Architecture des systèmes

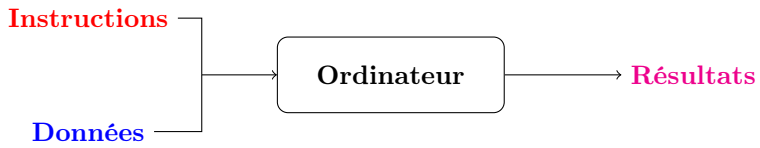
Grégoire Roumache

Octobre 2019

Table des matières

1 Concepts de base

1.1 Représentation n°1

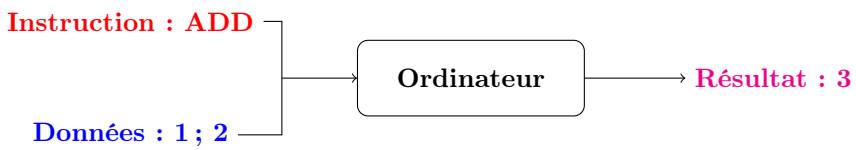


À retenir de ce modèle :

— Les flux d'informations sont **continus**. (flux = instructions, données, résultats).

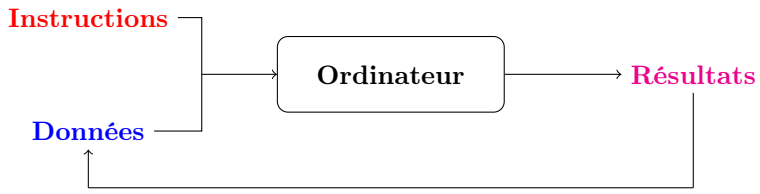
— L'ordinateur n'effectue que des opérations simples (ex : opérations arithmétiques).

Exemple :

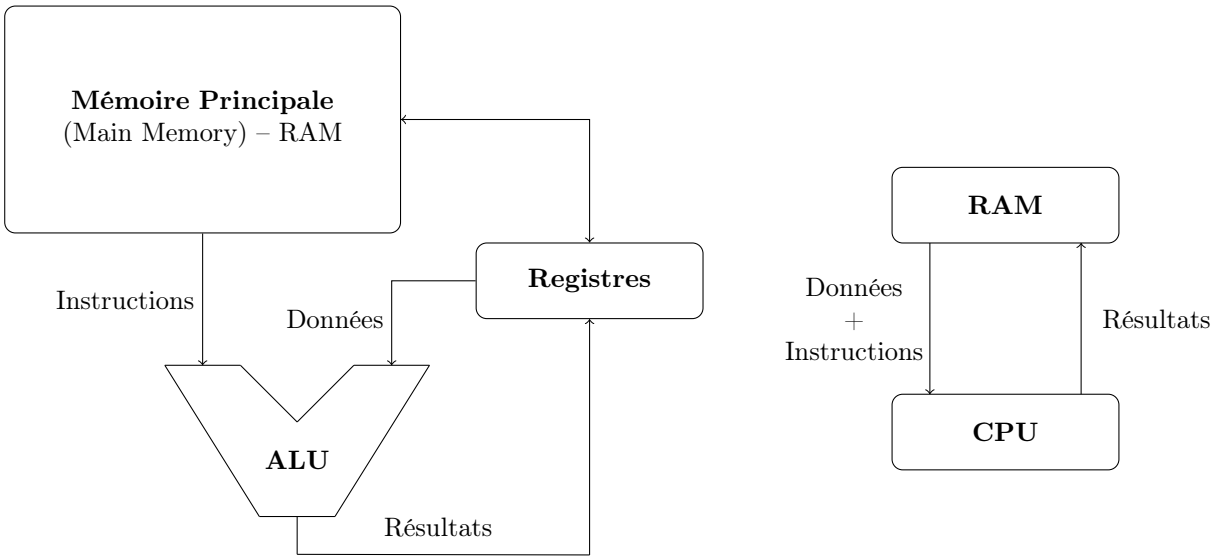


1.2 Représentation n°2

Représentation du processus :



Représentation des composants :



Remarque : ALU + Registres = CPU

> Exemple : C = A + B – instruction : `add A, B, C`
>
> 1. Charger les valeurs à additionner dans les registres A et B.
> 2. Effectuer l'addition.
> 3. Stocker/Enregistrer le résultat du registre C dans la mémoire principale/centrale.
>
> Remarque : dans le cours "main memory" est traduit par *mémoire centrale* au lieu de *mémoire principale*.

Définitions :

— Un **programme informatique** (code) est une série d'instructions ou commandes qui précise à la machine toute entière (pas seulement l'ALU) ce qu'elle doit faire.

— Un **registre** est un petit espace de stockage, très rapide, incorporé au CPU.

— Un **bus** (ex : data bus, instruction bus) est un dispositif servant à transférer des données.

Il y a 4 types d'instructions :

— les intructions **arithmétiques** (ex : add, sub) ;

— les intructions d'**accès mémoire** (ex : load, store) ;

— les instructions de **branchement** (ex : jump) ;

— les instructions **logiques** (ex : and, or).

1.3 Architecture & format d'instruction

Dans ce cours, on a une machine AR1, composée de :

— 1 ALU

— 4 registres : A, B, C, D

— 256 cellules de mémoire RAM (de 0 à 255) [1].

Formats d'instructions :

— Instruction arithmétique : `instruction source1, source2, destination` (ex : `add A, B, C`)

— Instruction d'accès mémoire : `instruction source, destination` (ex : `load #12, A`)

Remarques :

— Les arguments sont séparés par des virgules.

— Le symbole # est utilisé pour marquer l'adresse d'une cellule (ex : 5 et D sont des nombres, alors que #5 et #D sont des adresses).

1.4 Adressage relatif

Définitions :

— Une **adresse absolue** est l'adresse complète de destination.

— Une **adresse relative** est une adresse donnée par rapport à une adresse de référence.

$$\textbf{Adresse relative} = \textbf{Adresse de base} + \textbf{offset}$$

Exemples :

— `load #(D + 108), A`

— `store B, #(D + 108)`

Avantages :

— L'OS gère la localisation exacte des données en mémoire, pas le programmeur.

— La localisation des données en mémoire peut changer sans impacter le bon fonctionnement du programme.

2 Mécanismes d'exécution d'un programme

2.1 Opcodes & langage machine

Définitions :

— L'**opcode** (code opération) est une partie de l'instruction qui spécifie l'opération à effectuer (ex : `add`, `load`).

— Le **langage machine** est un langage de programmation qui peut être exécuté directement par l'ordinateur.

— L'ensemble des opérations disponibles est l'**instruction set**.

Les registres sont codés sur 2-bits et les opcodes sont codés sur 3-bits :

Mnemonic	Opcode
add	000
sub	001
load	010
store	011

Registre	Code
A	00
B	01
C	10
D	11

Toutes les instructions sont codées sur 16-bits (2 octets).

Remarque (notée "remarque importante" dans le cours) : Augmenter le nombre de bits utilisés pour coder une opération ou un registre permet d'augmenter le nombre d'opérations et de registres disponibles.

1. Architecture 8-bits, donc $2^8 = 256$ adresses mémoire.

Exemple :

2-bits $\implies$ 4 possibilités (registres/opcodes)
3-bits $\implies$ 8 possibilités (registres/opcodes)
4-bits $\implies$ 16 possibilités (registres/opcodes)

2.2 Instructions arithmétiques

ATTENTION :

— 1er bit = 0 $\implies$ manipulation de registres $\implies$ les 6 derniers bits sont non-utilisés.

— 1er bit = 1 $\implies$ utilisation d'une valeur immédiate $\implies$ le 2ème octet est une valeur immédiate (un nombre, pas un registre).

Mode registre (1er bit = 0) :

Premier octet :

0	1	2	3	4	5	6	7
mode	opcode			source 1		source 2	

Second octet :

8	9	10	11	12	13	14	15
destination		0	0	0	0	0	0

Les 6 derniers bits complètent l'écriture sur deux octets, ils ne servent qu'à remplir l'espace vide (padding).

Mode valeur immédiate (1er bit = 1) :

Premier octet :

0	1	2	3	4	5	6	7
mode	opcode			source		destination	

Second octet :

8	9	10	11	12	13	14	15
valeur immédiate de 8-bits							

Exemples :

`add C, D, A` : $\underbrace{0}_{mode}\ \underbrace{000}_{add}\ \underbrace{10}_{C}\ \underbrace{11}_{D}\ \underbrace{00}_{A}\ \underbrace{000000}_{padding}$

`sub A, D, C` : $\underbrace{0}_{mode}\ \underbrace{001}_{sub}\ \underbrace{00}_{A}\ \underbrace{11}_{D}\ \underbrace{10}_{C}\ \underbrace{000000}_{padding}$

`add C, 8, A` : $\underbrace{1}_{mode}\ \underbrace{000}_{add}\ \underbrace{10}_{C}\ \underbrace{00}_{A}\ \underbrace{00001000}_{8}$

`add 8, C, A` : $\underbrace{1}_{mode}\ \underbrace{000}_{add}\ \underbrace{10}_{C}\ \underbrace{00}_{A}\ \underbrace{00001000}_{8}$

`sub 25, D, C` : $\underbrace{1}_{mode}\ \underbrace{001}_{sub}\ \underbrace{11}_{D}\ \underbrace{10}_{C}\ \underbrace{00011001}_{25}$

Remarques :

— Les instructions en langage assembleur suivantes : `add C, 8, A` et `add 8, C, A`, ont la même instruction en langage machine.

— Avec l'opcode *sub*, on peut faire cette instruction : `sub 25, D, C` (C = 25 - D), mais pas celle-ci : `sub D, 25, C` (C = D - 25). Pour pouvoir le faire, il faudrait un autre opcode parce que la soustraction n'est pas commutative.

2.3 Instructions d'accès mémoire

ATTENTION : il y a 3 types d'instructions d'accès mémoire :

— **Accès immédiat** : on donne l'adresse directement.

— **Accès registre** : on copie la valeur contenue dans un registre.

— **Adresse relative** : adresse relative = adresse de base + offset.

Accès immédiat (mode = 1) : puisqu'on donne l'adresse directement, on n'a pas besoin d'un registre "source".

0	1	2	3	4	5	6	7
mode	opcode			0	0	destination	

8	9	10	11	12	13	14	15
valeur immédiate de 8-bits							

Accès registre (mode = 0) : on n'a besoin ni d'une seconde source, ni des 6 derniers bits.

0	1	2	3	4	5	6	7
mode	opcode			source 1		0	0

8	9	10	11	12	13	14	15
destination		0	0	0	0	0	0

Adresse relative (mode = 1) : adresse relative = adresse de base + offset. L'adresse de base est contenue dans le registre qu'on donne dans "base".

0	1	2	3	4	5	6	7
mode	opcode			base		destination	

8	9	10	11	12	13	14	15
offset de 8-bits							

2.4 Assembleur

Définitions :

— La traduction du programme en langage machine est réalisée par l'**assembleur**.

— Le "**programming model**" est un modèle qui décrit la machine.

Il y a 3 registres importants pour comprendre le modèle :

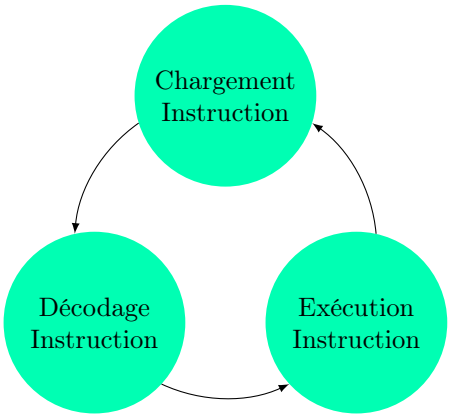
— Le "Program Counter" (PC). Il indique tout simplement l'adresse de la prochaine instruction à charger.

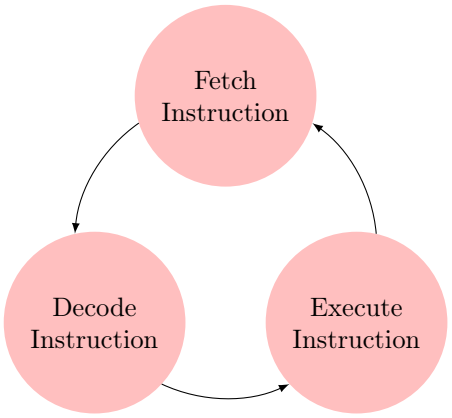
— L' "Instruction Register" (IR). L'instruction à exécuter est chargée dans ce registre.

— Le "Processor Status Word" (PSW). Les opérations arithmétiques vont écrire des informations sur leur déroulement dans PSW en mettant à 0 (false) ou 1 (true) des bits relatifs à une condition bien précise. Ainsi, pour faire un saut conditionnel (Conditionnal Branch), il suffit de vérifier le bit approprié dans PSW (principe de Flags).

2.5 Boucle Fetch-Decode-Execute

Pour exécuter un programme, l'ordinateur exécute une séquence d'instructions. Le processeur doit faire une boucle pour exécuter chaque instruction.





— **Fetch** : on va charger l'instruction à exécuter dans l'Instruction Register (son adresse est dans le Program Counter).

— **Decode** : on décode l'instruction située dans IR.

— **Execute** : on exécute l'instruction. Pendant ce temps, PC est incrémenté pour pointer sur la prochaine instruction à exécuter.

Remarque :

— 1 cellule mémoire = 1 octet et 1 instruction = 2 octets $\implies$ PC est incrémenté de 2 unités pour pointer sur la porchaine instruction.

— Un cycle fetch-decode-execute doit être réalisé complètement sur un battement d'horloge.

— Une instruction de branchement (comme `jump`) permet un saut vers une autre partie du programme.

— Instructions de saut conditionnel : `jumpz`, saute si le résultat de l'instruction précédente est 0 ; `jumpn`, saute si le résultat est négatif ; `jumpo`, saute si il y a un overflow.

— Un "label" (étiquette) est utilisé pour marquer l'emplacement d'un code vers lequel on veut sauter.

Exemple code : calcul de 5×4 $(= 5 + 5 + 5 + 5)$

```
      load 5, A
      load 4, B
      load 0, C
loop: sub B, 1, B
      add A, C, C
      sub B, 0, B
      jumpz end
      jump loop
end:
```

Le registre C contient 20. Remarque : normalement `sub` ne peut pas être utilisée comme ça, il faudrait une autre instruction (voir section "instructions arithmétiques").

3 Exécution en pipeline

3.1 Completion Rate

Définitions :

— Le **Completion Rate** doit être défini comme le nombre d'instructions terminées par unité de temps (instr/ns).

— À ne pas confondre avec le **temps d'exécution** (program execution time) d'une instruction (ns).

— Le temps d'exécutions d'un programme est donné par :

$$\text{Program Execution Time} = \frac{\text{Number of Instructions in program}}{\text{Instruction Completion Rate}}$$

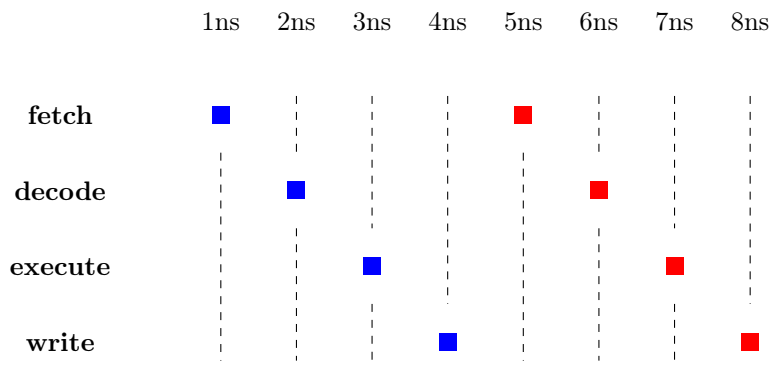
Pour augmenter les performances, 2 possibilités :

— Réduire le nombre d'instructions du programme.

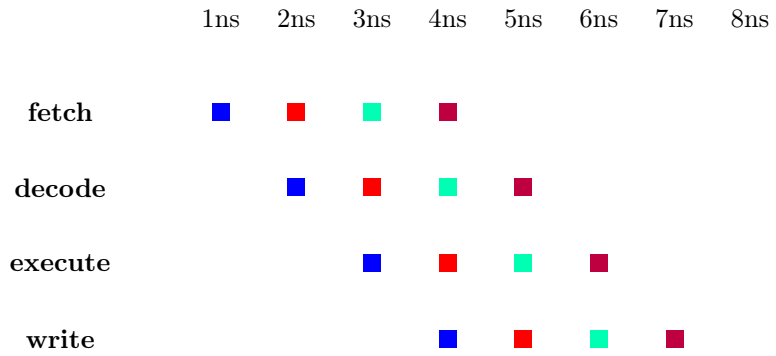
— Augmenter le *completion rate*.

Si le nombre d'instructions est fixé, il faut augmenter le *completion rate*. On peut le faire, sans réduire le temps d'exécution d'une instruction, en utilisant une pipeline.

Processeur "single cycle" (sans pipeline) :



Processeur avec pipeline (profondeur de 4) :



> Ici, on a exécuté 4 instructions en 7 ns au lieu de 16 ns. En fait, dès le début de la 5ème ns, une instruction est exécutée toutes les ns. Puisqu'on exécute 1 instruction/ns au lieu de 1 instruction/4 ns, on a un gain de 4 (0.25 instr/ns au lieu de 1 inst/ns).

Conséquences pour l'horloge :

— L'horloge ne cadence plus une instruction complète mais les phases d'une instructions.

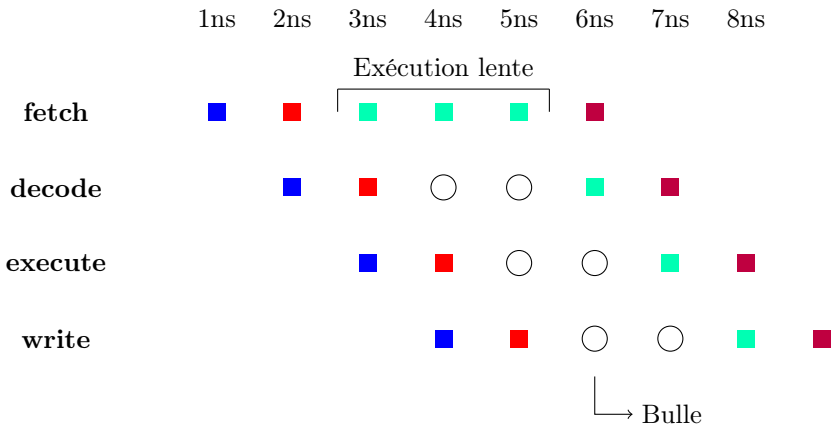
— Les pulsations sont bien plus courtes. Dans l'exemple : 4ns → 1ns.

Remarques :

— Si on fait plus d'étapes mais que chaque étape prend moins de temps, on peut encore réduire la durée d'exécution du programme.

— Si des étapes du pipeline nécessitent plus d'un cycle d'horloge, des bulles apparaissent et voyagent dans le pipeline, ce qui réduit les performances du processeur.

La **latence d'une instruction** peut traduire le temps réel qu'elle met pour traverser le pipeline en prenant en compte les bulles.



Remarque : dans le cours, les 4 séquences fetch, decode, execute et write sont représentées sur le schéma suivant,