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

Denis Bouhineau Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

22 mars 2020

Plan

Plan

Aujourd'hui

- Aujourd'hui nous allons étudier comment un processeur exécute un programme.

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.

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 (@)

Plan

Structure du Processeur vu du programmeur

Structure du Processeur vu du programmeur

- On considère un processeur comportant un seul registre de données directement visible par le programmeur : `ACC` (pour accumulateur).

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**.

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 ?

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 ?

La taille d'une adresse est 4 bits. D'où la taille de la mémoire : 16 données de 4 bits (adresses de 0000 à 1111...), *i.e.*, 64 bits

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
```

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 (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`.

0	2	load #3
1	3	
2	3	store 8
3	8	
et=4	5	add 8
5	8	
6	4	jmp et = jmp 4
7	4	
8		

Exemple de programme (2/2)

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

Exemple de programme (2/2)

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

Que calcule ce programme ?

Exemple de programme (2/2)

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

Que calcule ce programme ?

$ACC \leftarrow 3$	
$mem[8] \leftarrow ACC$	c'est-à-dire 3
$ACC \leftarrow mem[8] + ACC$	c'est-à-dire 6
	et recommencer
$ACC \leftarrow mem[8] + ACC$	c'est-à-dire $3 + 6 = 9$
	etc.

Plan

Interprétation des instructions ? (sémantique opérationnelle)

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

Interprétation des instructions ? (sémantique opérationnelle)

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

Reprenons l'exemple précédent :

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.

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.

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`.

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.

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
```

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.

Exécution du programme

Exécution du programme

$pc \leftarrow 0$

Exécution du programme

$pc \leftarrow 0$

$mem[0] = 2$ ($load$) $\quad acc \leftarrow 3$ ($mem[1]$) $\quad pc \leftarrow 2$

Exécution du programme

$pc \leftarrow 0$

$mem[0] = 2$ ($load$) $\quad acc \leftarrow 3$ ($mem[1]$) $\quad pc \leftarrow 2$

$mem[2] = 3$ ($store$) $\quad mem[mem[3] = 8] \leftarrow 3$ (acc) $\quad pc \leftarrow 4$

Exécution du programme

$pc \leftarrow 0$

$mem[0] = 2$ (*load*)	$acc \leftarrow 3$ ($mem[1]$)	$pc \leftarrow 2$
$mem[2] = 3$ (*store*)	$mem[mem[3] = 8] \leftarrow 3$ (acc)	$pc \leftarrow 4$
$mem[4] = 5$ (*add*)	$acc \leftarrow 3$ (acc) $+ 3$ ($mem[mem[5] = 8]$)	$pc \leftarrow 6$

Exécution du programme

$pc \leftarrow 0$

$mem[0] = 2$ (*load*)	$acc \leftarrow 3$ ($mem[1]$)	$pc \leftarrow 2$
$mem[2] = 3$ (*store*)	$mem[mem[3] = 8] \leftarrow 3$ (acc)	$pc \leftarrow 4$
$mem[4] = 5$ (*add*)	$acc \leftarrow 3$ (acc) $+ 3$ ($mem[mem[5] = 8]$)	$pc \leftarrow 6$
$mem[6] = 4$ (*jmp*)	$pc \leftarrow 4$ ($mem[7]$)	

Exécution du programme

$pc \leftarrow 0$

$mem[0] = 2\ (load)$	$acc \leftarrow 3\ (mem[1])$	$pc \leftarrow 2$
$mem[2] = 3\ (store)$	$mem[mem[3] = 8] \leftarrow 3\ (acc)$	$pc \leftarrow 4$
$mem[4] = 5\ (add)$	$acc \leftarrow 3\ (acc) + 3\ (mem[mem[5] = 8])$	$pc \leftarrow 6$
$mem[6] = 4\ (jmp)$	$pc \leftarrow 4\ (mem[7])$	
$mem[4] = 5\ (add)$	$acc \leftarrow 6\ (acc) + 3\ (mem[mem[5] = 8])$	$pc \leftarrow 6$

Exécution du programme

$pc \leftarrow 0$

$mem[0] = 2\ (load)$	$acc \leftarrow 3\ (mem[1])$	$pc \leftarrow 2$
$mem[2] = 3\ (store)$	$mem[mem[3] = 8] \leftarrow 3\ (acc)$	$pc \leftarrow 4$
$mem[4] = 5\ (add)$	$acc \leftarrow 3\ (acc) + 3\ (mem[mem[5] = 8])$	$pc \leftarrow 6$
$mem[6] = 4\ (jmp)$	$pc \leftarrow 4\ (mem[7])$	
$mem[4] = 5\ (add)$	$acc \leftarrow 6\ (acc) + 3\ (mem[mem[5] = 8])$	$pc \leftarrow 6$

...

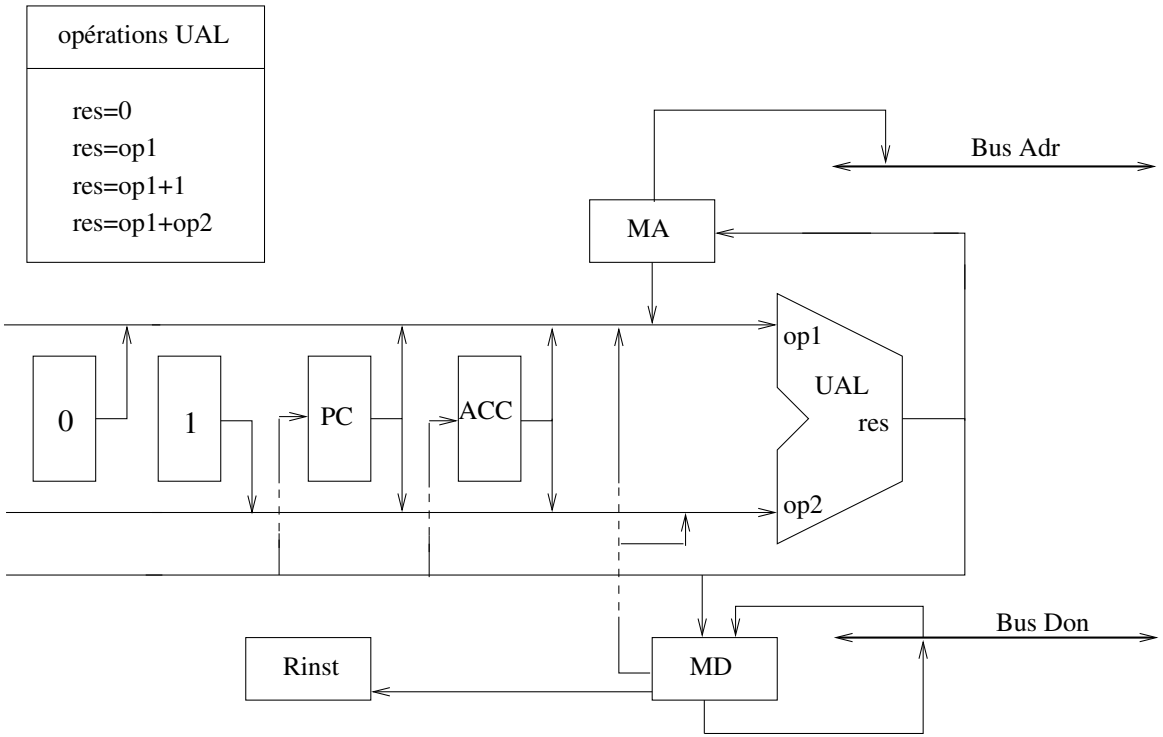
Plan

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.

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

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.

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),...

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 :

Micro-actions et micro-conditions

On fait des hypothèses FORTES sur les transferts possibles :

md ← **mem[ma]**	lecture d'un mot mémoire.	C'est la seule possibilité en lecture !
mem[ma] ← **md**	écriture d'un mot mémoire	C'est la seule possibilité en écriture !

Micro-actions et micro-conditions

On fait des hypothèses FORTES sur les transferts possibles :

md ← **mem[ma]**	lecture d'un mot mémoire.	C'est la seule possibilité en lecture !
mem[ma] ← **md**	écriture d'un mot mémoire	C'est la seule possibilité en écriture !
rinst ← **md**	affectation	C'est la seule affectation possible dans `rinst`

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`
reg$_0$ $\leftarrow$ **0**	affectation	reg$_0$ est `pc`, `acc`, `ma`, ou `md`

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`
reg$_0$ $\leftarrow$ **0**	affectation	reg$_0$ est `pc`, `acc`, `ma`, ou `md`
reg$_0$ $\leftarrow$ **reg**$_1$	affectation	reg$_0$ est `pc`, `acc`, `ma`, ou `md` reg$_1$ est `pc`, `acc`, `ma`, ou `md`

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`

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)

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.

Plan

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 d'un clear

Prochaine instruction en assembleur : `clear`
Exécution :

- lecture de l'instruction, positionne l'adresse : ma $\leftarrow$ pc

Exécution d'un clear

Prochaine instruction en assembleur : `clear`
Exécution :

- lecture de l'instruction, positionne l'adresse : ma $\leftarrow$ pc
- lecture de l'instruction, récupère l'instruction : md $\leftarrow$ Mem[ma]

Exécution d'un clear

Prochaine instruction en assembleur : `clear`
Exécution :

- lecture de l'instruction, positionne l'adresse : ma $\leftarrow$ pc
- lecture de l'instruction, récupère l'instruction : md $\leftarrow$ Mem[ma]
- lecture de l'instruction, sauvegarde : rinst $\leftarrow$ md

Exécution d'un clear

Prochaine instruction en assembleur : `clear`
Exécution :

- lecture de l'instruction, positionne l'adresse : ma $\leftarrow$ pc
- lecture de l'instruction, récupère l'instruction : md $\leftarrow$ Mem[ma]
- lecture de l'instruction, sauvegarde : rinst $\leftarrow$ md
- clear : acc $\leftarrow$ 0

Exécution d'un clear

Prochaine instruction en assembleur : `clear`
Exécution :

- lecture de l'instruction, positionne l'adresse : ma $\leftarrow$ pc
- lecture de l'instruction, récupère l'instruction : md $\leftarrow$ Mem[ma]
- lecture de l'instruction, sauvegarde : rinst $\leftarrow$ md
- clear : acc $\leftarrow$ 0
- avance la lecture : pc $\leftarrow$ pc + 1

Exécution d'un clear

Prochaine instruction en assembleur : `clear`
Exécution :

- lecture de l'instruction, positionne l'adresse : ma $\leftarrow$ pc
- lecture de l'instruction, récupère l'instruction : md $\leftarrow$ Mem[ma]
- lecture de l'instruction, sauvegarde : rinst $\leftarrow$ md
- clear : acc $\leftarrow$ 0
- avance la lecture : pc $\leftarrow$ pc + 1
- instruction suivante

Exécution d'une lecture

Prochaine instruction en assembleur : `load #8`

Exécution d'une lecture

Prochaine instruction en assembleur : `load #8`
Exécution :

- lecture de l'instruction, positionne l'adresse : ma $\leftarrow$ pc

Exécution d'une lecture

Prochaine instruction en assembleur : `load #8`
Exécution :

- lecture de l'instruction, positionne l'adresse : ma $\leftarrow$ pc
- lecture de l'instruction, récupère l'instruction : md $\leftarrow$ Mem[ma]

Exécution d'une lecture

Prochaine instruction en assembleur : `load #8`
Exécution :

- lecture de l'instruction, positionne l'adresse : ma $\leftarrow$ pc
- lecture de l'instruction, récupère l'instruction : md $\leftarrow$ Mem[ma]
- lecture de l'instruction, sauvegarde : rinst $\leftarrow$ md

Exécution d'une lecture

Prochaine instruction en assembleur : `load #8`
Exécution :

- lecture de l'instruction, positionne l'adresse : ma $\leftarrow$ pc
- lecture de l'instruction, récupère l'instruction : md $\leftarrow$ Mem[ma]
- lecture de l'instruction, sauvegarde : rinst $\leftarrow$ md
- avance la lecture : pc $\leftarrow$ pc + 1

Exécution d'une lecture

Prochaine instruction en assembleur : `load #8`
Exécution :

- lecture de l'instruction, positionne l'adresse : ma $\leftarrow$ pc
- lecture de l'instruction, récupère l'instruction : md $\leftarrow$ Mem[ma]
- lecture de l'instruction, sauvegarde : rinst $\leftarrow$ md
- avance la lecture : pc $\leftarrow$ pc + 1
- lecture de l'opérande, positionne l'adresse : ma $\leftarrow$ pc

Exécution d'une lecture

Prochaine instruction en assembleur : `load #8`
Exécution :

- lecture de l'instruction, positionne l'adresse : ma $\leftarrow$ pc
- lecture de l'instruction, récupère l'instruction : md $\leftarrow$ Mem[ma]
- lecture de l'instruction, sauvegarde : rinst $\leftarrow$ md
- avance la lecture : pc $\leftarrow$ pc + 1
- lecture de l'opérande, positionne l'adresse : ma $\leftarrow$ pc
- lecture de l'opérande, récupère l'opérande : md $\leftarrow$ Mem[ma]

Exécution d'une lecture

Prochaine instruction en assembleur : `load #8`
Exécution :

- lecture de l'instruction, positionne l'adresse : ma $\leftarrow$ pc
- lecture de l'instruction, récupère l'instruction : md $\leftarrow$ Mem[ma]
- lecture de l'instruction, sauvegarde : rinst $\leftarrow$ md
- avance la lecture : pc $\leftarrow$ pc + 1
- lecture de l'opérande, positionne l'adresse : ma $\leftarrow$ pc
- lecture de l'opérande, récupère l'opérande : md $\leftarrow$ Mem[ma]
- lecture de l'opérande, sauvegarde : acc $\leftarrow$ md

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

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 d'une addition

Prochaine instruction en assembleur : `add 8`

Exécution :

- lecture de l'instruction, positionne l'adresse : ma $\leftarrow$ pc

Exécution d'une addition

Prochaine instruction en assembleur : `add 8`

Exécution :

- lecture de l'instruction, positionne l'adresse : ma $\leftarrow$ pc
- lecture de l'instruction, récupère l'instruction : md $\leftarrow$ Mem[ma]

Exécution d'une addition

Prochaine instruction en assembleur : `add 8`

Exécution :

- lecture de l'instruction, positionne l'adresse : ma $\leftarrow$ pc
- lecture de l'instruction, récupère l'instruction : md $\leftarrow$ Mem[ma]
- lecture de l'instruction, sauvegarde : rinst $\leftarrow$ md

Exécution d'une addition

Prochaine instruction en assembleur : `add 8`

Exécution :

- lecture de l'instruction, positionne l'adresse : ma $\leftarrow$ pc
- lecture de l'instruction, récupère l'instruction : md $\leftarrow$ Mem[ma]
- lecture de l'instruction, sauvegarde : rinst $\leftarrow$ md
- avance la lecture : pc $\leftarrow$ pc + 1

Exécution d'une addition

Prochaine instruction en assembleur : `add 8`

Exécution :

- lecture de l'instruction, positionne l'adresse : ma $\leftarrow$ pc
- lecture de l'instruction, récupère l'instruction : md $\leftarrow$ Mem[ma]
- lecture de l'instruction, sauvegarde : rinst $\leftarrow$ md
- avance la lecture : pc $\leftarrow$ pc + 1
- lecture de l'opérande, positionne l'adresse : ma $\leftarrow$ pc

Exécution d'une addition

Prochaine instruction en assembleur : `add 8`

Exécution :

- lecture de l'instruction, positionne l'adresse : ma $\leftarrow$ pc
- lecture de l'instruction, récupère l'instruction : md $\leftarrow$ Mem[ma]
- lecture de l'instruction, sauvegarde : rinst $\leftarrow$ md
- avance la lecture : pc $\leftarrow$ pc + 1
- lecture de l'opérande, positionne l'adresse : ma $\leftarrow$ pc
- lecture de l'opérande, récupère l'opérande : md $\leftarrow$ Mem[ma]

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

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]

Exécution d'une addition

Prochaine instruction en assembleur : `add 8`

Exécution :

- lecture de l'instruction, positionne l'adresse : ma $\leftarrow$ pc
- lecture de l'instruction, récupère l'instruction : md $\leftarrow$ Mem[ma]
- lecture de l'instruction, sauvegarde : rinst $\leftarrow$ md
- avance la lecture : pc $\leftarrow$ pc + 1
- lecture de l'opérande, positionne l'adresse : ma $\leftarrow$ pc
- lecture de l'opérande, récupère l'opérande : md $\leftarrow$ Mem[ma]
- lecture de l'opérande, sauvegarde : ma $\leftarrow$ md
- lecture de l'opérande, déréférencement : md $\leftarrow$ Mem[ma]
- addition : acc $\leftarrow$ acc + md

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

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

Autres . . .

Autres instructions

- jmp :
  $\rightarrow$ similaire à load mais avec pc au lieu de acc

Autres . . .

Autres instructions

- jmp :
  $\rightarrow$ similaire à load mais avec pc au lieu de acc
- store :
  $\rightarrow$ similaire à add mais avec écriture au lieu de somme

Réalisation de l'ensemble sous forme d'un automate

Mise en forme

Réalisation de l'ensemble sous forme d'un automate

Mise en forme

- Ajout d'une initialisation globale : pc $\leftarrow$ 0

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 de l'ensemble sous forme d'un automate

Mise en forme

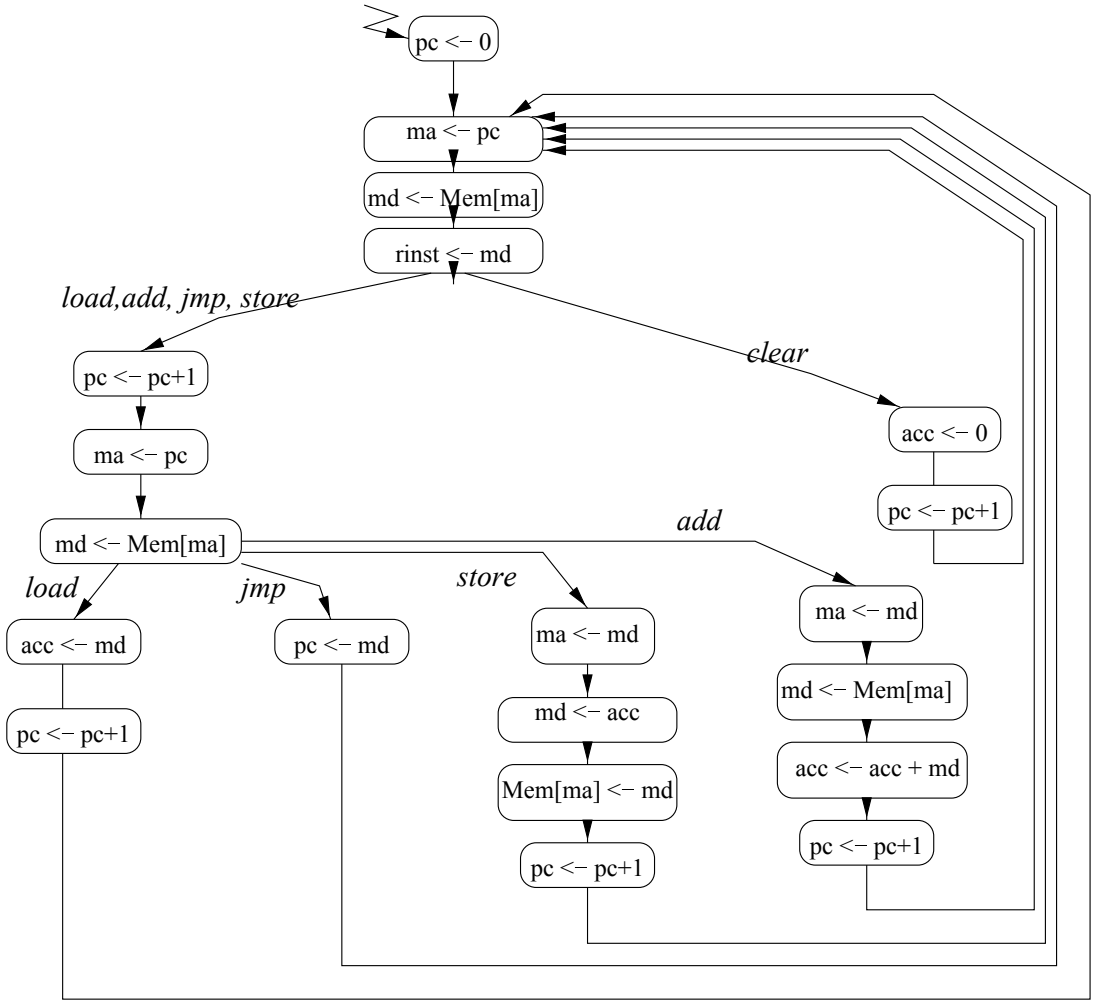
- Ajout d'une initialisation globale : pc $\leftarrow$ 0
- Mise en commun des premières étapes (identiques)
- Réalisation d'alternatives entre transitions dépendant d'instructions différentes

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

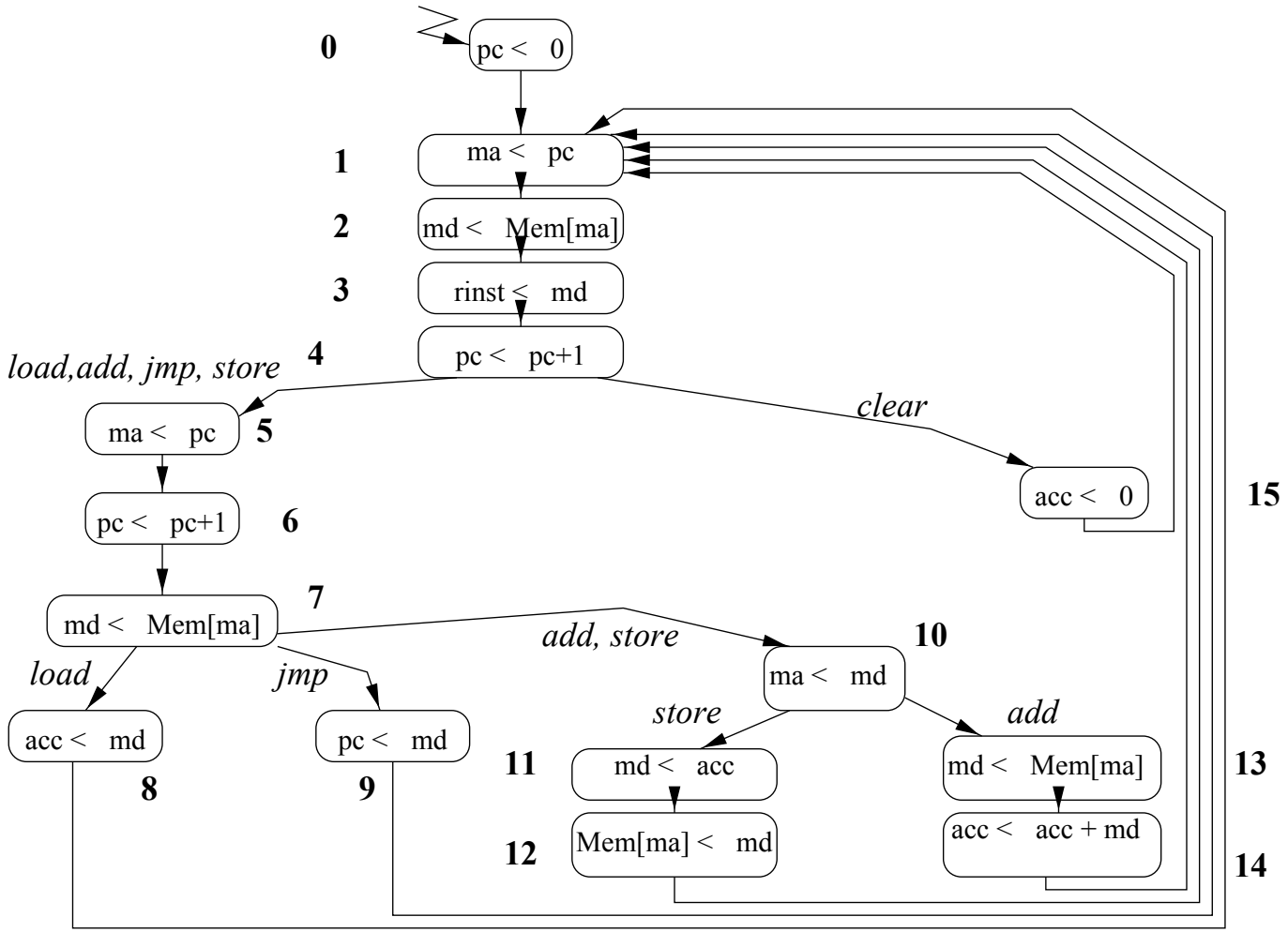
Cet automate est réalisé de manière systématique avec **du matériel** (des transistors).

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



Plan

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.

Contenu en mémoire

adresse	valeur	origine	
0	1	clear	
1	2	load	
2	8	val immédiate	
3	3	store	
4	15	adresse zone data	
5	5	add	
6	15	adresse zone data	
7	4	jump	
8	3	adresse de "ici"	
...	...	...	...
15	variable	non initialisée	

Contenu en mémoire

adresse	valeur	origine	
0	1	clear	
1	2	load	
2	8	val immédiate	
3	3	store	
4	15	adresse zone data	
5	5	add	
6	15	adresse zone data	
7	4	jump	
8	3	adresse de "ici"	
...	...	...	...
15	variable	non initialisée	

Exercice : Donnez le déroulement au cycle près du programme.

Déroulement

état	*pc*	*ma*	*md*	*rinst*	*acc*	*mem*[15]

Déroulement

état	*pc*	*ma*	*md*	*rinst*	*acc*	*mem*[15]
0	0					
1		0				
2			1			
3				1		
4	1					
15					0	
1		1				
2			2			
3				2		
4	2					
5		2				
6	3					
7			8			
8					8	
1		3				
2			3			
3				3		
4	4					
5		4				
6	5					
7			15			
10		15				

état	*pc*	*ma*	*md*	*rinst*	*acc*	*mem*[15]
11			8			
12						8
1		5				
2			5			
3				5		
4	6					
5		6				
6	7					
7			15			
10		15				
13			8			
14					16	
1		7				
2			4			
3				4		
4	8					
5		8				
6	9					
7			3			
9	3					
1	etc.					

Plan

Pipeline (chaîne de traitement)

Hypothèses :

Pipeline (chaîne de traitement)

Hypothèses : la chaîne de traitement d'une instruction comporte 5 étapes :

Pipeline (chaîne de traitement)

Hypothèses : la chaîne de traitement d'une instruction comporte 5 étapes :

IF : récupération de l'instruction

Pipeline (chaîne de traitement)

Hypothèses : la chaîne de traitement d'une instruction comporte 5 étapes :

IF : récupération de l'instruction

ID : récupération des opérandes

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

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

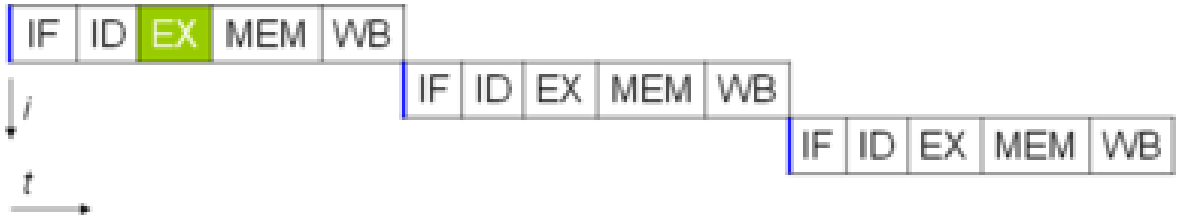
Pipeline (chaîne de traitement)

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

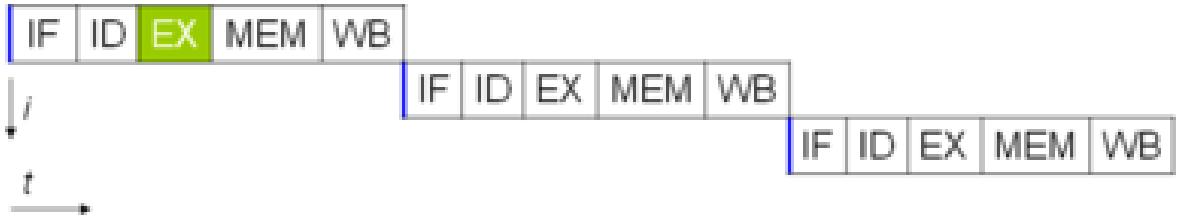
Exécution "simple" de trois instructions :



(source wikipedia)

Exécution de trois instructions

Exécution "simple" de trois instructions :



(source wikipedia)

- temps d'exécution : 15 Δ
- Peut mieux faire ...

Principe du pipeline

Principes :

Principe du pipeline

Principes :

- chaque étape est indépendante

Principe du pipeline

Principes :

- chaque étape est indépendante
- chaque étape peut-être réalisée par une partie du processeur différente

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

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)

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

Exécution de 100 instructions

- Temps d'exécution, si exécution simple : 500 Δ

Exécution de 100 instructions

- Temps d'exécution, si exécution simple : 500 Δ
- Temps d'exécution, si exécution pipelinée : 104 Δ

Exécution de 100 instructions

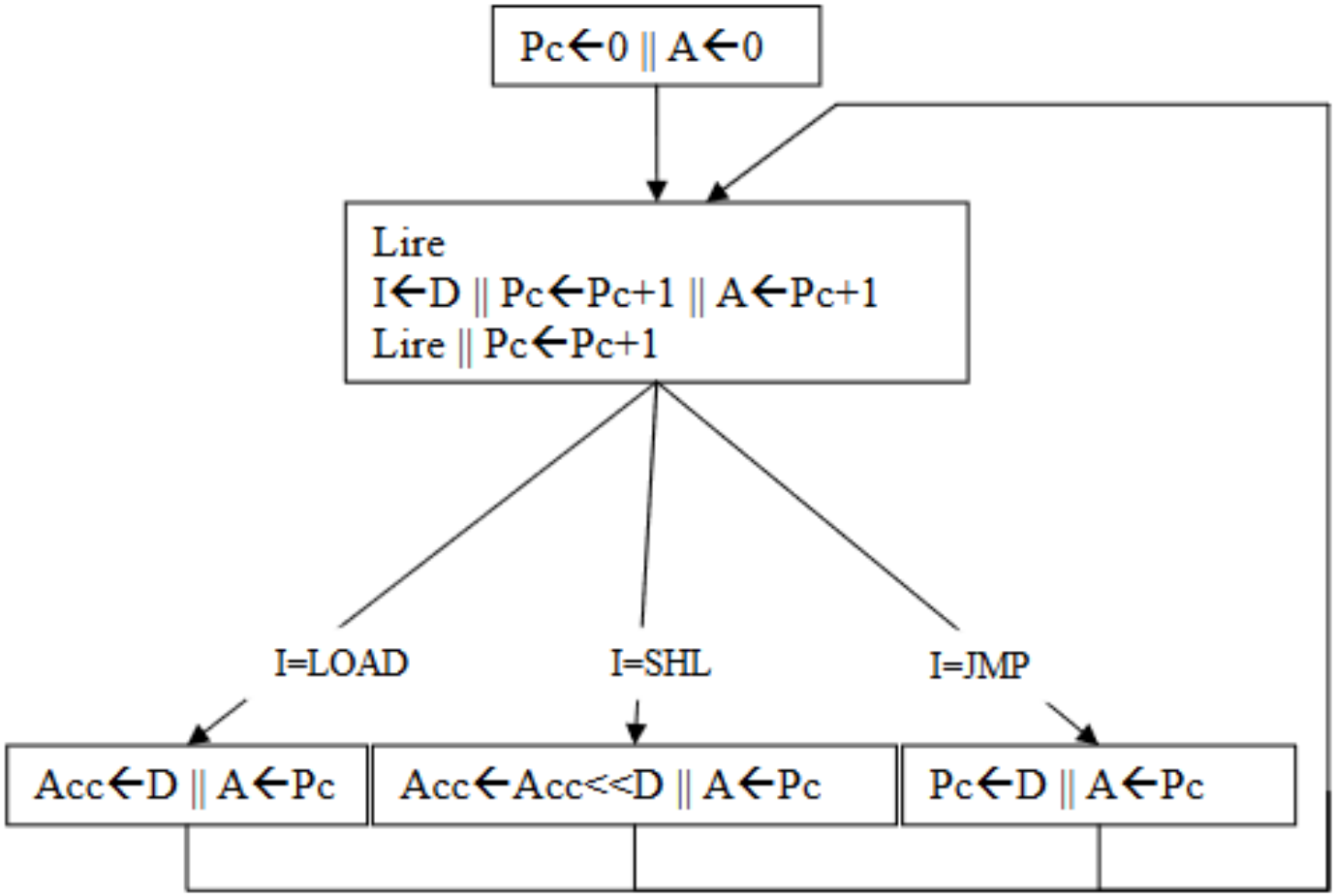
- Temps d'exécution, si exécution simple : 500 Δ
- Temps d'exécution, si exécution pipelinée : 104 Δ
- Gain espéré : processeur 5 fois plus rapide (la profondeur du pipeline)

Automate d'interprétation avec pipeline

Version d'un automate simple sans pipeline

Automate d'interprétation avec pipeline

Version d'un automate simple sans pipeline

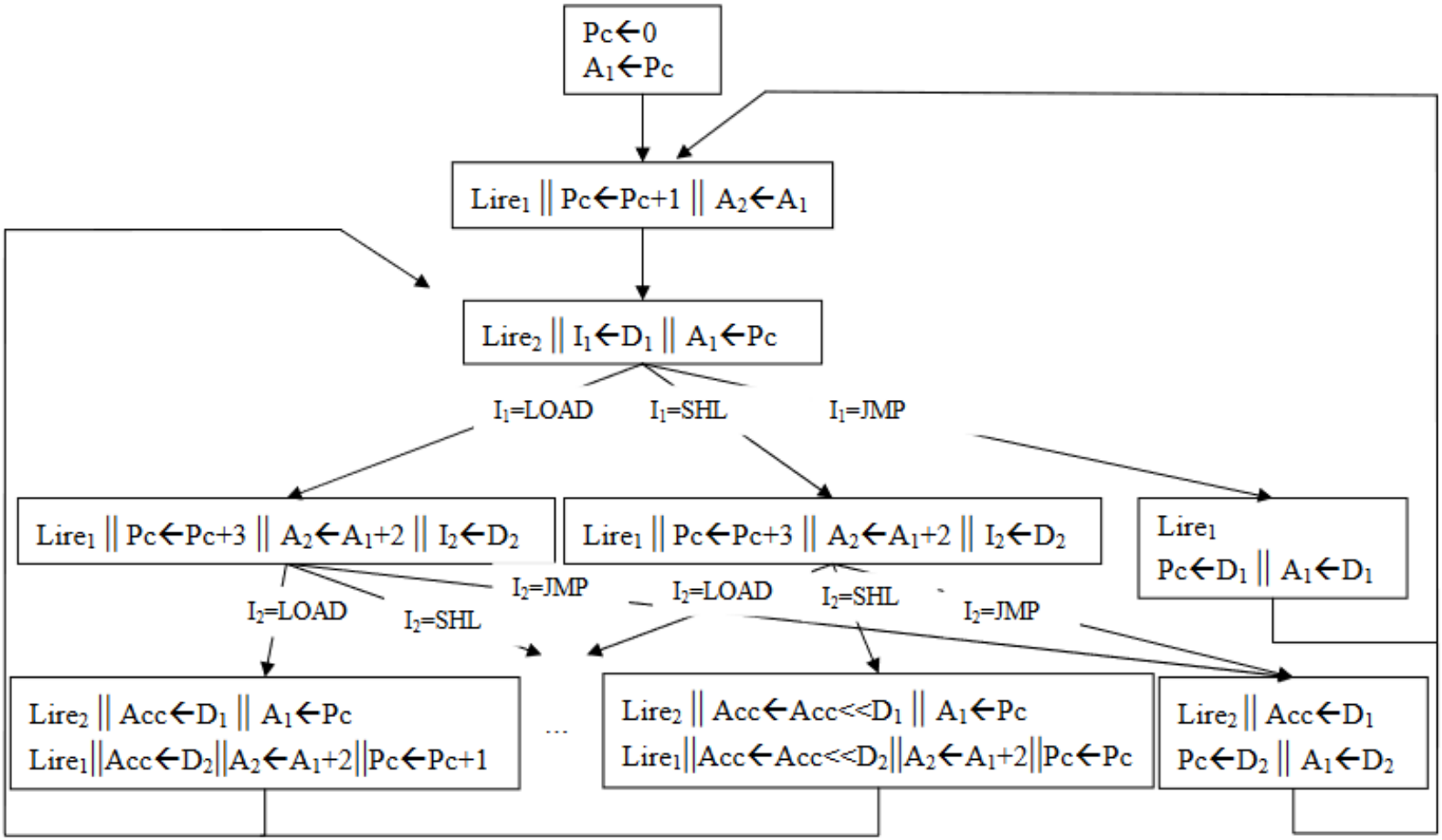


Automate d'interprétation avec pipeline

Version avec pipeline

Automate d'interprétation avec pipeline

Version avec pipeline



Variantes

Pour aller plus vite (?), des pipelines plus longs (plus profonds)

Variantes

Pour aller plus vite (?), des pipelines plus longs (plus profonds)

- Intel Itanium (2001, . . .), profondeur du pipeline : 10

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

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

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

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 !

Rupture de pipeline

Mais !

les pipelines peuvent être rompus :

Rupture de pipeline

Mais !
les pipelines peuvent être rompus :

- Branchement

Rupture de pipeline

Mais !
les pipelines peuvent être rompus :

- Branchement
- Dépendance entre données

Rupture de pipeline

Mais !
les pipelines peuvent être rompus :

- Branchement
- Dépendance entre données
- Interruption

Rupture de pipeline

Mais !
les pipelines peuvent être rompus :

- Branchement
- Dépendance entre données
- Interruption
- ...

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.