Architecture des ordinateurs

Exécution des programmes en langage machine

Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

15 janvier 2020

Organisation

- Cours (stephane.devismes@univ-grenoble-alpes.fr)
  http://www-verimag.imag.fr/~devismes/WWW/enseignements.html
  - Mercredi 9h30-11h00 (12 séances, 1h30)
- TD et TDE (emeric.malevergne@univ-grenoble-alpes.fr)
  - TD : Lundi 10h15-12h15 (11 séances, 21h)
  - TDE : Mardi 10h15-12h15 (11 séances, 21h)
- Note : 0.2 CC1 + 0.6 examen + 0.2 CC2
  CC1 = 0.5 Partiel + 0.5 Colles (moyenne des 2 Colles)
  CC2 = note de TP
- *Vacances hiver :* du 24 février au 28 février
- Partiels : 9-12 mars
- *Vacances printemps :* du 20 avril au 24 avril
- Examen : 11-26 mai (Jury : 5 juin)
- Deuxième session : 16-24 juin (Jury : 3 juillet)
- Supports : Slides à trous + sujets TD/TDE

But du cours

Comment les programmes sont exécutés par un processeur ?

Physique (ALM, L3)

- Processeur

- Circuits (séquentiels et/ou combinatoires)

- Transistors

Logique (INF451)

Processeur ≈ Machine (circuit intégré) qui traite des informations numériques

Rappel :

- numérique (digitale) : représentation d'une tension par une valeur entière
- analogique : aiguille du voltmètre

Bibliographie

- *Architectures logicielles et matérielles*, Amblard, Fernandez, Lagnier, Maraninchi, Sicard, Waille, Dunod 2000
- *Architecture des ordinateurs*, Cazes, Delacroix, Dunod 2003
- *Computer Organization and Design : The Hardware/Software Interface*, Patterson and Hennessy, Dunod 2003.
- *Processeurs ARM*, Jorda. DUNOD 2010.
- https://im2ag-moodle.e.ujf-grenoble.fr/course/view.php?id=336

Plan du cours

1. Codage des informations
2. Modèle de Von Neumann
3. Langage d'assemblage (ARM) et language machine
4. Instruction de rupture de séquence et programmation des structures de contrôle
5. Programmation à partir des automates reconnaisseurs
6. Programmation des appels de procédure et fonction (*3 séances*)
7. Introduction à la structure interne des processeurs
8. Vie des programmes
9. Organisation d'un ordinateur
10. Optimisations
11. Introduction aux circuits

Plan

Codage des informations et représentation des entiers par des vecteurs binaires

Exécution des programmes en langage machine

Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

15 janvier 2020

Définition

Un codage est une bijection entre un ensemble d'informations et un ensemble de naturels.

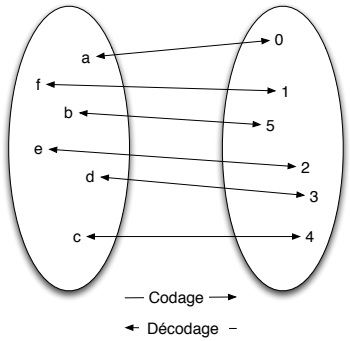


FIGURE – Exemple de bijection

Pour N informations, on choisit en général l'intervalle $[0, N-1]$. Fonctions de codage dans un sens, de décodage dans l'autre.

Exemples (1/3) : Codage des 16 couleurs du Commodore 64

B	$b_3b_2b_1b_0$		B	$b_3b_2b_1b_0$		B	$b_3b_2b_1b_0$	
0	0 0 0 0	noir	5	0 1 0 1	vert	10	1 0 1 0	rose
1	0 0 0 1	blanc	6	0 1 1 0	bleu	11	1 0 1 1	gris foncé
2	0 0 1 0	rouge	7	0 1 1 1	jaune	12	1 1 0 0	gris moyen
3	0 0 1 1	cyan	8	1 0 0 0	orange	13	1 1 0 1	vert pâle
4	0 1 0 0	violet	9	1 0 0 1	brun	14	1 1 1 0	bleu pâle
						15	1 1 1 1	gris pâle

Code_C64 (violet) = 4 ; Decode_C64 (12) = gris moyen.

Remarque

Pas de propriétés particulières : ordre, relation avec les couleurs, notion de clair/foncé, . . .

Exemples (2/3) : Codage des 16 couleurs sur les premiers PC couleurs

B	$b_3b_2b_1b_0$		B	$b_3b_2b_1b_0$		B	$b_3b_2b_1b_0$	
0	0 0 0 0	noir	5	0 1 0 1	violet	10	1 0 1 0	vert pâle
1	0 0 0 1	bleu	6	0 1 1 0	brun	11	1 0 1 1	cobalt
2	0 0 1 0	vert	7	0 1 1 1	gris	12	1 1 0 0	rose
3	0 0 1 1	cyan	8	1 0 0 0	noir pâle	13	1 1 0 1	mauve
4	0 1 0 0	rouge	9	1 0 0 1	bleu pâle	14	1 1 1 0	jaune
						15	1 1 1 1	blanc

Propriétés

Chaque bit a un ≪ sens ≫ : En regardant le code en base 2, on voit un bit de rouge (b2), un bit de vert (b1), un bit de bleu (b0) et un bit de clair (b3). Le codage du violet est fait de bleu et de rouge . . .

Remarque

Aujourd'hui, codage en niveaux d'intensité de rouge, vert et bleu (*RVB*) sur 8 bits.

Exemples (3/3) : Code ASCII (Ensemble des caractères affichables)

ASCII = ≪ American Standard Code for Information Interchange ≫

On obtient le tableau ci-dessous par la commande Unix `man ascii`

32	␣	33	!	34	"	35	#	36	$	37	%	38	&	39	'
40	(| 41 |)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	del

Code_ascii (q) = 113 ; Decode_ascii (51) = 3.

Ensemble muni de relations et d'opérations

Codage qui ≪ respecte ≫ les **relations et opérations**.

Par exemple, dans le code ASCII :

- respect de l'ordre alphabétique
- passage de majuscules à minuscule (code majuscule + 32 = code minuscule)
- respect de l'ordre des chiffres décimaux

Application en langage C

```
printf("%c",'A'+'b'-'a');

printf("%c",'A'+3);

char c;
scanf("%c",&c);
if(c>='a' && c<='z')
   printf("%c est une lettre minuscule",c);
```


UTF-8

- Codage extensible, compatible avec ASCII
- Permet de représenter plus d'un million de caractères

Caractères codés	Représentation binaire UTF-8	Signification
U+0000 à U+007F	0xxxxxxx	1 octet codant 1 à 7 bits
U+0080 à U+07FF	110xxxxx 10xxxxxx	2 octets codant 8 à 11 bits
U+0800 à U+0FFF	11100000 101xxxxx 10xxxxxx	3 octets codant 12 à 16 bits

Source wikipédia.
Orange : #octets supplémentaires (unaire) ; blanc : séparateur ; vert : bits pour le codage du caractère

Codage d'ensemble structuré : ex villes olympiques (1/2)

E= {Albertville, Athènes, Atlanta, Chamonix, Grenoble, Los Angeles, Melbourne, Mexico, Montréal, Paris, Pékin, Rome, Sydney, Tokyo}

Codage en utilisant une hiérarchie :

- niveau 1 : continents → Amérique, Asie, Europe, Océanie.
- niveau 2 : pays → en Amérique, Canada, E.U., Mexique.
- niveau 3 : régions → en France, Ile de France, Rhône-Alpes.
- niveau 4 : villes → en Rhône-Alpes, Alberville, Chamonix, ...

Les villes sont alors les feuilles d'un arbre.

Codage de chaque continent, pays, région, ville par un entier. Pour les objets de même niveau, on utilise l'ordre alphabétique.

Code des villes olympiques (2/2)

Le code d'une ville est alors un quadruplet.

continent	pays	région	ville	code quadruplet
Amérique	Canada		Montréal	(0, 0, 0, 0)
	Etats-Unis	Californie	Los Angeles	(0, 1, 0, 0)
		Géorgie	Atlanta	(0, 1, 1, 0)
	Mexique		Mexico	(0, 2, 0, 0)
Asie	Chine		Pékin	(1, 0, 0, 0)
	Japon		Tokyo	(1, 1, 0, 0)
Europe	France	Ile de France	Paris	(2, 0, 0, 0)
		Rhône-Alpes	Albertville	(2, 0, 1, 0)
			Chamonix	(2, 0, 1, 1)
			Grenoble	(2, 0, 1, 2)
	Grece		Athenes	(2, 1, 0, 0)
	Italie		Rome	(2, 2, 0, 0)
Océanie	Australie	New South Wales	Sydney	(3, 0, 0, 0)
		Victoria	Melbourne	(3, 0, 1, 0)

Si ce code vous parait « inventé », essayez donc de comprendre le code des unités d'enseignement, il y a quelques ressemblances ...

Codage d'ensemble structuré : autres exemples

Codage d'instructions, de commandes, d'ordres,...

On peut imaginer de coder par des N_uplets :

- des ordres, des commandes à un robot,
- des instructions d'ordinateur,
- ...

numéro du bras	direction	angle	vitesse de déplacement	...
3	Nord	135	haute	...
2	Sud	45	très faible	...

numéro de case mémoire	action	...
12345	remise à zéro	...
47	incrémentation	...

Correspondance entre n_uplet et naturel (2/2)

Produit cartésien : correspondance entre n_uplet et naturel (1/2)

$E = \{0, 1, 2, 3\} \times \{0, 1, 2, 3, 4\}$
Il y a 4*5=20 informations
Le code est un entier dans $\{0, 1, 2, .., 19\}$

Compléter le tableau ci-après :

	0	1	2	3	4
0					
1					
2					
3					

Conclusion sur le codage : Où est le code ?

- Le code n'est pas dans l'information codée.
  Par exemple : 14 est le code du jaune dans le code des couleurs du PC ou le code du couple (2,4) ou le code du bleu pâle dans le code du commodore 64.
- Pour interpréter, comprendre une information codée il faut connaître la règle de codage. Le code seul de l'information ne donne rien, c'est le système de traitement de l'information (logiciel ou matériel) qui « connait » la règle de codage, sans elle il ne peut pas traiter l'information.

Numération de position

En numération de position, avec N chiffres en base b on peut représenter les b^N naturels de l'intervalle [0, b^N-1]

Exemple : en base 10 avec 3 chiffres on peut représenter les 10^3 naturels de l'intervalle $[0, 999]$.

Avec N chiffres binaires (base 2) on peut écrire les 2^N naturels de l'intervalle $[0, 2^N - 1]$

Exercice : Enumérer les nombres représentables sur **3 chiffres binaires.**

Logarithme et taille de donnée (1 sur 2)

On ne s'intéresse qu'à la base 2 : un chiffre binaire est appelé bit.

Logarithme : opération réciproque de l'élévation à la puissance
Si $Y = 2^X$, on a $X = \log_2 Y$

Pour représenter en base 2, K naturels différents,
il faut $\lceil \log_2 K \rceil$ chiffres en base 2

Si K est une puissance de 2, $K = 2^N$, il faut N bits.
Si K n'est pas une puissance de 2, soit P la plus petite puissance de 2 telle que $P > K$, il faut $\log_2 P$ bits.

Logarithme et taille de donnée (2 sur 2)

Par exemple, on veut représenter les naturels de l'intervalle [0,...,9] :
- 10 entiers
- La puissance de 2 supérieure la plus petite est $16 = 2^4$

Il faut 4 bits.

De même que pour les naturels de l'intervalle [0,...,15]

Pour représenter les naturels de l'intervalle [0,...,255],
- il faut 8 bits ($2^8 = 256$)

Pour coder les naturels de [0,...,127],
- il faut 7 bits ($2^7 = 128$).

Quelques valeurs à connaître

X	2^X
0	1
1	2
2	4
3	8
4	16
8	256
10	1 024 ($\approx$ 1 000)
16	65 536
20	1 048 576 ($\approx$ 1 000 000, 1 Méga)
30	1 073 741 824 ($\approx$ 1 000 000 000, 1 Giga)
32	4 294 967 296

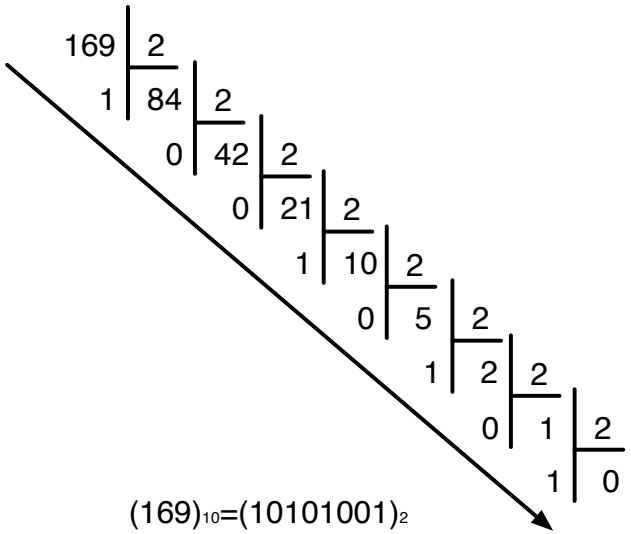
Conversion base 10 vers base 2 : Première méthode

La méthode des restes successifs donnent les chiffres en commençant par celui des unités.

169 = 84 $\times$ 2 + $\boxed{1}$ (chiffre des unités = 1)
169 = (42 $\times$ 2 + $\boxed{0}$)$\times$ 2 + 1 (chiffre suivant = 0)
169 = ((21 $\times$ 2 + $\boxed{0}$) $\times$ 2 + 0)$\times$ 2 + 1
169 = (((10 $\times$ 2 + $\boxed{1}$) $\times$ 2 + 0) $\times$ 2 + 0)$\times$ 2 + 1
169 = ((((5 $\times$ 2 + $\boxed{0}$) $\times$ 2 + 1) $\times$ 2 + 0) $\times$ 2 + 0)$\times$ 2 + 1
169 = (((((2 $\times$ 2 + $\boxed{1}$) $\times$ 2 + 0) $\times$ 2 + 1) $\times$ 2 + 0) $\times$ 2 + 0)$\times$ 2 + 1
169 = ((((((1 $\times$ 2 + $\boxed{0}$) $\times$ 2 + 1) $\times$ 2 + 0) $\times$ 2 + 1) $\times$ 2 + 0) $\times$ 2 + 0)$\times$ 2 + 1

On a ainsi $169_{10} = 10101001_2$

Conversion base 10 vers base 2 : Deuxième méthode



$(169)_{10}=(10101001)_2$

Conversion base 10 vers base 2 : Troisième méthode

Utilisation des puissances de 2 :

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

Conversion base 2 vers base 10

Soit $a_{n-1}a_{n-2}\dots a_1a_0$ un nombre entier en base 2

En utilisant les puissances de 2 :

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

$a_{n-1}a_{n-2}\dots a_1a_0$ vaut $a_{n-1}2^{n-1}+a_{n-2}2^{n-2}+\dots+a_12^1+a_02^0$ en base 10

Exemple : 1010 vaut
$1\times2^3+0\times2^2+1\times2^1+0\times2^0=2^3+2^1=8+2=10$

Exercices

- Sur 4 bits . . .

$2^3=$	$2^2=$	$2^1=$	$2^0=$
$8_{10}=$	$1_{10}=$	$4_{10}=$	
$10_{10}=$	$12_{10}=$	$6_{10}=$	
$13_{10}=$	$11_{10}=$	$5_{10}=$	

- Sur 8 bits . . .

$2^7=$	$2^6=$	$2^5=$	$2^4=$	$2^3=$	$2^2=$	$2^1=$	$2^0=$
$11_{10}=$		$67_{10}=$		$188_{10}=$			

Utilisation de la base 16 : hexadécimal

Les 16 chiffres sont 0, 1, .., 8, 9, A, B, C, D, E, F.
Ils représentent respectivement les naturels qui s'écrivent 0, 1, .. 8, 9, 10, .., 15 en décimal.

Ecrire les nombres de 0 à 15 en base 10, 2 et 16 :

base 10	base 2	base 16	base 10	base 2	base 16
0			8		
1			9		
2			10		
3			11		
4			12		
5			13		
6			14		
7			15		

Passage d'une base à l'autre

169 = $\boxed{10}\times16+\boxed{9}$
169 s'écrit A9 en hexadécimal

On obtient cette écriture en « remplaçant » dans l'écriture en base 2, 1010 1001 : 1010 par A et 1001 par 9

base	x	y	z	t
2	010010011101	100011110000		
16			0...0...8..	5...5...A..

Conversion base 16 vers base 10

Soit $a_{n-1}a_{n-2}\dots a_1a_0$ un nombre entier en base 16

$a_{n-1}a_{n-2}\dots a_1a_0$ vaut $a_{n-1}16^{n-1}+a_{n-2}16^{n-2}+\dots+a_116^1+a_016^0$ en base 10

Exemple : FF vaut $15\times16^1+15\times16^0=15\times16+15=255$

Exercice : codage par champs

On veut coder une information du style : `lundi 12 janvier`

Triplet
Codage du jour : {lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche} : entier entre 1 et 7
Codage du quantième du jour dans le mois : entier entre 1 et 31
Codage du mois : entier entre 1 et 12

lundi 12 janvier codé par :
date associée au code 011 00101 0011 :
date associée au code 111 11111 1111 :

Exercice : codage d'une instruction ARM (1 sur 2)

Les instructions sont codées sur 32 bits

31 28	27 26	25	24 21	20	19 16	15 12	11 8	7 0
cond	0 0	I	1 1 0 1	S	0 0 0 0	rd	0 0 0 0	immediat

FIGURE – Codage de l'instruction `mov` version 1

Exercice : quel est le codage de l'instruction `MOV r5, ♯12`

Exercice : codage d'une instruction ARM (2 sur 2)

31 28	27 26	25	24 21	20	19 16	15 12	11 4	3 0
cond	0 0	I	1 1 0 1	S	0 0 0 0	rd	0 0 0 0 0 0 0 0	rm

FIGURE – Codage de l'instruction `mov` version 2

Exercice : quel est le codage de l'instruction `MOV r5, r7`

Une première idée : bit signe = bit de poid fort, suivi de valeur absolue

Par exemple sur **4 bits** :

$$(+3)_2 = 0011,\ (-3)_2 = 1011,\ (-2)_2 = 1010$$

Problèmes :

- deux représentations pour zéro : 0000 ou 1000
- $(-3) < (-2)$, mais $1011 > 1010$
- $(3) + (-3) = 0$ mais $0011 + 1011 = 1110 \neq 0$
- algorithme d'addition complexe

Solution : Complément à deux

Sur n bits, en choisissant 00…000 pour le codage de zéro, il reste $2^n - 1$ possibilités de codage : la moitié pour les positifs, la moitié pour les négatifs.

Attention, ce n'est pas un nombre pair, l'intervalle des entiers relatifs codés ne sera pas symétrique.

Principe :

- Les entiers positifs sont codés par leur code en base 2
- Les entiers négatifs sont codés de façon à ce que code(a) + code(-a) = 0

D'où sur 8 bits, intervalle représenté $[-128, +127] = [-2^7, 2^7 - 1]$

- $x \geq 0 \quad x \in [0, +127]$: CodeC2(x)=x
- $x < 0 \quad x \in [-128, -1]$: CodeC2(x)=x+256 = x+2^8
  (x étant négatif et ≥ -128, x+2^8 est « codable » sur 8 bits)
  (x+$2^8 > 127$, donc pas d'ambiguïté)

CodeC2(a)+CodeC2(-a) = a-a+$2^8 = 0$ (sur 8 bits)

Complément à deux : exemples

CodeC2(3)=3, CodeC2(127)=127, CodeC2(0)=0,
CodeC2(-128)=-128+256=128, CodeC2(-1)=-1+256=255

En binaire sur 8 bits

base 10	base 2
3	0000 0011
127	0111 1111
0	0000 0000
-128	1000 0000
-1	1111 1111

Complément à deux sur 8 bits : tous les entiers relatifs

entier relatif	Code(base10)	CodeC2(base2)
-128	128	1000 0000
-127	129	1000 0001
-126	130	1000 0010
...		
-1	255	1111 1111
0	0	0000 0000
1	1	0000 0001
2	2	0000 0010
...		
12	12	0000 1100
...		
127	127	0111 1111

Correspondances entre les relatifs de l'intervalle [-8,+7] et le complément à 2 sur 4 bits

Établir un tableau où les codes seront représentés en base 10 et en base 2.

Complément à deux : trouver le code d'un entier négatif

Soit un entier relatif positif a codé par les n chiffres binaires :
$a_{n-1}a_{n-2}...a_1a_0$

$$\begin{aligned}
\text{valeur(-a)} &= 2^n - \text{valeur(a)} \\
&= 2^n - (a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + ... + a_1 2 + a_0) \\
&= (2^{n-1} + 2^{n-1}) - (a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + ... + a_1 2 + a_0) \\
&= (1 - a_{n-1})2^{n-1} + 2^{n-1} - (a_{n-2}2^{n-2} + ... + a_1 2 + a_0) \\
&= ..... \\
&= (1 - a_{n-1})2^{n-1} + (1 - a_{n-2})2^{n-2} + ... + (1 - a_0) + 1
\end{aligned}$$

Règle : écrire le code de la valeur absolue, inverser tous les bits, ajouter 1

Complément à deux binaire

- On inverse les bits de l'écriture binaire de sa valeur absolue, on fait ce qu'on appelle le complément à un,
- On ajoute 1 au résultat (les dépassements sont ignorés).

Exemple : 5 codé sur 8 bits
$(5)_2 = 00000101$
Complément à un : 11111010
Complément à deux : 11111011, c'est le code de (-5)

Opération inverse :
Complément à un de 11111011 : 00000100
Complément à deux : 00000101

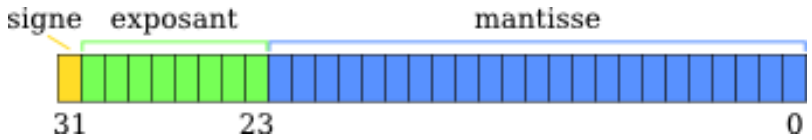
Analogie possible : dire [l'heure et] les minutes !

Horloge (presque) habituelle :

source Maison du monde.

Autre horloge, (presque) possible :

Les nombres à virgule flottante



source Wikipedia.

Les nombres à virgule flottante

- Norme **IEEE 754**
- Codage par champ (exemple sur 32 bits) : Signe (1 bit), Exposant (8 bits), Mantisse (23 Bits)
- Valeur $= (-1)^{signe} * 1,\text{Mantisse}_2 * 2^{Exposant-127}$
- Exceptions : 0, +Infini, -Infini, NaN, nombres proches de 0 ...
- Intervalle : [-3.4 10^{38} ;3.4 10^{38}] avec la moitié des nombres entre [-2 ;2]

Analogie possible : les calculatrices scientifiques

Affichage d'une calculatrice scientifique (simple) :

source Texas Instruments.

Les nombres à virgule flottante : code à trous ! (1/2)

```
double a=0.1;

while(a != 1.0){
     printf("Valeur de a : %f\n",a);
     a+=0.1;
     }
```

Que fait ce code ?

Boucle infinie !

0.1 n'est pas représentable !

Les nombres à virgule flottante : code à trous ! (2/2)

```
double a=0.1;
int i;

for(i=0;i<9;i++) a+=0.1;

printf("Valeur de a : %f\n",a);

if(a == 1.000000)  printf("gagné\n");
else printf("perdu\n");
```

Que fait ce code ?

```
Valeur de a : 1.000000
perdu
```

affichage arrondi !

Addition (1/3)

Rappel : en base 10

	Retenue	1	1	1	
	A		7	2	9
+	*B*		2	9	9
=	*A*+*B*	1	0	2	8

On a des retenues, des propagations de retenues

En base 2

	Retenue	0	0	1	1	
	A		1	0	1	1
+	*B*		0	0	1	1
=	*A*+*B*		1	1	1	0

Addition (2/3)

L'algorithme que le processeur exécute est le même que celui pratiqué à la main si ce n'est que la taille des données est bornée.

A, B deux naturels sur N bits, la somme A+B peut réclamer **un bit supplémentaire**

Le processeur gère un indicateur C (pour Carry) : on parle de retenue sortante

Exemple sur 4 bits :
$9_{10} + 8_{10} = 1001_2 + 1000_2$
La somme en base 2 s'écrit 10001 qui code bien l'entier 17_{10}
Pour le processeur les **4 bits** de résultat sont 0001 et C=1

Addition (3/3)

Somme de deux relatifs : même algorithme
Interprétation différente du résultat

n entier relatif représenté sur 4 bits : $n \in [-8, +7]$
$5_{10} + 2_{10} = 0101_2 + 0010_2 = 0111_2 = 7_{10}$
$5_{10} + 3_{10} = 0101_2 + 0011_2 = 1000_2 = -8_{10}$
car 1000 en complément à 2 représente -8 !!!

Dans ce cas, le processeur utilise l'indicateur *V* pour donner cette information.

$V = 1$ indique un débordement, auquel cas les deux dernières retenues sont de valeurs différentes.

NE PAS OUBLIER LA RETENUE SORTANTE !

Exercice

Donnez le résultat apparent des **additions d'entiers relatifs** suivantes, puis convertissez le résultat en base 10 :

0	1	1	0	0	1	1	0
0	0	0	1	0	1	0	0

0	1	1	0	0	1	1	0
0	0	1	1	0	0	1	0

Exercice

Donnez le résultat apparent **des soustractions d'entiers relatifs** suivantes, puis convertissez le résultat en base 10 :

0	0	0	1	1	0	1	1
0	0	1	1	0	0	1	0

0	1	0	0	1	1	1	1
0	0	1	1	0	0	1	0

Remarque : le flag V détecte aussi les débordements (overflow) sur les soustrations de relatifs

Autres opérations

- soustraction A-B = A + complémentaire à deux de B
  La différence n'est pas toujours possible . . .
- multiplication par une puissance de 2 : décalage à gauche
- division par une puissance de 2 : décalage à droite
- modulo par une puissance de 2 (mod), reste d'une division par 2^k
  - Rappel : en base 10, 3159 mod 10 = 9, 3159 div 10 = 315

Retour sur les entiers naturels

- $A - B = A + \text{CodeC2}(-B)$
- Que signifie C dans ce contexte ?
- $A - B$ sans le complément à deux : notion d'emprunt

$14 - 3$ (sur 8 bits)

$E = 0$	0	0	0	0	0	1	1	
	0	0	0	0	1	1	1	0
$-$	0	0	0	0	0	0	1	1
	0	0	0	0	1	0	1	1

$14 + \text{CodeC2}(-3)$ (sur 8 bits)

$C = 1$	1	1	1	1	1	0	0	
	0	0	0	0	1	1	1	0
$+$	1	1	1	1	1	1	0	1
	0	0	0	0	1	0	1	1

$C = \overline{E}$. Donc, C=1 *ssi* le résultat de la soustraction est correct, c-à-d, $A \geq B$.

Indicateurs

	naturel	**relatif**
overflow addition	$C = 1$	$V = 1$
overflow soustraction	$C = 0$	$V = 1$

2 autres indicateurs (flags) :

- N : bit de signe (1 si négatif)
- Z : test si nulle ($Z = 1$ si nulle)

Les indicateurs permettent aussi d'évaluer les conditions ($<$, $>$, $\leq$, $\geq$, $=$, $\neq$).

Pour évaluer une condition entre A et B, le processeur positionne les indicateurs en fonction du résultat de $A - B$.

Exemple : Supposons que A et B sont des entiers naturels. Alors, $A - B$ provoque un overflow (c'est-à-dire, $C = 0$) si et seulement si $A < B$.

Plan

Modèle de Von Neumann : qu'est ce qu'un ordinateur ?

Exécution des programmes en langage machine

Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

15 janvier 2020

Historique

Les premiers ordinateurs ont été construit pour des besoins militaires (calcul balistique, décryptage, ...)

- **1936**, Turing invente la *machine de Turing*, les concepts de programmation et de programme
- **Peu avant la seconde guerre mondiale**, les premières calculatrices électromécaniques, construites selon les idées de Turing
- **Lors de la 2e guerre mondiale**, le *Colossus* : le premier ordinateur fonctionnant en langage binaire
- **1945**, L'*ENIAC* est le premier ordinateur entièrement électronique ayant la puissance des machines de Turing
- **1948**, Small-Scale Experimental Machines, les premières machines à architecture de Von Neumann
- **1952**, IBM produit son premier ordinateur, l'*IBM 701*, pour la défense américaine
- **Années 70**, premiers *micro-ordinateurs*

Générations

- **Première génération :** (à partir de 1946) l'ENIAC
- **Deuxième génération :** (1956-1963) remplacement des tubes électroniques par des transistors
- **Troisième génération :** (1963-1971) circuit intégré
- **Quatrième génération :** (1971 à nos jours) microprocesseur

Qu'est ce qu'un ordinateur ?

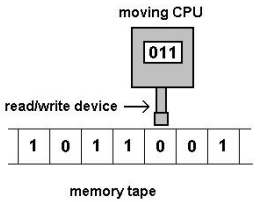
Vision concrète, matérielle

- des circuits : unité de calcul (UAL), mémoire
- des fils (nappes de fils)
- des histoires de potentiel
- du temps de calcul

Vision abstraite

- des modèles
- des instructions de calcul
- des fonctions de mémorisation
- des informations (bits, mots binaires. . .)

Un modèle (formel) de calcul : la machine de Turing (1/5)



- Un ruban divisé en cases. Le ruban est supposé de longueur infinie vers la gauche ou vers la droite.
- Une tête de lecture/écriture peut lire, écrire, et se déplacer vers la gauche ou vers la droite du ruban.
- Un registre d'état qui mémorise l'état courant. Le nombre d'états possibles est toujours fini. Il existe un état spécial appelé état de départ.
- Une table d'actions indique quel symbole écrire, comment déplacer la tête de lecture, et quel est le nouvel état, en fonction du symbole lu sur le ruban et de l'état courant. Si aucune action n'est applicable, la machine s'arrête.

Exemple de modèle de calcul : la machine de Turing (2/5)

Chaque case contient un symbole parmi un alphabet fini. L'alphabet contient un symbole spécial ≪ blanc ≫.

Utilisation : incrémentation d'un entier naturel écrit en binaire
Etat initial du ruban :

- Plusieurs cases consécutives remplies avec des 1 et des 0 avec plusieurs cases blanches à gauche (au moins deux) et plusieurs cases blanches à droite (au moins une).
- La tête de lecture est initialement sur le bit de poids fort (le plus à gauche).

Actions : placer la tête de lecture sur le bit de poids faible, incrémenter en propageant la retenue éventuelle, puis replacer la tête de lecture sur le bit de poids fort.

Exemple de modèle de calcul : la machine de Turing (3/5)

Etat initial : I

Etat	Lu	Ecrit	Direction	Etat
I	1	1	D	I
I	0	0	D	I
I	_	_	G	A
A	0	1	G	Ok
A	1	0	G	A
A	_	1	G	Ok
Ok	0	0	G	Ok
Ok	1	1	G	Ok
Ok	_	_	D	Fini

Exemple de modèle de calcul : la machine de Turing (4/5)

Donnez la trace d'exécution à partir de l'entrée 11100111

Exemple de modèle de calcul : la machine de Turing (5/5)

Modèle de machine concrète de Von Neumann

Modèle à la base des ordinateurs actuels

Une structure de stockage unique pour conserver à la fois les instructions et les données requises ou générées par le calcul.

De telles machines sont aussi connues sous le nom d'ordinateurs à programme stocké en mémoire.

Description du modèle de Von Neumann (1/3)

Trois entités :

- le processeur
  - **unité de calcul** appelée UAL (Unité Arithmétique et Logique)
  - **unité de contrôle** appelée aussi unité de commande
- la mémoire dite centrale
- le système d'entrées/sorties, liaisons avec l'extérieur (périphériques)
  - écran, clavier, souris
  - mémoire secondaire (disque, CD, DVD, bande, etc)
  - réseau (ethernet, wifi, etc)

Description du modèle de Von Neumann (2/3)

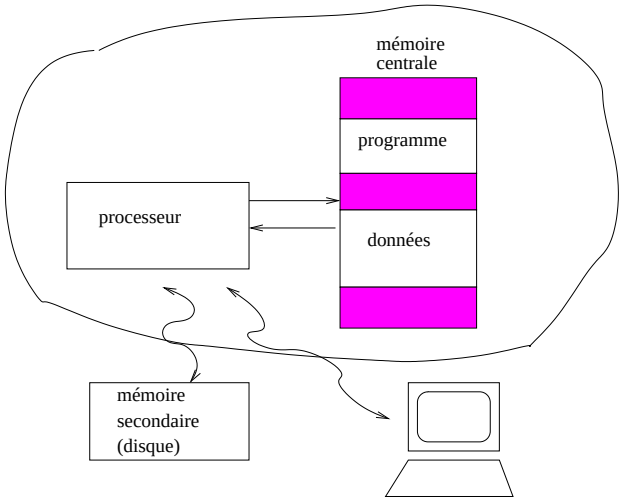


FIGURE – Processeur, mémoire et périphériques

Description du modèle de Von Neumann (3/3)

- le processeur exécute le programme contenu dans la mémoire centrale.
- la mémoire centrale contient à la fois les données et le programme qui "dit" au processeur quels calculs faire sur ces données.
- le système d'entrées-sorties permet de communiquer avec le monde extérieur (réseau, utilisateur, ...) et fait la liaison avec les mémoire secondaires

Remarque

Les programmes traitent des représentations des informations. Les programmes sont aussi des informations.

Mémoire centrale (vision abstraite)

La mémoire contient des informations prises dans un certain domaine

La mémoire contient un certain nombre (fini) d'informations

Les informations sont codées par des vecteurs binaires d'une certaine taille

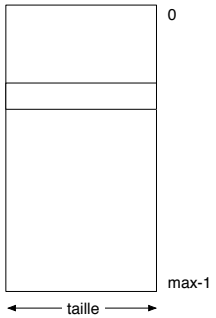


FIGURE – Mémoire abstraite

Mémoire centrale : notion d'adresse

Les informations contenues dans la mémoire sont désignables par une adresse

On parle aussi d'emplacement mémoire
On parle aussi d'emplacement précédent, suivant

Dans l'exemple précédent, une adresse est un numéro d'emplacement parmi [0, max-1].

Mémoire centrale (vision physique)

Toutes les informations sont représentées par des vecteurs binaires ce qui permet des connexions par des nappes de fils.

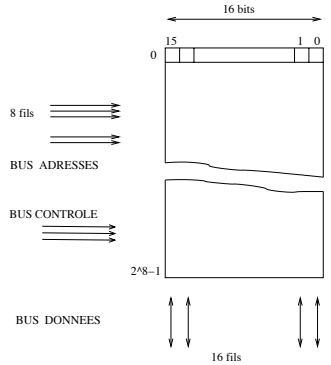


FIGURE – Mémoire physique d'info sur 16 bits avec adresses sur 8 bits

Mémoire centrale : informations de tailles variables

Les informations en mémoire sont :

- de taille 8 bits : octet (byte)
- de taille 16 bits : demi-mot (hword, halfword)
- de taille 32 bits : mot (word)
- de taille 64 bits : double-mot (dword, double word)

Mémoire centrale : vision logique

`Max` informations de 8 bits *i.e.* `Max` octets

`Max / 4` mots de 32 bits

Mémoire centrale : adresses en octets

Les adresses sont exprimées en nombre d'octets.

Par exemple, si on considère des mots de 32 bits (4 octets), si un mot est à l'adresse X, le suivant est à l'adresse X+4

Une adresse est aussi représentée sur un certain nombre de bits.

Par exemple, si les numéros d'emplacement vont de 0 à max-1, l'adresse est codée par un vecteur binaire de $\lceil \log_2 max \rceil$ bits.
Par exemple, une adresse codée sur 32 bits permet de désigner 2^{32} octets différents.

Alignements

Dans la plupart des systèmes, il y a des contraintes d'alignement :

- **Pas de contrainte pour les octets**, cependant
- l'adresse d'un **demi-mot** doit être **pair**,
- l'adresse d'un **mot** doit être multiple de **4** et
- l'adresse d'un **double-mot** doit être multiple de **8**.

En `ARM`, les corrections d'alignement sont effectuées via la directive `.balign`.

Par exemple, `.balign 4` permet de corriger l'alignement pour que la prochaine adresse utilisée soit multiple de 4.

Tailles des mémoires centrales (1/2)

Unités utilisées

- 1 kilooctet = 10^3 octets
- 1 mégaoctet = 10^6 octets
- 1 gigaoctet = 10^9 octets
- 1 téraoctet = 10^{12} octets
- 1 pétaoctet = 10^{15} octets

Tailles des mémoires centrales (2/2)

Ordinateur de poche, PDA (Personal Digital Assistant) : 4 Go

Ultra-portable : 8 à 32 Go

Ordinateur de bureau, station de travail : 16 à 32 Go

Super ordinateur (Sunway TaihuLight - NRCPC) : 1,31 Peta octets

Actions sur la mémoire centrale

Les informations mémorisées dans la mémoire centrale peuvent être :

- consultées (plusieurs fois)
- modifiées

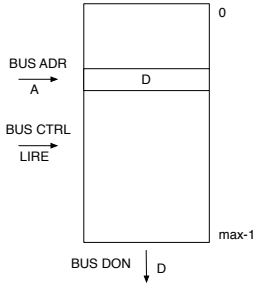
La mémoire centrale contient, non seulement des données mais aussi le programme.

Actions sur la mémoire : LIRE

La mémoire reçoit :

- un vecteur binaire (représentant une adresse A) sur le bus adresses,
- un signal de commande de lecture sur le bus de contrôle.

Elle délivre un vecteur binaire représentant la donnée D sur le bus données.



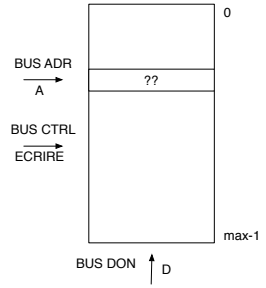
On note : `D <-- mem[A]`
`mem[A]` : emplacement mémoire dont l'adresse est A

Actions sur la mémoire : ECRIRE

La mémoire reçoit :

- un vecteur binaire (représentant une adresse A) sur le bus adresses,
- un vecteur binaire (représentant la donnée D) sur le bus données,
- un signal de commande d'écriture sur le bus de contrôle.

Elle inscrit (*peut-être*, voir tableau ci-après) la donnée D comme contenu de l'emplacement mémoire dont l'adresse est A



On écrit : `mem[A] <-- D`

Remarque : le bus de données est bidirectionnel

Actions sur la mémoire : NE RIEN FAIRE

Bus contrôle : ni lecture, ni écriture.

La mémoire garde indéfiniment (*peut-être*, voir tableau ci-après) les informations.

Différents types de mémoires (centrales)

nom	**volatilité**	**modifiabilité**
ROM mémoire morte	non	inscrite à la fabrication par l'usine de fabrication semi-conducteurs
Eprom (+ variantes)	non	modifiable (100 fois) par l'atelier PMIPME électronique
mémoire flash	non	modifiable 100 000 fois l'utilisateur de l'appareil de carte bancaire, téléphone, vitale
RAM mémoire vive	oui	modifiable autant de fois que le signal écriture sera activé

La mémoire centrale se divise entre mémoire volatile (RAM) (programmes et données en cours de fonctionnement) et mémoire permanente (ROM et EPROM) (programmes et données de base de la machine)

Autres types de mémoires (1/3)

- Les mémoires secondaires désignent les mémoires ≪ périphériques ≫ telles que le disque dur, CD-ROM, DVD, clé USB, ...

Taille des disques (mémoires secondaires) :

Ultra-portable : 200 à 500 Go
Ordinateur de bureau, station de travail : un Tera octets
Super ordinateur (Sunway TaihuLight - NRCPC) : 20 Peta octets

Autres types de mémoires (2/3)

- La mémoire cache est une mémoire relativement petite et rapide qui stocke les informations les plus utilisées d'une autre mémoire plus grande et plus lente.
- La mémoire cache peut être séparée en une partie pour les données et une partie pour les programmes
- La mémoire cache peut différencier plusieurs niveaux (L1, rapide ; L2, un peu moins rapide mais plus grosse)
- Les gains espérés dépendent des principes des localités spatiale et de localité temporelle

Autres types de mémoires (3/3)

- La mémoire virtuelle désigne des mécanismes proposant aux processeurs une vision idéale de la mémoire réelle
- La pagination traduit les adresses mémoires idéales en un couple (numéro de page, adresse dans la page) associé aux processus
- La mémoire secondaire peut être utilisée comme de la mémoire principale (swap)
- Ainsi des programmes peuvent s'exécuter dans un environnement matériel ne possédant pas assez de mémoire centrale

Résumé : processeur/mémoire

Processeur : circuit relié à la mémoire (bus adresses, données et contrôle)

La mémoire contient des informations de nature différentes :

- des données : représentation binaire d'une couleur, d'un entier, d'une date, etc.
- des instructions : représentation binaire d'une ou plusieurs actions à réaliser.

Le processeur, relié à une mémoire, peut :

- **lire** un mot : le processeur fournit une adresse, un signal de commande de lecture et reçoit le mot (une copie).
- **écrire** un mot : le processeur fournit une adresse ET une donnée et un signal de commande d'écriture.
- ne pas accéder à la mémoire.
- **exécuter** des instructions, ces instructions étant des informations lues en mémoire.

Entrées/Sorties : définitions

On appelle périphériques d'entrées/sortie les composants qui permettent :

- L'intéraction de l'ordinateur (mémoire et processeur) avec l'**utilisateur** (clavier, écran, ...)
- L'intéraction de l'ordinateur avec le **réseau** (carte réseau, carte WIFI, ...)
- L'accès aux **mémoires secondaires** (disque dur, clé USB...)

L'accès aux périphériques se fait par le biais de ports (usb, serie, pci, ...).

Sortie : ordinateur $\longrightarrow$ extérieur (*e.g.*, écran)

Entrée : extérieur $\longrightarrow$ ordinateur (*e.g.*, clavier)

Entrée/Sortie : ordinateur $\longleftrightarrow$ extérieur (*e.g.*, disque dur)

Les bus

Un bus informatique désigne l'ensemble des lignes de communication (câbles, pistes de circuits imprimés, ...) connectant les différents composants d'un ordinateur.

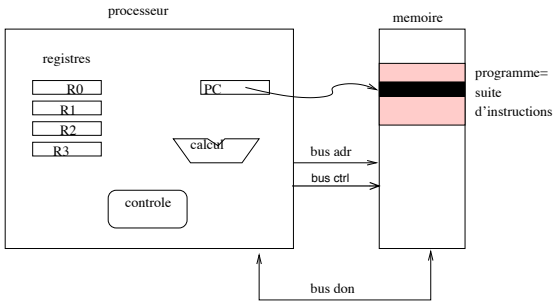
- Le bus de données permet la circulation des données.
- Le bus d'adresse permet au processeur de **désigner à chaque instant la case mémoire ou le périphérique** auquel il veut faire appel.
- Le bus de contrôle indique quelle est l'**opération que le processeur veut exécuter**, par exemple, s'il veut faire une écriture ou une lecture dans une case mémoire.

  On trouve également, dans le bus de contrôle, une ou plusieurs lignes qui permettent aux périphériques d'effectuer des demandes au processeur ; ces lignes sont appelées lignes d'interruptions matérielles (IRQ).

Composition du processeur

Le processeur est composé d'unités (ressources matérielles internes) :

- des registres : cases de mémoire interne
  Caractéristiques : désignation, lecture et écriture "simultanées"
- des unités de calcul (UAL)
- une unité de contrôle : (UC, *Central Processing Unit*)
- un compteur ordinal ou compteur programme : `PC`



Notion d'exécution séquentielle des instructions(1/3)

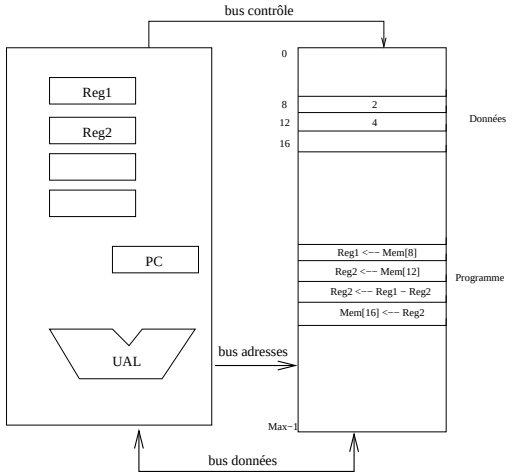


FIGURE – Un exemple d'exécution d'un programme

Notion d'exécution séquentielle des instructions (2/3)

Pour exécuter une suite d'instructions (un programme), le processeur applique l'algorithme suivant.

Répéter :

- **lire** une instruction à l'adresse contenue dans PC
- **exécuter** cette instruction
- **calculer** l'adresse de l'instruction suivante et mettre à jour PC

- **démarrage :** signal physique (RESET) → forcer une valeur initiale dans PC
- **arrêt :** jamais (sauf coupure de courant...).

Remarques : Dans la plupart des machines (machines portables, embarquées, ...) le fonctionnement peut être suspendu pour économiser de l'énergie

Notion d'exécution séquentielle des instructions (3/3)

- les instructions sont normalement exécutées dans l'ordre où elles sont écrites en mémoire

  ⟶ exécution séquentielle

- certaines instructions spécifiques donnent l'adresse de l'instruction suivante

  ⟶ rupture de séquence

Codage des instructions : langage machine

- Représentation d'une instruction en mémoire : un vecteur de bits
- Programme : suite de vecteurs binaires qui codent les instructions qui doivent être exécutées.
- Le codage des instructions constitue le Langage machine (ou *code machine*).
- Chaque modèle de processeur a son propre langage machine (on dit que le langage machine est natif)

Architectures Logicielles et Matérielles

- Architecture logicielle (ISA, *Instruction Set Architecture*)

  ⟶ organisation du langage machine et son utilisation

- Architecture matérielle

  ⟶ comment le langage machine est exécuté par du matériel

Informatique omniprésente (Ubiquitous Computing)

- Téléphone portable
- PDA
- Console de jeux
- Lecteur de cartes magnétiques
- Capteur (sensor)
- Puce RFID
- Voiture « intelligente »
- ...

Ordinateurs personnels

- Multi-coeurs (voir Multi-processeurs)
- Puissance : plusieurs gigahertz
- Mémoire centrale : plusieurs Gigaoctets
- Disque dur : plusieurs centaines de Gigaoctets
- **Sur les PC** : des pentium, AMD, ou des compatibles.
- **Sur les Macintosh** : des 68000, PowerPC, et maintenant des pentiums
- **Sur certaines consoles de jeux** : des ARM
- **Sur iceberg** : biprocesseur type Intel(R) Xeon(R)
- **Sur les PC des salles de TP** : processeur Intel Pentium

Supercalculateurs

Classement (20 juin 2016)

1. Sunway TaihuLight (NRCPC, Chine)
2. Tianhe-2 (NUDT, Chine)
3. Titan (CRAY, USA)

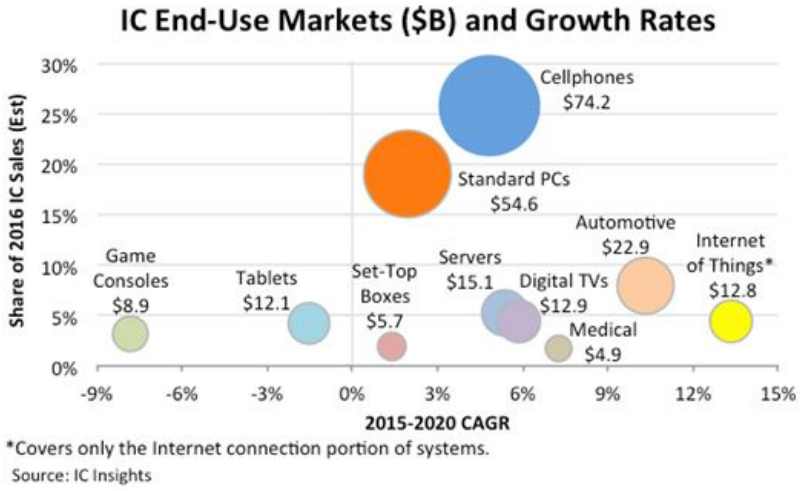
Caractéristiques du Sunway TaihuLight :

- Performance maximale : 125 pétaflops (125×10^{15} opérations à virgule flottante/seconde)
- 1,31 Peta-Octets de mémoire vive ($1{,}31 \times 10^{15}$ octets)
- 40 960 processeurs (SW26010 manycore 64-bit RISC processors avec 260 coeurs chacun, 1,45 GHz chacun)
- Puissance : 15,371 Méga Watt
- Budget final de 273 millions de dollars

Remarque

La tendance actuelle est de remplacer les supercalculateurs par des grappes de pc (cluster, grid-computing)

Marché des circuits intégrés



Langage d'assemblage, langage machine

Exécution des programmes en langage machine

Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

15 janvier 2020

Plan

Exemple

monprog.c :

```
/* monprog.c */
#include "malib.h"

#define max(a,b) ((a)>(b)?(a):(b))

main()
{
  int x,y,z;
  x = 10;
  y = 15;
  z = max(x,y);
  plus (&z);
}
```

malib.c :

```
void plus (int *ai) {
        *ai = (*ai) + 1;
      }
```

malib.h :

```
#ifndef MALIB

#define MALIB

/* incrementation d'un mot memoire */
extern void plus (int *);
#endif
```


Etapes de compilation

- **Précompilation :**
  `arm-eabi-gcc -E monprog.c > monprog.i`
  source : `monprog.c` → source ≪ enrichi ≫ `monprog.i`
- **Compilation :** `arm-eabi-gcc -S monprog.i`
  source ≪ enrichi ≫ → langage d'assemblage : `monprog.s`
- **Assemblage :** `arm-eabi-gcc -c monprog.s`
  langage d'assemblage → binaire translatable : `monprog.o` (fichier objet)
  même processus pour `malib.c` → `malib.o`
- **Edition de liens :** `arm-eabi-gcc monprog.o malib.o -o monprog`
  un ou plusieurs fichiers objets → binaire exécutable : `monprog`

Précompilation (*pre-processing*)

arm-eabi-gcc -E monprog.c > monprog.i

produit **monprog.i**

La précompilation réalise plusieurs opérations de substitution sur le code, notamment :

- suppression des commentaires.
- inclusion des profils des fonctions des bibliothèques dans le fichier source.
- traitement des directives de compilation.
- remplacement des macros

Exemple : arm-eabi-gcc -E monprog.c > monprog.i (2/2)

monprog.i :

```
# 1 "monprog.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "monprog.c"

# 1 "malib.h" 1

extern void plus (int *);
# 3 "monprog.c" 2

main()
{
  int x,y,z;
  x = 10;
  y = 15;
  z = ((x)>(y)?(x):(y));
  plus (&z);
}
```


Compilation

arm-eabi-gcc -S monprog.i

produit **monprog.s**

Le code source ≪ enrichi ≫ est transformé en langage d'assemblage

instructions et données.

Exemple : arl-eabi-gcc -S monprog.i

monprog.s :

```
.file "monprog.c"
.text
.align 2
.global main
.type main, %function
main:
@ args = 0, pretend = 0, frame = 12
@ frame_needed = 1, uses_..._args = 0
mov ip, sp
stmfd sp!, {fp, ip, lr, pc}
sub fp, ip, #4
sub sp, sp, #12
mov r3, #10
str r3, [fp, #-16]
mov r3, #15
str r3, [fp, #-20]
ldr r2, [fp, #-20]
ldr r3, [fp, #-16]
cmp r2, r3
movge r3, r2
str r3, [fp, #-24]
sub r3, fp, #24
mov r0, r3
bl plus
mov r0, r3
sub sp, fp, #12
ldmfd sp, {fp, sp, pc}
.size main, .-main
.ident "GCC: (GNU) 3.4.3"
```

Assemblage

arm-eabi-gcc -c monprog.s

produit `monprog.o`

Le code en langage d'assemblage (lisible) est transformé en code machine.

Le code machine se présente comme une succession de vecteurs binaires.

Le code machine ne peut pas être directement édité et lu. On peut le rendre lisible en utilisant une commande *od -x monprog.o*.

Le fichier `monprog.o` contient des instructions en langage machine et des données mais il n'est pas exécutable. On parle de binaire translatable.

Exemple : arm-eabi-gcc -c monprog.s

monprog.o (extrait) :

```
0000000 457f 464c 0101 6101 0000 0000 0000 0000
0000020 0001 0028 0001 0000 0000 0000 0000 0000
0000040 00cc 0000 0000 0000 0034 0000 0000 0028
0000060 0009 0006 c00d e1a0 d800 e92d b004 e24c
0000100 d00c e24d 300a e3a0 3010 e50b 300f e3a0
0000120 3014 e50b 2014 e51b 3010 e51b 0003 e152
0000140 3002 a1a0 3018 e50b 3018 e24b 0003 e1a0
0000160 fffe ebff 0003 e1a0 d00c e24b a800 e89d
0000200 4700 4343 203a 4728 554e 2029 2e33 2e34
0000220 0033 2e00 7973 746d 6261 2e00 7473 7472
0000240 6261 2e00 6873 7473 7472 6261 2e00 6572
0000260 2e6c 6574 7478 2e00 6164 6174 2e00 7362
0000300 0073 632e 6d6f 656d 746e 0000 0000 0000
0000320 0000 0000 0000 0000 0000 0000 0000 0000
*
0000360 0000 0000 001f 0000 0001 0000 0006 0000
0000400 0000 0000 0034 0000 004c 0000 0000 0000
0000420 0000 0000 0004 0000 0000 0000 001b 0000
0000440 0009 0000 0000 0000 0000 0000 02dc 0000
0000460 0008 0000 0007 0000 0001 0000 0004 0000
0000500 0008 0000 0025 0000 0001 0000 0003 0000
0000520 0000 0000 0080 0000 0000 0000 0000 0000
0000540 0000 0000 0001 0000 0000 0000 002b 0000
0000560 0008 0000 0003 0000 0000 0000 0080 0000
0000600 0000 0000 0000 0000 0000 0000 0001 0000
0000620 0000 0000 0030 0000 0001 0000 0000 0000
0000640 0000 0000 0080 0000 0012 0000 0000 0000
```


Edition de liens

arm-eabi-gcc monprog.o malib.o -o monprog

produit **monprog**

L'édition de liens permet de rassembler le code de différents fichiers.

A l'issue de cette phase le fichier produit contient du **binaire exécutable**.

remarque : ne pas confondre exécutable, lié à la nature du fichier, et « muni du droit d'être exécuté », lié au système d'exploitation.

Exemple : arm-eabi-gcc monprog.o malib.o -o monprog

monprog (extrait) :

```
0000000 457f 464c 0101 6101 0000 0000 0000 0000
0000020 0002 0028 0001 0000 8110 0000 0034 0000
0000040 d010 0001 0002 0000 0034 0020 0001 0028
0000060 0018 0015 0001 0000 8000 0000 8000 0000
0000100 8000 0000 2a3c 0000 2b4c 0000 0007 0000
0000120 8000 0000 0000 0000 0000 0000 0000 0000
0000140 0000 0000 0000 0000 0000 0000 0000 0000
*
0100000 c00d e1a0 dff0 e92d b004 e24c 0024 eb00
0100020 07fd eb00 6ff0 e91b f00e e1a0 c00d e1a0
0100040 d830 e92d 5058 e59f 3000 e5d5 0000 e353
0100060 b004 e24c a830 189d 4048 e59f 0004 ea00
0100100 3000 e594 3004 e283 3000 e584 e00f e1a0
0100120 f002 e1a0 3000 e594 2000 e593 0000 e352
0100140 fff6 1aff 3020 e59f 0000 e353 001c 159f
0100160 e00f 11a0 f003 11a0 3001 e3a0 3000 e5c5
0100200 a830 e89d aa3c 0000 a180 0000 0000 0000
0100220 aa24 0000 c00d e1a0 d800 e92d b004 e24c
0100240 a800 e89d 3040 e59f c00d e1a0 0000 e353
0100260 d800 e92d 0034 e59f b004 e24c 1030 e59f
0100300 e00f 11a0 f003 11a0 0028 e59f 3000 e590
0100320 0000 e353 a800 089d 301c e59f 0000 e353
0100340 a800 089d 6800 e89d dfc4 eaff 0000 0000
0100360 aa24 0000 aa40 0000 aa38 0000 0000 0000
0100400 c00d e1a0 d800 e92d b004 e24c a800 e89d
0100420 0016 e3a0 10e4 e28f 3456 ef12 00dc e59f
```

Comment exécuter un programme ? (1/3)

Sur un ordinateur dont le binaire est connu du processeur :

Ex : le fichier `monprog` contient du ≪ binaire Intel ≫ et je suis sur un PC dont la carte mère est équipée d'un processeur Intel.

- je lance le programme depuis un terminal ou l'interface graphique,
- le système d'exploitation installe le contenu du fichier exécutable dans la mémoire centrale (transfert depuis le disque dur + chargement en mémoire),
- puis il met l'adresse de début dans le compteur programme,
- et enfin le processeur commence l'exécution. . .

Remarque : l'emplacement de stockage du programme en mémoire est choisi par le système d'exploitation.

Comment exécuter un programme ? (2/3)

Sur un ordinateur qui ne connait pas le binaire que l'on produit

Ex : **arm-eabi-gcc monprog.c** produit du binaire **arm** qui n'est pas exécutable sur un PC Intel.

On utilise un émulateur, c'est-à-dire un programme qui mimique le comportement du système pour lequel le binaire est destiné (**arm-eabi-run monprog**).

L'émulateur simule l'exécution du binaire dans une mémoire qui lui est propre.

Cas particulier : Java

Comment exécuter un programme ? (3/3)

Sur les automatismes : ex. lecteur de carte électronique

Quand il écrit le programme, le programmeur connait précisément l'architecture. Il peut produire directement une image du programme dans la mémoire à une adresse fixe et ranger le binaire dans une **EPROM**.

Caractéristiques d'un langage

- un lexique : les mots que l'on peut écrire,
- une syntaxe : l'organisation des mots qui est permise,
- une sémantique : le sens des phrases, c'est-à-dire des successions de mots.

Classification

- Langage de haut-niveau :

  le programmeur peut **s'abstraire** des détails de fonctionnement de la machine, il peut manipuler des concepts élaborés (objet, structure...).
  Exemples : Java, Python, ADA, LISP...
- Langage de bas-niveau :

  le programmeur est obligé de prendre en compte des concepts **proches du fonctionnement de la machine** (adresse de l'instruction courante, place des valeurs en mémoire, ...).

Langages bas-niveaux

- Langage machine : suite de bits, interprétée par le processeur de l'ordinateur lors de l'exécution d'un programme.
  Remarque : Aujourd'hui, on ne programme plus directement en binaire mais c'était le cas pour les premiers ordinateurs !
- Langage d'assemblage (assembleur) : comme le langage machine mais sous une forme lisible par un humain : les combinaisons de bits du langage machine sont représentées par des symboles dits « mnémoniques ».
  Remarque : Il est la plupart du temps utilisé comme intermédiaire (lors de la compilation) entre un langage de haut-niveau et le langage machine.

Lexicographie

Lexique : liste des mots du langage, la façon dont ils s'écrivent (leur codage)

Pour le langage machine, les mots sont des suites de 0 et de 1 qui représentent des instructions ou des données. Les suites de 0 et de 1 sont **organisées par champs** pour former les **instructions**.

Exemple : l'instruction 36842_{10} dit qu'il faut ajouter le nombre actuellement dans le registre 17 avec celui qui est dans le registre AB et ranger le résultat en mémoire à l'adresse 27. Les informations 17, AB, 27 seront codées dans des champs différents du vecteur de bits représentant l'instruction.

Syntaxe

Syntaxe : Règles définissant la manière d'écrire et de placer les mots. Quels sont les groupements valides d'instructions (pour constituer des programmes) ?

En général, très souple : Il existe plusieurs façon de coder la même chose.

Sémantique (signification)

Quel est le sens du programme ?

Un programme est une suite (ou séquence, liste) de vecteurs de bits qui représentent des instructions.

Il y a un ordre.

Les instructions sont exécutées dans l'ordre où elles sont écrites.

On verra qu'il peut y avoir des ruptures de séquence.

Remarque : si une donnée se retrouve, suite à une erreur, repérée par le compteur programme, alors le processeur essaie de l'exécuter.

Instructions en langage machine

2 types d'instructions :

- Instructions de calcul (ou de déplacement d'information) entre des informations mémorisées.
- Instructions de rupture de séquence

Un langage machine par processeur

Pour tout processeur il existe un langage machine caractéristique (langage natif). Même s'il existe une compatibilité entre processeurs de la même famille.

Remarques :

- Si il y a 237 instructions dans le langage machine d'un certain processeur, elles peuvent être codées par un vecteur binaire de 8 bits ($2^7 = 128$; $2^8 = 256$) et donc il y a 19 vecteurs invalides.

  Que doit alors faire le processeur ?

  le processeur génère une exception (interruption particulière) qui mène généralement à l'arrêt du programme.
- Une instruction est une chaîne de bits, une donnée aussi . . .
- Lorsque l'on considère une chaîne de bits, on ne sait pas s'il s'agit d'une instruction ou d'une donnée. le code n'est pas dans l'information codée.

Instruction de calcul entre des informations mémorisées

L'instruction désigne la(les) source(s) et le destinataire. Les *sources* sont des cases mémoires, registres ou des valeurs. Le *destinataire* est un élément de mémorisation.

L'instruction code : destinataire, source1, source2 et l'opération.

désignation du destinataire	←	désignation de source1	oper	désignation de source2
mém, reg		mém, reg		mém, reg, valIMM

mém signifie que l'instruction fait référence à un mot dans la mémoire

reg signifie que l'instruction fait référence à un registre (nom ou numéro)

valIMM signifie que l'information source est contenue dans l'instruction

Exemples

- reg12 ← reg14 + reg1
- registre4 ← le mot mémoire d'adresse 36000 + le registre A
- reg5 ← reg5 - 1
- le mot mémoire d'adresse 564 ← registre7

Convention de noms

move, load, store

Instruction de rupture de séquence

- Fonctionnement standard : Une instruction est écrite à l'adresse X ; l'instruction suivante (dans le temps) est l'instruction écrite à l'adresse X+t (où t est la taille de l'instruction). C'est implicite pour toutes les instructions de calcul.
- Rupture de séquence : Une instruction de *rupture de séquence* peut désigner la prochaine instruction à exécuter (à une autre adresse).

Exemples

- `Branch 125` : l'instruction suivante est désignée par une adresse ≪ fixe ≫.
- `Branch -40` : l'instruction suivante est une adresse calculée.
- `Branch SiZero +10` : si le résultat du calcul précédent est ZERO, alors la prochaine instruction à exécuter est celle d'adresse ≪ adresse courante+10 ≫, sinon la prochaine instruction à exécuter est la suivante dans l'ordre d'écriture, c'est-à-dire à l'adresse ≪ adresse courante ≫ +t.

Utilisation des instructions de rupture de séquence

Les ruptures de séquence permettent de modifier le déroulement séquentiel de l'exécution des instructions.

Les ruptures de séquence servent à coder des choix ou des boucles.

Remarques : si l'instruction suivante d'une instruction est elle-même alors le processeur part en boucle ≪ infinie ≫.

Convention de noms

b, beq, bne, bgt, bge, blt, ble

Caractéristiques et Avantages

Caractéristiques :

- Langage textuel (au lieu de binaire) mais correspondance : une instruction en assembleur ↔ une instruction en langage machine.
- Séparation données/instructions.
- Pas nécessaire de manipuler les adresses réelles, on ne connait pas les adresses des données du programme !

Avantages :

- Lisibilité du code
- Niveau de langage plus facile à manipuler

Exemples

En ARM :

- add r4, r5, r6 signifie r4 ← r5 + r6.
  r5 désigne le contenu du registre, on parle bien sûr du contenu des registres, on n'ajoute pas des ressources physiques !

En SPARC :

- add g4, g5, g6 signifie g6 ← g4 + g5.

En 6800 (Motorola) :

- addA 5000 signifie regA ← regA + Mem[5000]
- addA ♯50 signifie regA ← regA + 50
- add r3, r3, [5000] signifie reg3 ← reg3 + Mem[5000]

Remarque : pas de règle générale, interprétations différentes selon les fabricants, quelques habitudes cependant concernant les mnémoniques (add, sub, load, store, jump, branch, clear, inc, dec) ou la notation des opérandes (#, `[xxx]`)

Langage textuel, notation des instructions

Une instruction =

- mot conventionnel = mnémonique
- désignation des instructions ou données manipulées.

A chaque notation est associée une convention d'interprétation.

Désignation des objets (1/7)

On parle parfois, improprement, de modes d'adressage. Il s'agit de dire comment on écrit, par exemple, la valeur contenue dans le registre numéro 5, la valeur *-8*, la valeur rangée dans la mémoire à l'adresse 0xff, . . .

Il n'y a pas de standard de notations, mais des standards de signification d'un constructeur à l'autre.

L'**objet** désigné peut être **une instruction** ou **une donnée**.

Désignation des objets (2/7) : par registre

Désignation registre/registre.

L'objet désigné (une donnée) est le contenu d'un registre. L'instruction contient le nom ou le numéro du registre.

- **En 6502 (MOS Technology) :** 2 registres A et X (entre autres)
  TAX signifie transfert de A dans X
  X ← contenu de A (on écrira X ← A).
- **ARM :** mov r4 , r5 signifie r4 ← r5.

Désignation des objets (3/7) : immédiate

Désignation registre/valeur immédiate.

La donnée dont on parle est littéralement écrite dans l'instruction

- **En ARM :** mov r4 , #5 ; signifie r4 ← 5.

Remarque : la valeur immédiate qui peut être codée dépend de la place disponible dans le codage de l'instruction.

Désignation des objets (4/7) : directe ou absolue

Désignations registre/directe ou absolue.

On donne dans l'instruction l'adresse de l'objet désigné. L'objet désigné peut être une instruction ou une donnée.

- **En 68000 (Motorola) :** move.l D3, $FF9002 signifie
  Mem[FF9002] ← D3.
  la deuxième opérande (ici une donnée) est désigné par son adresse en mémoire.
- **En SPARC :** jump 0x2000 signifie l'instruction suivante (qui est l'instruction que l'on veut désigner) est celle d'adresse 0x2000.

Désignation des objets (5/7) : indirect par registre

Désignation registre/indirect par registre

L'objet désigné est dans une case mémoire dont l'adresse est dans un registre précisé dans l'instruction.

- ldr r3, [r5] signifie r3 ← (le mot mémoire dont l'adresse est contenue dans le registre 5)
  On note r3 ← mem[r5].

Désignation des objets (6/7) : indirect par registre et déplacement

Désignation registre/indirect par registre et déplacement

L'adresse de l'objet désigné est obtenue en ajoutant le contenu d'un registre précisé dans l'instruction et d'une valeur (ou d'un autre registre) précisé aussi dans l'instruction.

- ldr r3, [r5, #4] signifie r3 ← mem[r5 + 4].
  La notation [r5, #4] désigne le mot mémoire (une donnée ici) d'adresse r5 + #4.
- **En 6800 :** jump [PC - 12] = le registre est PC, le déplacement -12.
  L'instruction suivante (qui est l'instruction que l'on veut désigner) est celle à l'adresse obtenue en calculant, au moment de l'exécution, PC - 12.

Séparation données/instructions

Le texte du programme est organisé en zones (ou segments) :

- zone TEXT : code, programme, instructions
- zone DATA : données initialisées
- zone BSS : données non initialisées par le programmeur, réservation de place en mémoire (valeur quelconque ou 0, ça dépend de l'architecture)

On peut préciser où chaque zone doit être placée en mémoire : la directive ORG permet de donner l'adresse de début de la zone (ne fonctionne pas toujours !).

Désignation des objets (7/7) : relatif au compteur programme

Désignation relative au compteur programme

L'adresse de l'objet désigné (en général une instruction) est obtenue en ajoutant le contenu du compteur de programme et une valeur précisée aussi dans l'instruction.

En ARM : b 20 signifie pc ← pc + 20

Exemple (1/2)

Dans l'exemple suivant on utilise la directive **org**. On suppose que les instructions sont toutes codées sur **4 octets** :

```
zone TEXT, org 0x2000
move r4, #5004
load r5, [r4]
jump 0x2000
.......
zone DATA,  org 0x5000
entier sur 4 octets : 0x56F3A5E2
entier sur 4 octets : 0xAAF43210
....
```

Exemple (2/2)

Quelle est la représentation en mémoire du programme ?

```
              memoire
           |             |
           |             |
   2000 --> | move...     |
   2004 --> | load...     |
   2008 --> | jump...     |
           |             |
           |             |
           |             |
   5000 --> |56 f3 a5 e2  |
           |aa f4 32 10  |
           |             |
           |             |
```

Quel est l'effet de ce programme ?

Etiquettes (1/4) : définition

Etiquette : nom choisi librement (quelques règles lexicales quand même) qui désigne une case mémoire. Cette case peut contenir une donnée ou une instruction.

Une étiquette correspond à une adresse.

Pourquoi ?

- L'emplacement des programmes et des données n'est à priori pas connu
  la directive ORG ne peut pas toujours être utilisée
- Plus facile à manipuler

Etiquettes (2/4) : exemple

```
zone TEXT
DD: move r4, #42
    load r5, [YY]
    jump DD

zone DATA
XX: entier sur 4 octets : 0x56F3A5E2
YY: entier sur 4 octets : 0xAAF43210
```


Etiquettes (3/4) : avantages

- Le programmeur n'a pas besoin de connaitre les adresses réelles pour comprendre ce que fait son programme.
- Le programme est plus facile à lire, à écrire.

Etiquettes (4/4) : correspondance étiquette/adresse

Supposons les adresses de début des zones TEXT et DATA respectivement 2000 et 5000
Il faut remplacer DD par 2000 et YY par 5004.

```
zone TEXT                        contenu de Mem[2000], ...
DD: move r4, #42                      move r4, #42
    load r5, [YY]                     load r5, [5004]
    jump DD                           jump 2000

zone DATA
XX: entier sur 4 octets : 0x56F3A5E2
YY: entier sur 4 octets : 0xAAF43210
```


Chargement

Correspondance étiquette $\leftrightarrow$ adresse

- Statiquement, dans un fichier si l'adresse de début de chaque zone est connue de l'outil d'assemblage (ou d'édition de liens) $\rightarrow$ de plus en plus rare
- Dynamiquement, juste avant l'exécution ; l'adresse de début est déterminée par le système, le programmeur ne la connaît pas

La phase de chargement consiste à installer un programme en mémoire.
L'adresse de début de la zone mémoire dans laquelle est installé le programme est appelée adresse de chargement.

Étiquette, application en ARM (lecture et écriture d'une variable

```
.data
        A :     .word 10
        B :     .word 15
.text
        .global main
        main :
                ldr r0,ptr_A    @ charge le contenu de l'adresse ptr_A (adresse de A) dans r0
                ldr r1,[r0]     @ charge le contenu de l'adresse A (10) dans r1
                ldr r0,ptr_B    @ charge le contenu de l'adresse ptr_B (adresse de B) dans r0
                str r1,[r0]     @ copie le contenu de r1 (10) à l'adresse B
                b exit
        ptr_A : .word A         @ adresse de la variable A
        ptr_B : .word B         @ adresse de la variable B
```


Autre exemple d'application : `es.s`

Nous considérons le schéma de programme suivant :

```
.data
        X :     .byte 3
                .byte 2
                .hword 1
.text
        .global main
        main :

        ...

        b exit
ptrX :  .word X
```

Autre exemple d'application : `es.s`

Qu'est-ce qu'affiche ce programme ?

```
ldr r0,ptrX
ldr r1,[r0]
bl EcrNdecimal32
```


Autre exemple d'application : `es.s`

Qu'est-ce qu'affiche ce programme ?

```
ldr r0,ptrX
ldrh r1,[r0]
bl EcrNdecimal32
```


Autre exemple d'application : `es.s`

Qu'est-ce qu'affiche ce programme ?

```
ldr r0,ptrX
ldrb r1,[r0]
bl EcrNdecimal32
```


Autre exemple d'application : `es.s`

Qu'est-ce qu'affiche ce programme ?

```
ldr r0,ptrX
ldrb r1,[r0,#1]
bl EcrNdecimal32
```

Autre exemple d'application : es.s

Qu'est-ce qu'affiche ce programme ?

```
ldr r0,ptrX
ldrb r1,[r0,#2]
bl EcrNdecimal32
```


Autre exemple d'application : es.s

Qu'est-ce qu'affiche ce programme ?

```
ldr r0,ptrX
mov r1,#2
strb r1,[r0]
ldr r1,[r0]
bl EcrNdecimal32
```


Autre exemple d'application : es.s

Qu'est-ce qu'affiche ce programme ?

```
ldr r0,ptrX
mov r1,#2
strh r1,[r0]
ldr r1,[r0]
bl EcrNdecimal32
```


Autre exemple d'application : es.s

Qu'est-ce qu'affiche ce programme ?

```
ldr r0,ptrX
mov r1,#2
str r1,[r0]
ldr r1,[r0]
bl EcrNdecimal32
```

Autre exemple d'application : `es.s`

En supposant que l'on donne 1 en entrée, qu'est-ce qu'affiche ce programme ?

```
ldr r1,ptrX
bl LireCar
ldrb r2,[r1]
mov r1,r2
bl EcrNdecimal8
```


Autre exemple d'application : `es.s`

En supposant que l'on donne 1 en entrée, qu'est-ce qu'affiche ce programme ?

```
ldr r1,ptrX
bl Lire8
ldrb r2,[r1]
mov r1,r2
bl EcrNdecimal8
```


Nombres relatives

Pour les nombres relatives :

- Utilisez `ldrsb` au lieu de `ldrb`
- Utilisez `ldrsh` au lieu de `ldrh`

Programmation des structures de contrôles

Exécution des programmes en langage machine

Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

15 janvier 2020

Plan

Exécution séquentielle *vs.* rupture de séquence : rôle du *PC*

registre *PC* : Compteur de programme, repère l'instruction à exécuter

A chaque cycle :

1. *bus d'adresse* $\leftarrow$ *PC* ; *bus de contrôle* $\leftarrow$ *lecture*
2. *bus de donnée* $\leftarrow$ Mem[*PC*] $=$ *instruction courante*
3. Décodage et exécution
4. Mise à jour de *PC* (par défaut, incrémentation)

Les instructions sont exécutées séquentiellement
sauf ruptures de séquence !

Différents types de séquencement

- initialisation ou lancement d'un programme
- séquencement ≪ normal ≫
- rupture de séquence inconditionnelle
- rupture de séquence conditionnelle
- appels et retours de procédure/fonction
- interruptions
- exécution ≪ parallèle ≫

Séquencement (1/7)

Initialisation ou lancement d'un programme

- Initialisation (*boot*)
  PC forcé à une valeur pré-établie (0)
- Lancement d'un programme (chargement)
  PC forcé à l'adresse de début de programme

Séquencement (2/7)

Séquencement ≪ normal ≫

Après chaque instruction le registre *PC* est incrémenté.

Si l'instruction est codée sur *k* octets : $PC \leftarrow PC + k$

Cela dépend des processeurs, des instructions et de la taille des mots.

- En **ARM**, toutes les instructions sont codées sur 4 octets. Les adresses sont des adresses d'octets. PC progresse de 4 en 4
- Sur certaines machines (ex. Intel), les instructions sont de longueur variable (1, 2 ou 3 octets). PC prend successivement les adresses des différents octets de l'instruction

Séquencement (3/7)

Rupture inconditionnelle

Une instruction de branchement inconditionnel force une adresse *adr* dans *PC*.

La prochaine instruction exécutée est celle située en Mem[*adr*]

Cas TRES particulier : **les premiers RISC (Sparc, MIPS)** exécutaient quand même l'instruction qui suivait le branchement.

Séquencement (4/7)

Rupture conditionnelle

Si une condition est vérifiée, **alors**
    PC est modifié
sinon
    PC est incrémenté normalement.

la condition est interne au processeur :
expression booléenne portant sur les *codes de conditions arithmétiques*

- *Z* : nullité,
- *N* : bit de signe,
- *C* : débordement (naturel) et
- *V* : débordement (relatif).

Séquencement (5/7)

Appels ou retours de procédures/fonctions

Il y aura un chapitre dédié au codage des procédures et fonctions

Séquencement (6/7)

Interruption

Mécanisme essentiel permettant entre autres de gérer les périphériques . . .

Si un certain signal physique externe est actif, PC prend une valeur lue en mémoire à une adresse conventionnelle pour exécuter une routine de gestion d'interruption (*interrupt handler* en anglais).

Séquencement (7/7)

Exécution ≪ parallèle ≫

Par ex. une machine superscalaire exécute plusieurs instructions ≪ en même temps ≫

D'autres types de séquences seront étudiés en cours d'architecture avancée

Désignation de l'instruction suivante

- Désignation directe : l'adresse de l'instruction suivante est donnée dans l'instruction.
- Désignation relative : l'adresse de l'instruction suivante est obtenue en ajoutant un certain déplacement (peut être signé) au compteur programme.

Remarques :

- le mode de désignation en **ARM** est uniquement **relatif**.
- en général, le déplacement est ajouté à l'adresse de l'instruction qui suit la rupture. C'est-à-dire, $PC + 4 +$ déplacement.
  En ARM, $PC + 8 +$ déplacement.

Exemple : Assembleur type ARM

Rappel : instructions codées sur 4 octets

etiquette	adresse correspondant a l'etiquette	instruction assembleur	codage effectif de l'instruction
	0x1028	CLR Reg6	0x********
e1 :	0x102C	BRANCH si cond e2	
	0x1030	xxx	xxx
	0x1034	xxx	xxx
	0x1038	xxx	xxx
e2 :	0x103C	yyy	0x********
	0x1040	xxx	xxx
	0x1044	xxx	xxx

Utilisation typique des sauts ou branchements

Programmation systématique en language d'assemblage de 3 types de constructions algorithmiques :

- Structures de contrôle de la programmation classique : *si . . . alors . . . sinon . . .*, *tant que . . . faire . . .*
- Automates avec actions (ou machines séquentielles avec actions)
- Procédures et fonctions

Codage des structures de contrôle : exemples traités

- `I1;` **si** `ExpCondSimple` **alors** `{I2; I3; I4;} I5;`
- `I1;` **si** `ExpCondSimple` **alors** `{I2; I3;}` **sinon** `{I4; I5; I6;} I7;`
- `I1;` **tant que** `ExpCond` **faire** `{I2; I3;} I4;`
- `I1;` **répéter** `{I2; I3;}` **jusqu'à** `ExpCond; I4;`
- `I1;` **pour** `(i←0 à N) {I2; I3; I4;} I5;`
- **si** `C1` **ou** `C2` **ou** `C3` **alors** `{I1;I2;}` **sinon** `{I3;}`
- **si** `C1` **et** `C2` **et** `C3` **alors** `{I1;I2;}` **sinon** `{I3;}`
- **selon** `a,b`

```
a<b : I1;
a=b : I2;
a>b : I3;
```


Codage des structures de contrôle : notations

On dispose de sauts et de sauts conditionnels notés :

- branch etiquette et
- branch si cond etiquette.

cond est une expression booléenne portant sur Z, N, C, V

ATTENTION : les conditions dépendent du **type**. Par exemple, la condition $<$ à utiliser est différente selon qu'un entier est un naturel ou un relatif (l'interprétation du bit de poids fort est différente !).

Toute autre instruction (affectation, addition, . . .) est notée `Ik`

Instruction *Si* ≪ *simple* ≫

`I1; si a=b alors {I2; I3; I4;} I5;`

a et b deux entiers dont les valeurs sont rangées respectivement dans les registres r1 et r2.

Une première solution

`I1; si a=b alors {I2; I3; I4;} I5;`

```
            I1
            calcul de a-b + positionnement de ZNCV
            branch si (egal a 0) a etiq_alors
            branch a etiq_suite
etiq_alors: I2
            I3
            I4
etiq_suite: I5
```


Une autre solution

`I1; si a=b alors {I2; I3; I4;} I5;`

```
            I1
            calcul de a-b + positionnement de ZNCV
            branch si (non egal a 0) a etiq_suite
            I2
            I3
            I4
etiq_suite: I5
```


Codage en ARM

`x←0;a←5;b←6; si a=b alors {x←1;} x←x+1;`

a et b dans r0, r2, x dans r1

Remarque : `égal à 0` équivalent à Z

Codage en ARM

`x←0;a←0;b←5; si a=b alors {x←1;} x←x+1;`

a et b dans r0, r2, x dans r1

Remarque : `non égal à 0` équivalent à $\overline{Z}$

Instruction *Si alors sinon*

```
I1; si ExpCond alors {I2; I3;} sinon {I4; I5; I6;} I7;
```


Une solution

```
I1; si ExpCond alors {I2; I3;} sinon {I4; I5; I6;} I7;
```

```
            I1
            evaluer ExpCond + ZNCV
            branch si faux a etiq_sinon
            I2
            I3
            branch  etiq_finsi
etiq_sinon: I4
            I5
            I6
etiq_finsi: I7
```


Codage en ARM

```
a←5;b←6; si a=b alors {x←1;} sinon {x←0;}
```

a et b dans r0, r2, x dans r1

Exécution

I.0
I.1
I.2
I.3
I.4
I.5
I.6
I.7

Ligne	r0	r2	?= ?	r1	proch Ligne
-1	?	?	?	?	0

Une autre solution

```
I1; si ExpCond alors {I2; I3;} sinon {I4; I5; I6;} I7;
```

```
            I1
            evaluer ExpCond + ZNCV
            branch si vrai a etiq_alors
            I4
            I5
            I6
            branch  etiq_finsi
etiq_alors: I2
            I3
etiq_finsi: I7
```


Codage en ARM

```
a←5;b←6; si a=b alors {x←1;} sinon {x←0;}
```

a et b dans r0, r2, x dans r1

Instruction *Tant que*

```
I1; tant que ExpCond faire {I2; I3;} I4;
```


Une première solution

```
I1; tant que ExpCond faire {I2; I3;} I4;
```

```
       I1
debut: evaluer ExpCond + ZNCV
       branch si faux fintq
       I2
       I3
       branch  debut
fintq: I4
```

Codage en ARM

`a←0; b←5; tant que a<b faire {x←a; a←a+1;} x←b;`

a, b dans r0, r2, x dans r1

Exécution

Ligne	r0	r2	?>=?	r1	proch Ligne
-1	?	?	?	?	0

I.0
I.1
I.2
I.3
I.4
I.5
I.6
I.7

Une autre solution

`I1; tant que ExpCond faire {I2; I3;} I4;`

```
          I1
          branch  etiqcond
debutbcle: I2
          I3
etiqcond:  evaluer ExpCond
          branch si vrai debutbcle
fintq:     I4
```


Codage en ARM

`a←0; b←5; tant que a<b faire {x←a; a←a+1;} x←b;`

a, b dans r0, r2, x dans r1

Instruction *Répéter*

```
I1; répéter {I2; I3;} jusqu'à ExpCond; I4;
```


Solution

```
I1; répéter {I2; I3;} jusqu'à ExpCond; I4;
```

```
           I1
debutbcle: I2
           I3
           evaluer ExpCond
           branch si faux debutbcle
           I4
```

Observer les différences entre ce codage et la solution du tant que avec test à la fin.

Codage en ARM

```
a←0;b←5; répéter {a←a+1;} jusqu'à a>b; x←b;
```

a, b dans r0, r2, x dans r1

Instruction *Pour*

```
I1; pour (i←0 à N) {I2; I3;I4;} I5;
```

Solution

`I1; pour (i←0 à N) {I2; I3;I4;} I5;`

```
I1
i<-0
tant que i<N
     I2
     I3
     I4
     i<-i+1
I5
```


Exemple : gestion de tableaux (1/3)

Déclaration d'un tableau de 8 entiers

Codage en ARM

`b←2;a←0; pour (i←0 à 3) {a←b; b←2*b;} a←b;`

i dans r0, 3 dans r3, a, b dans r1, r2

Exemple : gestion de tableaux (2/3)

Exemple : gestion de tableaux, version 2 (3/3)

Conversion de la chaîne ”Hello World !” en majuscule

Exemple 2 : chaîne de caractères

Déclaration de la chaîne ”Hello World !” en `ARM` dans le `.data` :

```
CH: .asciz ``Hello Word!''
```

`CH` est un tableau de caractères 12 caractères : les 11 caractères de la chaîne suivis d'un caractère spécial '\0' (♯0 en `ARM`)

`CH`[0] = 'H'
`CH`[4] = 'o'
`CH`[10] = '!'
`CH`[11] = '\0'

Exercice

Deux boucles imbriquées

```
pour (i=0 a N)
   pour (j=0 a K)
        I2;I3
```

Exercice

Expression conditionnelle complexe avec des *ou*

`si C1 ou C2 ou C3 alors I1;I2 sinon I3`

Solution avec évaluation minimale des conditions

oualors en algorithmique.

Codage en ARM

`x ← 0; pour i←0 à 24 pour j←0 à 9 { x ← x+1; }`

i, j dans r0, r4, 24 dans r2, 9 dans r3, x dans r1

Solution I

`si C1 ou C2 ou C3 alors I1;I2 sinon I3`

```
            evaluer C1
            branch si vrai etiq_alors
            evaluer C2
            branch si vrai etiq_alors
            evaluer C3
            branch si faux etiq_sinon
etiq_alors: I1
            I2
            branch  etiq_fin
etiq_sinon: I3
etiq_fin:
```

Codage en ARM

```
si r0=r1 ou r0=r2 ou r1=r2 alors r1←1; sinon r1←2;
```


Solution II

```
si C1 ou C2 ou C3 alors I1;I2 sinon I3

            evaluer C1
            branch si vrai etiq_alors
            evaluer C2
            branch si vrai etiq_alors
            evaluer C3
            branch si vrai etiq_alors
etiq_sinon: I3
            branch  etiq_fin
etiq_alors: I1
            I2
etiq_fin:
```


Codage en ARM

```
si r0=r1 ou r0=r2 ou r1=r2 alors r1←1; sinon r1←2;
```


Expression conditionnelle complexe avec des *ou*

```
si C1 ou C2 ou C3 alors I1;I2 sinon I3
```

Solution avec évaluation complète des conditions

- Evaluer chaque **Ci** dans un registre
- Utiliser l'instruction **ORR** du processeur.

Expression conditionnelle complexe avec des *et*

```
si C1 et C2 et C3 alors I1;I2 sinon I3
```

Solution avec évaluation minimale des conditions

etpuis en algorithmique.

Codage en ARM

```
si r0=r1 et r1=r2 et r2=r3 alors r1←1; sinon r1←0;
```


Solution

```
si C1 et C2 et C3 alors I1;I2 sinon I3
```

```
            evaluer C1
            branch si faux etiq_sinon
            evaluer C2
            branch si faux etiq_sinon
            evaluer C3
            branch si faux etiq_sinon
etiq_alors: I1
            I2
            branch  etiq_fin
etiq_sinon: I3
etiq_fin:
```


Construction *selon*

```
selon a,b:
  a<b : I1
  a=b : I2
  a>b : I3
```

Une solution consiste à traduire en si alors sinon.

```
si a<b alors I1
sinon si a=b alors I2
      sinon si a>b alors I3
```

Mais ARM offre une autre possibilité. . .

Solution

Instructions ARM conditionnelle.
Dans le codage d'une instruction, champ condition (bits 28 à 31).
Sémantique d'une instruction : si la condition est vraie exécuter l'instruction sinon passer à l'instruction suivante.

```
selon a,b:
  a<b : x<-x+5
  a=b : x<-x+1
  a>b : x<-x+9
```

```
a dans r0, b dans r1, x dans r2
    cmp r0,r1
    addlt r2, r2, #5
    addeq r2, r2, #1
    addgt r2, r2, #9
```

Que se passe-t-il si on remplace le addeq par un addeqs ?

Solution

```
        .data
X:      .word 7
        .text
        .global main
main:




        b exit
ptrX:   .word X
```


Enoncé : le nombre de 1

Traduisez l'algorithme suivant en ARM :

```
x, nb : entiers >= 0

nb<-0
tant que x<>0 faire
   si x mod 2 <> 0 alors nb<-nb+1
   x<-x div 2
fin tant que

afficher nb
```


Tableaux à deux dimensions

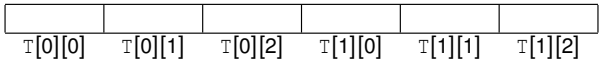
T[2][3] : tableau de 2 lignes à 3 colonnes

	0	1	2
0			
1			

En ARM, zone mémoire contiguë.

`.bss T: .skip 24 @ taille d'un élément (un entier) × #lignes × #colonnes :`

4 octets $\times 2 \times 3$



adresse de $T[0][0]$ = adresse de T
adresse de $T[0][1]$ = adresse de $T + 4$
adresse de $T[0][2]$ = adresse de $T + 8$
adresse de $T[1][0]$ = adresse de $T + 12$
adresse de $T[1][1]$ = adresse de $T + 16$
adresse de $T[1][2]$ = adresse de $T + 20$

Formule générale pour un tableau $T[n][m]$:

adresse de $T[i][j]$ =

Exemple : tables de multiplication (1/3)

```
.data   mess :      .asciz "entrez un entier entre 0 et 9"
        res :       .asciz "resultat"

.bss    x :         .word
        T :         .skip 400                   @ tables de multiplication T[10][10]

.text   .global main
        main :

                    @ code sur les deux slides suivants

                    b exit

        ptr_mess :  .word mess
        ptr_res :   .word res
        ptr_T :     .word T
        ptr_x :     .word x
```


Exemple : tables de multiplication, initialisation du tableau (2/3)

Exemple : tables de multiplication, interrogation d'une table (3/3)

Programmation à partir d'automates reconnaisseurs

Exécution des programmes en langage machine

Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

15 janvier 2020

Plan

Définition

Un automate avec actions à nombre fini d'états
est un 6-uplet $\langle E, S, s_0, O, f, g \rangle$:

1. E, ensemble fini non-vide : vocabulaire d'entrée, les éléments sont les **symboles** d'entrées. Les entrées peuvent aussi être des conditions portant sur les valeurs de variables.
2. S, ensemble fini non-vide : les états.
3. $s_0 \in S$: l'état initial.
4. O, ensemble fini non-vide : vocabulaire de sortie. Les éléments sont des actions sur les variables considérées.
5. f, fonction de transition : $f : E \times S \rightarrow S$.
6. g, fonction de sortie : $g : S \rightarrow O$.

Transitions

Si $f(e_i, s_j) = s_k$ on dit que l'état s_k est le successeur, ou état suivant de s_j pour l'entrée e_i.

On peut aussi parler d'une relation R (au lieu de la fonction f) comportant, notamment, le triplet (e_i, s_j, s_k). Cela revient au même si l'on précise des conditions sur les triplets permis.

Quand le **vocabulaire d'entrée** comporte des conditions booléennes portant sur les variables, le changement d'état peut avoir lieu seulement si la condition est vraie.

Pourquoi utiliser le modèle des automates ?

On parle aussi de :

- machines séquentielles
- automate d'état fini étendu
- et en anglais : Finite State Machine

Intérêt du modèle :

- Décrire le fonctionnement d'un système séquentiel
- Simuler le fonctionnement d'un système séquentiel
- Faire des preuves (équivalence, état atteignable) avant de passer à une réalisation logicielle **ou** matérielle.

Actions associées aux états ou aux transitions

- Actions associées aux états (automate de Moore)
  fonction de sortie : $S \to O$.

  A chaque état correspond une ou plusieurs actions. Quand l'automate est dans cet état, les actions sont exécutées. **Simultanément ou dans l'ordre**, il faut le préciser !

  Différence par rapport aux automates étendus vus au premier semestre dans le cours de Langage et Automates. Les actions étaient associées aux transitions.

- Actions associées aux transitions (automate de Mealy)
  fonction de sortie : $E \times S \to O$.

Utilisation des automates dans ce cours

Dans ce cours :

- Montrer une programmation systématique avec organisation d'un programme en mémoire.
- Pour décrire le fonctionnement d'un processeur, on utilisera le modèle d'automate avec actions.

Exemple (1/2)

On considère des nombres à virgule, écrits en binaire, avec les caractères $\{0, 1, \bullet, \sqcup\}$.

L'ensemble des caractères possibles est donc $\{0, 1, \bullet, \sqcup\}$
$\bullet$ est la virgule, $\sqcup$ est l'espace final.

On se limite à des nombres avec **au plus 4 chiffres avant la virgule et 4 après**.

Pour simplifier on supposera (dans un premier temps) qu'il n'y a pas d'erreurs.

Exemple (2/2)

Des nombres et leurs valeurs :

nombre	valeur
$\sqcup$	
011$\sqcup$	
$\bullet$1001$\sqcup$	
101$\bullet$01$\sqcup$	
101100$\bullet$01$\sqcup$	

Evaluation des nombres à virgule

On définit les variables suivantes :

- E est la valeur de **la partie entière naturelle du nombre** représenté.
- D est la valeur de **la partie naturelle après la virgule du nombre** représenté.
- N est **le nombre de chiffres après la virgule du nombre** représenté.

Exemple :

0110•1010␣ représente

Erreurs :

On pourrait détecter des erreurs :

- si E $>$ 15 (il y aurait plus de 4 chiffres avant la virgule), ou
- si N $>$ 4.

Modélisation

L'algorithme permettant :

- la reconnaissance d'un nombre
- le calcul des valeurs E, D et N

est exprimé sous forme d'un automate avec actions.
$Etats = \{Ini, E0, E1, Poi, D1, D0, Fin\}$.
L'automate atteint les états :

- E0 et E1 après avoir reconnu respectivement un 0 ou un 1 situé avant le •
- Poi après avoir reconnu le •
- D0 et D1 après avoir reconnu respectivement un 0 ou un 1 situé après le •

Des actions sont attachées aux états pour réaliser les calculs.

Modélisation : Graphe étiqueté

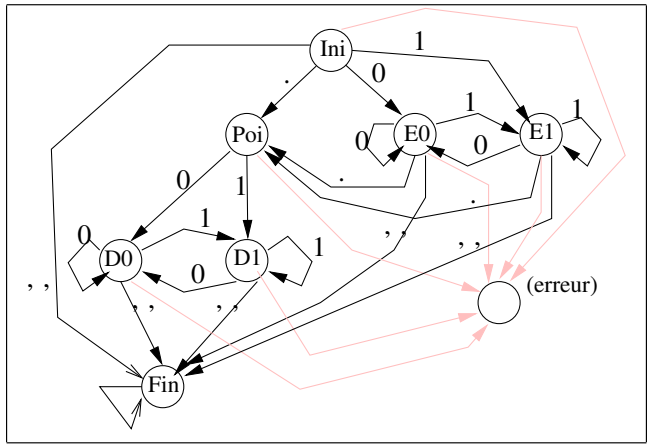


FIGURE – Automate évaluateur

Table de transitions

état de départ \ symbole	0	1	•	␣
Ini	*E*0	*E*1	*Poi*	*Fin*
*E*0	*E*0	*E*1	*Poi*	*Fin*
*E*1	*E*0	*E*1	*Poi*	*Fin*
Poi	*D*0	*D*1		
*D*0	*D*0	*D*1		*Fin*
*D*1	*D*0	*D*1		*Fin*
Fin	*Fin*	*Fin*	*Fin*	*Fin*

Etat	action
Ini	$E \leftarrow 0$, $D \leftarrow 0$, $N \leftarrow 0$
*E*0	$E \leftarrow 2 \times E$
*E*1	$E \leftarrow 2 \times E + 1$
Poi	rien
*D*0	$D \leftarrow 2 \times D$; $N \leftarrow N + 1$
*D*1	$D \leftarrow 2 \times D + 1$; $N \leftarrow N + 1$
Fin	rendre la main

Exemple

Supposons que les symboles en entrée sont 101.110␣.

Etat	E	D	N	symbole lu
Ini	0	0	0	1
$E1$	$2\times0+1=1$	0	0	0
$E0$	$2\times1=2$	0	0	1
$E1$	$2\times2+1=5$	0	0	.
Poi	5	0	0	1
$D1$	5	$2\times0+1=1$	$0+1=1$	1
$D1$	5	$2\times1+1=3$	$1+1=2$	0
$D0$	5	$2\times3=6$	$2+1=3$	␣
Fin	5	6	3	

En langage d'assemblage

```
adresse 0x0100 : (etat Ini)
                 mettre 0 dans la case memoire representant la variable E
                 mettre 0 dans la case memoire representant la variable N
                 mettre 0 dans la case memoire representant la variable D

                 lire entree
                 comparer entree, symbole '0'
                 branch 0x0300 (etat E0) si egal
                 ...
                 comparer entree, symbole ' '
                 branch 0x0800 (etat Fin) si egal
...

adresse 0x0300 : (etat E0)
                 lire la case memoire representant la variable E
                 multiplier son contenu par 2
                 ecrire le resultat dans la case memoire representant la variable E

                 lire entree
                 comparer entree, symbole '0'
                 branch 0x0300 (etat E0) si egal
                 ...
                 comparer entree, symbole ' '
                 branch 0x0800 (etat Fin) si egal
...

adresse 0x0800 : (etat Fin)
                 afficher sortie
                 rendre la main...
```


En langage haut-niveau

```
Etat <- Ini
While (Etat <> Fin (et non Erreur s'il y a lieu)) do
    switch Etat
           case Ini:
                  E<-0; D<-0; N<-0
                  READ entree
                  switch entree
                         case '0': Etat<-E0
                         case '1': Etat<-E1
                         case '.': Etat<-Poi
                         case ' ': Etat<-Fin
                         (case default: Etat<-Erreur)
                  end switch
           case E0:
                  ...
           ...
    end switch
done
Message de fin : Evaluation correct (ou incorrect s'il y a lieu)
```


En ARM (1/3)

```
.data
E :     .word 0
D :     .word 0
N :     .word 0

.bss
C :     .byte       @ Pour la lecture d'un caractère

.text
        .global main
main :

...

        b exit

ptrE :  .word E
ptrD :  .word D
ptrN :  .word N
ptrC :  .word C
```

En ARM (2/3)

```
main :

ini :
        ...

E0 :
        ...

E1 :
        ...

Poi :
        ...

D0 :
        ...

D1 :
        ...

FIN :
```


En ARM (3/3)

```
E0 :    ldr r0,ptrE         @ lecture E
        ldr r1,[r0]

        mov r1,r1,lsl #1    @ E x 2
        str r1,[r0]         @ écriture de E

        ldr r1,ptrC         @ adresse où écrire
        bl LireCar          @ lit un caractère et l'écrit à l'adresse précédente
        ldrb r0,[r1]        @ récupère le caractère lu dans r0

        @ test
        cmp r0,#48          @ ou #'0'
        beq E0
        cmp r0,#49          @ ou #'1'
        beq E1
        cmp r0,#46          @ ou #'.'
        beq Poi
        b FIN
```


Autre contexte de mise en oeuvre : (1/4)

On travaille avec un processeur relié à de la mémoire.
Les adresses mémoires sont des mots de 16 bits. La mémoire est organisée en **octets**.
Le processeur a 1 registre accumulateur `Acc` de taille **8 bits**.
Il y a un registre d'état qui comporte un bit : `Z`. La comparaison de l'accumulateur avec une valeur immédiate positionne le bit du mot d'état.
Le compteur de programme (`pc`) **repère l'instruction qui suit celle qui est en cours d'exécution.**

Autre contexte de mise en oeuvre : (2/4)

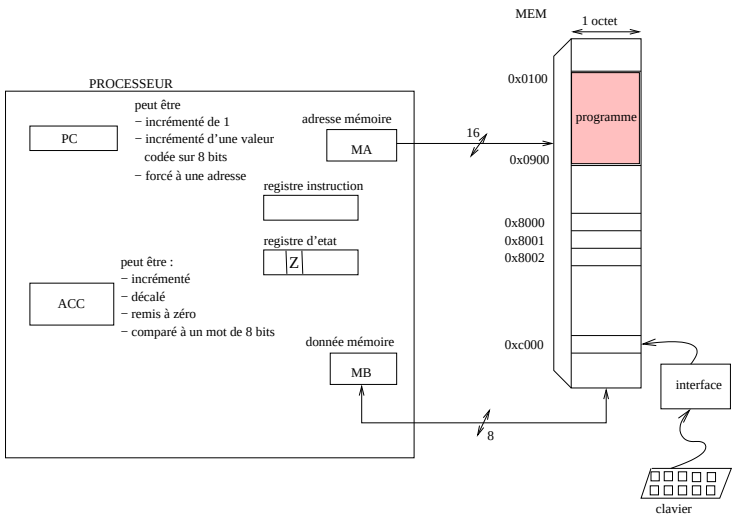
Opérations sur l'accumulateur :

- `load#` charger l'accumulateur avec une valeur immédiate
- `load` charger l'accumulateur avec un mot de la mémoire
- `add#` une valeur immédiate à l'accumulateur
- `lsl` décaler le contenu de l'accumulateur d'une position binaire à gauche
- `store` ranger le contenu de l'accumulateur dans la mémoire

Deux types d'instructions de rupture de séquence :

- branchement conditionnel : `bne` ou `beq`
  branchement relatif, opérande = déplacement sur **8 bits**
- branchement inconditionnel : `jmp`
  branchement absolu, opérande = adresse sur **16 bits**

Autre contexte de mise en oeuvre : (3/4)



Autre contexte de mise en oeuvre : (4/4)

Le code opération est toujours sur **1 octet** (le premier), l'opérande quand elle existe est sur **1 octet** (`vi`, `depl`) ou **2 octets** (`adr`, le premier représente les poids forts de l'adresse).

instruction	signification	codage
`load adr`	`Acc <-- Mem[adr]`	3 octets
`load# vi`	`Acc <-- vi`	2 octets
`store adr`	`Mem[adr] <-- Acc`	3 octets
`add# vi`	`Acc <-- Acc + vi`	2 octets
`lsl`	`Acc <-- Acc * 2`	1 octet
`cmp vi`	`Acc - vi`	2 octets
`beq depl`	si Z=1 `pc <-- pc + depl`	2 octets
`bne depl`	si Z=0 `pc <-- pc + depl`	2 octets
`jmp adr`	`pc <-- adr`	3 octets

On convient que la **lecture** à l'adresse `0xC000` fournit un octet qui est le code ASCII du caractère courant.

On range les valeurs `E`, `D` et `N` aux adresses `0x8000`, `0x8001` et `0x8002`.

Le système donne la main à notre programme qui débute à l'adresse `0x0100`.

Dans un langage d'assemblage particulier : schéma de solution

Le code de chaque état est installé en mémoire à une adresse différente pour chaque état. Le passage d'un état à un autre est fait par une rupture de séquence. On suppose qu'on implante le code des états *Ini*, *E*0, *E*1, *D*0, *D*1, *Poi* et *Fin* aux adresses 0x0100, 0x0200, 0x0300, 0x0400, 0x0500, 0x0800 et 0x0900.

```
0x0100          etat Ini
...             ...
0x????          fin de etat Ini
...             ...
0x0200          etat E0
...
0x????          fin de etat E0
...             ...
0x0300          etat E1
...             ...
0x????          fin de etat E1
...             ...
0x0400          etat D0
...             ...
0x????          fin de etat D0
...             ...
0x0500          etat D1
...             ...
0x????          fin de etat D1
...             ...
0x0800          Poi
...
```


Dans un langage d'assemblage particulier : exercice

Ecrire les parties de programmes correspondants aux états : *Poi*, *D*0 et *D*1.

Dans un langage d'assemblage particulier : solution (Pour *D*0)

Programmation des appels de procédure et fonction
Première séance

Exécution des programmes en langage machine

Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

15 janvier 2020

Sujet de réflexion

1. Que faudrait-il modifier si on ajoutait un état d'erreur ? Indiquer les modifications de l'automate et du programme.
2. Indiquer comment on pourrait modifier l'automate pour n'accepter que des octets avec 4 bits avant le • et 4 bits après.

Plan

Utilité-Nécessité des fonctions et procédures

A quoi servent les fonctions et procédures :

- **Structurer** le code (nommer un bloc d'instruction)
- Eviter de dupliquer du code
- Eviter les structures de contrôles imbriquées
- Permettre l'utilisation de variables **locales**
- Permettre la définition de bibliothèques
- Programmer avec de la **récursivité**
- Préparer la programmation orientée objet

Rappel : en C, et dans beaucoup de langages, tout ou presque est fonction. Il n'y a pas de script C (*i.e.*, code hors fonction). Par contre, il peut y avoir des variables globales (!)

Un exemple en langage de ≪ haut niveau ≫ (2 /2)

```
int PP(int x) {
int z, p;
   z = x + 1;
   p = z + 2;
   return (p);
}

main() {
int i, j, k;
   i = 0;
   j = i + 3;
   j = PP(i + 1);
   k = PP(2 * (i + 5));
}
```

- Il y a deux appels à la fonction `PP`
- Lors de l'appel `PP(i + 1)`, la valeur de l'expression `i+1` est passée à la fonction, c'est le paramètre effectif que l'on appelle aussi argument
- Après l'appel le résultat de la fonction est rangé dans la variable **j** : `j = PP(i+1)`
- Le 1er appel revient à exécuter le corps de la fonction en remplaçant `x` par `i+1` ; le 2ème appel consiste en l'exécution du corps de la fonction en remplaçant `x` par `2*(i+5)`

Un exemple en langage de ≪ haut niveau ≫ (1 /2)

```
int PP(int x) {
int z, p;
   z = x + 1;
   p = z + 2;
   return (p);
}

main() {
int i, j, k;
   i = 0;
   j = i + 3;
   j = PP(i + 1);
   k = PP(2 * (i + 5));
}
```

Analyse

- Le `main`, nommé appelant fait appel à la fonction `PP`, nommée appelée
- La fonction `PP` a un paramètre qui constitue une donnée, on parle de paramètre formel
- La fonction `PP` calcule une valeur de type entier, le résultat de la fonction
- Les variables **z** et **p** sont appelées variables locales à la fonction `PP`

Utilisation de registres

Chaque valeur représentée par **une variable ou un paramètre** doit être rangée quelque part en **mémoire** : mémoire centrale ou registres.

Dans un premier temps, utilisons **des registres**.

On fait un choix (pour l'instant complètement arbitraire) :

- *i*,*j*,*k* dans *r*0,*r*1,*r*2
- *z* dans *r*3, *p* dans *r*4
- la valeur *x* dans *r*5
- le résultat de la fonction dans *r*6
- si on a besoin d'un registre pour faire des calculs on utilisera *r*7 (variable temporaire)

Remarque :

Une fois, ces conventions fixées, on peut écrire le code de **la fonction** indépendemment **du code correspondant à l'appel**, mais cela demande beaucoup de registres.

Code en langage d'assemblage

```
PP :                    @ z ← x + 1
                        @ p ← z + 2
                        @ rendre p
        retourner

main :                  @ i ← 0
                        @ j ← i + 3
@ ——Début-1er-appel——
                        @ x ← i + 1
        appeler PP
                        @ j ← ...
@ ——Fin-1er-appel——
@ ——Début-2ème-appel——
                        @ r7 ← i + 5
                        @ x ← 2 * r7
        appeler PP
                        @ k ← ...
@ ——Fin-2ème-appel——
```

Problème :

appeler et retourner ?

Adresse de retour

Il existe une instruction de rupture de séquence particulière qui permet au processeur de garder l'adresse de l'instruction qui suit le branchement avant qu'il ne réalise le branchement, *i.e.*, avant qu'il ne transfère le contrôle.

Cette adresse est appelée adresse de retour.

On peut simuler cette instruction et la notion d'adresse de retour :

- Ajout d'une étiquette de retour (mais avec une utilisation très limitée, à un seul endroit d'appel/retour)
- Calcul de l'adresse de retour avant l'appel (mais attention : le `PC` avance au cours de l'exécution, `PC` vaut `PC`+8 au moment du `B`)

L'instruction de rupture de séquence particulière recherchée est une facilité justifiée pour des raisons d'efficacité et de garantie de respect des conventions.

Quel est le problème ?

Appel = branchement
instruction de rupture de séquence inconditionnelle (`BAL` ou `B`) ?

MAIS Comment revenir ensuite ?

Le problème du retour : comment à la fin de l'exécution du corps de la fonction, indiquer au processeur l'adresse à laquelle il doit se brancher ?

Où est gardée cette adresse ?

Dans le processeur **ARM**, l'instruction `BL` réalise un branchement inconditionnel avec **sauvegarde de l'adresse de retour** dans le registre nommé `lr` (*i.e.*, `r14`).

`BL` signifie *branch and link*

Attention : `BL` n'a rien à voir avec `B`

Attention : il ne faut pas modifier le registre `lr` pendant l'exécution de la fonction.

Rappel

On a déjà utilisé l'instruction `BL` en Travaux Pratiques :

```
MOV r1, #12
BL EcrHexa32
```


Codage complet de l'exemple

```
PP :    add r3, r5, #1      @ z ← x + 1
        add r4, r3, #2      @ p ← z + 2
        mov r6, r4          @ rendre p
        retour

main :  mov r0, #0          @ i ← 0
        add r1, r0, #3      @ j ← i + 3
@ ——Début-1er-appel——
        add r5, r0, #1      @ x ← i + 1
                            appel
        mov r1, r6          @ j ← PP(x)
@ ——Fin-1er-appel——
@ ——Début-2ème-appel——
        add r7, r0, #5      @ r7 ← i + 5
        mov r5, r7, lsl #1  @ x ← 2 * r7
                            appel
        mov r2, r6          @ k ← PP(x)
@ ——Fin-2ème-appel——
```


Exécution

```
l.0 PP :    add r3, r5, #1
l.1         add r4, r3, #2
l.2         mov r6, r4
l.3         mov pc, lr @ ou bx lr
l.4 main :  mov r0, #0
l.5         add r1, r0, #3
l.6         add r5, r0, #1
l.7         bl PP
l.8         mov r1, r6
l.9         add r7, r0, #5
l.10        mov r5, r7, lsl #1
l.11        bl PP
l.12        mov r2, r6
```

l.	r0	r1	r3	r4	r5	r6	lr	> l.
-1	?	?	?	?	?	?	?	4

Exercice

En supposant que la fonction `PP` est stockée à partir de l'adresse 0x8000 (hexadécimal) et que la fonction `main` est stockée immédiatement après la fonction `PP`.

Donnez la valeur du registre *lr* à chacun des deux appels de `PP`.

Solution

```
PP :    add r3, r5, #1
        add r4, r3, #2
        mov r6, r4
        mov pc, lr          @ ou bx lr

main :  mov r0, #0
        add r1, r0, #3
        add r5, r0, #1
        bl PP
        mov r1, r6
        add r7, r0, #5
        mov r5, r7, lsl #1
        bl PP
        mov r2, r6
```


Conclusion

- Paramètres : il faut une zone de stockage dynamique **commune** à l'*appelant* et à l'*appelé*

  L'appelant y range les valeurs **avant** l'appel et l'appelé y prend ces valeurs et les utilise
- Variables locales : il faut une zone de mémoire dynamique **privée** pour chaque procédure pour y stocker ses variables locales : il ne faut pas que cette zone interfère les variables globales ou locales à l'appelant
- Variables temporaires : elles ne doivent pas interférer avec les autres variables
- Généralisation : il faut que la méthode choisie soit généralisable afin de pouvoir générer du code

Remarque : on a généralement peu de registre à notre disposition (16 en ARM, mais plusieurs sont dédiés à des tâches spécifiques, *i.e.* PC, LR, ...)

Valeurs temporaires-problème de cohérence

Supposons que dans la fonction PP on ait un calcul compliqué à faire et que l'on utilise pour cela le registre *r*7.

```
PP :  ...
      modifie r7
      ...
```

Supposons qu'après l'appel `k= PP(2 * (i + 5))` on ait besoin de la valeur `i+5` et qu'on veuille la prendre dans *r*7 ...

```
main :
        ...
        add r7, r0, #5          @ r7 ← i + 5
        mov r5, r7, lsl #1      @ r5 ← 2 * r7
        bl PP                   @ modification de r7
        mov r2, r6
        ...
        utilisation de r7 qui contient i + 5   @ c'est incorrect !!!
```


Problèmes à résoudre

1. **Appels en cascade**
2. **Gestion de la récursivité**
3. **Gestion des variables et paramètres**
   les paramètres données, le résultat, les variables locales, les variables temporaires
4. **Passage de paramètre** (par adresse ou par valeur)

Point important : Les solutions proposées doivent être systématiques dans le sens où ce sont celles qui permettent à un compilateur de générer automatiquement le code « assembleur » des fonctions

Un premier problème : appels en cascade

Un programme appelle une procédure `P` qui elle-même appelle une procédure `Q`

programme		procédure `P`		procédure `Q`	
	...		...		...
	...		...		...
	appel `P`		appel `Q`		...
α	...	β	...		retour
	...		...		
	...		retour		

Problème

Lors de l'appel de `P`, l'adresse α est rangée dans *lr* et lors de l'appel de `Q`, l'adresse β est rangée aussi dans *lr* **et écrase la valeur précédemment sauvegardée.**

Un deuxième problème : fonctions récursives (1/2)

```
int fact (int x)
   if (x==0) then return 1
   else return x * fact(x-1);

// appel principal
int n, y;
   .... lecture d'un entier dans n
   y = fact(n);
   .... utilisation de la valeur de y
```

- Supposons que *x* est stocké dans *r*0 et le résultat de la fonction fact dans *r*1
- Considérons l'appel `fact`(3)
- `fact`(3)
- *r*0 ← 3
- `fact`(2)
- *r*0 ← 2
- On a perdu la valeur 3

Conclusion : appels en cascade

Stockage de l'adresse de retour

On ne peut pas travailler avec un seul registre pour sauvegarder les différentes adresses de retour.
Il faut sauvegarder une adresse pour chacun des différents appels.

Fonctions récursives (2/2)

Même chose avec les variables locales !

```
int fact (int x) {
int loc;
   if x==0
      loc = 1;
   else {
      loc = x ;
      loc = fact (x-1) * loc;
   };
   return loc;
}
```

Conclusion : fonctions récursives

Conclusion 1

On ne peut pas travailler avec une seule zone de paramètres, il en faut une pour chaque appel et pas pour chaque fonction.
Les paramètres effectifs (ou arguments) sont attachés à l'appel d'une fonction et pas à l'objet fonction lui-même

Conclusion 2

On ne peut pas travailler avec une seule zone pour les variables locales, il en faut une pour chaque appel et pas pour chaque fonction.
Les variables locales sont attachées à l'appel d'une fonction et pas à l'objet fonction lui-même

Solutions

Solution au prochain cours !

Programmation des appels de procédure et fonction
Deuxième séance (utilisation de la pile)
Exécution des programmes en langage machine

Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

15 janvier 2020

- Rappel du cours précédent :

  Pour définir procédures et fonctions il faut une mémoire dynamique [1] pour trouver une solution au problème des retours et au problème des paramètres.

1. Par opposition à « mémoire statique » : mémoire déclarée « en dure » dans le programme (`.data` ou `.bss`).

Plan

Exécutions possibles

X est faux
A1 A2 (B1 B2 B3) A3 (C1 (B1 B2 B3) C2 C3 C4) A4

X est vrai la première fois puis faux
A1 A2 (B1 B2 B3) A3 (C1 (B1 B2 B3) C2
(C1 (B1 B2 B3) C2 C3 C4) C3 C4) A4

X est vrai deux fois de suite puis faux
A1 A2 (B1 B2 B3) A3 (C1 (B1 B2 B3) C2
(C1 (B1 B2 B3) C2 (C1 (B1 B2 B3) C2 C3 C4)
C3 C4) C3 C4) A4

Problème : Comment mettre en place de tels enchaînements en langage machine ?

Appels en cascade : un exemple

- Soit A_i, B_i, C_i des instructions élémentaires
- Soit A, B, C des procédures, A étant la principale
- Soit X une expression booléenne
- Considérons le programme suivant :

procédure ***A***		procédure ***B***		procédure ***C***	
	$A1$		$B1$		$C1$
	$A2$		$B2$		B
	B		$B3$		$C2$
	$A3$				si X alors C
	C				$C3$
	$A4$				$C4$

Solution partielle

En `ARM`, l'adresse de retour est sauvegardée dans le registre `lr`.

Pour résoudre le problème précédent, il faut plusieurs registres ou cases mémoire.

Problèmes :
Le nombre de registres disponibles est limité.
Le nombre de places nécessaires n'est pas connu à l'avance.

Zones de mémoire dynamique

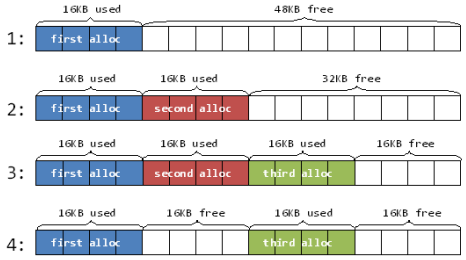
Parmi les zones de mémoire dynamique :

- le tas (heap) (malloc, free ; new, delete),
- la file mécanisme dit FIFO : *First In First Out* (Premier entré, premier sorti) (enfiler, défiler)
- la pile mécanisme dit LIFO : *Last In First Out* (Dernier entré, premier sorti) (empiler, dépiler)

Attention, le tas (heap) est aussi une structure de données qui permet de représenter un arbre dans un tableau (ex. : tri par tas), mais cela n'a que peu de rapport avec la zone de mémoire dynamique.

Notion de tas

Exemple : malloc(first) ; malloc(second) ; malloc(third) ; free(second) ;



(source Qualcomm)
Notions associées

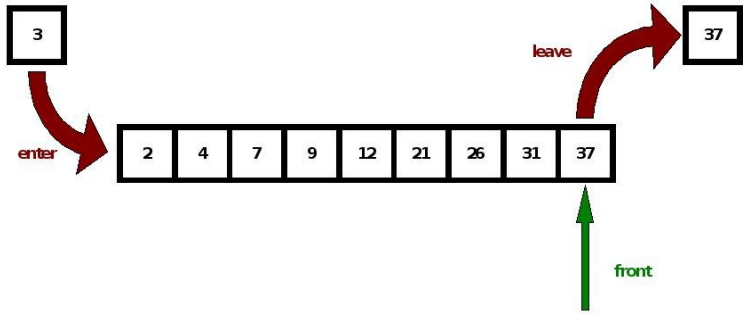
- fragmentation (et défragmentation), ramasse miette (garbage collecting),
- reallocation.

Défragmentation

Reallocation dans le tas

Notion de file

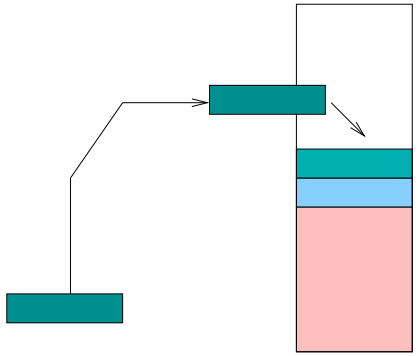
Exemple : enfiler(3) ; $X \leftarrow$ défiler() ;



(source wikipedia)

Notion de pile

Exemple : empiler(X), . . . (autres instructions hors pile) . . . , $Y \leftarrow$ dépiler()



Mécanisme de pile

Notion de tête de pile : dernier élément entré
L'élément en tête de pile est appelé *sommet*.

Deux opérations possibles :

Dépiler : suppression de l'élément en tête de la pile

Empiler : ajout d'un élément en tête de la pile

Comment réaliser une pile ? (1 /4)

- Une zone de mémoire,
- Un repère sur la tête de la pile
  SP : pointeur de pile, *stack pointer*
- Deux choix indépendants :
  - Comment progresse la pile : le sommet est **en direction des adresses croissantes (*ascending*) ou décroissantes (*descending*)**
  - Le pointeur de pile pointe vers une case vide (*empty*) ou pleine (*full*)

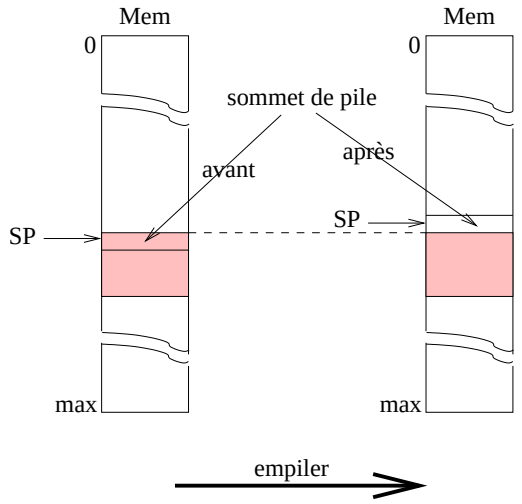
Comment réaliser une pile ? (2 /4)

Mem désigne la mémoire
sp désigne le pointeur de pile
reg désigne un registre quelconque

sens évolution	croissant	croissant	décroissant	décroissant
repère	1er vide	der^{er} plein	1er vide	der^{er} plein
empiler reg	Mem[sp]←reg sp←sp+1	sp←sp+1 Mem[sp]←reg	Mem[sp]←reg sp←sp-1	sp ←sp-1 Mem[sp]←reg
dépiler reg	sp←sp-1 reg←Mem[sp]	reg←Mem[sp] sp←sp-1	sp←sp+1 reg←Mem[sp]	reg←Mem[sp] sp←sp+1

Remarque : Il existe des instructions `ARM` dédiées à l'utilisation de la pile (exemple : pour la gestion full descending on utilise `STMFD` *sp*!, {registre(s)} ou `PUSH` {registre(s)} pour empiler et `LDMFD` *sp*!, {registre(s)} ou `POP` {registre(s)} pour dépiler)

Comment réaliser une pile ? (3 /4)



Comment réaliser une pile ? (3 /4)

En `ARM`, empiler `R3` (convention full descending) :

- `push {R3}`
- `stmfd SP!, {R3}`
- `str R3, [SP, #-4]!`
- `sub SP,SP, #4`
  `str R3, [SP]`

En `ARM`, dépiler `R3` (convention full descending) :

- `pop {R3}`
- `ldmfd SP!, {R3}`
- `ldr R3, [SP], #4`
- `ldr R3, [SP]`
  `add SP,SP, #4`

Appel/retour : utilisation d'une pile

Appel de procédure, deux actions exécutées par le processeur :

- sauvegarde de l'adresse de retour dans une pile
  c'est-à-dire **empiler la valeur *PC* + *taille***
- modification du compteur programme (rupture de séquence)
  c'est-à-dire ***PC*** ← ***adresse de la procédure***

Au retour, *PC* prend pour valeur l'adresse en sommet de pile puis le sommet est dépilé : ***PC*** ← **`depiler()`**.

Remarque : Ce n'est pas la solution utilisée par le processeur `ARM`.

Application sur l'exemple

La taille de codage d'une instruction est supposée être égale à 1

```
10  A1                                  20  B1
11  A2                                  21  B2
12  empiler 13; sauter à 20 (B)         22  B3
13  A3                                  23  retour: dépiler PC
14  empiler 15; sauter à 30 (C)
15  A4
```

```
30  C1
31  empiler 32; sauter à 20 (B)
32  C2
33  si X alors empiler 34; sauter à 30
34  C3
35  C4
36  retour: dépiler PC
```


Trace d'exécution

Trace d'exécution

Appel/retour : solution utilisée avec le processeur ARM

Lors de l'appel, l'instruction BL réalise un branchement inconditionnel **avec sauvegarde de l'adresse de retour** dans le registre nommé lr (*i.e.*, r14).

C'est le programmeur qui doit gérer les sauvegardes dans la pile !

si nécessaire . . .

Application à l'exemple

```
10      A1
11      A2
12      bl B            = (sauver 13 dans lr ; sauter à 20.0)
13      A3
14      bl C            = (sauver 15 dans lr ; sauter à 30.0)
15      A5

20.0    empiler lr
20.1    B1
21      B2
22      B3
23.0    depiler dans lr
23.1    mov pc, lr      @ ou bx lr (restaure lr dans le compteur programme)

30.0    empiler lr
30.1    C1
31      bl B            = (sauver 32 dans lr ; sauter à 20.0)
32      ...
        ...
36.0    dépiler vers lr
36.1    mov pc, lr      @ ou bx lr (restaure lr dans le compteur programme)
```


Gestion des variables, des paramètres : généralisation

La gestion des appels en cascade nous a montré que les adresses de retour nécessitent une gestion ≪ en pile ≫

En fait, c'est le fonctionnement général des appels de procédure qui a cette structure : chaque variable locale et/ou paramètre est rangé dans la pile et la case mémoire associée est repérée par son adresse.

Remarque

Lorsqu'une procédure n'en appelle pas d'autres,

on parle de procédure **feuille**
la sauvegarde dans la pile n'est pas nécessaire.

C'est le cas de la procédure *B* dans l'exemple.

```
20   B1
21   B2
22   B3
23   mov pc, lr @ ou bx lr
```


Exemple

```
procedure A {procedure principale, sans parametre}
var u : entier
   u=2; B(u+3); u=5+u; B(u)

procedure B(donnee x : entier)
var s, v : entier
   s=x+4 ; C(s+1); v=2;  C(s+v)

procedure C(donnee y : entier)
var t : entier
   t=5; ecrire(t*4); t=t+1
```

Flot d'exécution en partant de *A*

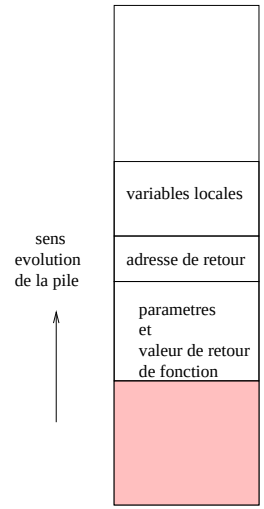
Remarques

Dans l'exemple précédent, nous observons une gestion des zones de mémoire nécessaires pour les paramètres et les variables en pile !

L'approche est identique pour tout : résultats de fonction, paramètres, *etc*.

Et il faut, dans la même pile, sauvegarder les adresses de retour (*cf.* problème des appels en cascade)

Organisation des informations dans la pile lors de l'exécution d'une procédure



Organisation du code

appelant `P` :
préparer les paramètres
`BL Q`
libérer la place allouée aux paramètres

appelé `Q` :
sauver l'adresse de retour
allouer la place pour les variables locales
corps de la fonction
libérer la place réservée pour les variables locales
récupérer adresse de retour
retour

Comment accéder aux variables locales et aux paramètres ?

On pourrait utiliser le pointeur de pile *SP* :

accès indirect avec déplacement : $[SP, \sharp dpl]$

$dpl >= 0$

Mais si on utilise la pile, par exemple pour sauvegarder la valeur d'un registre que l'on souhaite utiliser, il faut re-calculer les déplacements.

Pas pratique !

Pose des problèmes de généralisation

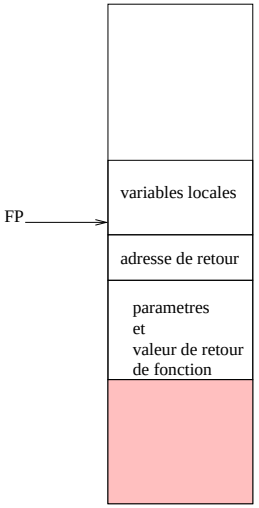
Accès aux variables et paramètres : *frame pointer* (1/2)

Utiliser un repère sur l'environnement courant (paramètres et variables locales) qui reste fixe pendant toute la durée d'exécution de la procédure.

Ce repère est traditionnellement appelé *frame pointer* en compilation

Un registre *frame pointer* existe dans la plupart des architectures de processeur : il est noté `fp` dans le processeur `ARM`.

Accès aux variables et paramètres : *frame pointer* (2/2)



Accès à un paramètre :

$[fp, \sharp dpl\_param]$

$dpl\_param > 0$

Accès à une variable locale :

$[fp, \sharp dpl\_varloc]$

$dpl\_varloc < 0$

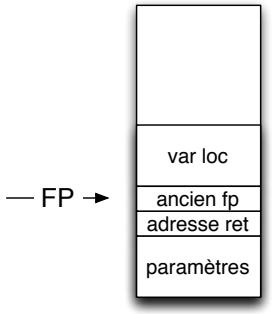
Organisation du code en utilisant le registre *frame pointer*

Comme pour le registre mémorisant l'adresse de retour, le registre `fp` doit être sauvegardé avant d'être utilisé.

appelant `P` :
préparer les paramètres
`BL Q`
libérer la place allouée aux paramètres

appelé `Q` :
sauver l'adresse de retour
sauver l'ancienne valeur de `fp`
placer `fp` pour repérer les nouvelles variables
allouer la place pour les variables locales
corps de la fonction
libérer la place réservée pour les variables locales
restaurer `fp`
récupérer adresse de retour
retour

Organisation de la pile lors de l'exécution avec *frame pointer*



D'où si les adresses sont sur **4 octets** :

- Accès aux variables locales :
  adresse de la forme `fp` $-\ 4\ -$ *déplacement*
- Accès aux paramètres :
  adresse de la forme `fp` $+8+$ *déplacement*

En ARM : code de *B*

```
B:
@ sauvegarde adresse retour
sub sp, sp, #4
str lr, [sp]

@ sauvegarde ancien fp
sub sp,sp,#4
str fp,[sp]

@ mise en place nouveau fp
mov fp,sp

@ reservation variables locales s,v
sub sp,sp,#8

@  s <- x+4
ldr r1, [fp,#+8]
add r1,r1,#4
str r1,[fp,#-4]

@ passage de s+1 en parametre de C
ldr r1, [fp,#-4]
add r1,r1,#1
sub sp,sp,#4
str r1,[sp]

bl C @ appel C

add sp,sp,#4 @ depile le parametre

@ v<-2
mov r1,#2
str r1,[fp,#-8]

@ passe de s+v en parametre de C
ldr r1, [fp,#-4]
ldr r2, [fp,#-8]
add r1,r1,r2
sub sp,sp,#4
str r1,[sp]

bl C @ appel C

add sp,sp,#4 @ depile parametre
add sp,sp,#8 @ depile s,v

@ retour a l'ancien fp
ldr fp,[sp]
add sp,sp,#4

@ recuperation adresse retour
ldr lr, [sp]
add sp,sp,#4

mov pc,lr @ ou bx lr (retour)
```


Exercice

- Ecrire en ARM le code des procédures *A* et *C*
- Montrer l'évolution de la pile, *pc*, *sp*, *fp*, *r*1, *r*2 au cours de l'exécution du programme proposé.

Solution : code de *A*

```
%sub sp,sp,#4
%str fp,[sp]
\begin{verbatim}
main:
@ sauvegarde lr inutile

@ sauvegarde ancien fp inutile

mov fp,sp @ mise en place nouveau fp

sub sp,sp,#4 @ reservation u

@ initialisation u <-2
mov r1, #2
str r1, [fp,#-4]

@ passage de u+3 en parametre
ldr r1, [fp,#-4]
add r1,r1, #3
sub sp,sp,#4
str r1,[sp]

bl B @ appel B

add sp,sp,#4 @ depile parametre
\end{verbatim}

\begin{verbatim}
@ u<-u+5
ldr r1, [fp,#-4]
add r1,r1, #5
str r1, [fp,#-4]

@ passage de u en parametre
ldr r1, [fp,#-4]
sub sp,sp,#4
str r1,[sp]

bl B @ appel B

add sp,sp,#4 @ depile parametre

add sp,sp,#4 @ depile u

bal exit
\end{verbatim}
%@ retour a l'ancien fp
%ldr fp,[sp]
%add sp,sp,#4
```

Solution : code de *C*

```
\begin{verbatim}
C:
@ sauvegarde adresse retour
sub sp, sp, #4
str lr, [sp]

@ sauvegarde ancien fp
sub sp,sp,#4
str fp,[sp]

@ mise en place nouveau fp
mov fp,sp

@ reservation variable locale t
sub sp,sp,#4

@ t <- 5
mov r1,#5
str r1,[fp,#-4]

@ Ecrire t*4
ldr r1,[fp,#-4]
mov r1,r1,lsl #2
bl EcrNdecimal32
\end{verbatim}
```

```
\begin{verbatim}
@ t<-t+1
ldr r1,[fp,#-4]
add r1,r1,#1
str r1,[fp,#-4]

add sp,sp,#4 @ depile t

@ retour a l'ancien fp
ldr fp,[sp]
add sp,sp,#4

@ recuperation adresse retour
ldr lr, [sp]
add sp,sp,#4

mov pc,lr @ retour
\end{verbatim}
```


Avec ou sans pile . . .

- Une utilisation habituelle : la sauvegarde du contexte
- La récursivité coûte cher : pas toujours !
- Une curiosité (?) : les bancs de registres RISC
- Une autre utilisation habituelle : l'évaluation d'expression
- Une étrangeté : la JVM, une machine virtuelle (!)

Compilation des fonctions

- La méthode systématique présentée ici (en particulier la convention d'appel) ne suit pas la norme de `gcc` (qui plus complexe car, en particulier, `gcc` essaie d'éviter de passer les paramètres par la pile quand c'est possible)
- Attention au effet de bord ! Par exemple, qu'affiche le programme suivant ?

```
#include <stdio.h>

int main(){
  int i=0;
  printf("%d < %d\n",i,++i);
  return 0;
}
```

$1 < 1$

Sauvegarde du contexte

Pour préserver une valeur, un contexte, . . . lors de l'exécution d'une tâche :

- empiler l'ensemble à sauvegarder,
- effectuer les tâches à exécuter,
- dépiler l'ensemble sauvegardé.

En particulier, pour préserver un ensemble de registres (en `ARM`), par ex. `R0`, `R2`, `R3` :

- `STMFD SP!, {R0, R2, R3}`
- effectuer les tâches à exécuter,
- `LDMFD SP!, {R0, R2, R3}`

Calcul récursif d'une somme (non terminal)

Idée reçue, la récursivité coûte toujours chère (?)

Exemple : calculer la somme des entiers de 1 à 10.
Solution récursive possible :
Inconvénients :

- après les appels récursifs, il reste des calculs à faire, il faut conserver l'état du contexte.
- 10 appels imbriqués, et autant de mémoire nécessaire

Illustration

main()
⇒ printf(. . . , sommeR(10))
⇒ sommeR(10)
⇒ 10 + sommeR(10-1)
⇒ 9 + sommeR(9-1)
⇒ ...
. . .
⇐
9 + 36
⇐
10 + 45
⇐
55
⇐
affiche 55
⇐
fin

Utilisation de la récursivité terminale

Solution récursive terminale :
Avantage :

- après les appels récursifs, plus de calcul à faire, le contexte peut être écrasé.
- plus d'imbrication nécessaire, c'est équivalent à une boucle . . . !

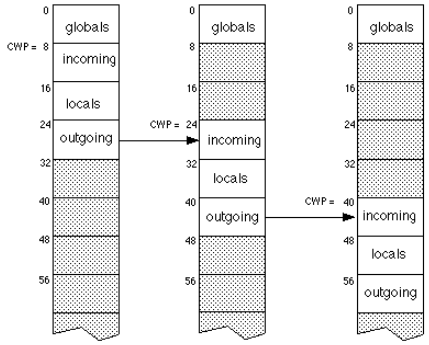
Illustration

main()
⇒ printf(. . . , sommeRT(10,0))
⇒ sommeRT(10,0)
⇔sommeRT(10-1,10+0)
⇔sommeRT(9-1,9+10)
⇔sommeRT(8-1,8+19)
⇔sommeRT(7-1,7+27)
. . .
⇐
55
⇐
affiche 55
⇐
fin

Récursive Terminale en ARM

Banc de registres RISC

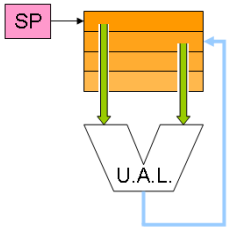
Une curiosité, le banc de registres RISC
Pour ceux qui ont beaucoup de registres . . .



(source M. Meyer)

Evaluation d'expression par une machine à pile

Machine à pile = Machine à 0 adresse :



(source Wikipedia)
Les opérandes viennent de la pile et vont vers la pile, il n'y a pas besoin de les indiquer (0 adresse) : add, sub, mult, div.
Sur la pile, les opérations : empiler x, dépiler y, swap, dup.

Evaluation d'expression par une machine à pile (1/2)

Evaluation de "d :=b*b-4*a*c" par une machine à pile :

init	{ }
empiler b	{ 10} (b)
dup	{ 10 10}
mult	{ 100 } (b*b)
empiler 4	{ 100 4}
empiler a	{ 100 4 2} (a)
mult	{ 100 8 } (4*a)
empiler c	{ 100 8 9 } (c)
mult	{ 100 72} (4*a*c)
sub	{ 28 }
depiler d	{ }

avec a=2, b=10, c=9, d =0
mult : depile, depile, multiplie les opérandes, empile le résultat

Evaluation d'expression par une machine à pile (2/2)

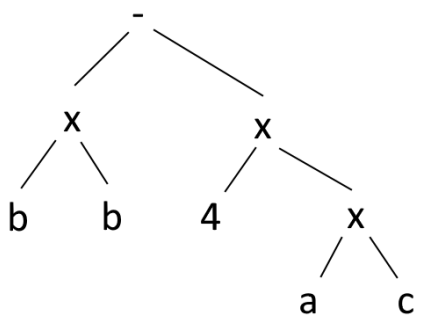
Evaluation de "d :=b*b-4*a*c" par une machine à pile :

init	{ }
empiler b	{ 10} (b)
empiler 4	{ 10 4}
empiler a	{ 10 4 2 } (a)
empiler c	{ 10 4 2 9} (c)
mult	{ 10 4 18}
mult	{ 10 72 }
swap	{ 72 10 }
dup	{ 72 10 10}
mult	{ 72 100 }
rsb	{ 28 }
depiler d	{ }

avec a=2, b=10, c=9, d =0

Evaluation d'expression par une machine à pile

Lien entre l'arbre de l'expression et les programmes d'évaluation pour une machine à pile



Combien d'adresse pour la JVM ?

Traduction de ce code Java :

Combien d'adresse pour la JVM ?

en Java bytecode :

Combien d'adresse pour la JVM ?

en Java bytecode (suite) :
La JVM (Java Virtual Machine) est une machine à pile (machine à 0 adresse, proche de la machine Forth) !

Plan

Programmation des appels de procédure et fonction (fin)

Exécution des programmes en langage machine

Denis Bouhineau Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

15 janvier 2020

Organisation du code (rappel)

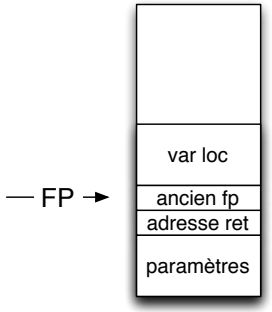
La gestion des appels de fonction est partagée entre l'appelant et l'appelé :

appelant `P` :
préparer et empiler les paramètres
appeler `Q` : `BL Q`
libérer la place allouée aux paramètres

appelé `Q` :
empiler l'adresse de retour
empiler la valeur de `fp`
placer `fp` pour repérer les nouvelles variables
allouer la place pour les variables locales
corps de la fonction `Q`
libérer la place allouée aux variables locales
dépiler `fp`
dépiler l'adresse de retour
retour à l'appelant (`P`) : `MOV PC, LR` ou `BX LR`

Organisation de la pile lors de l'exécution (rappel)

En cours d'exécution du corps de la fonction appelée :



- `fp` $-$ 4 $-$ *déplacement* : Accès aux variables locales
- `fp` $+8+$ *déplacement* : Accès aux paramètres

Rappel : les paramètres données (Qui ? Quand ? Où ?)

Les valeurs des paramètres effectifs sont calculées avant l'appel

C'est la procédure appelante qui connait les informations.

Deux étapes avant l'appel :

- Evaluation des paramètres,
- Stockage des valeurs des paramètres dans la pile.

Le resultat : questions

- Qui s'occupe(nt) du résultat ?
- Quand s'occuper du résultat ?
- Où mettre le résultat ?

Résultat d'une fonction (Qui ? Quand ? Où ?)

1. Le résultat d'une fonction est calculé par l'appelée
2. Le résultat doit être rangé à un emplacement accessible par l'appelante de façon à ce que cette dernière puisse le récupérer.

Il faut donc utiliser une zone mémoire commune à l'appelante et l'appelée.

Par l'exemple, **la pile**.

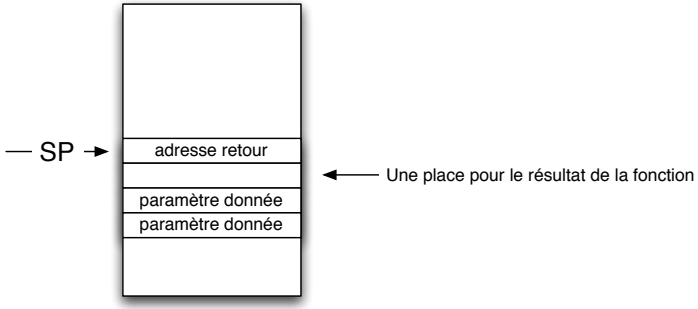
Résultat dans la pile (1/3)

1. avant l'appel, **L'appelant** réserve une place pour le résultat dans la pile
2. **L'appelée** rangera son résultat dans cette case dont le contenu sera récupéré par l'appelant après le retour

Résultat dans la pile (2/3)

Avant l'appel d'une fonction qui a deux paramètres données

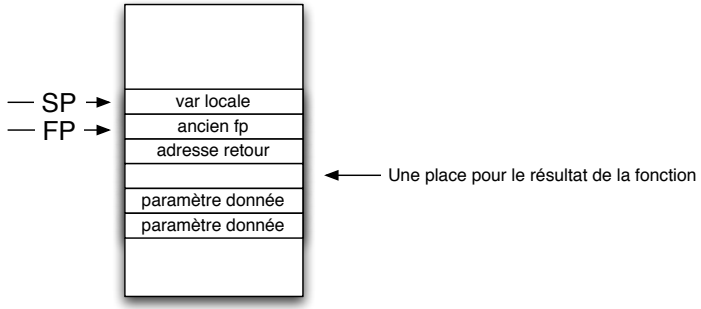
- Les valeurs des deux paramètres sont empilés
- Une case est réservée pour le résultat de la fonction



Résultat dans la pile (3/3)

Lors de l'exécution du corps de la fonction.

1. Les variables locales sont accessibles par une adresse de la forme : $fp - 4 - depl$ avec $depl \geq 0$,
2. Les paramètres données par les adresses : $fp + 8 + 4$ et $fp + 8 + 8$ et
3. La case résultat par l'adresse $fp + 8$.



Structure du code de l'appel de la fonction et du corps de la fonction

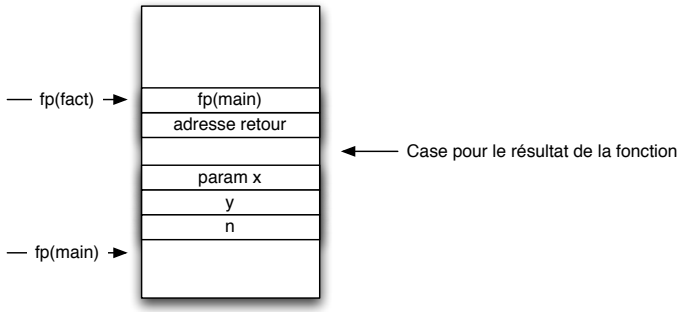
appelant `P` :
préparer et empiler les paramètres
réserver la place du résultat dans la pile
appeler `Q` : `BL Q`
récupérer le résultat
libérer la place allouée aux paramètres
libérer la place allouée au résultat

appelé `Q` :
empiler l'adresse de retour
empiler la valeur de `fp`
placer `fp` pour repérer les nouvelles variables
allouer la place pour les variables locales
corps de la fonction `Q`
le résultat est rangé en **fp+8**
libérer la place allouée aux variables locales
dépiler `fp`
dépiler l'adresse de retour
retour à l'appelant (`P`) : `MOV pc,lr` ou `BX lr`

Application : codage de la fonction factorielle

```
int fact (int x) {
   if (x==0) then return 1
   else return x * fact(x-1);}

main(){
int n, y;
   ...
   y = fact(n);
   ...
}
```


Application : état de la pile lors de l'exécution du corps de fact



Remarque : Dans cet exemple comme il n'y a pas de variables locales on pourrait se passer de mettre en place `fp` pour la fonction appelée. On l'a tout de même utilisé de façon à adresser les paramètres toujours de la même façon.

Application : la fonction `main`

```
main:
        ...
                                        @ r1 ← n
        @ sequence d'appel de fact(n)
        @ empiler la valeur de n, parametre donnee

        @ reserver la place pour le resultat de fact

        bl fact                         @ appel
                                        @ recuperation du resultat dans r1
                                        @ on range le resultat dans y
                                        @ on recupere la place allouee
                                        @ au resultat et au parametre
```

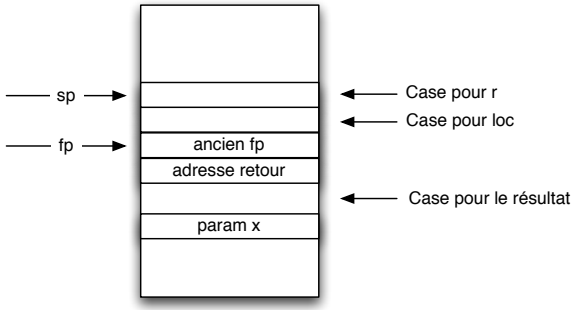

Application : la fonction `fact`

```
                                        @ empiler x-1
fact:
        @ empiler lr

                                        @ place pour resultat

        @ sauvegarde ancien fp et place fp(fact)

                                        bl fact
                                        @ recuperer le resultat de l'appel r1=fact(x-1)

                                        @ liberation parametre et resultat

        @ recuperer la valeur du parametre x : r0=x

                                        @ r0=x

        cmp r0, #0
        bne sinon                       @ r0 * r1 = x * fact(x-1)
alors:  @ ranger le resultat (1)        mul r3, r0, r1
        mov r1, #1                      @ ranger le resultat

        b fsi                   fsi:
sinon:  @ r0 contient x                 @ recuperer fp
        @ appel de fact(x-1)
        @ preparer parametre et resultat
        @ r1 = x-1                      @ retour
        sub r1, r0, #1

                                        @ ou bx lr
```

Exercice

Dérouler l'exécution avec $n = 3$ en dessinant les différents états de la pile, c'est-à-dire la zone de mémoire repérée par `sp`.

Etat de la pile lors de l'exécution du corps de factorielle juste après l'appel dans main



Application : codage d'une fonction factorielle avec des variables locales

```
int fact (int x) {
int loc, r;
   if x==0 { r = 1; }
   else {
      loc = fact (x-1); r = x * loc; }
   return r;
}

main () {
int n, y;
   ...
   y = fact(n);
   ...
}
```


Nouvelle version de la fonction `fact`

```
fact:   @ empiler adr retour
        @ mise en place fp
        @ place pour loc et r

        @ if x==0 ...
                    @ r0=x
alors:
                    @ r = 1
sinon:  @ appel fact(x-1)
        @ preparer param et resultat

                    @ r1=x-1
                                finsi:
                                        @ apres l'appel

                                        @ recuperer place var loc

                                        @ retour
                                                    @ case resultat
                                                    @ appel
                                                    @ recuperer resultat
                                                    @ desallouer param et res

                                                    @ loc=fact(x-1)
                                                    @ r0=x
                                                    @ r1=loc
                                                    @ x*loc
                                                    @ r=x*loc

                                                    @ return r

                                                    @ recuperer fp

                                                    @ ou bx lr
```

Variables temporaires

Problème :

- Les registres utilisés par une procédure ou une fonction pour des calculs intermédiaires locaux sont modifiés
- Or il serait sain de les retrouver inchangés après un appel de procédure ou fonction

Solution :

- Sauvegarder les registres utilisés : `r0, r1, r2`... dans la pile.
- Et cela doit être fait avant de les modifier donc en tout début du code de la procédure ou fonction.

Structure générale du code d'un appel et du corps de la fonction ou procédure

appelant `P` :
1) préparer et empiler les paramètres
2) si fonction, réserver une place dans la pile pour le résultat
3) appeler Q : `BL Q`
4) si fonction, récupérer le résultat
5) si fonction, libérer la place allouée au résultat
6) libérer la place allouée aux paramètres

appelée `Q` :
1) empiler l'adresse de retour (`lr`)
2) empiler la valeur `fp` de l'appelant
3) placer `fp` pour repérer les variables de l'appelée
4) allouer la place pour les variables locales
5) empiler les variables temporaires (registres) utilisées
6) **corps de la fonction**
7) si fonction, le résultat est rangé en **fp+8**
8) dépiler les variables temporaires (registres) utilisées
9) libérer la place allouée aux variables locales
10) dépiler `fp`
11) dépiler l'adresse de retour (`lr`)
12) retour à l'appelant : `MOV pc,lr` ou `BX` lr

Application à l'exemple de la fonction `fact`

Le code de la fonction `fact` utilise les registres `r0, r1, r2`.

```
fact:   @ empiler adr retour
        sub sp, sp, #4
        str lr, [sp]
        @ mise en place fp et allocation loc et r
        sub sp, sp, #4
        str fp, [sp]
        mov fp, sp
        sub sp, sp, #8
        @ sauvegarde de r0, r1, et r2 (empiler)

        @ if x==0 ...
        ...

        @ restaurer les registres r0, r1, r2 (depiler)

        @ desallouer var locales
        add sp, sp, #8
        ldr fp, [sp] @ ancien fp
        add sp, sp, #4
        @ depiler adr retour dans lr
        ldr lr, [sp]
        add sp, sp, #4
        mov pc, lr @ ou bx lr retour
```


Situation : comment faire +1 par programme ?

- Directe :

```
n : entier
   n = n+1;
```

- Par procédure :

```
procedure inc (x : entier)
   x = x+1;

n : entier
  inc(n);
```

- Catastrophe, cela ne marche pas
- Le +1 s'effectue pour l'élément situé sur la pile, pas sur l'original !
- C'est le drame du passage de paramètre par valeur
- **Solution :** passage de paramètre par référence, ou par adresse (paramètre donnée *vs.* paramètre résultat)

Remarque : des fois, ça marche !

Comment faire +1 sur le premier élément d'un tableau

- Par procédure :

```
procedure inc (t : tableau d'entiers)
   t[0] = t[0]+1;

Ns : tableau d'entiers
  inc(Ns);
```

- Cette fois cela marche :-)
- `Ns` et `t` sont des références ...
- C'est la suite du drame du passage de paramètre par valeur

Autre solution

Si on ne peut pas accéder à une référence ...

- Par fonction (et confier l'affectation à l'appelant) :

```
fonction inc (x : entier)
   retourne x+1;

n : entier
  n=inc(n);
```

- Par macro (si disponible)

Référence d'une variable ?

Comment obtenir la référence d'une variable ?

- cela dépend des langages ...
- parfois c'est impossible, déconseillé, difficile, ...
- en C, c'est son adresse : l'adresse de `x` se note `&x` ;
  → accès à une variable à partir de son adresse stockée dans `y` : `*y` ;
- en javascript (pour les variables globales), son nom peut servir de référence : `"n"` ;
  → accès à la valeur à partir de son nom : `window["n"]` ;

Réalisation, vocabulaire

On se place maintenant dans le cas d'une procédure ayant **des paramètres de type donnée** et **des paramètres de type résultat**.

```
procedure XX (donnees x, y : entier; resultat z : entier)
u,v : entier
   ...
   u=x;
   v=y+2;
   ...
   z=u+v;
   ...
```

- Les paramètres données **ne doivent pas être modifiés par l'exécution de la procédure** : les paramètres effectifs associés à *x* et *y* sont des expressions qui sont évaluées avant l'appel, les valeurs étant substituées aux paramètres formels lors de l'exécution du corps de la procédure.
- Le paramètre effectif associé au paramètre formel résultat est une variable **dont la valeur n'est significative qu'après l'appel de la procédure** ; cette valeur est calculée dans le corps de la procédure et affectée à la variable passée en argument.

Notations

Il existe différentes façons de gérer le paramètre `z`. Nous n'en étudions qu'une seule : la méthode dite du passage par adresse.

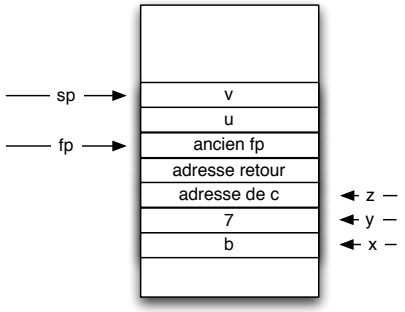
Nous utilisons la notation suivante :

```
procedure XX (donnees x, y : entier; adresse z : entier)
u,v : entier

   ...
   u=x;
   v=y+2;
   ...
   mem[z]=u+v;  @ mem[z] designe le contenu de la memoire d'adresse z
   ...
```


L'exemple d'appel traité

```
a,b,c : entier

   b=3;
   ....
   XX (b, 7, adresse de c);
```


Solution : état de la pile lors de l'exécution de la procédure `XX`



`main`

```
        .bss
a:      .skip 4
b:      .skip 4
c:      .skip 4
        .text
main:
        ...
                @ r0 ← valeur de b
                @ empiler b
                @ r0 ← 7
                @ empiler 7
                @ r0 ← adresse de c
                @ empiler adresse de c
        bl XX   ...
                ...
                b exit
ptr_a:  .word a
ptr_b:  .word b
ptr_c:  .word c
```

Procédure XX

XX:

...

@ $u \leftarrow x$

@ $v \leftarrow y + 2$

...

@ calcul de $u + v$

@ $r2 \leftarrow z$, i.e., adresse c
@ $mem[z] \leftarrow u + v$, i.e., mem[adresse c] $\leftarrow u + v$

...

Solution : la procédure principale

Exercice

Codez en ARM le programme suivant :

```
procédure swap(adresse x : entier, adresse y : entier)
  z : entier
  z=mem[x]
  si z > mem[y]
                  mem[x]=mem[y]
                  mem[y]=z
fin procédure

procédure main()
  a,b : entier
  a=9
  b=7
  afficher a
  afficher b
  swap(adresse a,adresse b)
  afficher a
  afficher b
fin procédure
```


Solution : la procédure swap

remarque : Swap en javascript

- Pour objets globaux :
- function swap(a,b) { // a, b : noms (references)
  var tmp = window[a] ;
  window[a] = window[b] ;
  window[b] = tmp ; }
  var x=0 ;
  var y=1 ;
  swap("x","y")
- Sans référence :
- function swap(a,b) {
  return [b, a] ; }
  var x=0 ;
  var y=1 ;
  ([x,y]) = swap([x,y]) ;

Autre utilisation du passage par adresse (en C).

Proposition pour le calcul de la longueur d'un texte :

```
#include <stdio.h>

typedef struct {  char txt[100];  int lng;} t_texte;

int longueur(t_texte p, int i) {
  if ((i==100) || (p.txt[i]==0) || (p.txt[i]==10) || (p.txt[i]==13)) {
    return 0;}
  else {
    return 1+longueur(p,i+1);}}

int main() {
t_texte unTexte;
  printf("Entree ?\n");
  scanf("%s",unTexte.txt);
  unTexte.lng = longueur(unTexte,0);
  printf("\nSortie :%d\n",unTexte.lng);
  return 0;}
```


Autre utilisation du passage par adresse (en ARM).

```
longueur: sub sp, sp, #16
stmfd sp!, {fp, lr}
add fp, sp, #4
sub sp, sp, #96
ldr r3, [fp, #108]
cmp r3, #100
beq .L2
[...]
.L2: mov r3, #0
b .L4
.L3: ldr r3, [fp, #108]
add r3, r3, #1
str r3, [sp, #88]
mov r1, sp
add r2, fp, #20
mov r3, #88
mov r0, r1
mov r1, r2
mov r2, r3
bl memcpy
add r3, fp, #4
ldmia r3, {r0, r1, r2, r3}
bl longueur
mov r3, r0
add r3, r3, #1
.L4: mov r0, r3
sub sp, fp, #4
ldmfd sp!, {fp, lr}
add sp, sp, #16
```

```
main: stmfd sp!, {fp, lr}
add fp, sp, #4
sub sp, sp, #200
ldr r0, .L6
bl puts
sub r3, fp, #108
ldr r0, .L6+4
mov r1, r3
bl scanf
mov r3, #0
str r3, [sp, #88]
mov r1, sp
sub r2, fp, #92
mov r3, #88
mov r0, r1
mov r1, r2
mov r2, r3
bl memcpy
sub r3, fp, #108
ldmia r3, {r0, r1, r2, r3}
bl longueur
mov r3, r0
str r3, [fp, #-8]
ldr r3, [fp, #-8]
ldr r0, .L6+8
mov r1, r3
bl printf
mov r3, #0
mov r0, r3
sub sp, fp, #4
ldmfd sp!, {fp, lr}
```


Autre utilisation du passage par adresse (en x86).

```
longueur:
.LFB0: pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl %edi, -4(%rbp)
cmpl $100, -4(%rbp)
je .L2
[...]
cmpb $13, %al
jne .L3
.L2: movl $0, %eax
jmp .L4
.L3: movl -4(%rbp), %eax
addl $1, %eax
subq $8, %rsp
pushq 112(%rbp)
pushq 104(%rbp)
pushq 96(%rbp)
[...]
pushq 24(%rbp)
pushq 16(%rbp)
movl %eax, %edi
call longueur
addq $112, %rsp
addl $1, %eax
.L4: leave
ret
```

```
main:
.LFB1: pushq %rbp
movq %rsp, %rbp
subq $112, %rsp
movl $.LC0, %edi
call puts
leaq -112(%rbp), %rax
movq %rax, %rsi
movl $.LC1, %edi
movl $0, %eax
call __isoc99_scanf
subq $8, %rsp
pushq -16(%rbp)
pushq -24(%rbp)
pushq -32(%rbp)
[...]
pushq -104(%rbp)
pushq -112(%rbp)
movl $0, %edi
call longueur
addq $112, %rsp
movl %eax, -12(%rbp)
movl -12(%rbp), %eax
movl %eax, %esi
movl $.LC2, %edi
movl $0, %eax
call printf
movl $0, %eax
leave
ret
```

Autre utilisation du passage par adresse (en C).

Proposition 2 pour le calcul de la longueur d'un texte (avec passage par adresse) :

```
#include <stdio.h>

typedef struct {  char txt[100];  int lng;} t_texte;

int longueur(t_texte *p, int i) {
  if ((i==100) || (p->txt[i]==0) || (p->txt[i]==10) || (p->txt[i]==13)) {
    return 0;}
  else {
    return 1+longueur(p,i+1);}}

int main() {
t_texte unTexte;
  printf("Entree ?\n");
  scanf("%s",unTexte.txt);
  unTexte.lng = longueur(&unTexte,0);
  printf("\nSortie : %d\n",unTexte.lng);
  return 0;}
```


Autre utilisation du passage par adresse (en ARM).

```
longueur:
stmfd sp!, {fp, lr}
add fp, sp, #4
sub sp, sp, #8
str r0, [fp, #-8]
str r1, [fp, #-12]
ldr r3, [fp, #-12]
cmp r3, #100
beq .L2
[...]
.L2: mov r3, #0
b .L4
.L3: ldr r3, [fp, #-12]
add r3, r3, #1
ldr r0, [fp, #-8]
mov r1, r3
bl longueur
mov r3, r0
add r3, r3, #1
.L4: mov r0, r3
sub sp, fp, #4
ldmfd sp!, {fp, lr}
bx lr
```

```
main: stmfd sp!, {fp, lr}
add fp, sp, #4
sub sp, sp, #104
ldr r0, .L6
bl puts
sub r3, fp, #108
ldr r0, .L6+4
mov r1, r3
bl scanf
sub r3, fp, #108
mov r0, r3
mov r1, #0
bl longueur
mov r3, r0
str r3, [fp, #-8]
ldr r3, [fp, #-8]
ldr r0, .L6+8
mov r1, r3
bl printf
mov r3, #0
mov r0, r3
sub sp, fp, #4
ldmfd sp!, {fp, lr}
bx lr
```


Autre utilisation du passage par adresse (en x86).

```
longueur: .LFB0:
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movq %rdi, -8(%rbp)
movl %esi, -12(%rbp)
cmpl $100, -12(%rbp)
je .L2
[...]
cmpb $13, %al
jne .L3
.L2: movl $0, %eax
jmp .L4
.L3: movl -12(%rbp), %eax
leal 1(%rax), %edx
movq -8(%rbp), %rax
movl %edx, %esi
movq %rax, %rdi
call longueur
addl $1, %eax
.L4:
leave
ret
```

```
main:
.LFB1: pushq %rbp
movq %rsp, %rbp
subq $112, %rsp
movl $.LC0, %edi
call puts
leaq -112(%rbp), %rax
movq %rax, %rsi
movl $.LC1, %edi
movl $0, %eax
call __isoc99_scanf
leaq -112(%rbp), %rax
movl $0, %esi
movq %rax, %rdi
call longueur
movl %eax, -12(%rbp)
movl -12(%rbp), %eax
movl %eax, %esi
movl $.LC2, %edi
movl $0, %eax
call printf
movl $0, %eax
leave
ret
```


Question : hors la pile ?

Si le programme est l'exécution d'une fonction principale (main) et que les paramètres (coté appelant, coté appelé), variables (locales, temporaires), résultat, sont dans la pile, que reste-t-il hors de la pile ?

Réponses :

- pas grand chose
- les variables globales
- le tas !
- (et des constantes [par exemple des textes])

Conclusion / Rappel : Structure générale du code d'un appel et du corps de la fonction ou procédure

appelant `P` :
1) préparer et empiler les paramètres (valeurs et/ou adresses)
2) si fonction, réserver une place dans la pile pour le résultat
3) appeler Q : `BL Q`
4) si fonction, récupérer le résultat
5) si fonction, libérer la place allouée au résultat
6) libérer la place allouée aux paramètres

appelée `Q` :
1) empiler l'adresse de retour (`lr`)
2) empiler la valeur `fp` de l'appelant
3) placer `fp` pour repérer les variables de l'appelée
4) allouer la place pour les variables locales
5) empiler les variables temporaires (registres) utilisées
6) **corps de la fonction**
7) si fonction, le résultat est rangé en **fp+8**
8) dépiler les variables temporaires (registres) utilisées
9) libérer la place allouée aux variables locales
10) dépiler `fp`
11) dépiler l'adresse de retour (`lr`)
12) retour à l'appelant : `MOV pc,lr` ou `BX lr`

Plan

Introduction à la structure interne des processeurs : une machine à 5 instructions

Exécution des programmes en langage machine

Denis Bouhineau Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

15 janvier 2020

Aujourd'hui

- Aujourd'hui nous allons étudier comment un processeur exécute un programme.
- Pour cela, nous allons considérer une machine simpliste :

  un processeur à 5 instructions.

clear (acc), load (#vi), add (@), store (@), jmp (@)

Structure du Processeur vu du programmeur

- On considère un processeur comportant un seul registre de données directement visible par le programmeur : `ACC` (pour accumulateur).
- La taille du codage d'**une adresse** et d'**une donnée** est **4 bits**.

Quelle est la taille de la mémoire ?

Les instructions

Les instructions sont décrites ci-dessous. On donne pour chacune une syntaxe de langage d'assemblage et l'effet de l'instruction.

- `clear` : mise à zéro du registre `ACC`.
- `load #vi` : chargement de la valeur immédiate `vi` dans `ACC`.
- `store ad` : rangement en mémoire à l'adresse `ad` du contenu de `ACC`.
- `jmp ad` : saut à l'adresse `ad`.
- `add ad` : mise à jour de `ACC` avec la somme du contenu de `ACC` et du mot mémoire d'adresse `ad`.

Codage des instructions

Les instructions sont codées sur **1 ou 2 mots de 4 bits chacuns** :

- le premier mot représente le code de l'opération (`clear`, `load`, `store`, `jmp`, `add`) ;
- le deuxième mot, s'il existe, contient une adresse ou bien une constante.

Le codage est le suivant :

`clear`	1	
`load #vi`	2	vi
`store ad`	3	ad
`jmp ad`	4	ad
`add ad`	5	ad

Exemple de programme (1/2)

```
    load #3
    store 8
et : add 8
    jmp et
```

Que contient la mémoire après assemblage (traduction en binaire) et chargement en mémoire ? On suppose que l'adresse de chargement est `0`.

Exemple de programme (2/2)

```
        load #3
        store 8
et :    add 8
        jmp et
```

Que calcule ce programme ?

Algorithme d'interprétation

En adoptant un point de vue fonctionnel, en considérant les ressources du processeur comme les variables d'un programme, l'algorithme d'interprétation des instructions peut être décrit de la façon suivante :

```
pc ← 0
tantque vrai
        selon mem[pc]
        mem[pc]=1 {clear} :   acc ← 0                          pc ← pc+1
        mem[pc]=2 {load} :    acc ← mem[pc+1]                  pc ← pc+2
        mem[pc]=3 {store} :   mem[mem[pc+1]] ← acc             pc ← pc+2
        mem[pc]=4 {jmp} :                                      pc ← mem[pc+1]
        mem[pc]=5 {add} :     acc ← acc + mem[mem[pc+1]]       pc ← pc+2
```

Exercice : Dérouler l'exécution du programme précédent en utilisant cet algorithme.

Interprétation des instructions ? (sémantique opérationnelle)

L'interprétation c'est *la description de l'exécution du programme*.

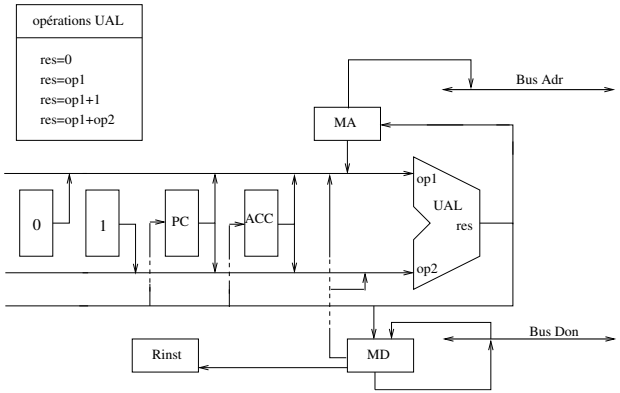
Reprenons l'exemple précédent :

- 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 `load`, l'interprète lit une information supplémentaire (ici, une valeur) dans le mot d'adresse 1.
- La valeur est alors chargée dans le registre `ACC`.
- Finalement, le compteur programme (`PC`) est modifié de façon à traiter l'instruction suivante.

Exécution du programme

Partie opérative

Le processeur comporte une partie qui permet de stocker des informations dans des registres (visibles ou non du programmeur), de faire des calculs (+, -, and,...). Cette partie est reliée à la mémoire par **les bus adresses et données**. On l'appelle Partie Opérative.



Structure de la partie opérative

- Selon l'organisation de cette partie opérative, un certain nombre d'actions de base sont possibles : on les appelle des micro-actions.
- Dans la partie opérative on a des registres : `pc`, `acc`, `rinst`, `ma` (memory address), `md` (memory data),...
- La partie opérative donne aussi des infos qui servent à élaborer les conditions (flag).

Micro-actions et micro-conditions

On fait des hypothèses FORTES sur les transferts possibles :

md $\leftarrow$ **mem[ma]**	lecture d'un mot mémoire.	C'est la seule possibilité en lecture !
mem[ma] $\leftarrow$ **md**	écriture d'un mot mémoire	C'est la seule possibilité en écriture !
rinst $\leftarrow$ **md**	affectation	C'est la seule affectation possible dans `rinst`
$\mathbf{reg_0} \leftarrow \mathbf{0}$	affectation	reg_0 est `pc`, `acc`, `ma`, ou `md`
$\mathbf{reg_0} \leftarrow \mathbf{reg_1}$	affectation	reg_0 est `pc`, `acc`, `ma`, ou `md` reg_1 est `pc`, `acc`, `ma`, ou `md`
$\mathbf{reg_0} \leftarrow \mathbf{reg_1} + 1$	incrémentation	reg_0 est `pc`, `acc`, `ma`, ou `md` reg_1 est `pc`, `acc`, `ma`, ou `md`
$\mathbf{reg_0} \leftarrow \mathbf{reg_1} + \mathbf{reg_2}$	opération	reg_0 est `pc`, `acc`, `ma`, ou `md` reg_1 est `pc`, `acc`, `ma`, ou `md` reg_2 est `pc`, `acc`, ou `md`

On fait aussi des hypothèses sur les tests : (rinst = entier)

Ces types de transferts et les tests constituent le langage des micro-actions et des micro-conditions.

Introduction

Le processeur comporte une partie qui permet :

- d'enchainer des calculs et/ou
- des manipulations de registres et/ou
- des accès à la mémoire.

C'est la Partie Contrôle, aussi appelée algorithme d'interprétation des instructions du processeur.

C'est une machine séquentielle (automate) avec actions.

Exécution d'un clear

Prochaine instruction en assembleur : `clear`
Exécution :

- lecture de l'instruction, positionne l'adresse : ma ← pc
- lecture de l'instruction, récupère l'instruction : md ← Mem[ma]
- lecture de l'instruction, sauvegarde : rinst ← md
- clear : acc ← 0
- avance la lecture : pc ← pc + 1
- instruction suivante

Exécution d'une lecture

Prochaine instruction en assembleur : `load #8`
Exécution :

- lecture de l'instruction, positionne l'adresse : ma ← pc
- lecture de l'instruction, récupère l'instruction : md ← Mem[ma]
- lecture de l'instruction, sauvegarde : rinst ← md
- avance la lecture : pc ← pc + 1
- lecture de l'opérande, positionne l'adresse : ma ← pc
- lecture de l'opérande, récupère l'opérande : md ← Mem[ma]
- lecture de l'opérande, sauvegarde : acc ← md
- avance la lecture : pc ← pc + 1
- instruction suivante

Exécution d'une addition

Prochaine instruction en assembleur : `add 8`
Exécution :

- lecture de l'instruction, positionne l'adresse : ma ← pc
- lecture de l'instruction, récupère l'instruction : md ← Mem[ma]
- lecture de l'instruction, sauvegarde : rinst ← md
- avance la lecture : pc ← pc + 1
- lecture de l'opérande, positionne l'adresse : ma ← pc
- lecture de l'opérande, récupère l'opérande : md ← Mem[ma]
- lecture de l'opérande, sauvegarde : ma ← md
- lecture de l'opérande, déréférencement : md ← Mem[ma]
- addition : acc ← acc + md
- avance la lecture : pc ← pc + 1
- instruction suivante

Autres . . .

Autres instructions

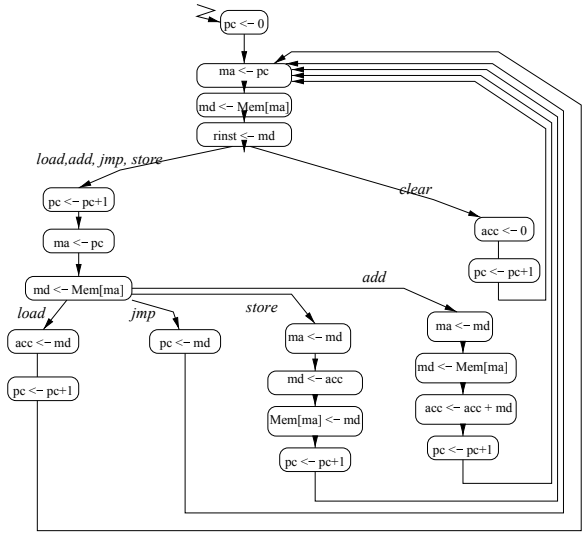
- jmp :
  → similaire à load mais avec pc au lieu de acc
- store :
  → similaire à add mais avec écriture au lieu de somme

Réalisation de l'ensemble sous forme d'un automate

Mise en forme

- Ajout d'une initialisation globale : pc $\leftarrow$ 0
- Mise en commun des premières étapes (identiques)
- Réalisation d'alternatives entre transitions dépendant d'instructions différentes
- Boucle sur la prochaine instruction

Une première version



Remarque : La notation de la condition `clear` doit être comprise comme le booléen `rinst = 1`.

Réalisation et optimisation

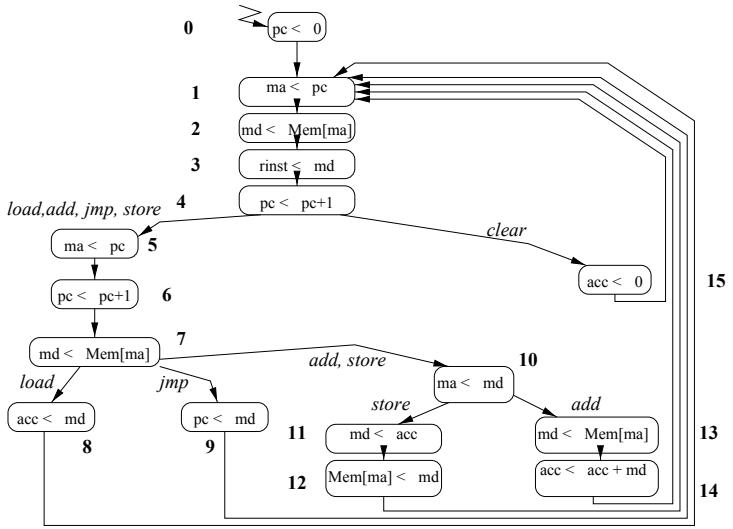
Cet automate est réalisé de manière systématique avec **du matériel** (des transistors).

Optimisation : minimiser le nombre d'états pour réduire la taille du processeur et accélérer le temps d'exécution d'une instruction.

Quelques pistes :

- `pc` $\leftarrow$ `pc +1` peut être fait en avance.
- `ma` $\leftarrow$ `md` commun à plusieurs chemins.

Version amélioration



Exemple de code

étiquette	mnémonique ou directive	référence	mode adressage
	.text		
debut :	clear		
	load	#8	immédiat
ici :	store	xx	absolu ou direct
	add	xx	absolu ou direct
	jmp	ici	absolu ou direct
	.data		
xx :			

Exercice : Que contient la mémoire après chargement en supposant que l'adresse de chargement est 0 et que *xx* est l'adresse 15.

Déroulement

état	*pc*	*ma*	*md*	*rinst*	*acc*	*mem*[15]

Contenu en mémoire

Exercice : Donnez le déroulement au cycle près du programme.

Déroulement

Pipeline (chaîne de traitement)

Hypothèses : la chaîne de traitement d'une instruction comporte 5 étapes :

- IF : récupération de l'instruction
- ID : récupération des opérandes
- EX : exécution de l'opération
- MEM : écriture ou lecture mémoire
- WB : écriture registre

Exécution de trois instructions

Exécution "simple" de trois instructions :

(source wikipedia)

- temps d'exécution : 15 Δ
- Peut mieux faire . . .

Principe du pipeline

Principes :

- chaque étape est indépendante
- chaque étape peut-être réalisée par une partie du processeur différente
- les données peuvent passer d'une partie du processeur à la suivante sans délai
- **conclusion :** le processeur peut faire du travail à la chaine, chaque partie du processeur peut travailler sur la chaîne des données

Exécution de cinq instructions

Exécution pipelinée de cinq instructions, les unes à la suite des autres :

(source wikipedia)

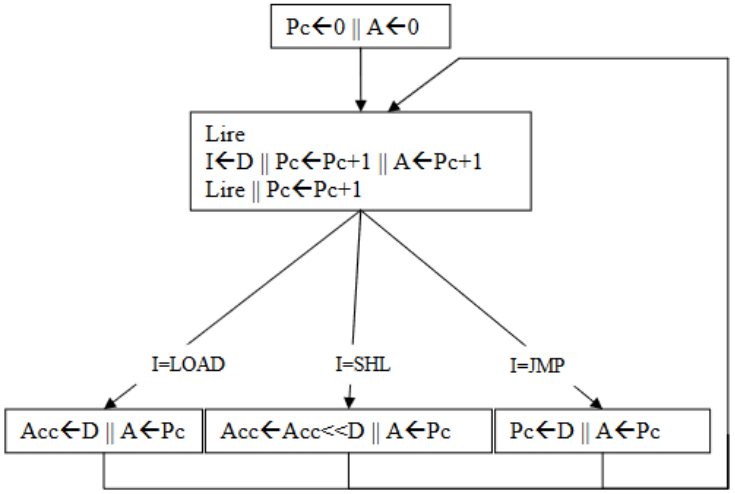
- temps d'exécution : 9 Δ

Exécution de 100 instructions

- Temps d'exécution, si exécution simple : 500 Δ
- Temps d'exécution, si exécution pipelinée : 104 Δ
- Gain espéré : processeur 5 fois plus rapide (la profondeur du pipeline)

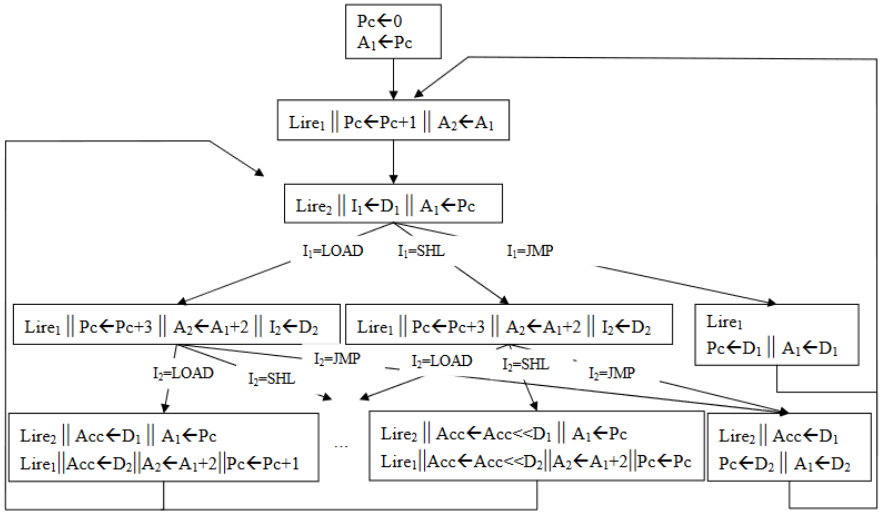
Automate d'interprétation avec pipeline

Version d'un automate simple sans pipeline



Automate d'interprétation avec pipeline

Version avec pipeline



Variantes

Pour aller plus vite (?), des pipelines plus longs (plus profonds)

- Intel Itanium (2001, . . .), profondeur du pipeline : 10
- AMD Optéron (2005, . . .), profondeur du pipeline : 12
- AMD K10 (2006, . . .), profondeur du pipeline : 16
- Intel premier Pentium 4 (2000, . . .), profondeur du pipeline : 20
- Intel Pentium 4 Prescott (2004, . . .), profondeur du pipeline : 32

Rupture de pipeline

Mais !
les pipelines peuvent être rompus :

- Branchement
- Dépendance entre données
- Interruption
- ...
- des bulles s'introduisent dans le pipeline !

Les gains espérés sont des **gains maximum**. En réalité, les gains obtenus sont moindres.

Plan

La vie des programmes

Exécution des programmes en langage machine

Denis Bouhineau Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

15 janvier 2020

Aujourd'hui

Nous allons étudier en détail les différentes étapes de compilation permettant de produire un exécutable à partir d'un ou plusieurs fichiers sources.

Remarque : lorsque l'on compile plusieurs fichiers sources en un seul exécutable, on parle de compilation séparée.

Analyse et synthèse

La compilation comporte deux étapes principales :

- Étape d'analyse
  - Pré-traitement
  - Analyse lexicale
  - Analyse syntaxique
  - Analyse sémantique
- Étape de synthèse de code
  - Génération de code intermédiaire
  - Optimisation de code intermédiaire
  - Génération de code cible

Dans ce cours, nous nous préoccuperons **surtout** de la seconde étape.

Compilation et interprétation

Un programme peut être :

- compiler, puis exécuter
- ou exécuter directement *via* un interpréteur

Compilateurs et interpréteurs partagent la première étape de travail (étape d'analyse).

Compilateurs et interpréteurs se distinguent au moment de l'exécution :

- le code cible produit par un compilateur est exécuté directement par la machine cible
- la structure intermédiaire obtenue par l'interpréteur est exécutée par l'interpréteur lui-même (comme sur une machine virtuelle)

Analyse (rapidement)

L'étape d'analyse se décompose en plusieurs tâches :

- **Précompilation :** Suppression des commentaires, inclusions, macros

  `arm-eabi-gcc -E monprog.c > monprog.i`

  Source `monprog.c` → source ≪ enrichi ≫ `monprog.i`
- **Analyse lexicale :** lecture de `monprog.i` à l'aide d'automates décrivant les éléments lexicaux autorisés (lexèmes)

  source ≪ enrichi ≫ `monprog.i` → `flux de lexèmes`
- **Analyse syntaxique :** analyse du `flux de lexèmes` (souvent à l'aide d'automates à pile) selon la grammaire décrivant le langage de programmation

  `flux de lexèmes` → `arbre syntaxique`
- **Analyse sémantique :** vérification globale des définitions, types déclarés, dépendances, *etc.* des éléments de l'arbre syntaxique

  `arbre syntaxique` → `arbre enrichi (décoré)` et dictionnaires

Analyse (rapidement)

Exemple :

monprog.c :

```
/* monprog.c */
#include "malib.h"

#define max(a,b) ((a)>(b)?(a):(b))

main()
{
  int x,y,z;
  x = 10;
  y = 15;
  z = max(x,y);
  plus (&z);
}
```

malib.c :

```
void plus (int *ai)
{
  (*ai) = (*ai) + 1;
}
```

malib.h :

```
#ifndef MALIB

#define MALIB

/* incrementation d'un mot memoire */
extern void plus (int *);
#endif
```

Analyse (rapidement, la précompilation) : arm-eabi-gcc -E monprog.c > monprog.i (2/2)

monprog.i :

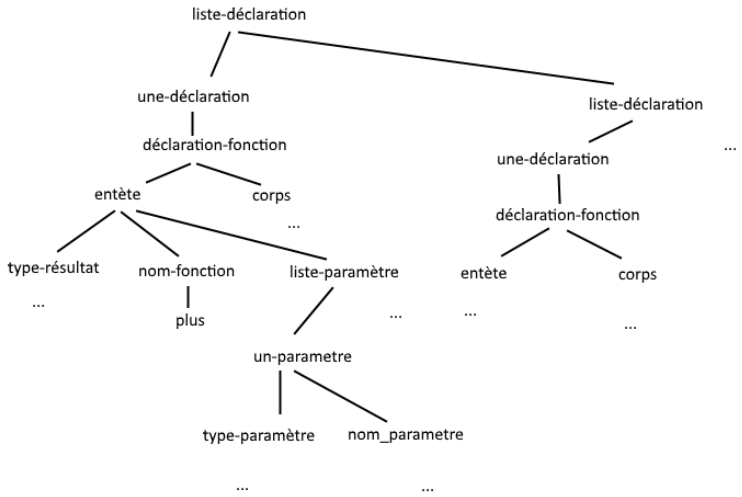
```
# 1 "monprog.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "monprog.c"
# 1 "malib.h" 1

extern void plus (int *);
# 3 "monprog.c" 2

main()
{
  int x,y,z;
  x = 10;
  y = 15;
  z = ((x)>(y)?(x):(y));
  plus (&z);
}
```


Analyse (rapidement, l'analyse lexicale)

le flux de lexèmes (identifiants : (a..zA..Z)(a..zA..Z0..9)*, nombres : (0..9)+(,(0..9)+) ?, mots clés : "int" | "void" | "if" | "for" | ... , *etc.*) :

```
(identifiant,z)
(operateur,"=")
(operateur,"(")
(operateur,"(")
(identifiant,x)
(operateur,")")
(operateur,">")
(operateur,"(")
(identifiant,y)
(operateur,")")
(operateur,"?")
(operateur,"(")
(identifiant,x)
(operateur,")")
(operateur,":")
(operateur,"(")
(identifiant,y)
(operateur,")")
(operateur,")")
(operateur,";")
(identifiant,plus)
(operateur,"(")
(operateur,"&")
(identifiant,z)
(operateur,")")
(operateur,";")
(operateur,"}")
(mot-cle,extern)
(type,void)
(identifiant,plus)
(operateur,"(")
(type,int)
(operateur,"*")
(operateur,")")
(operateur,";")
(mot-cle,main)
(operateur,"(")
(operateur,")")
(operateur,"{")
(type,int)
(identifiant,x)
(operateur,",")
(identifiant,y)
(operateur,",")
(identifiant,z)
(operateur,";")
(identifiant,x)
(operateur,"=")
(nombre,10)
(operateur,";")
(identifiant,y)
(operateur,"=")
(nombre,15)
(operateur,";")
```


Analyse (rapidement, l'analyse syntaxique)



Analyse (rapidement, l'analyse syntaxique)

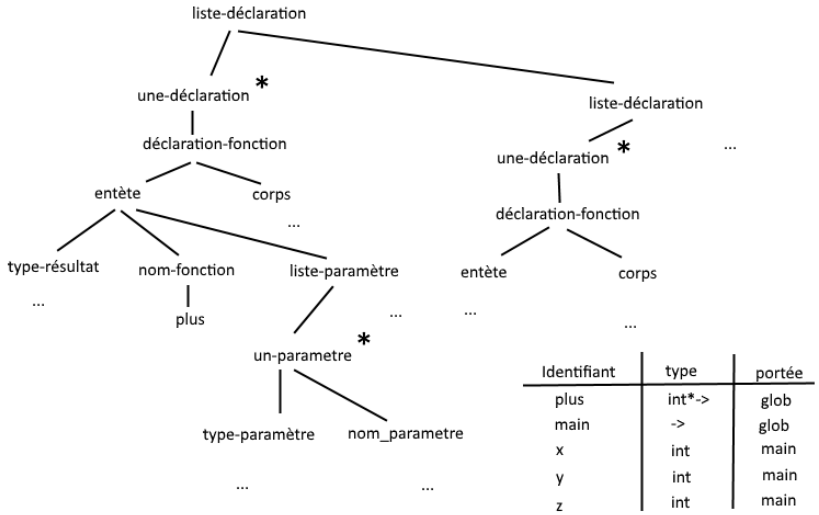
La syntaxe basée sur une grammaire

Exemple de règle de grammaire :

Affectation : := PartieGauche "=" Expression

Analyse (rapidement, l'analyse syntaxique)

L'arbre décoré + le dictionnaire des identifiants :



Identifiant	type	portée
plus	int*->	glob
main	->	glob
x	int	main
y	int	main
z	int	main

Un exemple en langage C

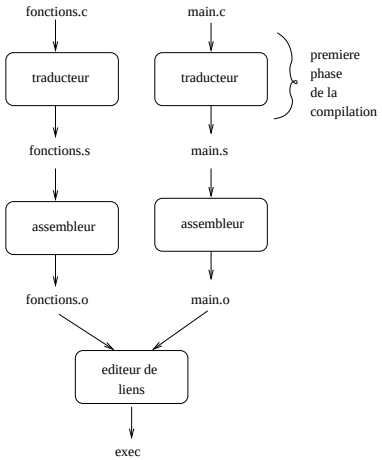
```
/* fichier fonctions.c */
 int somme (int *t, int n) {
 int i, s;
 s = 0;
 for (i=0;i<n;i++) s = s + t[i];
 return (s); }

 int max (int *t, int n) {
 int i, m;
 m = t[0];
 for (i=1;i<n;i++) if (m < t[i]) m = t[i];
 return (m); }
```

```
==================================
```

```
/* fichier main.c */
extern int somme (int *t, int n);
extern int max (int *t, int n);

#define TAILLE 10
static int TAB [TAILLE];

main () {
 int i,u,v;
 for (i=0;i<TAILLE;i++) scanf ("%d", &TAB[i]);
 u = somme (TAB, TAILLE);
 v = max (TAB, TAILLE); }
```

- Dans le fichier `main.c` les fonctions `somme` et `max` sont dites importées : elles sont définies dans un autre fichier.
- Dans le fichier `fonctions.c`, `somme` et `max` sont dites exportées : elles sont utilisables dans un autre fichier.

Compilation haut niveau



Remarque : la phase de traduction comporte une phase de pré-traitement dite phase de ≪ pré-compilation ≫ où le code source est transformé en code source ≪ enrichi ≫(les directives de pré-compilations — # — sont traitées)

- Pour ≪ compiler ≫, produire un exécutable, on enchaine les commandes :
  - `gcc -c fonctions.c`
  - `gcc -c main.c`
  - `gcc -o exec main.o fonctions.o`
- La commande `gcc -c main.c` produit un fichier appelé `main.o`.
- La commande `gcc -c fonctions.c` produit un fichier `fonctions.o`.
- Les fichiers `fonctions.o` et `main.o` contiennent du binaire translatable, c'est-à-dire, du code binaire qui ne peut pas directement être exécuté en mémoire.
- La commande `gcc -o exec main.o fonctions.o` produit le fichier `exec` qui contient du binaire exécutable. Ce fichier résulte de la liaison des deux fichiers objets (`.o`). On parle d'édition de liens.
- **Remarque :** `gcc` cache l'appel à différents outils (logiciels).

Exemple avec ARM : `essai.s` et `lib.s`

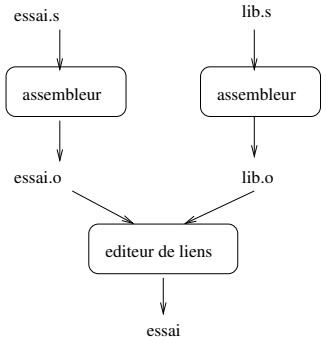
essai.s

```
    .text
    .global main
main:
      mov r0, #0
bcle: cmp r0, #10
      beq fin
       ldr r3, adr_xx
       ldr r2, [r3]
       bl add1
       str r2, [r3]
       add r0, r0, #1
       b bcle
fin:   b exit
adr_xx: .word xx
    .data
    .word 99
xx: .word 3
```

lib.s

```
    .text

    .global add1

add1 : add r2, r2, #1
       mov pc, lr
```

Compilation en assembleur



- Pour ≪ compiler ≫, on enchaine les commandes :
  - `arm-eabi-gcc -c essai.s`
  - `arm-eabi-gcc -c lib.s`
  - `arm-eabi-gcc -o essai essai.o lib.o`
- Les commandes `arm-eabi-gcc -c essai.s` et `arm-eabi-gcc -c lib.s` produisent les fichiers `essai.o` et `lib.o`.
- Les fichiers `essai.o` et `lib.o` contiennent du binaire translatable.
- La commande `arm-eabi-gcc -o essai essai.o lib.o` produit le fichier `essai` qui contient du binaire exécutable. Ce fichier résulte de la liaison des deux fichiers objets (`.o`). On parle d'édition de liens.
- **Remarque :** `arm-eabi-gcc` cache différents outils.
  - La commande `arm-eabi-gcc` appliqué à un fichier `.s` avec l'option `-c` correspond à la commande `arm-eabi-as`.
  - La commande `arm-eabi-gcc` avec l'option utilisée avec `-o` correspond à la commande d'édition de liens `arm-eabi-ld`.

Du langage C vers l'assembleur (ou un code intermédiaire)

- L'objectif d'un langage de haut niveau type langage C est de permettre au programmeur de s'abstraire des détails inhérents au fonctionnement de la machine.
- Il permet de manipuler des concepts bien plus élaborés.
- Mais il empêche la gestion de certains de ces détails.
- La première phase de la compilation consiste en la traduction systématique d'une syntaxe complexe en un langage plus simple et plus proche de la machine (langage machine ou code intermédiaire).

Exemple : traduction d'une conditionnelle

Prenons une conditionnelle :
`si Condition alors { ListeInstructions }`

elle sera traduite selon le schéma (récursif) :

```
etiq_debut:
            traduction(Condition) avec positionnement de ZNCV
            branch si (non vérifiée) a etiq_suite
            traduction(ListeInstruction)
etiq_suite:
```


Les schéma de traduction

Il y a ainsi des schémas (récursifs) de traduction prévus **pour toutes les règles de grammaire** décrivant les concepts du langage de programmation. Ces schémas sont définis pour un type de machine (large).

Exemples de schémas :

- pour l'évaluation d'opérateurs arithmétiques
- pour l'évaluation d'opérateurs relationnels
- pour l'affectation
- pour les structures de contrôle
- pour la définition de fonctions
- *etc.*

Problématique

L'objectif de l'assembleur est de produire du code binaire à partir du langage d'assemblage.

Il n'est pas toujours possible de produire du binaire qui puisse être directement copié en mémoire pour deux raisons principalement :

Cas 1 On ne connaît pas en général l'adresse à laquelle les zones **text** et **data** doivent être rangées en mémoire.

Cas 2 Le programme peut faire référence à des noms qui ne sont pas définis dans le fichier en cours de traduction.

- Dans le premier cas, on peut produire une **image** du binaire à partir de l'adresse 0, à charge du matériel de translater l'image à l'adresse de chargement pour l'exécution (il faut garder les informations permettant de savoir quelles sont les adresses à translater)
- Dans le deuxième cas on ne peut rien faire.

Premier cas

```
    .text
    .global main
main:
      mov r0, #0
bcle: cmp r0, #10
      beq fin
       ldr r3, adr_xx
       ldr r2, [r3]
       bl add1
       str r2, [r3]
       add r0, r0, #1
       b bcle
fin:   b exit
adr_xx: .word xx
    .data
    .word 99
xx: .word 3
```

L'adresse associée au symbole `xx` est : adresse de début de la zone `data` + 4 mais encore faut-il connaître l'adresse de début de la zone `data` !

Si on considère que la zone `data` est chargée à l'adresse **0**, l'adresse associée à `xx` est alors **4**. Si on doit translater le programme à l'adresse **2000**, il faut se rappeler que à l'adresse `adr_xx` on doit modifier la valeur **4** en **2000 + 4**.

Cette information à mémoriser est appelée une donnée de translation (**relocation** en anglais).

Deuxième cas

```
    .text
    .global main
main:
      mov r0, #0
bcle: cmp r0, #10
      beq fin
       ldr r3, adr_xx
       ldr r2, [r3]
       bl add1
       str r2, [r3]
       add r0, r0, #1
       b bcle
fin:   b exit
adr_xx: .word xx
    .data
    .word 99
xx: .word 3
```

- Dans le fichier `essai.o` il n'est pas possible de calculer le déplacement de l'instruction `bl add1` puisque l'on ne sait pas où est l'étiquette `add1` quand l'assembleur traite le fichier `essai.s`. En effet l'étiquette est dans un autre fichier : `lib.s`
- **Reprenons l'exemple en langage C.** Suite à la traduction en langage d'assemblage, dans le fichier `main.s` les références aux fonctions `somme` et `max` ne peuvent être complétées car les fonctions en question ne sont pas définies dans le fichier `main.c` mais dans `fonctions.c`.

Que contient un fichier .s ?

```
    .text
    .global main
main:
      mov r0, #0
bcle: cmp r0, #10
      beq fin
       ldr r3, adr_xx
       ldr r2, [r3]
       bl add1
       str r2, [r3]
       add r0, r0, #1
       b bcle
fin:   b exit
adr_xx: .word xx
    .data
    .word 99
xx: .word 3
```

- des directives : `.data`, `.bss`, `.text`, `.word`, `.hword`, `.byte`, `.skip`, `.asciz`, `.align`
- des étiquettes appelées aussi symboles
- des instructions du processeur
- des commentaires : `@ blabla`

Note : Parfois une directive (`.org`) permet de fixer l'adresse où sera logé le programme en mémoire. Cette facilité permet alors de calculer certaines adresses dès la phase d'assemblage.

Que contient un fichier .o ?

- un entête contenant la taille du fichier, les adresses des différentes tables, la taille de la zone de données non initialisées (`bss`), etc.
- la zone de données (`data`) parfois incomplète
- la zone des instructions (`text`) parfois incomplète
- les informations associées à chaque symbole rangées dans une section appelée : table des symboles.
- les informations permettant de compléter ce qui n'a pu être calculé... On les appelle informations de translation et l'ensemble de ces informations est rangé dans une section particulière appelée table de translation.
- une table des chaines à laquelle la table des symboles fait référence.

Exemple : essai.o, entête

On obtient l'entête avec la commande `arm-eabi-readelf -a essai.o`.

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 61 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            ARM
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          184 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           40 (bytes)
  Number of section headers:         9
  Section header string table index: 6
```


Exemple : essai.o, organisation des tables

On obtient l'entête avec la commande `arm-eabi-readelf -a essai.o`.

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 00002c 00  AX  0   0  1
  [ 2] .rel.text         REL             00000000 00033c 000018 08      7   1  4
  [ 3] .data             PROGBITS        00000000 000060 000008 00  WA  0   0  1
  [ 4] .bss              NOBITS          00000000 000068 000000 00  WA  0   0  1
  [ 5] .ARM.attributes   ARM_ATTRIBUTES  00000000 000068 000010 00      0   0  1
  [ 6] .shstrtab         STRTAB          00000000 000078 000040 00      0   0  1
  [ 7] .symtab           SYMTAB          00000000 000220 0000f0 10      8  12  4
  [ 8] .strtab           STRTAB          00000000 000310 000029 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```


Exemple : essai.o, zone data

On obtient la zone `.data` avec la commande `arm-eabi-objdump -s -j .data essai.o`.

```
essai.o:     format de fichier elf32-littlearm

Contenu de la section .data:
 0000 63000000 03000000
```

Exemple : essai.o, zone text

On obtient la zone `.text` avec la commande `arm-eabi-objdump -j .text -s essai.o`.

```
essai.o:     format de fichier elf32-littlearm

Contenu de la section .text:
 0000 0000a0e3 0a0050e3 0500000a 14309fe5
 0010 002093e5 fefffffeb 002083e5 010080e2
 0020 f7ffffea fefffffea 04000000
```


Exemple : essai.o, zone text

La zone `.text` avec désassemblage avec la commande `arm-eabi-objdump -j .text -d essai.o`.

```
Disassembly of section .text:

00000000 <main>:
   0: e3a00000  mov r0, #0

00000004 <bcle>:
   4: e350000a  cmp r0, #10
   8: 0a000005  beq 24 <fin>
   c: e59f3014  ldr r3, [pc, #20] ; 28 <adr_xx>
  10: e5932000  ldr r2, [r3]
  14: ebffffe  bl 0 <add1>
  18: e5832000  str r2, [r3]
  1c: e2800001  add r0, r0, #1
  20: eafffff7  b 4 <bcle>

00000024 <fin>:
  24: eafffffe  b 0 <exit>

00000028 <adr_xx>:
  28: 00000004  .word 0x00000004
```


Exemple : essai.o, table des symboles

On obtient l'entête avec la commande `arm-eabi-readelf -s essai.o`.

```
Symbol table '.symtab' contains 15 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 SECTION LOCAL  DEFAULT    1
     2: 00000000     0 SECTION LOCAL  DEFAULT    3
     3: 00000000     0 SECTION LOCAL  DEFAULT    4
     4: 00000000     0 NOTYPE  LOCAL  DEFAULT    1 $a
     5: 00000004     0 NOTYPE  LOCAL  DEFAULT    1 bcle
     6: 00000024     0 NOTYPE  LOCAL  DEFAULT    1 fin
     7: 00000028     0 NOTYPE  LOCAL  DEFAULT    1 adr_xx
     8: 00000028     0 NOTYPE  LOCAL  DEFAULT    1 $d
     9: 00000004     0 NOTYPE  LOCAL  DEFAULT    3 xx
    10: 00000000     0 NOTYPE  LOCAL  DEFAULT    3 $d
    11: 00000000     0 SECTION LOCAL  DEFAULT    5
    12: 00000000     0 NOTYPE  GLOBAL DEFAULT    1 main
    13: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND add1
    14: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND exit
```


Exemple : essai.o, table de translation

On obtient la table de translation avec la commande `arm-eabi-readelf -a essai.o`.

```
Relocation section '.rel.text' at offset 0x33c contains 3 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000014  00000d01 R_ARM_PC24        00000000   add1
00000024  00000e01 R_ARM_PC24        00000000   exit
00000028  00000202 R_ARM_ABS32       00000000   .data
```

Exemple : lib.o, organisation des tables

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 000008 00  AX  0   0  1
  [ 2] .data             PROGBITS        00000000 00003c 000000 00  WA  0   0  1
  [ 3] .bss              NOBITS          00000000 00003c 000000 00  WA  0   0  1
  [ 4] .ARM.attributes   ARM_ATTRIBUTES  00000000 00003c 000010 00      0   0  1
  [ 5] .shstrtab         STRTAB          00000000 00004c 00003c 00      0   0  1
  [ 6] .symtab           SYMTAB          00000000 0001c8 000070 10      7   6  4
  [ 7] .strtab           STRTAB          00000000 000238 000009 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```


Etapes d'un assembleur

1. Reconnaissance de la syntaxe (lexicographie et syntaxe) et repérage des symboles. Fabrication de la table des symboles utilisée par la suite dès qu'une référence à un symbole apparait.
2. Traduction = production du binaire.

Exemple : lib.o, tables des symboles

```
Symbol table '.symtab' contains 7 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 SECTION LOCAL  DEFAULT    1
     2: 00000000     0 SECTION LOCAL  DEFAULT    2
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 NOTYPE  LOCAL  DEFAULT    1 $a
     5: 00000000     0 SECTION LOCAL  DEFAULT    4
     6: 00000000     0 NOTYPE  GLOBAL DEFAULT    1 add1
```


Rôle de l'éditeur de liens

Le travail de l'éditeur de liens consiste à :

- Identifier les symboles définis et exportés d'un côté et les symboles non définis de l'autre (importés).
- Rassembler les zones de même type et effectuer les corrections nécessaires.

Remarque : L'édition de liens rassemble des fichiers objets. L'assembleur ne peut pas produire du binaire exécutable, il produit un binaire incomplet dans lequel il conserve des informations permettant de le compléter plus tard.
La phase d'édition de liens bien qu'elle permette de résoudre les problèmes de noms globaux produit elle aussi du binaire incomplet car les adresses d'implantation des zones `text` et `data` ne sont pas connues.

Phase de chargement : production de binaire exécutable

L'édition de liens peut être exécuter de façon statique ou de façon dynamique au moment ou on en a besoin.
Deux solutions dynamiques possibles :

- édition de liens au moment du chargement en mémoire (au lieu de rassembler le contenu de plusieurs fichiers, on ne charge que le code des fonctions utilisées, par ex. pour les bibliothèques) ou
- édition de liens au moment de l'exécution du code (appel de la fonction) ce qui permet de partager des fonctions et de ne pas charger en mémoire plusieurs fois le même code. (*e.g.*, les `DLLs` sous windows, *Dynamically Linked Libraries*)

Que contient un fichier exécutable ? entête

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 61 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            ARM
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x810c
  Start of program headers:          52 (bytes into file)
  Start of section headers:          144432 (bytes into file)
  Flags:                             0x2, has entry point, GNU EABI
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         24
  Section header string table index: 21
```


Que contient un fichier exécutable ? organisation des tables

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .init             PROGBITS        00008000 008000 000020 00  AX  0   0  4
  [ 2] .text             PROGBITS        00008020 008020 002500 00  AX  0   0  4
  [ 3] .fini             PROGBITS        0000a520 00a520 00001c 00  AX  0   0  4
  [ 4] .rodata           PROGBITS        0000a53c 00a53c 00000c 00   A  0   0  4
  [ 5] .eh_frame         PROGBITS        0000a548 00a548 00083c 00   A  0   0  4
  [ 6] .ctors            PROGBITS        00012d84 00ad84 000008 00  WA  0   0  4
  [ 7] .dtors            PROGBITS        00012d8c 00ad8c 000008 00  WA  0   0  4
  [ 8] .jcr              PROGBITS        00012d94 00ad94 000004 00  WA  0   0  4
  [ 9] .data             PROGBITS        00012d98 00ad98 00095c 00  WA  0   0  4
  [10] .bss              NOBITS          000136f4 00b6f4 000108 00  WA  0   0  4
  [11] .comment          PROGBITS        00000000 00b6f4 0001e6 00      0   0  1
  [12] .debug_aranges    PROGBITS        00000000 00b8e0 000350 00      0   0  8
  [13] .debug_pubnames   PROGBITS        00000000 00bc30 00069c 00      0   0  1
  [14] .debug_info       PROGBITS        00000000 00c2cc 00ea53 00      0   0  1
  [15] .debug_abbrev     PROGBITS        00000000 01ad1f 002956 00      0   0  1
  [16] .debug_line       PROGBITS        00000000 01d675 002444 00      0   0  1
  [17] .debug_str        PROGBITS        00000000 01fab9 001439 01  MS  0   0  1
  [18] .debug_loc        PROGBITS        00000000 020ef2 002163 00      0   0  1
  [19] .debug_ranges     PROGBITS        00000000 023058 0002e8 00      0   0  8
  [20] .ARM.attributes   ARM_ATTRIBUTES  00000000 023340 000010 00      0   0  1
  [21] .shstrtab         STRTAB          00000000 023350 0000df 00      0   0  1
  [22] .symtab           SYMTAB          00000000 0237f0 0013e0 10     23 213  4
  [23] .strtab           STRTAB          00000000 024bd0 0007d1 00      0   0  1
```


Que contient un fichier exécutable ? table des symboles

```
Symbol table '.symtab' contains 318 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00008000     0 SECTION LOCAL  DEFAULT    1
     2: 00008020     0 SECTION LOCAL  DEFAULT    2
     3: 0000a520     0 SECTION LOCAL  DEFAULT    3
 ......................
     9: 00012ea8     0 SECTION LOCAL  DEFAULT    9
 ......................
    74: 0000821c     0 NOTYPE  LOCAL  DEFAULT    2 bcle
    75: 0000823c     0 NOTYPE  LOCAL  DEFAULT    2 fin
    76: 00008240     0 NOTYPE  LOCAL  DEFAULT    2 adr_xx
    77: 00008240     0 NOTYPE  LOCAL  DEFAULT    2 $d
    78: 00012eb4     0 NOTYPE  LOCAL  DEFAULT    9 xx
 ......................
   230: 00008244     0 NOTYPE  GLOBAL DEFAULT    2 add1
 ......................
```

Que contient un fichier exécutable ? section data

```
Contents of section .data:
......................

 12ea8 00000000 00000000 63000000 03000000

......................
```


Que contient un fichier exécutable ? section text

```
00008218 <main>:
    8218: e3a00000  mov r0, #0

0000821c <bcle>:s
    821c: e350000a  cmp r0, #10
    8220: 0a000005  beq 823c <fin>
    8224: e59f3014  ldr r3, [pc, #20] ; 8240 <adr_xx>
    8228: e5932000  ldr r2, [r3]
    822c: eb000004  bl 8244 <add1>
    8230: e5832000  str r2, [r3]
    8234: e2800001  add r0, r0, #1
    8238: eafffff7  b 821c <bcle>

0000823c <fin>:
    823c: ea000007  b 8260 <exit>

00008240 <adr_xx>:
    8240: 00012eb4  .word 0x00012eb4

00008244 <add1>:
    8244: e2822001  add r2, r2, #1
    8248: e1a0f00e  mov pc, lr
```


Retour sur l'interprétation

Rappels : compilateurs et interpréteurs partagent la première étape de travail (étape d'analyse).

Compilateurs et interpréteurs se distinguent principalement au moment de la seconde étape (étape de synthèse) et de l'exécution :

- le code cible produit par un compilateur est exécuté directement par la machine cible ;
  à chaque exécution, c'est le même code produit qui est utilisé
- la structure intermédiaire (interne ou traduite en langage de bas niveau) obtenue par l'interpréteur est exécutée "dynamiquement" par l'interpréteur (comme sur une machine virtuelle) ;
  à chaque exécution, la production du code intermédiaire, la mise en place d'une machine virtuelle et l'exécution sur cette machine virtuelle sont renouvelées

Entre compilateurs et interpréteurs ?

A mi-chemin entre compilateurs et interpréteurs, d'autres organisations sont possibles :

- l'interpréteur peut conserver la mémoire de la production du code intermédiaire ou de l'arbre syntaxique correspondant au programme et de la mise en place d'une machine virtuelle
- certaines parties de code (bibliothèques, parties critiques, *etc.*) peuvent être compilées (statiquement ou dynamiquement), d'autres interprétés

Il y a une grande diversité d'organisations possibles intermédiaires.

Mode d'exécution des langages les plus utilisés

Parmi les langages les plus utilisés (*cf.* Tiobe Jan 18), lesquels peuvent-être compilés, lesquels peuvent-être interprétés :

- le langage `Java`
- le langage `C`
- le langage `C++`
- le langage `C#`
- le langage `Python`
- le langage `PHP`
- le langage `Javascript`

D'autres langages « interprétés » importants : `SQL`, `PostScript`.

Avantages, inconvénients

Pourquoi, comment, choisir entre compilation et interprétation ? Arguments classiques (sur les performances, la qualité du code, la productivité du programmeur) :

- l'exécution par interprétation est plus lente (surtout, si l'on ne compte pas le temps de compilation)
- l'exécution par interprétation facilite la mise au point (temps morts réduits, possibilité d'exécution de code incomplets, changement à chaud, débuggage dynamique, exécution interactive, *etc.*)
- l'exécution par interprétation facilite le portage (dans la mesure où il y a un interpréteur portable lui aussi ; pour la compilation, il faut non seulement un compilateur portable mais aussi une compilation [sans erreur] pour arriver au même résultat)

Avantages, inconvénients (suite)

Pourquoi, comment, choisir entre compilation et interprétation ?

- l'exécution par compilation améliore la sécurité (les fonctions « `eval` » n'existent souvent que dans les formes interprétées d'exécution, et l'exécution de code dynamique est une source importante de trous de sécurité, *e.g.* injection de code)
- les langages orientés interprétation ont plus souvent un typage dynamique, une gestion dynamique de la mémoire, *etc.* souvent plus « haut niveau » (moins de sécurité, mais plus grande productivité du programmeur)
- l'exécution par interprétation facilite les démarches « open source » (meilleur qualité de code, meilleur productivité, mais possible frein à la rentabilité)

Avantages, inconvénients (fin)

Pourquoi, comment, choisir entre compilation et interprétation ?

- l'exécution par interprétation nécessite le code du programme et le code de l'interpréteur, chacun peut évoluer séparément.
- au début de l'informatique, et chaque fois que l'informatique se déploie sur un nouveau dispositif aux ressources limitées, la compilation peut demander moins de ressource (sur le dispositif) et représenter la seule solution possible.

Le cas Java

Java est un langage semi-interprété :

- la première phase de la vie d'un programme java `monProg.java` consiste en une ≪ compilation ≫(`javac`) produisant un code intermédiaire (le java bytecode) `monProg.class`
- la seconde phase de la vie d'un programme java consiste en l'interprétation (`java`) du code intermédiaire `monProg.class` par la `JVM` (`Java Virtual Machine`)

`Java` n'est pas tout seul (`Python`, `C#`, `ProLog`, `Lisp`, `Scheme`, `Ocaml`, `Perl`, `Ruby`, `Erlang`, `Lua`, *etc.*)

Organisation Interne d'un ordinateur

Exécution des programmes en langage machine

Denis Bouhineau Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

15 janvier 2020

Remarque finale

A propos de bootstrap : un interpréteur demande toujours au départ un code exécutable (non interprété, même parfois produit ≪ à la main ≫, sans compilation) jusqu'au moment où l'interpréteur est le code exécuté par lui-même, on parle alors d'**auto-interprétation**.

Plan

Organisation de l'ordinateur

Il existe différentes sortes d'ordinateurs :

- Les *PC*
- Les ordinateurs portables
- Les serveurs
- Les super-calculateurs
- Les téléphones portables
- Dans l'électro-ménager, . . .

On parle souvent de systèmes embarqués **pour désigner un ordinateur qui ne ressemble pas à un ordinateur !** C'est-à-dire, pas de clavier, pas de souris, pas de disque, pas d'écran, mais un processeur avec un programme

Exemple : Le contrôleur de distribution d'essence d'une station service.

Points communs

Il y a des points communs :

- Processeur
- Mémoire
- Bus adresses, données
- Signaux de contrôle (*Read*/$\overline{\textit{Write}}$, autres),
- . . .
- Contrôleurs d'entrées-sorties
- Décodeur

Critères de classification

- Peut-on ajouter des programmes et les lancer ? (cartouche de jeu, par exemple ...)
- Peut-on ajouter des programmes en langage machine écrits, compilés et assemblés soi-même ?
- Y-a-t-il des mémoires secondaires ? (disque dur, clé USB, CD-Rom, . . .)

Vision externe (exemple)

- Un simple circuit avec 4 registres qui fait l'interface entre le périphérique et la machine (niveau OS)
- Le circuit est vu par le processeur (espace d'adressage) comme 4 mots :
  - `Mcommande` (un mot de commande)
  - `Métat` (un mot d'état)
  - `Mdonnéessort` (données sortantes)
  - `Mdonnéesentr` (données entrantes)
- Géré par un logiciel : le pilote

Lectures/écriture par le processeur (via le pilote)

Ecriture Lors d'une écriture (instruction `STORE`) dans un des trois registres `Mcommande, Métat, Mdonnéessort`, le processeur envoie la donnée à écrire dans le bus de donnée puis la donnée est transférée du bus de donnée au registre du circuit contrôleur d'Entrées-Sorties.

Lecture Lors d'une lecture (instruction `LOAD`) dans le registre `Mdonnéesentr` ou dans le registre `Métat`, le processeur récupère le contenu du registre `Mdonnéesentr` (resp. `Métat`) via le bus de donnée.

Les 4 mots (`Mcommande, Métat, Mdonnéessort, Mdonnéesentr`) seront supposés aux adresses CNTRL, CNTRL+1, +2, +3.

Attention : `CNTRL` est une adresse constante, pas déterminée par l'assembleur. Elle a un sens physique qu'on va voir maintenant...

Notations

Commande	**Signification**
`LOAD` reg, `Métat`	METTRE l'adresse `CNTRL` dans un registre `regad` puis `LOAD` reg, `[regad+1]`
`STORE` reg, `Mdonnéessort`	METTRE l'adresse `CNTRL` dans un registre `regad` puis `STORE` reg, `[regad+2]`

Utilisation du contrôleur d'entrées-sorties : exemple

Dans la documentation technique d'un contrôleur, certaines valeurs sont prédéfinies.

Considérons ici l'exemple (simpliste) d'un contrôleur graphique avec les valeurs prédéfinies suivantes :

- *libre* : cette valeur **dans le mot d'état** signifie que le contrôleur est prêt à accepter une commande. Le contrôleur tient à jour son état de disponibilité pour que le processeur puisse savoir si il est occupé ou non.
- *fond_écran* : cette valeur, **envoyée vers le mot de commande**, affiche un fond d'écran de la couleur contenue dans le registre de données sortantes.
- *rouge* : code de la couleur rouge, **peut être contenu dans le registre de données sortantes**.

Exercice

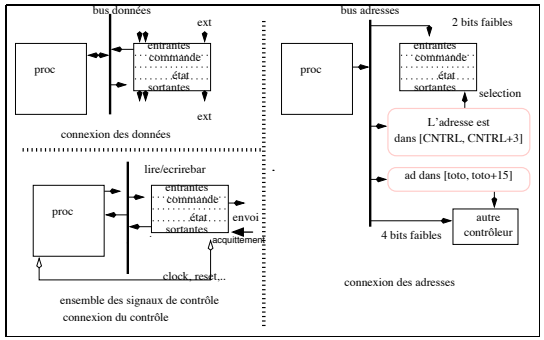
Ecrire un programme qui change la couleur de fond d'écran en rouge via le contrôleur graphique.

Remarque : On aurait de même des actions d'envoi :

- d'un caractère à l'écran,
- de positionnement de la tête de lecture d'un disque dur,
- ...

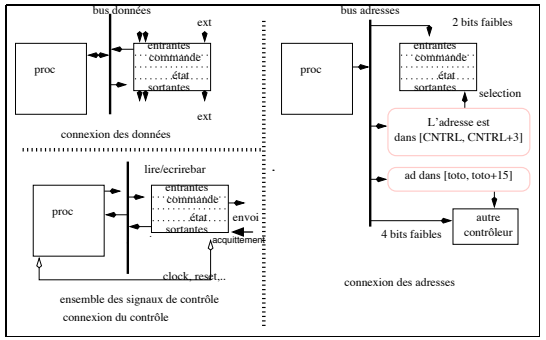
Les contrôleurs réels sont parfois très simples, parfois très complexes (simple affichage comme ci-dessus, contrôleur réseau...) parfois pour une seule commande, plusieurs données...

Etude du matériel d'entrées-sorties : *les entrées*



- bus données (lié au processeur)
- deux bits de bus adresses (pour sélectionner l'un des 4 mots `CNTRL` +0, +1, +2 ou +3)
- un signal de sélection provenant du décodeur d'adresses
- le signal *Read*/$\overline{\textit{Write}}$ du processeur
- un paquet de données (8 fils) venant du monde extérieur. Disons pour simplifier 8 interrupteurs
- le signal d'horloge (par exemple le même que le processeur). On peut raisonner comme si, à chaque front de l'horloge la valeur venant des interrupteurs était échantillonnée dans le registre `Mdonnéesentr`.
- une entrée ACQUITTEMENT si c'est un contrôleur de sortie.

Etude du matériel d'entrées-sorties : *les sorties*



- Il délivre sur le bus données du processeur le contenu du registre `Mdonnéesentr` si il y a sélection, lecture et adressage de `Mdonnéesentr`, c'est-à-dire si le processeur exécute une instruction `LOAD` à l'adresse `CNTRL` +3
- Il délivre sur le bus données du processeur le contenu du registre `Métat` si il y a sélection, lecture et adressage de `Métat`, c'est-à-dire si le processeur exécute une instruction `LOAD` à l'adresse `CNTRL` +1.
- On peut raisonner comme si le contenu du registre `Mdonnéessort` était affiché en permanence sur 8 pattes de sorties vers l'extérieur (8 diodes, par exemple).
- Une sortie ENVOI si c'est un contrôleur de sortie.

Exemple

00	`selrom`
01	`selrom`
02	`selrom`
...	
3E	`selrom`
3F	`selrom`
40	/////
...	
57	/////
58	`selcntr`
59	`selcntr`
5A	`selcntr`
5B	`selcntr`
5C	/////
...	
5F	/////
60	/////
...	
7E	/////
7F	/////
80	`selram`
81	`selram`
...	
FE	`selram`
FF	`selram`

- On raisonne avec des adresses sur 8 bits.
- L'adresse A est codé sur 8 bits $A7,\ldots,A0$.
- On représente les valeurs portées par ces 8 fils par deux chiffres hexadécimaux.
- On note val_x une valeur représentée en hexa.
- On dira que l'entrée $A \ll$ vaut $\gg 67_x$ pour dire que les 8 fils sont dans les états 0110 0111.

Sélecteur pour la ROM

Un mot (lu ou écrit) est pris dans la mémoire morte (ROM) si et seulement si :

Le boîtier physique de ROM, qui contient 64 mots reçoit deux types d'information :

Sélecteur pour le contrôleur d'entrées-sorties

Le mot (lu ou écrit) est pris dans le contrôleur d'entrées-sorties (vu précédemment) si et seulement si :

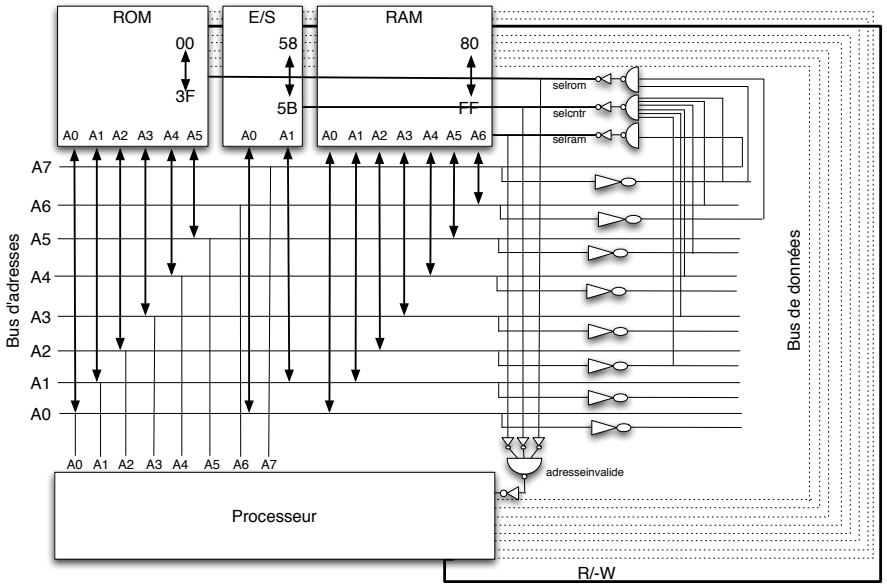
Le boîtier physique de contrôleur, qui contient 4 mots, reçoit deux types d'information :

Sélecteur pour la RAM

Le mot (lu ou écrit) est pris dans la mémoire vive (RAM) si et seulement si :

Le boîtier physique de RAM, qui contient 128 mots, reçoit deux types d'information :

Connexions processeur/contrôleur/mémoires/décodeur



Pour aller plus vite ...

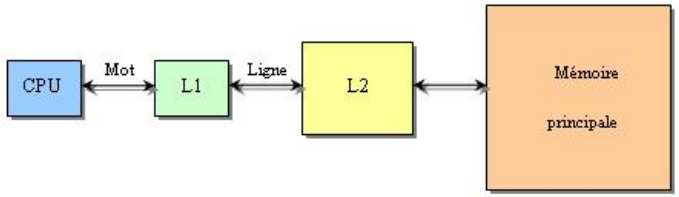
La mémoire est lente, chère et prend de la place !

Mais, principes de localité :

- Principe de localité spatiale :
  - l'accès à une donnée X va probablement être suivi d'accès à d'autres données Y, Z proches de X.
  - **exemple :** instructions, éléments de tableau.
- Principe de localité temporelle :
  - l'accès à une donnée X à un instant donné va probablement être suivi d'autres accès à cette même donnée.

Organisation mémoire

Structuration en caches de plusieurs niveaux



source wikipedia.

Pour le processeur, il n'y a qu'une mémoire : la Mémoire.

Vie du cache

Le cache répond aux demandes du processeur, **en lecture** :

- si la donnée est disponible : ok
- si la donnée n'est pas disponible
  - il faut aller la chercher en mémoire
  - faire de la place dans le cache (quelle autre donnée enlever ?)

Vie du cache

Le cache répond aux demandes du processeur, **en écriture** :

- si la donnée est disponible : ok
- si la donnée n'est pas disponible
  - il faut aller la chercher en mémoire
  - faire de la place dans le cache (quelle autre donnée enlever ?)

puis faire les modifications

- Écriture immédiate en mémoire ?
- Écriture différée ?

Cas particuliers : Données *vs* Programme

- **Données :**
  - modifiable
  - principe de localité respecté pour toutes les variables dans la pile (même lors des appels de fonction)
- **Programme :**
  - non modifiable (en général)
  - principe de localité rompu lors des appels de fonction

Conclusion : séparation des caches Données - Programme

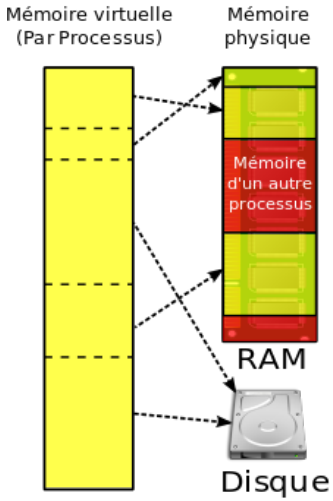
Pour aller plus loin ...

Pour le processeur, la mémoire n'est pas idéale ...

- La mémoire physique réelle (RAM) est trop petite
  - alors que la mémoire secondaire (HD) est immense
- Chaque processus voudrait avoir la mémoire à lui tout seul, identique d'une fois sur l'autre
- Il faudrait pouvoir partager certaines zones de la mémoire
- Il faudrait pouvoir protéger certaines zones de la mémoire

Extension de la mémoire physique

La mémoire vue du processeur
et en vrai ... (source wikipedia)



Partage de la mémoire et protection

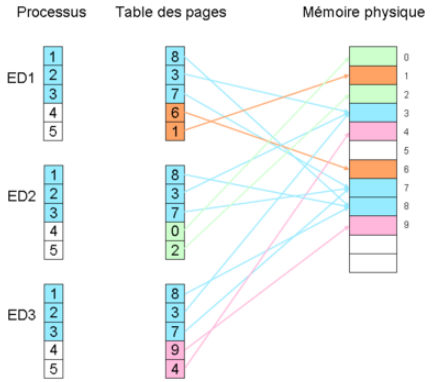
Mécanisme de pagination

- La mémoire réelle est organisée en pages
- Une adresse mémoire virtuelle (idéale) est associé à un couple **[numéro de page, adresse dans la page]**
- Les pages peuvent être protégées
- Les pages peuvent être partagées

Illustration

Partage du code entre trois processus exécutant le même programme, chacun sur des données particulières.

- Le programme est constitué de 3 pages de code
- Le programme comporte 2 pages de données



source wikipedia.

Remarque finale

Une partie de ce cours était à la frontière entre Architecture et Système.

- **Architecture :** parce qu'il y a des circuits impliqués
- **Système :** pour certains algorithmes (allocation) et concepts (processus)

Plan

Transistor, Portes Logiques et Circuits Combinatoires

Exécution des programmes en langage machine

Stéphane Devismes Fabienne Carrier

Université Grenoble Alpes

15 janvier 2020

Définition

Le transistor est le composant électronique actif fondamental en électronique utilisé principalement comme

- **interrupteur commandé** et
- pour l'**amplification**, mais aussi
- pour **stabiliser une tension**, **moduler un signal**
- ainsi que de nombreuses autres utilisations.

Type de transistors

Il y a deux principaux types de transistors :

- Le transistor à jonctions (technologie dite bipolaire).

  Dans cette famille, on trouve notamment les circuits **RTL** (*Resistor Transistor Logic*), **TTL** (*Transistor Transistor Logic*), **DTL** (*Diode Transistor Logic*) ou encore **ECL** (*Emitter Coupled Logic*).

- Le transistor à effet de champ (technologie dite unipolaire).

  Dans cette famille, on trouve notamment les circuits **MOS** (*Metal Oxyde Semiconductor*) qui sont en général moins rapides, sauf les plus récents comme **HMOS** ou **XMOS3**.

Remarque : Aujourd'hui, tous les processeurs sont en technologie **MOSFET**, ou des améliorations.

Définition

La forme la plus élémentaire de **circuit** est la porte logique.

Son comportement est dit **binaire** car il est caractérisé par deux états : l'état 0, qui représente la valeur logique faux et l'état 1, qui représente la valeur logique vrai.

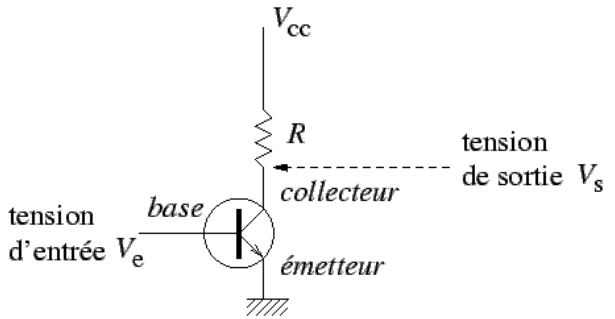
Les constructeurs utilisent dans certains cas une logique dite **positive** : l'état 1 correspond à une tension comprise entre 2 et 5 volts (niveau haut) et l'état 0 à une tension comprise entre 0 et 1 volt (niveau bas).

Dans d'autres cas, ils utilisent une logique dite **négative** où l'état 1 correspond au niveau bas, tandis que l'état 0 correspond au niveau haut.

Remarque : La transmission n'est pas instantanée : le délai de traversée d'une porte correspond au temps de propagation des signaux de l'entrée vers la sortie (de l'ordre de quelques dixième nanosecondes pour la porte la plus simple).

Fonctionnement d'un transistor

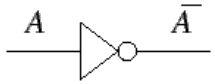
Un transistor fonctionne comme un **interrupteur** (ici, un transistor *à jonction NPN*).



- Si la tension d'entrée V_e est inférieure à une valeur critique, le transistor se bloque et n'est pas conducteur ; l'interrupteur est ouvert et la tension de sortie V_s est proche de V_{cc}, donc à un niveau haut.
- Si la tension d'entrée V_e est supérieure à la valeur critique, le transistor bascule, ce qui le rend conducteur ; l'interrupteur est fermé et la tension de sortie est au niveau bas, proche de la masse. La résistance R est conçue pour limiter le courant à travers le transistor, dans ce dernier cas.

Porte NON

La porte logique la plus simple est celle qui réalise une inversion logique. Elle est symbolisée par :



Elle est construite avec un seul transistor :

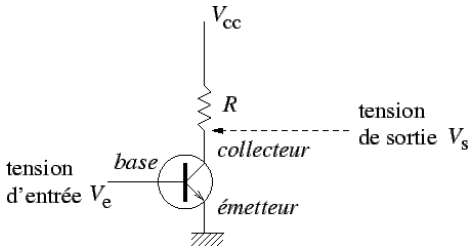


Table de vérité

V_e	V_s
0	1
1	0

Porte NON-ET

La porte logique du NON-ET :

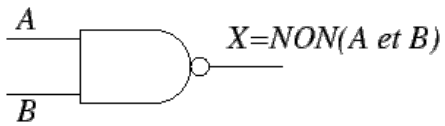


Table de vérité

V_1	V_2	V_s
0	0	1
0	1	1
1	0	1
1	1	0

Porte NON-OU

La porte logique du NON-OU :

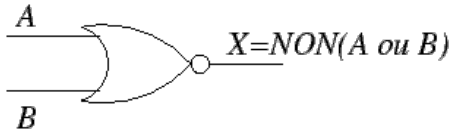


Table de vérité

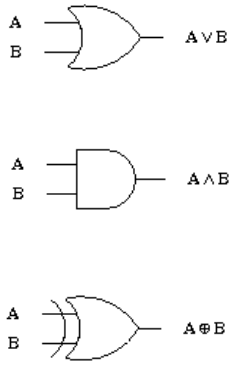
V_1	V_2	V_s
0	0	1
0	1	0
1	0	0
1	1	0

Fonctions booléennes

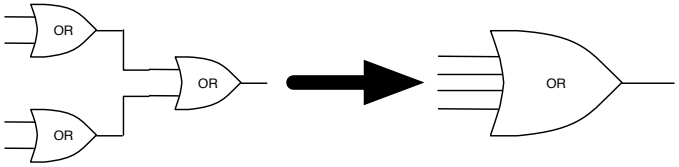
Avec les trois portes **NON, NON-ET et NON-OU** ont obtient une logique complète : toute fonction booléenne peut être décrite avec ces opérateurs !

Exemple : Pour obtenir un **OU** il suffit de relier un **NON-OU** et un **NON**.

Autres symboles



Simplifications



- Possible car commutativité et associativité
- Pour n'importe quel nombre d'entrées ≥ 2
- Possible aussi pour **et, non et, non ou**

Définition

Un circuit combinatoire est un circuit comportant **des entrées** et **des sorties** booléennes et dont les sorties sont une expression logique des valeurs d'entrées.

Principes des circuits intégrés

Un circuit intégré est une plaquette de silicium sur laquelle sont intégrées les portes du circuit. La plaquette est encapsulée dans un boîtier avec sur les côtés des broches permettant les connexions électriques.

Ces circuits sont classés suivant la densité d'intégration, c'est-à-dire le nombre de portes ou transistors par circuits (ou par mm^2) :

SSI	*Small Scale Integration*	1 à 10 portes par circuit
MSI	*Medium Scale Integration*	10 à 100
LSI	*Large Scale Integration*	100 à 100 000
VLSI	*Very Large Scale Integration*	plus de 100 000

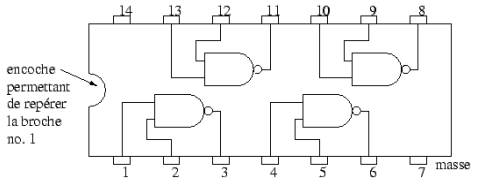


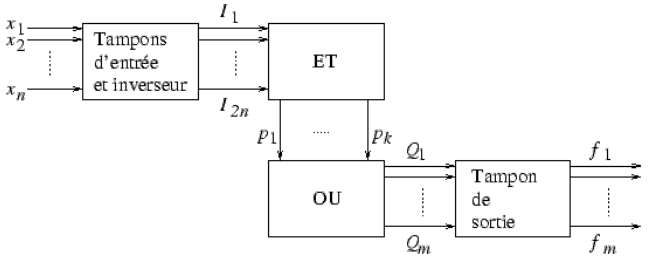
FIGURE – Un circuit SSI dans un boîtier à 14 broches : circuit TTL 7400 de Texas Instruments.

Circuit FPGA

Ce sont des réseaux logiques programmables (**Field Programmable Logic Array** ou **Field Programmable Gate Array**) qui permettent de réaliser des fonctions lorsqu'elles sont sous la forme d'une somme de produits.

Considérons par exemple 20 broches dont 12 correspondent à des entrées et 6 à des sorties. Les 12 entrées sont inversées, ce qui fournit 24 variables internes. Ces 24 variables sont toutes connectées à 50 portes ET, ce qui donne 1 200 fusibles, au départ intacts.

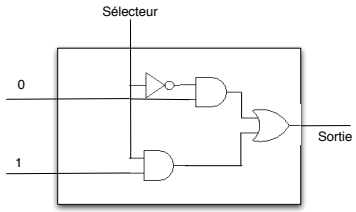
Programmer le circuit consiste alors à détruire certains fusibles pour obtenir les produits de la fonction à programmer. Les 6 sorties proviennent de 6 portes OU qui reçoivent chacune les sorties des 50 portes ET. On obtient ainsi 300 fusibles dont certains sont détruits pour obtenir la somme des termes de la fonction.



Multiplexeur

Un multiplexeur comporte 2^n entrées, 1 sortie et n lignes de sélection. La configuration des n lignes de sélection fournit une valeur parmi les 2^n entrées et connecte cette entrée à la sortie.

Exemple : $n = 1$



Décodeur

Un décodeur comprend n entrées et 2^n sorties, la sortie activée correspondant à la configuration binaire du mot formé par les n entrées. Un tel circuit sert à sélectionner des adresses de la mémoire.

Exemple : Un décodeur 3 vers 8 ($n = 3$).

Comparateur

Un comparateur à $2 \times n$ entrées et 1 sortie, les $2 \times n$ entrées formant deux mots de n bits : A et B. La sortie vaut 1 si le mot $A = B$, 0 sinon.

Exemple : $n = 2$

Décaleur

Un décaleur est formé de $(n+1)$ entrées $D_1, ..., D_n$, C et de n sorties $S_1, ..., S_n$ et opère un décalage de 1 bit sur les entrées $D_1, ..., D_n$. Si $C = 1$, il s'agit d'un décalage à droite et si $C = 0$, d'un décalage à gauche.

Exemple : $n = 3$

Demi-additionneur

Un demi-additionneur additionne deux bits (A et B) et évalue la retenue éventuelle (R).

A	B	$S = A + B$	R
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Additionneur

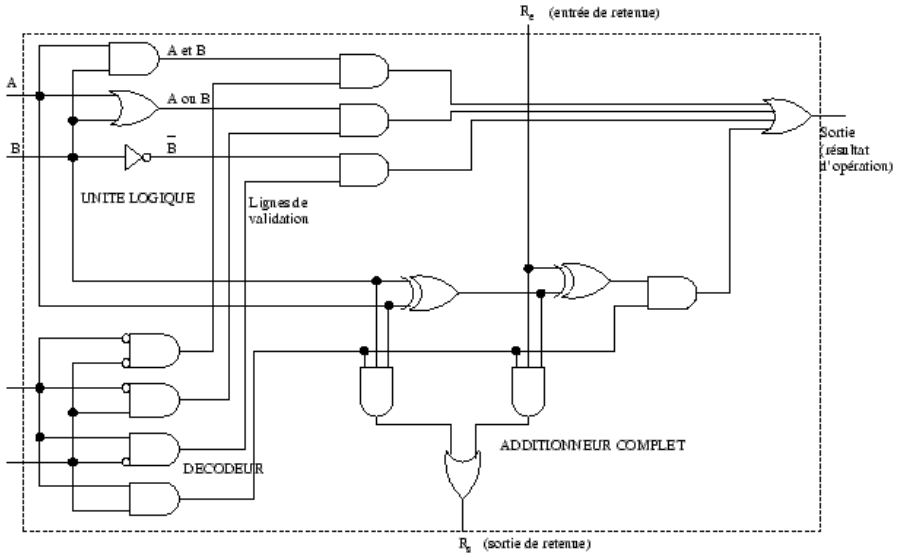
Un additionneur additionne deux bits (A et B) et une retenue (R_e) et évalue la retenue éventuelle (R_s).

A	B	R_e	$S = A + B$	R_s
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Du demi-additionneur à l'additionneur

Addition de mots (sur 4 bits)

UAL



Conclusion

- Avec **des transistors** on fait **des portes logiques**
- Avec **des portes logiques** on fait **des circuits combinatoires**
- Avec **des circuits combinatoires** on fait **des UALs, des unités de commandes. . .**
- Mais pour faire un processeur, il faut aussi faire des registres
- Il faut aussi pouvoir faire de la mémoire

Solution : circuit séquentiel, mais c'est une autre histoire . . .