/lancement sont à envisager dans deux situations :

- La procédure échoue, et le programme demandé ne peut être chargé ; le programme demandé n'existe pas, ou bien le fichier n'est pas lisible, ou bien il n'y a pas assez de place libre pour installer le programme en mémoire, etc.
- Le programme a pu être lancé, et s'est terminé. Il peut avoir lui-même rendu un code de retour.

Comme évoqué ci-dessus, la procédure de chargement/lancement a un comportement très particulier : lorsque le chargement du programme réussit, la procédure de chargement *se transforme* en ce programme, par manipulations directes des informations présentes dans la pile. Lorsque le programme lancé se termine, le contrôle revient directement dans l'appelant du chargeur. La procédure de chargement n'a donc de résultat que lorsqu'elle échoue ; le résultat entier code la cause de l'erreur, parmi celles suggérées ci-dessus.

Lorsque le chargement réussit, l'appelant du chargeur reçoit directement le résultat du programme chargé.

2.1.3 Passage des paramètres

Nous détaillons le fonctionnement de la procédure de chargement/lancement, et en particulier la structure de ses paramètres et de son résultat, dans le cas où les appels de procédures sont réalisés directement par la pile (Cf. Chapitre 13, le schéma de codage des appels de procédures dans un langage d'assemblage style 68000). Nous utilisons le mécanisme de retour des résultats de fonction simples décrit au chapitre 13.

Dans le cas d'appels de procédures réalisés par un mécanisme de fenêtre de registres (comme pour le processeur SPARC) ou de toute autre manière, la solution décrite ci-dessous doit être adaptée. Les principes de base restent les mêmes, toutefois.

Un exemple de contenu de la pile tel qu'il doit être installé par l'appelant de la procédure de chargement/lancement est donné figure 20.1-(b).

2.2 Allocation de mémoire pour l'installation des programmes à charger

2.2.1 Cas général

Dans le cas général d'un système multi-utilisateurs ou simplement multitâches (Cf. Partie VI), les besoins en zones mémoire pour le chargement des programmes surviennent dans un ordre quelconque. Il est tout à fait possible d'observer un comportement du système dans lequel les chargements et terminaisons de programmes A et B sont entrelacés (début de A, début de B, fin de A, fin de B).

Il est donc nécessaire de réaliser une allocation mémoire dispersée générale, dont l'interface est décrite au chapitre 4, paragraphe 4. Les fonctions d'allocation et libération font partie du système de base, leur code (Zones TEXT, DATA et BSS) est présent en mémoire, dans la zone système, pendant toute la durée de vie du système ; elles n'ont donc pas à être elles-mêmes chargées en mémoire, ce qui repousserait le problème de l'allocation. Ces deux fonctions mettent à jour les informations qui décrivent l'occupation de la mémoire à un moment donné (tableau de marquage de zones libres, ou liste de blocs chaînés, etc.). Ces informations sont également présentes en mémoire vive pendant toute la durée de vie du système.

2.2.2 Cas du système simple

Dans le cas que nous allons détailler, nous supposerons que le système n'est ni multi-utilisateurs, ni multitâches. Dans ce cas les besoins en zones mémoire pour le chargement des programmes suivent le schéma dernier alloué/premier libéré. En effet, la procédure d'initialisation charge l'interprète, qui peut charger des programmes utilisateur, lesquels peuvent eux-mêmes charger d'autres programmes, mais tout cela se déroule comme une suite d'appels de procédures

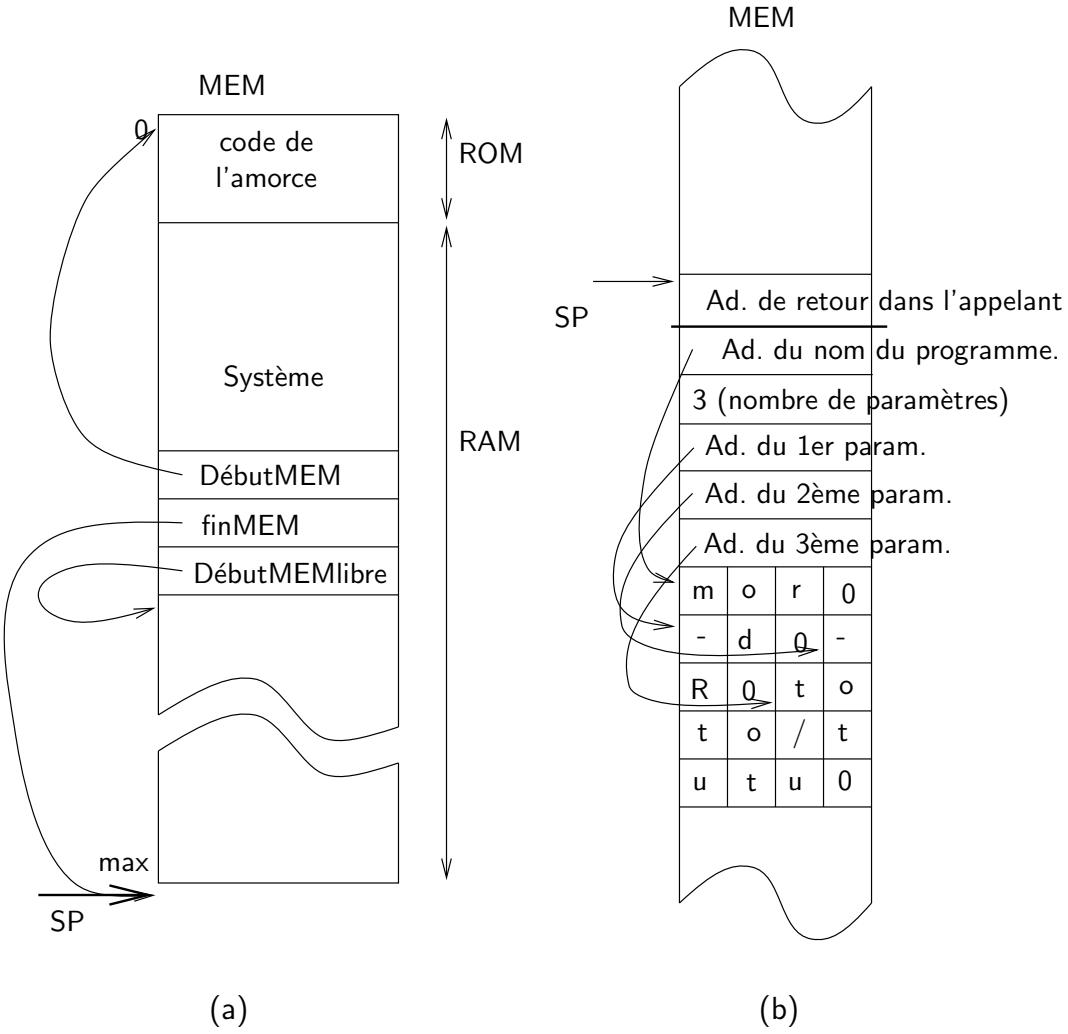


FIG. 20.1 – (a) : Contenu de la mémoire après la première étape du démarrage.
(b) : Contenu détaillé de la pile installé par l'appelant de la procédure de chargement/lancement : le nom du programme est `mor`, les paramètres à lui transmettre sont `-d`, `-R` et `toto/tutu`.

(malgré le comportement atypique de la procédure de chargement décrit plus haut).

La mémoire disponible pour le chargement des programmes peut donc être gérée en pile. Attention, il ne faut pas confondre cette zone de mémoire vive gérée en pile, et la zone appelée *pile d'exécution* que nous avons décrite au chapitre 13, qui sert aux variables locales et paramètres des blocs du programme en cours d'exécution.

L'adresse DébutMemLibre joue le rôle de pointeur de pile de la zone de mémoire utilisée pour le chargement du code des programmes. Ce pointeur (mémorisé dans une case mémoire et/ou dans un registre dédié du processeur), est initialisé par la procédure d'initialisation, comme mentionné plus haut. Pour réserver une zone pour le chargement d'un programme, la procédure de chargement doit déplacer ce pointeur vers les adresses plus grandes (vers le bas sur la figure 20.1-(a)).

Des considérations de symétrie voudraient que cette même procédure de chargement s'occupe de replacer le pointeur DébutMemLibre à sa valeur précédente, lorsque le programme qui a été chargé et lancé se termine, et que la zone qu'il occupait peut être récupérée. Le comportement particulier de la procédure de chargement nous en empêche : lorsque le chargement réussit, on ne revient pas dans la procédure de chargement (ce qui aurait permis de déplacer DébutMemLibre), mais directement dans l'appelant du chargeur. Ce comportement est assuré par trois points particuliers de l'algorithme détaillé ci-dessous, que nous repérons par les lettres α, β et γ.

La solution que nous décrivons ci-dessous consiste à faire en sorte que tout programme (l'interprète chargé le premier, les programmes qu'il charge, les programmes chargés par ceux-là, etc.) dispose, dans la zone de pile où se trouvent ses paramètres, d'une adresse mémoire qui lui permet de repérer la zone occupée par ses zones TEXT, DATA et BSS.

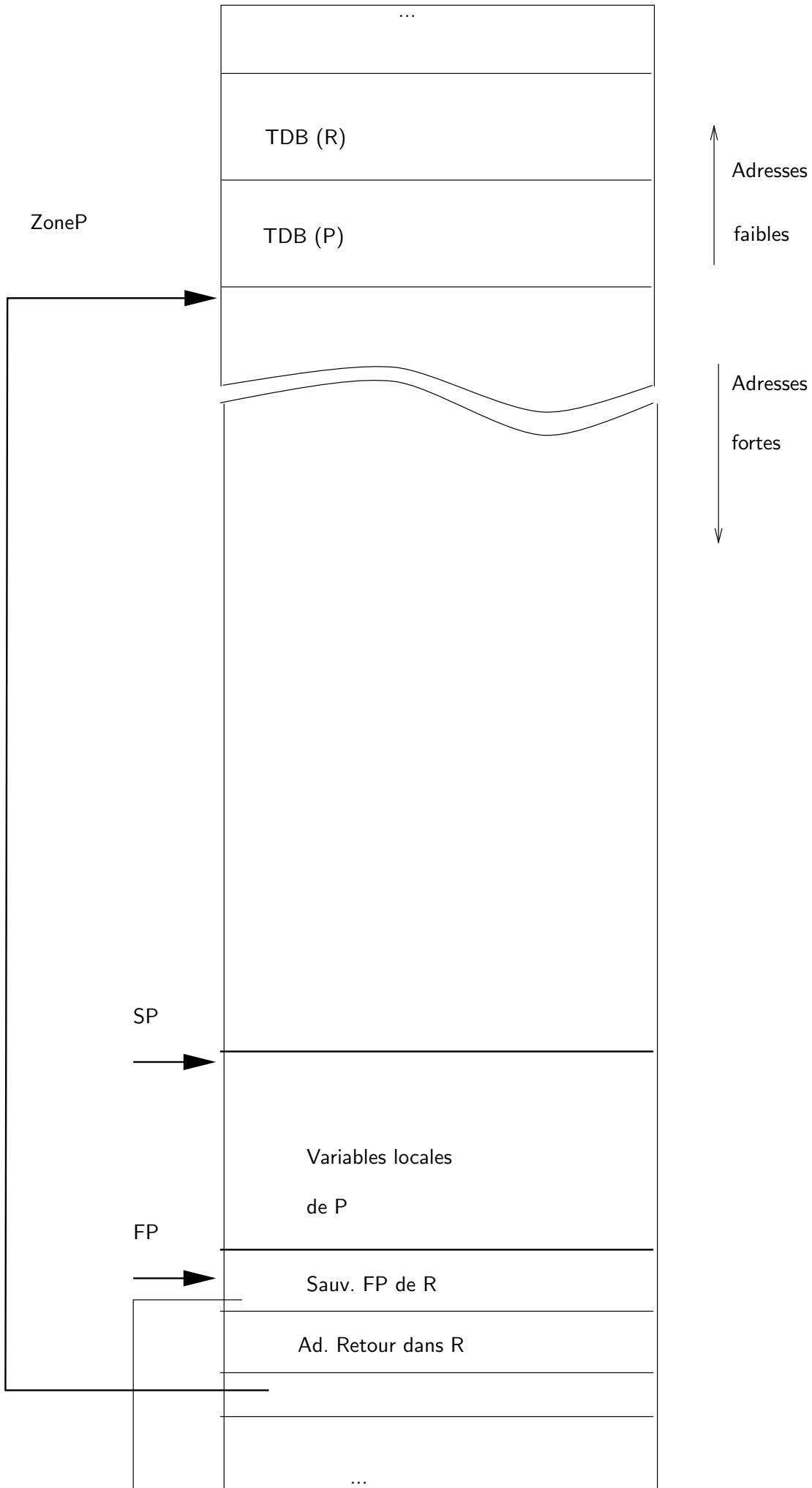
2.3 Déroulement du chargement/lancement

La figure 20.2 illustre les états successifs de la mémoire (pile d'exécution et zone occupable par le code et les données des programmes lancés), depuis une situation où un programme Ps'apprête à lancer un programme Q, jusqu'à la situation où le programme Q a été installé et est en cours d'exécution, prêt à charger et installer lui-même un autre programme.

2.3.1 Etat de la mémoire quand un programme s'apprête à en charger et lancer un autre

La figure 20.2-(a) illustre l'invariant à respecter : un programme P est en cours d'exécution, et s'apprête à demander le chargement/lancement d'un programme Q.

Les zones TEXT, DATA et BSS du programme P sont en mémoire vive,


...
TDB (R)
TDB (P)
ZoneP
Adresses
faibles
Adresses
fortes
SP
Variables locales
de P
FP
Sauv. FP de R
Ad. Retour dans R
...

vers les adresses faibles. Le contexte dynamique de P occupe une portion de pile située vers les adresses fortes, entre les adresses pointées par les registres SP (pointeur de pile) et FP (pointeur de base d'environnement).

Au-dessus (vers les adresses faibles) de la zone occupée par les zones TEXT, DATA et BSS de P, on trouve les zones TEXT, DATA et BSS des programmes qui ont lancé P (R sur la figure), dont la procédure d'initialisation du système, tout en haut.

En dessous (vers les adresses fortes) de l'adresse pointée par FP, on trouve les environnements des procédures de P appelées par la procédure principale et, encore en dessous, les environnements des procédures du programme qui a lancé P.

Nous décrivons ci-dessous les étapes du chargement/lancement.

2.3.2 Appel de la procédure de chargement/lancement

La figure 20.2-(a) décrit l'état de la mémoire lorsque le programme P s'apprête à charger un programme Q.

La figure 20.2-(b) décrit l'état de la mémoire après les étapes suivantes :

– Appel de la procédure C de chargement/lancement par la procédure du programme P en cours d'exécution : cette procédure empile les paramètres pour C, qui comportent : le nom du fichier contenant le programme Q à lancer, et les paramètres pour Q ; le nom du fichier est placé tout en haut (en dernier). La structure détaillée des paramètres pour C est donnée figure 20.1-(b).
L'appel de C proprement dit empile l'adresse de retour dans P (α) .

– Installation de l'environnement de la procédure de chargement/lancement (voir chapitre 13 et la gestion du lien dynamique) : sauvegarde du pointeur de base de l'environnement de l'appelant dans la pile, et mise en place du pointeur de base de l'environnement de la procédure de chargement ; déplacement du pointeur de pile pour ménager un espace pour les variables locales de la procédure.
Les variables locales de C sont décrites par le lexique suivant :

Taille : un entier { *taille de mémoire nécessaire à l'installation du programme* }
PFichier : un descripteur de fichier { *Cf. Chapitre 19* }
EnTete : une entête de fichier exécutable{ *Cf. Chapitre 18* }

2.3.3 Exécution de la procédure de chargement/lancement

La figure 20.2-(c) illustre l'état de la mémoire après l'étape décrite dans ce paragraphe.

Le code de la procédure de chargement/lancement commence par accéder au paramètre qui donne l'adresse du nom du programme à charger (on suppose que c'est un nom absolu pour l'instant) puis appelle la procédure d'ouverture

de fichier d'après son nom (Cf. Chapitre 19). Les paramètres à lui passer sont : l'adresse du nom de fichier, le mode d'ouverture (ici "lecture"), l'adresse de la variable PFichier. La procédure d'ouverture peut échouer, et rendre un code d'erreur dans un registre ; dans ce cas la procédure de chargement se termine, en conservant ce code de retour (voir dernier point).

Lorsque l'ouverture du fichier s'est bien passée, la variable PFichier est pertinente, et peut servir à réaliser les accès suivants. L'étape suivante est l'appel de la procédure du SGF qui permet de lire une portion du fichier de la taille de EnTete, à partir du début du fichier. Les paramètres à lui passer sont : l'adresse de PFichier, l'adresse de EnTete. Si tout se passe bien, la variable EnTete contient ensuite la description des zones du fichier exécutable, ainsi qu'un marqueur qui indique la nature du fichier. Si ce n'est pas un fichier exécutable, cela constitue un nouveau cas où la procédure de chargement échoue, en rendant un code d'erreur. Si le fichier est bien un fichier exécutable, on peut poursuivre.

Le code de la procédure de chargement/lancement consiste ensuite à calculer la taille de la zone mémoire nécessaire à l'installation du code et des données du programme Q. L'en-tête du fichier exécutable donne les tailles respectives des zones TEXT, DATA et BSS du programme. La variable Taille est affectée à la somme de ces tailles (éventuellement arrondie au multiple de 4 ou de 8 supérieur, si ce n'est pas déjà fait dans le fichier exécutable, pour satisfaire à des contraintes d'alignement en mémoire vive).

Il faut ensuite allouer une zone mémoire pour le programme à lancer et déterminer ainsi *l'adresse de chargement*. Nous avons vu au paragraphe 2.2.2 que la zone de mémoire utilisée pour installer le code des programmes que l'on charge est gérée en pile, dans notre système simple. Il suffit donc de déplacer le pointeur de début de la zone de mémoire libre, à partir de son ancienne position. Cette position est connue et vaut, pendant l'exécution de C, ZoneP = MEM [MEM[FP] + Δ]]. Il suffit donc de calculer ZoneQ ⟵ ZoneP + Taille.

La figure 20.2-(c) montre le résultat de l'allocation : la zone disponible pour Q est comprise entre les adresses ZoneP incluse et ZoneQ exclue.

La procédure C poursuit en recopiant le fichier exécutable en mémoire vive, à partir de l'adresse ZoneP, vers les adresses fortes, c'est-à-dire entre ZoneP incluse et ZoneQ = ZoneP + taille exclue. Elle applique ensuite l'algorithme de translation des adresses. Pour cela il faut lire dans le fichier toujours ouvert la zone des données de translation TEXT et la zone des données de translation DATA. Le fichier ne sert plus à rien ensuite, et peut donc être refermé. Le mot mémoire qui contient l'adresse du nom du fichier à charger (juste sous l'adresse de retour dans l'appelant du chargeur, marqué d'une '*' sur la figure) peut être écrasé, puisque le nom ne sert plus à rien. On y range l'adresse ZoneQ (Cf. Figure 20.2-(c)).

L'un des champs de l'en-tête donne le décalage Décal du point d'entrée du programme Q par rapport au début de sa zone TEXT : c'est l'adresse relative

de sa procédure principale. Ce décalage est stocké dans un registre, puis le pointeur de pile est ramené à la base de l'environnement de la procédure de chargement par SP ⟵ FP. Le pointeur FP est replacé sur la base de l'environnement de l'appelant par FP ⟵ MEM[FP]. On calcule l'adresse absolue du point d'entrée du programme à lancer, d'après l'adresse du début de la zone qui lui a été allouée (ZoneP) et le décalage du point d'entrée. Cette adresse absolue est rangée dans la pile à l'adresse SP : MEM[SP] ⟵ ZoneP + Décal (β).

2.3.4 Lancement effectif du programme chargé

Il ne reste plus qu'une étape pour atteindre l'état décrit par la figure 20.2-(d), dans lequel le programme Q est installé et en cours d'exécution. On s'est ramené à l'invariant décrit par la figure 20.2-(a), Q peut lui-même charger et lancer un autre programme.

Une instruction type rts du langage machine 68000 suffit (γ) : son exécution dépile l'adresse absolue du point d'entrée du programme Q dans le compteur programme. On entre donc dans le code de la procédure principale du programme Q avec SP pointant sur l'adresse de retour dans P, et FP pointant sur la base de l'environnement de P. Le prologue de la procédure principale de Q installe la sauvegarde du pointeur de base, déplace FP, puis déplace SP pour ménager la place des variables locales de la procédure principale de Q.

Les paramètres pour Q, qui lui avaient été transmis par P via C, sont disponibles dans la pile à l'endroit habituel, à partir de FP+3× 4 (les adresses étant stockées dans 4 octets).

2.3.5 Terminaison de la procédure de chargement/lancement

En cas d'erreur lors du chargement, la procédure de chargement/lancement se termine comme une procédure normale, avec un code de retour transmis à son appelant P.

Lorsque le chargement se passe bien, au contraire, la procédure de chargement/lancement ne se termine pas comme une procédure normale : elle se transforme en la procédure qui correspond au programme chargé, qui lui a la structure d'une procédure normale.

Quand la procédure principale du programme chargé se termine, par une séquence de terminaison normale de procédure, elle trouve en sommet de pile l'adresse de retour dans l'appelant du chargeur. On retourne donc dans l'appelant du chargeur directement.

2.4 Allocation dynamique de mémoire par le programme lancé

Nous avons décrit comment est réalisée l'allocation de mémoire pour un programme, lors de son chargement en mémoire. Outre la mémoire allouée une fois pour toutes, qui contient son code et ses données, le programme, lorsqu'il s'exécute, utilise également la pile d'exécution.

Il nous reste à étudier comment et où se font les éventuelles allocations dynamiques de mémoire, demandées par le programme chargé, qui peut faire appel à des procédures Allouer, Libérer telles qu'elles sont décrites au chapitre 4.

Tout d'abord, le code de ces procédures peut faire partie du système de base, et être installé en mémoire en permanence. Il peut aussi être stocké dans une bibliothèque de fonctions, liée au code de notre programme. Dans ce cas les procédures font partie du code du programme chargé.

En quoi consiste l'algorithme des actions Allouer et Libérer ? La première question est : quelle zone de mémoire vive peut-elle être réservée par le programme ? Autrement dit, où reste-t-il de la place, et pour quelle partie de la mémoire le programme lancé a-t-il le moyen de déclarer que la zone est réservée à son usage exclusif ?

Dans le cas de notre système simple, il suffit que le programme chargé augmente artificiellement la zone de mémoire qui lui a été allouée pour ses zones TEXT, DATA et BSS lors de son chargement. Pour cela, la procédure d'allocation d'une zone de taille T appelée par la procédure principale du programme P doit exécuter :

résultat $\longleftarrow$ MEM[FP + Δ]
MEM[FP + Δ] $\longleftarrow$ MEM[FP + Δ] + T

La zone de mémoire comprise entre l'ancienne valeur de MEM[FP + Δ] comprise et la nouvelle valeur exclue est utilisable par le programme P comme il l'entend. Il en connaît l'adresse, ayant reçu le résultat résultat de la procédure d'allocation.

Les procédures d'allocation et de libération peuvent ou non profiter d'une gestion dispersée de la mémoire.

3. Programmation de l'interprète de commandes

L'interprète de commandes est le programme lancé par la procédure globale de démarrage du système. Il est fait pour être actif pendant toute la durée de vie du système ; quand sa procédure principale se termine, le contrôle revient dans la procédure de démarrage (Cf. Paragraphe 1.3).

3.1 Lecture de commandes et utilisation du chargeur/lanceur

L'algorithme le plus simple enchaîne une lecture de ligne de commandes, l'analyse lexicale et syntaxique de ce texte, et l'invocation de la procédure de chargement/lancement pour exécuter le programme requis par l'utilisateur.

```
Lexique
    Fin : un booléen ; L : un tableau de caractères
    NomProg : un tableau de caractères
    Param : un tableau de tableaux de caractères
    NbParam : un entier ≥ 0
Algorithme
    Fin ⟵ faux
    répéter jusqu'à Fin
        { Lecture d'une ligne de commande par appel du pilote de clavier. }
        L ⟵ ...
        { Analyse lexicale et syntaxique de la ligne L : fournit NomProg, Param et
        NbParam }
        si NomProg = "Quitter" alors
            Fin ⟵ vrai
        sinon
            { Passage de paramètres au chargeur/lanceur : les paramètres à desti-
            nation du programme à lancer d'abord, le nom du programme à lancer
            en dernier. }
            i parcourant 1..NbParam : Empiler la chaîne Param[i]
            Empiler la chaîne NomProg
            i parcourant 1..NbParam :
                Empiler les adresses des chaînes précédentes
            empiler NbParam
            empiler adresse de NomProg
            appel de la procédure charger_lancer
            { On revient directement là quand le programme chargé a terminé. }
            dépiler paramètres de charger_lancer
        { Fin de l'interprète. On retourne dans l'appelant du chargeur qui l'avait
        installé, c'est-à-dire le programme de démarrage, qui s'occupe de terminer
        le système proprement. }
```

3.2 Commandes intrinsèques et informations gérées par l'interprète

Nous avons déjà vu le traitement particulier du mot Quitter dans l'algorithme de l'interprète donné ci-dessus. Rien n'empêche d'imaginer de nombreux autres mots ainsi interprétés directement par le programme interprète de commandes, sans que cela se traduise par le chargement en mémoire d'un fichier exécutable disponible dans le SGF.

3.2.1 Commandes intrinsèques de l'interprète de commandes

Nous reprenons Figure 20.3 l'algorithme d'interprétation des commandes, en illustrant comment sont traitées les commandes intrinsèques. Noter la priorité de la commande intrinsèque sur une commande qui aurait le même nom, et qui correspondrait à un fichier exécutable effectivement disponible dans le SGF.

Toutefois la reconnaissance des noms de commandes intrinsèques se fait sur une égalité stricte des chaînes de caractères, et il en géneral possible de contourner cette priorité grâce aux différentes désignations d'un même fichier qu'offre un SGF à structure hiérarchique. Par exemple, dans un interprète de commandes typique d'UNIX comme `csh`, `pushd` est une commande intrinsèque (*built-in command* en anglais ; taper `which pushd` dans un interprète `csh` pour voir apparaître le message `pushd : shell built-in command`). Si l'on écrit un programme dont le fichier exécutable s'appelle `pushd`, dans le répertoire courant, il suffit de taper `./pushd` pour contourner la détection des noms de commandes intrinsèques par l'interprète.

3.2.2 Caractères spéciaux du langage de commandes

On décrit une expérience à faire dans un système permettant de désigner un ensemble de fichiers à l'aide d'un caractère spécial.

Dans les interprètes de commandes du système UNIX, `*.c` représente l'ensemble des fichiers de suffixe `.c`. La commande `ls` affiche à l'écran la liste des fichiers du répertoire courant. La commande `ls *.c` permet alors d'afficher la liste des fichiers de suffixe `.c`. Lorsque l'on donne cette commande à l'interprète du langage de commandes, par qui est traité le caractère spécial `*` ?

Une expérience à réaliser pour guider la réflexion est la suivante : écrire un programme `EcrireParam` qui accepte des paramètres sur la ligne de commandes et les affiche (Figure 20.4). Demander son exécution dans un interprète de commandes, en tapant `EcrireParam *.c`, et observer l'affichage.

Si l'on observe `*.c`, c'est que le caractère `*` n'est pas interprété de manière spécifique par l'interprète de commandes ; si l'on observe effectivement la liste de tous les noms de fichiers terminés en `.c` du répertoire courant, c'est que le caractère `*` est interprété par l'interprète de commandes.

3.2.3 Gestion de répertoires courants

Nous avons déjà vu au chapitre 19 que le système de gestion de fichiers peut proposer une notion de *répertoire courant*, ainsi que les fonctions qui permettent de le fixer et de l'interroger. L'information sur le répertoire courant est une variable globale du système de gestion de fichiers, et les fonctions qui la consultent ou la modifient sont implantées comme des programmes exécutables disponibles dans le SGF. Lorsque l'utilisateur tape la commande `GetRepCour` dans un interprète de commandes, le fichier exécutable correspon-

```
Lexique
    Fin : un booléen
    L : un tableau de caractères
    NomProg : un tableau de caractères
    Param : un tableau de tableaux de caractères
    CommandesIntrinsèques : un ensemble de tableaux de caractères
        { Les noms des commandes directement traitées dans la boucle d'in-
        terprétation, dont "Quitter" }
    NbParam : un entier ≥ 0
Algorithme
    { Diverses initialisations }
    Fin ⟵ faux
    répéter jusqu'à Fin
        { Lecture d'une ligne par appel du pilote de clavier. }
        L ⟵ ...
        { Analyse lexicale et syntaxique de la ligne L : fournit NomProg, Param
        et NbParam }
        si NomProg ∈ CommandesIntrinsèques
            selon NomProg
                NomProg = "comm1" : ...
                NomProg = "comm2" : ...
                NomProg = "Quitter" : Fin ⟵ vrai
                ...
            sinon
                { Séquence d'appel du chargeur/lanceur, voir paragraphe 3.1 ci-
                dessus. }
    { Fin de l'interprète. }
```

FIG. 20.3 – Commandes intrinsèques de l'interprète de commandes

```
void main(int argc, char *argv[])
{
     int i ;
     for (i=0 ;i<argc ;i++)
         printf("Argument no %d : %s ", i, argv[i]) ;
}
```

FIG. 20.4 – Un programme C qui imprime ses paramètres

dant est chargé en mémoire, exécuté, rend un résultat, et ce résultat est affiché par l'interprète de commandes.

Il est important de noter que, si le SGF n'offre pas la notion de répertoire courant, il est impossible de l'introduire au niveau de l'interprète de commandes. L'information sur le répertoire courant devrait être une variable du programme interprète, et les fonctions de consultation et modification de cette variable des commandes intrinsèques. La prise en compte d'une notion de répertoire courant devrait être assurée par l'algorithme de l'interprète, qui devrait préfixer systématiquement les noms de fichiers non absolus (ceux qui ne commencent pas par un caractère slash, dans la syntaxe UNIX) par la chaîne de caractères répertoire courant. Mais l'interprète de commandes *n'a aucun moyen de savoir* lesquels, parmi les arguments d'une commande, sont des noms de fichiers. Seul le programme lancé sait lesquels de ses paramètres sont des noms de fichiers, et pour ceux-là il appelle les fonctions d'accès du SGF, lesquelles peuvent préfixer le nom par le répertoire courant. La commande `cd` (*change directory*) des interprètes de UNIX usuels ne peut donc pas être une commande intrinsèque : elle fait appel à la fonction correspondante du SGF.

En revanche, il est possible de gérer, dans l'interprète de commandes, une *pile* de répertoires. On introduit alors `empiler` et `dépiler`, comme spécifiés au chapitre 4, paragraphe 5.1, sous forme de commandes intrinsèques. Par exemple, dans l'interprète `csh` de systèmes UNIX, ces commandes s'appellent `pushd` et `popd`. Une telle commande met à jour la variable pile de l'interprète de commandes, et appelle la fonction SetRepCour du SGF.

3.2.4 Gestion d'une liste de chemins d'accès privilégiés aux commandes

Une autre fonction couramment offerte par les interprètes de commandes est la gestion d'une liste de chemins d'accès privilégiés aux commandes que l'utilisateur est susceptible de taper.

L'interprète de commandes gère alors un tableau des préfixes possibles pour la désignation d'un fichier et utilise celui-ci dans l'ordre lors de l'accès à un fichier.

L'utilisateur dispose de commandes intrinsèques permettant d'afficher la liste des préfixes dans l'ordre utilisé par l'interprète et de modifier cette liste. En général, l'interprète de commande est initialisé avec une liste de préfixes prédéfinie. Par exemple dans le système UNIX, on trouve `/usr/bin` et `/usr/local/bin` dans la valeur initiale de la variable `path`, spécifiant les préfixes possibles pour les fichiers.

Nous donnons Figure 20.5 ci-dessous un complément à l'algorithme de l'interprète de commandes du paragraphe 3.1. Nous supposons que le préfixe est séparé du nom du fichier par le caractère slash comme en UNIX.

```
Lexique
   ...
   Préfixes : un tableau de tableaux de caractères
      { Les préfixes possibles pour un nom de fichier correspondant à une
      commande }
   NbPréfixes : un entier ≥ 0
   ok : un booléen ; j : un entier
Algorithme
   { Diverses initialisations dont celle du tableau Préfixes et de la variable
   NbPréfixes }
   ...
   répéter jusqu'à Fin
      ok ⟵ faux ; j ⟵ 1
      tant que (non ok) et (j ≤ NbPréfixes)
         Empiler les paramètres à destination du programme à lancer
         Empiler la chaîne Préfixes[j] & "/" & NomProg
            { On a empilé le nom du programme à lancer. & est l'opérateur
            de concaténation de chaînes de caractères }
         { Empiler les adresses des chaînes précédentes, le nombre de pa-
         ramètres et l'adresse du nom du programme à lancer }

         Appel du chargeur/lanceur
         { Si le chargeur ne trouve pas le fichier demandé, il rend un code
         d'erreur FICHIER_NON_TROUVE (Cf. Paragraphe 2.1.2). }

         si résultat = FICHIER_NON_TROUVE
         alors j ⟵ j+1 { on va essayer avec un autre préfixe }
         sinon ok ⟵ vrai { on sort de la boucle }

      si (non ok)
      alors Ecrire ("Commande non trouvée")
      sinon
         { le programme a été exécuté, dépiler les paramètres du char-
         geur/lanceur }
   { Fin de l'interprète. }
```

FIG. 20.5 – Gestion d'une liste de chemins d'accès

4. Fonctions évoluées des interprètes de commandes

4.1 Langages de commandes graphiques

Les interprètes de commandes ne sont pas nécessairement textuels. Ainsi les systèmes d'exploitation qui offrent des interfaces graphiques évoluées proposent un langage de commandes dit *icônique* : les principales commandes sont représentées par des icônes, ou des boutons, sur lesquels il suffit de cliquer pour activer la commande. Dans ce cas il n'y a pas de phase d'analyse lexicale et syntaxique des textes tapés par l'utilisateur.

Un exemple typique concerne la manipulation qui permet de déplacer ou copier des fichiers dans une interface type WINDOWS ou MACINTOSH : en cliquant sur une icône qui représente un fichier, et en déplaçant la souris, on déplace ou copie le fichier. Dans un langage de commandes textuel, cela correspond à une commande du genre : `mv nom1 nom2`, où `nom1` et `nom2` sont des noms absolus ou relatifs du système de gestion de fichiers.

4.2 Structures de contrôle

Les langages de commandes icôniques sont souvent plus faciles à utiliser que les langages textuels, mais moins puissants. En effet, les langages de commandes textuels peuvent être étendus facilement jusqu'à devenir de véritables langages de programmation, par l'ajout de structures de contrôle. Pour réaliser le même traitement sur tous les fichiers d'un répertoire qui se terminent en `.fig`, on écrit une boucle. Dans le langage `csh` des systèmes UNIX, on écrit par exemple :

```
foreach i (‘ls‘)
# pendant l’exec. de la boucle, la variable $i parcourt
# tous les fichiers obtenus par la commande ls.
set j=‘basename $i .fig‘    # recuperer nom de base, sans extension
if ($i  == $j) then         # le fichier ne se termine pas par .fig
    echo $i " : fichier ignore"
else
    echo -n "[fichier " $i "traite ..."
    ... commandes portant sur le fichier de nom $i
    echo "]"
endif
end
```

On reconnaîtra aisément dans le texte de programme ci-dessus des structures de contrôle similaires à celles présentées au chapitre 4. Les actions de base, en revanche, sont très différentes. Dans un langage de programmation ordinaire, on y trouve principalement l'affectation et les entrées/sorties. Ici on

a affaire à un langage dédié à la description de commandes système. Parmi les actions de base du langage de commandes, on trouve donc en particulier l'invocation de tous les programmes système de base (appel du programme `ls` du SGF dans l'exemple ci-dessus).

Sixième partie

Architecture des systèmes matériels et logiciels complexes

Chapitre 21

Motivations pour une plus grande complexité

Les parties I à V de cet ouvrage présentent tout ce qui est nécessaire à la construction d'un *ordinateur*, au sens défini dans l'introduction. La machine simple étudiée comprend un organe de calcul (le processeur), de la mémoire et des périphériques d'entrée/sortie. Nous l'avons d'autre part équipé du *logiciel de base* minimal nécessaire à son utilisation : un système de gestion de fichiers, des outils de création de programmes, un interprète de commandes.

Toutefois les ordinateurs réels sont des systèmes matériels et logiciels plus complexes que cela. L'objet de ce chapitre est de comprendre les motivations qui président à la définition de systèmes matériels et logiciels *complexes*, représentatifs de la plupart des ordinateurs actuels.

> *Nous commençons par définir précisément, paragraphe 1., ce que nous appelons par la suite un système complexe. Pour répondre aux besoins nés des utilisations complexes envisagées, un nouveau mécanisme est détaillé au paragraphe 2. et son application aux besoins des systèmes complexes est évoquée paragraphe 3. Le paragraphe 4. donne un plan détaillé de la partie VI de cet ouvrage.*

1. Qu'appelle-t-on système complexe ?

Les systèmes avec lesquels nous avons l'habitude de travailler sont *multitâches* et *multi-utilisateurs*. Ce sont deux notions très différentes, et quasiment indépendantes l'une de l'autre, que nous détaillons ci-dessous. Par ailleurs ces systèmes donnent des droits différents à différents programmes.

1.1 Systèmes multitâches

Considérons une situation courante d'utilisation d'un ordinateur personnel : l'utilisateur humain est occupé à taper un courrier, dans une fenêtre d'un

éditeur de textes quelconque ; l'horloge affichée en permanence sur l'écran progresse régulièrement ; un transfert de fichiers demandé par l'intermédiaire d'un butineur Internet est en cours, etc. L'ordinateur considéré possède un processeur, voire 2, mais de toutes façons en nombre largement inférieur au nombre d'activités différentes qui semblent se dérouler simultanément. Il faut donc bien imaginer que la ressource *processeur* est partagée entre les différents programmes qui supportent les activités de l'utilisateur. Pourtant ce partage est quasiment non perceptible par l'être humain : l'éditeur de textes se comporte de manière uniforme dans le temps (l'écho à l'écran des caractères tapés au clavier ne paraît pas suspendu) ; l'horloge avance régulièrement et marque l'heure juste, etc. D'autre part l'utilisateur éclairé qui crée de nouveaux programmes destinés à cet ordinateur n'a pas à se préoccuper, en général, du partage du processeur. Il écrit des programmes sans savoir s'ils seront exécutés par un système monotâche ou multitâches. Il nous faut donc maintenant supposer l'existence d'un mécanisme de gestion de la ressource processeur, capable d'intervenir sur le déroulement normal d'un programme.

1.2 Systèmes multi-utilisateurs

Les systèmes que nous utilisons couramment sont également multi-utilisateurs, c'est-à-dire que des personnes différentes peuvent utiliser les ressources de la machine, simultanément ou séquentiellement. S'ils l'utilisent simultanément, ce n'est qu'un cas particulier de fonctionnement multitâches ; il faut toutefois prévoir plusieurs organes périphériques, typiquement plusieurs couples écran/clavier connectés au même processeur. C'est le cas des ordinateurs *serveurs*, sur lesquels sont branchés de nombreux *terminaux*.

La spécificité d'un fonctionnement multi-utilisateurs tient à la notion de *propriété* qui peut être attachée aux informations persistantes comme les fichiers.

La notion de propriété des données conduit à proposer des mécanismes de *protection*, à différents niveaux : l'accès par mot de passe qui permet de protéger les espaces disques, c'est-à-dire les données persistantes des différents utilisateurs ; un mécanisme actif pendant les exécutions de programmes, de manière à éviter (ou du moins à contrôler sérieusement), les interactions entre programmes de différents utilisateurs.

1.3 Hiérarchie de droits

Un *système d'exploitation* assure un partage de compétences entre ce qu'on appelle les *utilisateurs*, et les *concepteurs* de systèmes d'exploitation. On peut établir une hiérarchie, depuis l'utilisateur du traitement de texte, qui met l'ordinateur sous tension et lance un logiciel, jusqu'au développeur qui a programmé les pilotes de disques et d'imprimante, en passant par celui qui a programmé le traitement de textes. La protection, dans ce contexte, consiste

à introduire des mécanismes qui permettent, entre autres, d'interdire à l'utilisateur final de réaliser des opérations de très bas niveau sur le disque. Les accès aux entrées/sorties sont réservés aux programmes qui ont été conçus par les développeurs du système d'exploitation. Il faut donc un mécanisme de vérification que tous les accès aux entrées/sorties, faits apparamment par l'utilisateur final, se font bien, en réalité, à travers des programmes dûment certifiés. Cette vérification se fait pendant l'exécution, sans que l'utilisateur ne s'en aperçoive.

2. Scrutation

En fait tout se ramène à un problème de *scrutation*. De façon intuitive, dans un système complexe il va falloir scruter quelques événements intérieurs ou extérieurs, c'est-à-dire consulter une information de temps en temps, et prévoir un branchement à un traitement adéquat si elle a une valeur particulière. Cette information peut être liée au temps qui passe ou à une infraction aux droits d'accès par exemple. Cela peut permettre de gérer les aspects de partage de temps, les aspects de protection et de réagir à des erreurs.

Observons tous les niveaux où il est possible d'insérer une action entre des instructions pour assurer une régularité de la scrutation. On trouve grossièrement trois niveaux : les programmes en langage de haut niveau des utilisateurs ; les programmes en assembleur produits par compilation ; l'interprète du langage machine.

On n'a pas vraiment le choix : la scrutation doit être effectuée le *plus bas possible*.

Il est intéressant de réaliser la scrutation *une seule fois*, de ne pas faire confiance aux divers programmes utilisateurs pour ça, et de faire en sorte que les couches de niveau le plus haut soient le plus possible indépendantes de ce mécanisme de scrutation ; en effet le programmeur de haut niveau n'a parfois même pas connaissance des événements à observer. Il faut donc que la scrutation soit prévue dans le processeur, et pour cela il faut intervenir dans l'algorithme d'interprétation des instructions.

On sait intégrer la scrutation dans l'interprète de langage machine *sans trop le ralentir* ce qui n'est pas le cas si on l'intègre dans les couches supérieures. Le coût en temps de la scrutation est nul ou dérisoire en la réalisant au niveau de l'interprète de langage machine.

Notons que la scrutation matérielle n'ajoute pas d'états au graphe de contrôle du processeur, seulement des transitions. Par du matériel, un éclatement n_aire après un état est réalisé de la même façon qu'un éclatement binaire. Ce n'est pas le cas pour les systèmes programmés ou microprogrammés.

Le fait de placer la scrutation dans l'interprète de langage machine permet d'avoir un certain contrôle sur la fréquence à laquelle se fait la scrutation. Au contraire si on le fait dans les programmes utilisateurs, on ne maîtrise pas bien

le lien avec le temps physique ; on pourrait placer deux scrutations de part et d'autre d'une boucle qui se révèlerait très longue pour certaines données.

Le mécanisme de scrutation est aussi utilisé pour la détection de certaines erreurs. Supposons par exemple que l'on exécute une division avec un diviseur nul. Qui se rend compte du problème ? Le programmeur peut encadrer toute opération de division de son programme de haut niveau par un test ; il traite les cas d'erreur comme bon lui semble. Ou bien le compilateur peut produire un code assembleur qui contient exactement ces tests, et un saut à une procédure de traitement d'erreur définie dans le compilateur. Comment l'information sur cette erreur remonte-t-elle au programme, ou au moins à la personne qui fait exécuter le programme ?

Lors de l'exécution d'une division, le processeur vérifie que le diviseur n'est pas nul. Le cas échéant, une action particulière est déclenchée. Les différents programmeurs peuvent avoir connaissance des résultats de cette action et programmer un message d'erreur pour l'utilisateur final.

3. Mécanisme d'interruption : définition et types d'utilisations

3.1 Définition et utilisations typiques

Le mécanisme d'*interruption* est une manière astucieuse de faire coopérer le matériel et le logiciel en utilisant le mécanisme de scrutation. On ne sait pas la mettre en oeuvre avec les éléments présentés dans les chapitres précédents.

Une interrution est la prise en compte d'un événement qui s'est produit dans le processeur lui-même ou dans une autre unité matérielle qui lui est associée.

Une interruption peut être liée à l'instruction en cours d'exécution. On parle alors d'*interruption interne*.

Considérons par exemple le mécanisme de changement de fenêtre de registres du SPARC (Cf. Chapitre 13). Lors de l'exécution d'une instruction `save` pour décaler la fenêtre, il faut s'assurer que c'est encore possible. Dans le cas où il ne reste pas de fenêtre libre il est nécessaire de réaliser la sauvegarde d'une fenêtre dans la pile. Lors de l'exécution d'un `restore` il faut éventuellement réaliser la restauration d'une fenêtre depuis la pile. Le schéma de codage que nous avons proposé impose de faire, en début de procédure un décalage dans un sens avec test de débordement et sauvegarde, à la fin un décalage dans l'autre sens avec test de vide et restauration. En réalité, sur le SPARC, on écrit seulement `save`, `restore`, et le processeur assure les sauvegardes et restaurations de façon transparente pour le programmeur. C'est le mécanisme des interruptions qui est mis à contribution.

Il existe des interruptions qui ne sont pas liées à l'instruction en train de s'exécuter sur le processeur. On les appelle *interruptions externes*.

Par exemple, la prise en compte d'une baisse de tension peut permettre d'interrompre un traitement en cours d'exécution pour effectuer une sauvegarde sur disque d'informations importantes. Cela peut être réalisé de manière complètement transparente pour notre programme, grâce au mécanisme des interruptions. Le programmeur n'a pas besoin d'ajouter dans ses programmes une action de vérification que l'alimentation de l'ordinateur fonctionne correctement !

De même la détection des début et fin de traitement d'une unité d'entrées/sorties peut permettre d'optimiser l'utilisation du périphérique.

On appelle *mécanisme d'interruption* une scrutation incluse dans le processeur et permettant de lancer des programmes associés. Ce mécanisme est plus ou moins configurable. Par exemple lors de l'assemblage d'un ensemble d'unités périphériques à un processeur, on pourra fixer les adresses des programmes traitant les événements liés à ces périphériques. En général, il est prévu un système de branchement avec indirection, les adresses de branchement étant stockées dans une table en mémoire.

3.2 Aspects temporels de l'utilisation

On peut distinguer deux types d'utilisations du mécanisme d'interruption selon l'évolution temporelle des événements à gérer (Cf. Chapitre 6).

Dans le cas où le rythme des événements à prendre en compte est complètement imposé de l'extérieur, on peut supposer, par exemple pour gérer une montre, que l'on reçoit un signal toutes les secondes. Le programme bien fait ne doit pas *perdre* de secondes : il est donc capable de faire la mise à jour de sa variable interne en moins de temps qu'une seconde. Si le programme est mal fait ou la machine pas assez rapide, le fonctionnement sera incorrect.

Dans le cas de gestion d'entrées/sorties, il s'agit d'un *dialogue* entre le processeur et un dispositif situé à l'extérieur. Autrement dit, les signaux à scruter sont tels qu'il ne se passe rien d'intéressant en dehors des périodes où le processeur a *demandé* une action ; d'autre part quand il reçoit ce qu'il a demandé, il répond en disant qu'il a bien consommé, et le dispositif extérieur peut se permettre de lui envoyer une nouvelle requête.

Signalons une vision particulière du système d'interruptions. Lorsqu'on programme sur une machine déjà dotée d'un système d'exploitation, on voudrait pouvoir utiliser les bibliothèques de programmes de ce système. Pour cela il faudrait connaître leurs adresses en mémoire. C'est évidemment impossible. Une liaison *statique* est donc exclue. Une utilisation du mécanisme d'interruption, à travers les appels superviseurs que nous détaillerons dans le chapitre 24, permet quand même d'utiliser ces programmes.

4. Plan de la suite

Pour inclure la gestion de mécanisme d'interruptions dans le processeur il faut remettre en cause l'interprète du langage machine.

Dans la suite, nous étudions :
- le mécanisme d'interruption lui-même : principe et résolution des problèmes que son introduction pose, en particulier les problèmes de relation de temps entre le processeur et le monde extérieur.
- la définition des fonctions et du fonctionnement d'un système complexe comme défini plus haut, en utilisant ce nouveau mécanisme.

Dans le chapitre 22 nous présentons un mécanisme d'interruption simple. On étudie la modification de l'interprète de langage machine, et on présente les problèmes liés. On introduit pour cela une machine simple comportant un seul fil d'interruption. Cela nous permet de montrer que ce mécanisme d'interruption permet déjà un partage de temps élémentaire entre un programme principal et un traitant d'interruption. Nous étudions le cas typique de la pendule : le processeur est partagé entre deux activités, et les instructions de l'un et l'autre programmes sont entrelacées de manière non maîtrisable par le programmeur.

La lecture de ce chapitre est suffisante pour comprendre les aspects d'architecture matérielle liés à l'introduction du mécanisme d'interruption. La suite constitue une introduction aux couches basses des systèmes d'exploitation multitâches et multi-utilisateurs.

Le partage de temps rudimentaire vu au chapitre 22 est limité au cas de deux programmes, dont l'un est suffisamment court. Dans le cas général (par exemple lorsqu'aucun des programmes ne s'arrête !), il faut inventer un mécanisme un peu plus sophistiqué : la notion de *processus* et le *partage de temps*, géré par le traitant d'interruption. C'est une brique de base importante pour la construction de systèmes complexes. C'est l'objet du chapitre 23.

Dans le dernier chapitre (24) nous étudions le fonctionnement détaillé des interruptions et leurs utilisations. Nous complétons le processeur du chapitre 22 pour traiter les aspects liés à la gestion d'interruptions multiples. Nous évoquons les problèmes de protection entre les diverses activités se déroulant dans une machine. Nous montrons enfin que le mécanisme des interruptions permet d'optimiser la gestion des entrées/sorties présentée au chapitre 16.

Chapitre 22

Le mécanisme d'interruption

Nous allons doter un processeur d'un mécanisme capable de scruter le monde extérieur fréquemment, régulièrement et à des instants bien identifiables. Le choix qui est fait est de scruter ENTRE les instructions. On pourrait tout aussi bien construire un processeur où la scrutation aurait lieu toutes les 4 instructions. Ce mécanisme permet de mettre en place un partage du temps entre le programme de l'utilisateur et une tâche activée en fonction des résultats de la scrutation. Ce mécanisme sera enrichi dans les chapitres ultérieurs d'une scrutation à des instants particuliers pendant l'exécution d'une instruction.

Le processeur décrit au chapitre 14 ne convient pas. Il était fait pour montrer les principes de base de ce qu'*est* un processeur. Il était trop simple pour supporter les extensions dont la scrutation a besoin.

> *La première étape du travail consiste (paragraphe 1.) à se doter d'un processeur pouvant être étendu aisément par un mécanisme de scrutation élémentaire (paragraphe 2.). Nous donnons au paragraphe 3. une utilisation* typique *de ce mécanisme : la mise à jour de l'heure. De nombreuses autres utilisations existent. Au paragraphe 4., nous évoquons les problèmes liés aux accès concurrents à des informations partagées.*

1. Architecture d'un processeur pour la multiprogrammation

1.1 Partie opérative du processeur

La figure 22.1 décrit l'organisation de la partie opérative du processeur[1]. Elle est inspirée des P.O. types déjà connues. Le processeur dispose de plusieurs registres de N bits (32, typiquement, mais peu importe). Bus adresses et bus

[1] S'assurer avant de continuer la lecture de ce chapitre qu'on ne confond pas la Partie Contrôle (PC) d'un processeur et son PC (Program Counter) ou son registre d'état et le registre contenant l'état de sa Partie Contrôle.

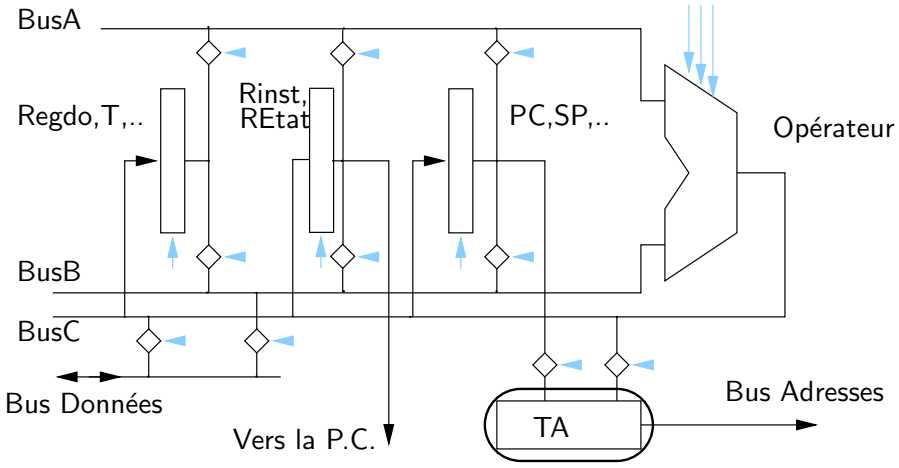


FIG. 22.1 – Partie opérative schématique du processeur. Les commandes apparaissent en grisé, mais leurs noms ne sont pas donnés.

données vers la mémoire font aussi N bits. Les adresses de mots sont des multiples de 4. Les registres sont identifiés par un numéro ou un nom. Les noms sont principalement une facilité mnémotechnique. Les instructions ont 3 opérandes au plus. Comme sur le SPARC le registre de numéro 0 contient la constante 0. Les registres apparaissent sur la figure comme Regdo. SP vaut pour *Stack Pointer*; il est utilisé comme pointeur de pile. PC vaut pour *Program Counter*; c'est le compteur programme. Le programmeur peut les manipuler par des instructions spéciales.

Le processeur comporte un registre d'état REtat dont le contenu est accessible en permanence par la partie contrôle.

Tous les registres, y compris le registre d'état, peuvent être transmis via les bus A et B vers l'opérateur. La sortie de l'opérateur peut être transmise dans n'importe quel registre via un bus C. L'opérateur est doté de toutes les opérations nécessaires : extraction des p bits de poids faible, génération de constantes, fabrication des bits d'état Z, N, C, V, etc.

Des commutateurs entre le bus données bidirectionnel vers la mémoire et les bus B et C permettent les communications dans les deux sens : mémoire vers registres et registres vers mémoire.

Certains registres ne sont pas directement accessibles par le programmeur dans les instructions : un registre particulier, connu sous le nom de Registre Instruction Rinst, dont le contenu est accessible en permanence dans la partie contrôle du processeur; un registre tampon général T sert d'intermédiaire de stockage de valeurs temporaires; un registre TA sert de Tampon d'Adresse, son contenu est transmis sur le bus adresses.

1.2 Répertoire d'instructions

Les instructions du langage machine sont codées sur un ou deux mots mémoire. Nous les décrivons et montrons quel langage d'assemblage pourrait être le correspondant de ce langage machine. Le processeur dispose des instructions :

- Chargement du contenu d'un mot mémoire dans un registre et rangement du contenu d'un registre dans un mot mémoire :

  Pour ces deux instructions l'adresse mémoire est obtenue en ajoutant le contenu d'un registre et une constante contenue dans le deuxième mot de l'instruction. On peut imaginer si besoin des variantes précisant la taille de l'opérande et la façon de compléter les opérandes courts. Syntaxiquement, en langage d'assemblage, cela donne :
  `ld [no_regA, constante], no_regB`,
  `st no_regB, [no_regA, constante]`
  où `no_reg` est un numéro de registre.

  Quand le registre à ajouter à la constante est le registre 0, il serait naturel d'admettre aussi la syntaxe :
  `ld [constante], no_regB`,
  `st no_regB, [Constante]`.

  Il est, par ailleurs possible de doter le langage d'assemblage d'une pseudo-instruction `set` pour installer une constante dans un registre ; c'est particulièrement utile quand cette valeur est donnée par l'intermédiaire d'une étiquette qui représente une adresse.

- Opérations en tout genre entre deux registres, ou entre un registre et une constante et rangement du résultat dans un troisième registre :

  La constante est dans le deuxième mot de l'instruction. Ces instructions ne positionnent pas les bits Z, N, C, V du registre mot d'état REtat. Comme dans le SPARC, il existe des variantes des instructions qui positionnent les indicateurs arithmétiques. Ces opérations avec le registre 0 comme destinataire positionnent seulement le mot d'état. Syntaxiquement, en langage d'assemblage, cela donne par exemple :
  `add no_reg_source1, no_reg_source2, no_reg_dest`,
  `sub no_reg_source1, constante , no_reg_dest`,
  `addcc no_reg_source1, no_reg_source2, no_reg_dest`.

- Instructions de saut : `jmp` ou `jsr destinataire`

  Le deuxième mot des instructions de saut est ajouté au contenu d'un registre pour obtenir l'adresse de saut. Pour `jsr`, il y a empilement de l'adresse de retour. Syntaxiquement, en langage d'assemblage, cela donne : `jmp no_reg, constante`. On pourrait écrire : `jmp constante` au lieu de `jmp r0, constante`.

- Retour de sous-programme : `rts`, correspondant au `jsr`.

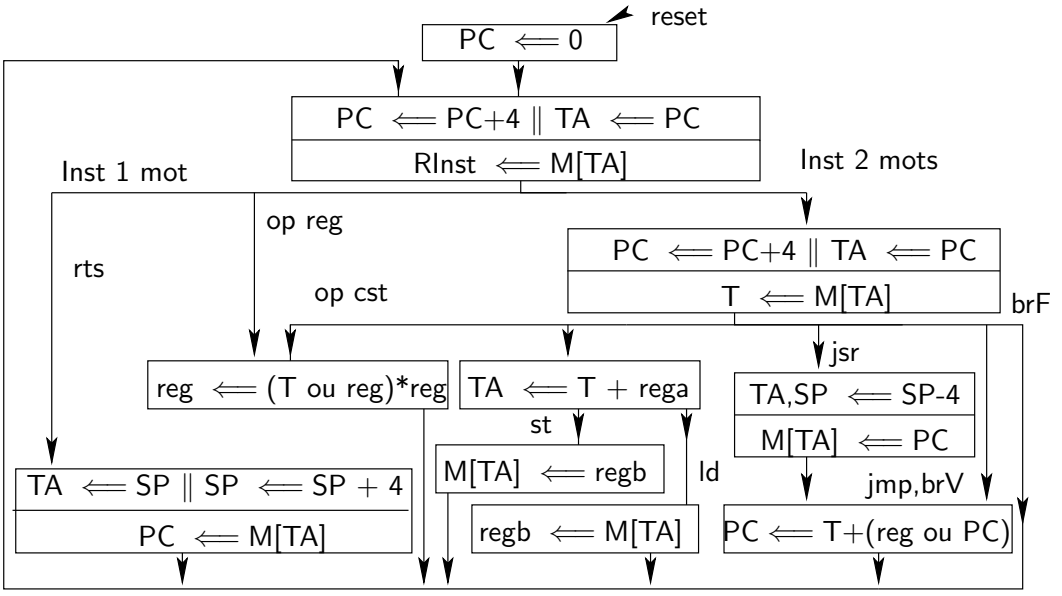


FIG. 22.2 – Graphe de contrôle du processeur, première version. Certains rectangles comportent 2 lignes distinctes, correspondant à deux états successifs, donc à deux microactions consécutives. Les deux états seraient séparés par une transition portant sur le prédicat toujours vrai.

– Branchement conditionnel : `br destinataire`
Les conditions portent sur des combinaisons de bits du mot d'état REtat. L'adresse effective de destination est obtenue en ajoutant le deuxième mot de l'instruction et le compteur programme. La condition est donnée dans un champ de l'instruction. Syntaxiquement, en langage d'assemblage, cela donne : `br_cond constante`

– D'autres instructions seront ajoutées dans les versions ultérieures de cette machine.

On peut récapituler ces instructions dans un tableau :

taille	instruction	options	champ1	champ2	champ3
1 mot	ret. de ss-prog.				
1 mot	arithmétique	+ cc	reg_{s1}	reg_{s2}	reg_d
2 mots	arithmétique	+ cc	reg_{s1}	Const.	reg_d
2 mots	accès mém. LD		reg_a	Const.	reg_b
2 mots	accès mém. ST		reg_a	Const.	reg_b
2 mots	saut	sauv ad. retour		Const.	reg_a
2 mots	branchement		cond.	Const.	

1.3 Interprétation des instructions

Le graphe de contrôle spécifiant le comportement de la partie contrôle de ce processeur est donné figure 22.2 ; les instructions codées sur 1 mot nécessitent un accès mémoire et les instructions codées sur 2 mots en demandent deux. Un certain nombre de détails n'apparaissent pas comme les opérations effectivement réalisées par l'opérateur (figurées par une * dans le graphe de contrôle). De même nous ne précisons pas la façon dont est effectivement faite la mise à jour du compteur programme PC par `T+reg` ou `T+PC`, que nous notons : `PC` $\Longleftarrow$ `T+(reg ou PC)`, le mode d'adressage et la condition de branchement (brV représente tous les branchements où la condition est évaluée à vrai, et brF les branchements où la condition est fausse). Ce sont là deux exemples de paramétrisation.

2. Introduction d'un mécanisme de scrutation élémentaire

Comprendre le mécanisme de scrutation nécessite un double point de vue : une vision interne, ce qui se passe à l'intérieur du processeur, et une vision externe, ce que le programmeur peut programmer et observer.

Du point de vue externe, le mécanisme de scrutation se comporte comme un appel de sous-programme mais déclenché par un changement d'état sur un fil plutôt qu'en raison de l'exécution d'une instruction.

On introduit dans le processeur ce mécanisme de scrutation du monde extérieur. Un fil d'entrée du processeur IRQ (pour Interrupt ReQuest) est en entrée de la partie contrôle. Ce fil provient directement du monde extérieur. Cela apparaît sur la figure 22.4. Un sous-programme, nommé conventionnellement TRAITANT, est rangé en mémoire. Il se termine nécessairement par une instruction nouvelle : `rti`. Comme on le verra dans le graphe de contrôle de la figure 22.3 le déroulement de l'instruction `rti` ressemble à celui de l'instruction `rts`. Nous les différencierons plus tard.

L'ensemble ainsi constitué, un processeur doté de nouvelles fonctionnalités et un nouveau type de programmes, constitue ce que l'on appelle un *système d'interruptions*.

2.1 Le mécanisme de base, vision interne

Comme précédemment, la partie contrôle du processeur peut calculer un nouvel état en fonction du code opération contenu dans Rinst, de l'état du processeur contenu dans REtat et de l'état courant de la partie contrôle. De plus la partie contrôle a accès au bit IRQ et peut calculer son nouvel état en fonction de la valeur de ce bit.

Le fil IRQ est pris en compte entre deux instructions, c'est-à-dire juste après

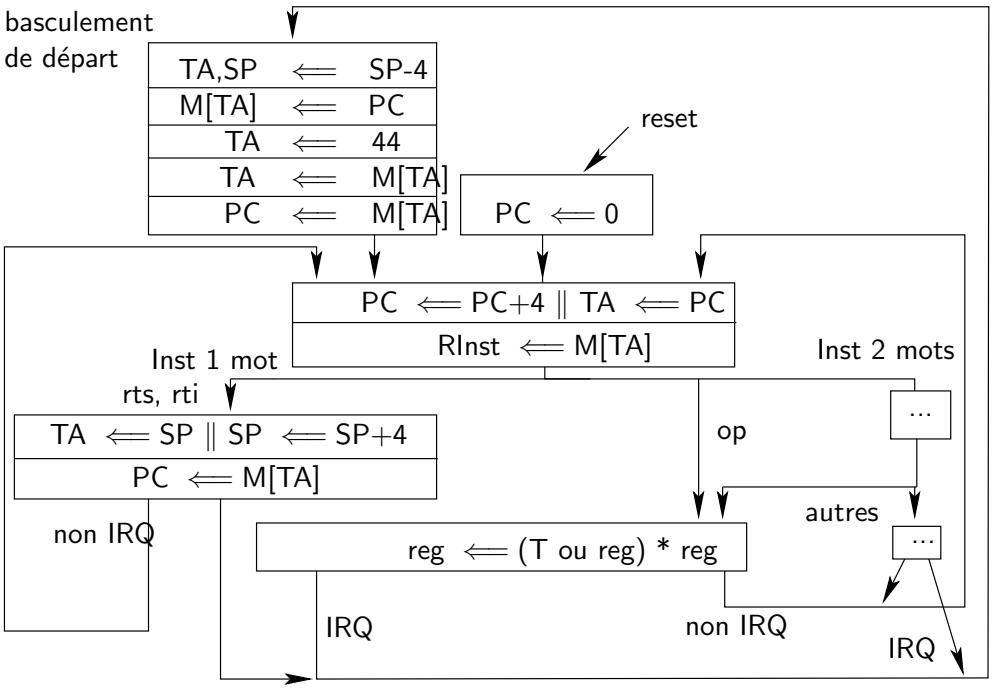


FIG. 22.3 – Implantation du mécanisme de base dans le graphe de contrôle du processeur.

la dernière microaction d'une instruction. Sur le graphe de contrôle (Cf. Figure 22.3), tout dernier état d'une instruction a donc deux successeurs. Pour un des successeurs cela revient à changer d'instruction, comme précédemment, pour l'autre cela revient à mettre en route une séquence d'états particulière : *le basculement de départ*. On trouve parfois le terme de changement de contexte.

• Si IRQ vaut 0, le processeur passe normalement à l'instruction suivante de l'instruction qui vient de finir. La suivante peut être celle écrite à la suite ou, si la dernière instruction est un branchement, c'est l'instruction qui se trouve à l'adresse de saut. Dans les deux cas, au moment de la prise en compte de IRQ, l'adresse de cette instruction suivante est dans le compteur programme PC.

• Si IRQ vaut 1, la séquence de basculement de départ a lieu : l'adresse de l'instruction suivante présente dans PC est sauvegardée, dans la pile, à la façon d'un `jsr`. Puis la nouvelle adresse, celle de début du sous-programme TRAITANT, est forcée dans le compteur programme. Il y a ainsi *basculement* du programme normal au programme TRAITANT. Cette adresse n'est pas fournie par une instruction comme dans le cas du JSR. Elle est forcée par le processeur lui-même en interne. Cela peut être une constante. Ou, comme nous le choisissons ici, une valeur rangée en mémoire à une adresse pré-établie constante (Ici, 44 ; dans les vraies machines une valeur choisie telle qu'elle tombe au début ou à la fin de la mémoire). Notons que l'accès au sous-programme TRAITANT est réalisé via une double indirection. Cela permet d'envisager une extension

à plusieurs traitants dont les adresses de début seraient stockées à partir de l'adresse 6000.

> *On discutera, sous le nom de vectorisation du choix de cette technique au paragraphe 1.5 du chapitre 24. L'exercice E22.1 permet de discuter aussi ce point.*

L'exécution du `rti` consiste ici à dépiler l'adresse de retour comme le `rts`. Il y a alors basculement du programme TRAITANT au programme normal.

2.2 Le mécanisme de base, vision externe

Suivons sur un exemple ce que voit le programmeur lors de l'occurrence d'une interruption. La situation est représentée par la figure 22.4. Supposons que les différentes parties de programme représentées ne comportent pas de saut hors de ces parties (vers les zones représentées en grisé).

La mémoire comporte un programme aux adresses 1000 à 2000. La pile est placée aux adresses de 4000 à 5000, où il doit, naturellement, y avoir de la mémoire vive. Quand la pile est vide, SP contient 5000. Le sous-programme TRAITANT commence à l'adresse 3000. Il se termine par une instruction `rti`, à l'adresse 3500 sur le schéma. La valeur 3000 est rangée à l'adresse 6000. La valeur 6000 est rangée à l'adresse 44.

On peut observer un historique des valeurs successives du compteur programme et du pointeur de pile sur la figure 22.5.

2.3 Commentaires

Le déroulement envisagé précédemment soulève plusieurs questions, notamment de vocabulaire. Nous allons les envisager.

1. Quel est le nom de ce mécanisme ? Mécanisme d'*interruption*. On dit que le programme en cours d'exécution a été interrompu. Le sous-programme TRAITANT s'appelle un *traitant d'interruption*.
2. Le départ vers le sous-programme TRAITANT est provoqué par un événement matériel (le niveau 1 sur le fil IRQ), le retour de sous-programme est provoqué par un événement logiciel (l'exécution de l'instruction `rti`). Cette dissymétrie est-elle normale ? Oui. C'est là toute l'astuce de ce mécanisme. On parle de *basculement* de départ et de *basculement* de retour. On dit souvent *départ en interruption, retour d'interruption*.
3. Le basculement de retour fait temporellement partie du TRAITANT, mais pas le basculement de départ. C'est une convention plutôt qu'autre chose. Le basculement de départ n'est ni dans le programme interrompu ni dans le TRAITANT. Il est temporellement entre les deux.
4. Que se passe-t-il si il n'y a pas d'instruction `rti` dans TRAITANT ? Rien de spécial. Le processeur continue d'exécuter des instructions. Il n'y aura jamais de retour au programme interrompu. Il s'agit évidemment d'une erreur analogue à l'absence d'un `rts` en fin d'un sous-programme.

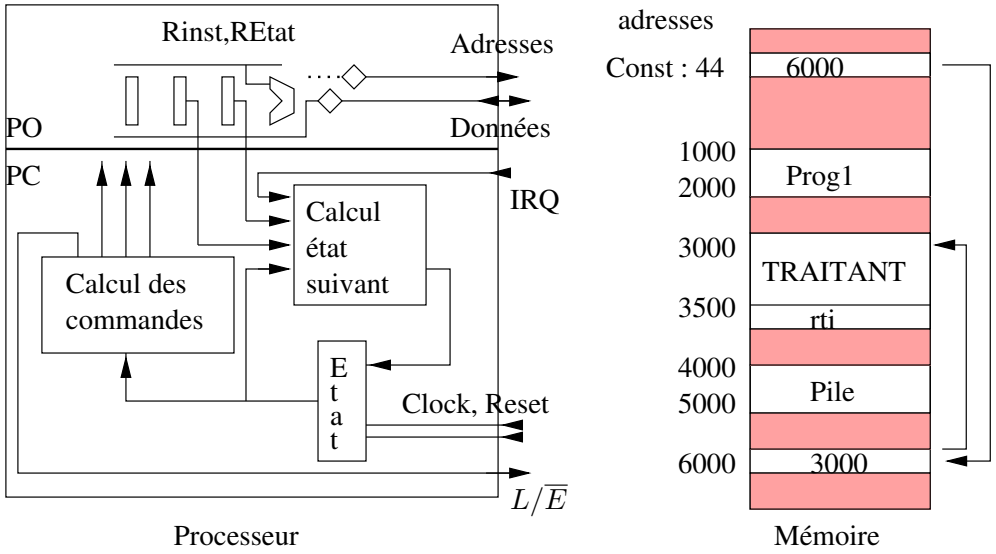


FIG. 22.4 – Le processeur, ses liens avec l'extérieur et le contenu de la mémoire.

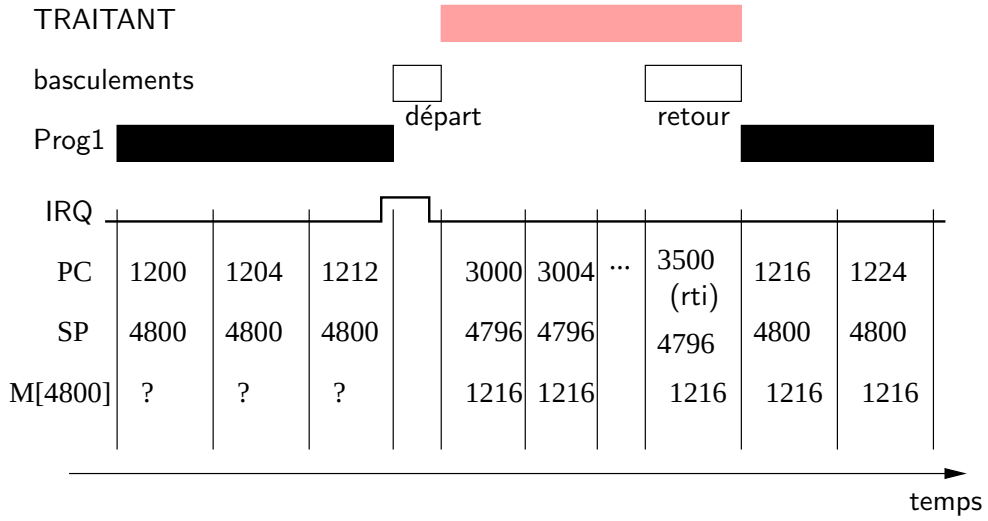


FIG. 22.5 – Diagrammes d'évolution des valeurs de PC et SP. Une tranche de temps verticale pleine correspond à l'exécution d'une instruction. La tranche verticale vide correspond au basculement de départ. Les basculements de départ et de retour sont mis en évidence. Arbitrairement certaines instructions sont sur 1 mot, d'autres sur 2.

5. Combien de temps dure le basculement de départ ? Il n'y a pas de règle absolue. Sur notre exemple il dure 5 états, c'est-à-dire un temps du même ordre que l'exécution d'une instruction.
6. Pourquoi n'y a-t-il pas une adresse de début du TRAITANT fournie lors du basculement de départ ? Parce que ce départ n'est provoqué que par le changement d'état d'un fil. Un fil ne peut pas coder une adresse. On pourrait imaginer *plusieurs* fils codant cette adresse, mais d'où viendrait-elle ? On pourrait imaginer que l'adresse est quelque part dans le programme . . .mais ce n'est pas possible car l'interruption arrive à l'insu du programme.
7. Ne pourrait-on avoir un mécanisme analogue pour un deuxième fil IRQ′ ou pour un événement imprévu interne au processeur ? La réponse est oui et l'étude en est faite au chapitre 24. Il y aura aussi une instruction dont le déroulement provoque le basculement de départ. De même on verra que la présence d'une instruction invalide peut provoquer un basculement de départ.
8. reset est-il une interruption ? Un signal reset fait repartir le processeur dans son état initial. Il interrompt bien l'exécution. Il y a une sorte de basculement de départ (PC remis à 0) mais pas de basculement de retour correspondant.

   Normalement sur un ordinateur, le reset provient soit d'un petit montage électronique qui envoie une impulsion à la mise sous tension, soit d'un bouton à la disposition de l'utilisateur.
9. Que se passe-t-il si une requête survient juste entre une instruction qui modifie les indicateurs Z, N, C et V et une instruction de branchement qui les exploite, alors que le sous-programme TRAITANT peut modifier les valeurs de ces indicateurs ? C'est une grosse faute dans la programmation du traitant. Au retour du traitant, le branchement conditionnel se fait sur la base de valeurs des indicateurs qui ne sont pas les bonnes.

Plus généralement, TRAITANT est comme tout sous-programme, il peut modifier les valeurs contenues dans les registres, le *contexte d'appel*. Il convient alors de sauvegarder et de restaurer ce contexte même si le terme d'appel est incorrect ici. On n'a pas précisé exactement les registres concernés par les instructions de chargement/rangement des registres. Si c'est possible, une solution pour se prémunir contre le problème est que le sous-programme TRAITANT commence par sauvegarder le mot d'état et se termine en restaurant ce registre. REtat fait en effet partie du contexte du programme interrompu.

On notera que certains processeurs offrent cette sauvegarde/restauration du mot d'état dans le basculement de départ et dans l'exécution du `rti` sans besoin d'instructions d'empilement/dépilement du mot d'état. C'est le choix que nous ferons dans la version plus complète (Cf. Figure 22.6).

Le choix est différent selon les processeurs. Le SPARC ne sauvegarde pas le mot d'état, il faut le programmer dans le sous-programme TRAITANT, le

6502 ne sauvegarde que le compteur programme et le registre d'état, le 6809 sauvegarde tous les registres lors du basculement. Quel que soit le choix, le basculement de retour a évidemment l'action symétrique adéquate. Cela modifie la durée des basculements. Le programmeur de traitants d'interruption doit être vigilant au choix fait sur sa machine.

2.4 Problèmes de synchronisation entre le processeur et le monde extérieur

Comme dans toute communication, il faut se préoccuper de synchronisation. Le monde extérieur, qui émet les requêtes d'interruption, est le maître de l'échange. Le processeur n'a qu'à suivre le rythme s'il peut. S'il ne peut pas, tant pis. L'extérieur ne vérifie pas. Ce cas peut présenter deux symptômes :

1. Le fil IRQ est à 1 après l'exécution de la première instruction du sous-programme TRAITANT. Le processeur empile à nouveau une adresse de retour et reforce l'adresse de début de TRAITANT dans le PC. Ce n'est peut-être pas ce que l'on cherche. Dans le paragraphe suivant, on montrera comment éviter cela.

   On notera deux raisons possibles pour que IRQ soit encore à 1 : soit il est resté à 1, soit il est passé à 0 pendant un instant puis remonté à 1.

2. Le fil IRQ est à 0 après la première instruction de TRAITANT mais repasse à 1 pendant l'exécution de TRAITANT. Le processeur interrompt le traitement en cours. Il y a donc deux occurrences différentes de TRAITANT en cours d'exécution, comme pour un appel de sous-programme récursif. Ce n'est en général pas souhaité.

Le monde extérieur et le processeur se synchronisent par un protocole simplifié de poignée de mains. Le processeur n'accepte de nouvelle requête que lorsqu'il a terminé de traiter la précédente. Pour une vraie poignée de mains, il reste à l'extérieur à n'émettre une requête que si la précédente a été traitée. C'est l'objet du paragraphe suivant.

2.5 Solutions aux problèmes de synchronisation

Les différents problèmes de synchronisation évoqués sont traités par l'introduction d'un bit, conventionnellement nommé `I`, comme *Inhibé*, contenu dans le mot d'état.

Les différentes modifications pour traiter la synchronisation apparaissent dans le nouveau graphe de contrôle figure 22.6.

Quand `I` est à vrai, le processeur ne prend pas en compte les requêtes d'interruption. On dit que les interruptions sont *masquées*. Quand `I` est à faux, le processeur considère les requêtes.

Au moment du basculement de départ en interruption, le mot d'état et l'adresse de retour sont sauvés dans la pile puis le bit `I` est mis à vrai. Pen-

dant toute l'exécution de TRAITANT, le bit `I` reste à vrai et aucune demande d'interruption ne sera prise en compte : en sortie d'un état `ope` l'état suivant est forcément l'état d'acquisition du code opération comme on le voit sur la figure 22.6.

Au moment du basculement de retour, l'ancienne valeur de `I` est restaurée grâce à la restauration du mot d'état. C'est ce qui guide le choix que nous avons fait de sauver le registre d'état par sauvegarde automatique.

Avec cette solution, l'extérieur peut émettre des requêtes à tout moment, c'est le processeur qui reste maître de l'échange.

Deux instructions supplémentaires sont ajoutées : `clri` et `seti` pour CLeaRI et SetI. Elles permettent de mettre `I` à vrai ou à faux. Cela permet au programmeur d'autoriser la prise en compte d'interruptions pendant l'exécution de TRAITANT ou, à l'inverse, de rendre certaines portions de code ininterruptibles. La nécessité de ces instructions apparaît dans le paragraphe 4.

Par ailleurs, pour pouvoir mettre au point les programmes TRAITANT, dont la terminaison est nécessairement un `rti`, il est très commode de pouvoir déclencher le basculement de départ par une instruction. Nous ajoutons cette instruction nommée ici `swi` (pour SoftWare Interrupt). Cette instruction sera aussi utilisée pour forcer volontairement une interruption dans les chapitres ultérieurs.

3. Un exemple détaillé d'utilisation : mise à jour de la pendule

Nous présentons ici une utilisation typique du système d'interruptions. Nous choisissons la fonction de gestion de l'heure courante. Dans un premier temps cette fonction est présentée *sans* faire appel au mécanisme d'interruptions (ce qui n'est pas réaliste), puis ce mécanisme est introduit.

3.1 Réalisation de la pendule sans interruption

On suppose l'ordinateur doté d'un oscillateur générant un signal périodique, qui délivre une impulsion toutes les secondes. On ne connait pas trop pour le moment la durée de cette impulsion. On verra plus tard (Cf. Paragraphe 3.3).

On veut réaliser une pendule logicielle, qui maintient l'heure courante dans trois variables globales du système : `heures`, `minutes` et `secondes`. Ces variables sont utilisées par beaucoup de fonctions du système d'exploitation : envoi de courriers, datation des fichiers, ...

Cette pendule logicielle est un programme qui incrémente l'heure courante toutes les secondes. On utilise pour cela une procédure qui fait avancer la représentation de l'heure d'une seconde. On donne figure 22.7 le programme en langage d'assemblage. On se resservira de cette procédure. Naturellement elle pourrait être écrite dans un langage de haut niveau.

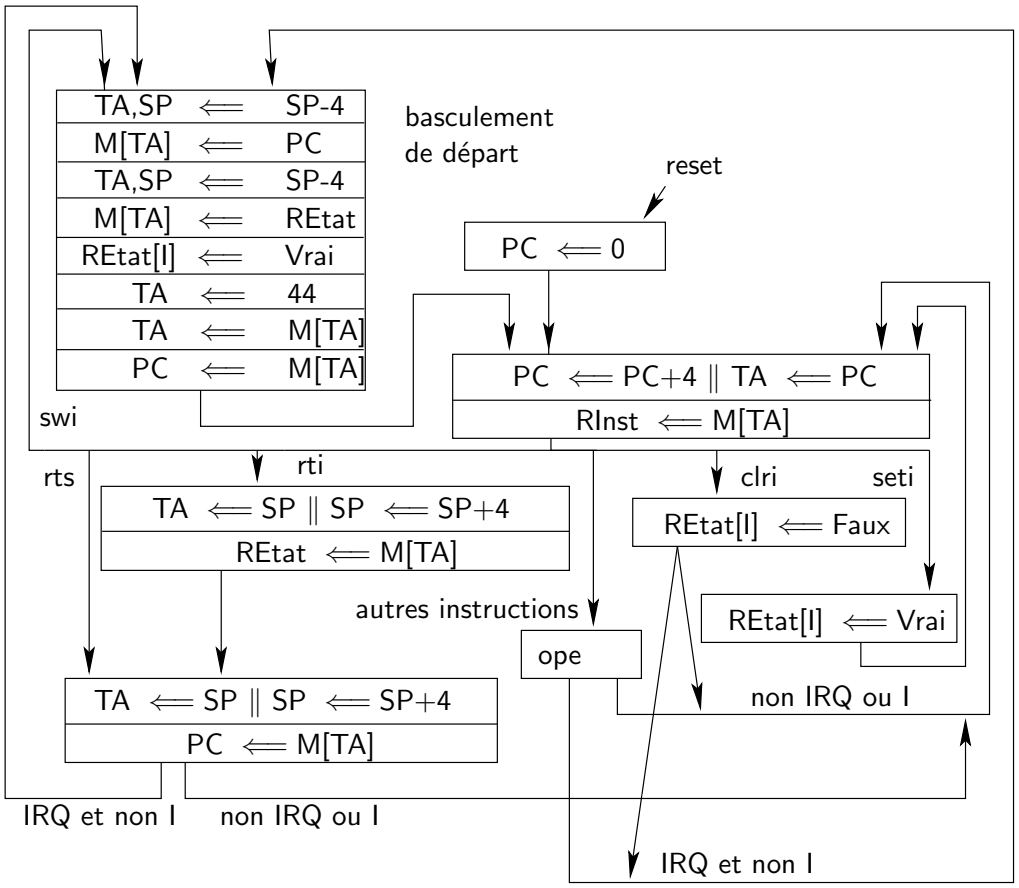


FIG. 22.6 – Graphe de contrôle avec tout le traitement du mécanisme d'interruption. Les instructions non liées aux interruptions ne figurent plus.

```
         .data
secondes :long 0
minutes :.long 0
heures : .long 0
         .text
plus1sec /* utilise R4 comme registre de travail */
         ld [secondes], R4   /* ou [R0 + secondes] */
         add R4, 1, R4       /* secondes = secondes +1 */
         st R4, [secondes]
         subcc R4, 60, R0    /* si sec < 60 : terminé */
         blu retour
         st R0, [secondes]   /* secondes = 0 */
         ld [minutes], R4
         add R4, 1, R4       /* minutes = minutes +1 */
         st R4, [minutes]
         subcc R4, 60, R0    /* si min < 60 : terminé */
         blu retour
         st R0, [minutes]    /* minutes = 0 */
         ld [heures], R4
         add R4, 1, R4       /* heures = heures +1 */
         st R4, [heures]
         subcc R4, 24, R0    /* si heures < 24 : terminé */
         blu retour
         st R0, [heures]     /* heures = 0 */
retour : rts
```

FIG. 22.7 – Procédure d'incrémentation de 1 seconde

```
while (1)
{    /* attendre une impulsion sur le signal */
     while (signal == 0) ; /* boucle vide ; sortie si signal = 1 */
     while (signal == 1) ; /* boucle vide ; sortie si signal = 0 */
     /* il y a bien eu une impulsion. */
     plus1sec () ;
}
```

FIG. 22.8 – Programme pendule

Le squelette du programme pendule, sans utilisation du mécanisme d'interruption, pourrait être celui de la figure 22.8 où `signal` est le nom d'une variable correspondant au signal périodique qui peut être lu par le processeur.

A l'évidence ce programme principal utilise toutes les ressources d'un ordinateur, mais peu efficacement ! Naturellement il n'est pas question que l'ordinateur scrute en permanence le circuit générateur d'impulsion et ne fasse rien d'autre.

3.2 Introduction du mécanisme d'interruption

La pendule que nous cherchons à réaliser est une utilisation typique d'un système d'interruption. Le signal IRQ `est` directement l'impulsion. Le mécanisme d'interruption permet d'incrémenter l'heure à l'insu de l'utilisateur. Les programmes de l'utilisateur s'exécutent presque normalement, ils sont interrompus une fois par seconde, les variables heures, minutes et secondes sont mises à jour et le programme interrompu reprend son exécution. Le programme TRAITANT est approximativement le sous-programme `plus1sec` donné figure 22.7.

Supposons par exemple que nous voulions évaluer la durée d'un calcul dans un programme en C. La tâche, donnée figure 22.9, consiste tout simplement à lire l'heure courante en fin de calcul et à retrancher l'heure de début du calcul.

Sans mécanisme d'interruption, l'unique processeur de l'ordinateur peut exécuter un seul programme, le calcul ou la pendule, mais pas les deux à la fois. La mesure devient possible en utilisant le signal périodique comme signal d'interruption et en exécutant l'incrémentation de l'heure dans le corps du traitant d'interruption.

Le traitant d'interruption donné figure 22.10 se résume à un appel à la procédure d'incrémentation, encadré par un prologue et un épilogue de sauvegarde dans la pile et de restauration des registres utilisés par la procédure. Au lieu d'appeler la procédure plus1sec, comme on le fait ici, il serait possible de simplement écrire ses instructions dans traitant.

Le chronogramme de la figure 22.11 illustre le déroulement d'un calcul durant un peu plus de deux secondes. A l'instant initial i précédant le calcul, la lecture de l'heure courante donne l'heure de début de calcul (3 :59 :57). L'arrivée du signal IRQ provoque un départ en interruption aux instants d_1, d_2 et d_3 et suspend l'exécution du calcul le temps d'incrémenter l'heure courante.

Lors des retours d'interruption aux instants r_1, r_2 et r_3 le calcul reprend à la nouvelle heure courante. Après lecture de l'heure courante en fin de calcul à l'instant f (4 :00 :00), la soustraction de l'heure de début donne la durée d'exécution, soit 3 secondes.

Le lecteur sourcilleux aura noté que le temps mesuré pour le calcul comporte en fait aussi le temps de mise à jour de l'heure par les différentes exécutions de traitant. Il en concluera, à juste titre, que la mesure du temps d'exécution d'un programme est une tâche délicate. On doit être conscient de l'ordre de

```
SecondesDébut= secondes ;
MinutesDébut = minutes ;
HeuresDébut  = heures ;
             CALCUL ; /* Le calcul proprement dit */
SecondesFin  = secondes ;
MinutesFin   = minutes ;
HeuresFin    = heures ;
Durée        = SecondesFin - SecondesDébut
             + 60 * (MinutesFin − MinutesDébut) ;
             + 3600 * (HeuresFin − HeuresDébut) ;
```

FIG. 22.9 – Mesure d'un temps de calcul

```
traitant :
Prologue :add sp, -4, sp
          st R4, [sp]          /*empiler Reg de travail*/
Appel :   /* utilise le Reg de travail R4 */
          jsr plus1sec
Epilogue  ld [sp] , R4
          add sp, +4, sp       /*dépiler Reg de travail*/
          rti
```

FIG. 22.10 – Traitant d'interruption de mise à l'heure

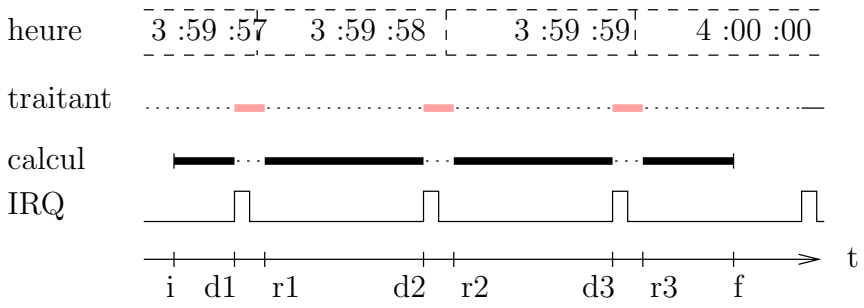


FIG. 22.11 – Chronogramme d'exécution de la pendule fonctionnant par interruption

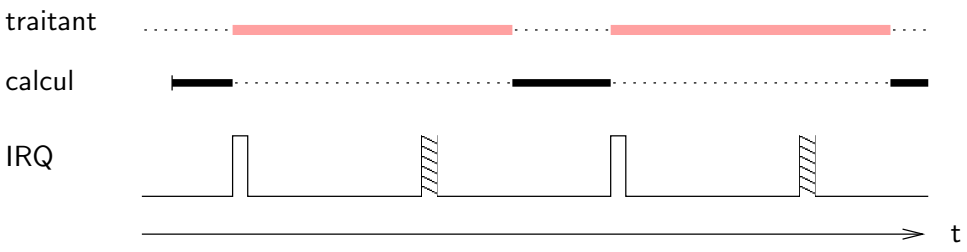


FIG. 22.12 – La fréquence des interruptions dépasse la capacité de traitement du processeur : les impulsions hachurées ne sont pas prises en compte.

grandeur du temps de la mesure par rapport au temps du calcul.

3.3 Hypothèses temporelles nécessaires au bon fonctionnement

Il pourrait être intéressant de disposer d'une pendule d'une précision supérieure. Le principe de réalisation peut être conservé : il suffit d'augmenter la fréquence du signal d'interruption et de gérer une variable supplémentaire donnant les fractions de secondes. Nous pourrions ainsi imaginer la réalisation par logiciel d'une pendule indiquant les millisecondes voire les microsecondes.

Toutefois la vitesse de calcul du processeur est constante et la fréquence du générateur d'impulsions ne peut pas être augmentée ainsi indéfiniment. Plus nous voulons augmenter la précision de la pendule, plus sa gestion consomme de temps de calcul et ralentit l'exécution du calcul dont il faut mesurer la durée. Ce que l'on mesure devient de plus en plus imprécis. Il y a plus grave : la modification inconsidérée des caractéristiques des impulsions par rapport aux possibilités du processeur peut conduire hors d'un fonctionnement normal.

Le système présenté a des limites et nous allons examiner quelques-unes de ces situations.

3.3.1 Les impulsions ne doivent pas être trop fréquentes

Les interruptions ne doivent pas survenir avec une période inférieure au temps d'exécution du traitant plus celui des deux basculements départ et retour. Dans le cas contraire, évidemment, des requêtes finiraient par ne pas être satisfaites.

Cette contrainte temporelle reflète simplement le fait que, à la fréquence d'interruption maximale la modification périodique de l'heure courante consomme la totalité du temps du processeur et ne peut plus être accélérée. Si la fréquence est trop élevée, certaines impulsions sont ignorées (voir chronogramme 22.12).

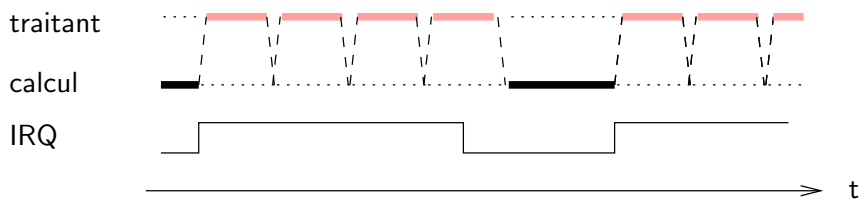


FIG. 22.13 – L'impulsion du signal d'interruption, trop longue, est comptabilisée 4 fois.

3.3.2 Les impulsions ne doivent pas être trop longues

La durée des impulsions ne doit pas excéder le temps d'exécution du traitant. Au moment du départ en interruption, l'action du signal IRQ est neutralisée par la mise à 1 du bit I du registre d'état (voir chronogramme 22.13). Mais, à la fin du traitant, l'instruction de retour `rti` rétablit l'ancienne valeur du registre d'état avec I à 0. Si le signal IRQ est encore à 1, il déclenche à nouveau un départ en interruption. La même impulsion est comptabilisée plusieurs fois.

Parmi les remèdes envisageables, on peut construire un oscillateur matériel délivrant un signal de forme adéquate. L'oscillateur peut également être muni d'une entrée de remise à 0 de IRQ, activée par une instruction du traitant ou par un signal d'acquittement émis par le processeur au moment du départ en interruption. On retrouve alors une solution basée sur un protocole de dialogue en poignée de mains (Cf. Chapitre 6).

Une solution purement matérielle serait de calibrer la durée de l'impulsion en insérant un détecteur de front (Cf. Chapitre 9) cadencé par un sous-multiple de l'horloge du processeur.

3.3.3 Les impulsions ne doivent pas être trop courtes

A contrario (voir chronogramme 22.14), le processeur peut ignorer une impulsion trop courte. En effet, le processeur ne teste IRQ qu'au moment où il s'apprête à lire une nouvelle instruction. L'instruction la plus longue à exécuter définit la durée minimale des impulsions pour garantir leur prise en compte. Dans notre exemple il s'agit de `jsr` qui s'exécute en sept cycles (Cf. Figure 22.2). `rti` dure un cycle de plus que `jsr`, mais l'arrivée d'une impulsion pendant l'exécution de `rti` poserait déjà un problème de période du signal IRQ inférieure à la durée du traitant.

Là aussi la solution consiste à exiger que l'émetteur d'interruption maintienne sa requête tant qu'elle n'a pas été prise en considération. Cela revient, encore, à une poignée de mains.

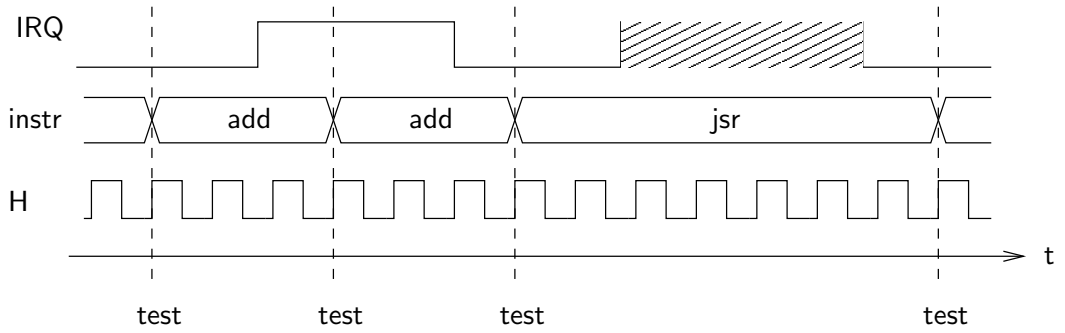


FIG. 22.14 – L'impulsion d'une durée de 4 cycles d'horloge du processeur est prise en compte si l'instruction en cours est `add` et ignorée parce que trop courte dans le cas de `jsr`. Sur cette figure, le départ en interruption entre les deux instructions `add` n'est pas représenté.

3.4 De l'interruption périodique au partage de temps généralisé

La pendule est un exemple de partage de temps entre différentes activités concurrentes. Observé à une échelle de temps macroscopique, notre ordinateur semble exécuter deux programmes en même temps : le calcul et la mise à jour de l'heure.

La pendule met en oeuvre une forme très rudimentaire de partage de temps : il n'y a que deux activités concurrentes et l'un des programmes est une boucle sans fin. Dans le chapitre 23, nous verrons comment étendre le mécanisme de partage de temps à un nombre quelconque d'activités concurrentes (les processus) et à des programmes ordinaires (non limités à des boucles sans fin).

4. Notion de concurrence et d'atomicité des opérations

La pendule est un premier exemple de partage de temps du processeur entre deux programmes *concurrents* qui entrent en compétition pour son utilisation : le programme de calcul et le sous-programme de mise à jour de l'heure courante.

Le partage de variables entre programmes concurrents pose un problème d'atomicité des opérations sur ces variables, qui peut se traduire par des valeurs incohérentes de ces variables.

Soit une opération de consultation ou de modification d'un ensemble de variables effectuée par un programme. L'atomicité de l'opération signifie que tout autre programme concurrent ne peut accéder aux variables qu'à des instants tels que l'opération est soit réalisée entièrement, soit pas commencée du tout.

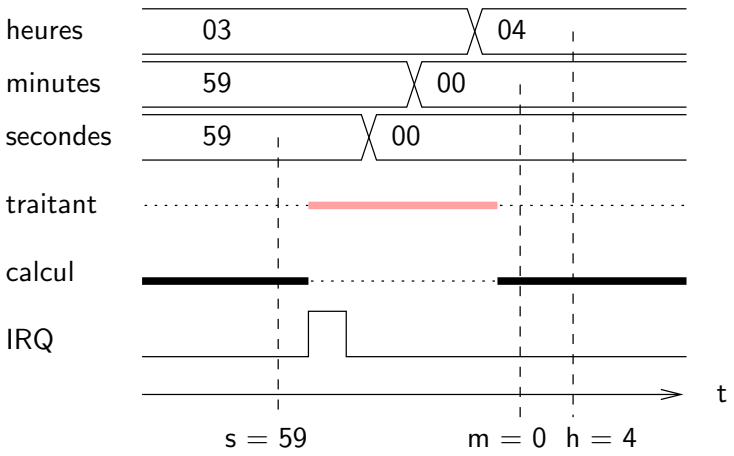


FIG. 22.15 – Problèmes d'atomicité : l'interruption arrive après que le programme a lu les secondes et avant qu'il lise les minutes. L'heure lue est 4 :00 :59 au lieu de 3 :59 :59 ou 4 :00 :00.

Supposons que l'on réalise un transfert de 1000 francs d'un compte bancaire X crédité de 10000 francs vers un compte bancaire Y crédité de 2000 francs. Le transfert consiste à débiter X de 1000 francs puis à créditer Y de la même somme. De l'état initial (10000,2000), le couple (X,Y) passe par un état transitoire (9000,2000) pour atteindre l'état final (9000,3000).

On dit que l'opération de transfert est *atomique*, si un programme s'exécutant en concurrence avec celui qui réalise le transfert peut observer (X,Y) dans l'état initial ou dans l'état final, mais pas dans l'état transitoire (9000,2000).

Dans notre exemple de la pendule, le programme de calcul peut lire une heure erronée. Supposons qu'une interruption arrive à 3h 59mn 59s où le programme termine son calcul et lit l'heure courante pour en calculer la durée (figure 22.15).

Le scénario suivant montre que la lecture de l'heure courante n'est pas atomique et peut rendre une valeur erronée.

Le programme de calcul lit les secondes (Instruction `SecondesFin = secondes`) et trouve 59. L'interruption est prise en compte juste après et fait passer l'heure courante à 4 :00 :00. Au retour d'interruption, le programme interrompu poursuit sa consultation de l'heure courante et lit les minutes (00) et les heures (4). L'heure obtenue par le programmme n'est ni 3 :59 :59, ni 4 :00 :00, mais 4 :00 :59. C'est une erreur. Le même genre de problème peut être mis en évidence avec deux mises à jour concurrentes. Sur un ordinateur simple à un seul processeur, le remède consiste à suspendre la prise en compte des interruptions pendant l'accès aux variables partagées pour le rendre atomique. C'est la principale utilité des instructions `clri` et `seti` que nous avons décrites au paragraphe 2.5.

5. Exercices

E22.1 : Choix de l'adresse de début du traitant
Dans le basculement de départ apparaît une double indirection pour l'obtention de l'adresse de début du traitant. Typiquement de la mémoire morte correspondrait à l'adresse 44, et de la mémoire vive à l'adresse 6000. L'utilisateur peut ainsi écrire à l'adresse 6000 l'adresse de son choix (ici 3000) où commence le traitant.

Que faut-il modifier dans le graphe de contrôle si l'on suppose qu'un traitant débute nécessairement à l'adresse 36848 ? Cela offre-t-il la même souplesse d'emploi ? Comment pourrait-on modifier la machine pour que le traitement du IRQ soit (doublement) indirectement référencé par l'adresse 44 et celui du `swi` par l'adresse 48 au lieu de l'être tous les deux par 44 (Cf. Figure 22.6) ?

E22.2 : Et en vrai ?
Se procurer dans les bibliothèques et librairies (papier ou électroniques) les modes d'emploi détaillés de processeurs simples. Etudier leur système d'interruptions. Prolonger avec des processeurs complexes. Prolonger sur des ordinateurs.

Chapitre 23

Partage de temps et processus

Dans le chapitre 22, nous avons développé un exemple simple de partage de temps du processeur entre une tâche de fond réalisant un calcul et un travail périodique de mise à l'heure de la pendule.

Dans ce chapitre, nous montrons que le mécanisme d'interruption permet de réaliser un partage de temps entre plusieurs travaux, dont les nombre, nature et durée sont quelconques.

A chaque exécution de programme correspond une entité dynamique. Le nom consacré par la littérature pour désigner cette entité est *processus*, mais on peut aussi trouver d'autres termes tels que *tâche* ou *activité* .

Lors de l'exécution d'un programme le système doit maintenir un certain nombre d'informations (adresse de chargement, fichiers ouverts, etc.). Par souci de précision, dans ce chapitre, la notion de *processus* est implicitement associée à la gestion interne des structures de données du système d'exploitation et *activité* renvoie au contraire à une vision externe du comportement du système et de l'avancement des programmes exécutés. Cette distinction permet de lever les ambiguïtés sur la définition des instants auxquels le processeur passe d'un programme à l'autre.

> *Le paragraphe 1. présente le mécanisme de partage du temps du processeur entre plusieurs activités. Le paragraphe 2. précise les structures de données associées à chaque processus. Le paragraphe 3. détaille le mécanisme de commutation d'un processus à un autre. La création d'un processus et sa terminaison font l'objet du paragraphe 4.*

1. Principe et définitions

Les systèmes dits multitâches sont capables d'exécuter plusieurs programmes à la fois. Ils sont très utiles et permettent à plusieurs utilisateurs de travailler en même temps sur un ordinateur. Ils autorisent également un utilisateur à faire plusieurs choses à la fois, par exemple éditer un fichier pendant la compilation d'un autre fichier et lire en même temps l'heure affichée

par un troisième programme. Nous restons toutefois ici dans le contexte des machines à un seul processeur.

Un programme est une entité statique composée d'instructions décrivant une suite d'opérations à réaliser pour aboutir au résultat recherché. A chaque exécution d'un programme correspond une activité et un processus associé dans les tables du système d'exploitation, créés lors du lancement du programme et détruits quand l'exécution du programme se termine.

Il ne faut pas confondre programme avec processus ou activité. Si un utilisateur lance une compilation et une édition, il y a deux activités et le système d'exploitation crée deux processus exécutant deux programmes différents (le compilateur et l'éditeur). A deux compilations simultanées correspondent deux processus distincts partageant le même fichier exécutable mais des données différentes.

Les ordinateurs capables d'exécuter réellement plusieurs instructions dans plusieurs processeurs sont des machines à architecture dite parallèle, dotées d'autant de processeurs que de calculs simultanés. On trouve dans le commerce de telles machines, puissantes mais onéreuses, inadaptées pour un utilisateur qui n'a pas besoin d'une grande puissance de calcul mais qui ne veut pas être bloqué durant une compilation.

Les systèmes d'exploitation multitâches ont été conçus pour ce genre d'utilisation. Ils gèrent un partage de temps du processeur et donnent aux utilisateurs l'impression de disposer d'autant de processeurs (virtuels) que de travaux à mener de front. Les contraintes matérielles sur l'ordinateur se résument essentiellement à une taille de mémoire suffisante pour contenir l'ensemble des programmes à exécuter et à l'existence d'un mécanisme d'interruption.

La présentation du mécanisme de partage de temps dans ce paragraphe et les paragraphes 2. et 3. suppose que les programmes ont déjà été lancés et que les processus préexistent.

1.1 Mécanique d'entrelacement par interruption périodique

L'horloge de l'ordinateur génère un signal d'interruption périodique. On prendra garde à ne pas confondre l'horloge de l'ordinateur, de période de l'ordre de quelques millisecondes, avec celle du processeur de période de l'ordre de quelques nanosecondes. Le traitant de cette interruption réalise le partage du temps entre plusieurs activités. Le chronogramme de la figure 23.1 illustre ce partage entre trois activités a1, a2 et a3 (et leurs processus associés p1, p2 et p3), élection étant la partie du traitant qui effectue le choix de la nouvelle activité.

L'observation du système débute alors que le processeur est attribué à l'activité a1. L'exécution de a1 est suspendue par le premier départ en interruption (instant di_1) et sera reprise au retour de la troisième interruption (instant ri_3).

Au retour de la première interruption (instant ri_1), le processeur reprend

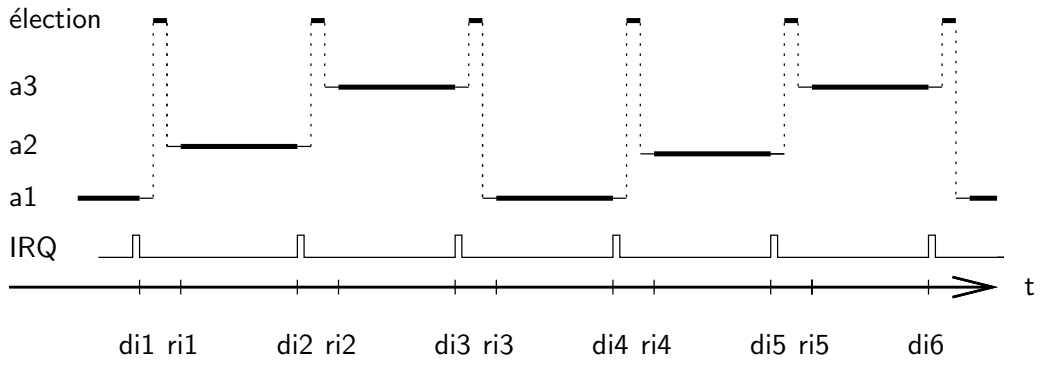


FIG. 23.1 – Multiplexage du processeur entre trois activités

l'exécution de l'activité a2. Cette dernière est à son tour suspendue par le deuxième départ en interruption (di_2) et sera reprise lors du quatrième retour d'interruption (ri_4).

L'intervalle de temps entre les deuxième et troisième interruptions est utilisé par a3, dont l'exécution sera reprise puis suspendue à nouveau aux instants respectifs ri_5 et di_6.

Le même scénario se reproduit toutes les trois interruptions et le processeur travaille à tour de rôle pour la première activité, la deuxième, et ainsi de suite jusqu'à la dernière, après quoi le processeur est réaffecté à la première activité et cette séquence se reproduit à l'identique dans un nouveau cycle.

Le processus associé à l'activité en cours d'exécution est le processus *actif* ou *élu*. Les autres processus sont dits *prêts* (sous entendu à poursuivre leur exécution lorsque le processeur leur sera attribué de nouveau). Dans le paragraphe 3. du chapitre 24 nous verrons que les processus peuvent aussi se bloquer en attente d'événements tels que des fins d'entrées/sorties par exemple.

On parle de *commutation d'activité*, ou de *commutation de processus*, pour désigner le passage d'une activité, et évidemment d'un processus, à une autre. La figure 23.2 en détaille le déroulement. Cette commutation est déclenchée par une interruption. Le TRAITANT est alors appelé traitant de commutation.

Lors du retour d'interruption, une activité doit retrouver la machine dans l'état où elle l'a laissée lors de sa suspension. L'état de la machine est sauvegardé dans des structures de données que nous étudions en détail au paragraphe 2..

La première étape de la commutation (s) est représentée en traits fins sur les figures 23.1 et 23.2. Elle sauvegarde le contexte (essentiellement le contenu des registres du processeur) de l'activité suspendue (a2) dans les structures de données du processus actif (p2).

Le mécanisme de départ en interruption intégré au processeur sauvegarde automatiquement certains registres. La sauvegarde des autres registres est explicitement programmée dans le prologue du traitant de commutation.

Le déroulement de l'interruption se termine par une séquence symétrique de

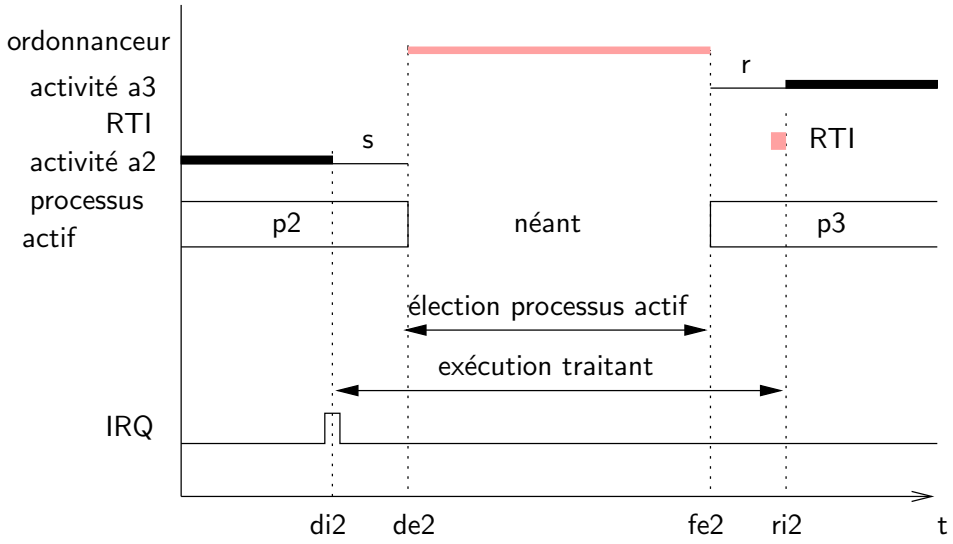


FIG. 23.2 – Détail de la commutation entre les activités a2 et a3

restauration (r) du contexte du (nouveau) processus actif (p3) depuis les structures de données convenables. Une partie de cette séquence est programmée dans l'épilogue du traitant, et l'autre réalisée automatiquement par l'instruction `rti`. On a évidemment symétrie par rapport au départ.

Le système d'exploitation comporte des procédures chargées d'élire un nouveau processus actif. Ces procédures constituent l'*ordonnanceur*. Le corps du traitant appelle les procédures de l'ordonnanceur.

L'exécution des programmes des utilisateurs, l'activité, cesse dès le départ en interruption et ne reprend qu'au retour d'interruption. Mais on notera que du point de vue de la gestion des structures de données du système, la commutation effective de processus a lieu entre la fin de la sauvegarde et le début de la restauration, qui utilisent les structures de données de l'ancien (p2) et du nouveau (p3) processus actifs.

L'ordonnanceur n'est pas une activité comme une autre. Il ne résulte pas du lancement d'une commande et le système ne maintient pas de structures de données pour lui ; ainsi, l'ordonnanceur n'est pas un processus.

On note une différence par rapport à l'exécution d'un `rti` normal : ici l'activité reprise n'est pas nécessairement celle qui a été interrompue lors du départ en interruption. L'ordonnanceur peut choisir et installer une *autre* activité.

1.2 Notion de politique d'ordonnancement

La stratégie de répartition du temps du processeur aux différents processus est appelée ordonnancement (scheduling en anglais) des processus. Deux aspects sont à considérer : l'origine de la décision d'effectuer une commutation et, une fois cette décision prise, l'algorithme d'élection du nouveau processus.

L'origine de la commutation peut se faire avec ou sans réquisition.

Il existe des politiques d'ordonnancement *sans* réquisition : le processus actif décide lui-même de l'instant auquel il libère le processeur et appelle explicitement la séquence d'instructions qui réalise la commutation. De telles politiques autorisent malheureusement l'écriture (volontaire ou non) de programmes monopolisant le processeur.

Au contraire, dans les politiques *avec* réquisition du processeur, les commutations sont forcées par interruption du processus actif. Dans le paragraphe précédent, nous avons décrit une allocation avec réquisition par tranches de temps (cadencée par un signal d'interruption périodique).

L'ordonnanceur attribue à tour de rôle un quantum de temps aux processus pour qu'ils *accèdent* au processeur.

L'algorithme d'élection illustré par la figure 23.1 est très simple : les processus accèdent à tour de rôle au processeur selon un ordre *premier arrivé, premier servi*. Cette méthode simple d'ordonnancement est connue sous le nom d'allocation en tourniquet.

Il existe des stratégies d'allocation plus élaborées que le tourniquet, gérant des priorités entre les processus, ces priorités pouvant être statiques ou évoluer dynamiquement en fonction du comportement du processus.

Dans ce chapitre, nous nous en tiendrons à l'allocation en tourniquet par tranche de temps. Dans cette politique simple d'ordonnancement, le seul paramètre de réglage est la durée d'une tranche de temps. Son choix résulte d'un compromis. La réduction de quantum tend à améliorer la capacité de réaction du système et son aptitude à supporter de nombreux travaux interactifs alors que la minimisation du coût de gestion du partage pousse au contraire à réduire autant que possible la fréquence des commutations.

Un choix judicieux de durée de tranche de temps donnera aux utilisateurs qui ne peuvent observer le système qu'à une échelle de temps macroscopique l'illusion d'une exécution simultanée des processus.

Considérons à titre d'exemple un système multitâches et des utilisateurs en train d'éditer des fichiers et une tranche de temps de deux millisecondes. Supposons que le travail de mise à jour de l'affichage après la frappe d'un caractère ne dure pas plus d'une tranche de temps. En supposant une vitesse maximale de saisie de dix caractères par seconde, le système pourrait supporter cinquante éditions simultanées avec un temps de réponse maximal d'un dixième de seconde et donner à chaque utilisateur l'illusion qu'il ne travaille que pour lui. Les programmes de calcul intensifs s'exécuteront évidement moins vite, mais ceci n'est pas trop gênant dans la mesure où les utilisateurs peuvent faire autre chose pendant les calculs de longue durée.

2. Structures de données associées aux processus

Nous supposons dans ce paragraphe que le départ en interruption sauve tous les registres du processeur dans la pile, excepté le registre pointeur de pile. De nombreux processeurs ne sauvegardent automatiquement que quelques registres. Il en va ainsi pour le processeur du chapitre 22, qui n'empile que le registre d'état et le compteur programme. On peut cependant toujours se ramener à l'équivalent d'une sauvegarde de l'ensemble des registres ordinaires lors du départ en interruption en insérant une action spéciale au début du prologue et à la fin de l'épilogue de TRAITANT, procédure qui complète la sauvegarde automatique effectuée par le processeur (Cf. Exercice E23.1).

2.1 Besoin de zones propres à chaque processus

La possibilité d'entrelacer les exécutions de plusieurs programmes présents simultanément en mémoire n'est pas une exclusivité des systèmes d'exploitation multitâches.

Le système simple monotâche décrit dans la partie V offre une telle possibilité : nous avons vu au chapitre 20 que l'interprète de commande lit une ligne de commande, déclenche le chargement-lancement du programme exécutable spécifié, attend la fin de son exécution et lit une nouvelle ligne de commande. Il y a ainsi entrelacement entre l'exécution de l'interprète et celui de la commande lancée. Mais l'entrelacement respecte une propriété *dernier lancé, premier terminé* qui permet de gérer l'allocation de mémoire et les exécutions de programmes comme des appels de procédures. Cette gestion repose sur une pile globale du système partagée par l'ensemble des programmes.

Cette organisation n'est pas applicable aux systèmes multitâches qui entrelacent autant d'exécutions de programmes que de processus indépendants, selon des politiques d'ordonnancement plus ou moins élaborées. Ces exécutions sont logiquement indépendantes les unes des autres et n'interfèrent entre elles que dans la mesure où elles entrent en compétition pour l'utilisation du temps du processeur. Il n'existe par exemple aucun lien entre les appels et retours de procédure respectifs de processus distincts. C'est pourquoi les processus sont dotées de *piles privées*.

L'exécution des programmes ne doit pas être modifiée, mais seulement ralentie, par le partage de temps et le support d'exécution initialement fourni par le système simple monotâche doit être reproduit dans le cadre de chacun des processus. La préservation du comportement des programmes exécutés implique en particulier que chaque processus dispose de ses propres zones de mémoire texte, données et pile privées auxquelles les autres processus ne doivent normalement pas accéder.

Il faut de plus que la valeur des registres observée par le programme ne soit pas modifiée par les diverses suspensions d'exécution. Chaque processus

est doté à cet effet d'une zone de mémoire privée destinée à la sauvegarde et à la restauration du contenu des registres, y compris le pointeur de pile SP (dans le prologue et l'épilogue du traitant de commutation).

Enfin il est nécessaire que la suspension n'ait pas d'incidence directe sur les entrées et sorties effectuées par le processus suspendu. Il suffit pour cela de doter les processus de tables de fichiers ouverts individuelles et de terminaux de dialogue distincts. Les systèmes multi-utilisateurs sont évidement dotés d'autant d'écrans-claviers que de personnes utilisant simultanément la machine. Mais lorsque les processus sont lancés par le même individu depuis le même écran-clavier, l'écran peut être divisé en fenêtres qui constituent autant de terminaux virtuels.

Quand un processus est actif, les registres du processeur (Rétat, PC, SP, registres données) contiennent des valeurs à jour. Quand le processus est inactif, les informations pertinentes sont en mémoire dans la zone de sauvegarde du processus.

> *On peut ainsi assimiler un processus à une machine virtuelle exécutant un programme, dont la mémoire est consituée des zones texte, données et pile allouées au processus, et dont le processeur est représenté par la zone de sauvegarde des registres en mémoire.*

2.2 Organisation effective des zones mémoire de chaque processus

2.2.1 Structure de données en mémoire

Une partie de la mémoire est occupée par les instructions et les variables globales du système d'exploitation : elle contient entre autres l'amorce de démarrage, les pilotes de périphériques, le système de gestion de fichiers, l'ordonnanceur et les traitants.

La figure 23.3 décrit les structures de données en mémoire représentant les processus. Le système d'exploitation gère une table proc dont la taille définit un nombre maximal de processus. Chaque entrée de cette table est une structure, nommée *descripteur* (de taille TPROC) qui décrit un processus. Les processus peuvent être identifiés par leur numéro d'entrée, ou pid (process identifier), dans cette table.

Dans la méthode du tourniquet, les processus dont l'exécution est suspendue sont chaînés dans une file des (processus) prêts. Deux variables globales du système mémorisent les pid respectifs du premier et du dernier processus de cette file (tête-prêts et queue-prêts). Le pid du processus élu (en cours d'exécution) est stocké dans la variable globale actif.

Le reste de la mémoire utilisée est structuré en autant d'exemplaires de zones texte, données et pile que de processus.

Le champ suivant du descripteur permet de lier les processus prêts entre eux dans la file des prêts. Le champ état du descripteur décrit le statut du

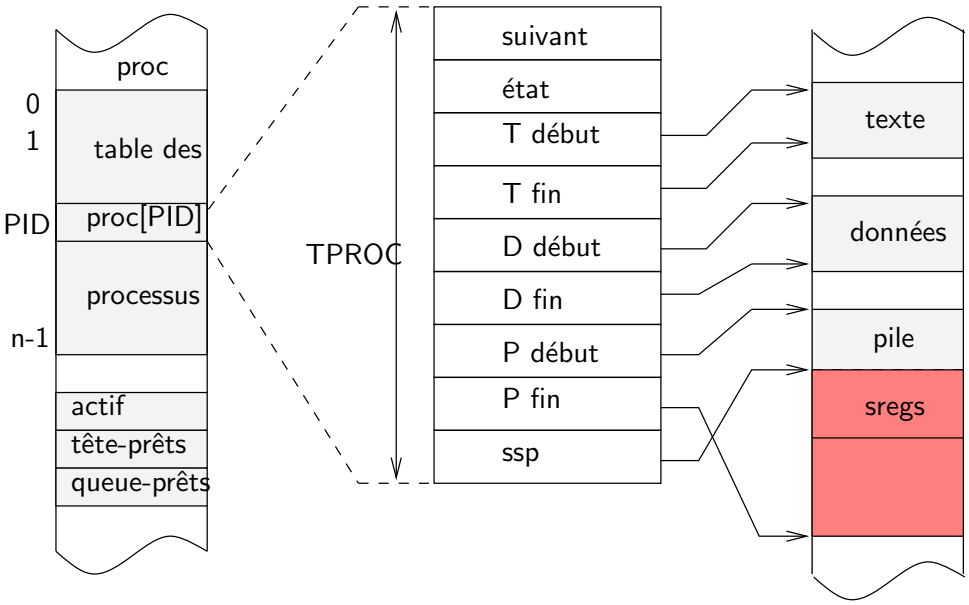


FIG. 23.3 – Représentation d'un processus en mémoire : à gauche les variables globales du système dont la table des processus, au centre le détail d'un descripteur de processus et à droite les zones de mémoires affectées à un processus (texte, données et pile du processus). La portion de zone pile déjà utilisée est représentée en gris plus foncé.

processus : actif, prêt ou inexistant (pour les entrées de la table encore inoccupées). Le descripteur contient aussi les adresses de début et de fin des zones texte (T), données (D) et pile (P) allouées au processus. Le champ ssp contient une copie du registre pointeur de pile SP.

2.2.2 Echanges entre les registres et les structures de données du processus

Quand le processus est actif, les valeurs à jour sont dans les registres du processeur. De plus, il existe en mémoire une zone de sauvegarde des registres dont le contenu n'est significatif que si le processus n'est pas actif. Cette zone comporte deux parties physiquement séparées.

Une première partie occupe le sommet de la zone de pile du processus. On la note sregs. Elle est destinée aux registres ordinaires dont le contenu est empilé lors du départ en interruption.

Une autre partie se réduit à la copie du registre pointeur de pile SP qui n'est pas sauvegardé au moment du basculement de départ en interruption. Elle est dans le champ ssp du descripteur du processus dans la table des processus. Ce champ ssp contient l'adresse de la zone sregs. Cette information est sauvegardée par le prologue du TRAITANT de commutation, et, symétriquement, restaurée par l'épilogue.

3. Organisation du traitant de commutation

Le traitant de commutation consiste principalement en l'élection d'un nouveau processus. Il contient de plus un prologue et un épilogue pour assurer la préservation des valeurs convenables du pointeur de pile.

Une partie du traitant de commutation déroule ainsi l'algorithme d'élection et manipule les tables de processus en mémoire. L'algorithme peut être plus ou moins complexe selon la politique d'ordonnancement adoptée. Par ailleurs, une même stratégie d'ordonnancement est susceptible d'être utilisée sur des machines différentes. Cette partie du traitant est réalisée sous la forme de procédures ordinaires écrites dans un langage de programmation (par exemple C) portable d'une machine à l'autre, et compilées.

L'autre partie du traitant, qui gère la mécanique de sauvegarde et de restauration et l'appel des fonctions de l'ordonnanceur, est nécessairement écrite en langage d'assemblage. Sa programmation nécessite en effet l'accès à des ressources autres que la mémoire (registres du processeur, notamment SP, masquage et démasquage des interruptions, retour d'interruption) que les langages de programmation évolués ne permettent pas de manipuler.

3.1 Election d'un processus

La première partie de la commutation (procédure actifversprêt) remet le processus actif suspendu dans l'ensemble des processus prêts. La seconde partie (procédure prêtversactif) sélectionne un des processus prêts et en fait le nouveau processus actif.

```
actifversprêt () {
  proc[actif].état = PRET;
  ajouter_prêts (actif);    }
```

```
prêtversactif () {
  actif = retirer_prêts ();
  proc[actif].état = ACTIF; }
```

Dans l'algorithme du tourniquet, l'ajout et le retrait portent respectivement sur la queue et la tête de la file des prêts.

```
ajouter_prêts(pid p) {
  proc[p].suivant = NULL;
  if (tête_prêts == NULL)
      tête_prêts = queue_prêts = p;
   else { proc[queue_prêts].suivant = p;
      queue_prêts = p;}  }

pid retirer_prêt () {
  pid élu;
  élu = tête_prêts;
  tete_prêts = proc[tete_prêts].suivant;
  return (élu);    }
```

3.2 Commutation de processus : une mise à jour de structures de données

La gestion du registre SP par le traitant a pour but de *berner* le processeur. Au basculement de départ, SP est celui de l'activité interrompue (soit A1, associée au processus p1). Les registres sont empilés dans la pile du processus p1. Le TRAITANT installe une nouvelle activité (soit A2) et modifie SP : la pile est alors celle du traitant. On a déjà vu que le TRAITANT n'est pas un processus. Au basculement de retour, `rti`, ce n'est pas l'ancienne activité A1 qui sera relancée, mais une nouvelle (soit A3, associée au processus p3). Les registres, en particulier le compteur programme, seront dépilés depuis la pile du processus p3.

On arrive ainsi à passer alternativement parmi plusieurs activités alors que le système d'interruptions n'est prévu que pour commuter entre UNE activité et un TRAITANT.

Si le processus actif est suspendu et réélu dans la foulée (c'est en particulier le cas s'il est le seul processus dans le système), le traitant de commutation se comporte comme un traitant d'interruption ordinaire : le processeur est rétabli dans son état d'avant l'interruption et le programme interrompu reprend normalement son exécution.

Dans le cas contraire, la restauration du registre SP le fera pointer sur la pile d'un autre processus et au retour du traitant le processeur reprendra l'exécution de cet autre processus (suspendu lors d'une interruption précédente).

Lors de la programmation du traitant on doit gérer deux problèmes :

- le passage d'une pile à une autre permettant à chaque processus de retrouver, lorsqu'il redevient actif, les registres du processeur dans l'état où ils étaient lorsque ce processus a été interrompu.
- la sauvegarde des adresses de retour et éventuellement de paramètres dans une pile, si l'on programme le traitant en réalisant des appels de procédures (Cf. Chapitres 12 et 13).

Dans la suite nous présentons la réalisation du traitant en deux versions. Une première version donne un squelette du traitant réalisé sans aucun appel de procédure. L'objectif est de mettre uniquement l'accent sur la gestion des piles du processus interrompu et du processus relancé. La deuxième version, correspondant mieux à ce qui est fait dans un système d'exploitation, est donnée complètement et utilise des appels de procédure pour chacune des parties du traitant.

Dans les deux versions, et dans chacune à plusieurs endroits, on retrouve des structures de données et un calcul identique : proc est l'adresse de la table des processus, actif est l'adresse du mot mémoire contenant le pid du processus actif, TPROC est une constante donnant la taille du descripteur d'un processus et deltassp est le déplacement du champ ssp du descripteur.

Les programmes sont écrits en assembleur du processeur décrit dans le chapitre 22. Le registre R0 contient la constante 0 et n'a pas besoin de sauvegarde.

Le fragment de code suivant range dans le registre R1 l'adresse du descripteur du processus actif :

```
set actif, R1
ld [R1], R1
mul R1, TPROC, R1
set proc, R2
add R1, R2, R1
```

Ce fragment de code sera remplacé par l'expression : `INITdesc (R1, actif)`.

3.2.1 Version sans appel de procédures

La version présentée dans la figure 23.4 est purement pédagogique.

Rappelons le choix fait concernant les interruptions : tous les registres ordinaires sont sauvegardés dans la pile et sont bien évidemment restaurés lors du retour (`rti`).

Pour fixer les idées nous appelons Pprec le processus interrompu et Pnouv le processus relancé. La figure 23.5 décrit les piles en mémoire, les registres du processeur et les descripteurs des processus Pprec et Pnouv avant l'exécution de l'instruction rti.

Les registres ordinaires sont d'abord empilés par le processeur dans la pile de Pprec. Les registres ainsi sauvegardés reflètent l'état du processeur lors de l'interruption. Puis SP est sauvegardé dans le champ ssp du descripteur de Pprec. Puis un processus est élu, soit Pnouv, et la liste des processus prêts est mise à jour, ainsi que la variable actif. Après l'épilogue, SP repère le sommet de pile du processus Pnouv qui avait été interrompu précédemment. Au sommet de la pile de Pnouv se trouvent les registres du processeur avec leur valeur au moment où Pnouv avait été interrompu.

3.2.2 Version avec appel de procédures

La solution précédente ne donne pas totalement satisfaction : l'élection du processus et les parties prologue et épilogue sont, en général, réalisées par des appels de procédures, avec éventuellement des paramètres. Ces mécanismes font de nouveau usage d'une pile.

L'ordonnanceur n'étant pas un processus, se pose le problème de la pile à utiliser pour la gestion des appels de `actifversprêt` et `prêtversactif`. Deux approches peuvent être considérées : la première consiste à faire de l'ordonnanceur une entité dotée de sa propre pile ; l'autre revient à faire exécuter l'élection en utilisant la pile du processus actif suspendu pour appeler les deux fonctions d'ordonnancement.

Nous développons dans ce paragraphe le programme de traitement de commutation de processus en considérant que l'ordonnanceur dispose d'une pile privée lui servant à gérer ses informations propres. L'exercice E23.2 montre l'autre solution.

```
traitant :
prologue :INITdesc (R1, actif)  /* actif est Pprec */
          st SP, [R1+deltassp]  /* proc[Pprec].ssp = SP */

élection :/* code dans le paragraphe précédent */

epilogue :INITdesc (R1, actif)  /* actif est Pnouv */
          ld [R1+deltassp], SP  /* SP = proc[Pnouv].ssp */

pointobs :/* Point d'observation */
reprise : rti /* restaure le contenu des registres ordinaires */
```

FIG. 23.4 – Traitant de commutation de processus, version sans appel de procédures

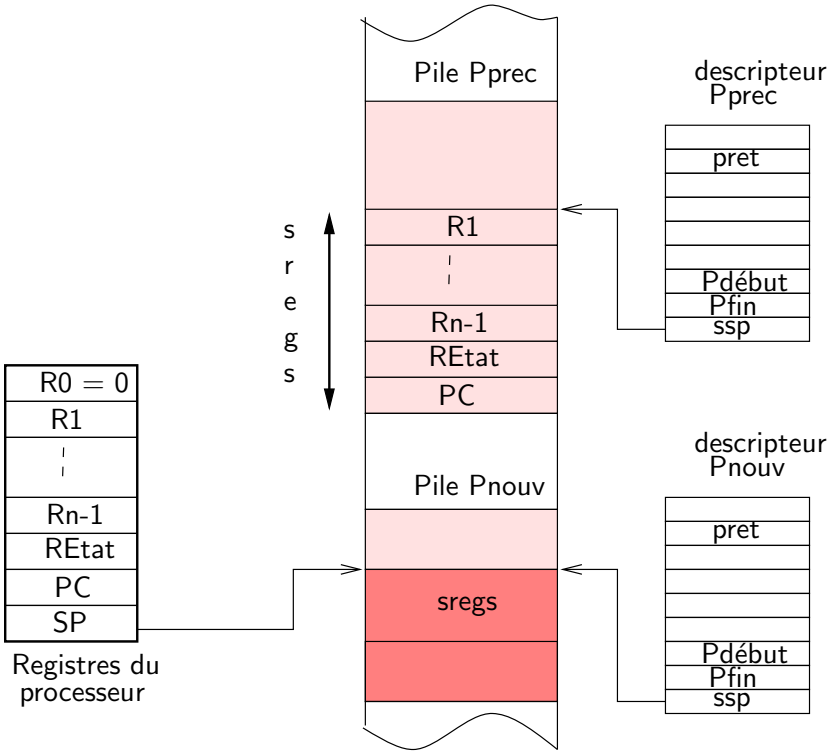


FIG. 23.5 – Contenu de la mémoire et des registres du processeur lors d'une commutation du processus Pprec au processus Pnouv, au point d'observation avant l'exécution de l'instruction `rti`

```
traitant :
/* les registres ordinaires ont été empilés par le processeur
dans la pile de Pprec */

prologue : jsr sauver
pointobsA : /* Point d'observation */
election : /* masquer IT : atomicité d'accès à file des prêts */
           seti
           /* Election du nouveau proc. dans pile spéciale*/
           set pileordo , SP
           jsr actifverspret
           jsr pretversactif     /* actif = Pnouv */
           cli /* démasquer IT : fin d'exclusion mutuelle */
epilogue : jsr restaurer
reprise :  rti /* restaure les registres autres que SP */

sauver :   INITdesc (R1, actif)  /* actif est Pprec */
           st SP, [R1+deltassp]  /* proc[Pprec].ssp = SP */
           rts

restaurer :
pointobsB : /* Point d'observation */
           INITdesc (R1, actif)  /* actif est Pnouv */
           ld [R1+deltassp], SP  /* SP = proc[Pnouv].ssp */
pointobsC : /* Point d'observation */
           /* Corrige l'adresse de retour pour exécution
           du rts de la procédure restaurer */
           set reprise, R1
           st R1, [SP] /* sommet de pile = l'adresse reprise */
pointobsD : /* Point d'observation */
           rts /* retour à reprise */
```

FIG. 23.6 – Traitant de commutation de processus utilisant une pile privée

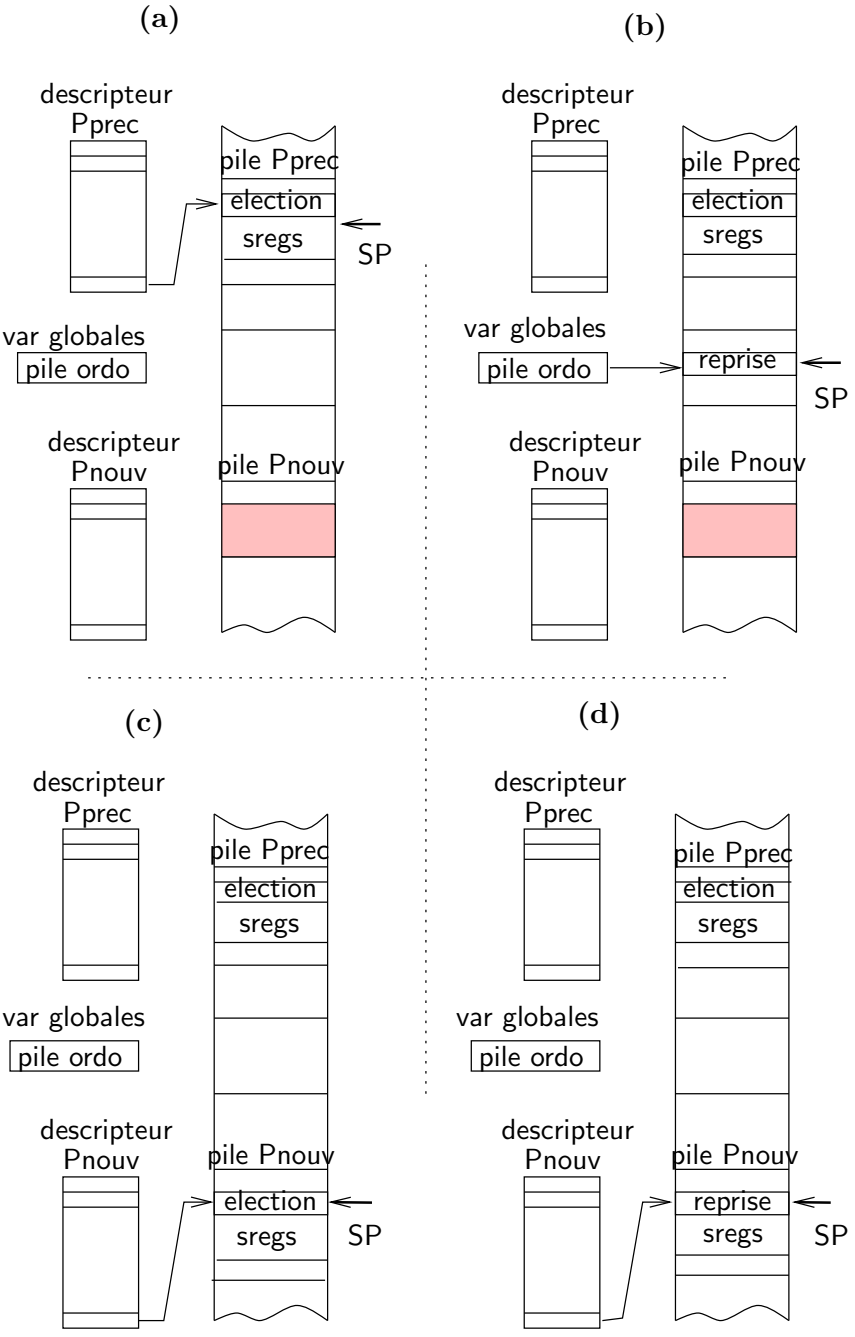


FIG. 23.7 – Contenu de la mémoire lors de l'exécution du traitant de commutation de processus muni d'une pile privée (Cf. Figure 23.6)

La figure 23.6 décrit le programme traitant. Le programme utilise la nouvelle variable globale pileordo qui est l'adresse du sommet de la pile privée de l'ordonnanceur. Les procédures `actifversprêt` et `prêtversactif` sont celles définies au paragraphe 3.1.

La figure 23.7 illustre le contenu des piles à différents instants de l'exécution du traitant. Nous allons maintenant détailler l'exécution pas à pas du traitant. Comme dans le paragraphe précédent nous appelons Pprec le processus interrompu et Pnouv le processus élu.

A l'entrée du traitant (prologue), le registre SP est le sommet de la pile de Pprec. Lors du départ en interruption le processeur a sauvegardé les registres (sauf SP) dans cette pile (zone sregs).

L'appel jsr sauver a pour effet la sauvegarde au sommet de la pile de Pprec de l'adresse de retour election. Le code de la procédure sauver stocke SP dans le champ ssp du descripteur de Pprec. La valeur de SP sauvegardée est donc décalée par rapport à la position effective de sregs dans la pile de Pprec. Lors de l'exécution de rts, le processeur prend son adresse de retour dans la pile de Pprec et SP repère le mot en dessous (Cf. Figure 23.7-a).

Le traitant exécute alors la partie élection. Nous verrons dans le chapitre 24 que le processeur peut être doté de plusieurs sources d'interruptions et que les traitants d'interruption sont eux-mêmes interruptibles. Le masquage des interruptions durant l'élection du nouveau processus actif garantit l'atomicité des opérations de mise à jour de la file des prêts. Ce problème d'atomicité a déjà été analysé sur l'exemple de la pendule du chapitre 22. Le traitant met alors en place la pile privée de l'ordonnanceur. SP est chargé avec l'adresse du sommet de cette pile qui est ensuite utilisée pour gérer les adresses de retour et le passage de paramètres lors des appels `actifversprêt` et `prêtversactif`.

Lors de l'appel jsr restaurer, SP repère toujours la pile de l'ordonnanceur. C'est donc dans celle-ci qu'est sauvegardée l'adresse de retour reprise (Cf. Figure 23.7-b).

Lors de l'exécution de restaurer, le champ ssp du descripteur de Pnouv qui avait été sauvegardé lors de la précédente interruption de Pnouv, repère le mot au-dessus de la zone sregs de Pnouv. En effet, Pnouv avait été précédemment interrompu de la même façon que ce que nous venons de décrire pour Pprec ; le sommet de la pile de Pnouv contient donc l'adresse election. SP est mis à jour avec ce champ ssp (Cf. Figure 23.7-c) mais ce n'est pas l'adresse qui convient pour l'exécution du rts de restaurer.

Il faut donc corriger avec la bonne adresse c'est-à-dire reprise avant d'exécuter rts (Cf. Figure 23.7-d). Après le retour de restaurer, SP repère le sommet de la pile de Pnouv qui contient la zone sregs. Le traitant exécute alors `rti` qui a pour effet la restauration de tous les autres registres.

L'utilisation d'une pile séparée pour l'ordonnanceur permet de gérer facilement n'importe quelle profondeur d'appel de procédures dans l'algorithme d'élection : il suffit de dimensionner cette pile en conséquence.

Par contre le surcoût de gestion du partage de temps pénalise les politiques

d'élection complexes et le recours à des algorithmes d'élection exigeant une pile de grande taille est peu probable.

L'exécution de l'élection dans le contexte du processus interrompu, autrement dit en utilisant la pile de ce dernier plutôt qu'une pile privée, est envisageable et simplifie la réalisation du traitant (Cf. Exercice E23.2).

4. Création et destruction de processus

Dans un système simple monotâche l'exécution d'un programme résulte d'un simple appel du chargeur/lanceur par l'interprète de commande (Cf. Chapitre 20). Elle se termine également de manière simple par l'exécution d'un retour.

Le passage à un système multitâches multiplexant plusieurs processus sur le processeur pose un certain nombre de questions : Comment et par qui est créé un processus ? Que devient le chargeur-lanceur ? Comment l'interprète de commande lance-t-il l'exécution d'un programme ? Comment se termine un processus ? Comment l'exécution de l'interprète de commande est-elle suspendue durant l'exécution du programme lancé ?

Le principe de fonctionnement est relativement simple : le lancement d'un programme est une création de processus exécutant le programme lancé. La création d'un processus est un mécanisme de filiation : un processus (appelé fils) est créé par un autre processus (appelé père). On parle parfois de naissance et de mort d'un processus. La terminaison normale d'un processus est une autodestruction.

Après création, père et fils s'exécutent comme deux processus indépendants. Les descripteurs de processus sont enrichis de nouveaux champs décrivant la structure arborescente de filiation des processus. Le père doit aussi avoir la possibilité de se suspendre jusqu'à la terminaison d'un fils. Ce comportement est entre autres celui d'un interprète de commandes textuel, qui crée un processus fils pour exécuter la commande, puis en attend la terminaison avant de lire une nouvelle ligne de commande.

Ainsi un processus n'est plus seulement actif ou prêt mais peut être suspendu. Il faut compléter la notion d'état d'un processus par l'état *en attente d'un événement* qui peut être la fin d'un autre processus par exemple. On parle au chapitre 24 d'autres extensions liées aux processus.

Les systèmes multi-utilisateurs associent à chaque processus un identificateur d'utilisateur propriétaire (uid) qui définit les ressources (telles que les fichiers) auxquelles il pourra accéder (via les procédures du système d'exploitation). Il existe généralement un utilisateur particulier nommé superutilisateur (à ne pas confondre avec le mode superviseur), auquel aucune limitation de droit d'accès n'est appliquée.

4.1 Création

La procédure de création retourne au processus père appelant le pid du fils créé, ou un code d'erreur (manque de mémoire ou d'entrée libre dans la table des processus ; fichier inexistant ou ne contenant pas un binaire exécutable valide pour la machine, ou droits du père insuffisants). Elle possède un surensemble des paramètres du chargeur-lanceur (excepté le début de mémoire libre) auquel elle se subtitue. Les paramètres supplémentaires sont relatifs à la gestion des droits d'accès du processus créé. Par défaut, le processus fils appartient au même utilisateur que le processus père : seuls les processus appartenant au superutilisateur sont autorisés à créer des fils appartenant à un autre utilisateur.

La procédure de création alloue au fils une entrée libre pid_fils de la table des processus ainsi qu'une pile système si elle n'est pas incluse dans pid_fils.

Comme dans le système simple, l'en-tête du fichier exécutable est lue et trois zones de mémoire texte, données et pile utilisateur sont allouées au processus créé. Le contenu des sections texte et données du fichier exécutable est chargé dans la mémoire allouée au processus ; l'algorithme de réimplantation est alors appliqué.

La procédure de création reproduit les paramètres passés par le père dans la pile utilisateur du fils et initialise le champ sommet de pile système (ssp) de pid_fils ainsi que la zone de sauvegarde pointée par ce dernier.

Le fils est ensuite inséré dans l'arborescence de filiation des processus et dans la liste des processus prêts.

Noter que l'initialisation de la zone de sauvegarde dans la pile système simule un processus préexistant interrompu par une interruption de commutation et remis dans la file des prêts.

4.2 Terminaison

La procédure d'autodestruction a pour paramètre un code de retour à transmettre au processus père, indiquant soit une exécution correcte, soit l'erreur ayant causé la terminaison prématurée du fils. Ce paramètre de l'appel de terminaison est recopié dans l'entrée de la table des processus correspondant au processus exécutant l'appel (donc le processus actif), où le père pourra le consulter ultérieurement.

Les zones texte, données et pile processus sont désallouées et rajoutées à l'espace mémoire disponible.

Dans la table des processus, la routine de destruction réinitialise le champ père de chacun de ses processus fils, qui deviennent orphelins, et purge les fils *fantômes* de la table des processus (les entrées correspondantes de la table des processus sont remises dans la liste des entrées libres).

Le processus en cours d'autodestruction devient un processus fantôme ou *zombie* (mort-vivant) qui n'évolue plus et qui n'utilise plus d'autre ressources

que son entrée dans la table des processus.

La libération de cette dernière sera effectuée ultérieurement par le processus père : elle contient le code de retour destiné au père. Pour cela, le processus qui s'autodétruit réveille son père si ce dernier est bloqué en attente de terminaison d'un fils. Si le processus qui s'autodétruit est orphelin, il se purge lui-même de la table des processus et disparaît.

La procédure de destruction se termine dans les deux cas par l'élection d'un nouveau processus actif.

4.3 Attente de terminaison d'un fils

La procédure d'attente retourne un code d'erreur si la liste des fils du processus est vide. Sinon, elle retire de celle-ci un fils fils_mort terminé (dans l'état fantôme ou zombie), recopie le code de retour de ce dernier dans la variable passée par le père en paramètre de l'appel, purge le fils de la table des processus et retourne fils_mort à l'appelant.

En l'absence de fils terminé, le processus père appelant exécute une procédure qui le bloque dans un état d'attente de termisaison d'un fils, et reprend tout depuis le début après avoir été réveillé par un fils qui se termine.

4.4 Le modèle UNIX

Nous avons vu précédemment que la procédure de création permet à un processus père de créer un fils avec un fichier exécutable distinct de delui du père, et sous une identification différente.

Le système d'exploitation UNIX offre trois primitives distinctes dont la combinaison permet d'arriver au même résultat.

- `fork` crée par clonage un processus fils à l'image du père. Les zones mémoire du père sont dupliquées à l'identique chez le fils. Le fils hérite de l'identification du père. Juste après la création, le fils ne se distingue du père que par un pid (entrée dans la table des processus) différent. Le retour de `fork` est exécuté par les deux processus et retourne au père le pid du fils et au fils une valeur nulle.
- `exec` est appelé par un processus pour changer le programme qu'il exécute. Le contenu du fichier passé en paramètre remplace celui des zones mémoires hérité du père.
- `setuid` permet à un processus de changer d'utilisateur propriétaire.

Le scénario classiquement employé est le suivant : le père crée un fils via `fork`, le fils change d'identification par `setuid`, puis appelle `exec` pour lancer son propre programme exécutable.

D'autre part, aucun processus fils ne libère lui-même son entrée dans la table des processus : la purge des zombies est réalisée par le père (réel ou adoptif). Les fils ne deviennent pas orphelins : à la mort du père, la paternité

est transférée au processus initial du système. Après la phase de démarrage, le processus initial entre dans une boucle dans laquelle il attend la terminaison d'un fils.

4.5 Scénario de démarrage : création d'un premier processus

Nous considérons ici les cas d'un ordinateur auquel sont reliés un certain nombre de terminaux. Un fichier sur disque décrit ces terminaux ainsi que l'ensemble des utilisateurs autorisés à se connecter.

A l'initialisation le processeur exécute un traitant de démarrage (amorce) résidant en mémoire morte et contenant un pilote de disque rudimentaire. Après que le système ait été chargé en mémoire, que les structures de données aient été initialisées, qu'un certain nombre de vérifications aient été faites, la procédure d'amorçage fabrique un pseudo processus pour appeler la procédure de création de processus et créer le processus initial appartenant à un utilisateur privilégié : le *superutilisateur*.

Le fichier exécutable passé en paramètre (init) contient le programme de démarrage du système. Celui-ci consulte un fichier donnant la liste des terminaux via lesquels les utilisateurs sont autorisés à se connecter. Pour chacun d'eux, il crée un processus gérant la procédure de connexion (login) avec pour paramètre le nom du terminal à gérer.

Ce dernier processus affiche la bannière de connexion. Il saisit ensuite le nom du compte et le mot de passe (en supprimant l'écho à l'écran) et les compare au contenu du fichier de déclaration des utilisateurs. Dans le cas où la connexion est acceptée, le processus de connexion crée un processus appartenant à l'utilisateur connecté pour exécuter le programme associé à cet utilisateur dans le fichier de déclaration. Cela peut être par exemple un interprète de commandes.

L'interprète de commande fonctionne comme celui du système simple à deux différences près : l'appel de la procédure charger_lancer est remplacé par un appel superviseur de création de processus suivi d'un appel d'attente de terminaison ; lorsque la commande est une déconnexion (logout), le processus interprète de commande exécute un appel de terminaison et meurt.

La terminaison du processus termine la session de travail, réveille le processus père qui réaffiche la bannière et attend une nouvelle connexion d'un utilisateur sur le terminal.

5. Exercices

E23.1 : Sauvegarde partielle ou totale des registres

Lors d'un départ en interruption, notre processeur ne sauve que le compteur programme et le registre d'état dans la pile. Dessiner l'état de la pile système au début du traitant. Nous donnons ci-dessous le traitant modifié et la procédure empiler. Décrire l'état de la pile après l'exécution du `rts` de empiler. Ecrire la procédure désempiler correspondante.

```
Traitant :                empiler :
    jsr empiler              sub SP, delta_reg7, SP
    jsr sauver               st  REG1, [SP + delta_reg1]
    ...                      st  REG2, [SP + delta_reg2]
    jsr restaurer            ...
    jsr desempiler           st  REG6, [SP + delta_reg6]
    rti                      ld  [SP + delta_reg7], REG1
                             st  REG1, [SP]
                             st  REG7, [SP + delta_reg7]
                             rts
```

E23.2 : Réalisation du traitant de commutation sans utilisation d'une pile privée

Nous donnons ci-dessous une autre réalisation du traitant de commutation. Il n'existe pas de pile privée pour l'ordonnanceur. Cela signifie que les informations à empiler ou à récuperer au sommet de la pile lors des appels et retours de procédures le sont dans les piles des processus interrompu et élu. Par rapport aux programmes du paragraphe 3.2.2 certaines lignes de code ont été supprimées dans la partie election et dans la procédure restaurer.

Etudier l'exécution de ce traitant en dessinant les différents états du contenu des piles et montrer que le programme traitant est bien correct. En particulier, remarquer que l'adresse de retour pointée par le champ ssp du processus suspendu sera modifiée par chacun des trois appels de procédure suivants, qui utilisent sa pile. A la fin du traitant, l'adresse effectivement présente dans la pile du processus suspendu est celle empilée par l'appel de restaurer (reprise).

```
traitant :                          sauver :
prologue : jsr sauver                  INITdesc (R1, actif)
                                       st  SP, [R1+deltassp]
election : seti                        rts
           jsr actifversprêt
           jsr prêtversactif        restaurer:
           cli                         INITdesc (R1, actif)
epilogue : jsr restaurer               ld  [R1+deltassp], SP
           rti                         rts
```

Chapitre 24

Généralisation du mécanisme d'interruption et applications

Nous avons introduit au chapitre 23 la notion de *processus*, brique de base de la construction des systèmes complexes. Ce mécanisme plus avancé permet de donner l'impression de *vraiment* traiter deux tâches simultanément.

Il nous reste maintenant à étudier comment généraliser le mécanisme des interruptions à plusieurs sources d'interruption et gérer les problèmes de protection des informations des différentes activités se déroulant *simultanément*.

Par ailleurs, les différentes approches d'entrées/sorties étudiées au chapitre 16 utilisent principalement une phase d'attente active où le processeur ne fait rien d'autre que de scruter un coupleur en attendant que le périphérique soit en état de communiquer. L'enrichissement du système d'interruption et l'utilisation du mécanisme de processus vont permettre de partager véritablement le temps du processeur entre des tâches de calcul et des tâches d'entrées/sorties. Nous gardons toutefois la synchronisation élémentaire : le coupleur maintient la requête tant qu'il ne reçoit pas de la part du processeur un acquittement signifiant que la requête est prise en compte.

Dans ce chapitre nous montrons comment se généralise le mécanisme d'interruption (paragraphe 1.) et comment s'en servir pour apporter une réponse aux différents besoins identifiés plus haut. Nous détaillons en particulier la notion de mode superviseur/mode utilisateur *(paragraphe 2.) du point de vue du programmeur et du point de vue du processeur. Pour utiliser au mieux cette protection, nous étendons le processeur simple présenté au chapitre 22 avec quelques registres et de la circuiterie de calcul d'adresses. Parmi les différentes utilisations du mécanisme d'interruption, nous étudions particulièrement les entrées/sorties (paragraphe 3.) où le partage de temps est le plus justifié.*

1. Classification des différentes sources d'interruption

Dans le chapitre 22 nous avons considéré deux sources d'interruptions possibles, un fil IRQ et une instruction `swi`. Nous avons mentionné l'utilisation du `swi` principalement pour pouvoir mettre au point un programme traitant en provoquant par logiciel un basculement de départ.

Pour mettre en place pleinement différentes possibilités de partage de temps, de nombreuses autres sources d'interruptions existent dans les machines. Les traitements correspondants permettent de simplifier la tâche du programmeur d'application en laissant dans le système d'exploitation de nombreux aspects dont la programmation est délicate. En effet le système d'exploitation comporte déjà de nombreux traitants.

Nous présentons les différentes types d'interruptions selon trois critères. On s'intéresse d'abord à la *source* des interruptions : celle-ci peut provenir de l'intérieur du processeur ou de l'extérieur. Le paragraphe 1.1 présente les différentes interruptions de façon détaillée selon leur source.

Par ailleurs les interruptions ont divers degré de *synchronicité* avec l'exécution des instructions : les interruptions peuvent survenir à des instants prévisibles à une certaine étape de l'exécution des instructions ou à n'importe quel moment. Dans le second cas elles ne sont considérées qu'ENTRE les instructions comme dans le chapitre 22.

Le troisième critère est la *fonction* d'une interruption. Les différentes fonctions des interruptions dans la machine sont : la communication entre l'ordinateur et le monde extérieur, la détection et l'éventuelle correction d'erreurs diverses et les appels, voulus par le programmeur, de fonctions particulières déjà programmées dans le logiciel de base livré avec la machine. Dans la pratique ces trois critères ne sont pas totalement indépendants.

Dans le texte nous ferons souvent référence à la notion de *superviseur* qui est étudiée en détail au paragraphe 2.

1.1 Sources externes et sources internes

Il importe de distinguer les sources d'interruption selon leur origine physique et logique. Certaines proviennent de l'intérieur du processeur, d'autres de l'extérieur.

1.1.1 Sources internes

Parmi les *sources internes*, citons tout ce qui est lié à l'exécution d'une instruction.

Instruction SoftWare Interrupt (`swi`) On a déjà mentionné l'existence de cette instruction. Elle est utilisée prioritairement pour appeler une fonction

déjà existante du système d'exploitation et il existe des machines où le code du `swi` comporte une constante précisant lequel des services du système est appelé. Voir figure 24.1 la prise en compte du code `swi`.

Instructions invalides Si le registre instruction comporte N bits, il y a 2^N codes possibles. Il se peut que certains codes ne soient ceux d'aucune instruction. On parle de *code invalide*. Même sur le processeur simple étudié au chapitre 14 certains codes étaient invalides. Il est classique de générer une interruption interne dans ce cas. Voir figure 24.1 la prise en compte du code invalide.

Certains codes invalides sont documentés comme de vraies instructions. Cette technique peut être utilisée pour *étendre* le jeu d'instructions sans pour autant donner les ressources matérielles correspondantes. On peut par exemple avoir ainsi des instructions de calcul en virgule flottante alors que le processeur ne dispose pas d'une unité matérielle de calcul sur cette représentation. Il y a alors un traitant spécifique qui réalise par programme l'équivalent de ce que devrait faire l'instruction.

> **Remarque :** Dans la préhistoire de l'informatique certains processeurs se mettaient en cas de code invalide à exécuter une séquence non documentée. Il fallait, pour y comprendre quelque chose, prendre les équations de la réalisation de la partie contrôle et retrouver les ϕ des tableaux de Karnaugh. Chose que le programmeur *normal* refuse de faire. Faire, si nécessaire, l'exercice E14.5 pour voir la difficulté.

Donnée invalide Il existe des machines où une interruption est déclenchée en cas d'apparition d'une instruction de division avec un opérande diviseur nul. Une autre situation d'interruption se produit si il y a accès à un mot alors que l'adresse n'est pas multiple de 4 dans une machine où une telle convention existe. Sur certaines machines des instructions particulières peuvent faire déborder certaines ressources. C'est le cas sur le SPARC où les instructions `save` ou `restore` peuvent provoquer des interruptions pour débordements de fenêtre.

Instructions spéciales Beaucoup de machines ont des classes d'instructions *spéciales* (voir paragraphe 2.3). Le processeur à deux (ou plus) modes de fonctionnement. Le mode de fonctionnement est matériellement un bit du mot d'état. Un exemple sera fourni sous le nom de mode utilisateur/mode superviseur dans ce chapitre. Dans le mode 1, toutes les instructions sont licites, dans le mode 2 certaines ne sont pas licites. Au moment du décodage du registre instruction, la prise en compte du bit de mode dans le mot d'état et la rencontre d'une instruction illicite pour le mode courant engendrent une interruption interne. On parle alors de *violation de priviliège*. Les instructions de changement de mode ne peuvent évidemment pas être toujours licites.

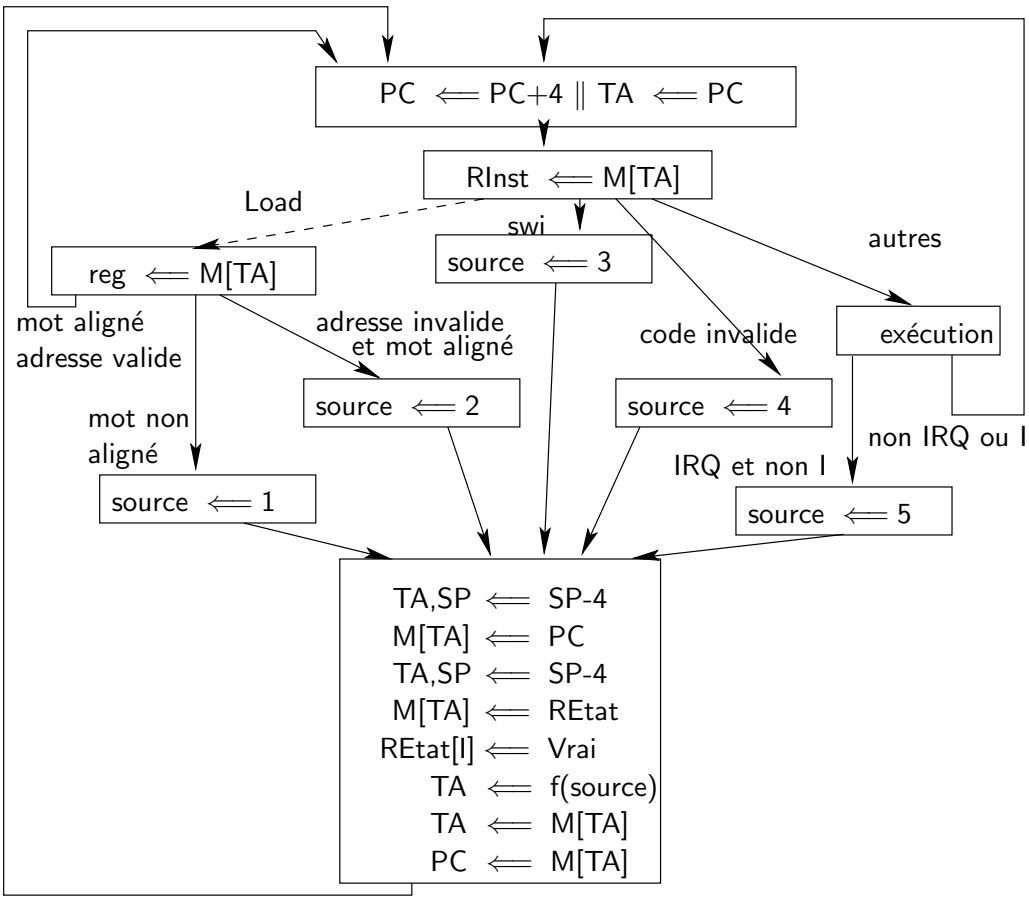


FIG. 24.1 – Graphe de contrôle (très partiel) du processeur acceptant plusieurs sources d'interruptions

1.1.2 Sources externes

Les *sources externes* sont diverses. On a déjà mentionné l'existence d'un fil IRQ. Il peut y avoir plusieurs fils de requête au lieu d'un seul. Les fils venant de l'extérieur du processeur peuvent véhiculer toutes sortes d'information sur l'état du monde ou le déroulement des instructions.

Etat du monde Certaines sources informent l'ordinateur d'un *état du monde* : depuis le petit montage électronique qui détecte une faiblesse de l'alimentation de l'ordinateur et fait commuter sur une tâche de recopie de la mémoire centrale sur disque dur, jusqu'au clic souris en passant par l'impulsion d'horloge ou la frappe d'un caractère au clavier. Nous reparlons de la gestion d'entrées/sorties en utilisant le système d'interruption au paragraphe 3. On trouve aussi de plus en plus d'ordinateurs qui détectent : *il ne s'est rien passé depuis N minutes* et passent automatiquement en mode basse consommation.

Exécution problématique D'autres sources résultent de problèmes lors de l'exécution d'instructions. On a donné dans le chapitre 15 un décodage d'adresse ; si une adresse émise sur le bus adresse ne correspond à aucune zone mémoire installée, et si le décodage est complet, un signal *adresse invalide* peut être fabriqué par le décodeur. Ce signal peut être une requête d'interruption. Remarquons que le concepteur du processeur peut prévoir dans quels états de la partie contrôle une telle requête peut se produire : uniquement dans les états où il y a accès mémoire. Il installe alors un test après chacun de ces états. Sur la figure 24.1 on a figuré ces tests après un seul état, celui d'exécution de l'instruction `load`. C'est évidemment un souci de simplification ; le même test aurait lieu après chaque état où figure un accès mémoire.

Par ailleurs, le processeur qui accède en lecture à la mémoire peut émettre un signal précisant si l'accès est à une instruction ou à une donnée. Si un système de protection est installé entre les zones texte et données des processus, un décodage d'accès invalide peut être installé.

1.2 Synchronicité des interruptions et du déroulement des instructions

Certains signaux d'interruptions passent à l'état actif à un instant précis et prévisible par rapport au déroulement de l'instruction. C'est le cas de toutes les interruptions internes et des interruptions externes liées au décodage d'adresse. On parlera alors d'interruptions *synchrones*.

Par contre les requêtes venant des périphériques divers, à travers des coupleurs, peuvent survenir n'importe quand. On parlera d'interruptions *asynchrones*. Les signaux sont alors échantillonnés sur l'horloge du processeur pour être conservés dans une bascule D, et ne sont pris en compte qu'à des instants pré-établis. Sur la figure 24.1 figure le test de IRQ après l'exécution des instructions marquées *autres*. Il aurait lieu à la fin de chaque instruction.

1.3 Fonctions des interruptions

Le mécanisme d'interruption est utilisé pour la mise en oeuvre de trois fonctionnalités importantes dans les systèmes complexes.

Une première fonctionnalité correspond au *partage de temps*, à l'optimisation des opérations d'entrées/sorties et la prise en compte des alarmes venant de la périphérie : ce sont les interruptions proprement dites.

La deuxième fonctionnalité concerne la *gestion des erreurs* et anomalies déclenchées par l'exécution des programmes : on parle alors plutôt de *déroutement* que d'interruption.

La troisième est liée à l'*invocation des routines* du système d'exploitation, par exemple celles des pilotes de périphériques avec les privilèges adéquats pour accéder à toutes les ressources de l'ordinateur : on parle d'*appel au superviseur*.

1.4 Non indépendance des critères

Une interruption interne est forcément synchrone : appels au superviseur, code ou données invalides, etc.

Une interruption externe est synchrone lorsqu'il s'agit d'une détection d'un mauvais déroulement d'une instruction : accès mémoire non autorisé, etc.

Une interruption externe est asynchrone lorsqu'elle permet de gérer les communications avec le monde extérieur.

1.5 Identification de la requête et sélection du traitant

Puisqu'il y a plusieurs sources d'interruptions il est obligatoire de pouvoir identifier la source de l'interruption et de pouvoir associer un TRAITANT particulier pour chaque source.

L'organisation type est d'avoir, après le basculement de départ, une valeur de PC caractéristique de la source de la requête. Il convient donc de réaliser *d'une certaine façon* la fonction qui donne une adresse de début pour chaque cause.

Le problème se pose différemment selon que les différentes requêtes sont simultanées ou non.

1.5.1 Forçage du Compteur Programme

Il y a principalement trois façons de forcer le compteur programme au moment du basculement de départ.

La solution la plus répandue consiste à *forcer une valeur* dans un registre intermédiaire source. La valeur est caractéristique de la cause d'interruption (voir figure 24.1). Cette valeur est ensuite utilisée pour fabriquer l'adresse de début du traitant.

Certains processeurs de la famille INTEL établissent un *dialogue entre le processeur et le coupleur* après une requête d'interruption asynchrone. Cela permet de localiser la source qui fournit alors un numéro permettant de fabriquer l'adresse de début du traitant.

Une *solution par programme* est possible aussi : le processeur scrute les différents mot d'état des coupleurs par le programme TRAITANT et se branche, conditionnellement, à une table d'adresses pré-établie.

1.5.2 Choix parmi des sources simultanées

La prise en compte de multiples sources d'interruptions demande de régler le problème du choix parmi les candidats possibles. Par exemple lorsqu'un accès à un mot a lieu à une adresse non multiple de 4 dans une zone de mémoire où aucun boitier n'est installé, quel traitement sera fait ? Celui pour accès invalide ou celui de mémoire non installée ?

Nous distinguerons quatre façons de faire ce type de choix. Elles correspondent à des métiers de l'informatique différents.

Dans la première solution le choix est fait à la *conception du processeur*. Il est établi par matériel à l'intérieur du processeur. Le choix n'est évidemment pas modifiable par l'utilisateur final de l'ordinateur. Le métier concerné est celui de concepteur de processeur.

Dans la deuxième solution le choix est fait par *matériel à l'extérieur* du processeur. L'utilisateur de l'ordinateur ne peut en général pas faire de modification mais l'architecte de machine a la charge de réaliser cette fonction. Le métier concerné est celui de concepteur d'ordinateurs à base de processeurs existants.

Dans la troisième solution le choix de prise en compte parmi des requêtes est fait *par logiciel*. Il s'agit d'une couche logicielle de bas niveau, enfouie dans le système d'exploitation. L'utilisateur final n'y peut rien. Le métier concerné est celui de concepteur de logiciel de base d'ordinateur.

Enfin, sur la plupart des machines, par appel à des routines système, l'utilisateur peut écrire des programmes gérant un certain nombre d'événements. Le lecteur du présent livre risque bien d'être un jour dans cette situation !

Le premier choix est entièrement matériel. Supposons 3 signaux de requêtes simultanés possibles à un instant donné irq1, irq2, irq3. Il est facile d'introduire dans le graphe de contrôle du processeur 4 états E1, E2, E3 et E4 successeurs de l'état de test avec le séquencement suivant :

```
état_suivant = si irq1 alors E1 sinon
    si irq2 alors E2 sinon
        si irq3 alors E3 sinon E4
```

Par matériel, cela revient à utiliser les booléens irq1, $\overline{\text{irq1}}$.irq2, $\overline{\text{irq1}}.\overline{\text{irq2}}$.irq3, et $\overline{\text{irq1}}.\overline{\text{irq2}}.\overline{\text{irq3}}$ comme conditions de validation des arcs dans le graphe de l'automate de contrôle. C'est ce qui est fait sur la figure 24.1 pour les sources *mot non aligné* et *adresse invalide* après l'état de lecture mémoire et chargement d'un registre dans l'exécution de l'instruction de chargement.

Le deuxième cas s'applique pour les machines où les coupleurs fournissent chacun un fil de requête, par exemple jusqu'à 16 fils, mais où un circuit externe au processeur, nommé *encodeur de priorité d'interruptions*, ne fournit au processeur que le numéro, sur 4 bits, du fil actif de plus fort numéro. Le câblage du clic souris sur l'entrée 3, de l'horloge sur l'entrée 14 et du contrôleur de disque sur l'entrée 7 fixe alors les priorités respectives de ces différentes sources possibles. Si les 3 sources émettent des requêtes simultanées, c'est l'horloge (de niveau 14) qui sera servie.

Les troisième et quatrième cas ont lieu quand plusieurs sources sont reliées par un seul fil physique au processeur selon un montage à base d'une porte NOR en NMOS où la résistance de rappel à l'alimentation est, par exemple, interne au processeur (Cf. Figure 24.2). Le processeur entame alors une scrutation, par logiciel, *dans un ordre donné*, des registres d'états des différents coupleurs pour identifier celui qui est l'auteur de la requête. Dans le cas de requêtes

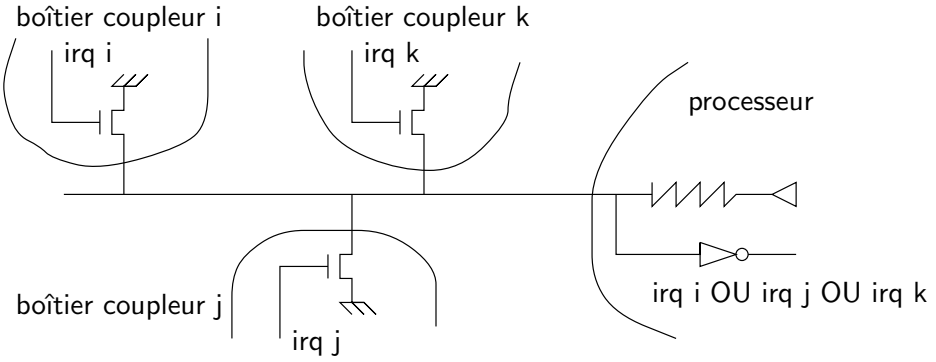


FIG. 24.2 – Liaison des requêtes d'interruption sur un seul fil

simultanées, l'ordre de scrutation fixe la priorité des sources. Le premier scruté ayant émis une requête est servi. Il faut, naturellement, que les demandeurs non servis maintiennent leur requête.

Naturellement, il est possible de cumuler les différents choix, en respectant une certaine cohérence.

1.5.3 Choix parmi des sources hiérarchisées

On a mentionné dans le chapitre 22 le traitement réservé à une requête d'interruption survenant *pendant* l'exécution d'un TRAITANT. Le bit d'inhibition I permet d'autoriser ou d'interdire la prise en compte de telles requêtes à ce moment. Sur certaines machines le programmeur a accès à ce bit.

Cette technique peut être généralisée à plusieurs sources. Si un processeur a 3 sources d'interruptions, irq1, irq2, irq3, on peut convenir que irq3 peut toujours interrompre le TRAITANT de irq2 ou irq1. De même que irq2 peut toujours interrompre le TRAITANT de irq1. Il reste alors au programmeur, par manipulation de bits I_1, I_2, I_3 du mot d'état à décider si il autorise ou non à ce que irq_k interrompe le TRAITANT de irq_k.

Il va de soi que si une requête peut interrompre un traitant il ne faut pas que le traitant de commutation de contexte puisse être interrompu n'importe quand. Il convient donc d'accorder une priorité élevée aux interruptions horloge et aux appels superviseur. Il reste, néanmoins, à protéger les traitants les uns des autres.

1.6 Problème de la reprise d'une instruction

Les processeurs SPARC ont une protection contre une utilisation d'une nouvelle fenêtre qui viendrait à écraser une fenêtre déjà utilisée. Si un tel accès est tenté par une instruction, une interruption interne est émise. Le traitant d'interruption doit alors sauver en mémoire, dans une zone gérée en pile, le contenu des registres menacés. Au retour du traitant d'interruption, il convient

alors de *reprendre* l'instruction qui a causé l'interruption, et non pas, comme dans le cas général de passer à la suivante.

Les machines VAX disposent d'une instruction d'évaluation de polynôme. La valeur de la variable est rangée à un endroit et un tableau en mémoire contient les N coefficients. N peut être grand. L'exécution de cette instruction peut donc durer *un certain temps*. Une interruption très prioritaire peut alors survenir. Et les concepteurs de la machine peuvent décider d'interrompre l'instruction de calcul du polynôme sans perdre les calculs déjà faits. Se pose alors le problème de savoir si il est possible de reprendre l'instruction interrompue en cours d'exécution. Cela nécessite de sauvegarder, au moment du basculement de départ, non seulement l'adresse de retour égale dans ce cas à l'adresse de l'instruction en cours d'exécution, mais aussi le numéro de l'état où a eu lieu l'interruption ou le compteur de boucle indiquant où en est l'exécution.

2. Protection entre processus, notion de superviseur

Dans un système multi-utilisateurs bien conçu, un processus ne peut pas accéder librement aux ressources physiques de l'ordinateur (mémoire centrale, espace disque, temps du processeur) et les monopoliser.

On ne peut imaginer qu'un programmeur sur une machine multi-utilisateur puisse écrire un programme en langage d'assemblage qui aille écrire dans la structure de données de l'ordonnanceur de processus et s'alloue la totalité du temps.

La confidentialité des informations est également importante. Imaginons qu'un enseignant soit en train d'éditer un sujet d'examen sur le serveur de son établissement : il est plus que souhaitable qu'un processus d'un étudiant, travaillant sur la même machine, ne puisse pas lire ni modifier la mémoire du processus de l'enseignant ou le contenu du fichier sur le disque ! La nécessité de mécanismes de protection est encore plus évidente dans les domaines bancaires ou militaires.

Différents aspects de sécurité sont couverts par la notion de mode superviseur, qui s'oppose à mode utilisateur.

2.1 Notion de mode superviseur et de mode utilisateur

Le processeur est doté de deux modes de fonctionnement, *utilisateur* et *superviseur*. Le principe consiste à exécuter le code de l'utilisateur propriétaire du processus dans un mode restreint dit *utilisateur*, ou *restreint*, dans lequel le processeur ne peut accéder à la mémoire centrale en dehors des zones qui lui sont allouées (texte, données, pile), ni aux périphériques. Les routines du noyau du système d'exploitation sont au contraire exécutées dans un mode

privilégié dit *superviseur* ou *noyau*, leur permettant d'accéder librement à toute la mémoire et à tous les périphériques de la machine.

Le mode d'exécution est défini par un nouveau booléen S (pour superviseur) du registre d'état du processeur. La valeur de ce bit est affichée vers l'extérieur lors des accès mémoire. Le circuit de décodage d'adresses vérifie l'adresse et le sens de chaque accès mémoire effectué par le processeur et s'assure qu'ils correspondent bien aux zones mémoires attribuées au processus actif et à son mode d'exécution.

De plus certaines instructions étant accessibles seulement en mode superviseur les programmes en mode utilisateur sont cantonnés. Par exemple les instructions de modification, voire de consultation, de certains registres du processeur ne sont exécutables qu'en mode superviseur. La rencontre d'une instruction réservée sans que le bit S soit dans l'état convenable provoque une interruption interne, analogue à la rencontre d'un code invalide.

La seule façon pour un programme s'exécutant en mode utilisateur pour accéder au système d'exploitation est connue sous le nom d'*appel au superviseur*. Elle permet au programmeur d'utiliser les procédures du système d'exploitation.

2.2 Cahier des charges des appels au superviseur

Le cahier des charges du mécanisme d'appel d'une procédure du noyau du système est le suivant :

- le passage en mode superviseur doit être couplé et restreint aux seuls appels de procédures bien identifiées du système,
- le mode en vigueur (donc tout ou partie du registre d'état, au minimum le bit S) doit être mémorisé lors de l'appel au superviseur et rétabli lors du retour : certaines procédures sont en effet susceptibles d'être appelées indifféremment par d'autres procédures du système en mode noyau ou par du code utilisateur en mode restreint,
- il faut assurer la liaison dynamique entre l'appelante et la routine système, dont l'adresse peut varier d'un ordinateur à l'autre (par exemple parce qu'ils disposent de pilotes de périphériques différents), qui reposait dans notre système simple sur une table de branchement,
- il faut contraindre le branchement aux seules entrées définies dans la table. Dans le cas contraire, un utilisateur malveillant pourrait court-circuiter les vérifications (de droit d'accès) effectuées dans le prologue de la routine système en se branchant directement sur le corps de celle-ci, voire appeler sa propre routine créée de toutes pièces à la place de la procédure système.

En résumé, il s'agit d'appeler un traitant dont l'adresse est définie implicitement par un numéro de service dans une table de vectorisation, de sauvegarder le (bit S du) registre d'état en plus de l'adresse de retour et de positionner S à 1.

Nous pouvons pour cela réutiliser le mécanisme d'interruptions vectorisées, avec trois modifications mineures : 1) l'appel est déclenché explicitement par

une instruction (`swi`) ; 2) le numéro de vecteur est un paramètre de l'instruction d'interruption logicielle `swi`. Dans ce contexte d'utilisation l'instruction d'appel au superviseur est souvent nommée `trap` ; 3) le bit `S` est mis à 1 lors du basculement de départ ; c'est pourquoi le matériel doit effectuer une sauvegarde de tout ou partie du registre d'état, au minimum le bit `S`.

2.3 Instructions privilégiées

Naturellement, l'exécution de toute instruction autre que l'appel superviseur et susceptible de modifier le bit S doit être interdite en mode utilisateur. Les instructions correspondantes sont dites *privilégiées* et déclenchent un déroutement de violation de privilège si on tente de les exécuter en mode restreint.

Toutes les instructions affectant un ou plusieurs booléens du registre d'état autres que ceux formant le code condition de branchement (Z, N, C, V) sont privilégiées, à commencer par l'instruction de retour d'interruption.

Les registres d'adresses dont nous parlons au paragraphe 2.7.1 utilisés pour installer le mécanisme de protection sont en accès réservé. Leur écriture, parfois leur lecture, depuis le mode utilisateur constitue une violation.

Cette restriction inclut le contrôle de la prise en compte des interruptions externes : en masquant les interruptions, un processus pourrait désactiver le mécanisme de partage de temps et monopoliser le processeur.

2.4 Séparation des piles, deux pointeurs de pile

Il subsiste un trou de sécurité plus subtil : la sauvegarde du registre d'état et de l'adresse de retour s'effectuent en mode superviseur et à l'adresse donnée par le pointeur de pile. Or l'affectation du pointeur de pile, utilisée dans la gestion des appels de procédures, ne peut être privilégiée. Le système ne dispose donc d'aucune garantie de validité de l'adresse contenue dans le pointeur de pile. Un départ en interruption pourrait donc se faire avec une référence de pile incorrecte. Or juste après le processeur est en mode superviseur.

C'est pourquoi on peut doter le processeur de deux pointeurs de pile, un pour chaque mode d'exécution, le pointeur de pile à utiliser étant sélectionné par la valeur du bit S.

On alloue à chaque processus une pile système destinée au seul code exécuté en mode noyau. Le tableau contenant la pile système pourrait par exemple être un champ supplémentaire de la structure de données représentant les processus.

Le pointeur de pile système est automatiquement activé avant la sauvegarde lors du départ en interruption, déroutement sur erreur, ou appel superviseur. En mode superviseur, une instruction privilégiée permet de manipuler le pointeur de pile utilisateur. En mode utilisateur, l'accès au pointeur de pile système est interdit. SP est le nom du pointeur de pile actif dans le mode d'exécution courant, les commandes d'accès aux registres tiennent compte

du mode d'exécution. L'appel de procédure, qui fait usage de SP, est ainsi indépendant du mode dans lequel s'exécutent les procédures.

2.5 Réalisation des appels au superviseur

Les programmes utilisateurs sont liés (dynamiquement) avec les bibliothèques d'interface livrées avec le système, qui encapsulent et masquent le mécanisme d'appel superviseur. Pour accéder à un service système, le processus exécute un appel de procédure normal, en mode utilisateur, en respectant la convention d'appel de procédure standard d'empilement de l'adresse de retour et des paramètres dans la pile utilisateur. Puis la procédure appelée, la routine de bibliothèque d'interface, exécute à son tour une instruction d'appel superviseur avec le numéro de service à invoquer.

Le traitant d'appel superviseur sauvegarde le contenu des registres du processeur dans la pile système et effectue un saut au code correspondant à la primitive demandée, via une table de branchement indexée par le numéro de service passé par la fonction de bibliothèque.

En utilisant l'adresse contenue dans le pointeur de pile utilisateur, le traitant peut retrouver les paramètres passés par l'appelante dans la pile utilisateur et les recopier dans la pile système pour appeler la routine chargée du traitement.

La procédure noyau invoquée vérifie que le processus actif possède les droits nécessaires pour l'opération considérée, et que les paramètres appartiennent à la plage de valeurs légales. Dans le cas d'une lecture de fichier, la routine vérifiera entre autres que le propriétaire du processus a le droit de lecture sur le fichier et que le tampon de réception (paramètres adresse et taille du tampon) est bien inclus dans la zone de données affectée au processus actif.

Réciproquement, le code d'erreur retourné par la fonction noyau est passé en sens inverse à la fonction de bibliothèque qui a exécuté l'appel superviseur.

La convention d'appel stipule généralement que la valeur retournée par la fonction est stockée dans un registre du processeur. Dans ce cas, la fonction système peut tout simplement déposer un code de retour dans la zone de sauvegarde des registres dans la pile système. Cette valeur sera désempilée par l'épilogue du traitant d'appel superviseur, qui restaure la valeur des registres, et la fonction de bibliothèque la récupérera dans le registre correspondant.

2.6 Protection généralisée

Il existe des machines pourvues de plusieurs (≥ 2) anneaux ou niveaux de protection (à ne pas confondre avec le niveau de prise en compte des interrutions externes), numérotés de 0 à $p-1$. L'anneau intérieur de numéro 0 correspond au mode noyau dépourvu de toute restriction. Les privilèges accordés aux processus diminuent avec les numéros d'anneaux et l'anneau de

numéro $p-1$ correspond au mode utilisateur dans lequel toutes les protections sont activées.

Chaque réduction de niveau de protection met en jeu un mécanisme analogue à l'appel superviseur. Chaque anneau est doté de sa propre pile et la recopie des paramètres entre niveaux d'appels différents est gérée par matériel pour des raisons d'efficacité.

Dans la description du système d'exploitation MULTICS [Org72] (antérieur à UNIX) on trouve une discussion de la raison d'être et de l'utilisation des anneaux de protection. Le système d'exploitation OS/2 utilise également ce type de protection.

En dépit de leur intérêt, les anneaux de protection sont quelque peu tombés en désuétude. Ils sont supportés par les processeurs de la famille 80x86 d'INTEL à partir du 80286.

2.7 Protection par définition de zones mémoire

Nous décrivons ici un mécanisme de protection par *découpage de zones mémoire* et nous en donnons une extension simple, connue sous le nom de translation d'adresses.

2.7.1 Motivation, comportement et réalisation matérielle

Nous avons vu que chaque processus peut accéder à une zone texte, une zone données et une zone pile en mémoire. Pour garantir qu'un processus n'accède pas aux zones de ses voisins, nous dotons le processeur de 6 registres supplémentaires. Ces registres seraient inclus dans une partie de calcul d'adresse CA qui ne figure pas sur la figure 22.1. On la situerait dans la zone du registre TA. Ces 6 registres doivent contenir en fonctionnement normal les adresses de début et de fin des 3 zones de texte, données et pile du processus en cours d'exécution. Le bloc CA contient de plus 6 comparateurs entre l'adresse courante et ces 6 constantes (Cf. Figure 24.3).

Ces 6 registres sont mis à jour lors de l'étape d'épilogue du traitant d'interruption qui assure la commutation de processus (Cf. Figure 23.6). Les valeurs à y installer figurent dans le descripteur du processus.

La protection peut se faire à différents niveaux que nous allons aborder successivement.

Un premier niveau de vérification consiste à s'assurer, à la volée, lors de chaque accès mémoire à une addresse ad que :

Début_texte $\leq$ ad $\leq$ Fin_texte

Début_données $\leq$ ad $\leq$ Fin_données

Début_pile $\leq$ ad $\leq$ Fin_pile

Si aucune de ces conditions n'est satisfaite (t=d=p=0), l'accès mémoire est annulé (signal AccèsMem mis à 0) et une interruption est émise. C'est évidemment une interruption interne parfaitement synchrone. Son traitement

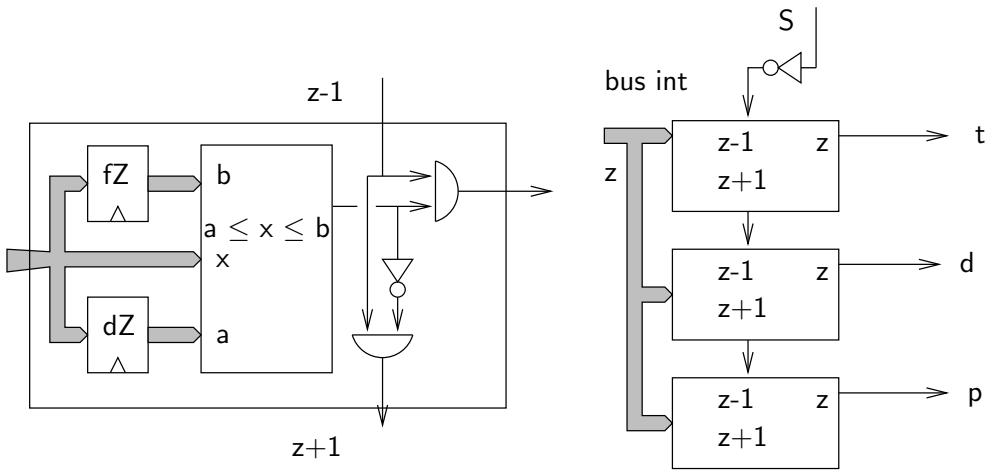


FIG. 24.3 – Comparateur simple entre une adresse et des débuts (dZ) et fin (fZ) de zones. Les trois comparateurs sont cascadés. L'entrée S est le bit `superviseur`. En effet la vérification est faite en mode Utilisateur seulement.

donne lieu à un traitant d'erreur d'adressage ressemblant à celui pour adresse invalide ou mot non aligné.

Un deuxième niveau de vérification plus poussée peut être installé tenant compte : du type d'accès (lecture ou écriture) ; du type d'objet accédé (donnée ou instruction), en fait il s'agit plus exactement de l'instant d'accès dans le graphe de contrôle (phase acquisition du code opération ou phase autre) ; du mode d'exécution (superviseur ou utilisateur). Le circuit combinatoire de vérification est un peu plus complexe dans ce cas.

2.7.2 Translation d'adresses

Un autre mécanisme utile peut venir compléter cette protection : il est possible que deux processus soient en réalité l'exécution du même programme sur des données différentes. Que l'on pense, par exemple, à un compilateur travaillant en multitâche pour différents utilisateurs programmeurs. La zone `text` est alors partagée et les zones `data` et `bss` sont dupliquées. La translation des adresses dans la zone `text` ne peut plus être réalisée lors du chargement puisque ces adresses, pour un même code exécuté, repèrent des endroits différents.

Une solution classique est de n'avoir que des programmes qui, virtuellement, ne travaillent qu'avec des adresses mémoires qui ne sont pas absolues, même si l'on parle de mode d'adressage absolu. Les adresses dans le code machine sont, en fait, des déplacements par rapport à une adresse de début de zone. L'adresse effective en mémoire d'une donnée est obtenue en ajoutant ce déplacement et le contenu du registre `Début_données`. Le déplacement reçoit souvent le nom d'*adresse logique* et l'adresse effective en mémoire celui d'*adresse physique*.

Ce mécanisme reçoit le nom de *translation d'adresses*. Il suppose qu'à chaque accès à la mémoire une addition soit faite entre l'adresse logique émise

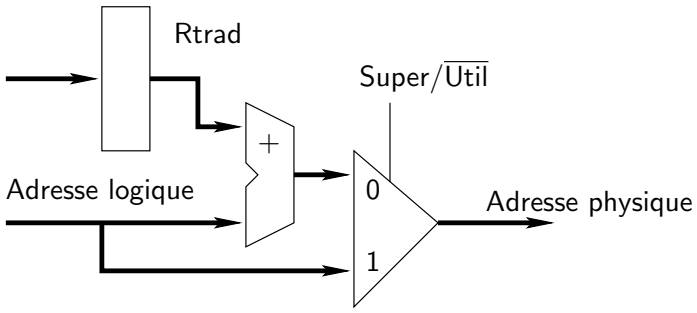


FIG. 24.4 – Matériel pour la translation d'adresses

par le processeur et le contenu d'un registre de traduction Rtrad. Dans beaucoup de cas les données des programmes s'exécutant en mode superviseur ne sont pas soumises à ce traitement (Cf. Figure 24.4).

En cumulant protection et translation d'adresse, on a un vérificateur d'accès pour chaque zone. Ces différents vérificateurs voient leurs résultats cumulés par un simple circuit qui donne un signal d'autorisation d'accès à la mémoire et le déplacement correspondant si il y a UN accès permis, un signal d'interruption interne si aucun accès n'est permis.

3. Entrées/sorties gérées par interruption

Rappelons le mécanisme, élémentaire et peu réaliste, mis en place pour faire des entrées/sorties (Cf. Chapitre 16) :

> Le processeur, via le programme pilote de périphérique, écrit une commande dans le registre du coupleur puis entre dans une boucle logicielle d'attente. Cette boucle scrute le registre d'état du coupleur jusqu'à ce que l'action correspondante soit terminée.

Nous décrivons dans cette partie comment il est possible d'éviter le phénomène d'attente active lors des entrées/sorties en utilisant le mécanisme des interruptions.

3.1 Interruptions en provenance des périphériques

La solution consiste à utiliser la possibilité donnée aux coupleurs d'émettre une interruption dès qu'ils ont terminé une tâche. Ils conservent un mot d'état indiquant qu'ils ont effectivement terminé.

Cette interruption est, naturellement, scrutée par le processeur. Mais il s'agit là d'une scrutation matérielle, faite par l'automate Partie Contrôle du processeur. Il n'y a donc plus de programme de scrutation et le processeur peut exécuter des instructions tout en maintenant cette observation nécessaire du fil de requête d'interruption IRQ.

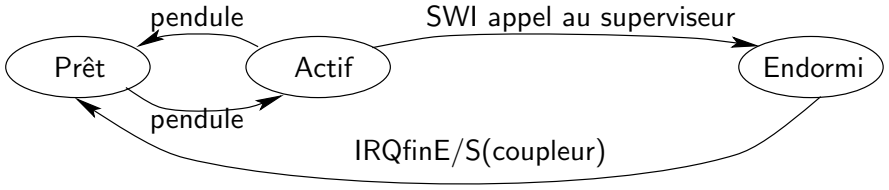


FIG. 24.5 – Les 3 états d'un processus et les causes de changement

Lorsque le processeur a détecté cette requête, le traitant peut lire le mot d'état du coupleur. Mais une seule fois suffit maintenant à confirmer que c'est bien *ce* coupleur qui a émis la requête.

3.2 Gestion de processus avec état *endormi*

Nous venons de voir que le processeur n'a plus besoin de scruter par programme le mot d'état d'un coupleur lancé dans une action d'entrée/sortie. Mais que peut-il faire d'autre ? Un programme comportant une lecture d'un caractère au clavier et le traitement de ce caractère ne peut évidemment pas se poursuivre tant que le caractère n'est pas obtenu. On va donc dire que ce programme, le processus qui lui est associé en fait, *s'endort* ou *se bloque*.

Les états des processus Aux deux états Prêt et Actif des processus, nous ajoutons un état Endormi (ou Bloqué). Ces 3 états sont représentés sur la figure 24.5. Détaillons les raisons qui font passer un processus d'un état à un autre :

1. Un processus Actif qui n'a pas d'entrées/sorties en cours peut être mis sur la file des Prêts à la suite d'une interruption causée par la pendule périodique de l'ordinateur.
2. Un processus Prêt, en tête de file, peut être remis en état Actif à la suite d'une interruption causée par la pendule périodique de l'ordinateur.
   Ces deux situations ne sont pas nouvelles par rapport au chapitre 23.
3. Si un processus Actif a besoin d'une action d'entrée/sortie, il fait un appel système (par une instruction `swi`) en donnant toutes les informations sur la nature de l'entrée/sortie souhaitée. Le système d'exloitation lance l'action en écrivant dans le coupleur les commandes nécessaires et met le processus en état Endormi.
4. Un processus Endormi sur attente du résultat d'une certaine action d'entrée/sortie est remis dans la file des Prêts par le gestionnaire de processus dès que l'interruption de fin d'entrée/sortie est émise par le coupleur en question.

La figure 24.6 donne deux situations d'entrelacement de quatre processus. Dans la situation a, seul le processus P2 lance une action d'entrée/sortie et l'action est faite rapidement par rapport à la période de la pendule. Dans la situation b, P2 et P3 lancent des actions et le traitement de l'entrée/sortie de P2 est un peu plus long.

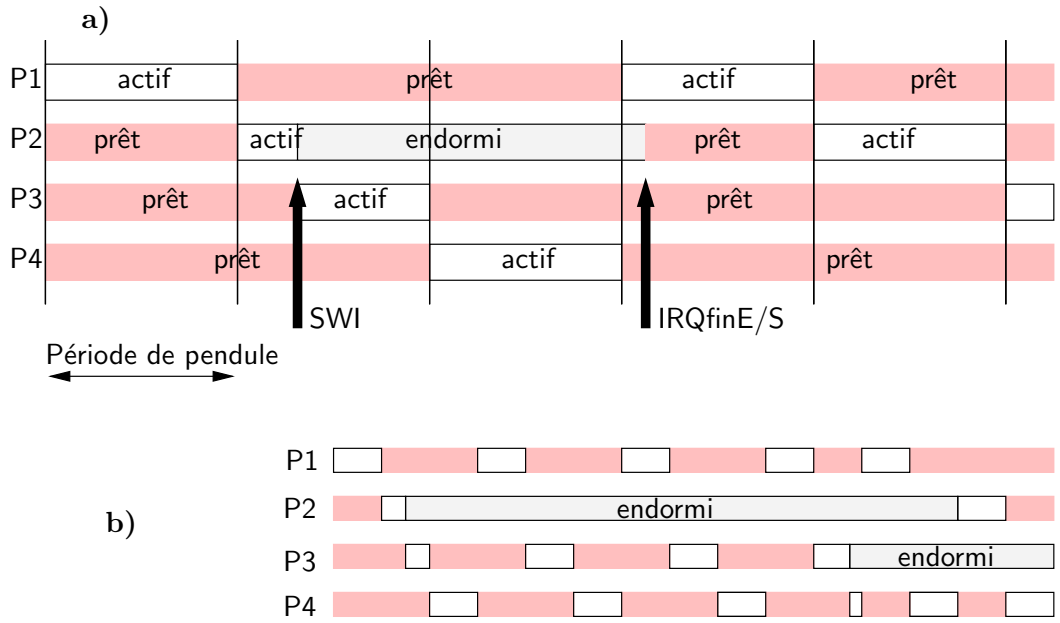


FIG. 24.6 – Deux situations d'entrelacement de 4 processus

```
attendre_périphérique () {
     dormeur = actif ;
     proc[actif].endormir = VRAI ;
     /* autoriser IT au niveau du coupleur */
     *commande_coupleur = AUTORISER_IT ;
     appel_superviseur_dormir ; }
```

FIG. 24.7 – Procédure d'attente

3.3 Programmation des traitants d'interruption

Dans la procédure d'attente du périphérique (Cf. Figure 24.7), le processus actif exécute un appel superviseur qui lui permet de se bloquer.

On trouve ainsi quatre traitants dans notre système : le traitant d'appel superviseur qui effectue l'écriture sur le coupleur, le traitant d'appel superviseur de blocage (Cf. Figure 24.8), le traitant d'interruption de réveil (Cf. Figure 24.9) et le traitant d'interruption de commutation du chapitre 23 qui gère les tranches de temps.

Noter que trois de ces traitants manipulent la structure de données décrivant les processus et qu'ils doivent donc se protéger les uns des autres pour accéder cette dernière (Cf. Paragraphe 4. du chapitre 22) , d'où le masquage des interruptions au niveau du processeur pour garantir l'exclusion mutuelle.

Procédure d'attente La procédure `attendre_périphérique` consiste à enregistrer le processus dans une variable (dormeur) du pilote de périphérique,

```
traitant_dormir () {
     empiler () ;
     sauver () ;
     masquer_it () ;
     /* la manipulation de la structure décrivant les processus */
     /* nécessite une exclusion mutuelle */
     si proc[actif].endormir alors actif_vers_bloqué () ;
     démasquer_it () ;
     restaurer () ;
     dépiler () ; }
actif_vers_bloqué () {
     proc[actif].état = BLOQUE ;
     prêt_vers_actif () ; }
```

FIG. 24.8 – Traitant de blocage

autoriser le coupleur à émettre une interruption lorsque le périphérique devient prêt et à s'endormir.

Traitant de blocage Le traitant `appel_superviseur_dormir` effectue la transition Actif vers Bloqué du processus qui l'appelle. Dans un premier temps, nous supposons que ce blocage est inconditionnel. Dans l'algorithme, le blocage n'est effectué que si la variable booléenne `endormir` du processus est vraie. La raison d'être de ce test sera expliquée après la présentation du traitant de réveil.

Traitant d'interruption de réveil La procédure de traitement `appel_superviseur_dormir` bloque le processus jusqu'à l'arrivée d'une interruption de réveil indiquant que le périphérique est prêt.

Le traitant de réveil accède au coupleur pour annuler l'autorisation d'émettre une interruption (sans quoi l'interruption serait prise en compte à nouveau au retour du traitant, le périphérique n'ayant pas été servi), réveille le processus bloqué (dont l'identité est stockée dans la variable dormeur du pilote) et le remet dans l'ensemble des processus prêts. Plus tard, le processus réveillé redevient actif et effectue le transfert de donnée avec le périphérique.

Là encore, nous considérons dans un premier temps que le réveil est toujours effectué par le traitant et nous verrons ensuite pourquoi celui-ci doit être conditionné par la variable endormir.

En l'absence de précaution particulière, il existe un aléa de fonctionnement lié au fait que le blocage du processus et l'autorisation des interruptions au niveau du coupleur ne sont pas effectuées de manière atomique.

Dans le cas favorable, l'interruption de réveil arrive après que le processsus se soit endormi et tout se passe correctement. Supposons à présent que la fin

```
traitant_reveil () {
      empiler () ;
      masquer_it () ;
      /* la manipulation de la structure décrivant les processus */
      /* nécessite une exclusion mutuelle */
      si proc[actif].état == BLOQUE ;
      alors
      bloqué_vers_actif () ;
      sinon
      /* le processus n'a pas eu le temps de s'endormir */
      proc[actif].endormir = FAUX ;
      démasquer_it () ;
      dépiler () }
bloqué_vers_actif () {
      proc[dormeur].état = PRET ;
      ajouter_prêt (dormeur) ; }
```

FIG. 24.9 – Traitant d'interruption de réveil

de la tranche de temps allouée au processus actif se termine juste après que le processus a autorisé le coupleur a l'interrompre et juste avant d'appeler `traitant_dormir`. Le processus se retrouve dans la file des prêts.

L'arrivée de l'interruption de réveil avant que le processus ne redevienne actif pour exécuter l'appel superviseur de blocage pose deux problèmes : le signal de réveil est perdu et le processus ne se réveillera jamais après s'être endormi ; plus grave, le traitant de réveil, croyant le processus déjà endormi, va essayer de l'ajouter en queue de la file des prêts à laquelle le processus appartient déjà et laisser la file dans un état incohérent.

On trouve deux manières principales de résoudre le problème. La première consiste à masquer les interruptions au niveau du processeur avant d'autoriser le coupleur à en émettre, de telle sorte que l'autorisation au niveau du coupleur et le blocage du processus deviennent atomiques (les interruptions étant démasquées au moment du réveil du nouveau processus actif au retour du traitant de blocage). La seconde consiste à mémoriser l'arrivée éventuelle du signal de réveil entre le moment où le coupleur est autorisé à interrompre et celui où le processus se bloque : le blocage du processus est annulé si le signal de réveil est arrivé entre temps (endormi remis à faux par le traitant de réveil).

L'utilisation de la variable de processus endormir illustre la deuxième stratégie. Elle présente l'avantage de fonctionner également dans le cas où le processus souhaiterait attendre plusieurs événements et répondre au premier qui se présente (la primitive UNIX select correspond à ce type de fonctionnalité).

Le problème de la file vide Lorsque le processus actif s'endort, la file des prêts peut être vide, tous les processus étant bloqués en attente

d'entrées/sorties et l'ordonnanceur n'a plus de processus candidat à élire.

Une stratégie consiste à indiquer dans la variable actif l'absence de processus et à exécuter une instruction spéciale qui stoppe le processeur (qui cesse d'exécuter des instructions) jusqu'à l'arrivée d'une interruption de réveil d'un processus (si le processeur ne peut être stoppé, il est toujours possible de lui faire exécuter une boucle de test de la file des prêts). Le premier processus réveillé par une interruption rétablit le fonctionnement normal du processeur et transite directement de l'état bloqué à l'état actif.

Une autre méthode consiste à modifier la procédure `retirer_prêt` de manière à retourner, en l'absence de processus prêt, un processus gardien spécial qui exécute une boucle infinie contenant un appel superviseur *dormir*.

Dès qu'il devient actif, le gardien se bloque et perd ausitôt le processeur. Si aucun processus n'a été réveillé depuis, le gardien est élu à nouveau et se bloque derechef, et ainsi de suite jusqu'à qu'un processus ait été réveillé et que la file des prêts soit non vide. La tête de la file des prêts est alors élue immédiatement.

4. Pour aller plus loin

Nous avons, dans la partie VI de ce livre, présenté la notion de processus et montré comment gérer différentes activités s'exécutant en pseudo parallélisme sur un seul processeur.

Dans un système d'exploitation la notion de processus est encore plus large : un utilisateur lambda peut créer lui-même des processus qui peuvent échanger des informations et se synchroniser de différentes façons. On parle alors de *programmation concurrente*.

Pour avoir un exposé complet des notions de synchronisation entre processus et trouver des compléments d'information sur la protection, le lecteur peut consulter par exemple un des ouvrages suivants : [Kra85, Bac90, Tan94, SG96].

Index

Bibliographie

[Aa91] H. Abelson et all : *Revised Report on the Algorithmic Language Scheme*. W. Clinger and J. Rees, 1991. 2.2.2

[AL78] Rodnay Zaks Austin Lesea : *Techniques d'interface aux microprocesseurs*. Sybex, 1978. 2.1.1

[Bac90] M.J. Bach : *Conception du système UNIX (traduction française de "The design of the Unix operating system, 1986)*. Ed. Masson, Ed Prentice-Hall, 1990. 4.

[BB83] P. Berlioux et Ph. Bizard : *Algorithmique – construction, preuve et évaluation des programmes*. Dunod, 1983. 2.4.2, 10.22

[BB88] P. Berlioux et Ph. Bizard : *Algorithmique – structures de données et algorithmes de recherche*. Dunod, 1988. 5.

[Ben91] Cl. Benzaken : *Systèmes formels – Introduction à la logique et à la théorie des langages*. Masson, 1991. 1.1.2, 1.2

[BGN63] A.W. Burks, H.H. Goldstine et J. Von Neumann : *Preliminary discussion of the logical design of an electronic computing instrument, (article de 1946), dans John von Neumann, Collected works, volume V*. Ed. Pergamon Press, 1963. 2.2, 1.1

[BRWSV87] R. Brayton, R. Rudell, A. Wang et A. Sangiovanni-Vincentelli : Mis : a multiple-level logic optimization system. *IEEE Transaction on CAD*, pages 1062–1081, novembre 1987. 4.1

[Bry86] R.E. Bryant : Graph-based algorithms for boolean functions manipulation. *IEEE Transaction on Computers*, (8):677–692, août 1986. 3.3, 3.3.2

[CDLS86] M. Cand, E. Demoulin, J.L. Lardy et P. Senn : *Conception des circuits intégrés M.O.S. Eléments de base - perspective*. Eyrolles, collection technique et scientifique des télécommunications, 1986. 2.2.4, 3.2

[CGV80] P.-Y. Cunin, M. Griffith et J. Voiron : *Comprendre la compilation*. Springer Verlag, 1980. 2.4.2, 1.1.1, 2.2.1, 1.2

[CW96] J. P. Colinge et F. Van De Wiele : *Physique des dispositifs semi-conducteurs*. Editions De Boek Université, Paris, Bruxelles, 1996. 1.3

[GDS98] N. Garcia, A. Damask et S. Schwarz : *Physics for Computer Science Students (with emphasis on atomic and semi-conductor physics)*. Editions Springer-Verlag, 1991, 1998. 1.3

[Gir95] J.-Y. Girard : *La machine de Turing*. Seuil, 1995. 2.1

[HP94] J. Hennessy et D. Patterson : *Organisation et conception des ordinateurs. L'interface matériel/logiciel*. Ed. Dunod, 1994. 3.3, 14

[Ifr94] G. Ifrah : *L'histoire universelle des chiffres : l'intelligence des hommes racontée par les nombres et le calcul (2 tomes)*. Robert Laffont, collection Bouquins, 1994. 2.1.1

[Int97] INTEL : *The complete guide to MMX technology.* McGraw-Hill, 1997. 1.4.1

[Kar53] M. KARNAUGH : The map method for synthesis of combinational logic. *Circuits AIEE Transactions on Communications and Electronics*, pages 593–599, 1953. 4.

[KB90] R.E. Bryant K. BRACE, R. Rudell : Efficient implementation of a bdd package. *Proceedings of the 27th ACM/IEEE Design Automation Conference IEEE0738*, 1990. 4.3

[Kra85] S. KRAKOWIAK : *Principes des systèmes d'exploitation des ordinateurs.* Dunod Informatique, 1985. 4.2, 4.

[Kun65] Jean KUNTZMANN : *Algèbre de Boole.* Dunod, 1965. 4.

[Kun67] Jean KUNTZMANN : *Sélection parmi des communications présentées au colloque consacré à l'Algèbre de Boole.* Dunod, 1967. 4.

[Las98] J. LASSÈGUE : *Turing.* Les belles lettres - collection Figures du savoir, 1998. 2.1

[Liv78] C. LIVERCY : *Théorie des programmes - schémas, preuves, sémantique.* Dunod, 1978. 2.

[Mul89] J.M. MULLER : *Arithmétique des ordinateurs. opérateurs et fonctions élémentaires.* Masson, collection Etudes et Recherches en Informatique, 1989. 2.2

[Org72] J.E. ORGANICK : *The MULTICS system : an examination of its structure.* The MIT Press, Cambridge MA, 1972. 2.6

[SFLM93] P.-C. SCHOLL, M.-C. FAUVET, F. LAGNIER et F. MARANINCHI : *Cours d'informatique : langages et programmation.* Masson, 1993. 1., 1.4, 2.1, E8.9

[SG96] A. SILBERSCHATZ et P. B. GALVIN : *Principes des systèmes d'exploitation.* Addison-Wesley, 1996. 4.

[SL96] André SEZNEC et Fabien LLOANSI : Etude des architectures des microprocesseurs mips r10000, ultrasparc et pentiumpro. Publication interne 1024, INRIA, Programme 1 - Architecture parallèles, bases de données, réseaux et systèmes distribués – Projet CAPS, mai 1996. 1.4.4

[Tan94] A. TANENBAUM : *Systèmes d'exploitation : systèmes centralisés, systèmes distribués.* InterEditions, 1994. 4.

[Tur54] A. M. TURING : Solvable and unsolvable problems. *Science News*, 31:7–23, 1954. 2.1

[WM94] R. WILHELM et D. MAURER : *Les compilateurs : théorie, construction, génération.* Masson, 1994. 1.1.1, 2.2.1, 1.2

[Zak80] Rodnay ZAKS : *Applications du 6502.* Sybex, 1980. 2.1.1