Architecture des ordinateurs

Gianluca QUERCINI

Enseignant-chercheur,

CentraleSupélec, Laboratoire de Recherche en Informatique, Univ. Paris-Saclay

Département Informatique, Bureau D2.20

gianluca.quercini@centralesupelec.fr

Séquence 0, 2017 - 2018

L'ordinateur

- **Ordinateur** : machine conçue pour faire des calculs.
  - « Computer » en Anglais : personne qui fait des calculs.
  - Le mot « computer » a été utilisé pour indiquer une machine qui fait des calculs à partir de 1945 (ENIAC – Eckert et Mauchly).

- Mais aujourd'hui un ordinateur fait beaucoup plus que ça….

- Résoudre des problèmes suivant un programme.
  - **Programme** : séquence d'instructions décrivant la façon dont le problème doit être résolu.

- Deux catégories d'ordinateurs :
  - Ordinateur personnel (*personal computer* ou *PC*).
  - Système embarqué (*embedded system*).
    - intégré dans un système afin de le contrôler et le piloter (lave-linges).

Objectifs du cours

Objectif principal : Démystifier l'ordinateur et ses composants.

- Analyser les principaux composants d'un ordinateur et leur rôle.
  - Processeur, mémoire, périphériques….
- Décrire l'organisation et le fonctionnement de ces entités.
  - Fonctionnement d'un processeur, de la mémoire…
- Comprendre l'exécution d'un programme.
  - Comment le processeur comprend et exécute-t-il un programme Python?

Les composants d'un ordinateur

Processeur = CPU

Mémoire principale

Périphériques

Mémoire secondaire

Serial/USB/Firewire

Serial/USB/Firewire

Réseau

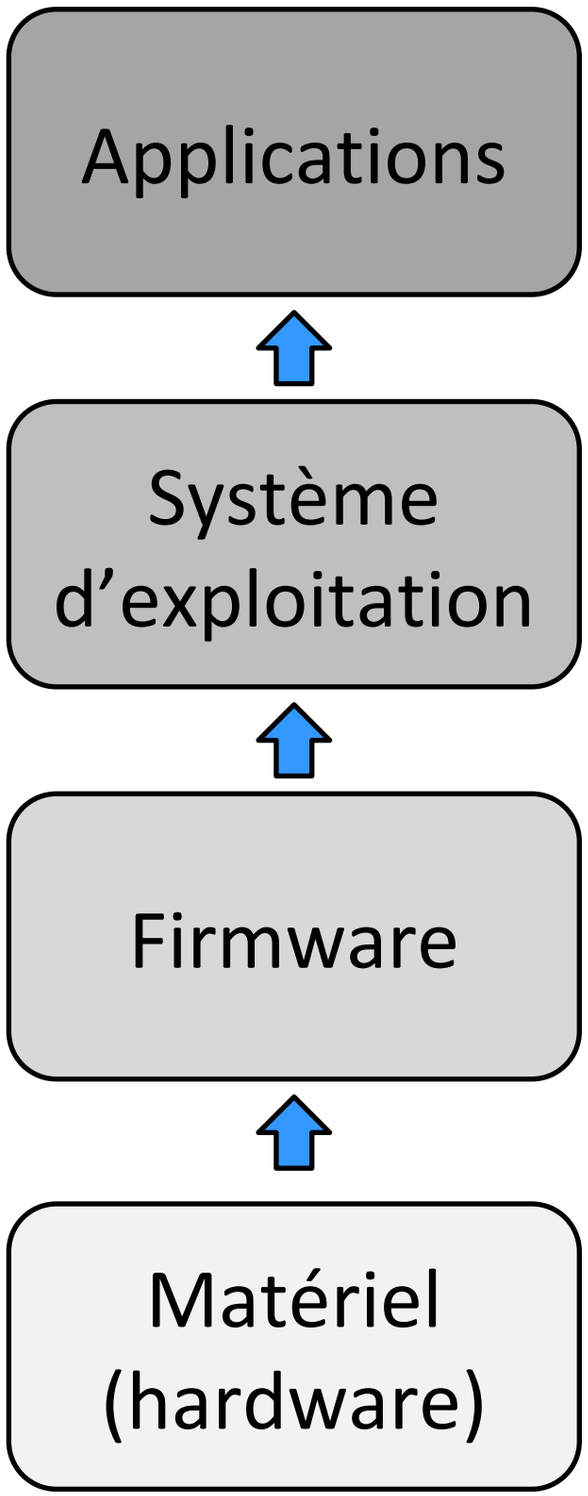
input

output

Audio

Carte graphique

Logiciel (Software)



Logiciels utilisés par les utilisateurs.

Logiciel pour le contrôle de l'ordinateur. Fournit une interface aux utilisateurs.

Logiciel(s) pour la gestion et le contrôle du matériel.

Architecture de l'ordinateur

Architecture de l'ordinateur : étude du fonctionnement des composants internes d'un ordinateur.

- Fonctionnement logique interne des composants.
- Dialogue entre composants.
- Type des données manipulées et leur codage.

Organisation du cours

- 7 **cours magistraux** (CM).
  - 1 CM = 1h30
  - Présentation orale des concepts principaux du cours.

- 3 **bureaux d'études** (BE).
  - 1 BE = 2 TD = 3h
  - Travaux dirigés en salle machine.

- 2 **études de laboratoire** (EL).
  - 1 EL = 3 TP = 4h30
  - Travaux pratiques en salle machine.

Questions ?

- Pendant le cours.
- Après le cours.
- En dehors du cours :
  - Département informatique, Bureau D2.20
  - Courriel : gianluca.quercini@centralesupelec.fr
- **Site du cours :** http://wdi.supelec.fr/architecture/

Programme

- **Chapitre 1** : Circuits logiques (15/09/2017).
  - Circuits combinatoires.
  - Circuits séquentiels.

- **Chapitre 2** : Arithmétique des processeurs (15/09/2017).
  - Codage des entiers naturels et relatifs.
  - Codage des nombres réels.
  - Codage des caractères.

- **Chapitre 3** : Le processeur (15/09/2017).
  - Composants d'un processeur.
  - Fonctionnement d'un processeur.

- **Bureau d'étude 1** : Circuits combinatoires et séquentiels (19/09/2017)

Programme

- **Chapitre 4** : Langages de programmation (20/09/2017).
  - Langages compilés et interprétés.
  - Appel de fonction et notion de pile.
  - Fonctions récursives.

- **Chapitre 5** : Mémoire (20/09/2017).
  - La mémoire RAM
  - La mémoire secondaire.
  - Notion de mémoire cache.

- **Bureau d'étude 2** : Traduction d'instructions Python en langage d'assemblage (20/09/2017).

Programme

- **Chapitre 6** : Entrées/Sorties (21/09/2017).
  - Notion de bus.
  - Entrées/Sorties mappées en mémoire.
  - Notion d'interruption.

- **Chapitre 7** : Notions avancées sur les processeurs (21/09/2017).
  - Architectures RISC et CISC.
  - Aperçu d'un processeur réel.

- **Chapitre 8** : Aperçu de l'histoire de l'informatique (21/09/2017).

- **Bureau d'étude 3** : MicroPython, entrées/sorties, interruptions (25/09/2017).

Programme

- **Etude de laboratoire 1** : Réalisation d'un microprocesseur.
- **Etude de laboratoire 2** : Entrées/sorties.

CentraleSupélec

université PARIS-SACLAY

LABORATOIRE DE RECHERCHE EN INFORMATIQUE

CHAPITRE I
Circuits logiques

Architecture des ordinateurs

Circuit logique

- Circuit électronique qui calcule une ou plusieurs fonctions logiques.

- Un circuit logique utilise deux valeurs logiques (**bits – *binary digits***) :
  - un signal entre 0V et 1V : **bit 0**.
  - un signal entre 2V et 5V : **bit 1**.

- Les circuits logiques utilisent une arithmétique binaire.
  - **Bit** (chiffre binaire): unité de mémoire de base
  - **1 octet (byte)** = 8 bits, **1 KB (Kilobyte)** = 2^{10} bytes = 1 024 octets
  - **1 MB (Megabyte)** = 2^{20} bytes = 1 048 576 octets, **1 GB (Gigabyte)** = 2^{30} bytes
  - **1 TB (Terabyte)** = 1000 GB, **1 PB (Petabyte)** = 1000 TB, **1 EB (Exabyte)** = 1000 PB
  - **1 ZB (Zettabyte)** = 1000 EB, **1 YB (Yottabyte)** = 1000 ZB
- **Arithmétique hexadécimale** pour rendre plus compacte la notation binaire.
  - 16 chiffres : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

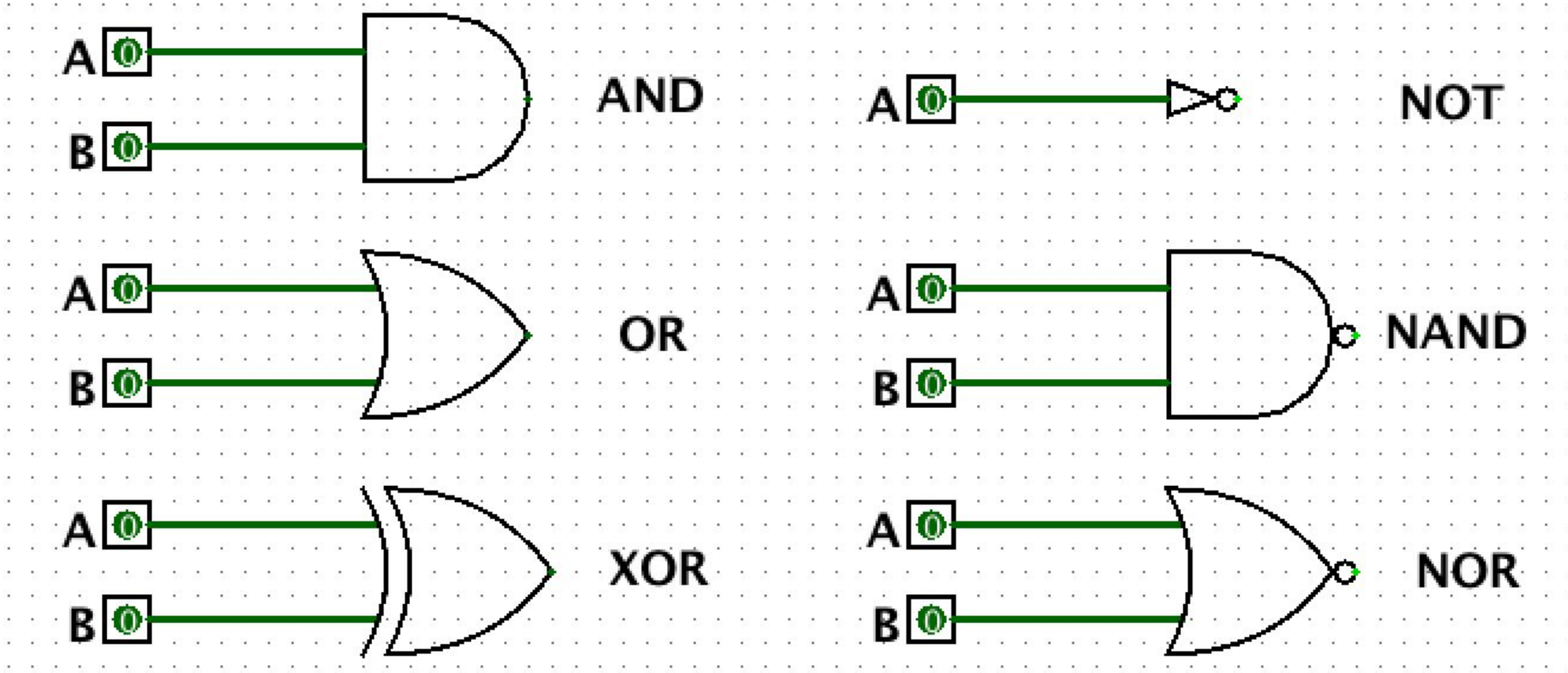
Algèbre de Boole

- Les circuits logiques peuvent être décrits à l'aide de l'**algèbre de Boole**.
  - opérations et fonctions sur des variables logiques qui ne peuvent prendre que les valeurs 0 (faux) et 1 (vrai)
  - George Boole (1815 - 1864) : mathématicien britannique.
- **Fonction booléenne.**
  - **Entrée :** une ou plusieurs variables booléennes.
  - **Sortie :** 0 ou 1.
  - Décrite à l'aide d'une **table de vérité**.
- **Fonctions booléennes de base.**
  - Soient A, B deux variables booléennes
  - AND (A•B), OR (A+B), NOT ($\bar{A}$), NAND ($\overline{A \cdot B}$), NOR ($\overline{A+B}$), XOR ($A \oplus B$)

A	B	$\bar{A}$	$\bar{B}$	A•B	A+B	$\overline{A \cdot B}$	$\overline{A+B}$	A⊕B
0	0	1	1	0	0	1	1	0
0	1	1	0	0	1	1	0	1
1	0	0	1	0	1	1	0	1
1	1	0	0	1	1	0	0	0

Portes logiques

- **Porte logique** : circuit logique de base.
  - Réalisation des fonctions booléennes élémentaires (NOT, NAND, NOR, AND, OR, XOR).
  - réalisée avec des **transistors**.

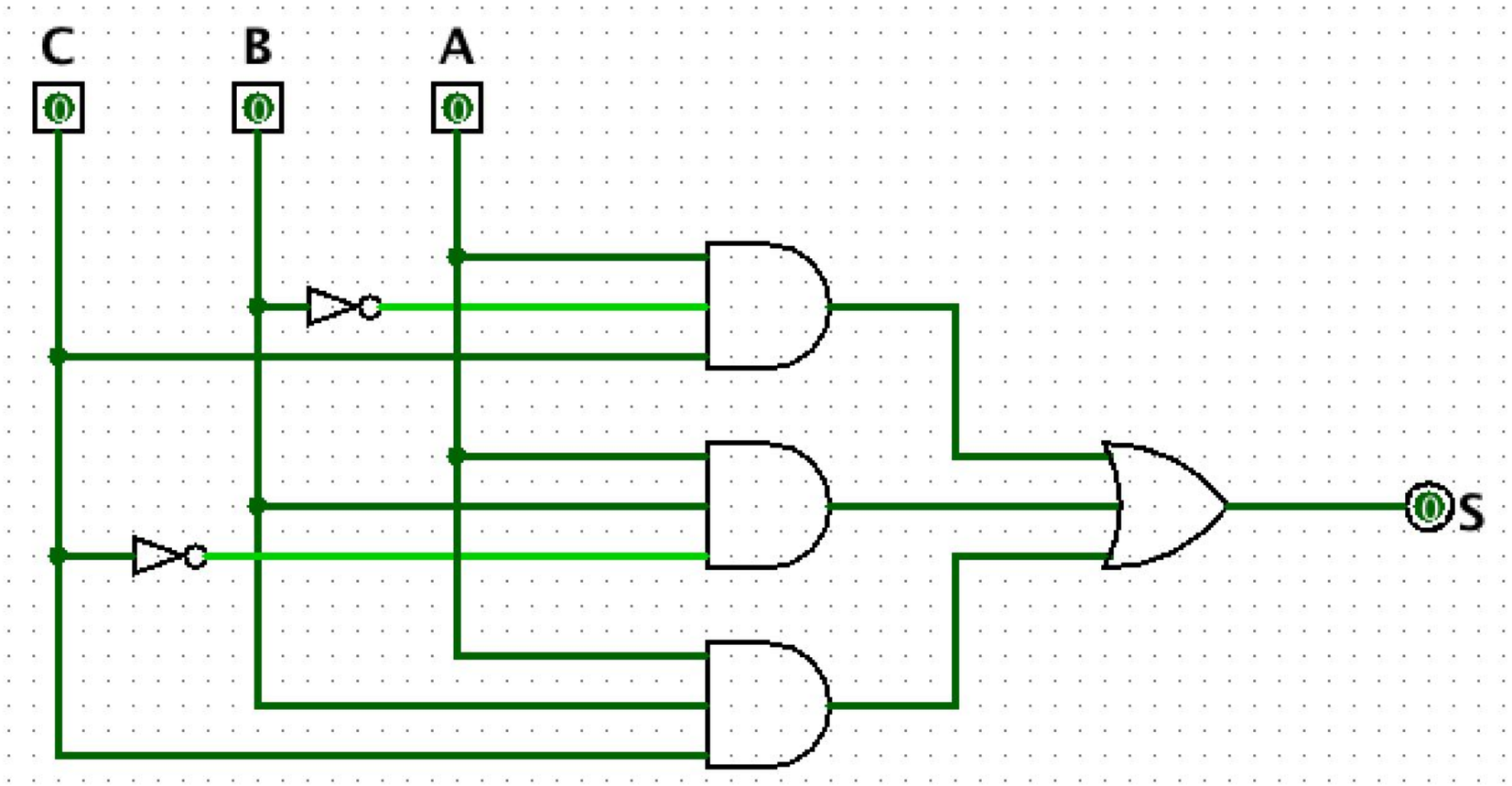


- Une porte logique peut avoir plusieurs entrées.

Exemple de circuit logique

- Un circuit logique est un ensemble de portes logiques interconnectées.

$$S = A \bullet \overline{B} \bullet C + A \bullet B \bullet \overline{C} + A \bullet B \bullet C$$



- Deux types de circuits logiques : **combinatoires** et **séquentiels**.

Circuits logiques combinatoires

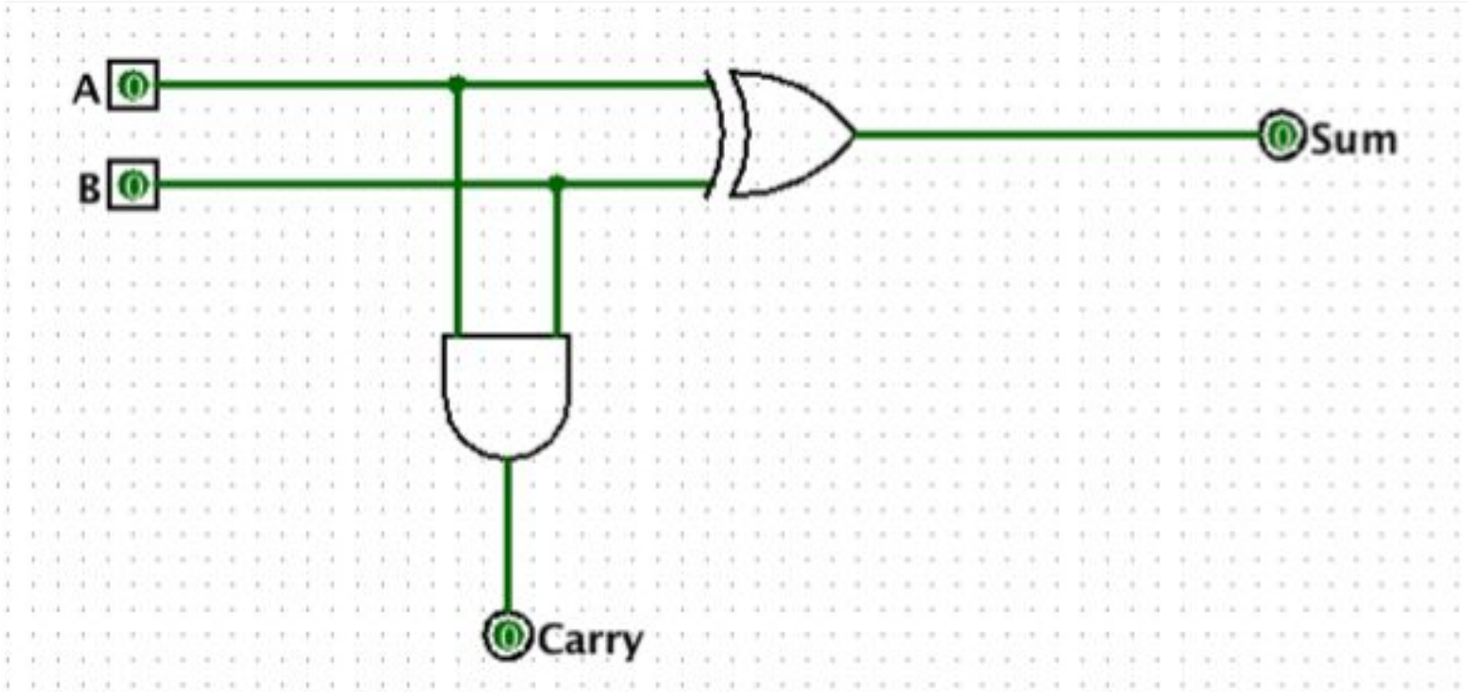
- **Circuit combinatoire :** la valeur des sorties ne dépend que de la valeur des entrées.

- Les composants d'un ordinateur (par ex. un processeur) contiennent beaucoup de circuits combinatoires :
  - **Additionneur** : circuit pour additionner deux nombres.
  - **Multiplexeur** : circuit qui redirige une de ses entrées vers la sortie.
  - **Démultiplexeur** : circuit qui redirige une entrée vers une de ses sorties.
  - **Décodeur** : active (met la valeur 1 sur) une des ses sorties selon un code en entrée.
  - **Codeur** : pour une entrée active (valeur 1), fournit un code en sortie.

L'additionneur à 1 bit

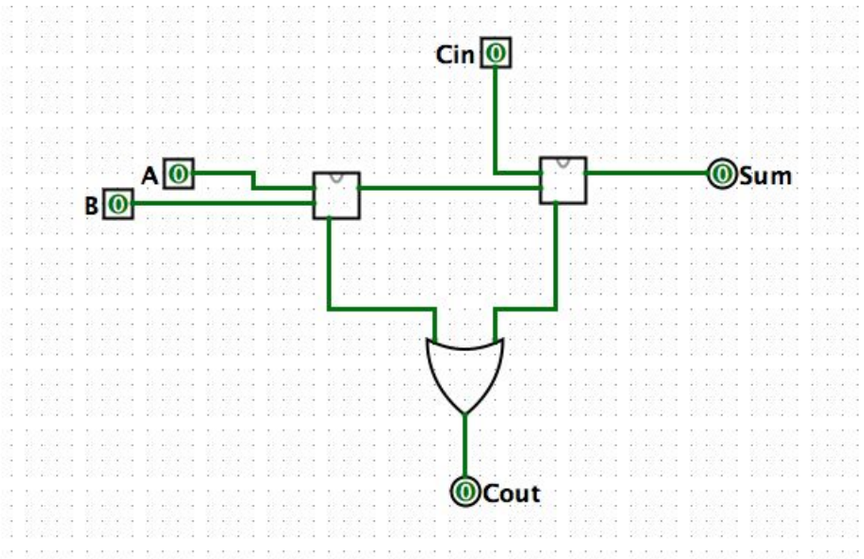
- Le composant de base est le **demi-additionneur** (*half-adder*).
- Réalise l'addition de deux nombres codés chacun sur un 1 bit.
- Deux entrées : **A** et **B**, 1 bit chacun.
- Deux sorties : **Sum** (somme) et **Carry** (retenue), 1 bit chacun.

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



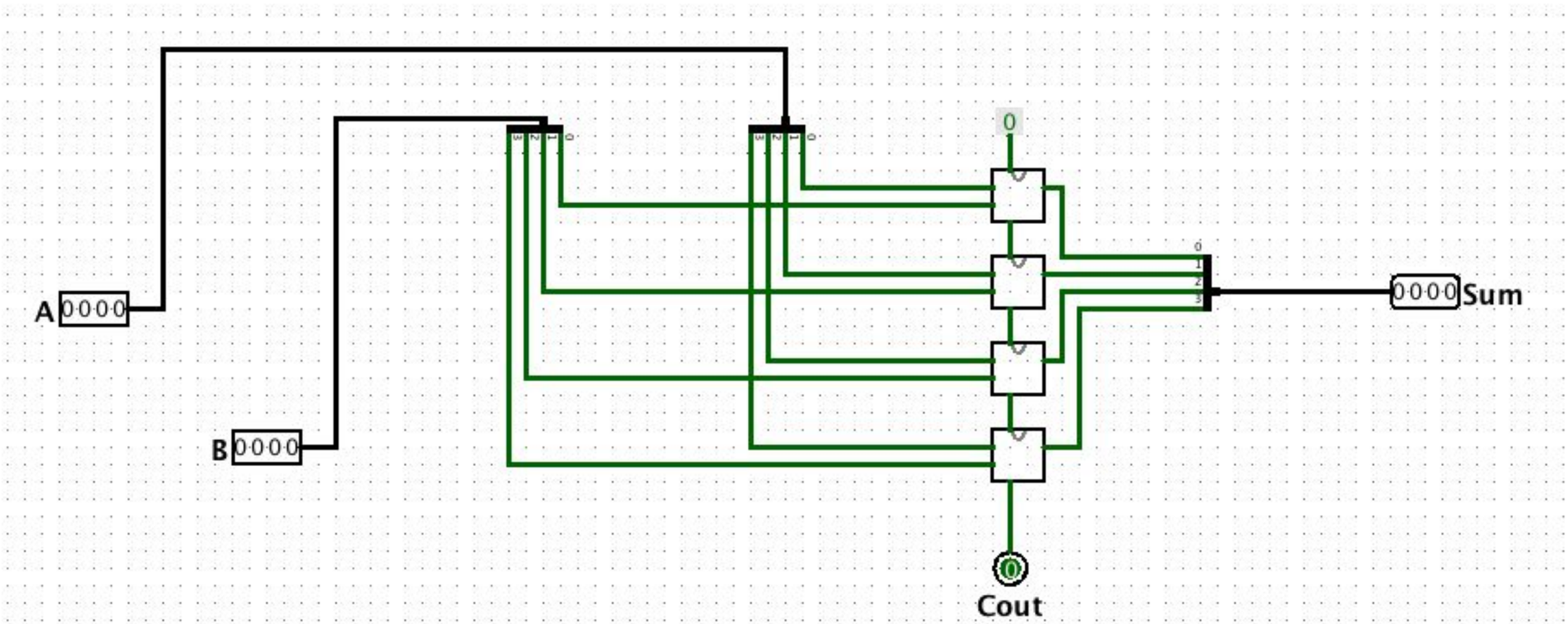
L'additionneur à 1 bit

- L'**additionneur complet** (*full adder*) :
  - Réalise l'addition de deux nombres codés chacun sur un 1 bit.
  - Prend aussi en compte une éventuelle retenue en entrée.
- Trois entrées : **A**, **B**, **Cin** (retenue en entrée), 1 bit chacun.
- Deux sorties : **Sum** (somme) et **Cout** (retenue en sortie), 1 bit chacun.
- Idée : mettre en série deux demi additionneurs.
  - Un pour obtenir **S = A + B**.
  - L'autre pour obtenir **Sum = Cin + S**.
  - Problème : comment combiner les deux retenues produites par les deux demi additionneurs pour obtenir **Cout**?

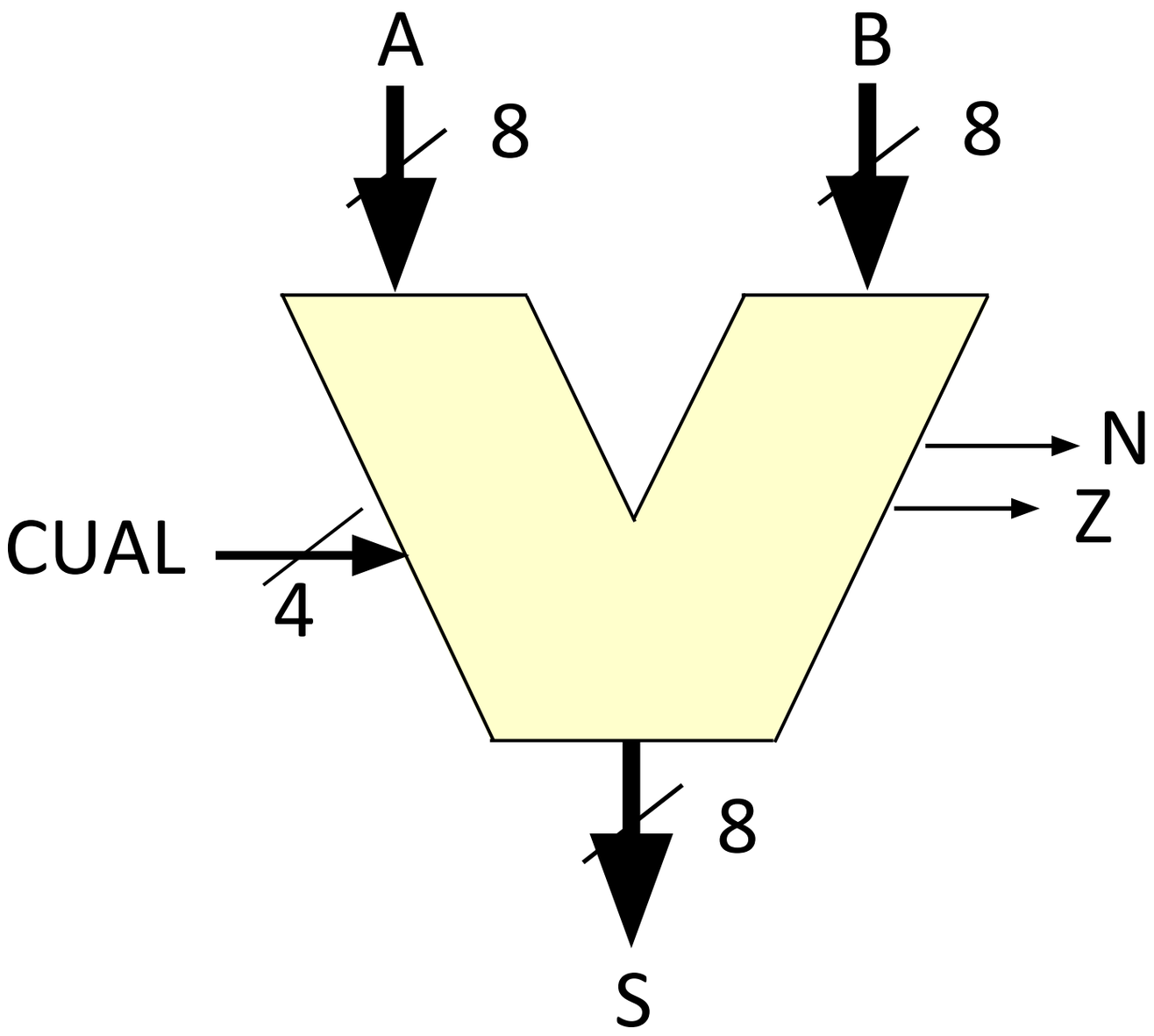


L’additionneur à n bits

- Idée : mettre en cascade *n* additionneurs complets à 1 bits.



Unité Arithmétique Logique (UAL)



CUAL	S
0000	A
0001	B
0010	A + B
0011	A - B
0100	B - A
0101	-A
0110	-B
0111	<<A
1000	<<B
1001	>>A
1010	>>B

Circuits séquentiels

- **Circuit séquentiel :** la valeur des sorties dépend de la valeur des entrées actuelles et passées.

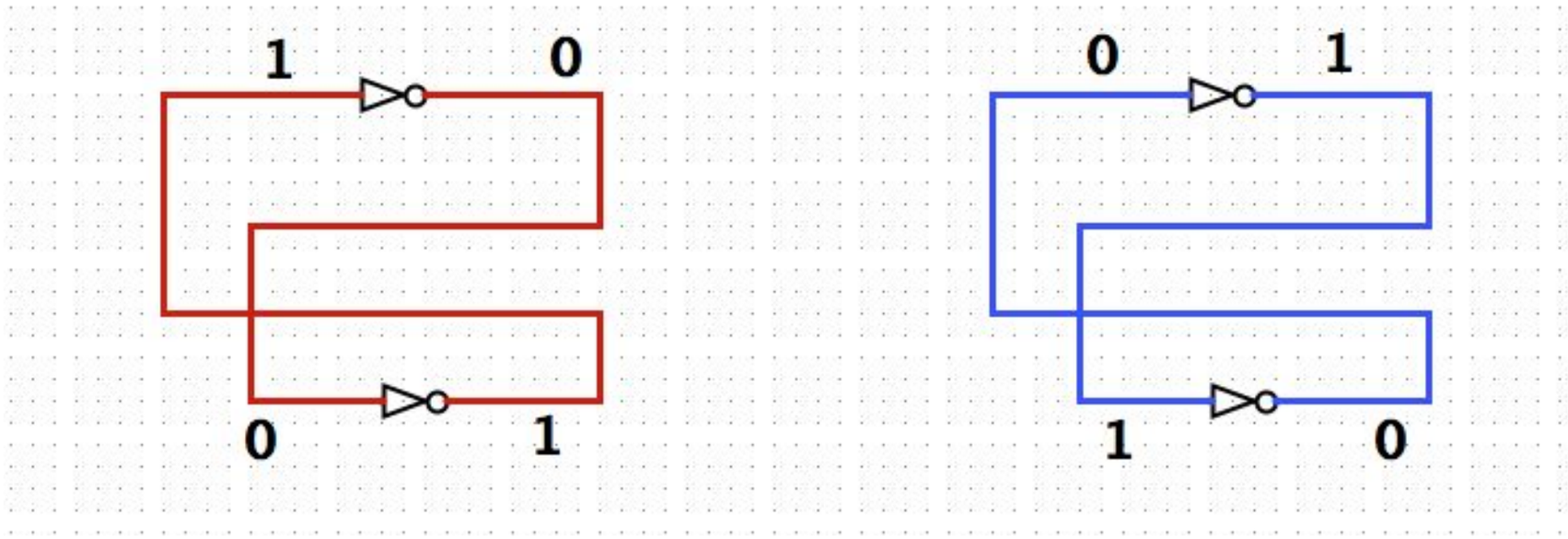
- Présence d'une boucle dans le circuit ⟶ circuit séquentiel.

- Notion de **mémoire**.

- Notion de **temps** (horloge).

Bistable

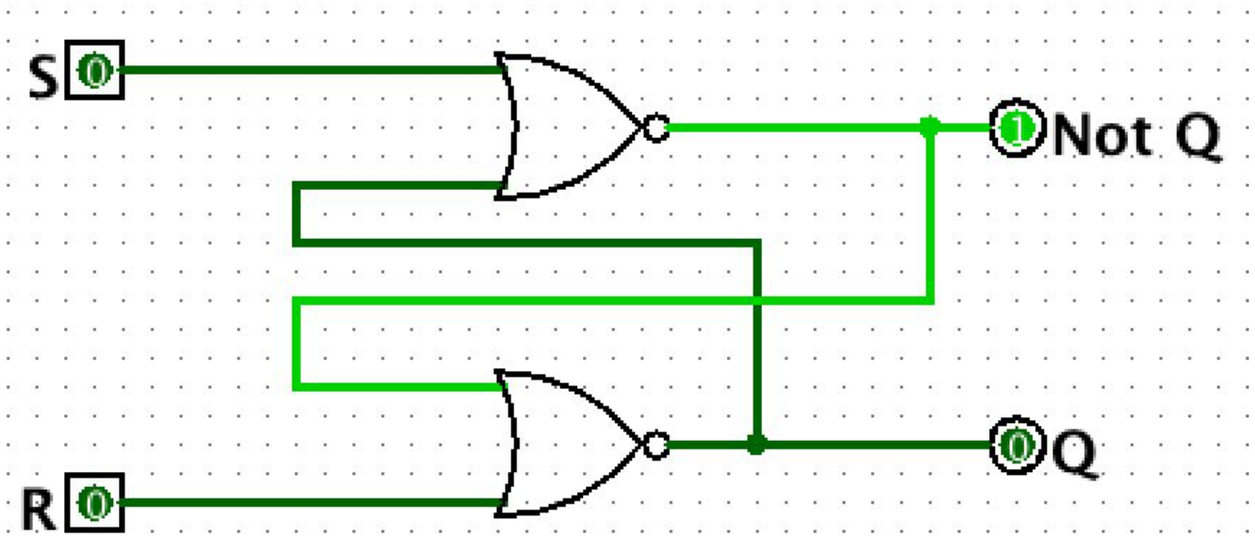
- **Bistable** : 2 valeurs stables dans le temps.



- Notion de mémorisation.

- Plusieurs façons de réaliser un bistable avec des **bascules**.
  - Une bascule est un circuit séquentiel qui réalise une bistable.

Bascule RS

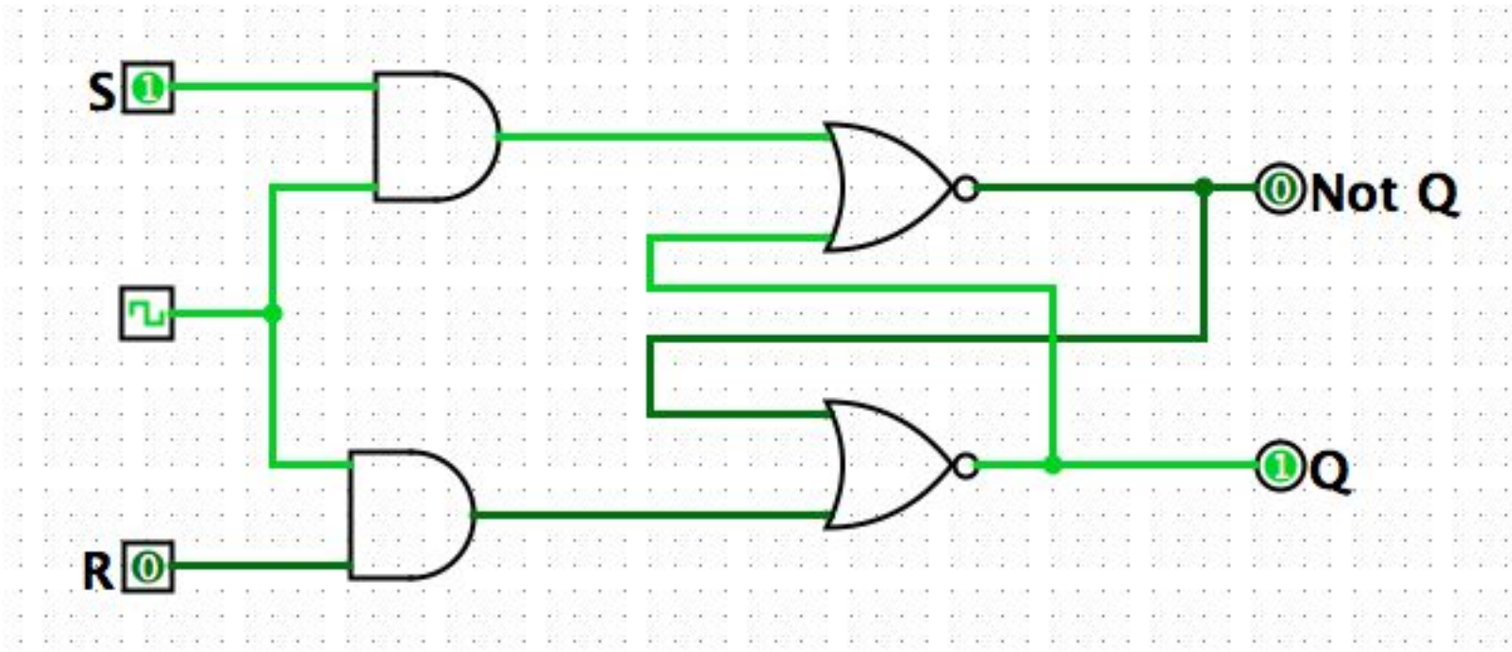


S	R	Q
0	0	val. précédente
0	1	0
1	0	1
1	1	(interdit)

← Mémorisation

Bascule RS synchrone

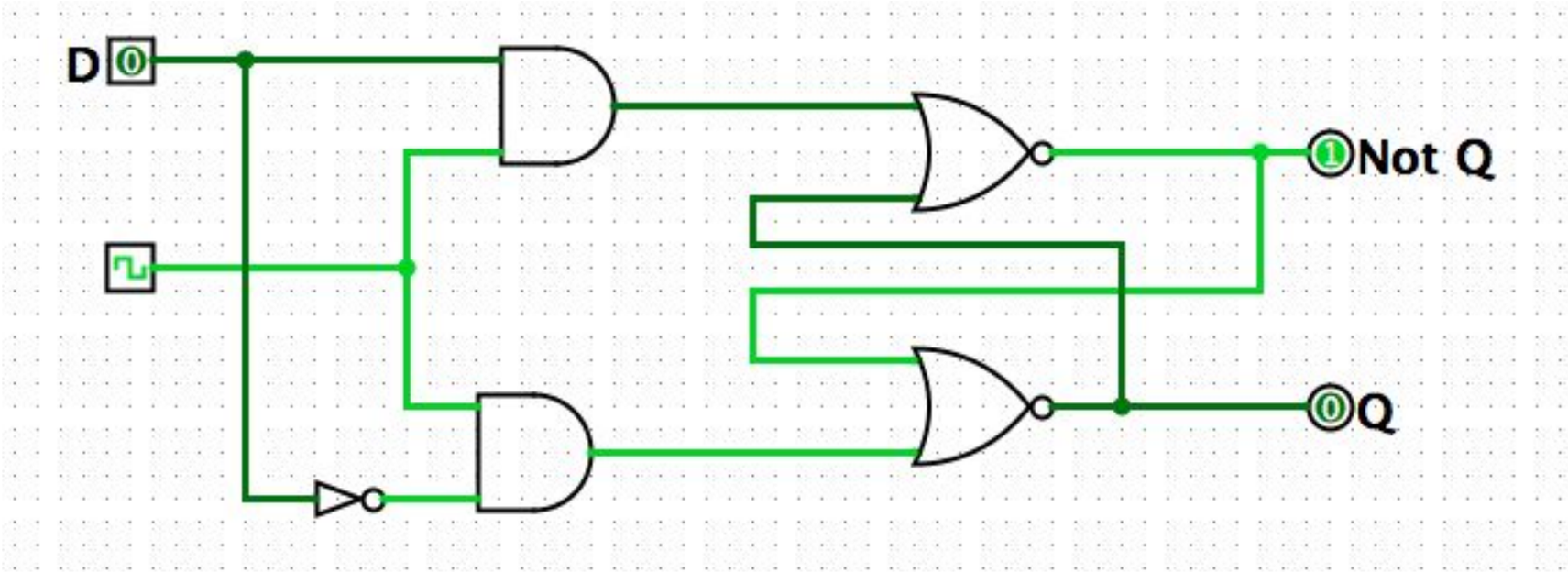
- Introduction de la notion d'**horloge** (*clock*).
  - Horloge : signal périodique (1 demi-période à 0, l'autre à 1).
- Permet la lecture d'une valeur à un instant bien déterminé.
  - pour faire face au **délai de propagation** dans un circuit logique.



- Problème : la configuration S = R = 1 devrait être interdite.
- Solution : bascule D.

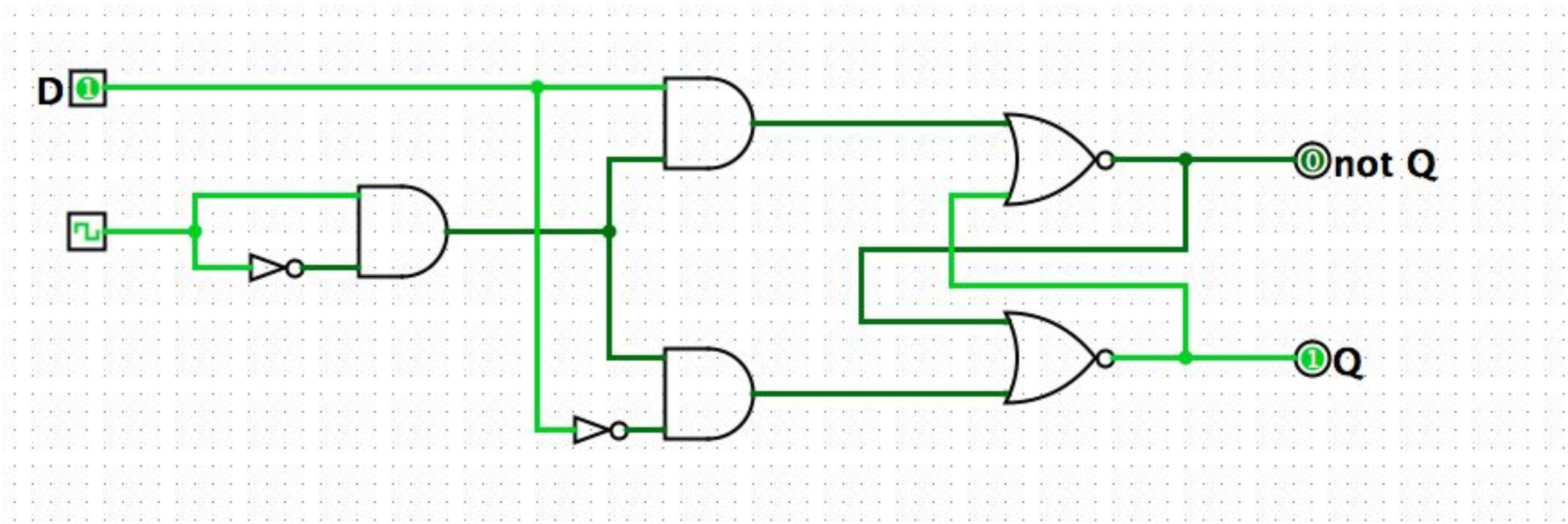
Bascule D

Bascule D : mémoire de 1 bit.



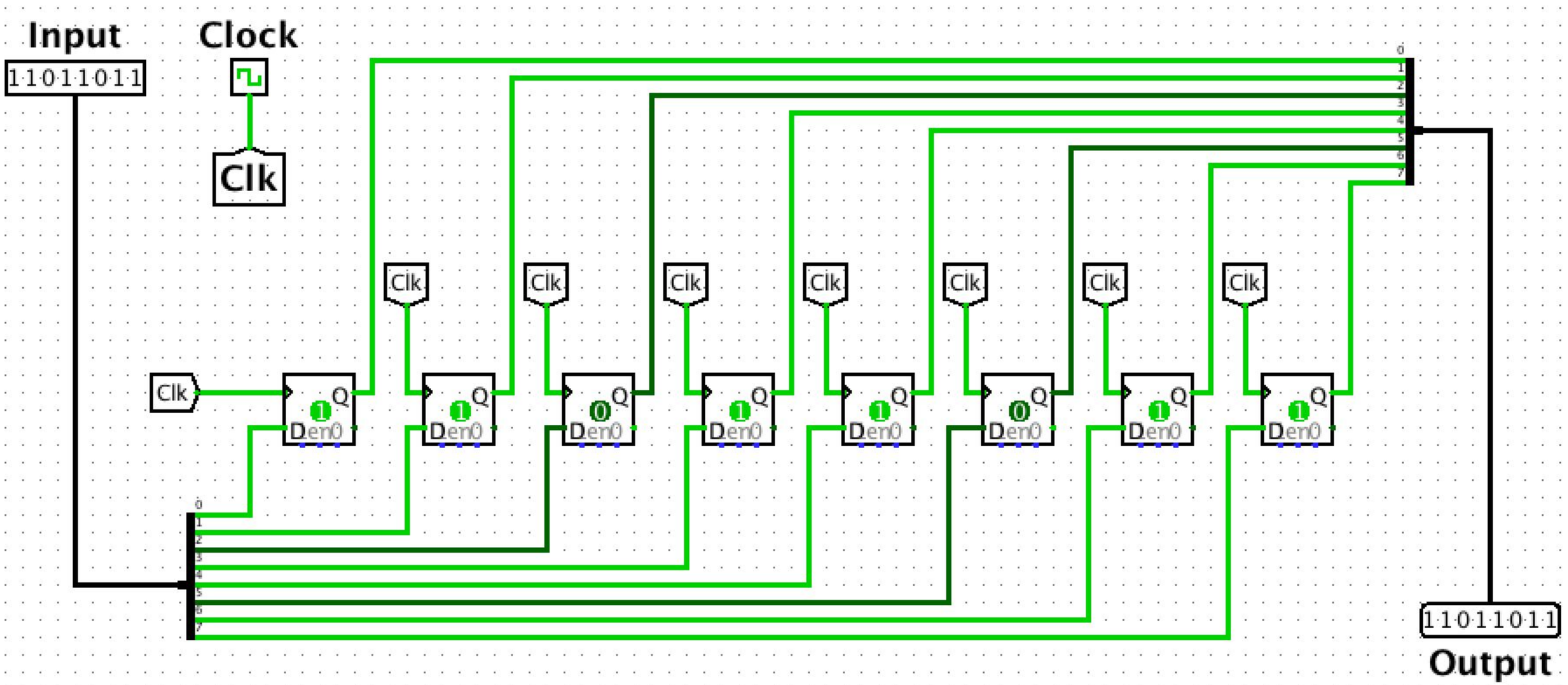
Bascule D flip-flop

- Variante de la bascule D.
- La valeur en entrée est propagée à la sortie sur le **front montant** de l'horloge



Registres

- **Registre** : mémoire rapide utilisée dans les processeurs.

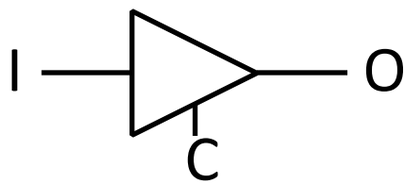


Registre 8 bits

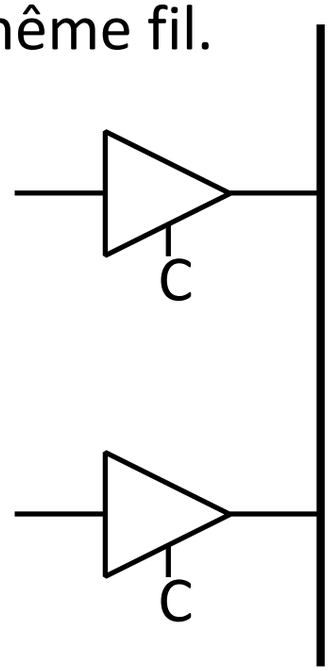
Logique à trois états

- La sortie des circuits logiques que nous avons vus peut avoir deux états :
  - **Etat 0** : niveau logique bas.
  - **Etat 1** : niveau logique haut.
- Il y a des circuits qui se basent sur une logique à trois états.
  - **Etat 0** : niveau logique bas.
  - **Etat 1** : niveau logique haut.
  - **Etat indéfini** : haute impédance.

Tampon trois états (tri-state buffer)



Utilisé pour connecter plusieurs circuits au même fil.



CentraleSupélec

université PARIS-SACLAY

LABORATOIRE DE RECHERCHE EN INFORMATIQUE

CHAPITRE II
Arithmétique des processeurs

Arithmétique des processeurs

- Un processeur est composé de plusieurs circuits logiques.
  - circuits combinatoires (par ex.: UAL).
  - circuits séquentiels (registres).

- Un circuit logique utilise seulement deux valeurs : 0 et 1 (bits).

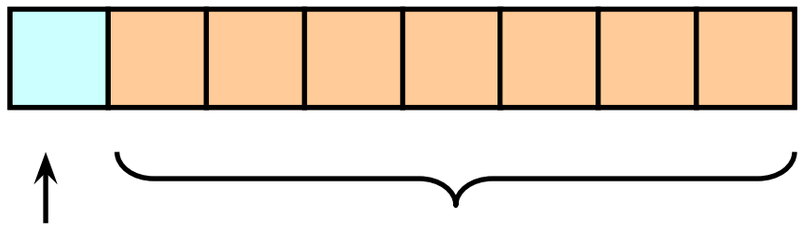
- Toute l'information manipulée par un processeur doit être codée en binaire.
  - nombres entiers, réels, textes, images…

- Pour les entiers naturels : représentation binaire classique.

- Pour les autres types de données, besoin d'un **codage**.

Entiers relatifs : 1) bit de signe

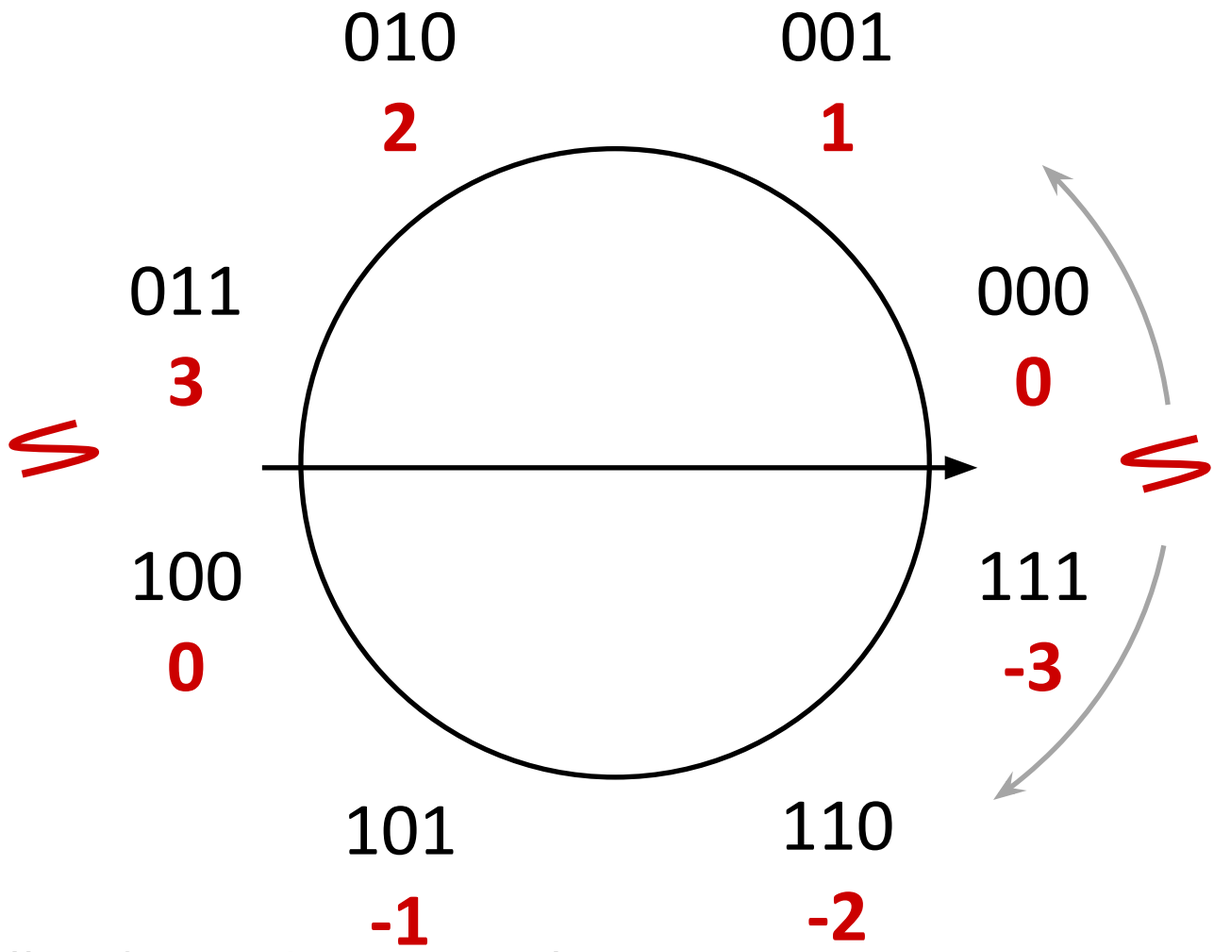
Notation pour un entier k



0 si k ≥ 0

1 si k ≤ 0

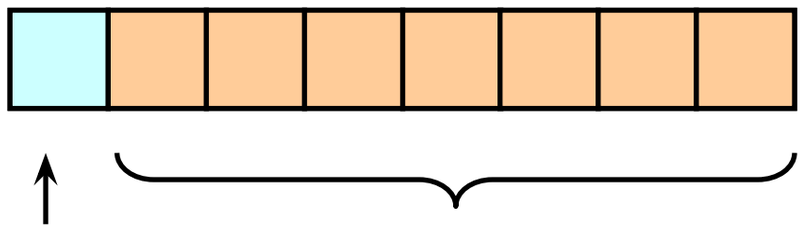
Exemple sur 3 bits



Problèmes :

- 2 représentations de 0
- addition et soustraction différentes de celles des entiers naturels

Entiers relatifs : 2) complément à 1



Valeurs entre $-2^{k-1} + 1$ et $2^{k-1} - 1$

bit de signe
0 si k ≥ 0
1 si k ≤ 0

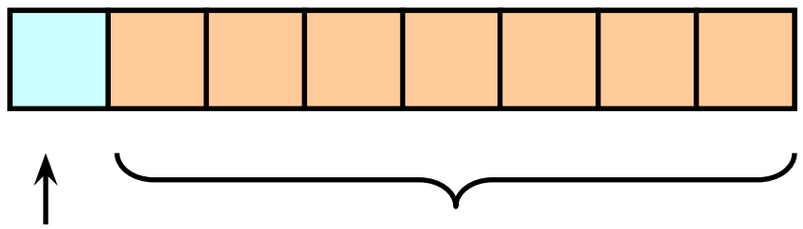
|k| ⟶ Si k ≤ 0, les chiffres de |k| sont inversés

Décimal	Positif	Négatif
0	0000	1111
1	0001	1110
2	0010	1101
3	0011	1100
4	0100	1011
5	0101	1010
6	0110	1001
7	0111	1000

Problème avec la somme si les deux opérandes ont signe négatif ou signe différent mais résultat positif.

11001 + (-6)
11010 = (-5)
10011 (-12) ⟶ résultat correct - 1

Entiers relatifs : 2) complément à 2



Valeurs entre -2^{k-1} et $2^{k-1} - 1$

bit de signe |k|

0 si k ≥ 0

1 si k ≤ 0

Si k ≤ 0, les chiffres de |k| sont inversés et on ajoute 1

Décimal	Positif	Négatif
0	0000	0000
1	0001	1111
2	0010	1110
3	0011	1101
4	0100	1100
5	0101	1011
6	0110	1010
7	0111	1001
8	-	1000

Une seule représentation pour le 0

Précision finie

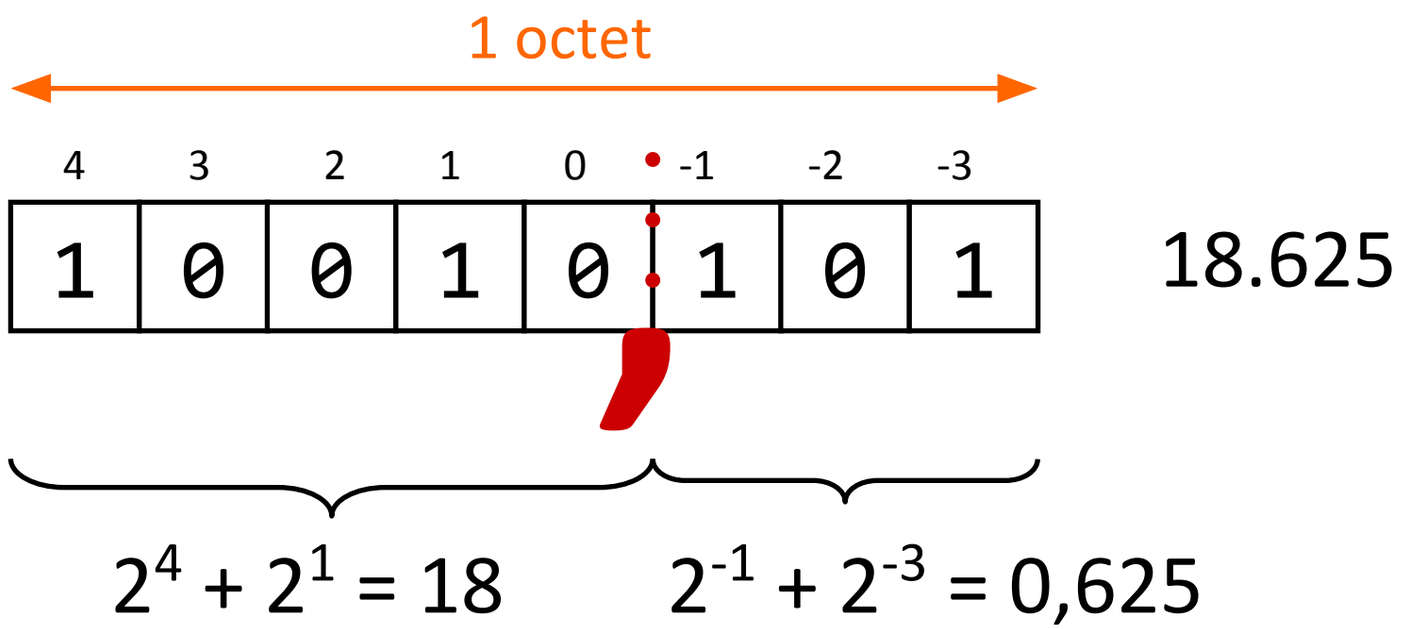
- Le processeur (l'UAL) manipule seulement des nombres ayant une taille de représentation fixée : 1, 2, 4 ou 8 octets.
- Dans les langages de programmation, cela se traduit par des types ayant un domaine fini → débordements (overflow) possibles.
- **Exemple de débordement.**
  - Sur 8 bits en complément à deux, valeur maximale 127 (0111111).
  - Qu'est-ce qu'il se passe quand on fait 120 + 8 ?
- 120 (01111000) + 8 (00001000) = -128 (10000000)!!

```
  01111000 (120) +
  00001000     (8)  =
----------------------------
10000000 (-128!)
```

Nombres en virgule fixe

- La présence d'une virgule ne change pas les algorithmes de calcul, il faut simplement ajuster la position de la virgule dans le résultat.

- Exemple : sur un octet, virgule fixe avec 3 bits après la virgule.

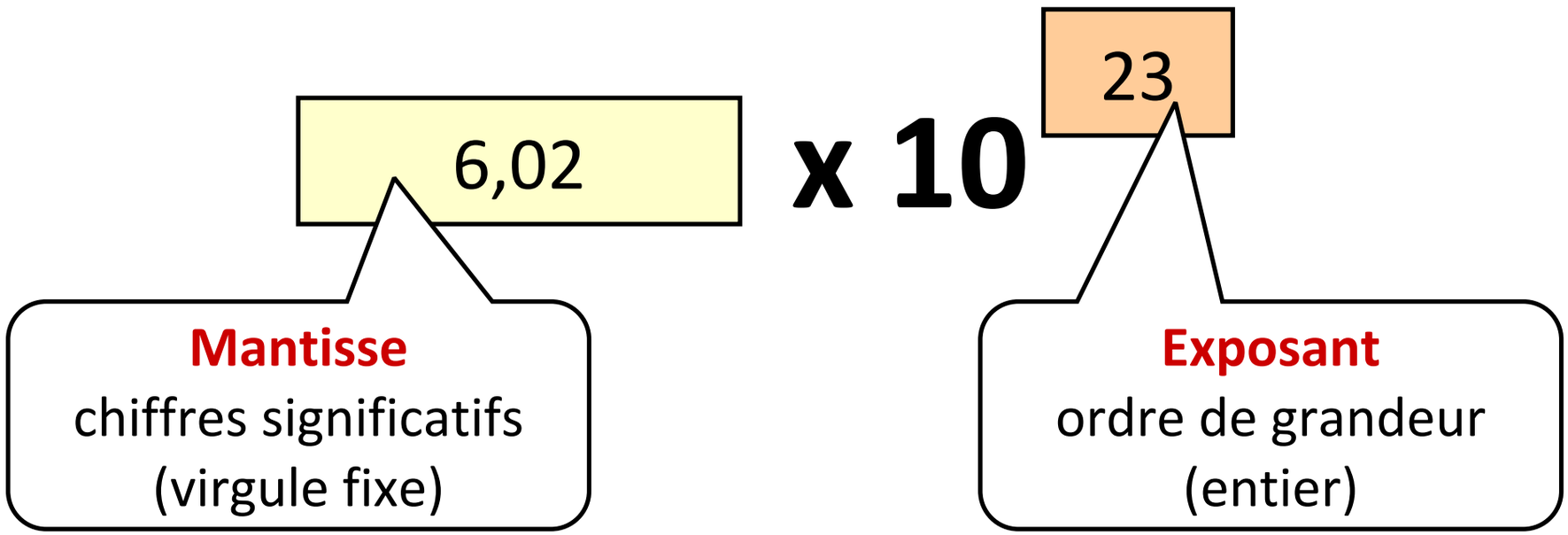


Nombres en virgule flottante

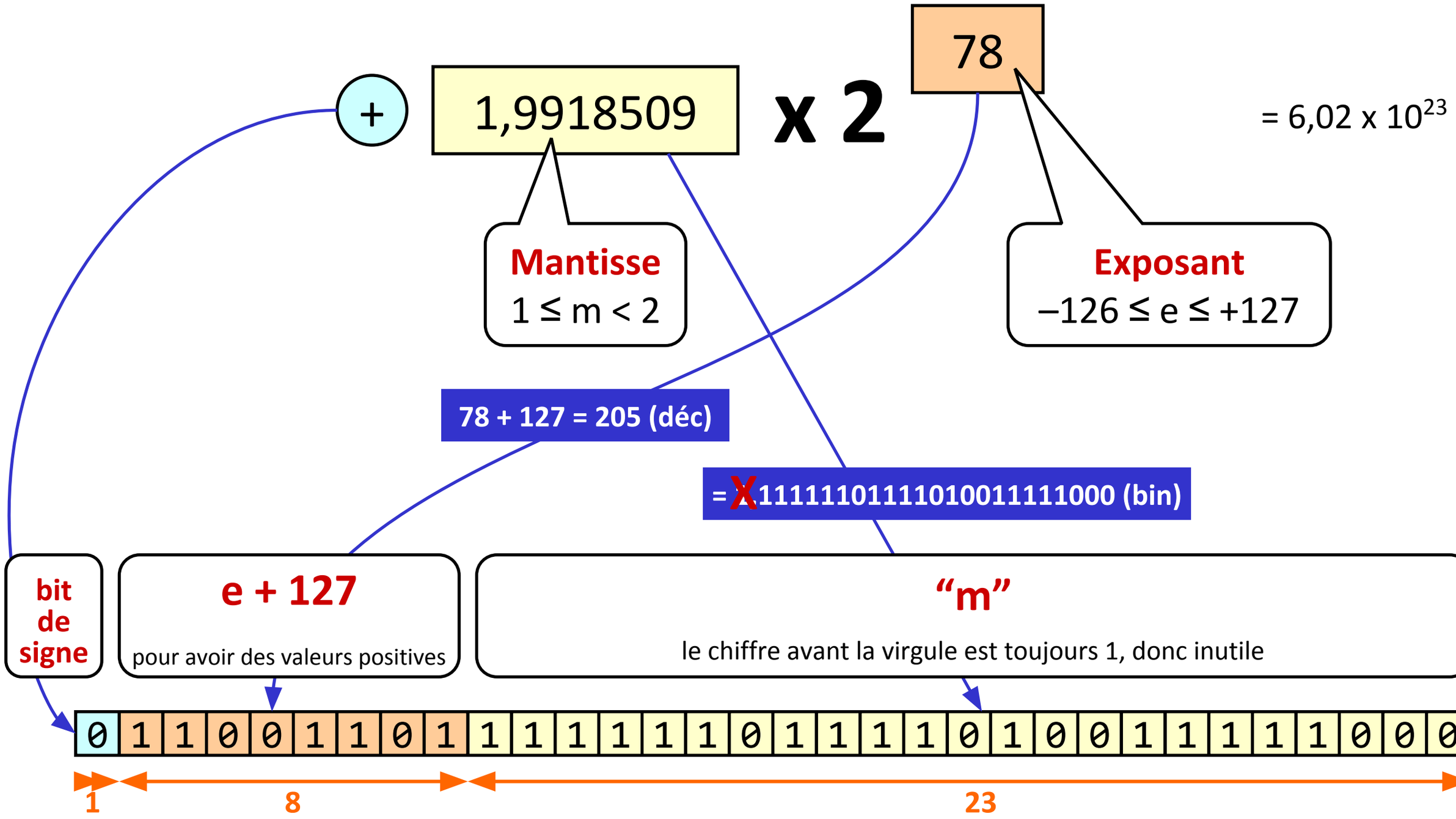
- La représentation en virgule fixe ne permet pas de manipuler des très grands nombres (ou de très petits).

- Ceux-ci sont pourtant nécessaires (physique par exemple).

- Représentation en virgule flottante dérivée de la notation scientifique :



IEEE 754 “binary32”



Notions sur le codage de caractères

- Toute donnée informatique est vue comme une séquence (finie) d'octets, c-à-d d'entiers entre 0 et 255 ($00 et $FF).

- Exemple : un fichier est une séquence d'octets, indépendamment de son contenu

```
$FF $D8 $FF $E0 $00 $10 $4A $46 $49
$46 $00 $01 $01 $01 $01 $2C...
```


```
$50 $4B $03 $04 $14 $00 $06 $00 $08
$00 $00 $00 $21 $00 $4B $51...
```

- Besoin de conventions pour coder/décoder les informations

Unicode - Implémentations

- **UTF-8** : codage à longueur variable

Forme	Bits utiles	Code points
0xxxxxxx	1 à 7	0 à 127
110xxxxx **10**xxxxxx	8 à 11	128 à 2 047
1110xxxx **10**xxxxxx **10**xxxxxx	12 à 16	2 048 à 65 535
11110xxx **10**xxxxxx **10**xxxxxx **10**xxxxxx	17 à 21	65 536 à 1 114 111

𝄞U+1D11E →décimal 119 070 → binaire 000 011101 000100 011110 (21 bits)
--> Code UTF-8 **11110**000 **10**011101 **10**000100 **10**011110

- **UTF-16 :** les code points plus utilisés sur 16 bits, les autres sur 32 bits (avec des calculs…pas détaillés).
- **UTF-32 :** représentation d'un code point sur 32 bits.

CentraleSupélec

université PARIS-SACLAY

LABORATOIRE DE RECHERCHE EN INFORMATIQUE

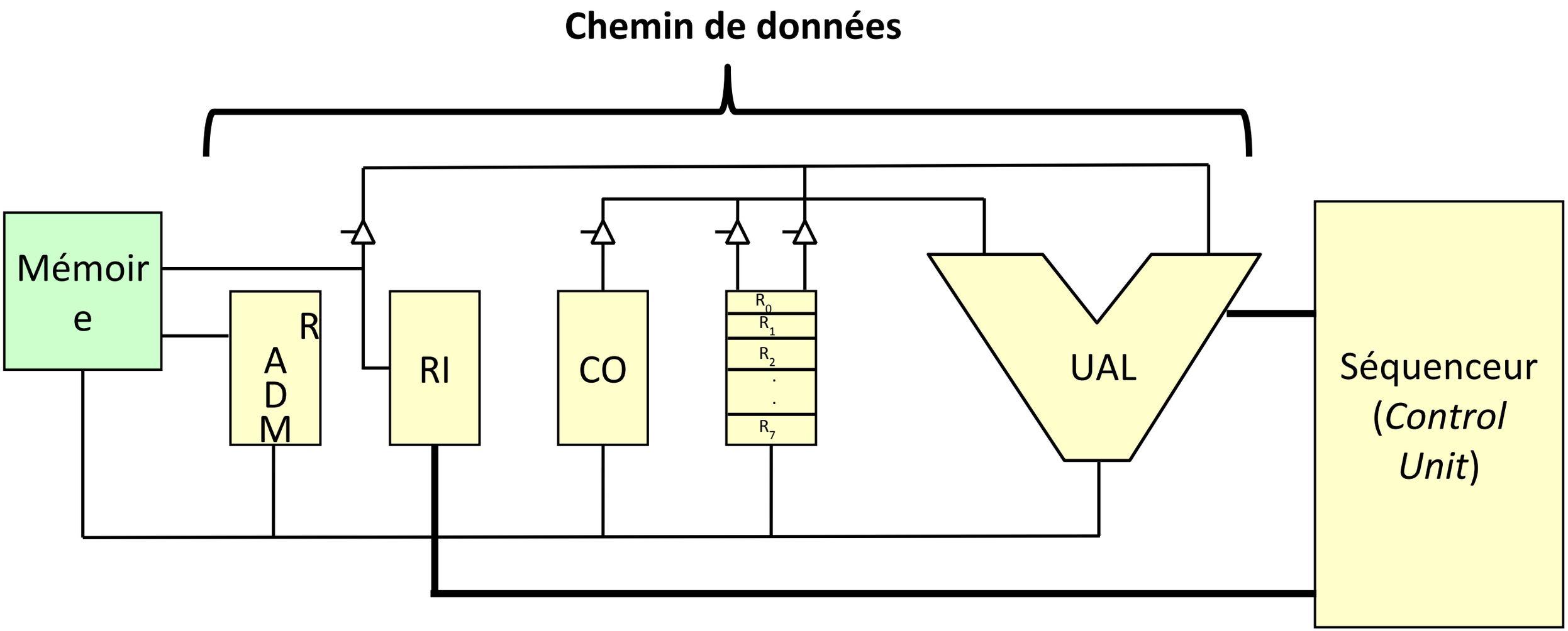
CHAPITRE III

Le processeur

Le processeur

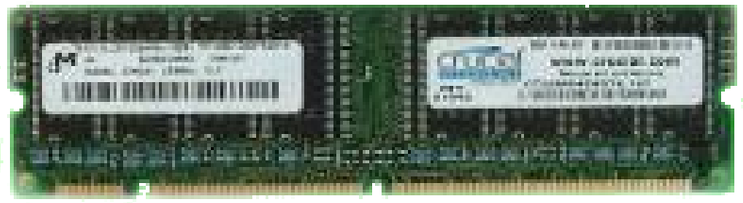
- **Processeur** : cerveau de l'ordinateur.
- Il se compose de deux parties.

Rôle : Exécuter des **programmes** (séquence d'**instructions**)

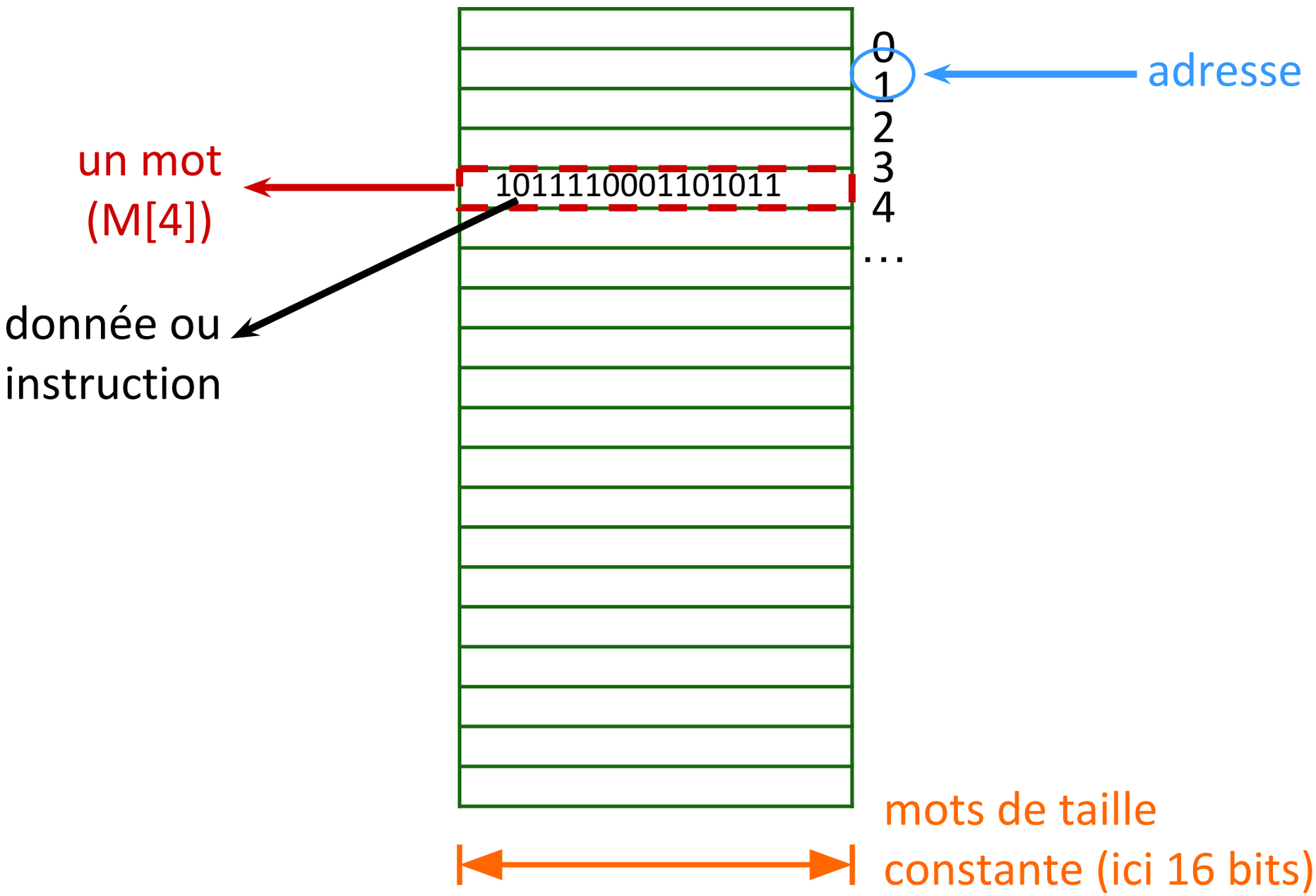


- **Chemin de données** : registres, UAL, bus.
- **Séquenceur** : régit le fonctionnement du chemin de données.

La mémoire

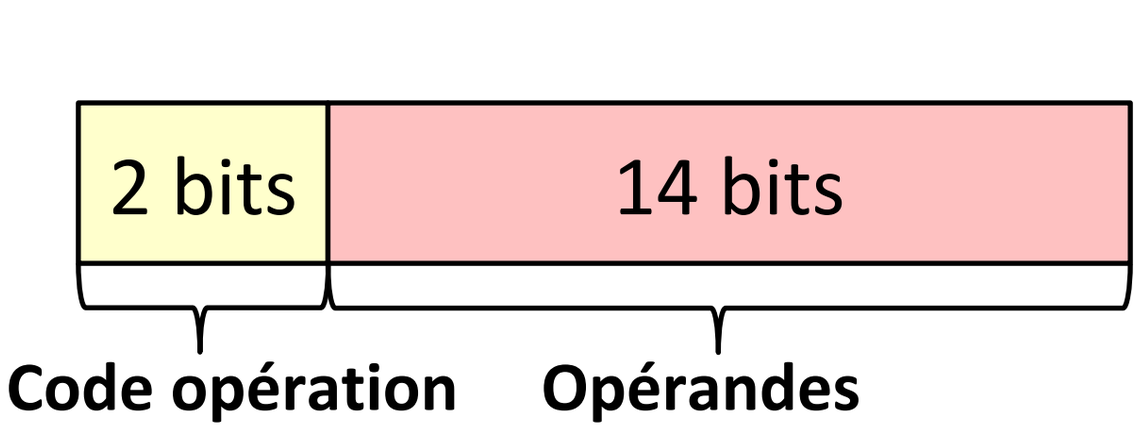


Mémoire



Instructions

- **Instruction** : opération élémentaire qu'un processeur peut effectuer.
  - nombre binaire sur 16 bits.
- **Jeu d'instructions** : ensemble d'instructions qu'un processeur peut exécuter.
- **Exemple** : un processeur avec quatre instructions (plus tard on verra un exemple beaucoup plus complexe!!).
  - **LOAD** rx a : $R_{rx} \leftarrow M[a]$.
  - **STORE** rx a : $M[a] \leftarrow R_{rx}$.
  - **ADD** rx ry rz : $R_{rx} \leftarrow R_{ry} + R_{rz}$.
  - **SUB** rx ry rz : $R_{rx} \leftarrow R_{ry} - R_{rz}$.
- Soit *I* un nombre binaire sur 16 bits. Comment le processeur sait-il à quelle instruction *I* correspond?



Codes opérations
- 00 : LOAD
- 01 : STORE
- 10 : ADD
- 11 : SUB

Choix arbitraire!!

Instructions

- **LOAD** 0 5 : $R_0 \leftarrow M[5]$

00	00000000000101

- **LOAD** 1 6 : $R_1 \leftarrow M[6]$

00	00100000000110

- **ADD** 2 0 1 : $R_2 \leftarrow R_0 + R_1$

10	0100000001XXXXX

- **STORE** 2 6 : $M[6] \leftarrow R_2$

01	01000000000110

Codes opérations
- 00 : LOAD
- 01 : STORE
- 10 : ADD
- 11 : SUB

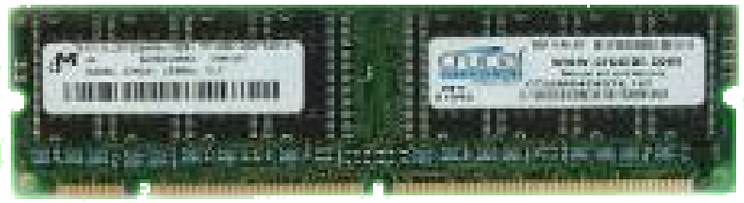
Programme

Langage machine		Assembleur
00	00000000000101	**LOAD** 0 5
00	00100000000110	**LOAD** 1 6
10	0100000010XXXXX	**ADD** 2 0 1
01	01000000000110	**STORE** 2 6

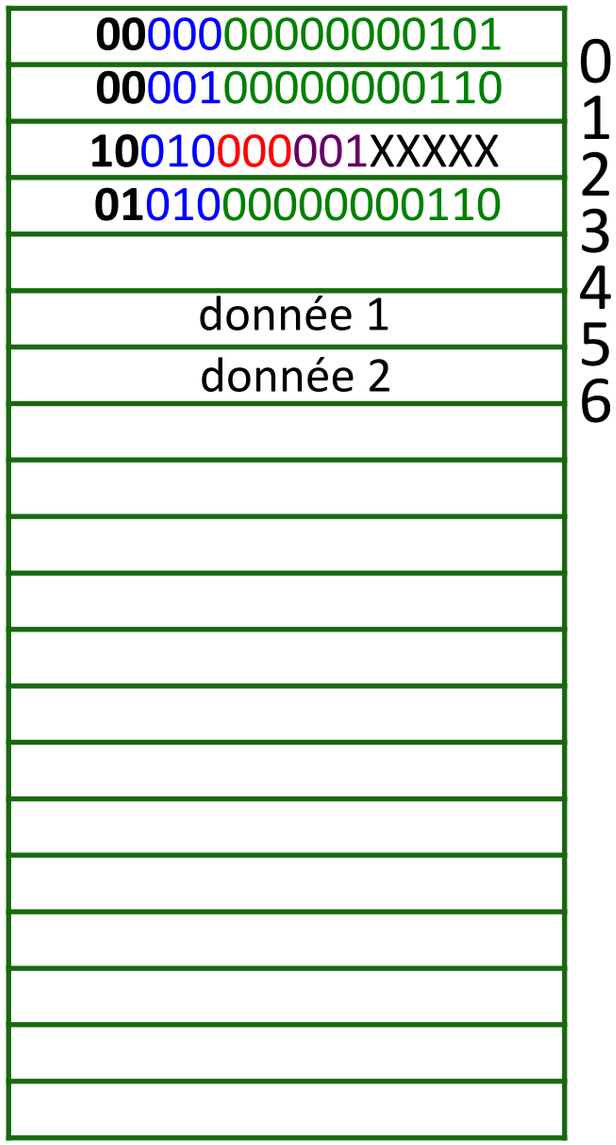
Codes opérations
- 00 : LOAD
- 01 : STORE
- 10 : ADD
- 11 : SUB

Exécution d'un programme

- Les instructions d'un programme sont stockées à des adresses contiguës (la réalité est plus complexe).
- Le processeur lit de la mémoire les instructions du programme une par une.
- L'adresse de la prochaine instruction à lire est stockée dans le **Compteur Ordinal** (**CO**) – Program Counter (PC) – un registre du processeur.



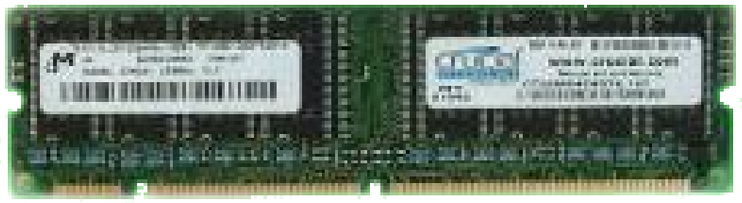
Mémoire



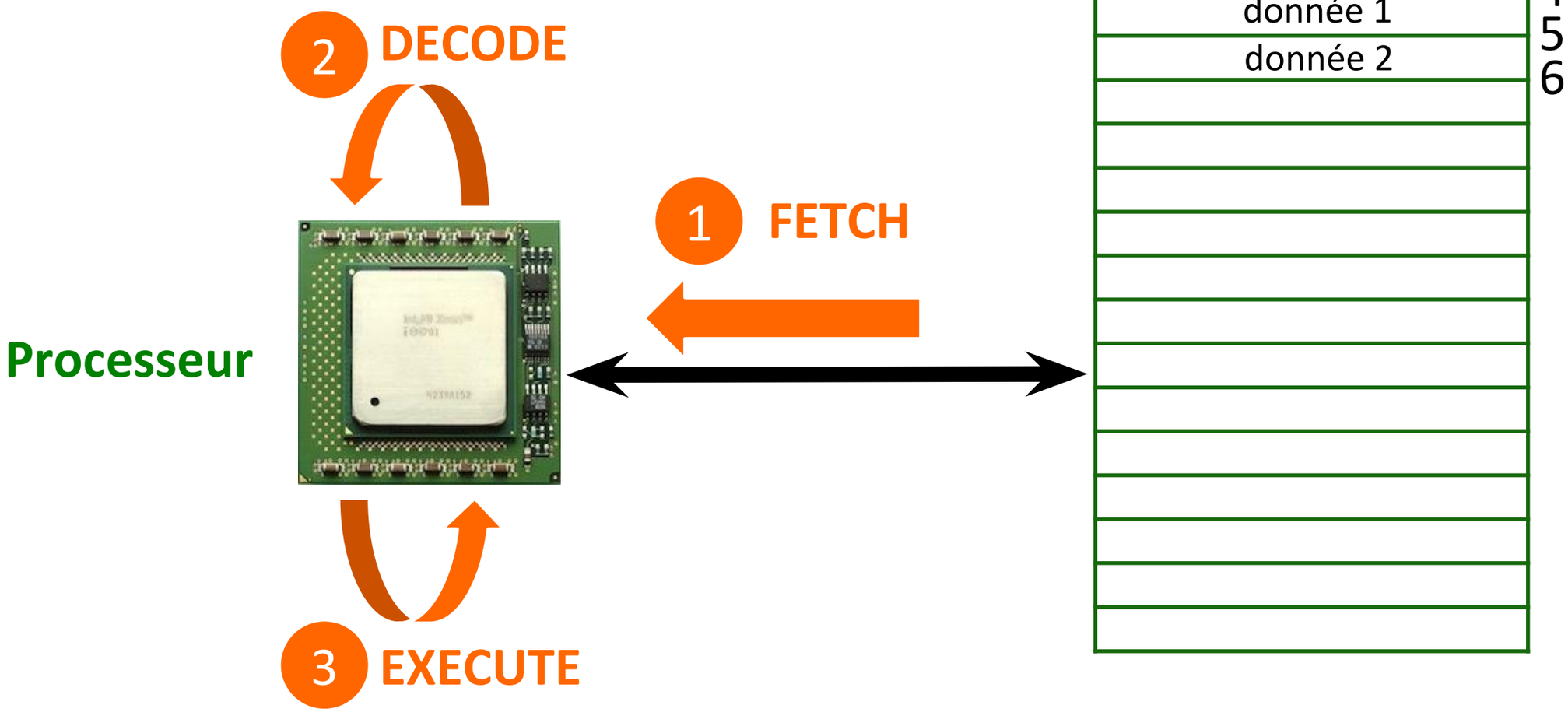
Processeur

Exécution d'un programme

1. **Fetch** : l'instruction contenue à l'adresse indiquée par le CO est copiée dans le **registre d'instruction (RI) –** ou *Instruction Register* (IR) – un registre du processeur.
2. **Decode** : Le processeur « comprend » l'instruction à exécuter.
3. **Execute** : L'instruction est exécutée par le processeur. La valeur contenue dans le CO est incrémenté.

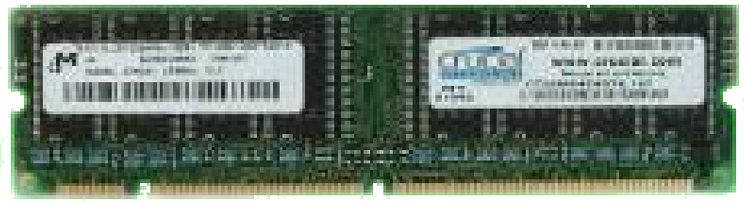


Mémoire



Architecture de Von Neumann

- L'architecture que nous avons présentée a été décrite en 1945 par John Von Neumann.
- L'architecture de Von Neumann est à la base du fonctionnement des ordinateurs modernes.
- Trois principes à retenir:
  1. La mémoire est une matrice de mots, chaque mot a son adresse.
  2. Un mot peut contenir soit une donnée soit une instruction.
  3. Les mots ont une taille constante.



Mémoire

0000000000000101	0
0000100000000110	1
1001000000 01XXXXX	2
0101000000000110	3
	4
donnée 1	5
donnée 2	6

Processeur

MiniArm – Notre processeur du cours

- Nous allons définir *MiniArm*, un processeur simplifié.
  - cependant suffisamment complexe pour comprendre les principes qui sont à la base du fonctionnement d'un processeur réel.

- Registres : RI, CO, RADM, 8 registres banalisés (R_0,…, R_7), registre d'état (SR).
  - tous les registres sont à 16 bits.

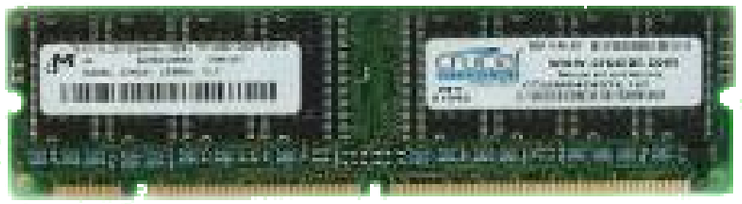
- UAL à 16 bits.

- Jeu de 9 instructions.
  - petit mais suffisant pour coder des programmes complexes.
  - instructions de **branchement** (jump).

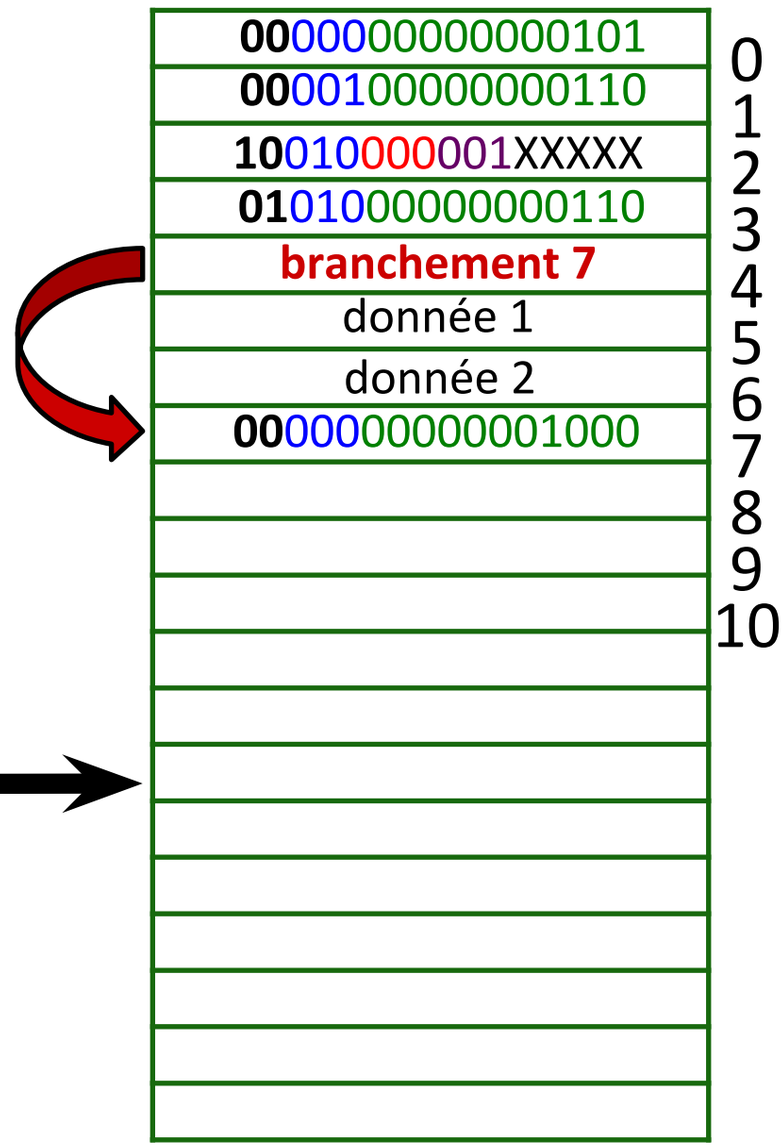
- Introduction de la notion de **mode d'adressage**.

Instructions de branchement

- Instruction de **branchement** : saute à une adresse spécifiée pour poursuivre l'exécution du programme.
- Utile quand on veut faire appel à une fonction dont on connaît l'adresse de la première instruction.



Mémoire



Processeur

Mode d'adressage

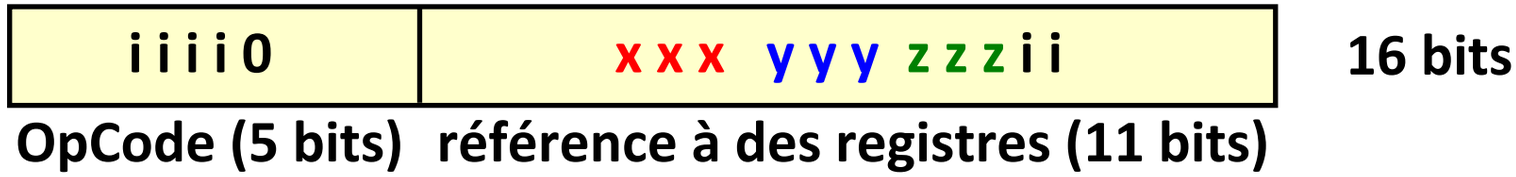
- Une instruction peut avoir zéro ou plusieurs opérandes.
  - opérandes **sources** : ceux qui ne sont pas changés après l'exécution de l'instruction.
  - opérandes **cibles** : ceux qui sont changés après l'exécution de l'instruction.
  - **Exemple :** dans **LOAD** 0 5, 0 est l'opérande cible, 5 est l'opérande source.
- **Mode d'adressage** : règle qui régit la manière dont une instruction identifie ses opérandes.
  - pour simplifier, nous considérons seulement les opérandes sources.
- Dans l'instruction **LOAD** 0 5 :
  - L'opérande source (5) est *immédiatement* disponible dans l'instruction (il ne faut pas le chercher dans un registre).
  - Ceci est un exemple de mode d'**adressage immédiat**.
- Dans l'instruction **ADD** 2 0 1 :
  - L'instruction contient les identifiants des deux registres (0 et 1) qui contiennent les opérandes sources.
  - Ces opérandes ne sont pas immédiatement disponibles dans l'instruction.
    - il faut les chercher dans les deux registres.
  - Ceci est un exemple de mode d'**adressage direct**.

Modes d'adressage de *MiniArm*

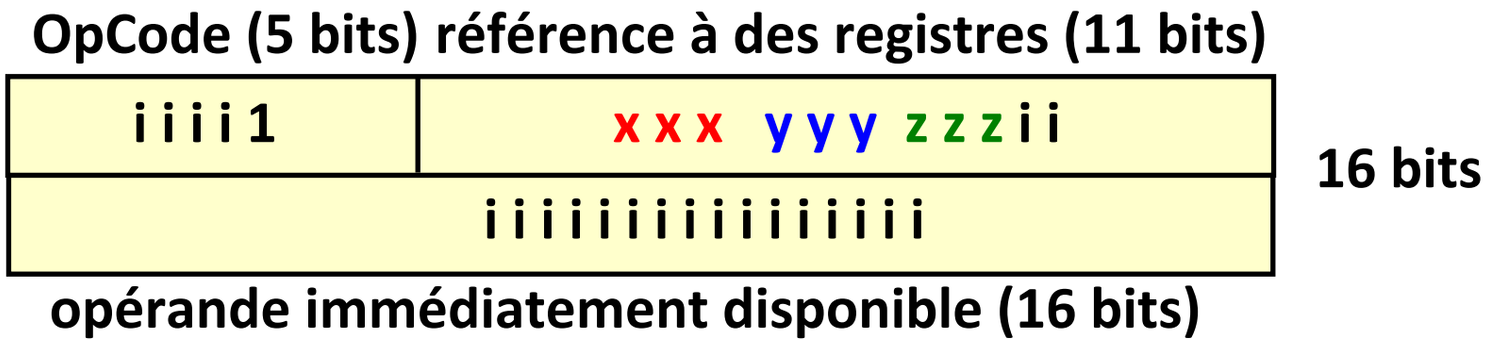
- Mode d'**adressage immédiat** : l'opérande est codé dans l'instruction.
  - *immédiatement* disponible.
- Mode d'**adressage direct** : l'opérande est dans un registre (dont l'identifiant est codé dans l'instruction).
  - disponible à partir *directement* de l'instruction.
  - il suffit de le chercher dans le registre indiqué dans l'instruction.
- Chaque instruction de *MiniArm* a une version en adressage immédiat et en adressage direct pour les opérandes sources.
  - les opérandes cibles sont toujours des identifiants de registres.
  - quelques exceptions que nous allons voir.
- Normalement un processeur réel a plus de deux modes d'adressages.
  - pour les opérandes sources et cibles.

Format des instructions de *MiniArm*

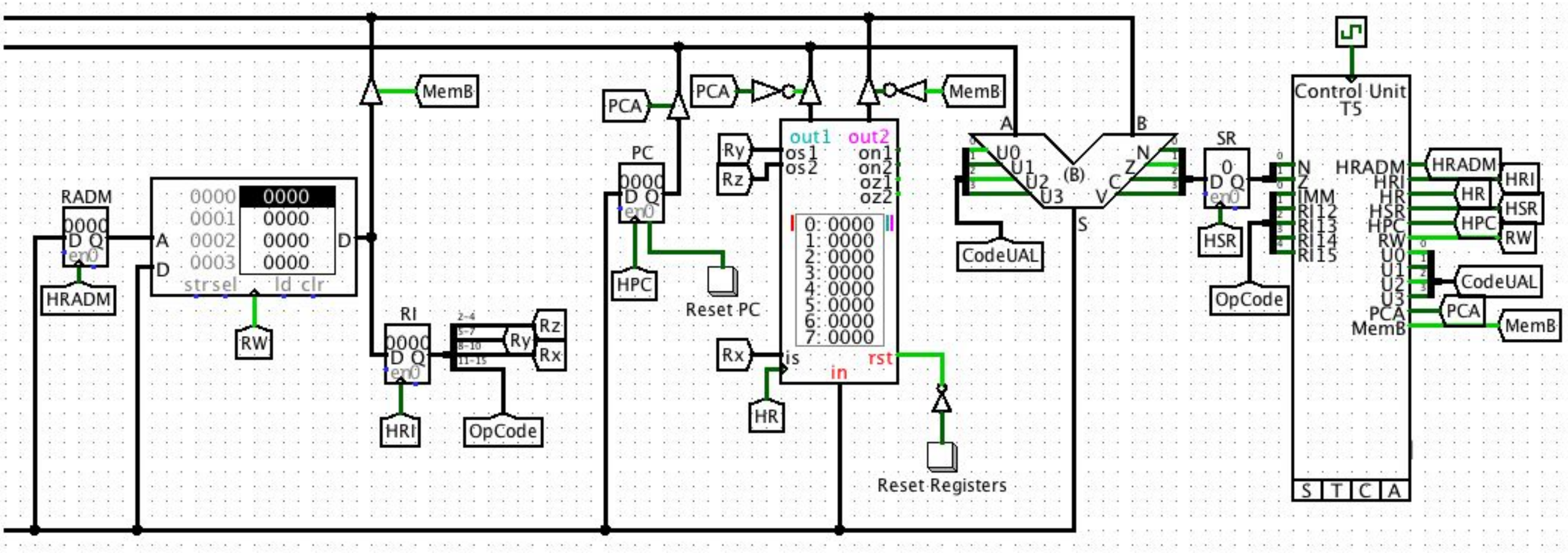
- Les instructions qui utilisent l'**adressage direct** sont codées sur **1 mot**.
- Le dernier bit de *OpCode* à **0** indique que cette instruction utilise un adressage direct.
- Possibilité de référencer jusqu'à 3 registres banalisés (3 bits pour chaque registre).



- Les instructions qui utilisent l'**adressage immédiat** sont codées sur **2 mots**.
- Le dernier bit de *OpCode* à **1** indique que cette instruction utilise un adressage immédiat.
- Le deuxième mot contient la valeur de l'opérande immédiatement disponible.



Le processeur *MiniArm*



OpCode (5 bits)	rx (3 bits)	ry (3 bits)	rz (3 bits)	
opérande auxiliaire (16 bits)				

Instructions de *MiniArm*

- **LDR** : copie la valeur d'un mot de mémoire dans un registre.
- **STR** : copie la valeur d'un registre dans un mot de mémoire.
- **MOV** : copie une valeur d'un registre dans un autre registre.
- **ADD** : copie la somme des valeurs de deux registres dans un registre.
- **SUB** : copie la soustraction des valeurs de deux registres dans un registre.
- **CMP** : compare les valeurs de deux registres et modifie les signaux du registre d'état en conséquence.
- **BEQ, BLT, B** : instructions de branchement.

LDR

- **LDR adressage direct : ldr** rx, ry
- **Effet** : $R_{rx} \leftarrow M[R_{ry}]$

OpCode (5 bits)	3 bits	3 bits	3 bits	2 bits
00000	rx	ry		

- **LDR adressage immédiat : ldr** rx, a
- **Effet** : $R_{rx} \leftarrow M[a]$

OpCode (5 bits)	3 bits	3 bits	3 bits	2 bits
00001	rx			
	a			

STR

- **STR adressage direct : str** rz, ry
- **Effet** : M[R_{ry}]← R_{rz}

OpCode (5 bits)	3 bits	3 bits	3 bits	2 bits
00010		ry	rz	

- **STR adressage immédiat : str** rz, a
- **Effet** : M[a] ← R_{rz}

OpCode (5 bits)	3 bits	3 bits	3 bits	2 bits
00011			rz	
		a		

MOV

- **MOV adressage direct : mov** rx, ry
- **Effet** : $R_{rx} \leftarrow R_{ry}$

OpCode (5 bits)	3 bits	3 bits	3 bits	2 bits
00100	rx	ry		

- **MOV adressage immédiat : mov** rx, v
- **Effet** : $R_{rx} \leftarrow v$

OpCode (5 bits)	3 bits	3 bits	3 bits	2 bits
00101	rx			
	v			

ADD

- **ADD adressage direct : add** rx, ry, rz
- **Effet** : $R_{rx} \leftarrow R_{ry} + R_{rz}$

OpCode (5 bits)	3 bits	3 bits	3 bits	2 bits
00110	rx	ry	rz	

- **ADD adressage immédiat : add** rx, ry, v
- **Effet** : $R_{rx} \leftarrow R_{ry} + v$

OpCode (5 bits)	3 bits	3 bits	3 bits	2 bits
00111	rx	ry		
	v			

SUB

- **SUB adressage direct : sub** rx, ry, rz
- **Effet** : $R_{rx} \leftarrow R_{ry} - R_{rz}$

OpCode (5 bits)	3 bits	3 bits	3 bits	2 bits
01000	rx	ry	rz	

- **SUB adressage immédiat : sub** rx, ry, v
- **Effet** : $R_{rx} \leftarrow R_{ry} - v$

OpCode (5 bits)	3 bits	3 bits	3 bits	2 bits
01001	rx	ry		
	v			

CMP

- **CMP adressage direct : cmp** ry, rz
- **Effet** : N ⟵ (R_{ry} < R_{rz}); Z ⟵ (R_{ry} = R_{rz})

OpCode (5 bits)	3 bits	3 bits	3 bits	2 bits
01010		ry	rz	

- **CMP adressage immédiat : cmp** ry, v
- **Effet** :N ⟵ (R_{ry} < v); Z ⟵ (R_{ry} = v)

OpCode (5 bits)	3 bits	3 bits	3 bits	2 bits
01011		ry		
		v		

BEQ

- **BEQ adressage direct : beq** rz
- **Effet** : CO ⟵ R_{rz} si Z=1

OpCode (5 bits)	3 bits	3 bits	3 bits	2 bits
01100			rz	

- **BEQ adressage immédiat : beq** a
- **Effet** : CO ⟵ a si Z=1

OpCode (5 bits)	3 bits	3 bits	3 bits	2 bits
01101				
a				

BLT

- **BLT adressage direct : blt** rz
- **Effet** : CO ⟵ R_{rz} si N=1

OpCode (5 bits)	3 bits	3 bits	3 bits	2 bits
01110			rz	

- **BLT adressage immédiat : blt** a
- **Effet** : CO ⟵ a si N=1

OpCode (5 bits)	3 bits	3 bits	3 bits	2 bits
01111				
a				

B

- **B adressage direct : b** rz
- **Effet** : CO ← R_{rz}

OpCode (5 bits)	3 bits	3 bits	3 bits	2 bits
10000			rz	

- **B adressage immédiat : b** a
- **Effet** : CO ← a

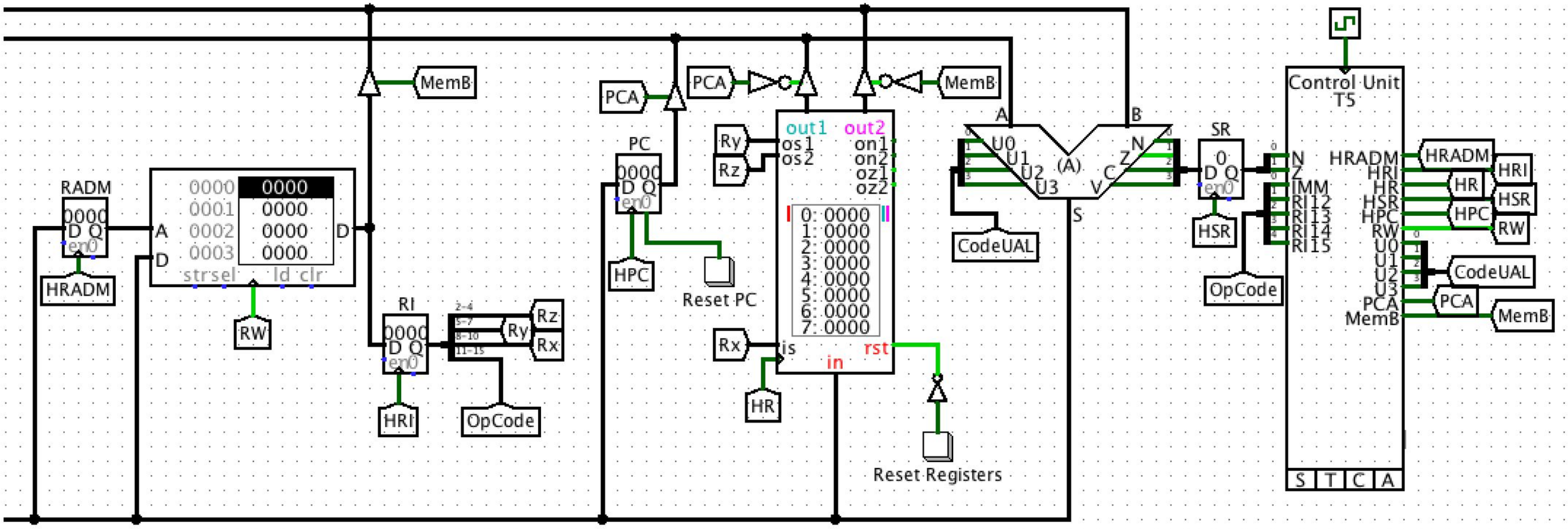
OpCode (5 bits)	3 bits	3 bits	3 bits	2 bits
10001				
a				

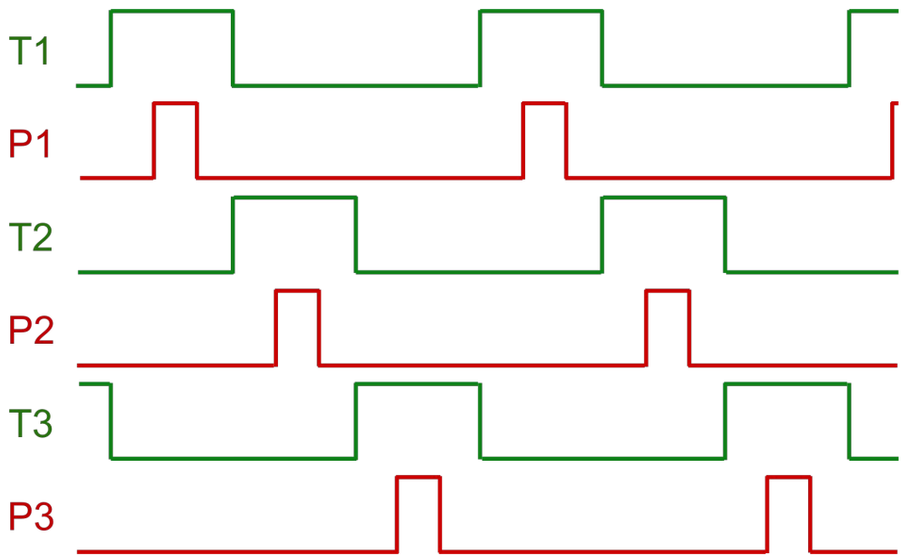
Exécution des instructions

- Pour exécuter des instructions, il faut piloter le chemin des données.
  - il faut changer les valeurs des signaux (COA, MemB, CodeUAL, …) pour que les données aillent « aux bons endroits ».
  - exemple de ADD.

- Le chemin de données est piloté par le **séquenceur**.

Le séquenceur de *MiniArm*





Signal T : temps de **configuration**

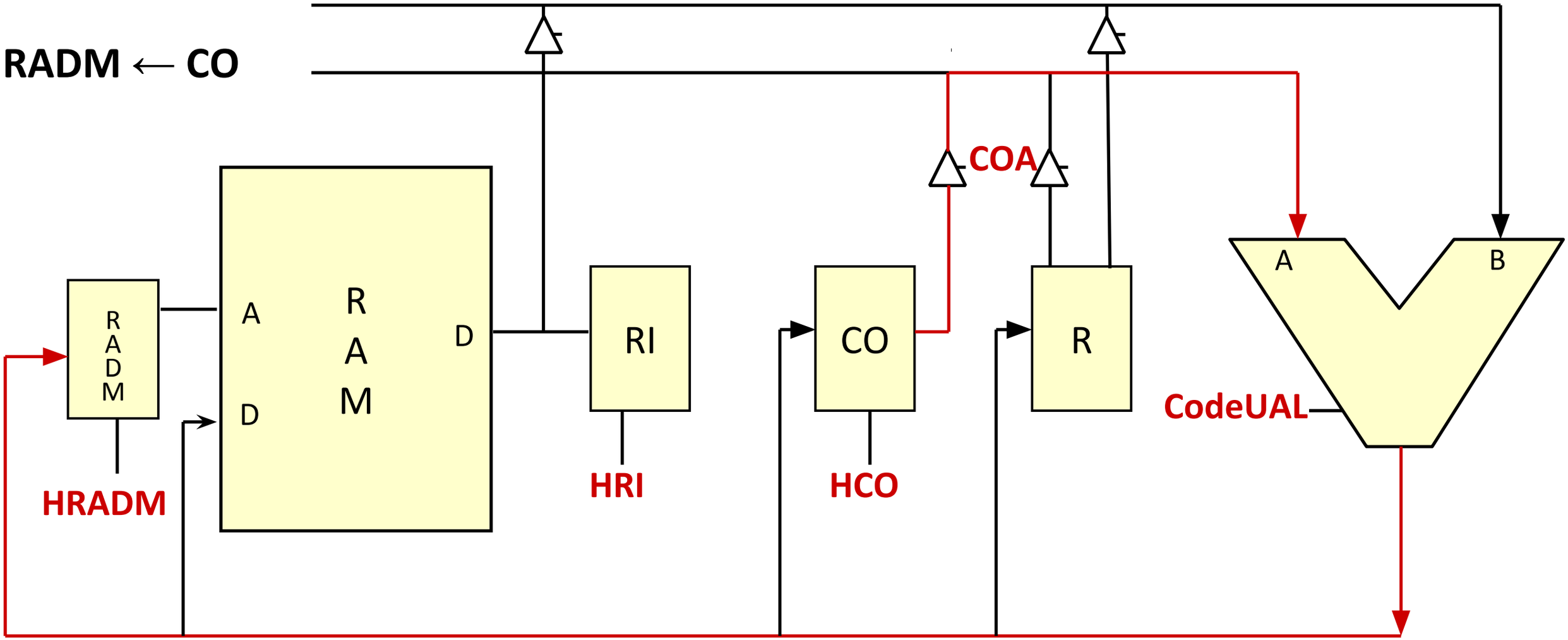
Signal P : temps de **mémorisation**

Réalisation du séquenceur – Étapes

- Création d'un tableau des temps
  - Une ligne pour chaque instruction
  - Une colonne pour chaque temps d'horloge (T1, P1, T2, P2..)
  - Pour chaque temps, les signaux du chemin de données à activer (= mettre à 1)

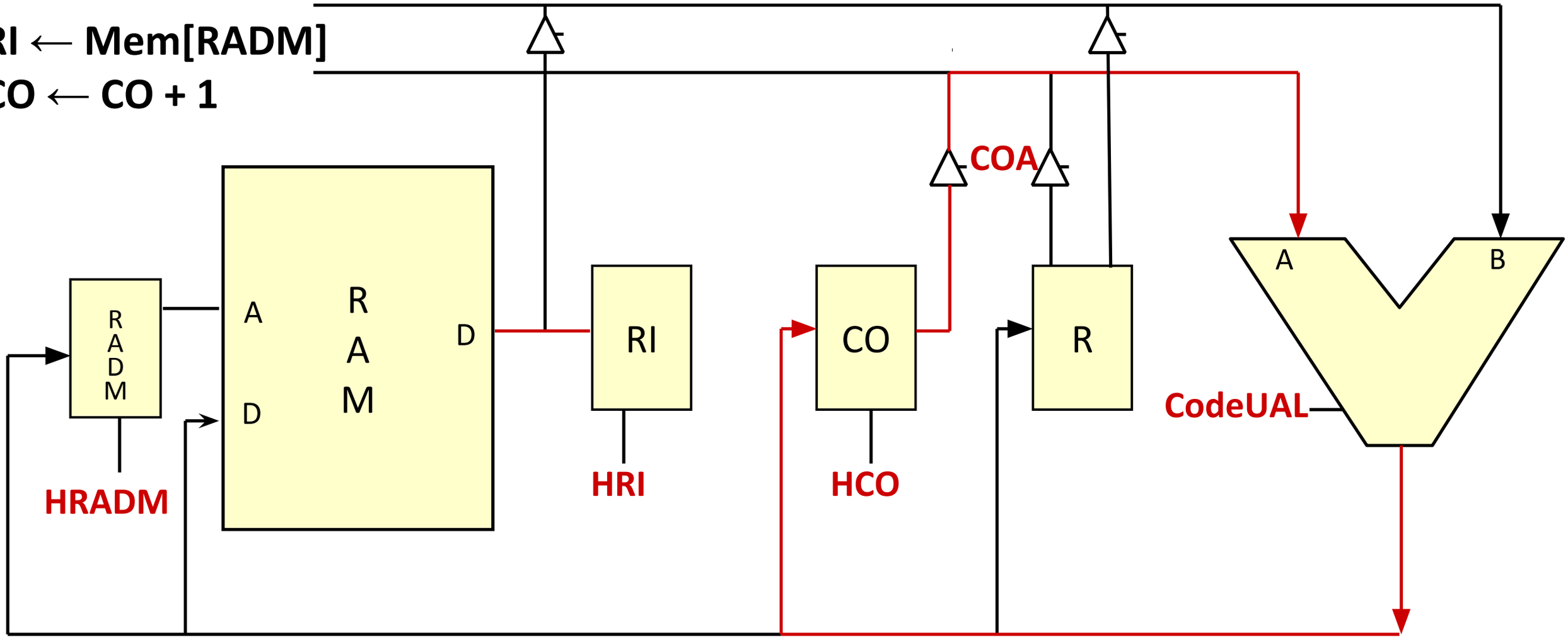
- Dérivation d'un ensemble d'équations logiques
  - Une équation pour chaque signal du chemin de données
  - Une équation communique quand un signal doit être activé
  - Une équation est fonction des entrées du séquenceur et des signaux T, P

Tableau des temps – Fetch



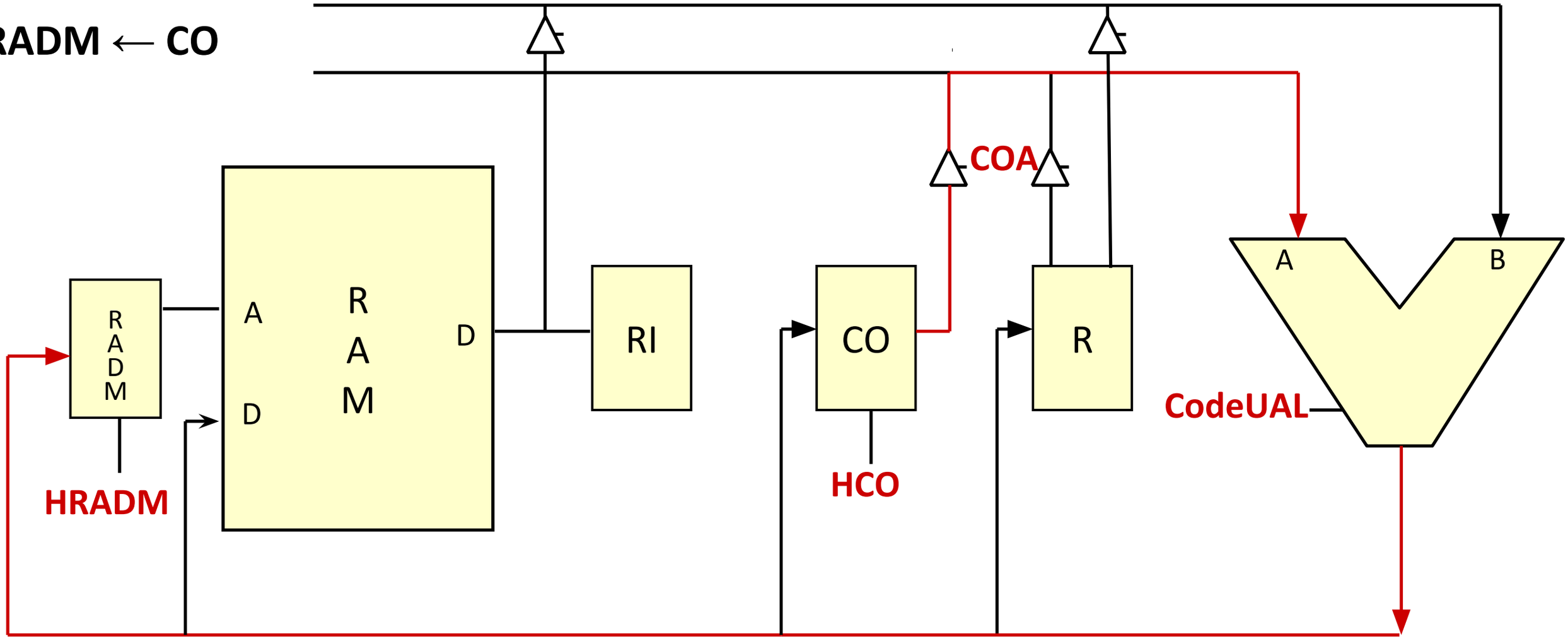
T1	P1	T2	P2
COA = 1 CodeUAL= A	HRADM		

Tableau des temps – Fetch



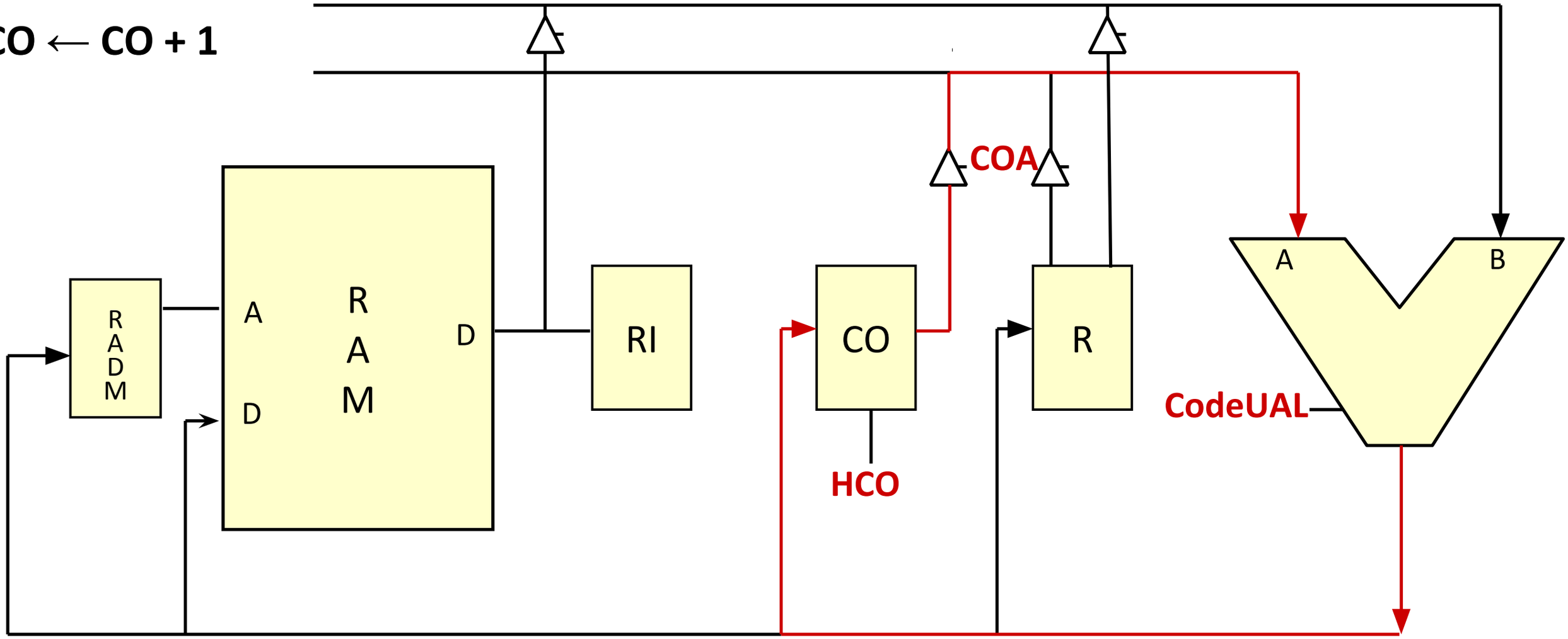
T1	P1	T2	P2
COA = 1 CodeUAL= A	HRADM	COA = 1 CodeUAL = A + 1	HRI; HCO

Tableau des temps – Execute (mov rx, v)



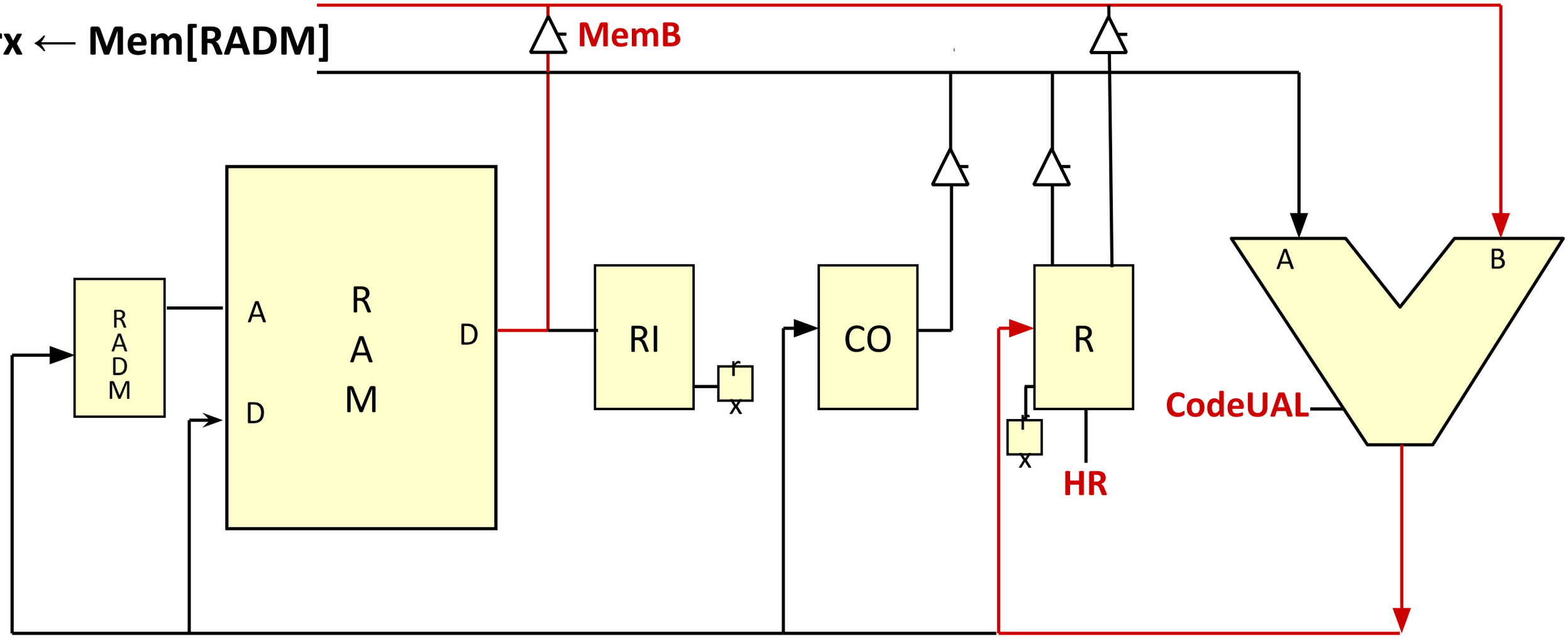
T3	P3	T4	P4	T5	P5
COA=1 CodeUAL = A	HRADM				

Tableau des temps – Execute (mov rx, v)



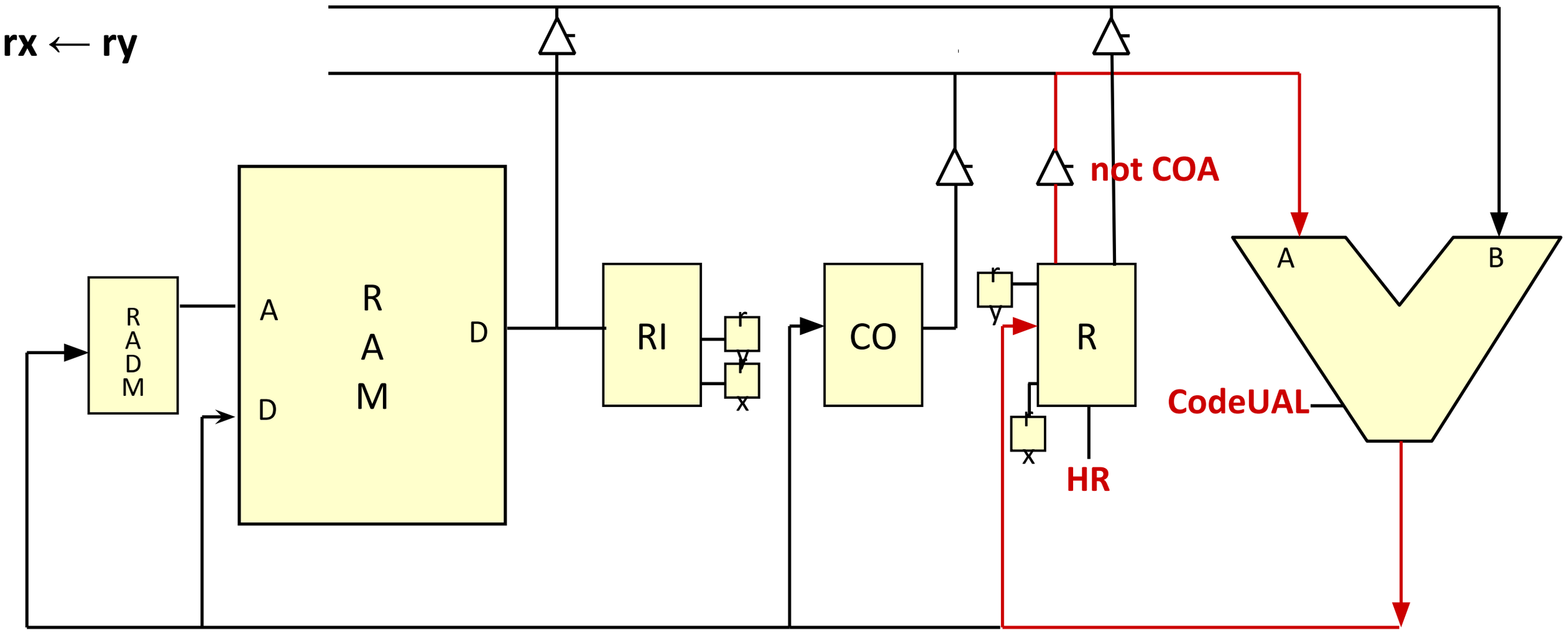
T3	P3	T4	P4	T5	P5
COA=1 CodeUAL = A	HRADM	COA = 1 CodeUAL = A + 1	HCO		

Tableau des temps – Execute (mov rx, v)



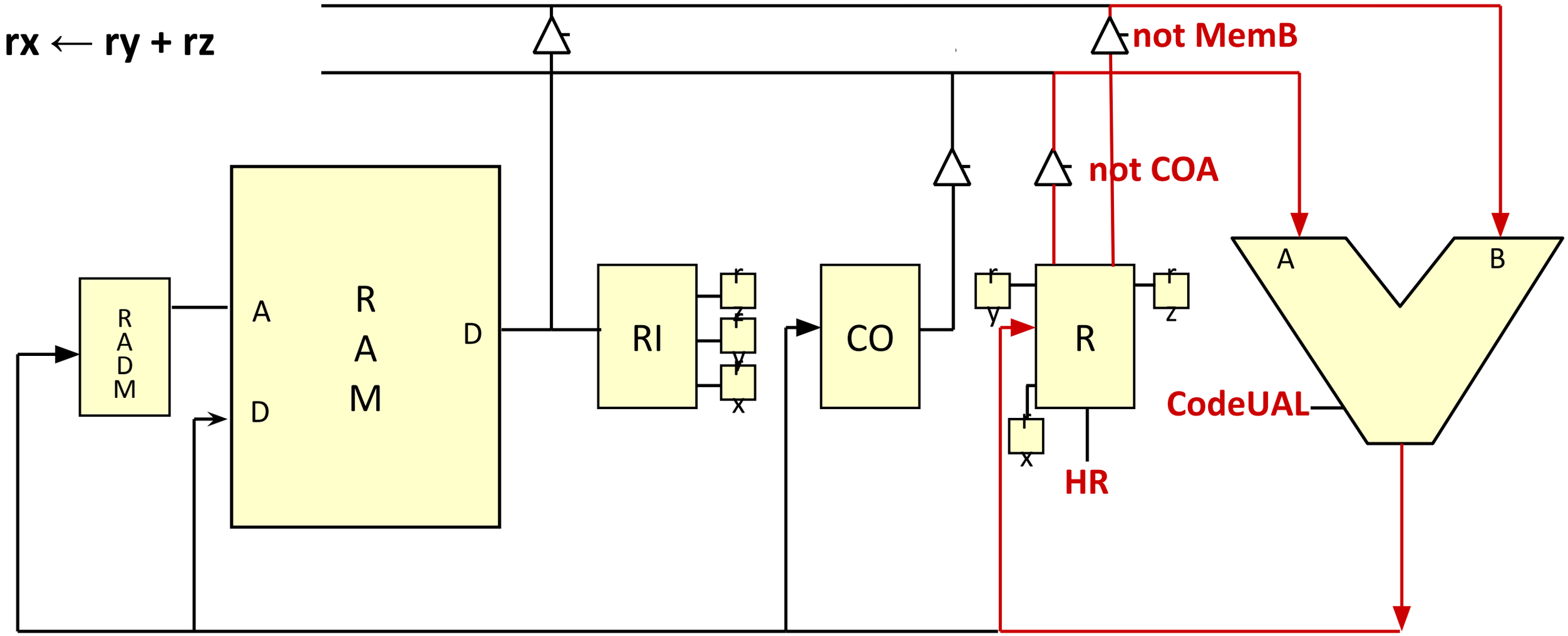
T3	P3	T4	P4	T5	P5
COA=1 CodeUAL = A	HRADM	COA = 1 CodeUAL = A + 1	HCO	MemB = 1 CodeUAL = B	HR

Tableau des temps – Execute (mov rx, ry)



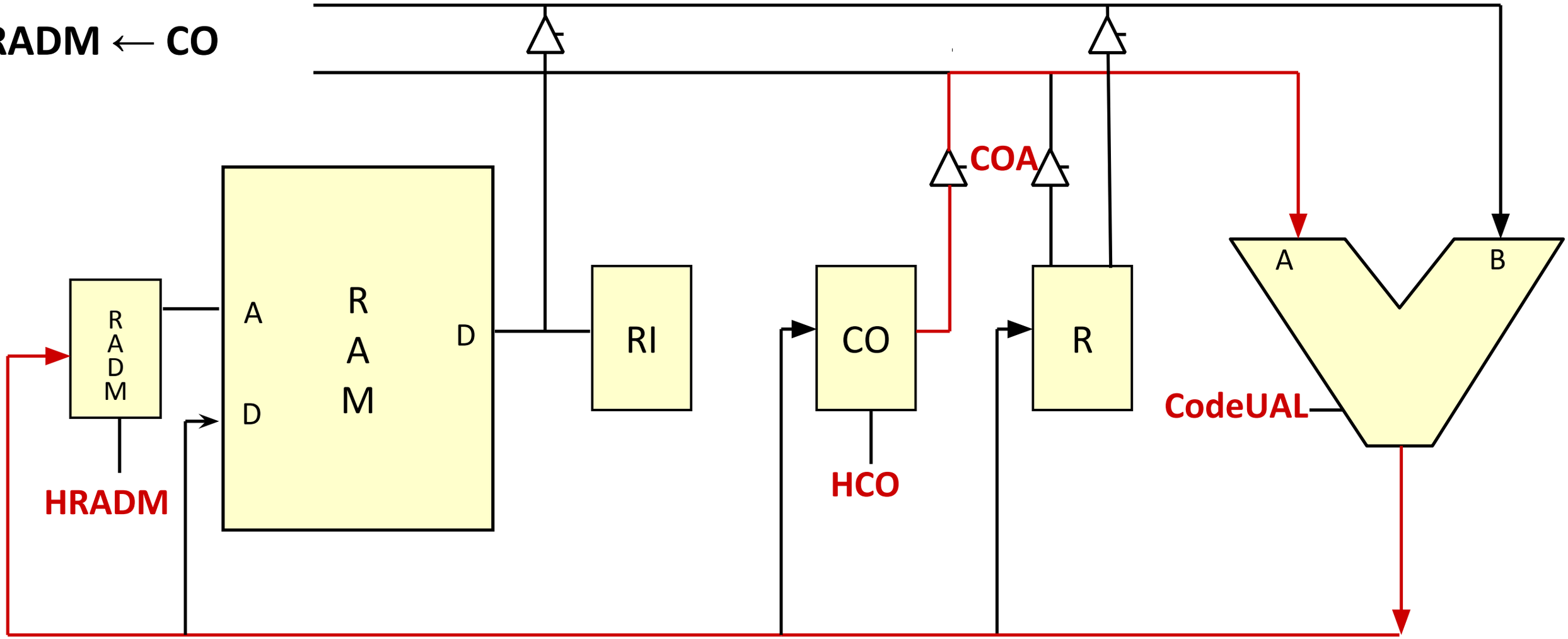
T3	P3	T4	P4	T5	P5
				CUAL = A	HR

Tableau des temps – Execute (add rx, ry, rz)



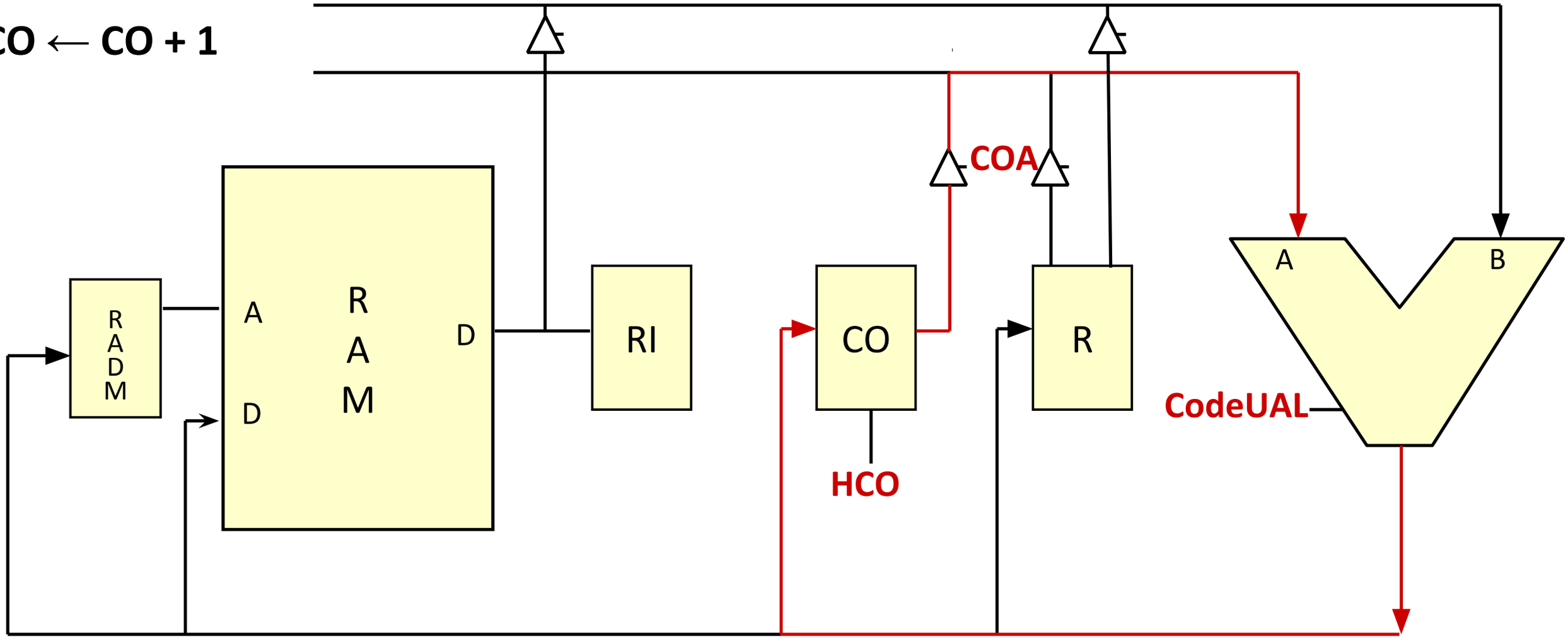
T3	P3	T4	P4	T5	P5
				CodeUAL = A + B	HR

Tableau des temps – Execute (beq a)



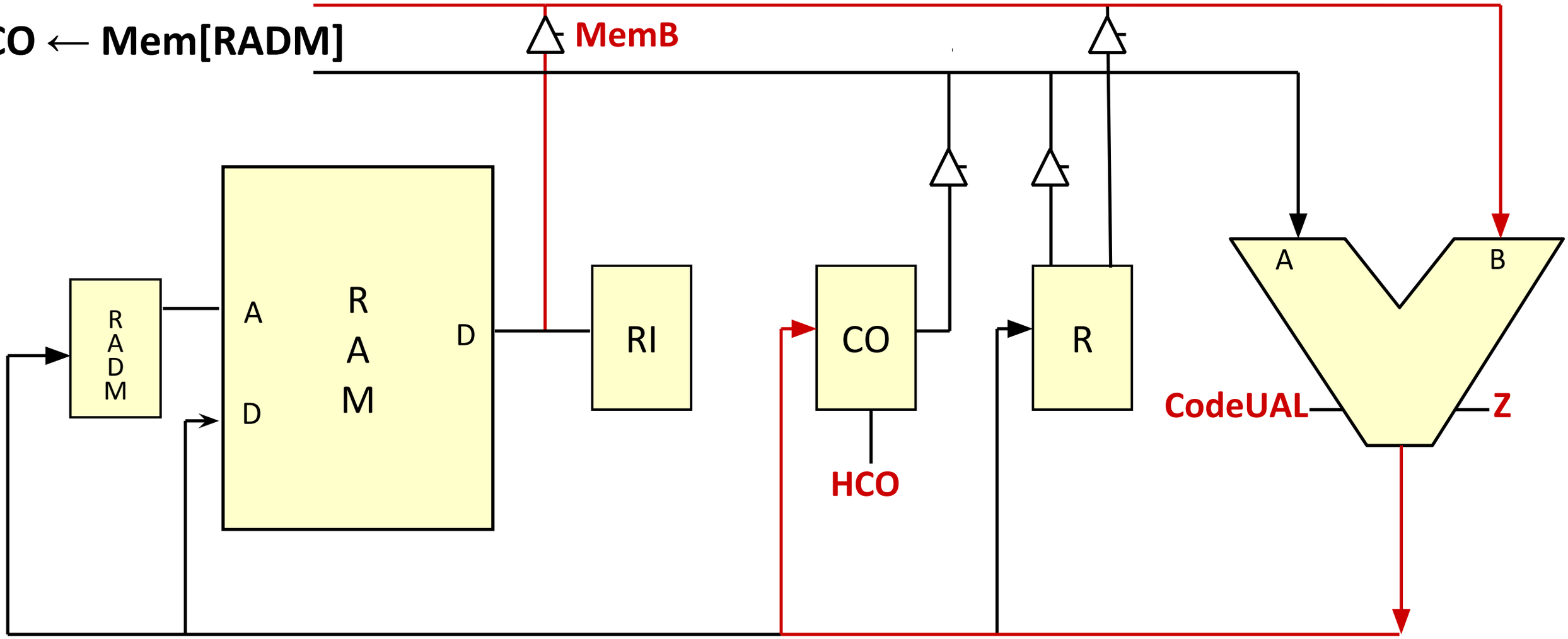
T3	P3	T4	P4	T5	P5
COA=1 CodeUAL = A	HRADM				

Tableau des temps – Execute (beq a)



T3	P3	T4	P4	T5	P5
COA=1 CodeUAL = A	HRADM	COA = 1 CodeUAL = A + 1	HCO		

Tableau des temps – Execute (beq a)



T3	P3	T4	P4	T5	P5
COA=1 CodeUAL = A	HRADM	COA = 1 CodeUAL = A + 1	HCO	MemB = 1 CodeUAL = B	HCO si Z

Équations logiques

Inst.	T1	P1	T2	P2	T3	P3	T4	P4	T5	P5
mov rx, #val	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HRI HCO	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HCO	MemB=1 CodeUAL=B	HR
mov rx, ry	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HRI HCO					CodeUAL=A	HR
add rx, ry, rz	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HRI HCO					CodeUAL=A + B	HR
beq address	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HRI HCO	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HCO	MemB=1 CodeUAL=B	HCO si Z

Équations logiques

Inst.	T1	P1	T2	P2	T3	P3	T4	P4	T5	P5
mov rx, #val	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HRI HCO	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HCO	MemB=1 CodeUAL=B	HR
mov rx, ry	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HRI HCO					CodeUAL=A	HR
add rx, ry, rz	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HRI HCO					CodeUAL=A + B	HR
beq address	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HRI HCO	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HCO	MemB=1 CodeUAL=B	HCO si Z

- COA = T1

Équations logiques

Inst.	T1	P1	T2	P2	T3	P3	T4	P4	T5	P5
mov rx, #val	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HRI HCO	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HCO	MemB=1 CodeUAL=B	HR
mov rx, ry	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HRI HCO					CodeUAL=A	HR
add rx, ry, rz	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HRI HCO					CodeUAL=A + B	HR
beq address	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HRI HCO	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HCO	MemB=1 CodeUAL=B	HCO si Z

- COA = T1 + T2

Équations logiques

Inst.	T1	P1	T2	P2	T3	P3	T4	P4	T5	P5
mov rx, #val	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HRI HCO	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HCO	MemB=1 CodeUAL=B	HR
mov rx, ry	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HRI HCO					CodeUAL=A	HR
add rx, ry, rz	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HRI HCO					CodeUAL=A + B	HR
beq address	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HRI HCO	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HCO	MemB=1 CodeUAL=B	HCO si Z

- COA = T1 + T2 + T3 * IMM + T4*IMM

Équations logiques

Inst.	T1	P1	T2	P2	T3	P3	T4	P4	T5	P5
mov rx, #val	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HRI HCO	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HCO	MemB=1 CodeUAL=B	HR
mov rx, ry	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HRI HCO					CodeUAL=A	HR
add rx, ry, rz	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HRI HCO					CodeUAL=A + B	HR
beq address	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HRI HCO	COA=1 CodeUAL=A	HRADM	COA=1 CodeUAL=A+1	HCO	MemB=1 CodeUAL=B	HCO si Z

- COA = T1 + T2 + T3 * IMM + T4*IMM
- HRADM = P1 + P3*IMM
- HR = P5 * !BEQ ou HR = P5 * (MOV + ADD)
- HCO = P2 + P4 * IMM + P5 * BEQ * Z

CentraleSupélec

université PARIS-SACLAY

LABORATOIRE DE RECHERCHE EN INFORMATIQUE

CHAPITRE IV

Langages de programmation

Architecture des ordinateurs

Langage machine : quels problèmes?

```
2900 0000 2A00 0001 3128 3A40 0001 5840
000B 7800 0004 8800 000B
```

- Programme utilisant le langage machine du processeur *MiniArm.*
- Difficile à lire : chaque instruction est une suite de nombres binaires (**bits**).
  - Qu'est-ce que ce programme fait?

Interagir directement avec la machine est difficile : fossé sémantique.

Langage d'assemblage

```
@begin    mov r1,#0x0000
          mov r2,#0x0001
@loop     add r1,r1,r2
          add r2,r2,#0x0001
          cmp r2,#0x000B
          blt loop
@end      b  end
```

- **Langage d'assemblage :** représentation symbolique du langage machine
  - Plus facile à apprendre, il ne faut pas manipuler du code binaire/hexadécimal.
- **Assembleur** : programme traduisant langage d'assemblage en langage machine
  - L'assembleur est écrit en langage machine.
- Programmes en langage d'assemblage : complexes.
  - Il s'agit encore de langage machine.
  - Il faut bien connaître les détails techniques du processeur.

Langage de haut niveau

```
total = 0;
for i in range(1, 11):
      total += i
print total
```

- **Langage de haut niveau :** expression d'instructions en utilisant des mots et des symboles mathématiques.
  - Facile à apprendre.
  - Il ne nécessite pas une connaissance du processeur de la machine.

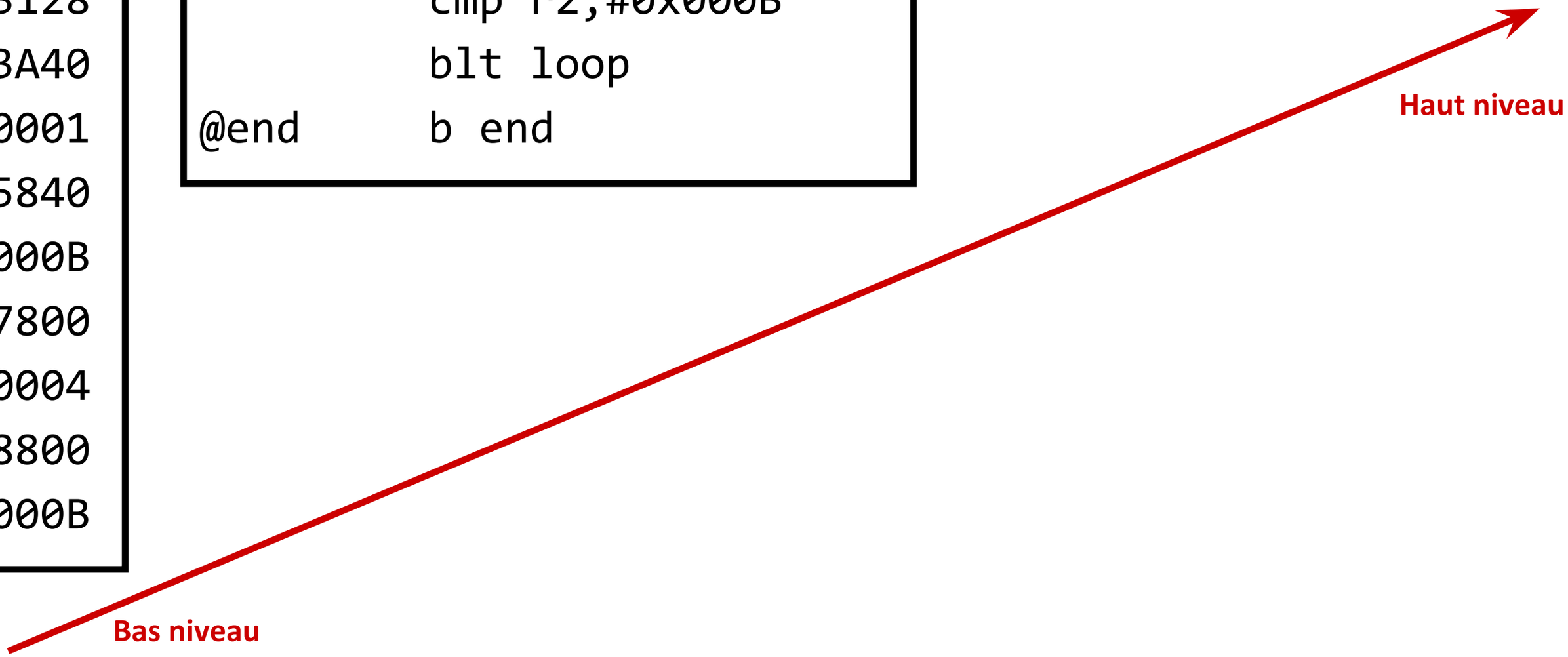
- Langages de haut niveau : Java, C, C++, Pascal, Python …

Pouvez-vous voir les bénéfices?

```
2900
0000
2A00
0001
3128
3A40
0001
5840
000B
7800
0004
8800
000B
```

```
@begin    mov r1,#0x0000
          mov r2,#0x0001
@loop     add r1,r1,r2
          add r2,r2,#0x0001
          cmp r2,#0x000B
          blt loop
@end      b end
```

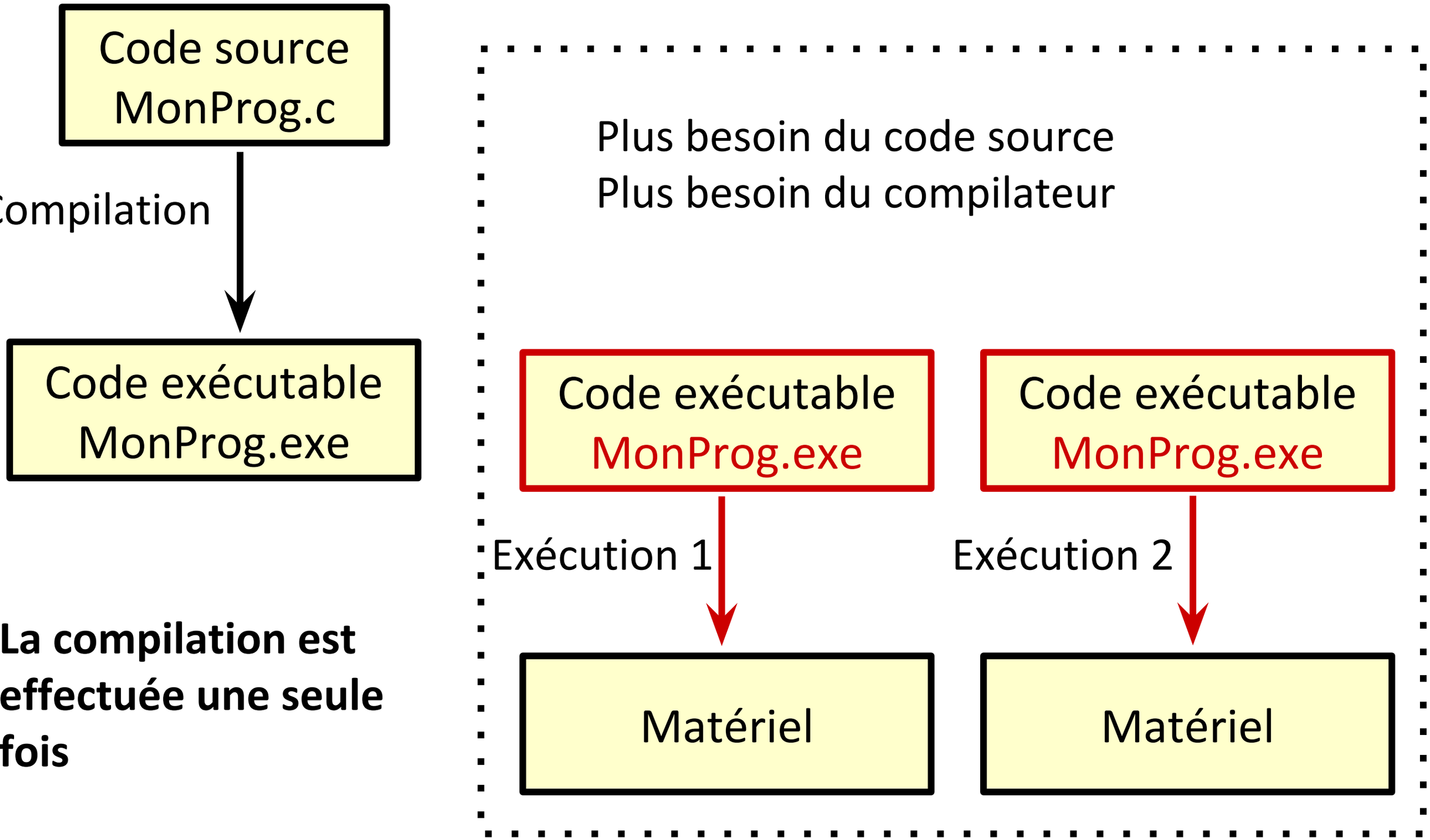
```
total = 0;
for i in range(1, 11):
      total += i
print total
```



Fossé sémantique

- Le processeur « parle » binaire.
- Le programmateur « parle » Python.
- Il faut un intermédiaire pour faire communiquer le programmateur et le processeur.
- Deux options :
  - **compilation**.
  - **interprétation**.

Langages compilés



La compilation est effectuée une seule fois

Langages compilés

Source
MonProg.c

```
int difference (int x, int y)
{
      int r;
      r = x-y;
      return r;
}
```

Exécutable
MonProg.exe

mov edx, [ebp+0xc]	**0110110011001100**
mov eax, [ebp+0x8]	**0110011011001000**
sub eax, edx	**1100101111010110**
mov [ebp-0x4], eax	**1010011011001110**
mov eax, [ebp-0x4]	**0110011011011100**

Qui compile le compilateur?

- Un compilateur est un programme
  - préférable d'utiliser un langage de haut niveau pour l'écrire.
  - par exemple, on peut écrire un compilateur C en Pascal.
  - mais un compilateur Pascal doit déjà exister.

- Nous avons besoin d'un compilateur pour traduire du code C dans le code machine d'un nouveau processeur.

- **Premier choix** : écrire le compilateur en langage machine (pas sympa!)
  - Les premiers compilateur étaient écrits en langage machine (Fortran, 1954 - 1957)

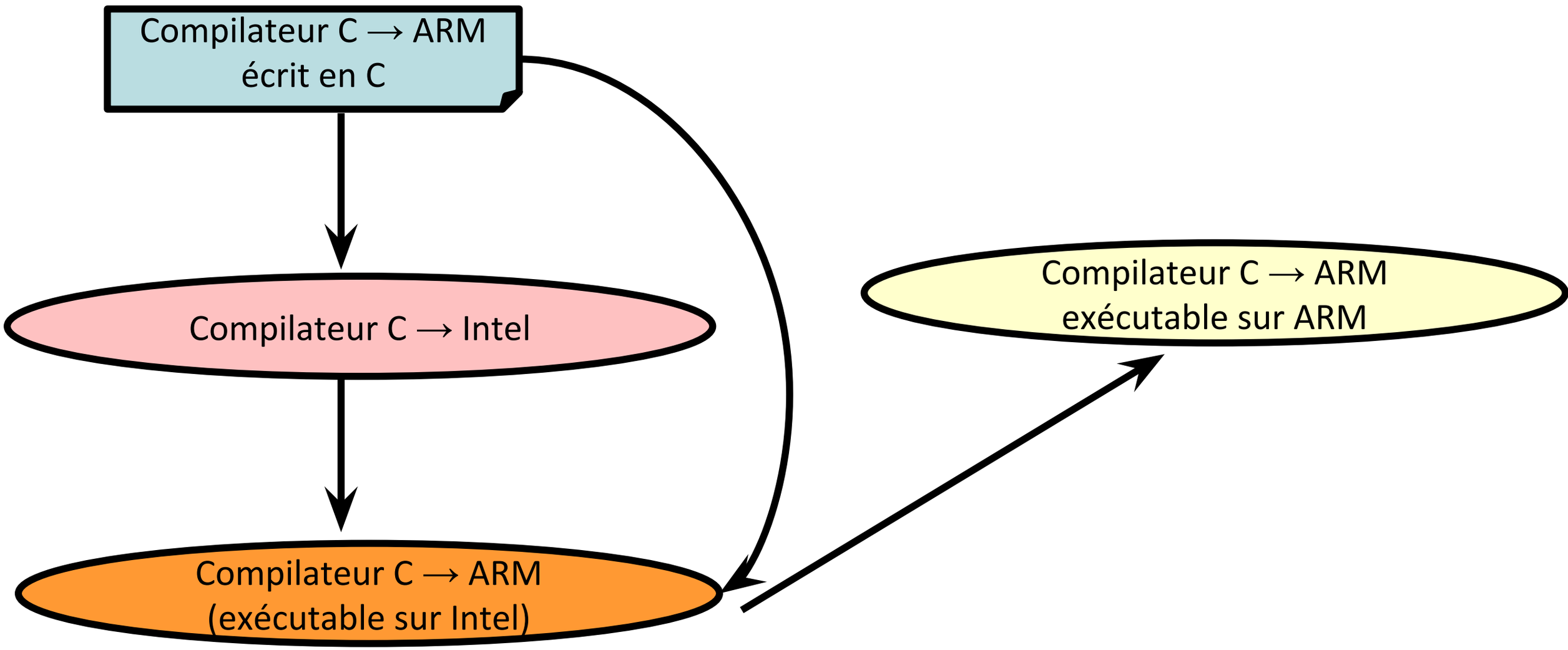
- **Deuxième choix** : **bootstrapping**
  - On crée un compilateur en C qui compile du code C !
    - paradoxe?

Bootstrapping

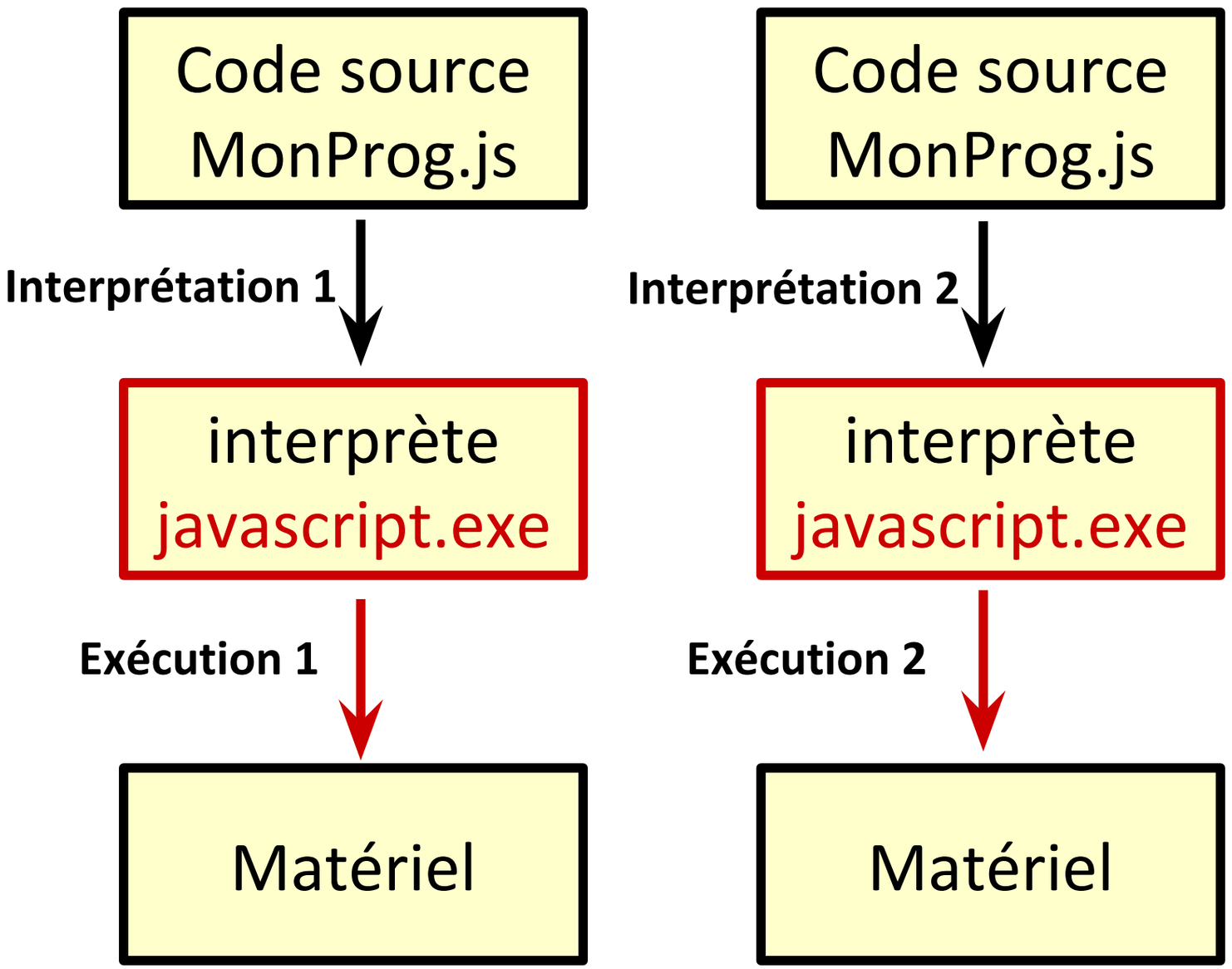
Machine A
Intel CPU

But : compilateur C → ARM

Machine B
ARM CPU



Langages interprétés



Toujours besoin du code source

Toujours besoin de l'interprète

Langages interprétés

Code source de l'interprète javascript.c

```
char word[100];
char prog[100000];
// charger fichier prog
...
// boucle d'exécution
do{

readNextWord(word,prog);
 if (!strcmp(word,"if"))
  ... tester ce qui vient
 if (!strcmp(word,"for"))
  ... répéter ce qui vient
 if (!strcmp(word,"="))
  ... affecter
 if (!strcmp(word,"=="))
  ... calcul booléen égalité
 ...
} while (progPasFini);
```

compilation

```
function factorial (n){
  x=1;
  for (i=2;i<=n;i++) x=x*i;
return x; }
```

Interprétation

```
011010001010011
011010001010011
011010001010011
011010001010011
011010001010011
011010001010011
011010001010011
```

javascript.exe

Exécution

Matériel

Langages interprétés

- **Avantages**
  - **souplesse**, on peut modifier le code sans avoir à régénérer un fichier exécutable.
  - **sécurité**, l'interprète vérifie si un programme fait des opérations dangereuses.
  - **portabilité**, on peut exécuter du code sur un ordinateur quelconque où l'interpréteur est installé.
  - **accessibilité** du code source.
- **Inconvénients**
  - plus lent : il faut retraduire les instructions à chaque fois
  - le code source est toujours accessible
- **Approches hybrides :** langages compilés/interprétés (par ex., Java, Python)
  - Code source : MyClass.java
  - Un compilateur convertit le fichier source MyClass.java en fichier MyClass.class
  - MyClass.class est écrit en **bytecode**.
  - Bytecode : jeu d'instructions indépendant d'un processeur réel, il n'est pas directement exécutable par le processeur
  - Le Bytecode est exécuté par la **Java Virtual Machine**.

Langages de haut niveau : concepts

- Les langages de programmation de haut niveau proposent des concepts abstraits qui ne sont pas présents dans le langage machine.
- Types de données : entiers, réels, string….
- Structures de contrôle : *if....then....else, while, for*, fonctions.

- Représentation des données au niveau du processeur :
  - Mots de 8, 16, 32 ou 64 bits.
  - Un mot permet d'identifier une instruction, une adresse et/ou une donnée.

- Représentation des données dans un langage de programmation haut niveau :
  - booléens : faux, vrai
  - entiers : complément à deux
  - flottants : norme IEEE 754
  - caractères : Unicode, UTF-8
  - tableaux et structures : mots consécutifs en mémoire.

Structures de contrôle

- Le langages de programmation fournissent un ensemble de structures de contrôle.
  - **conditions** : *if…then…else; switch…case*
  - **boucles** : *while*; for
  - **appels de fonction**.

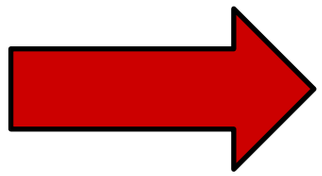
- Ces structures utilisent des opérateurs de comparaison :
  - <=, >=, ==, >, <

- Comment ces structures de contrôle sont-elles traduites par le compilateur en langage machine?

- Notre processeur *MiniArm* dispose d'un ensemble limité d'instructions:
  - cmp, b, beq, blt

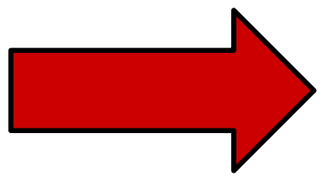
Conditions

```
if r0 == r1 :
    r0 = r1 – r0
else :
    r1 = r0 – r1
```



```
    cmp r0, r1
    beq vrai
    sub r1, r0, r1
    b fin
@vrai    sub r0, r1, r0
@fin  b fin
```

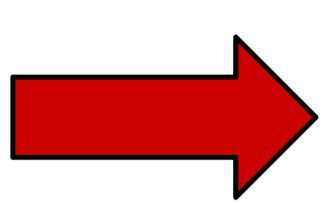
```
if r0 != r1 :
    r0 = r1 – r0
else :
    r1 = r0 – r1
```



```
    cmp r0, r1
    beq faux
    sub r0, r1, r0
    b fin
@faux    sub r1, r0, r1
@fin  b fin
```

Conditions

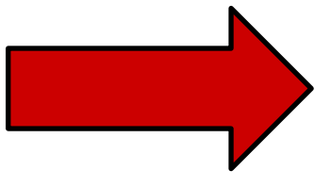
```
if r0 < r1 :
    r0 = r1 – r0
else :
    r1 = r0 – r1
```



```
    cmp r0, r1
    blt vrai
    sub r1, r0, r1
    b fin
@vrai sub r0, r1, r0
@fin   b fin
```

Conditions

```
if r0 <= r1 :
    r0 = r1 – r0
else :
    r1 = r0 – r1
```

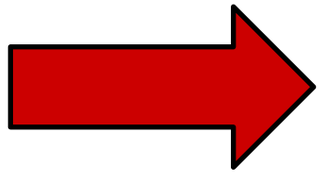


```
if r1 < r0 :
    r1 = r0 – r1
else :
    r0 = r1 – r0
```

```
    cmp r1, r0
    blt vrai
    sub r0, r1, r0
    b fin
@vrai sub r1, r0, r1
@fin   b fin
```

Conditions

```
if r0 > r1 :
    r0 = r1 – r0
else :
    r1 = r0 – r1
```

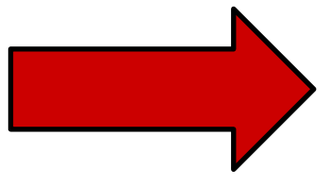


```
if r1 < r0 :
    r0 = r1 – r0
else :
    r1 = r0 – r1
```

```
    cmp r1, r0
    blt vrai
    sub r1, r0, r1
    b fin
@vrai sub r0, r1, r0
@fin   b fin
```

Conditions

```
if r0 >= r1 :
    r0 = r1 – r0
else :
    r1 = r0 – r1
```

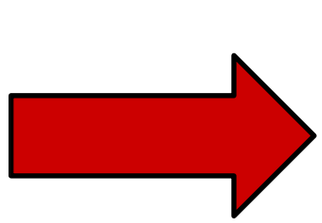


```
if r0 < r1 :
    r1 = r0 – r1
else :
    r0 = r1 – r0
```

```
    cmp r0, r1
    blt vrai
    sub r0, r1, r0
    b fin
@vrai    sub r1, r0, r1
@fin   b fin
```

Boucles

```
r2 = 0
while r0 >= r1 :
    r0 = r1 – r0
    r2 = r2 + 1
```

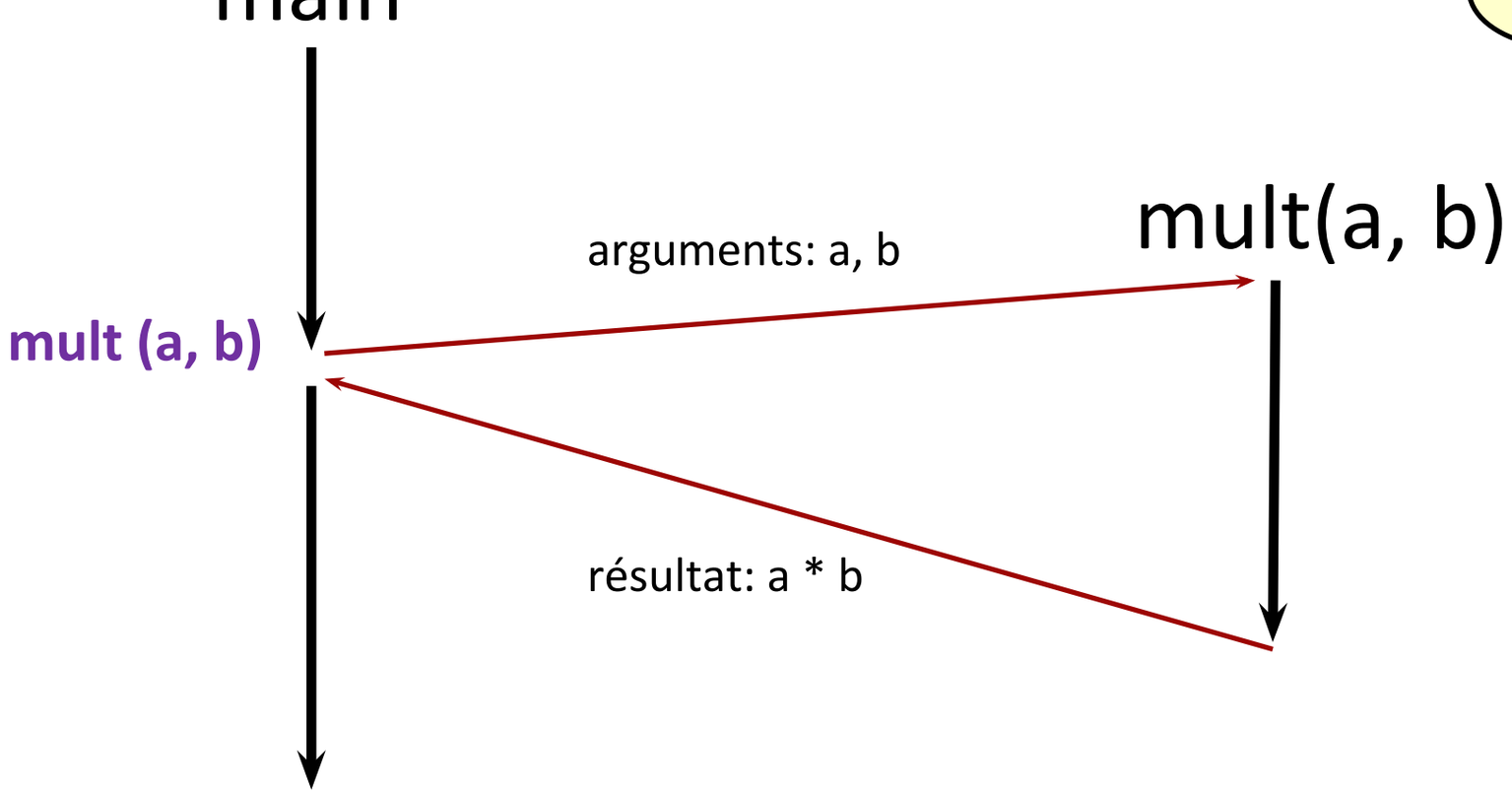


```
        mov r2, #0
@loop   cmp r0, r1
        blt fin
        sub r0, r1, r0
        add r2, r2, #1
        b loop
@fin
```

Appel de fonctions

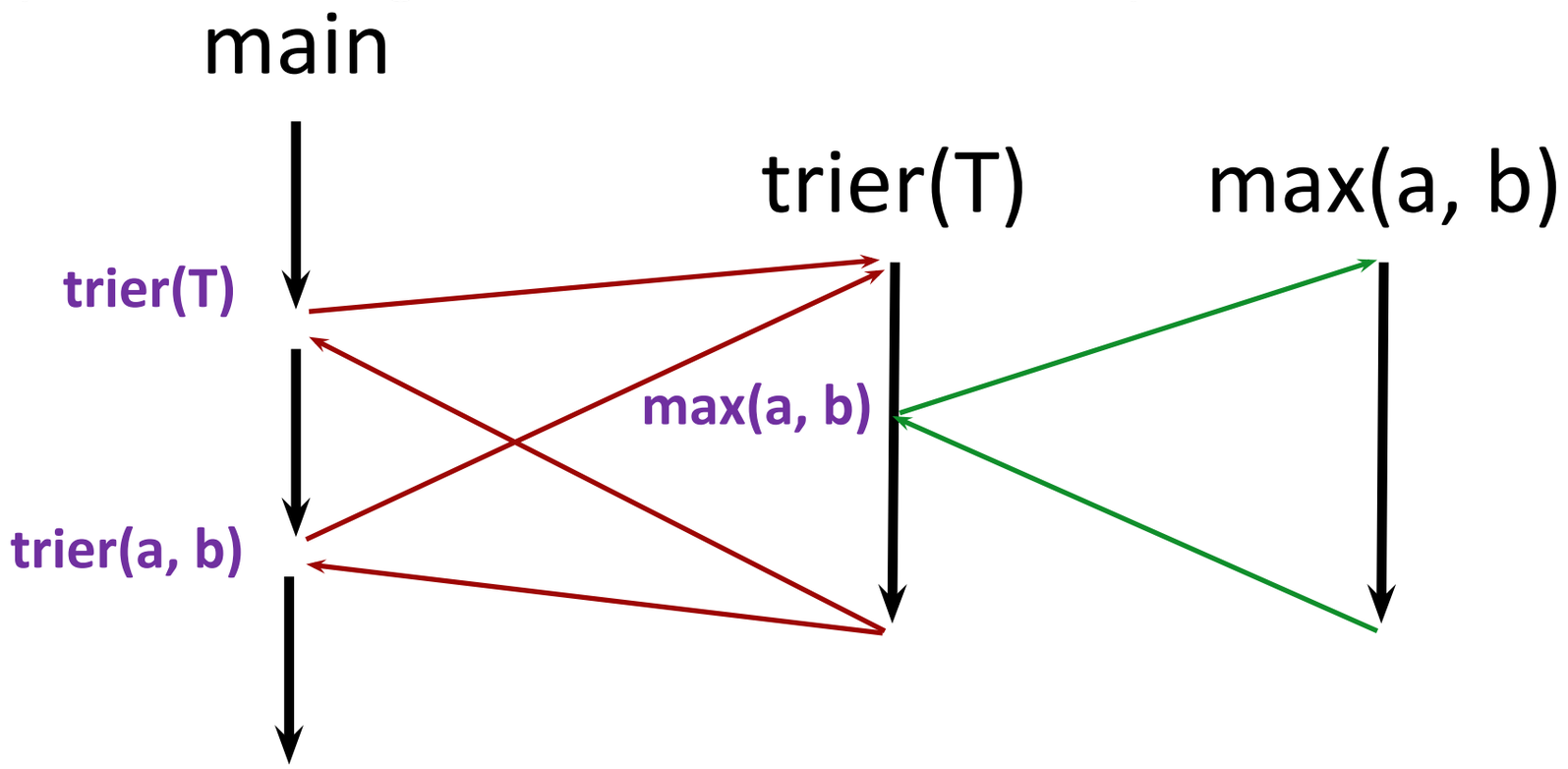
- La possibilité de définir et appeler des fonctions est fondamentale dans tout langage moderne.
- Une fonction est définie une fois seulement.
- Une fonction peut être appelée plusieurs fois
  - Réutilisation du code

Une fonction est identifiée par l'adresse de sa première instruction

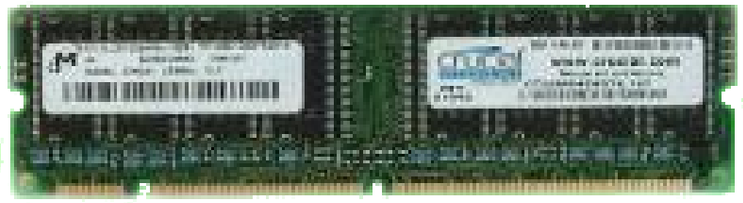


Appel de fonction

- Le programme appelant une fonction doit reprendre son exécution une fois l'exécution de la fonction terminée.
- Besoin de sauvegarder l'adresse de l'instruction qui suit l'appel de fonction.
  - Cette adresse est contenue dans le compteur ordinal.
  - On appelle cette adresse : **adresse de retour.**
- Besoin de passer des arguments à la fonction. Où peut-on les sauvegarder?

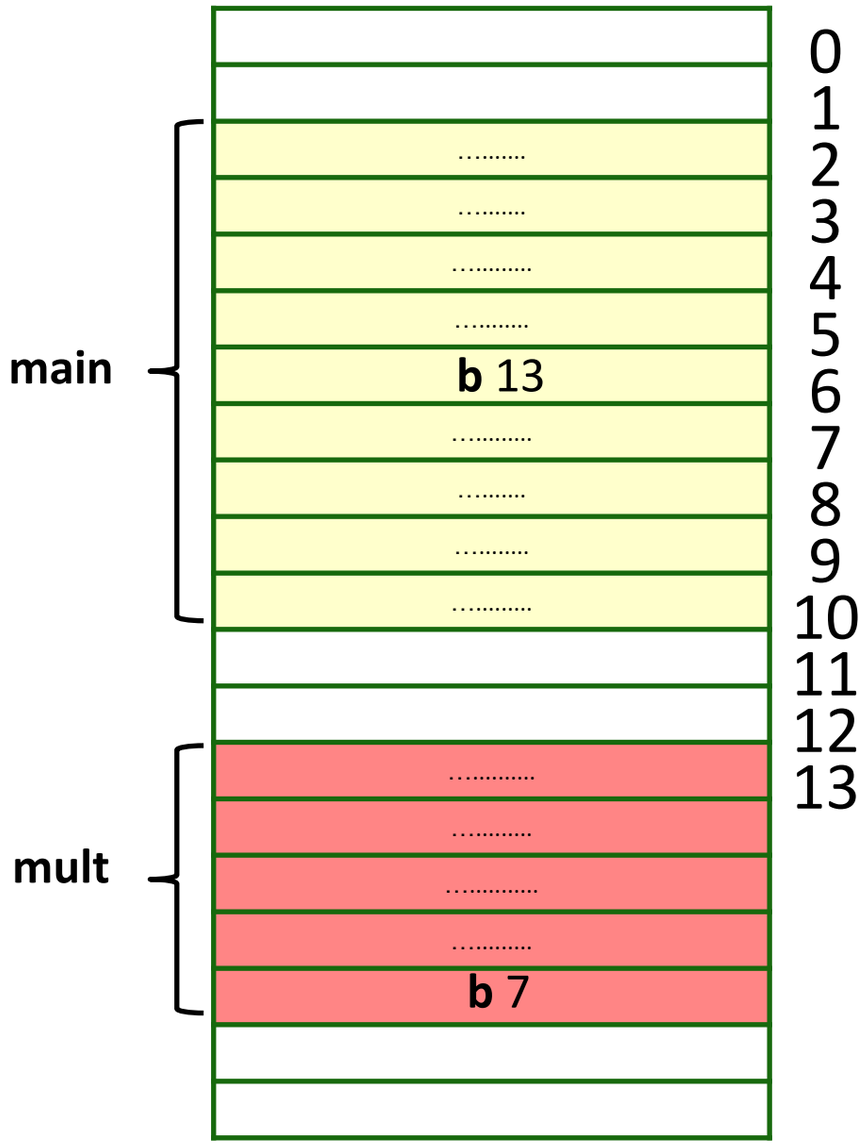


Appel de fonction



Mémoire

- Un appel de fonction s'effectue avec une instruction de branchement.
- Comment la fonction *main* passe-t-elle ses arguments à la fonction *mult*?
- Comment la fonction *mult* passe-t-elle son résultat à la fonction *main*?
- Comment la fonction *mult* sait-elle que l'**adresse de retour** est 7?



Appel de fonction

- La fonction *main* sauvegarde les deux arguments (3 et 4) dans les registres *r0* et *r1* respectivement.
- La fonction *main* sauvegarde l'adresse de retour dans le registre *r7*.
- La fonction *mult* trouve ses arguments dans les registres *r0* et *r1*.
- La fonction *mult* place le résultat dans *r2*.
- A la fin, la fonction *mult* saute à l'adresse de retour contenue dans le registre *r7*.

r0	r1	r2	r3	r4	r5	r6	r7
3	4						ret

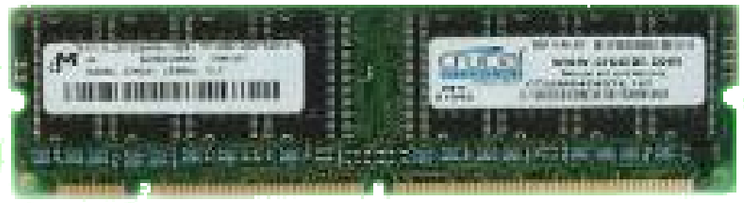
```
@main   mov r0, #3        @mult   mov r2, #0
        mov r1, #4        @loop   cmp r0, #0
        mov r7, next                beq  fin
        b   mult                    add r2, r2, r1
@ret    …..                         sub r0, r0, #1
                          b   loop
                              @fin    mov r0, r2
                               b   r7
```

Problèmes.

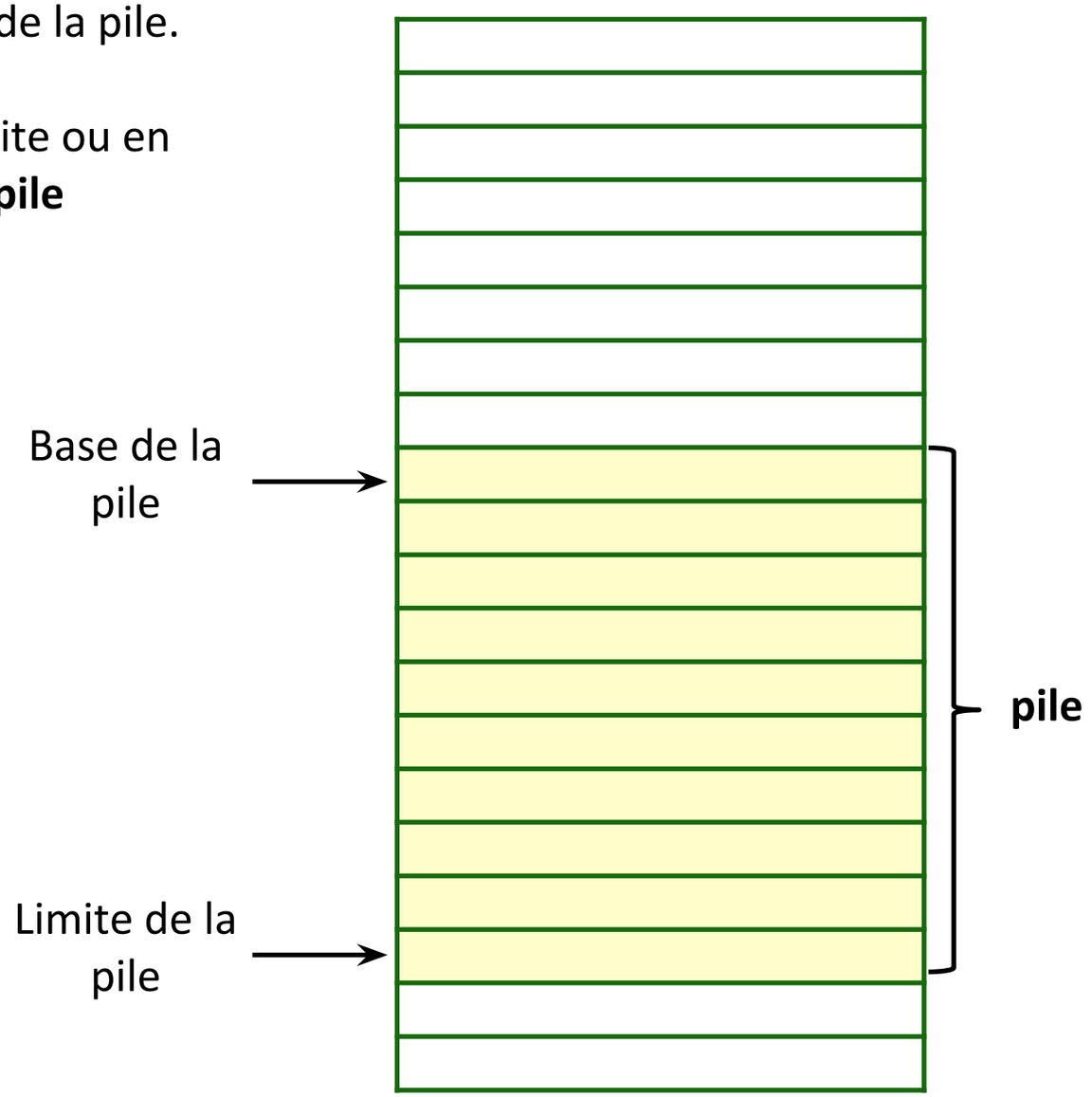
- *mult* écrase la valeur du registre *r2*. Et si ce registre était utilisé par le programme *main*?
- Si *mult* appelle une autre fonction, elle écrase la valeur du registre *r7*.
- Même problème dans le cas de fonctions récursives.

Le concept de pile

- La pile (*stack*) est une zone de mémoire de taille variable.
- La **base de la pile** est la première adresse utilisable de la pile.
- La limite de la pile est la dernière adresse utilisable de la pile.
- Essayer d'ajouter des éléments en dessous de la limite ou en dessus de la base provoque un **débordement de la pile** (*stack overflow*).

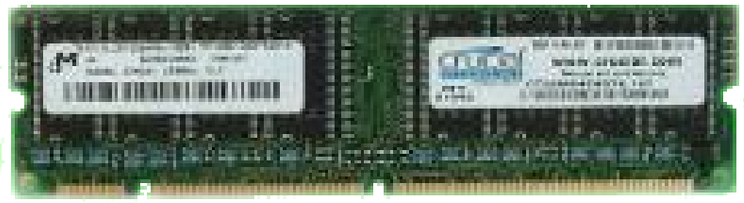


Mémoire

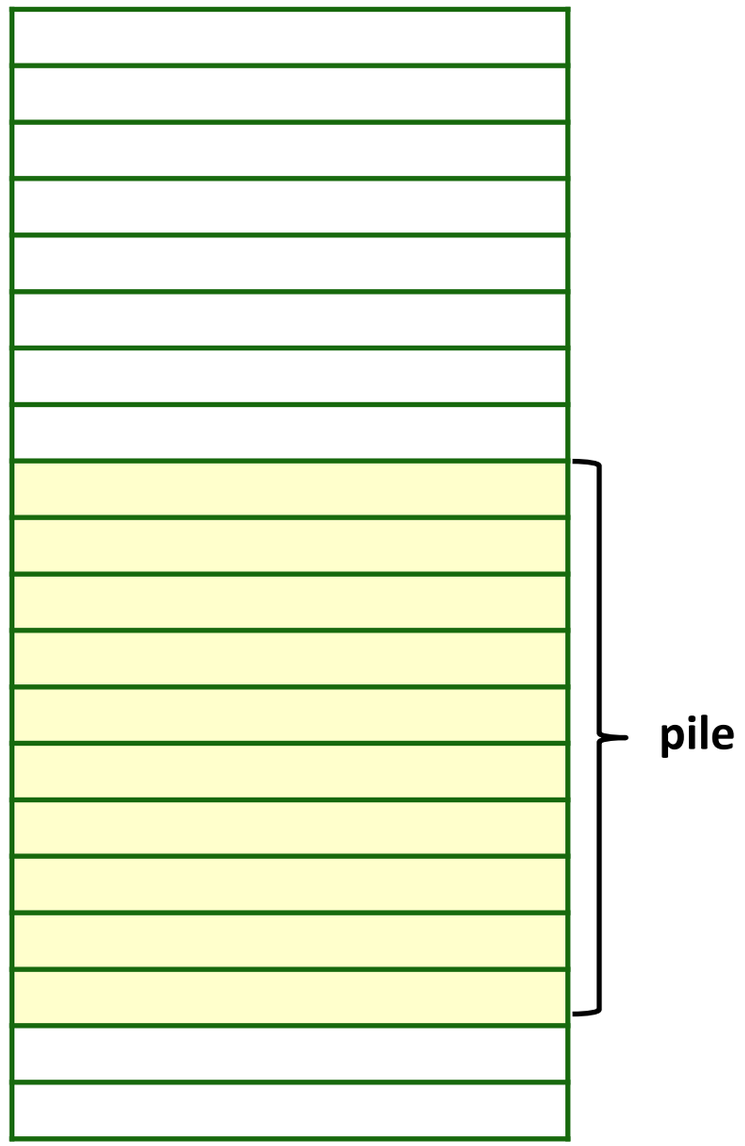


Le concept de pile

- L'opération permettant d'ajouter une donnée à la pile est nommée **push**.
- L'adresse de la dernière donnée ajoutée à la pile est gardée dans un registre du processeur qui s'appelle **stack pointer** (SP).



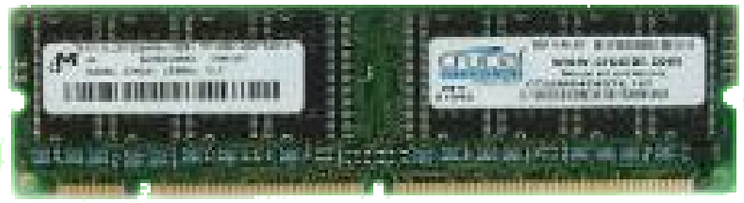
Mémoire



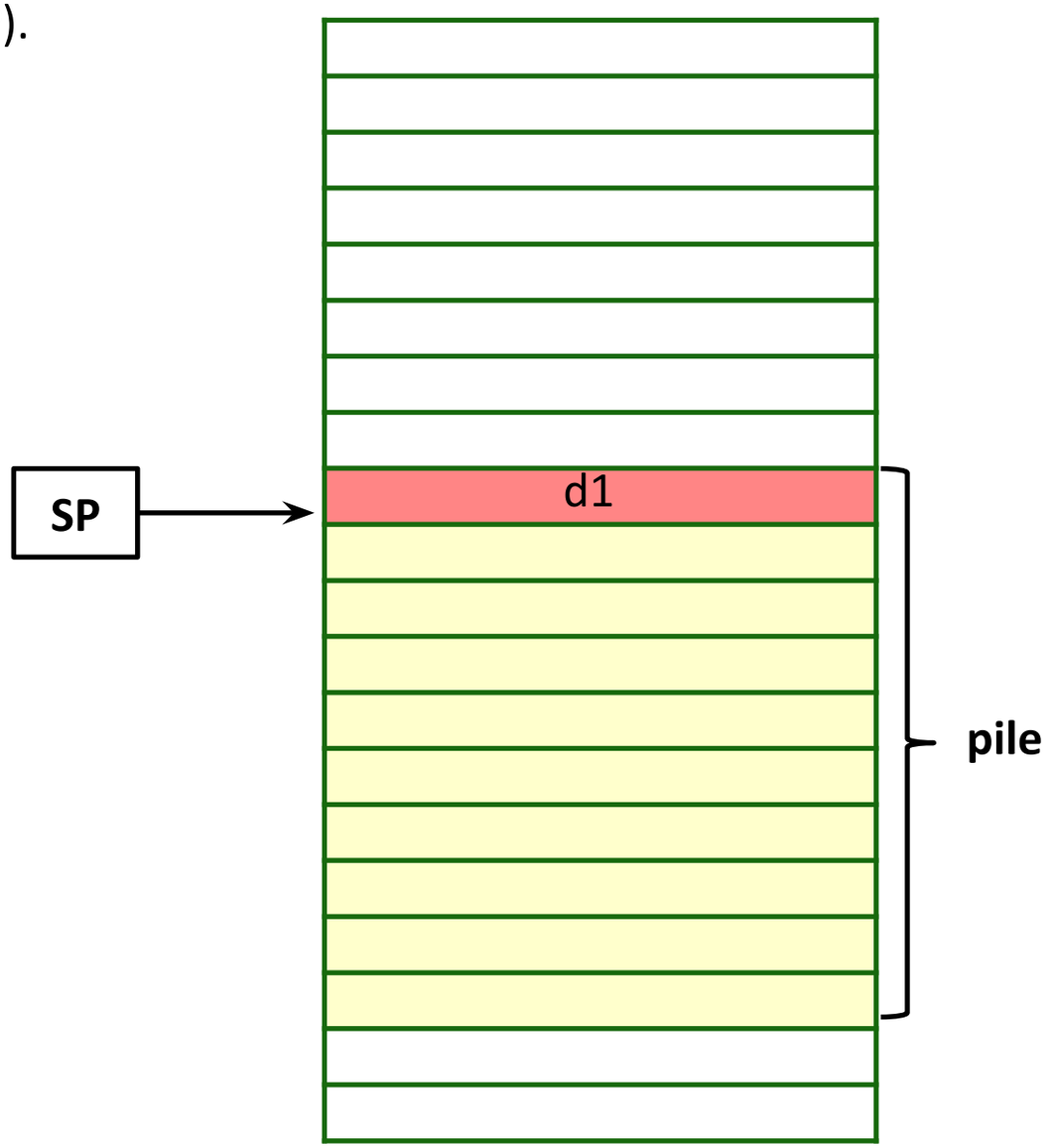
Le concept de pile

- L'opération permettant d'ajouter une donnée à la pile est nommée **push**.
- L'adresse de la dernière donnée ajoutée à la pile est gardée dans un registre du processeur qui s'appelle **stack pointer** (SP).

push d1

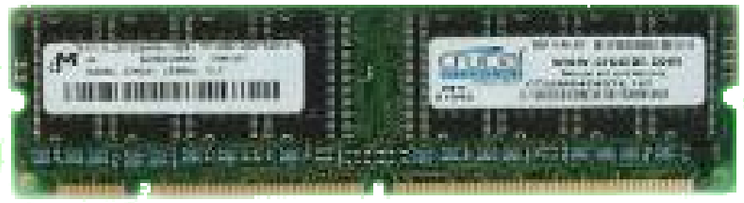


Mémoire



Le concept de pile

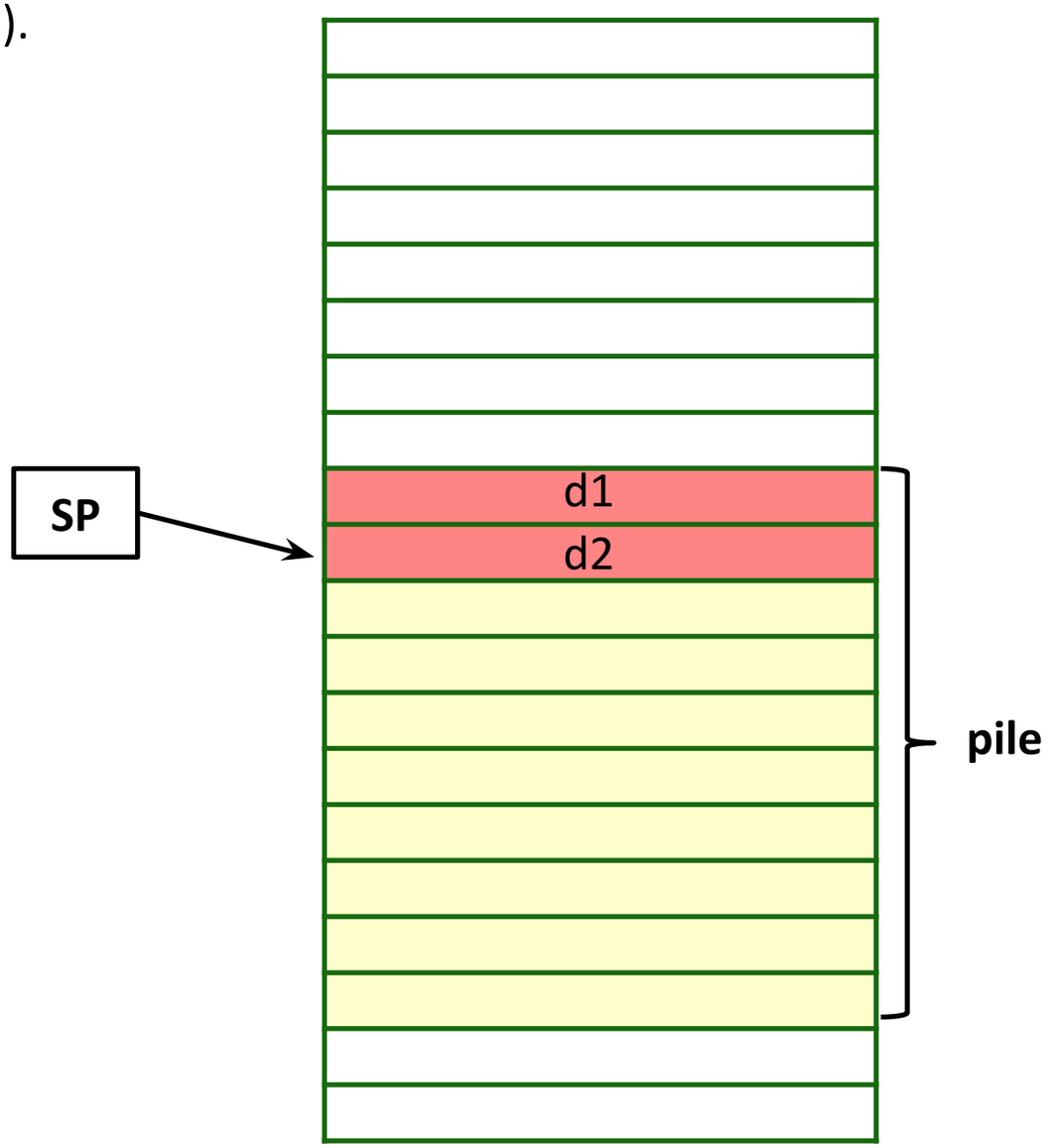
- L'opération permettant d'ajouter une donnée à la pile est nommée **push**.
- L'adresse de la dernière donnée ajoutée à la pile est gardée dans un registre du processeur qui s'appelle **stack pointer** (SP).



Mémoire

push d1

push d2



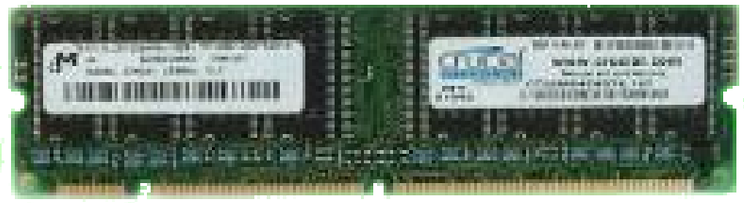
Le concept de pile

- L'opération permettant d'ajouter une donnée à la pile est nommée **push**.
- L'adresse de la dernière donnée ajoutée à la pile est gardée dans un registre du processeur qui s'appelle **stack pointer** (SP).

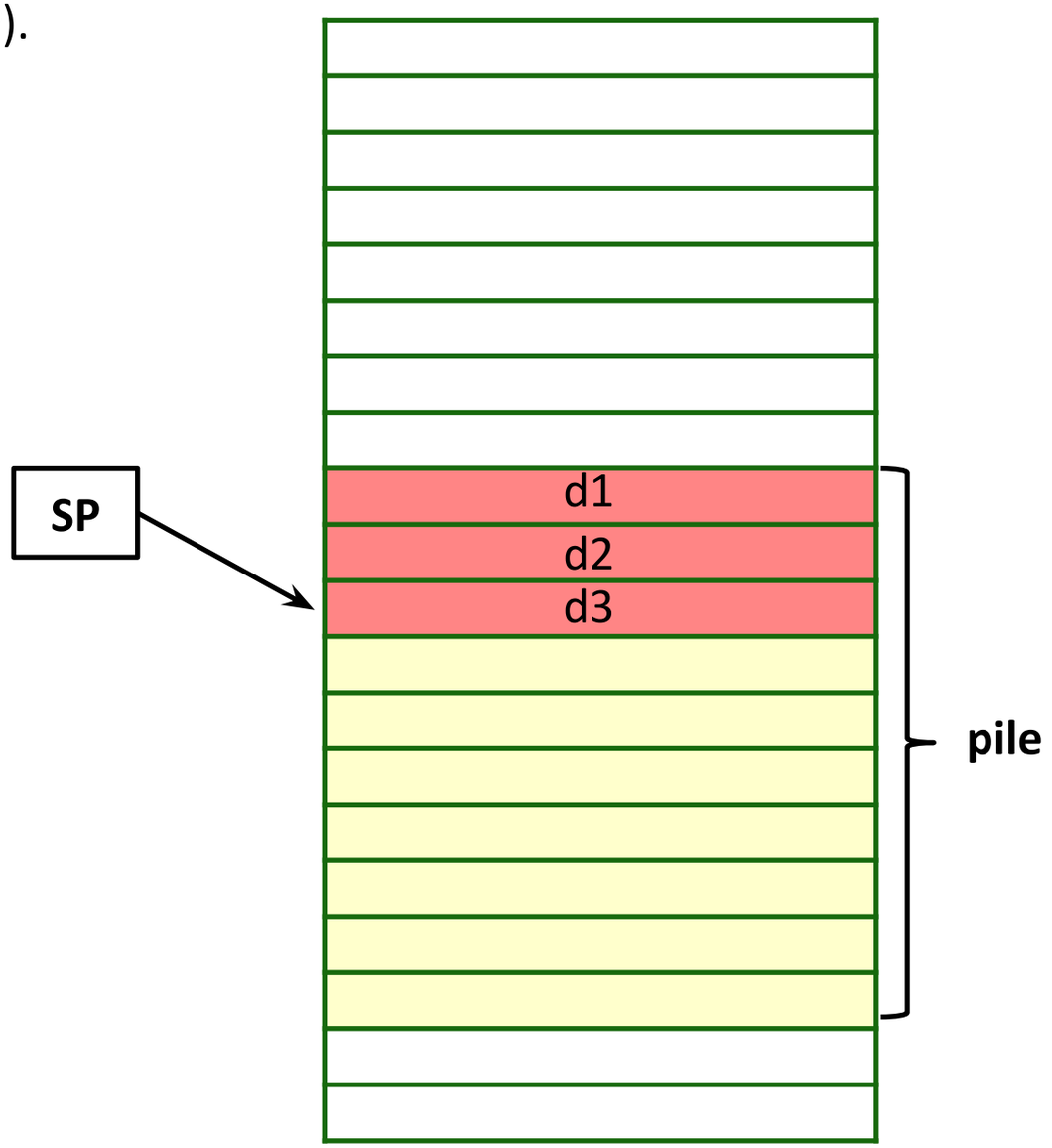
push d1

push d2

push d3



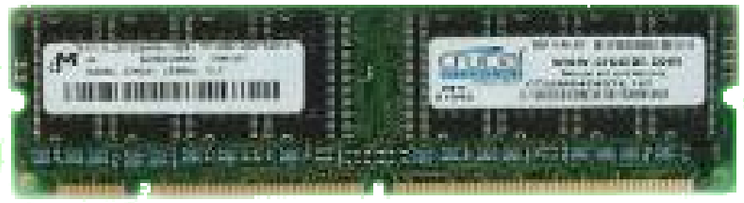
Mémoire



Le concept de pile

- L'opération permettant d'ajouter une donnée à la pile est nommée **push**.

- L'adresse de la dernière donnée ajoutée à la pile est gardée dans un registre du processeur qui s'appelle **stack pointer** (SP).

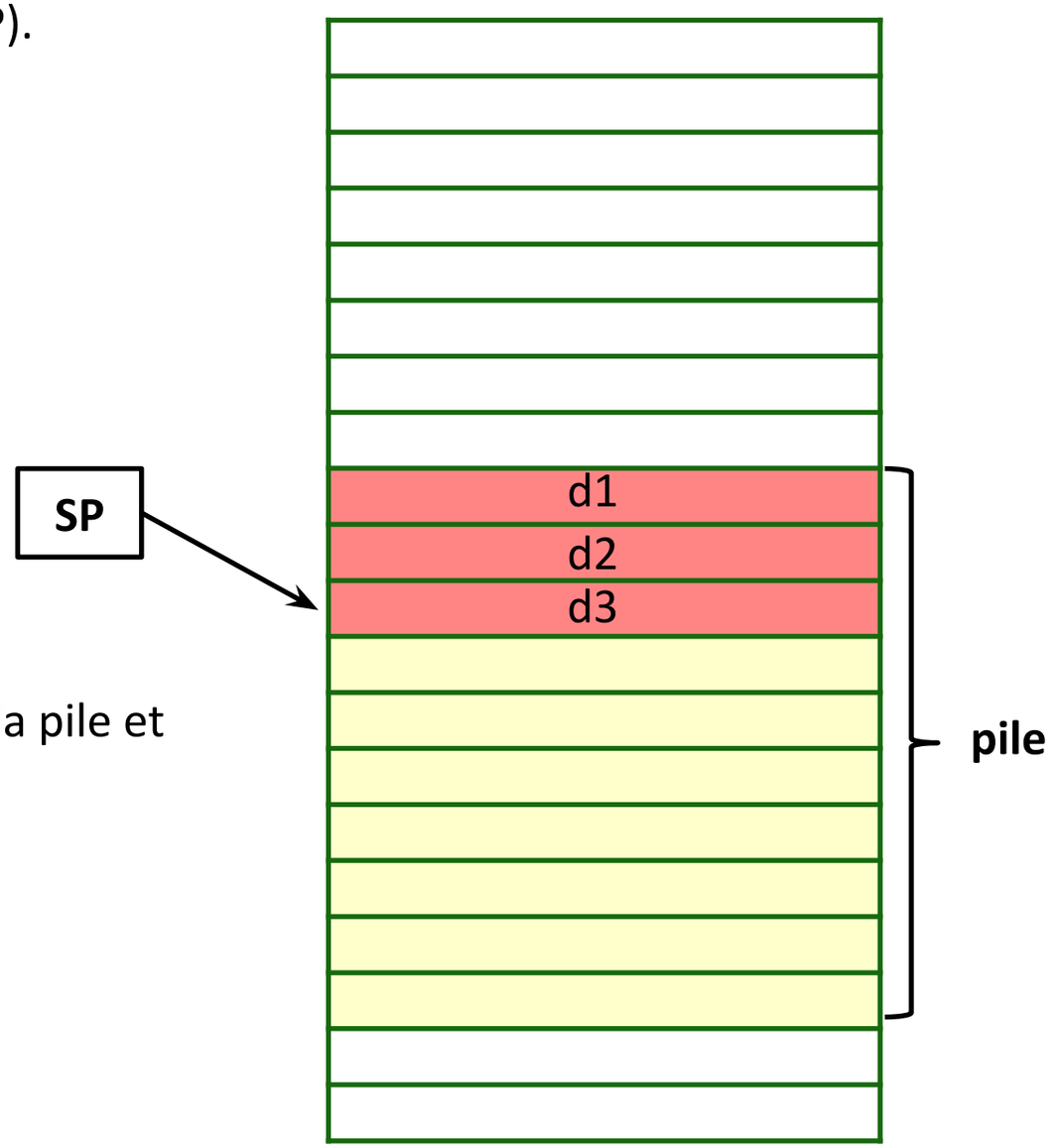


Mémoire

push d1

push d2

push d3

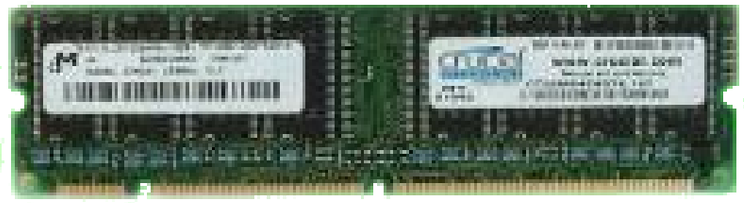


- L'opération pour éliminer la dernière donnée ajoutée à la pile et la copier dans un registre du processeur s'appelle **pop**.

Le concept de pile

- L'opération permettant d'ajouter une donnée à la pile est nommée **push**.

- L'adresse de la dernière donnée ajoutée à la pile est gardée dans un registre du processeur qui s'appelle **stack pointer** (SP).

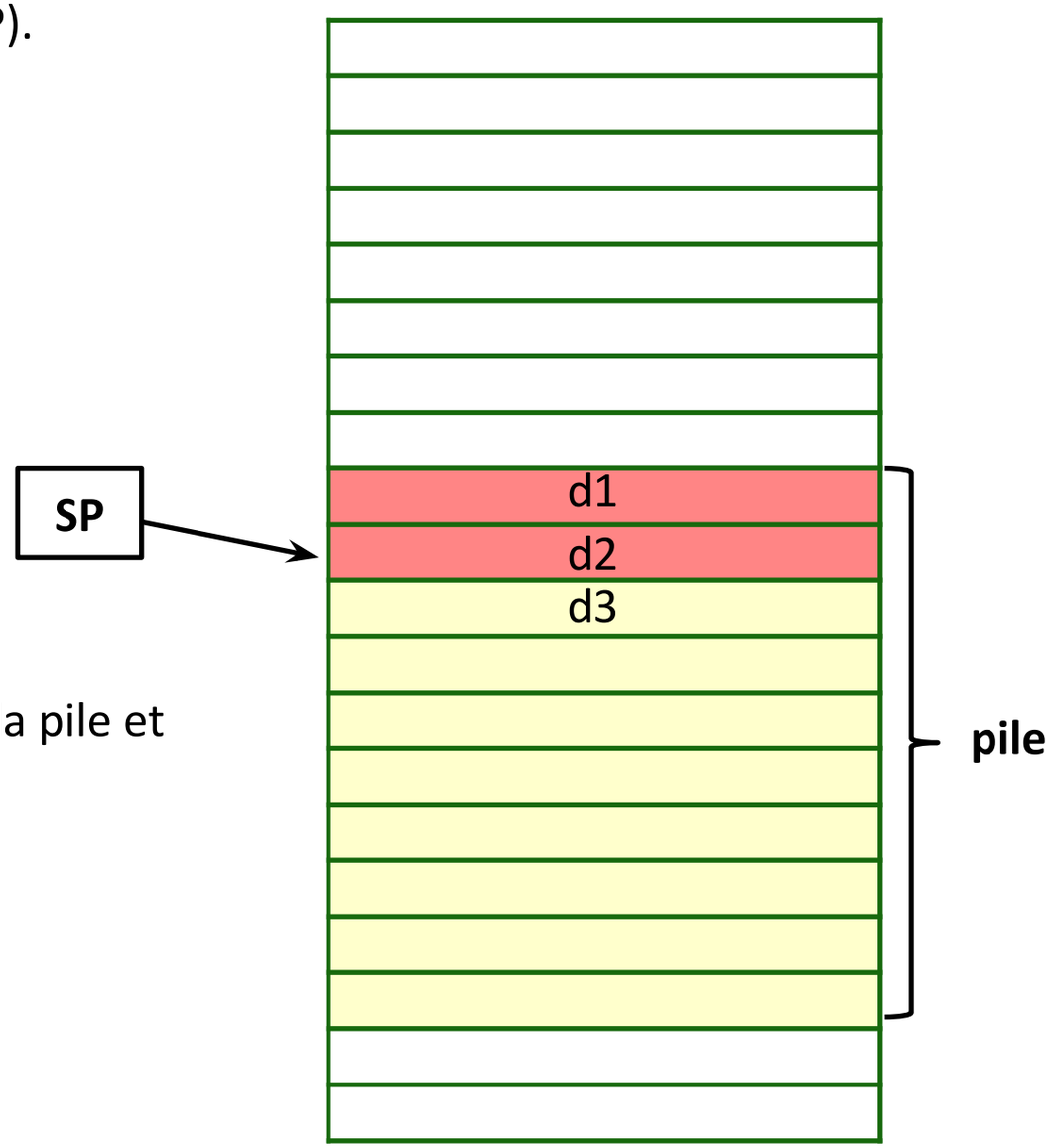


Mémoire

push d1

push d2

push d3



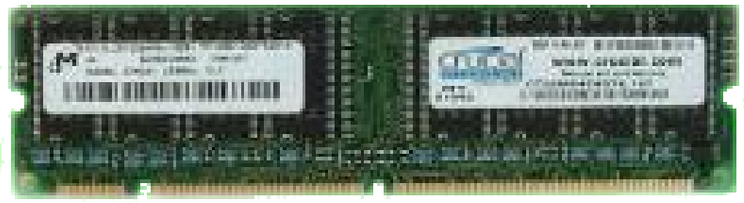
- L'opération pour éliminer la dernière donnée ajoutée à la pile et la copier dans un registre du processeur s'appelle **pop**.

pop r1 r1 **d3**

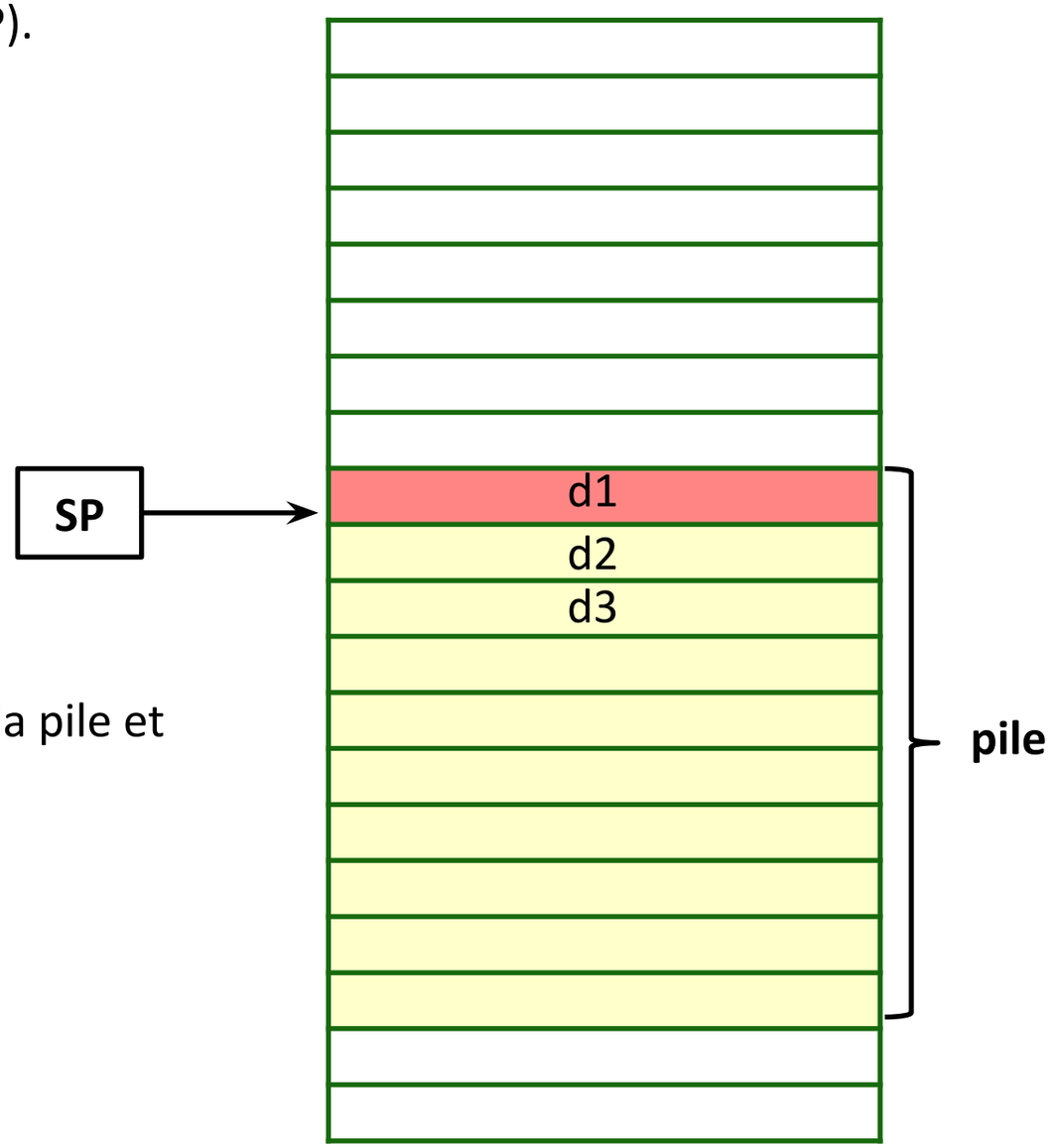
Le concept de pile

- L'opération permettant d'ajouter une donnée à la pile est nommée **push**.

- L'adresse de la dernière donnée ajoutée à la pile est gardée dans un registre du processeur qui s'appelle **stack pointer** (SP).



Mémoire



push d1

push d2

push d3

- L'opération pour éliminer la dernière donnée ajoutée à la pile et la copier dans un registre du processeur s'appelle **pop**.

pop r1 r1 **d3**

pop r2 r2 **d2**

Appel de fonction avec une pile

```
@main   mov sp, stack
        mov r0, #3
        mov r1, #4
        mov r7, #next
        b   mult
@next   .....

@stack  rmw 1
```

```
@mult   push r2
        mov r2, #0
@loop   cmp r0, #0
        beq  fin
        add r2, r2, r1
        sub r0, r0, #1
        b   loop
    @fin    mov r0, r2
        pop r2
        b   r7
```

Fonction récursive

- La fonction **main** met l'argument (2) dans le registre **r0.**
- La fonction **rec** met l'argument (x-1) dans le registre **r0**.
- Les deux fonctions mettent l'adresse de retour dans le registre **r7**.
- La fonction **rec** met le résultat dans le registre **r2**.

```
main:
    rec(2)

rec(x) :
    if x == 0:
        return 1;
    else:
        return rec(x-1) + x
```

Fonction récursive

Code machine de la fonction **main.**

```
main:
   rec(2)

rec(x) :
   if x == 0:
      return 1;
   else:
      return rec(x-1) + x
```

```
@main    mov sp, stack
         mov r0, #2
         push r2
         mov r7, rtn
         b    rec
@rtn     mov r0, r2
         pop r2
@fin     b    fin

@stack   rmw 1
```

Fonction récursive

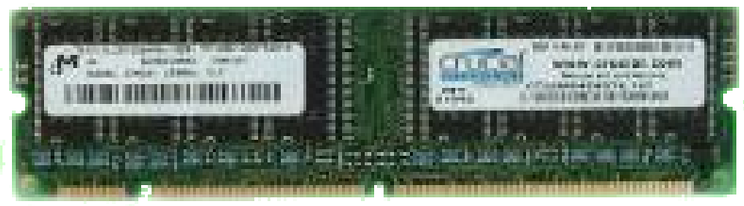
Code machine de la fonction **rec.**

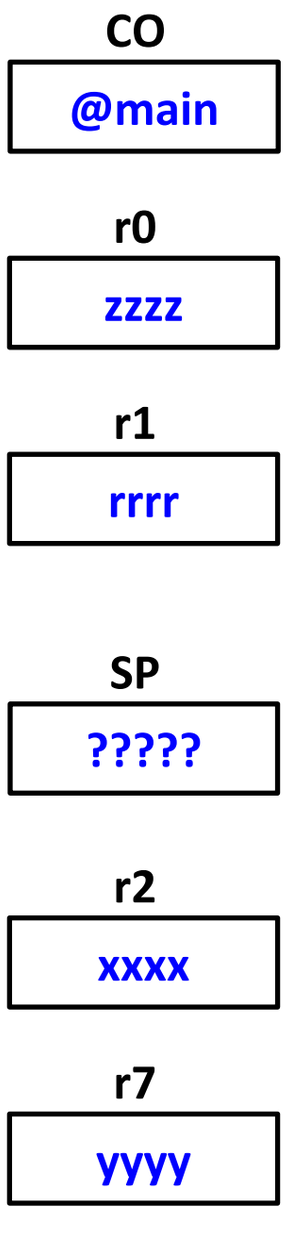
```
main:
   rec(2)

rec(x) :
   if x == 0:
      return 1;
   else:
      return rec(x-1) + x
```

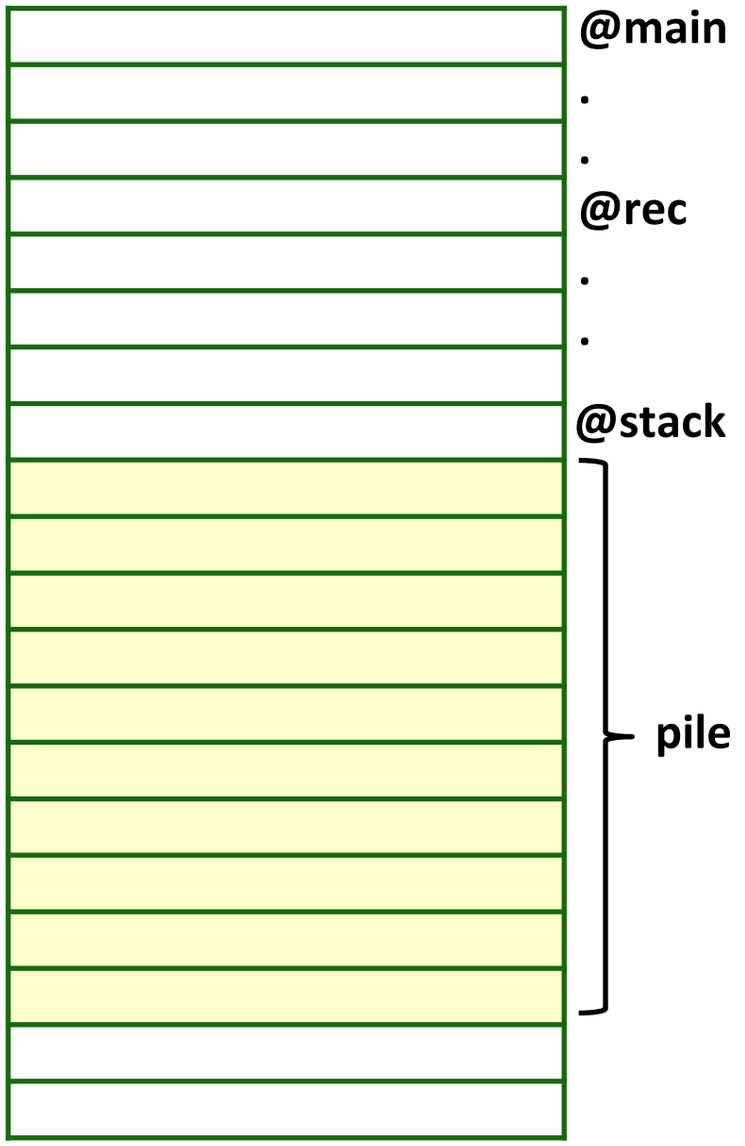
```
@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b      rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b      r7
```

Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile

@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```





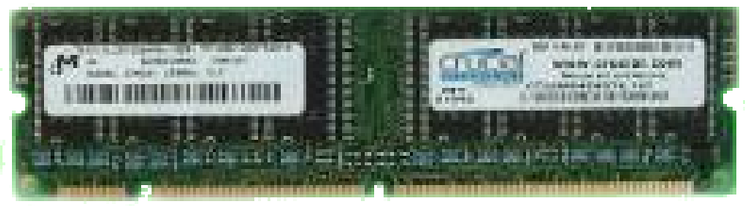


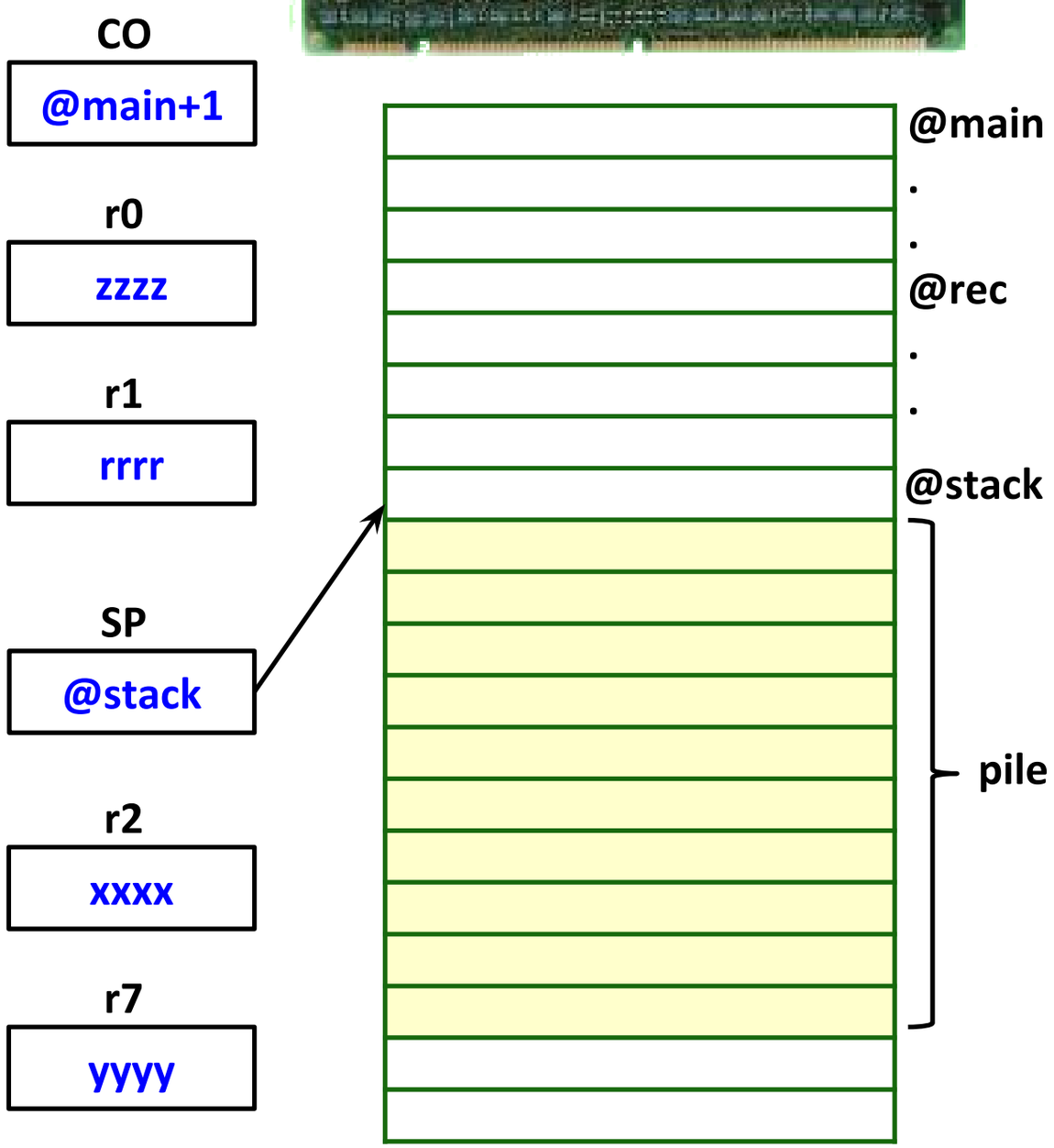
Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1
```

pile

```
@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```

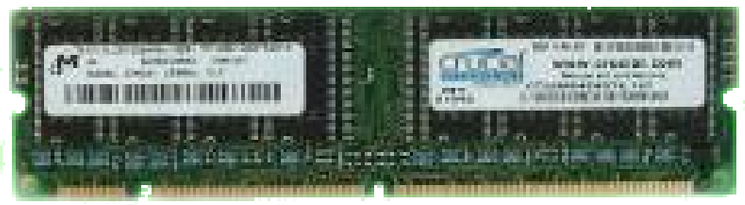


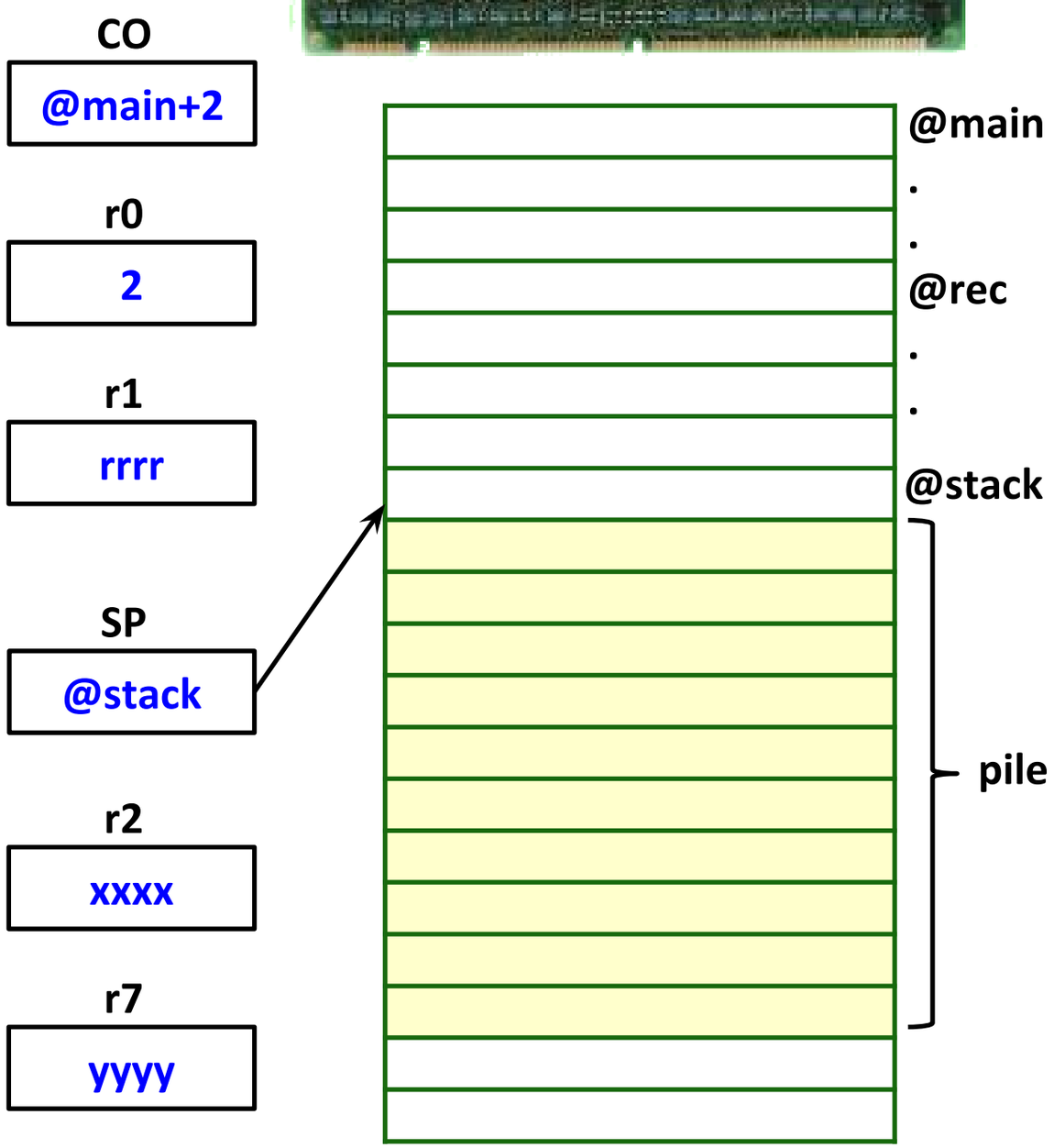


Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile
```

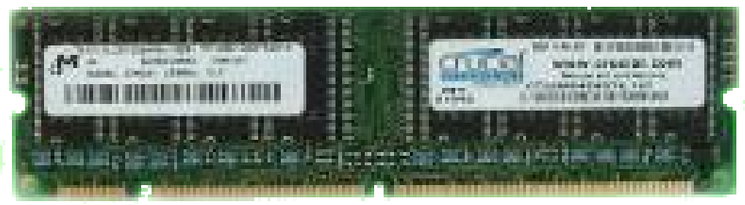
```
@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```

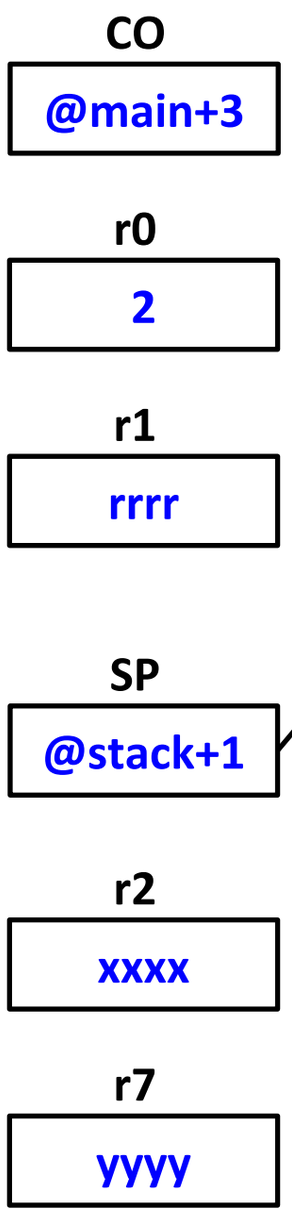


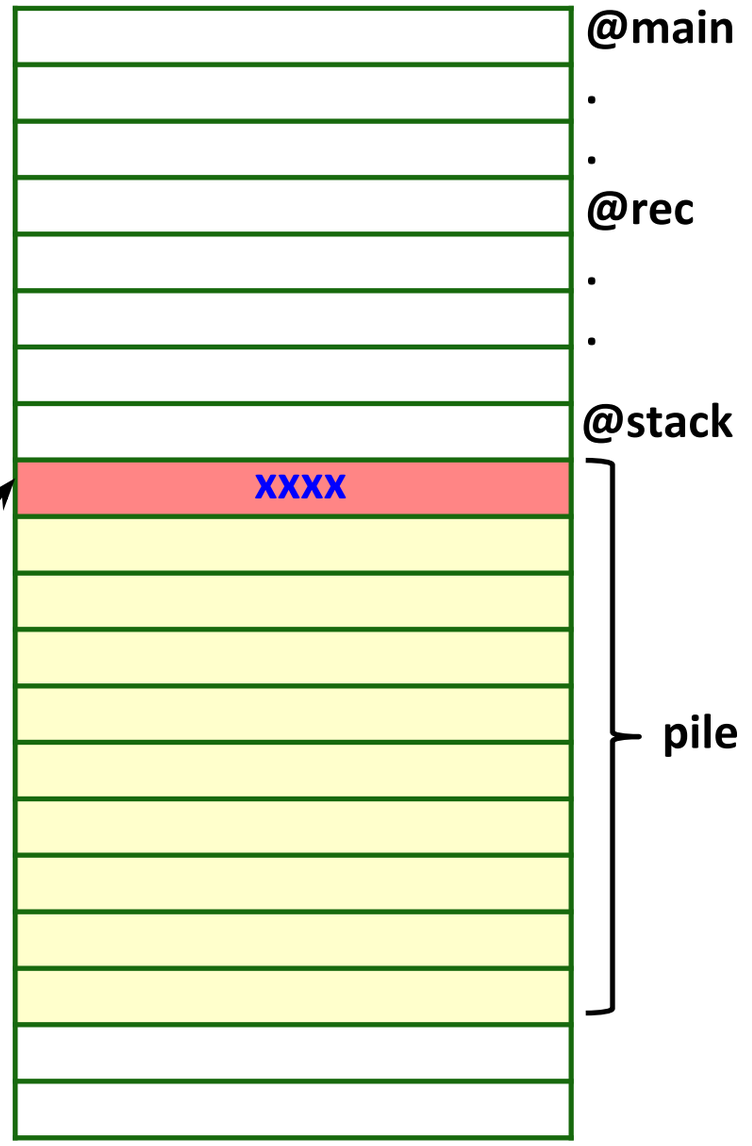


Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile

@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```

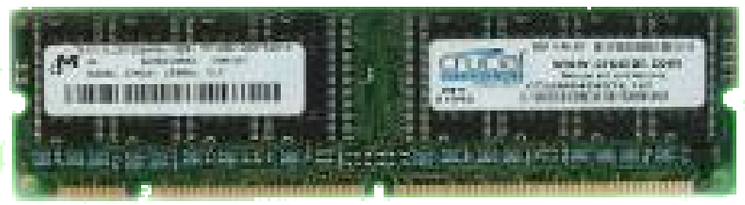


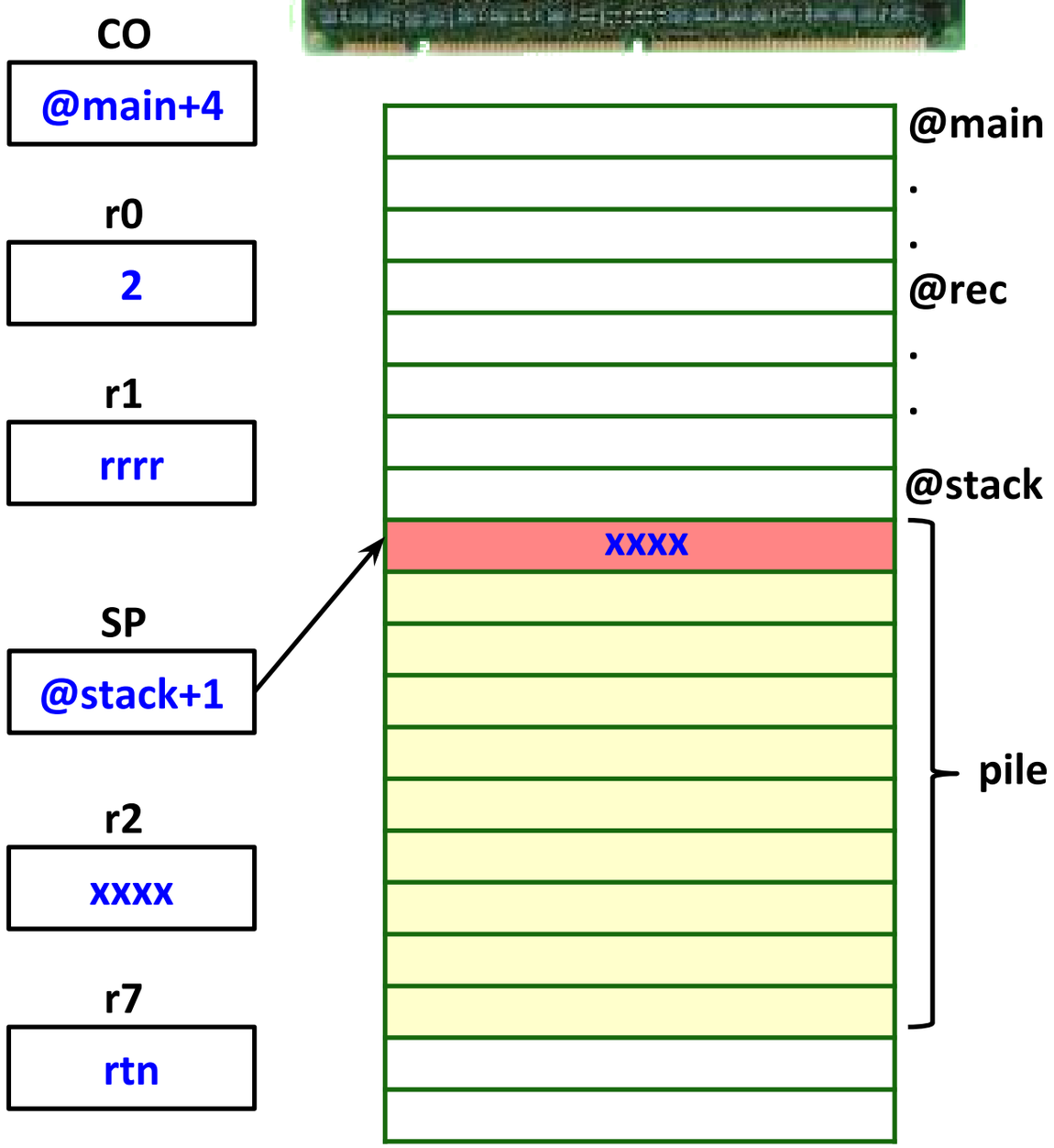




Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b   rec
@rtn    mov r0, r2
        pop r2
@fin    b   fin

@stack  rmw 1

        pile

@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b   rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b   r7
```

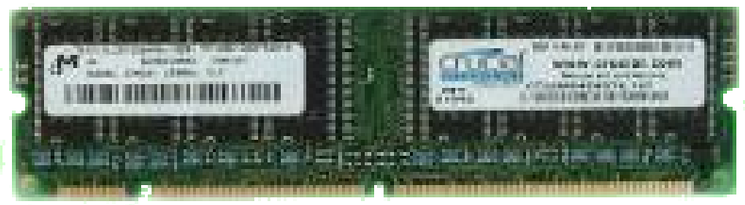


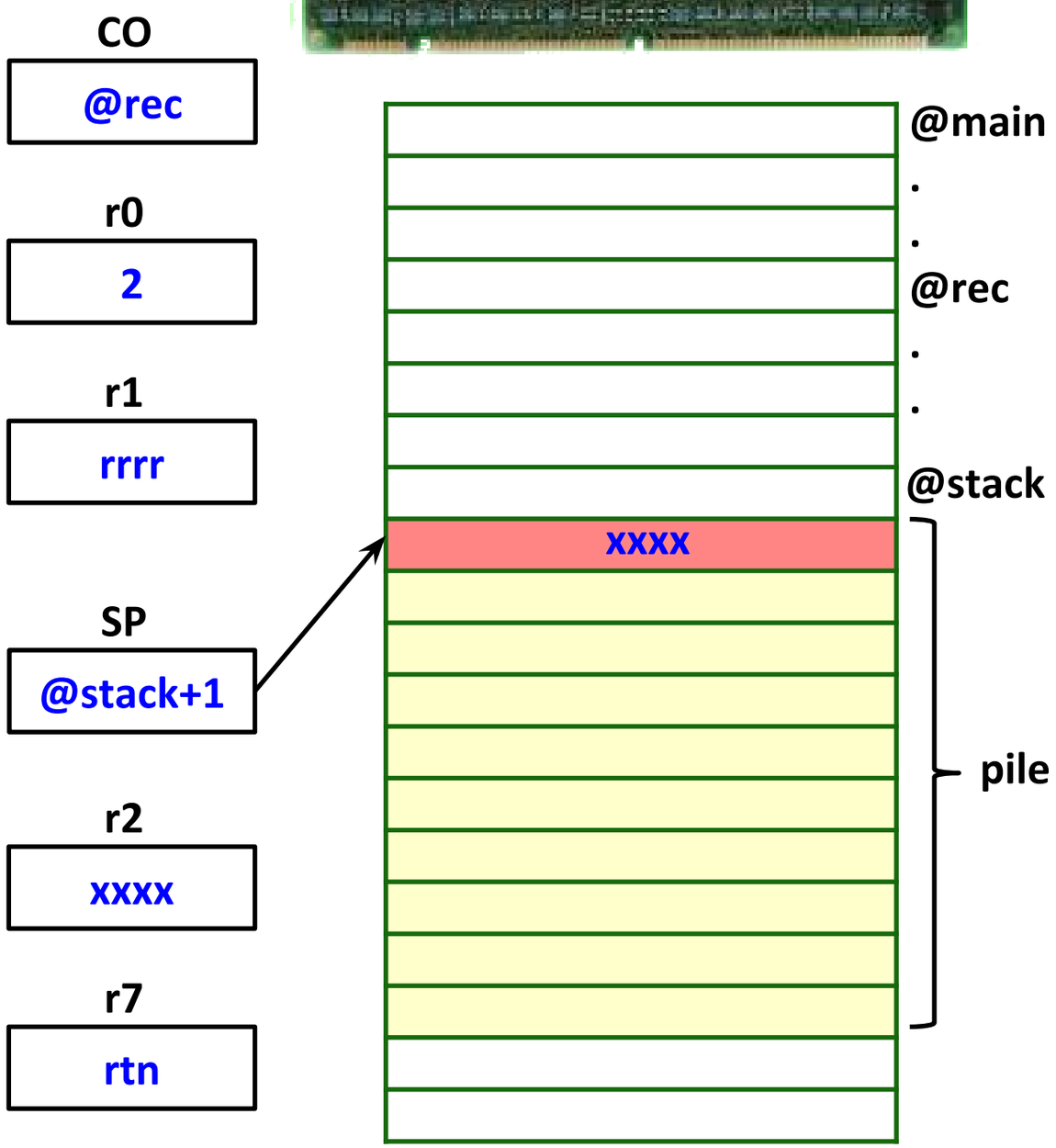


Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile
```

```
@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```

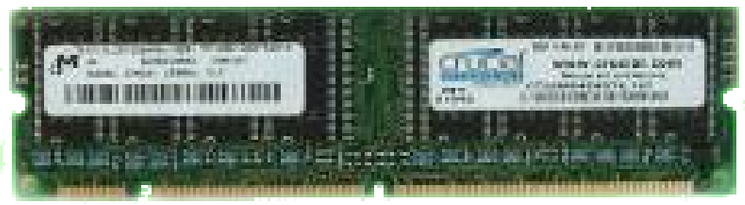


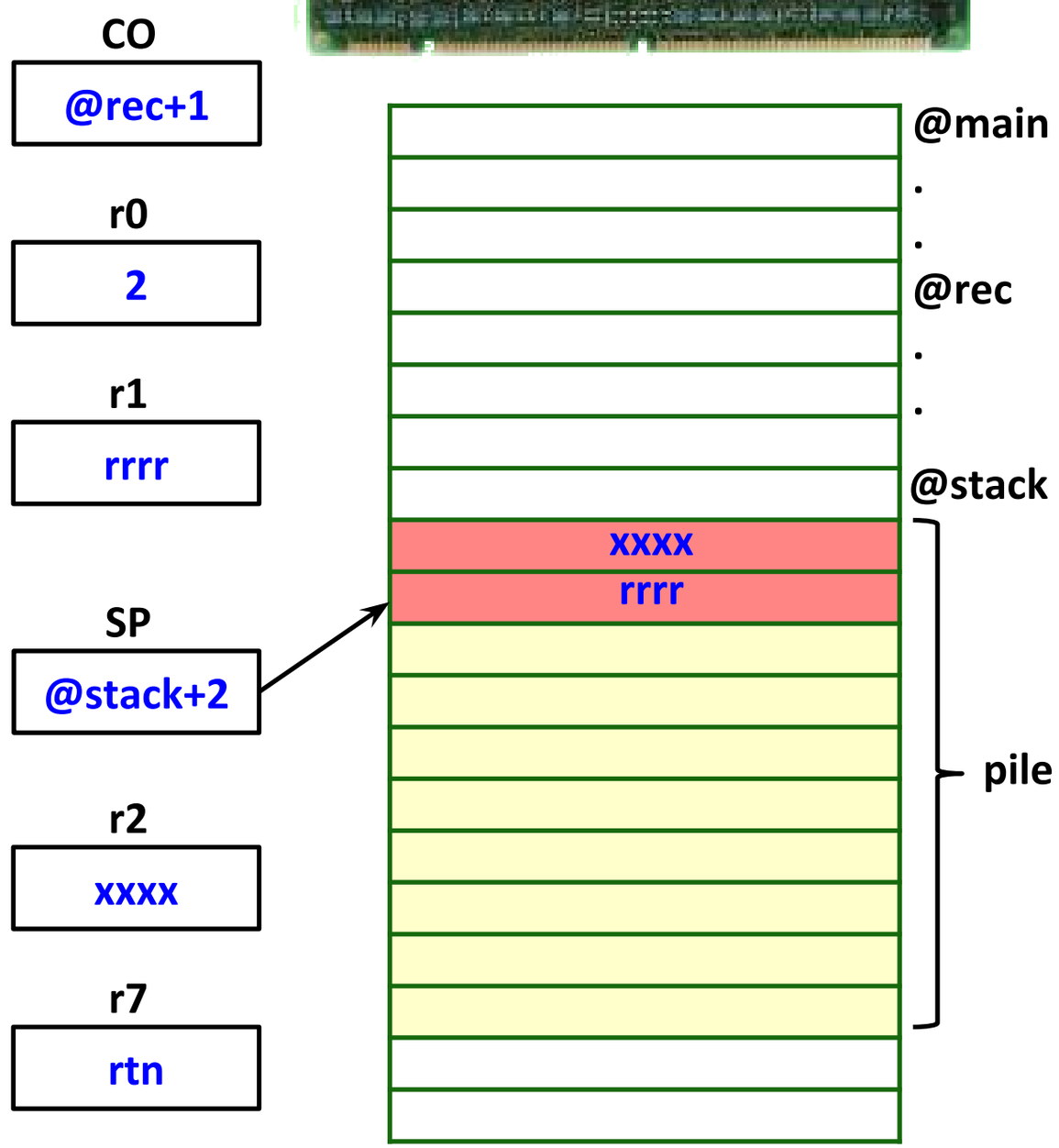


Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile
```

```
@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```

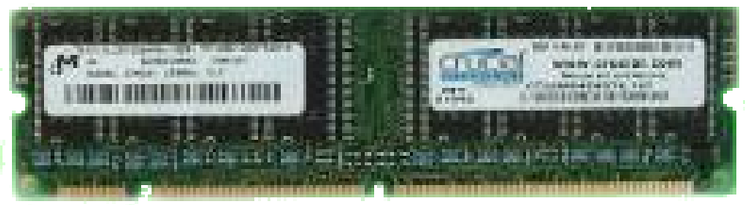


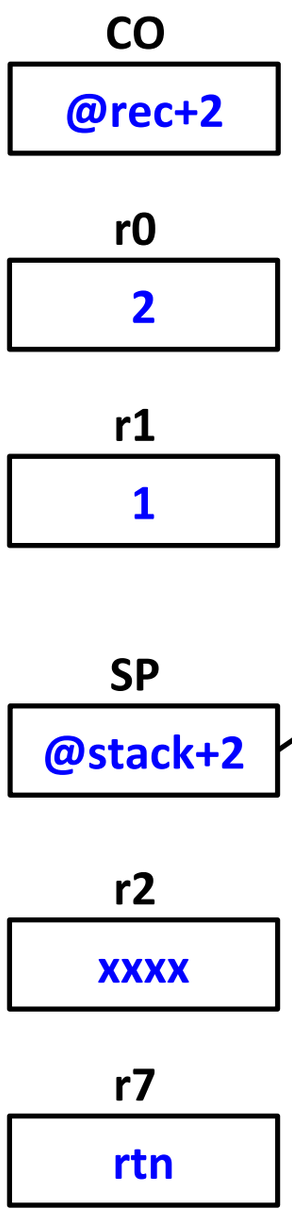


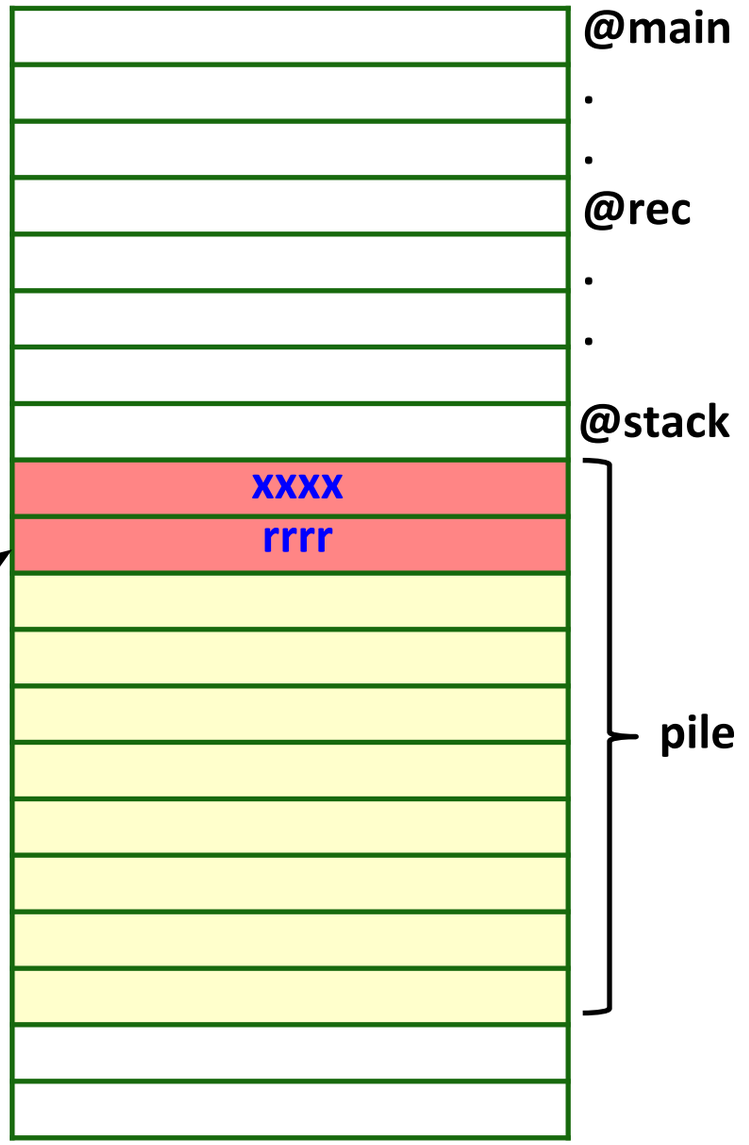
Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile
```

```
@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```





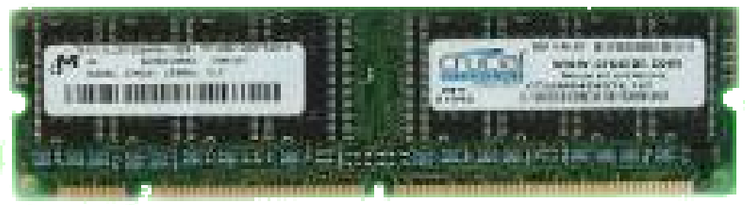


Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1
```

pile

```
@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```



CO
@rec+3

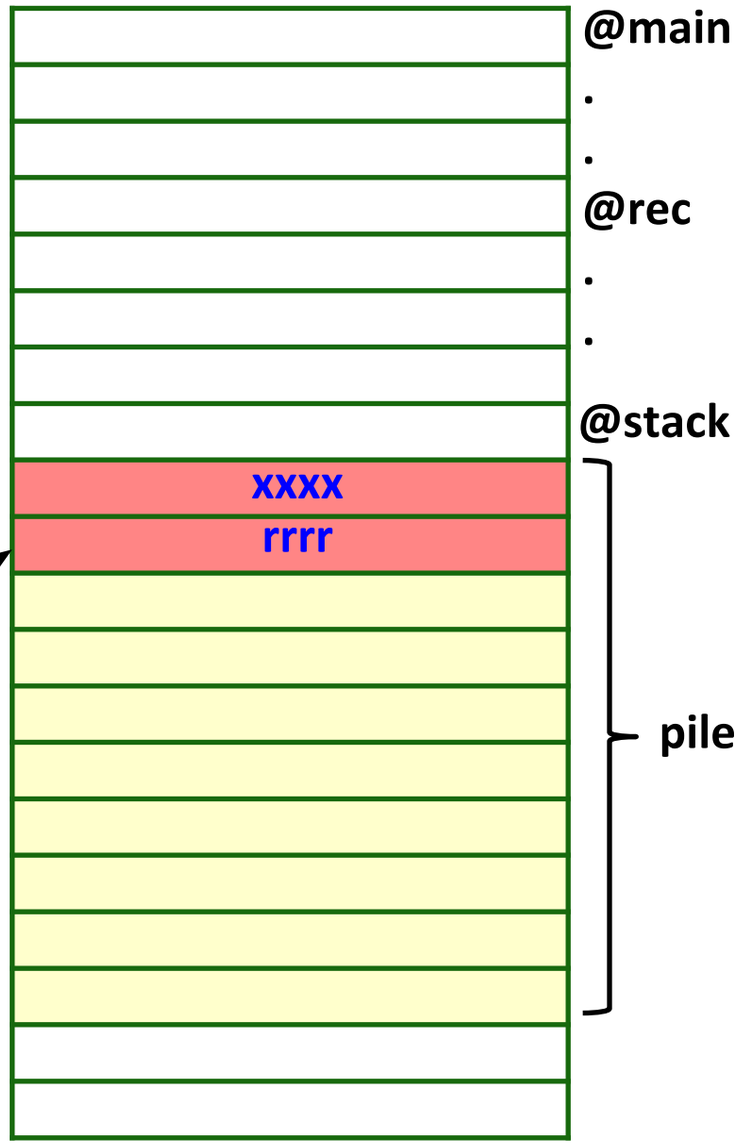
r0
2

r1
1

SP
@stack+2

r2
xxxx

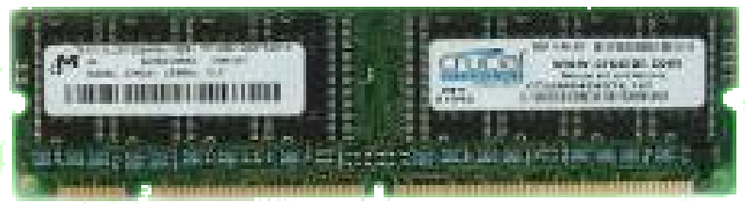
r7
rtn

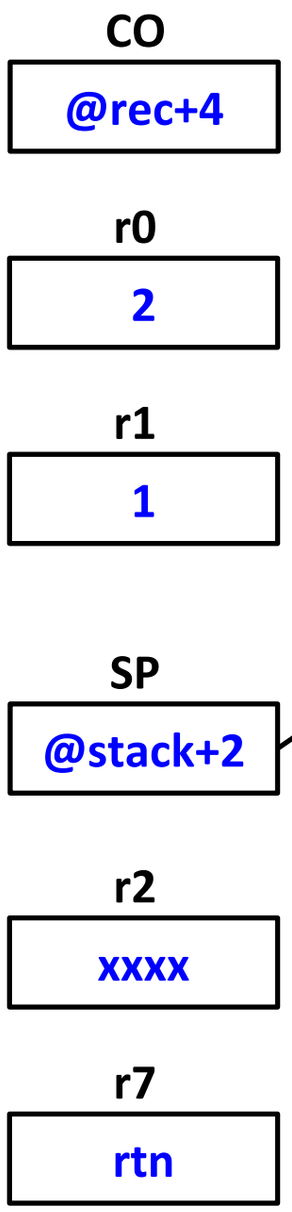


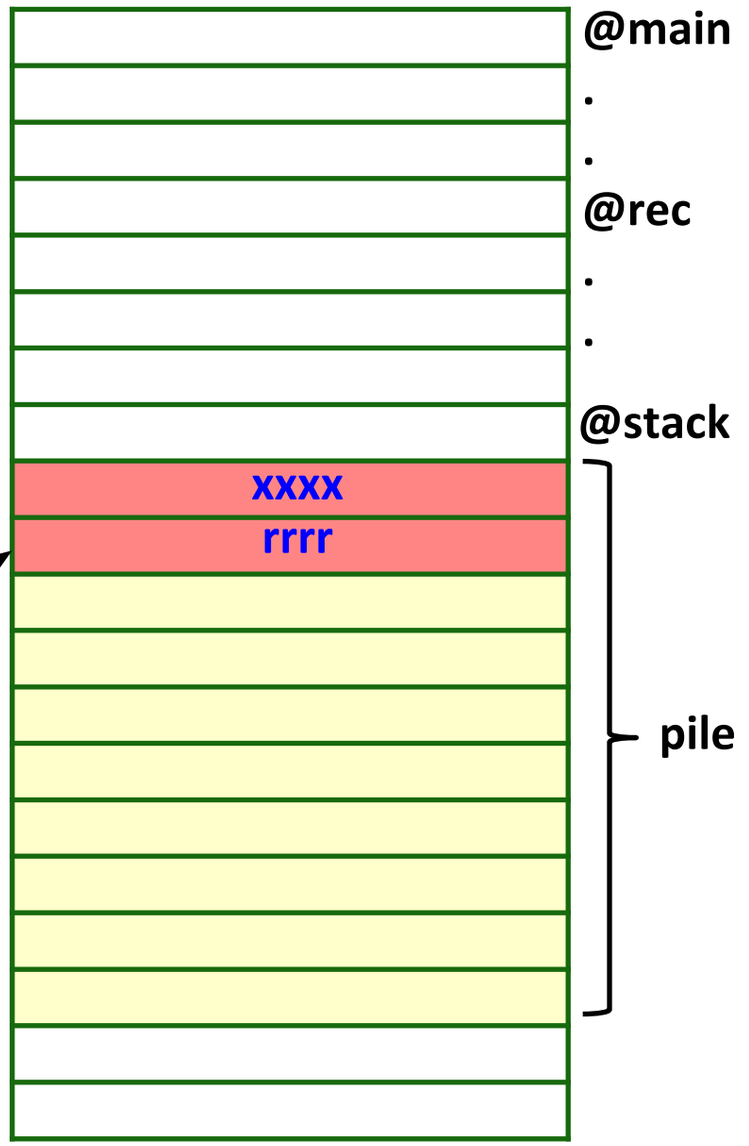
Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile
```

```
@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```

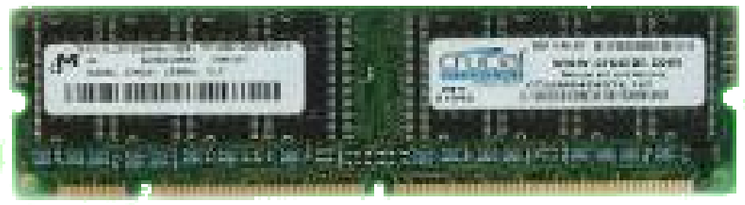


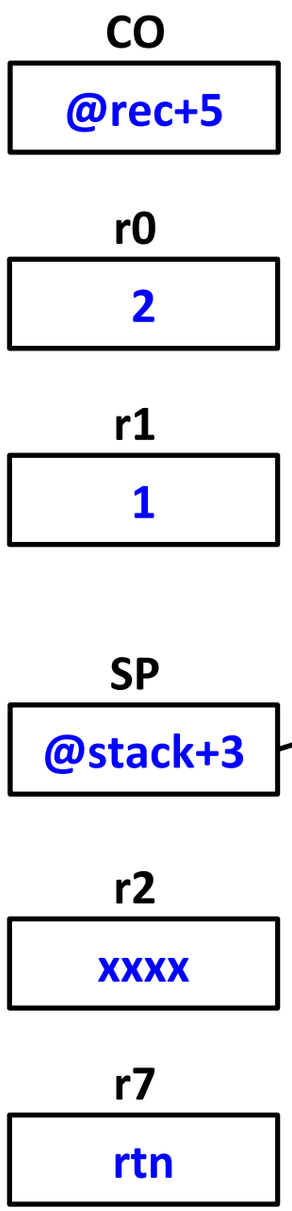


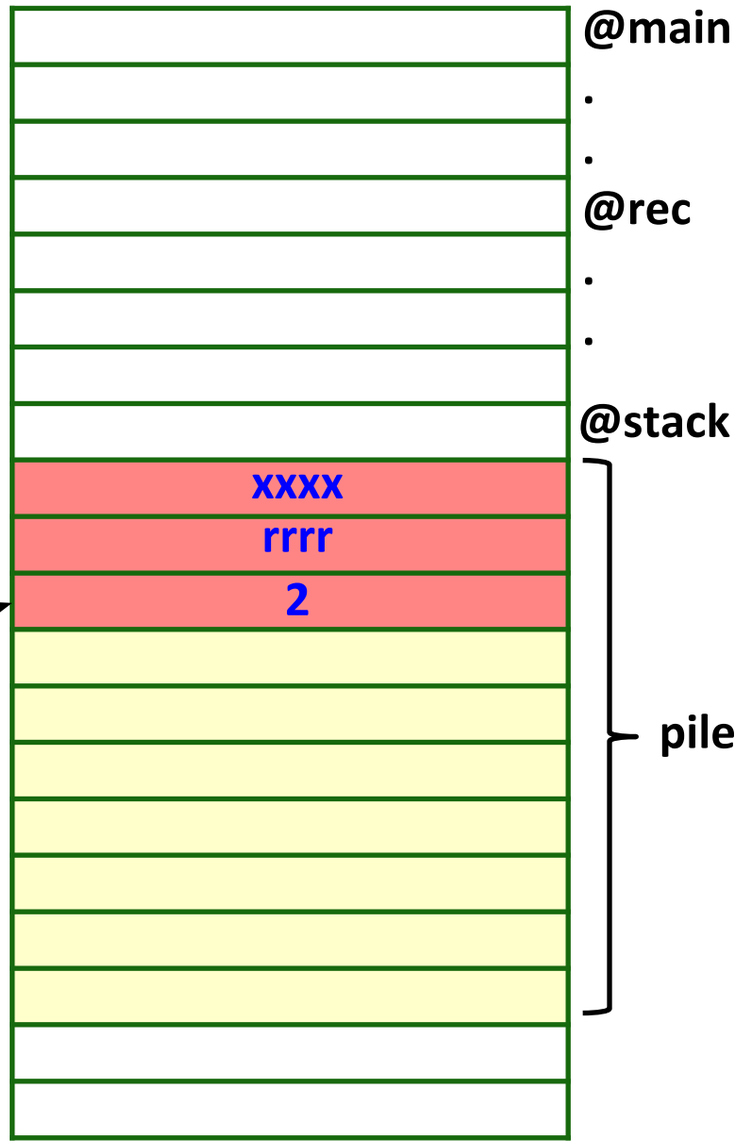


Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile

@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```

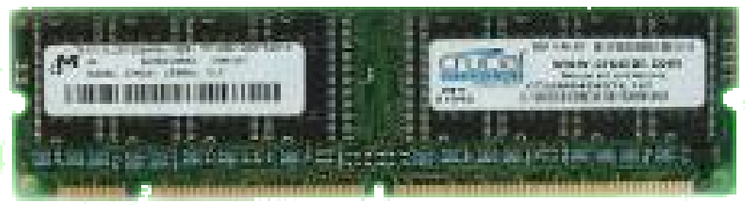


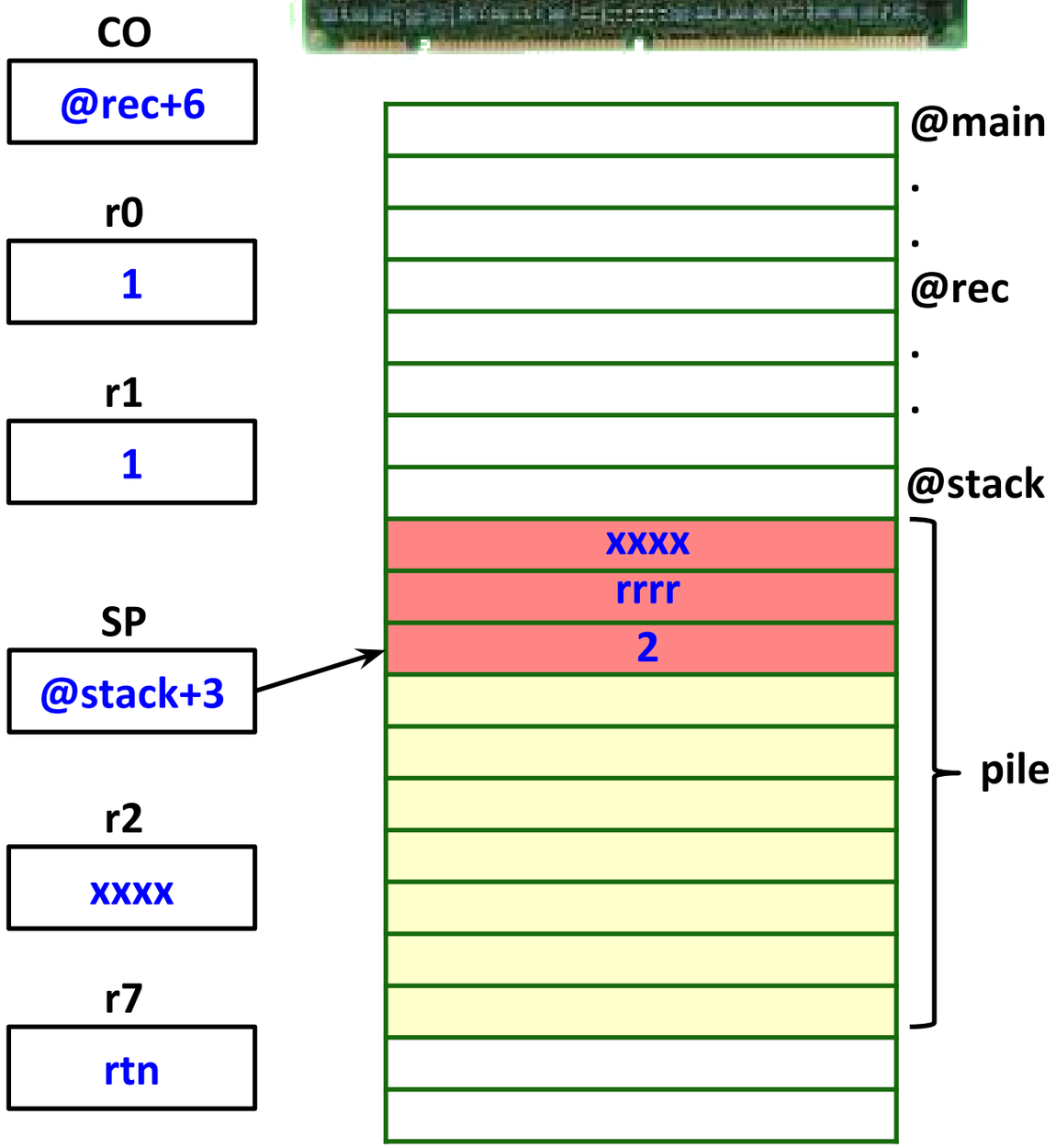




Fonction récursive

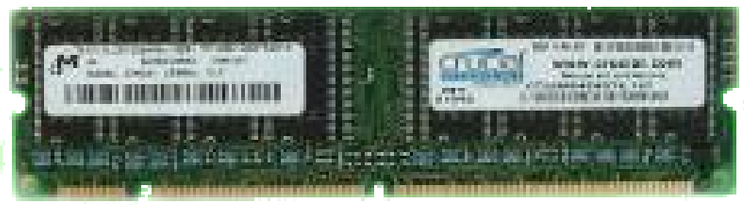
```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile

@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```

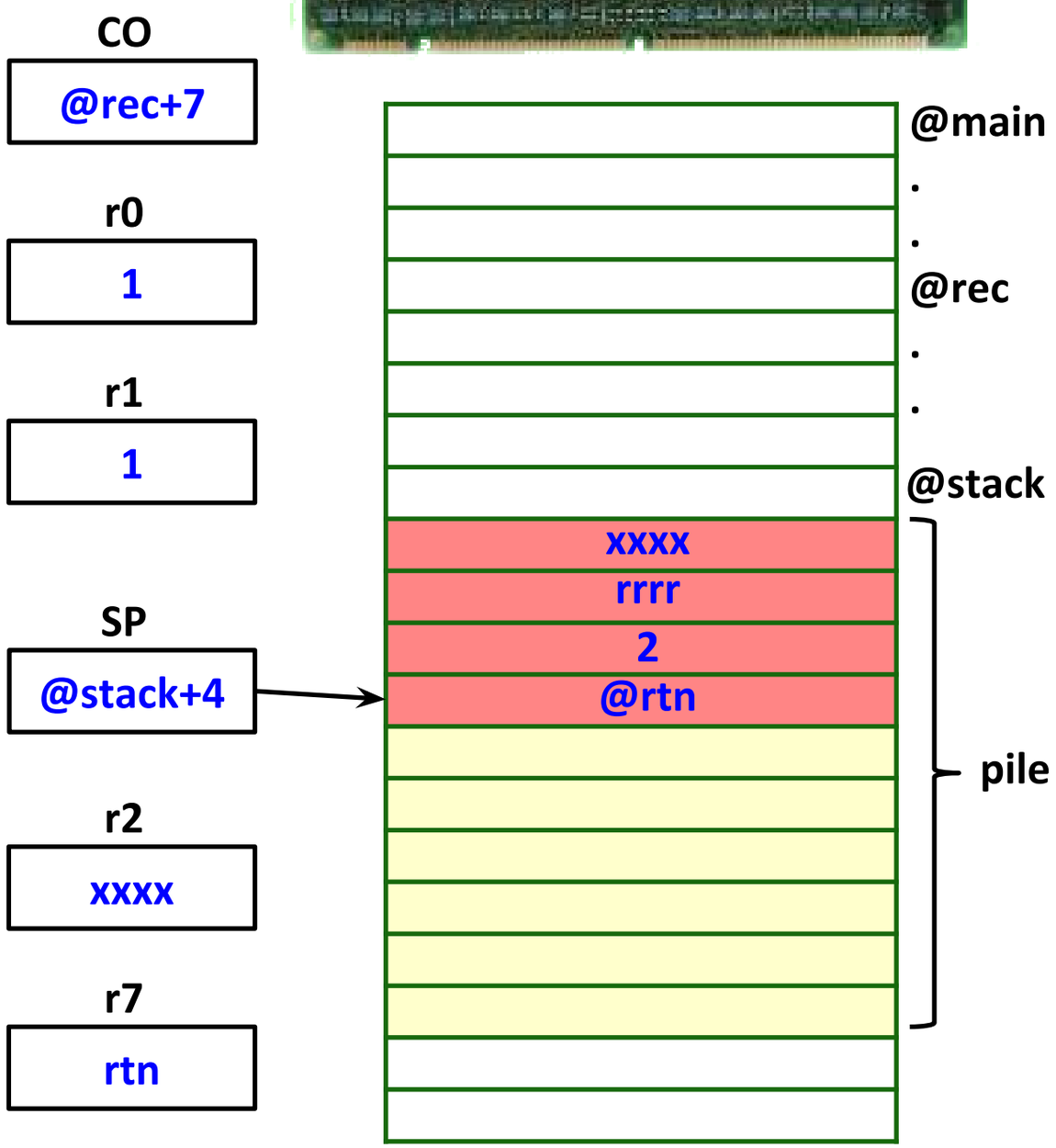




Fonction récursive

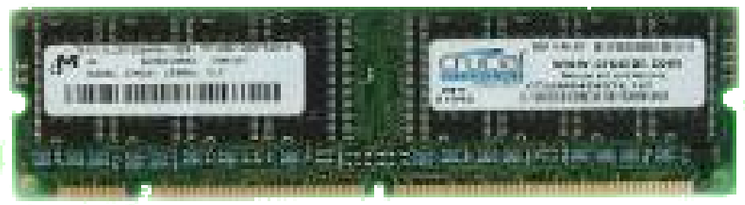
```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile

@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```

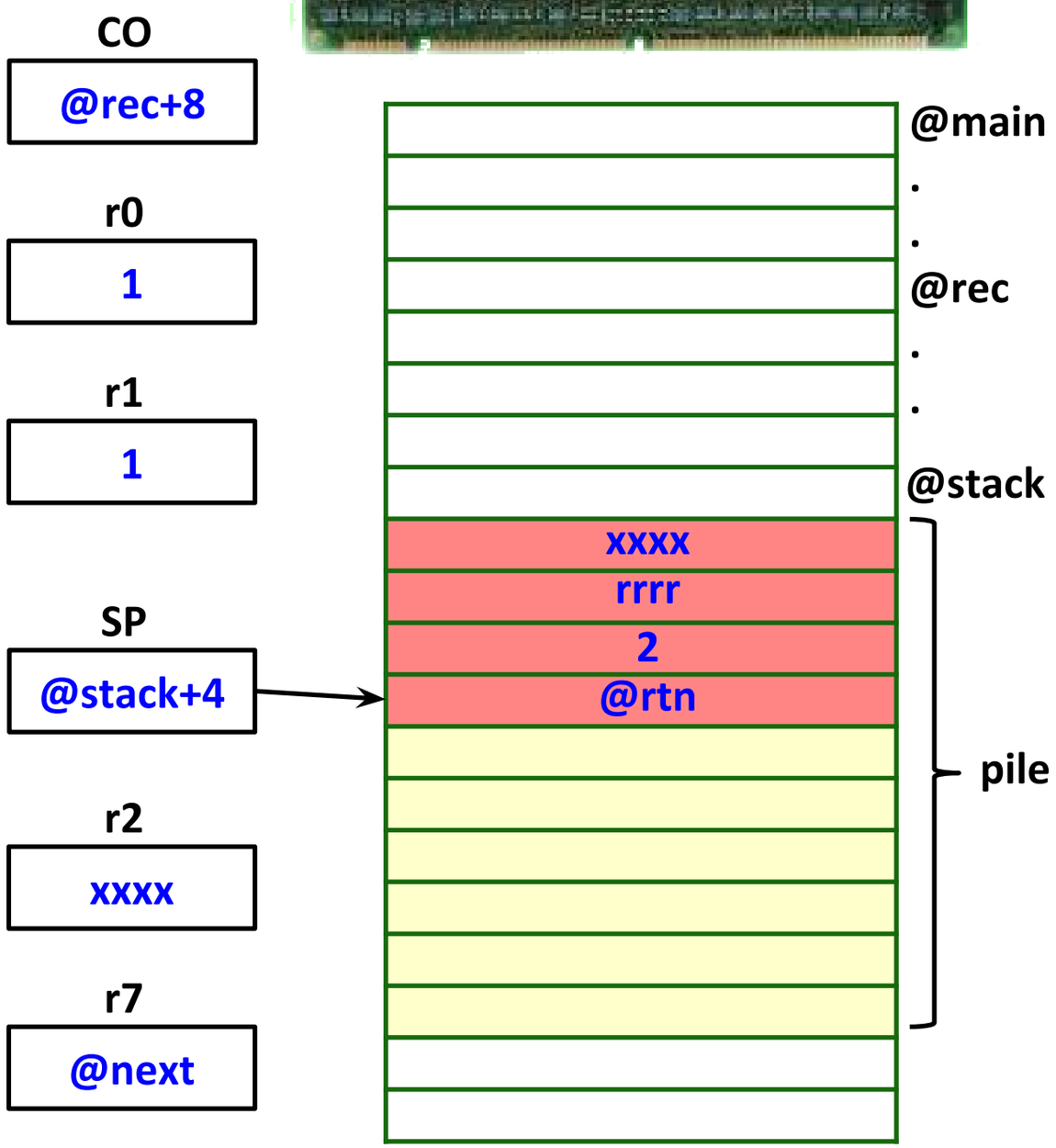




Fonction récursive

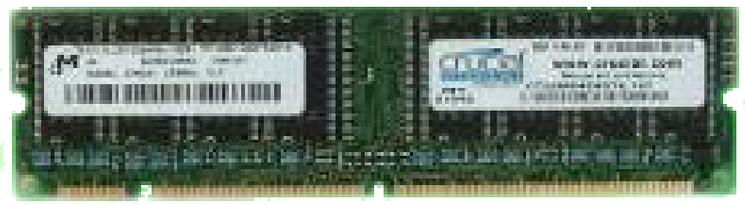
```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile

@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```

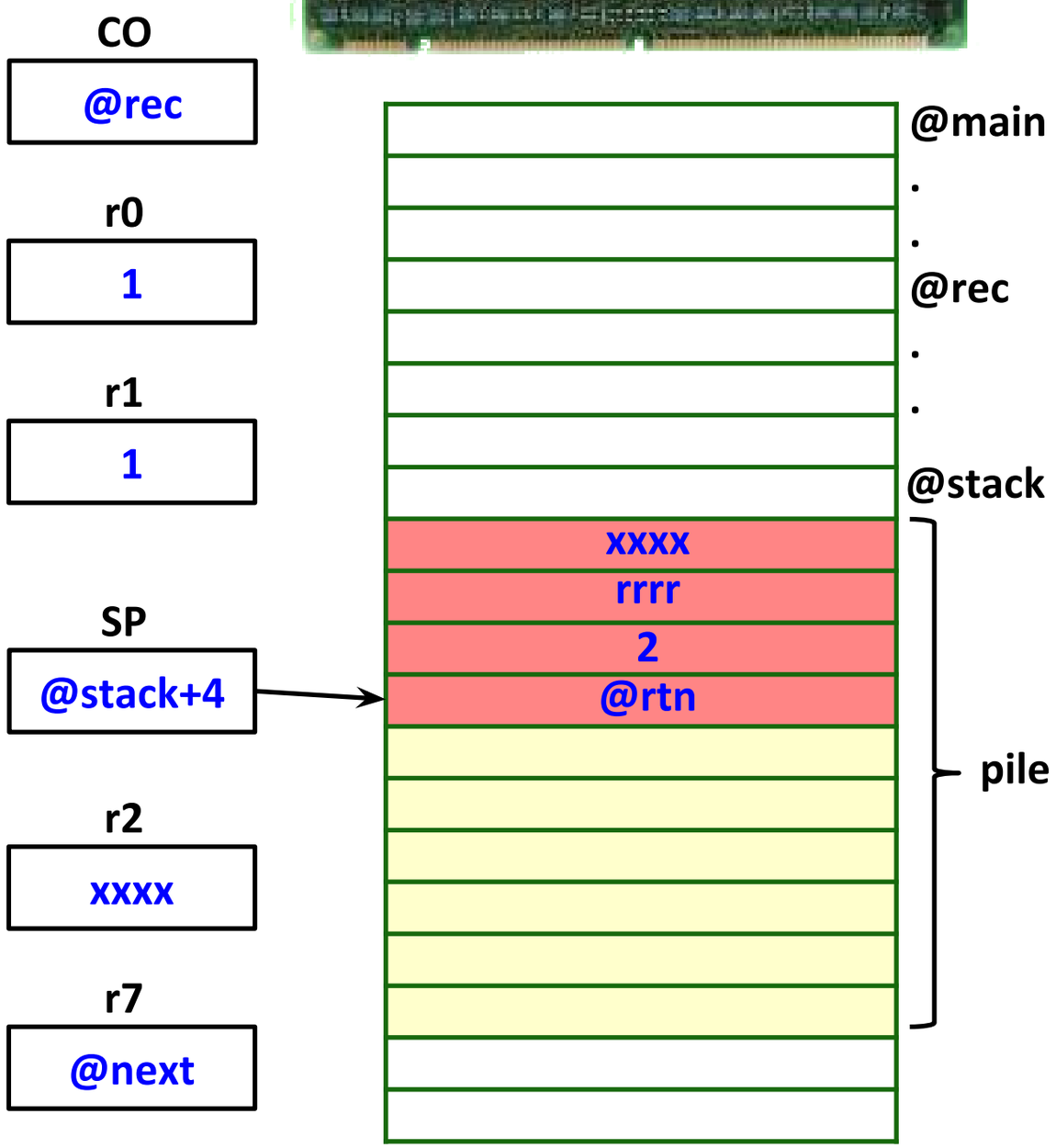




Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile

@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```



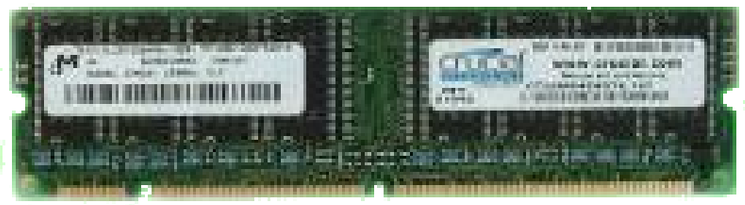


Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1
```

pile

```
@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```



CO
@rec+1

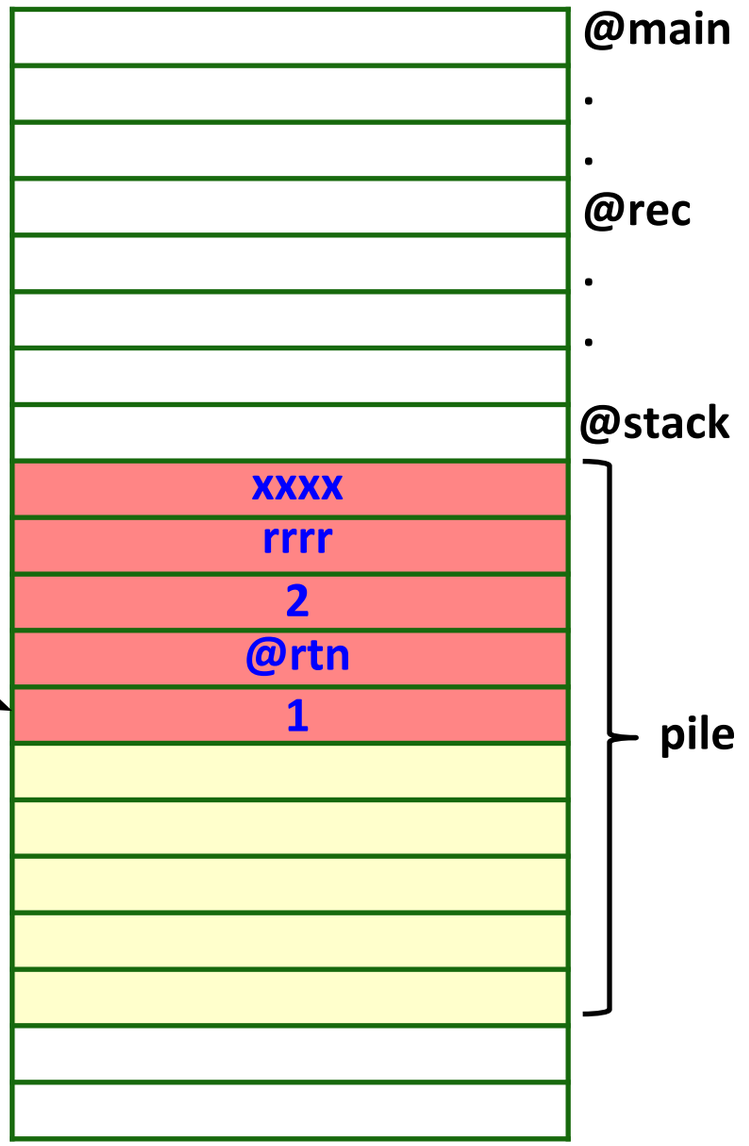
r0
1

r1
1

SP
@stack+5

r2
xxxx

r7
@next

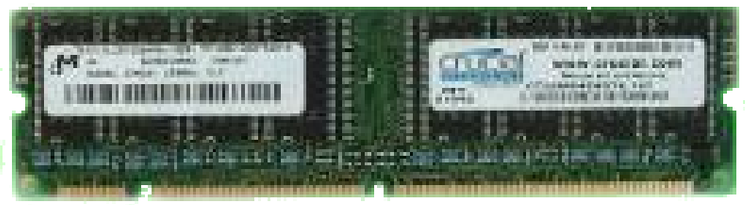


Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1
```

pile

```
@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```



CO
@rec+2

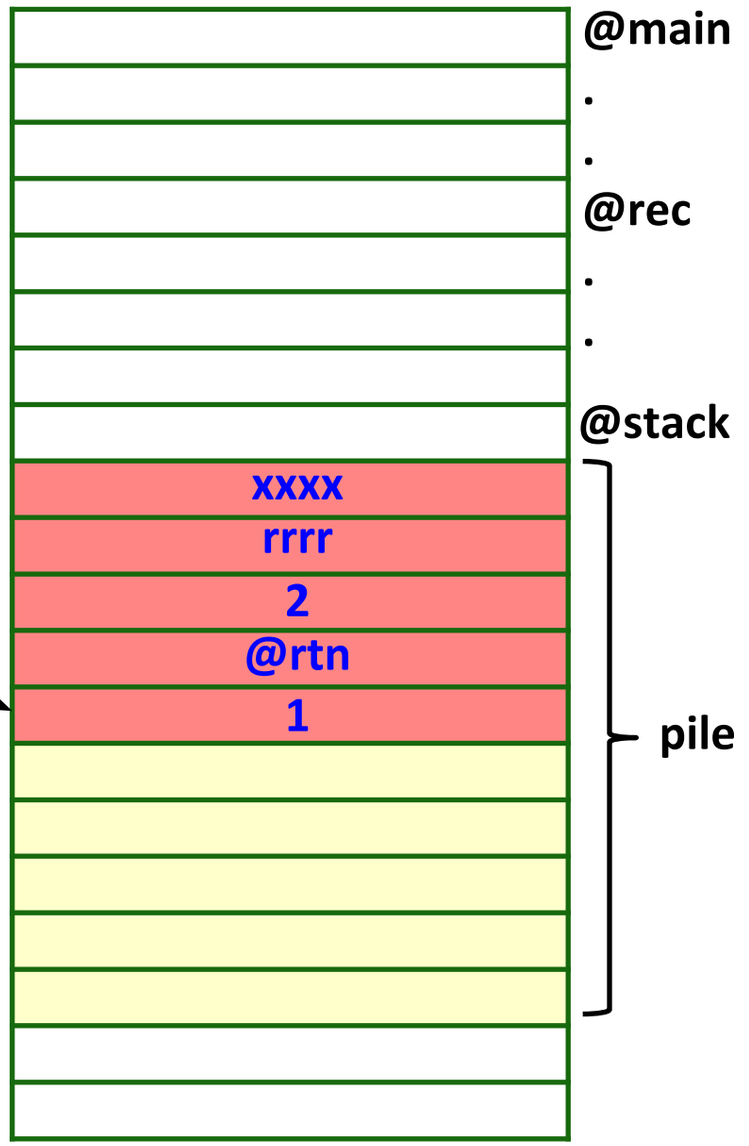
r0
1

r1
1

SP
@stack+5

r2
xxxx

r7
@next

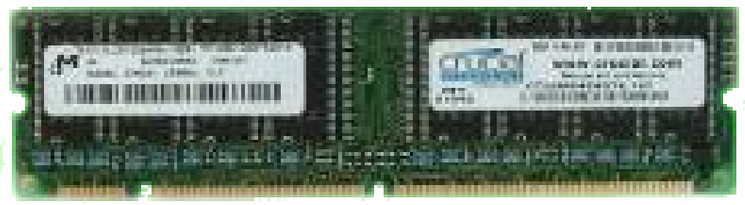


Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1
```

pile

```
@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```



CO
@rec+3

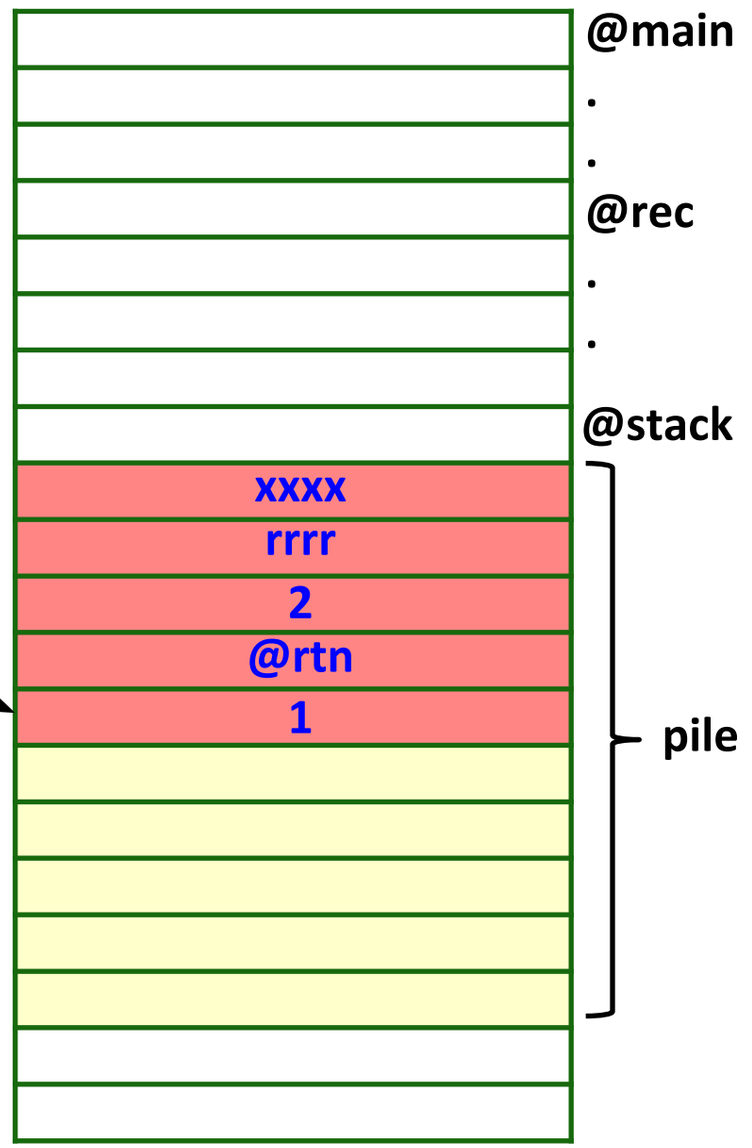
r0
1

r1
1

SP
@stack+5

r2
xxxx

r7
@next

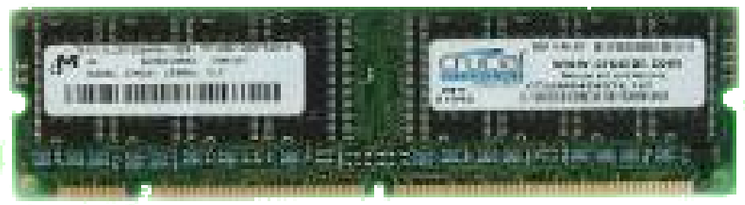


Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1
```

pile

```
@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```



CO
@rec+4

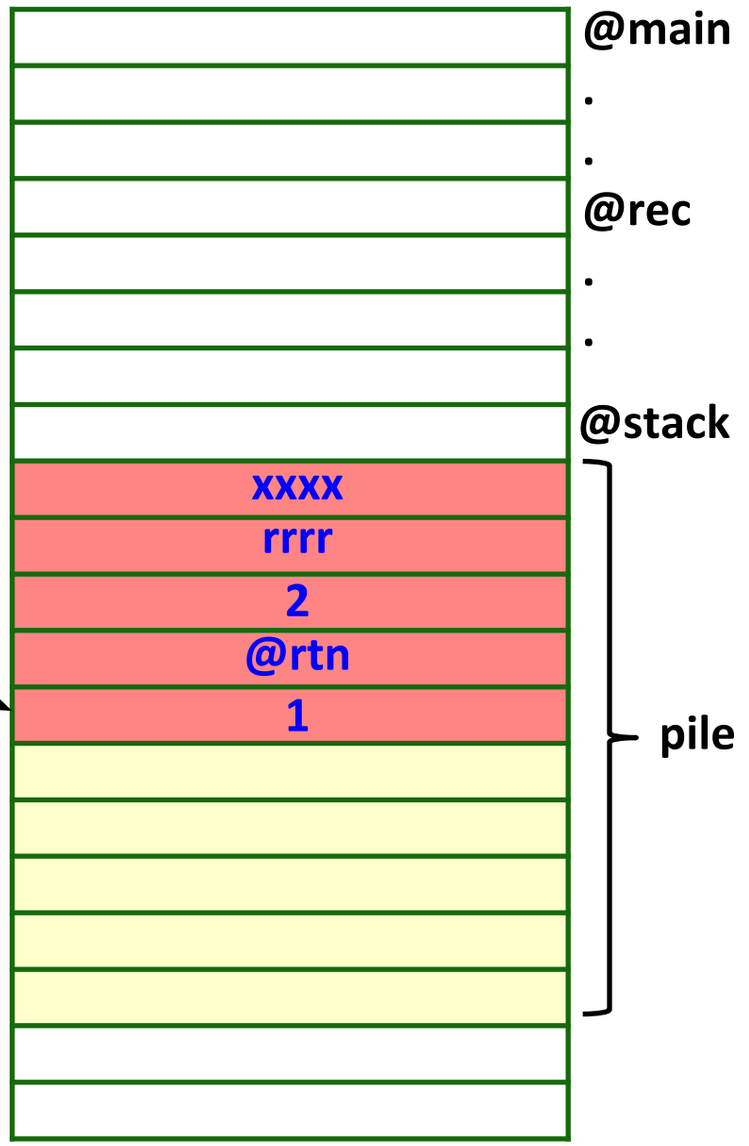
r0
1

r1
1

SP
@stack+5

r2
xxxx

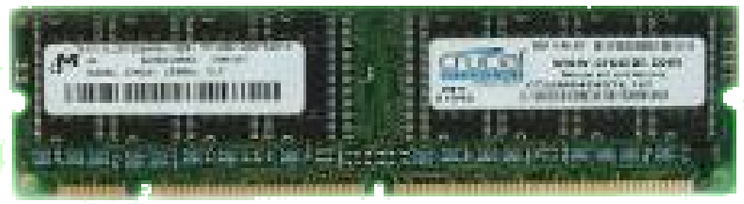
r7
@next



Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```

pile



CO
@rec+5

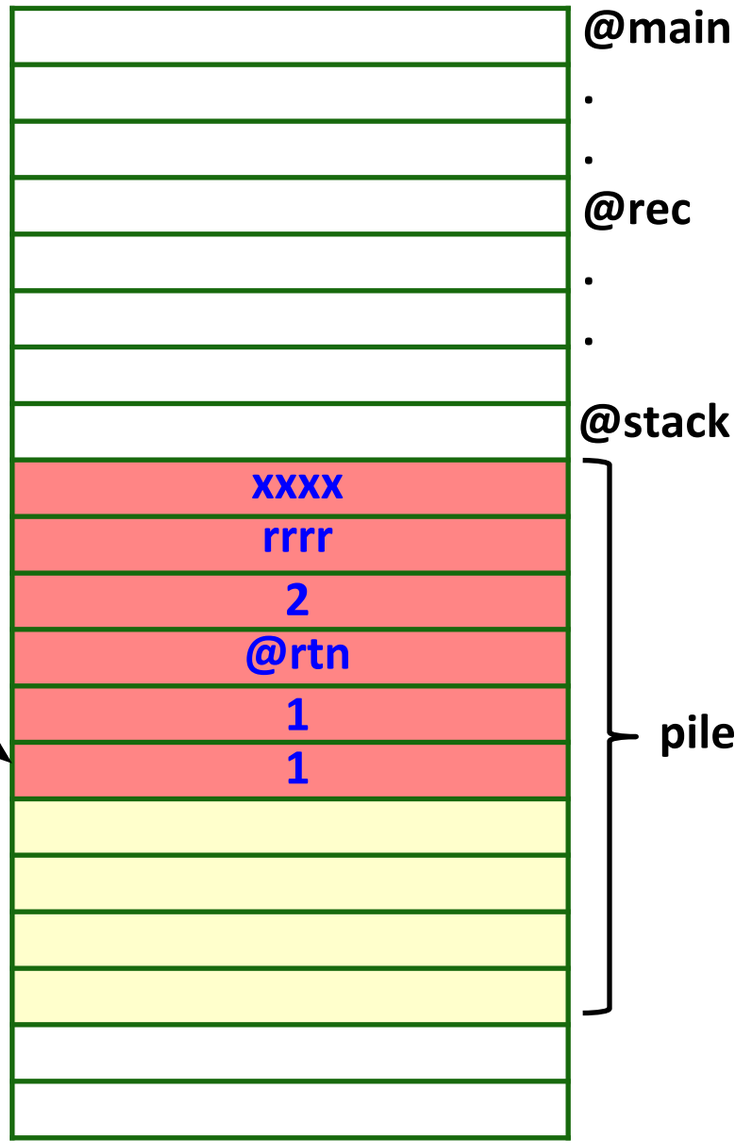
r0
1

r1
1

SP
@stack+6

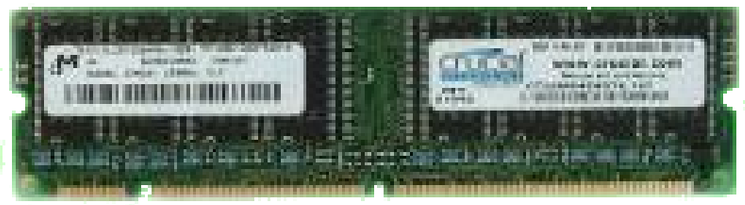
r2
xxxx

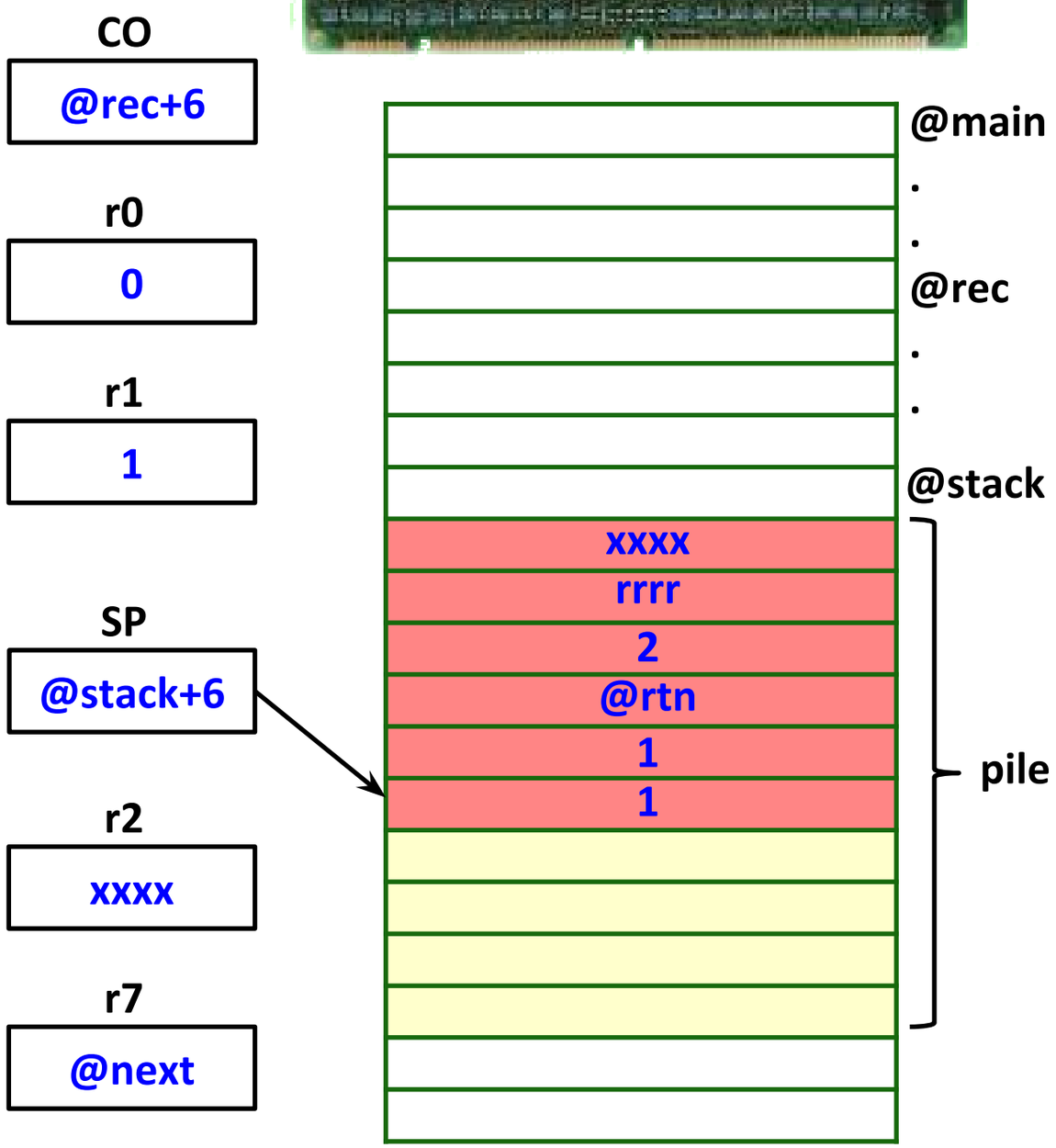
r7
@next



Fonction récursive

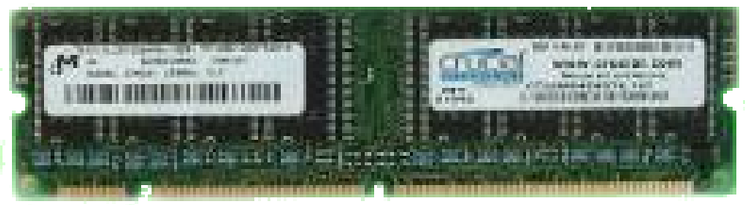
```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile

@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```

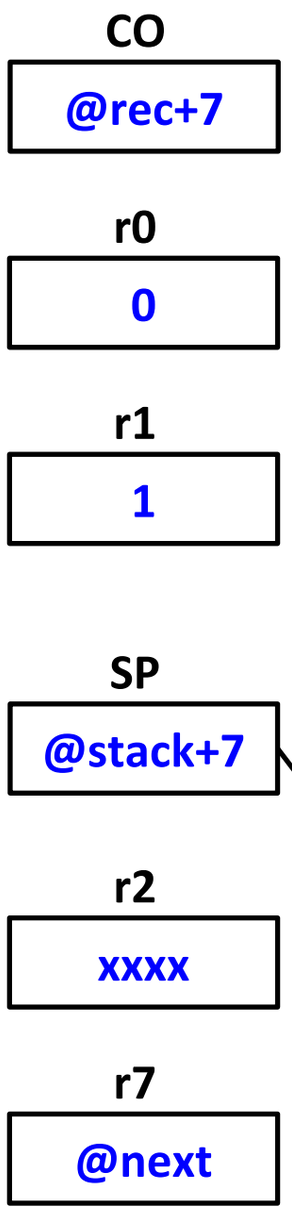


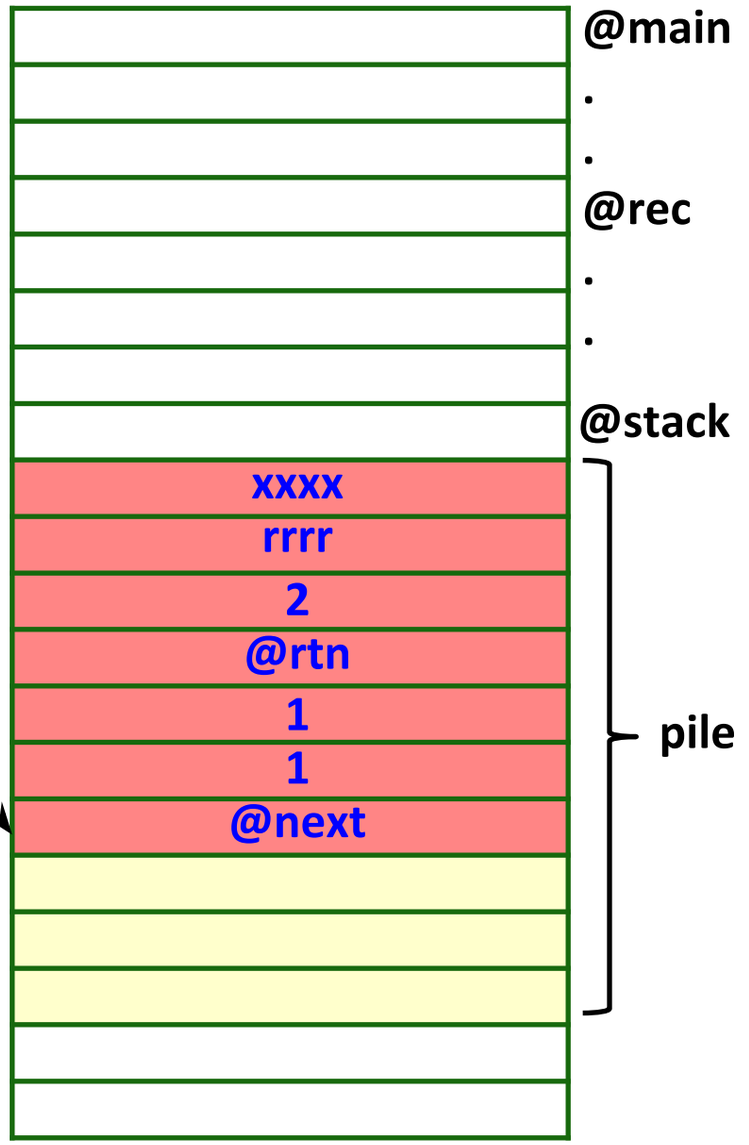


Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile

@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```





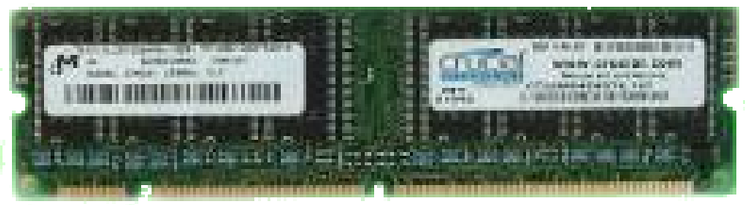


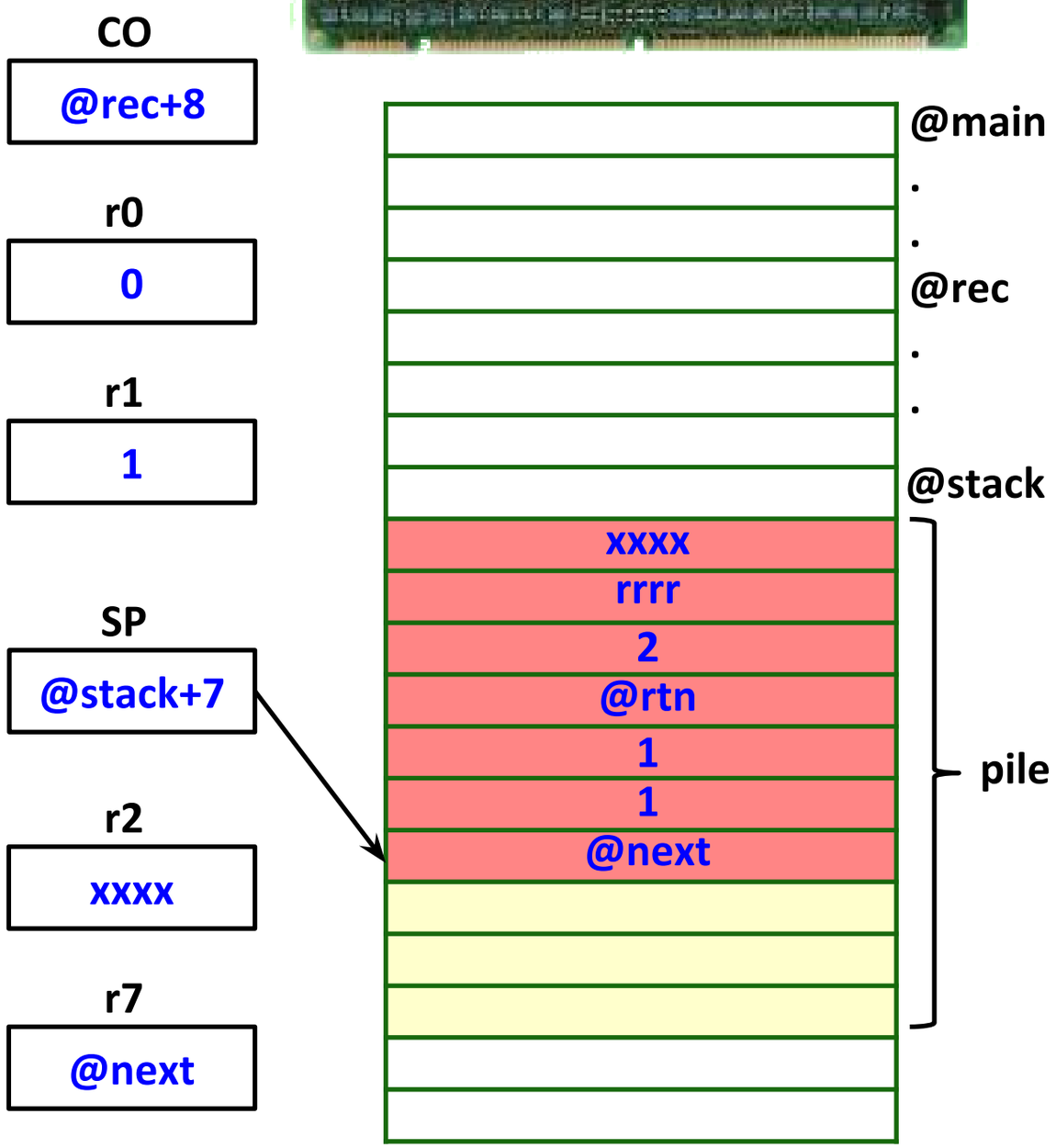
Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1
```

pile

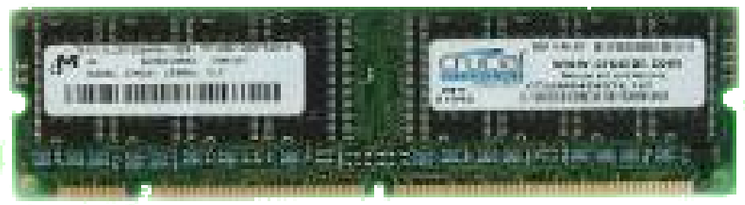
```
@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```

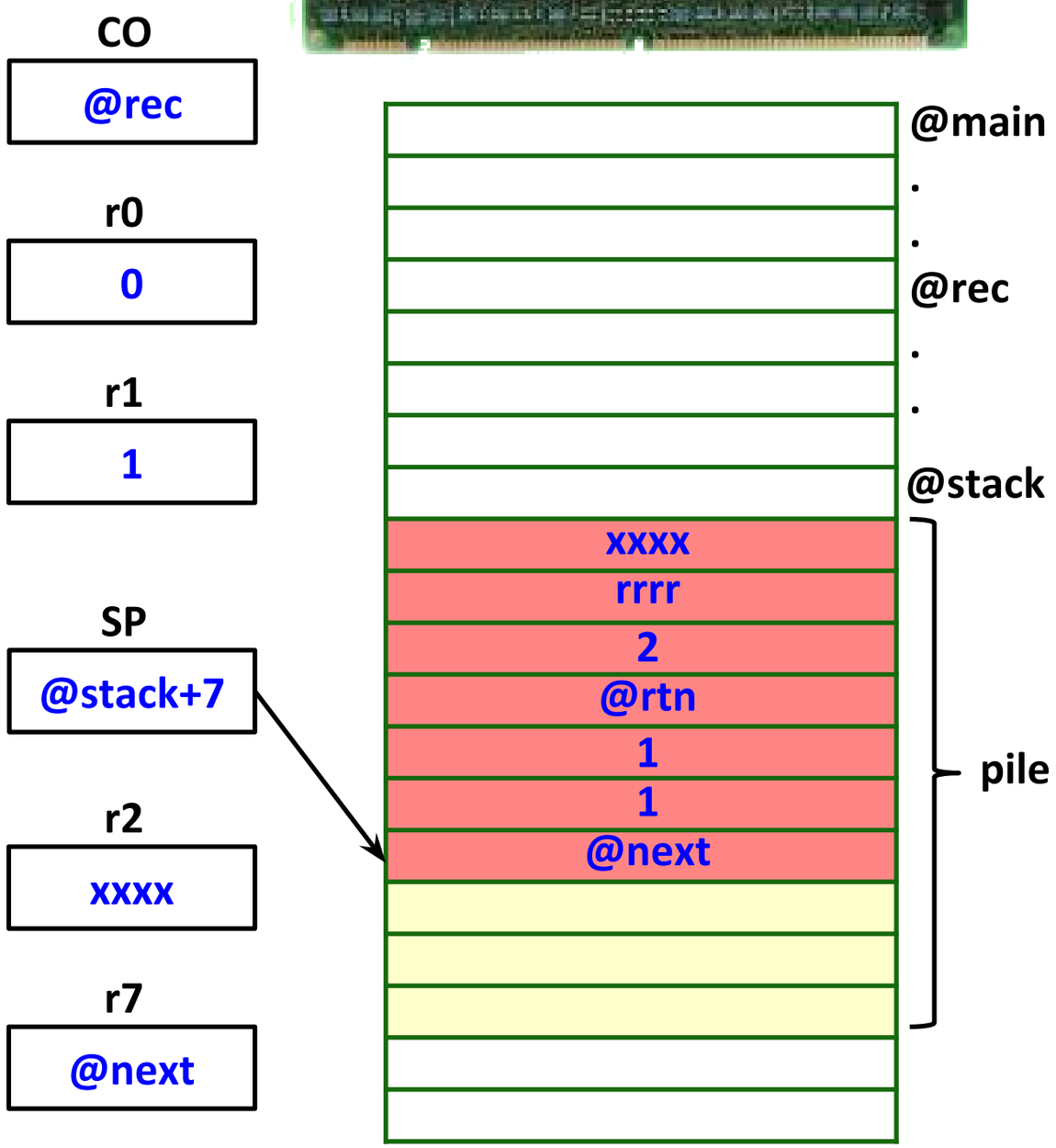




Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile

@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```



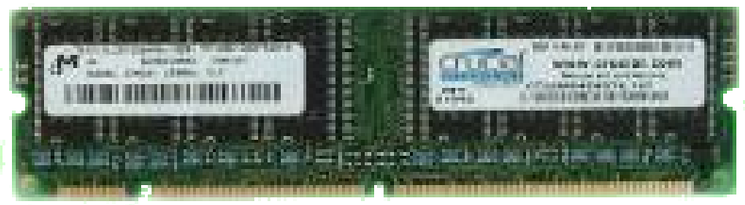


Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1
```

pile

```
@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```



CO
@rec+1

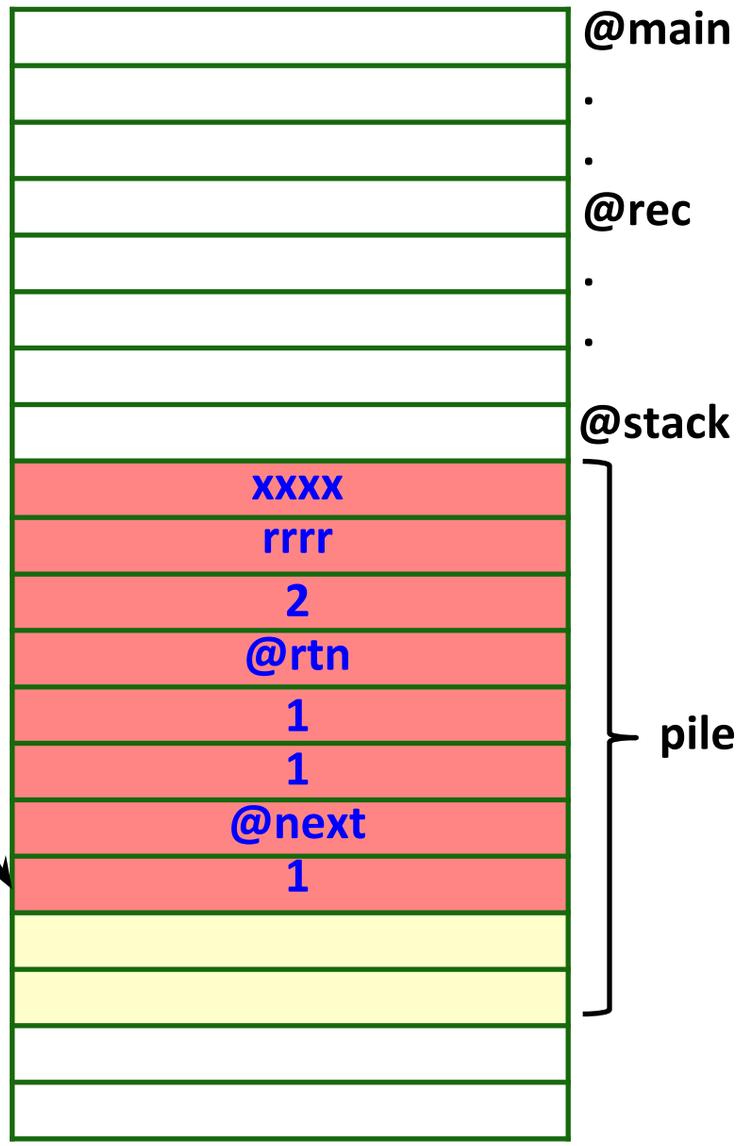
r0
0

r1
1

SP
@stack+8

r2
xxxx

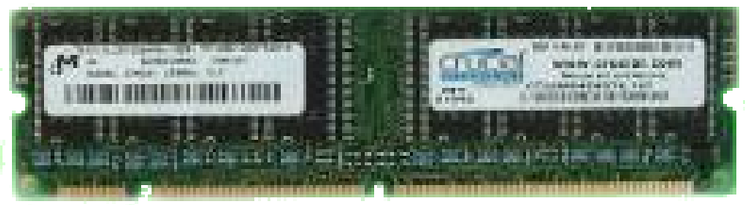
r7
@next

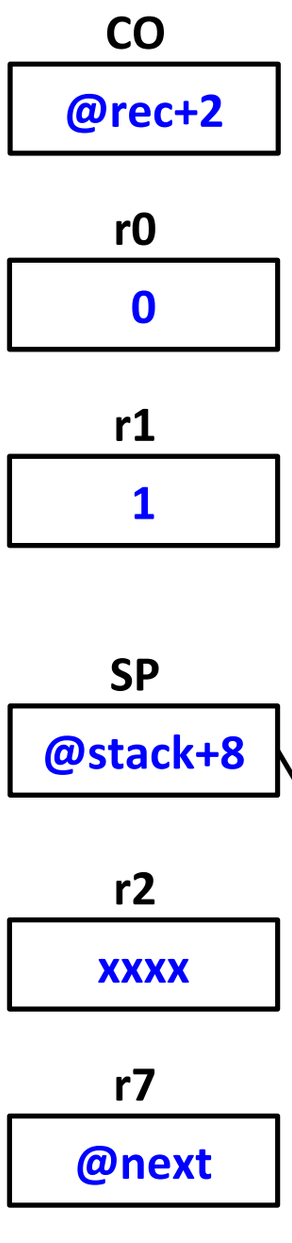


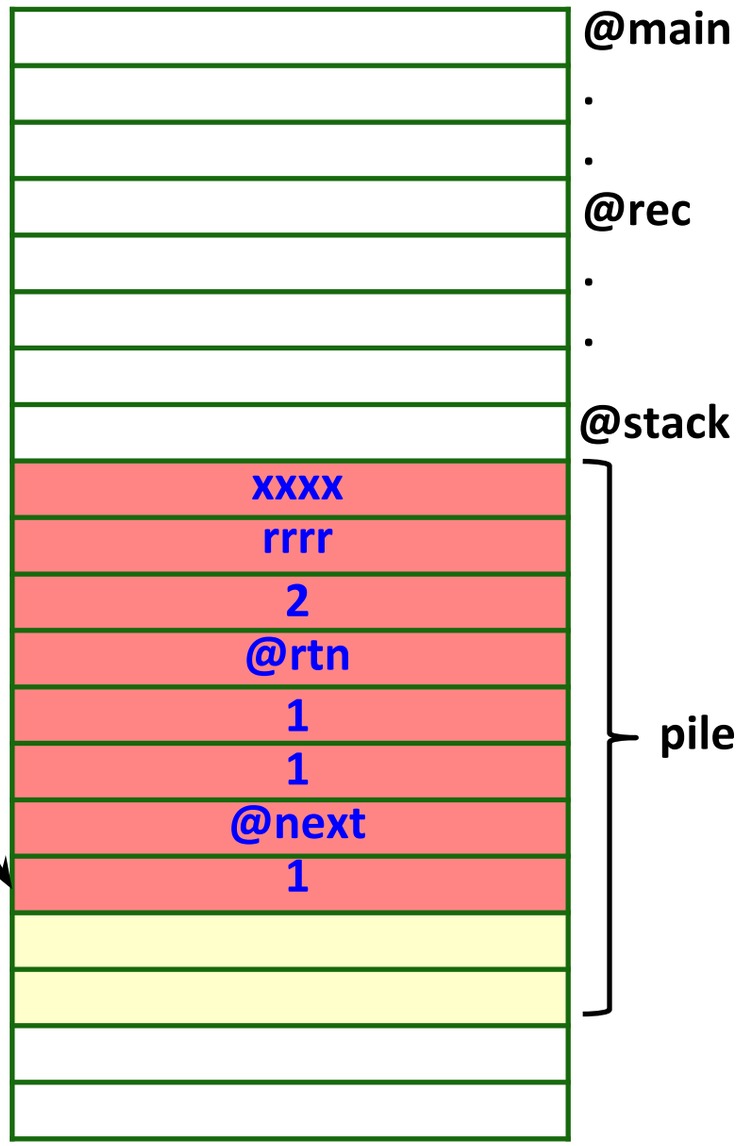
Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile
```

```
@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```



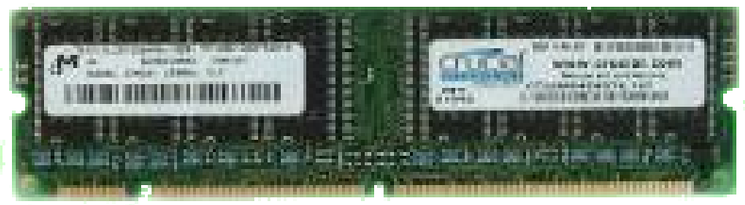


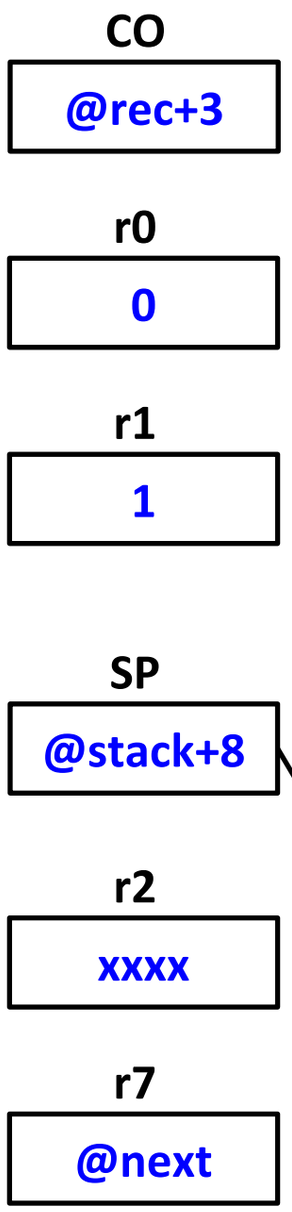


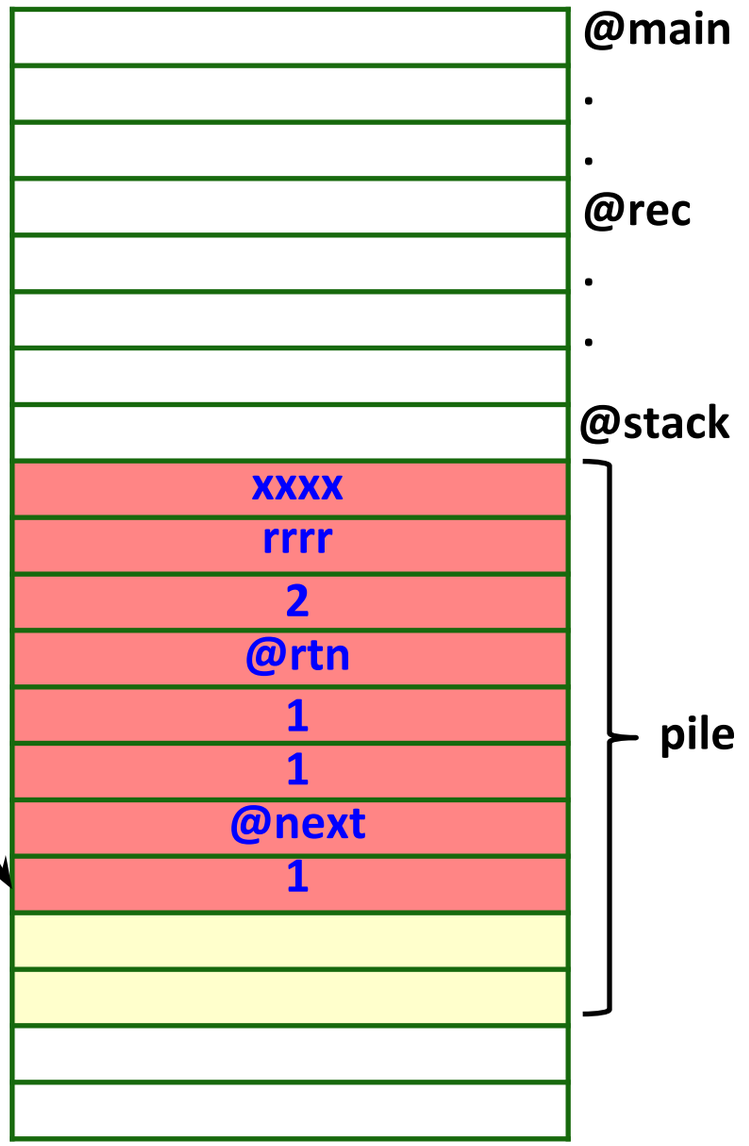
Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile
```

```
@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```





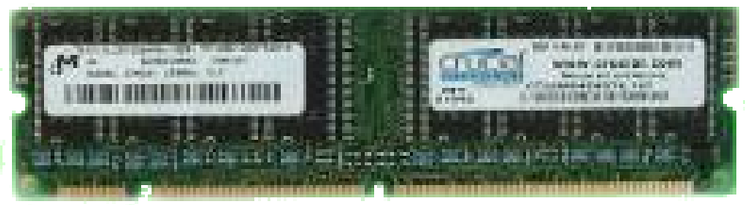


Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1
```

pile

```
@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```



CO
@end

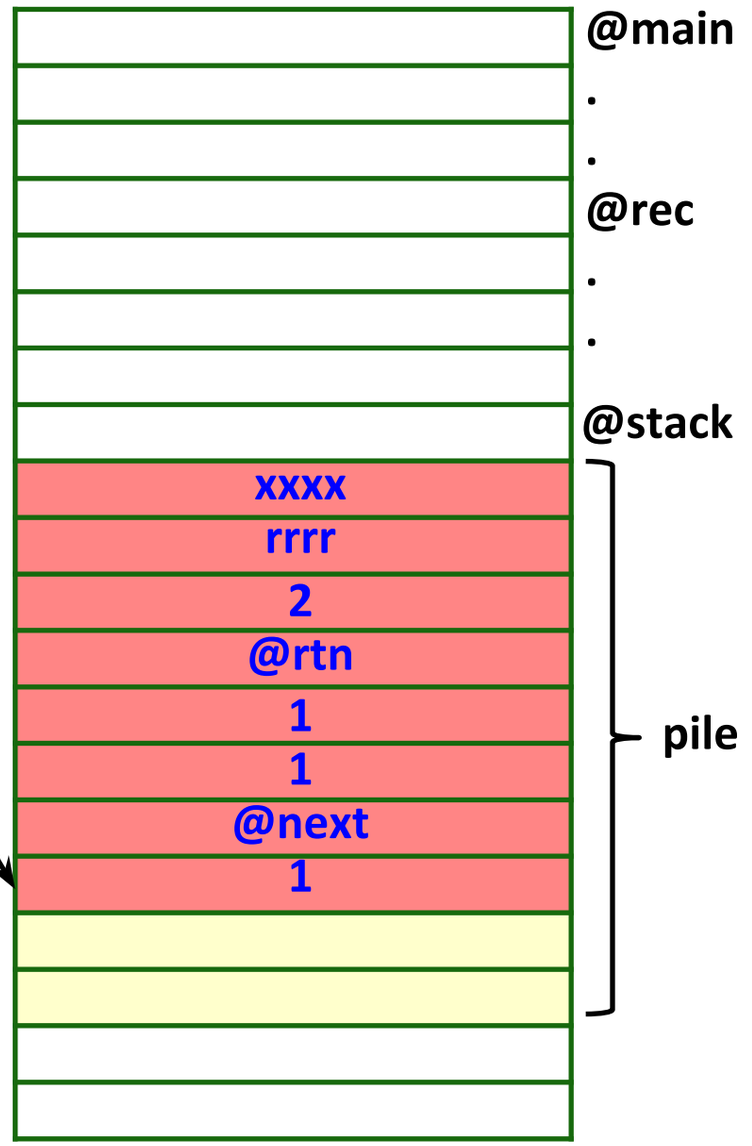
r0
0

r1
1

SP
@stack+8

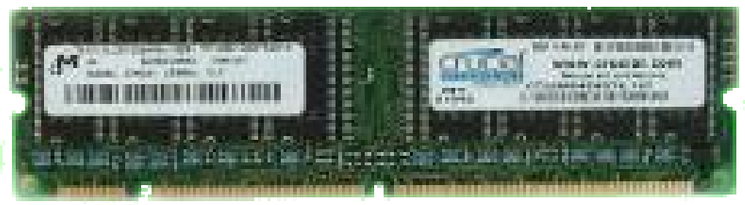
r2
xxxx

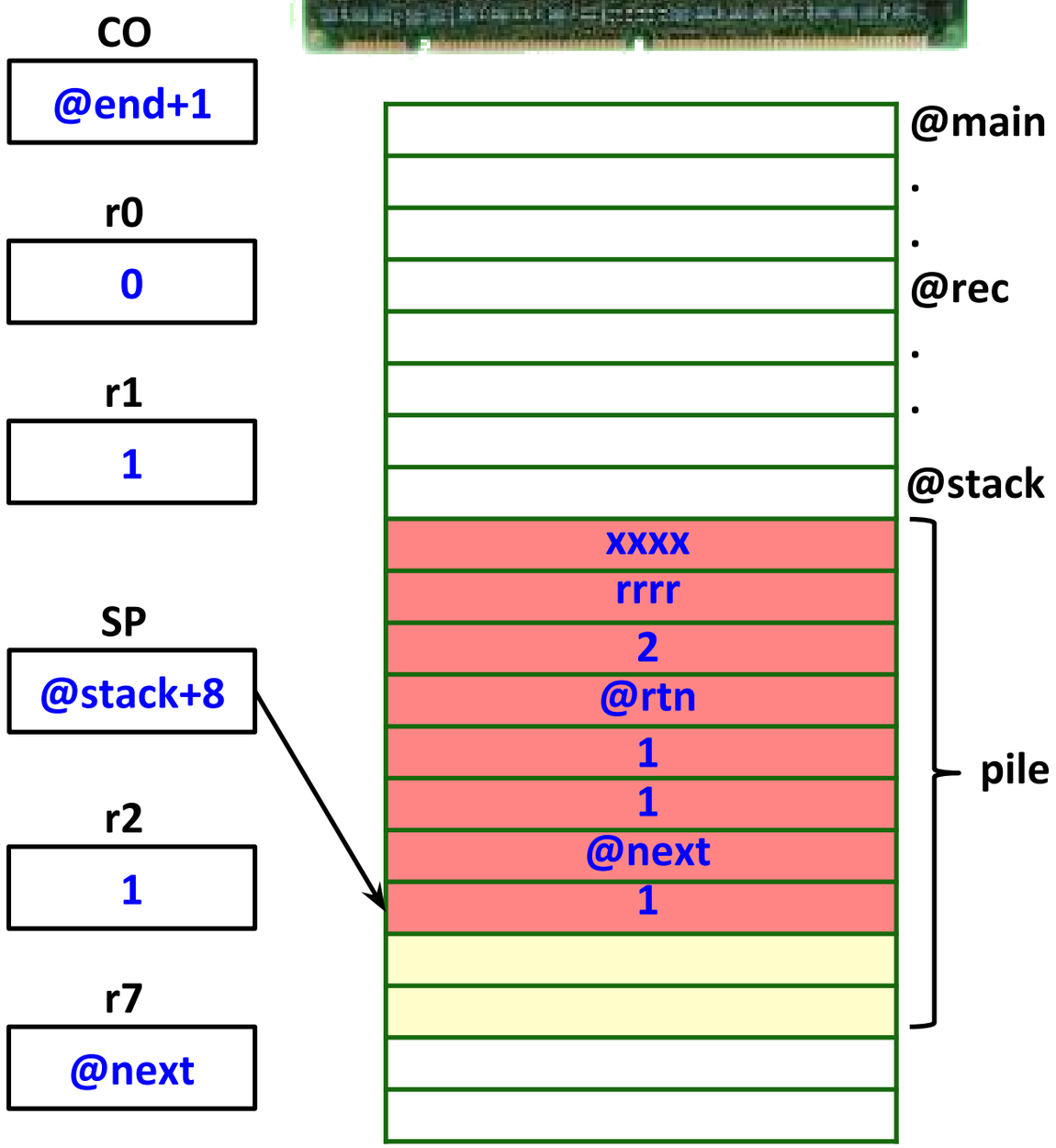
r7
@next



Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile

@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```



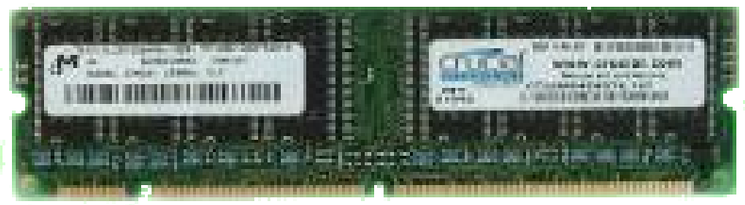


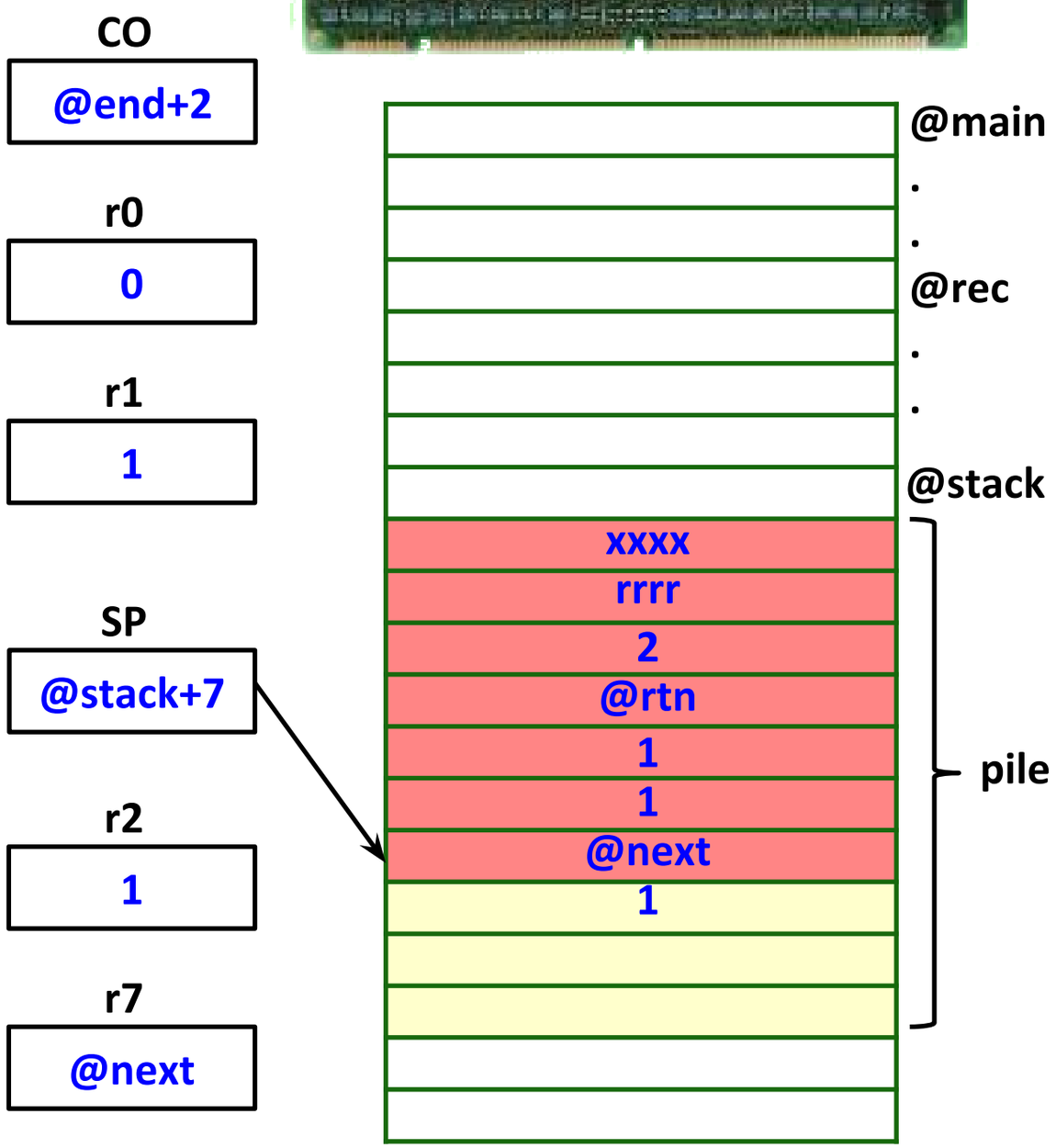
Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1
```

pile

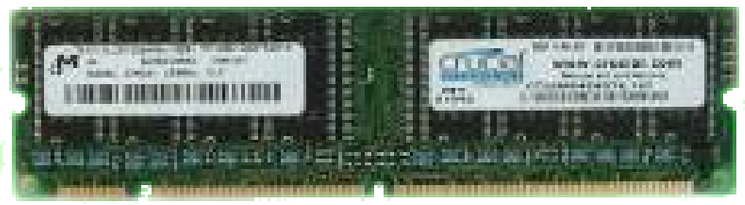
```
@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```





Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile

@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```



CO
@next

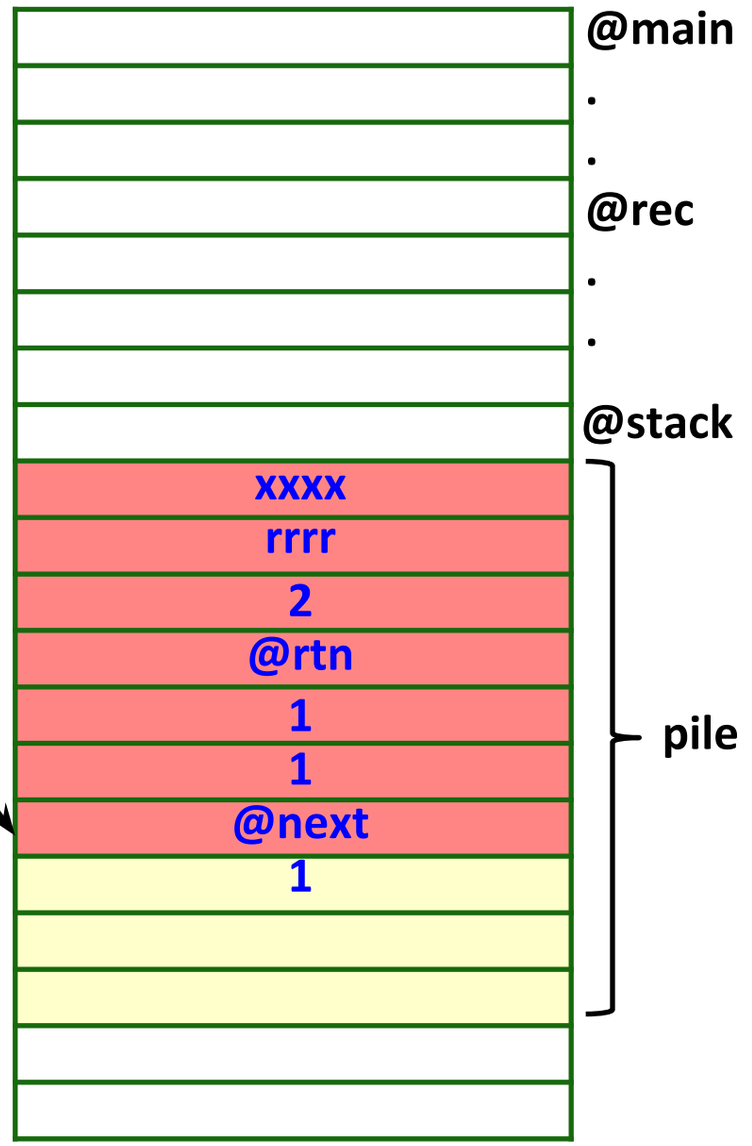
r0
0

r1
1

SP
@stack+7

r2
1

r7
@next

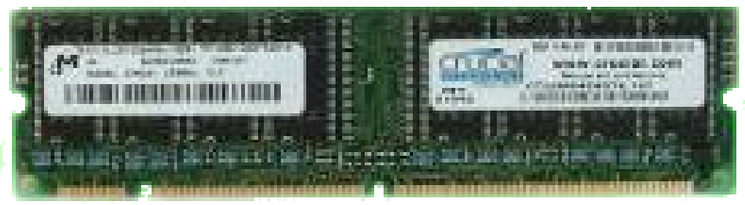


Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1
```

pile

```
@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```



CO: @next+1

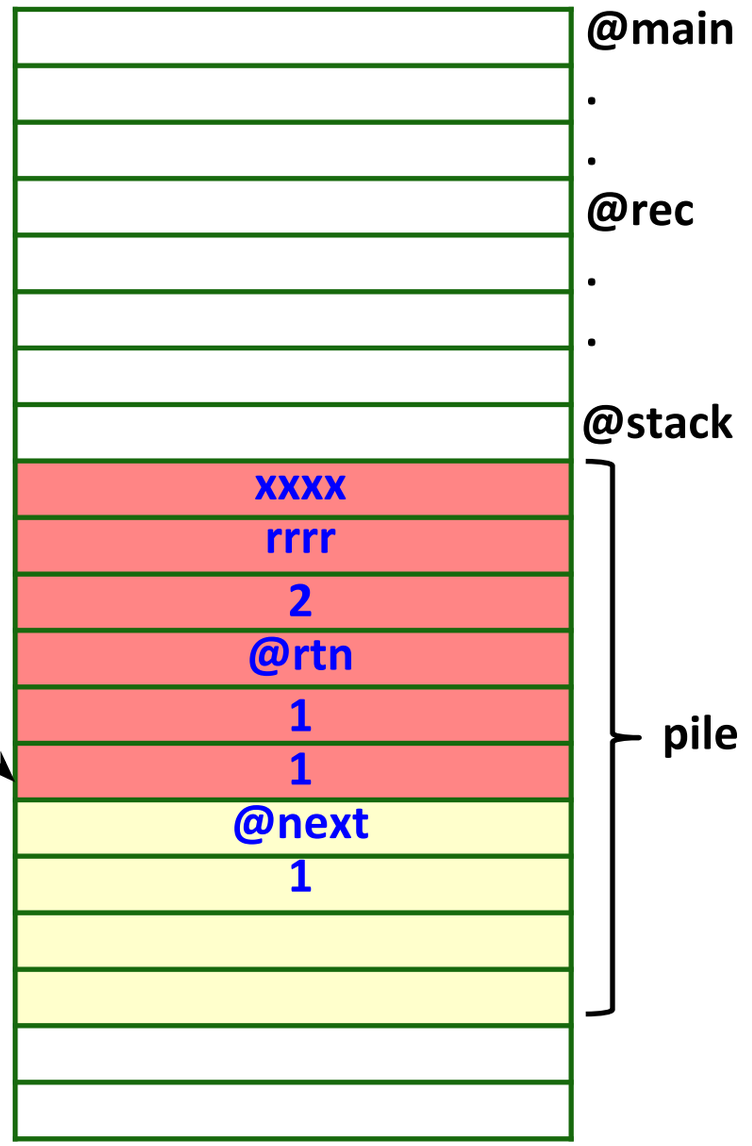
r0: 0

r1: 1

SP: @stack+6

r2: 1

r7: @next

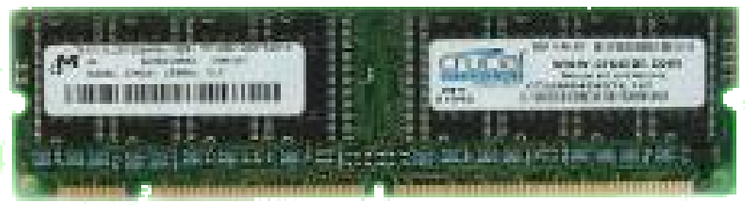


Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1
```

pile

```
@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```



CO: @next+2

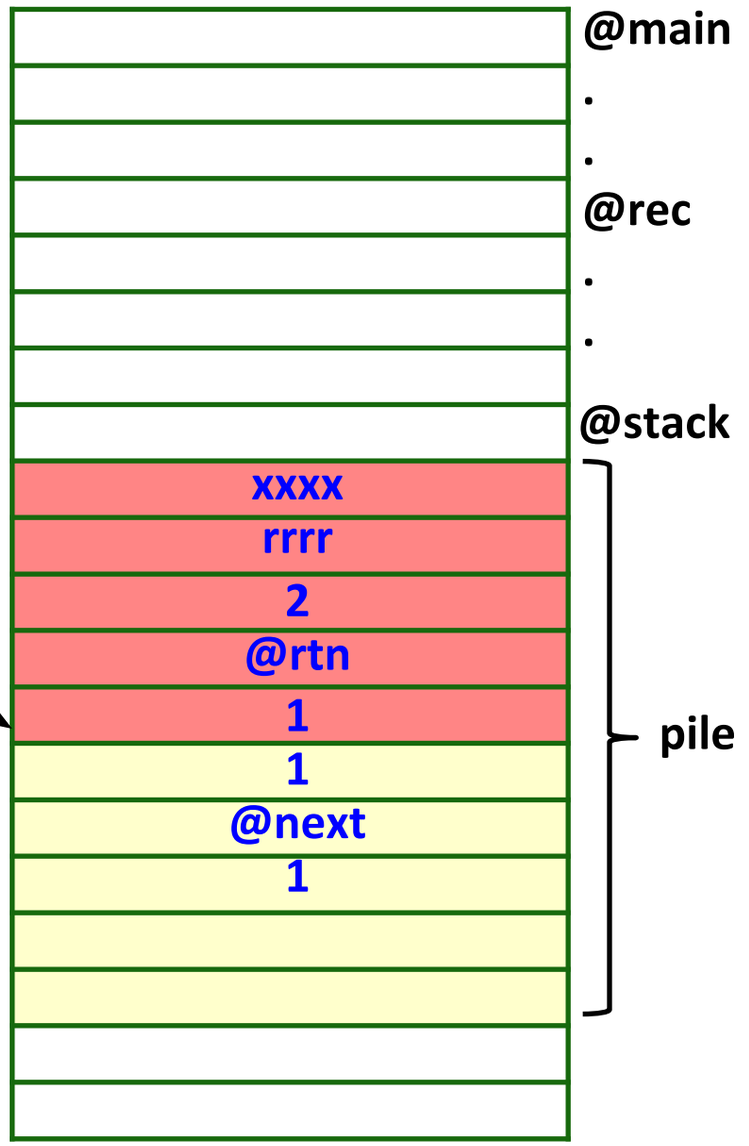
r0: 1

r1: 1

SP: @stack+5

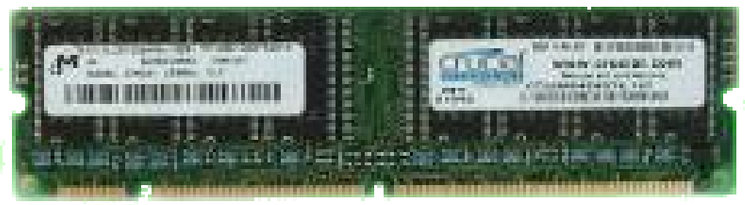
r2: 1

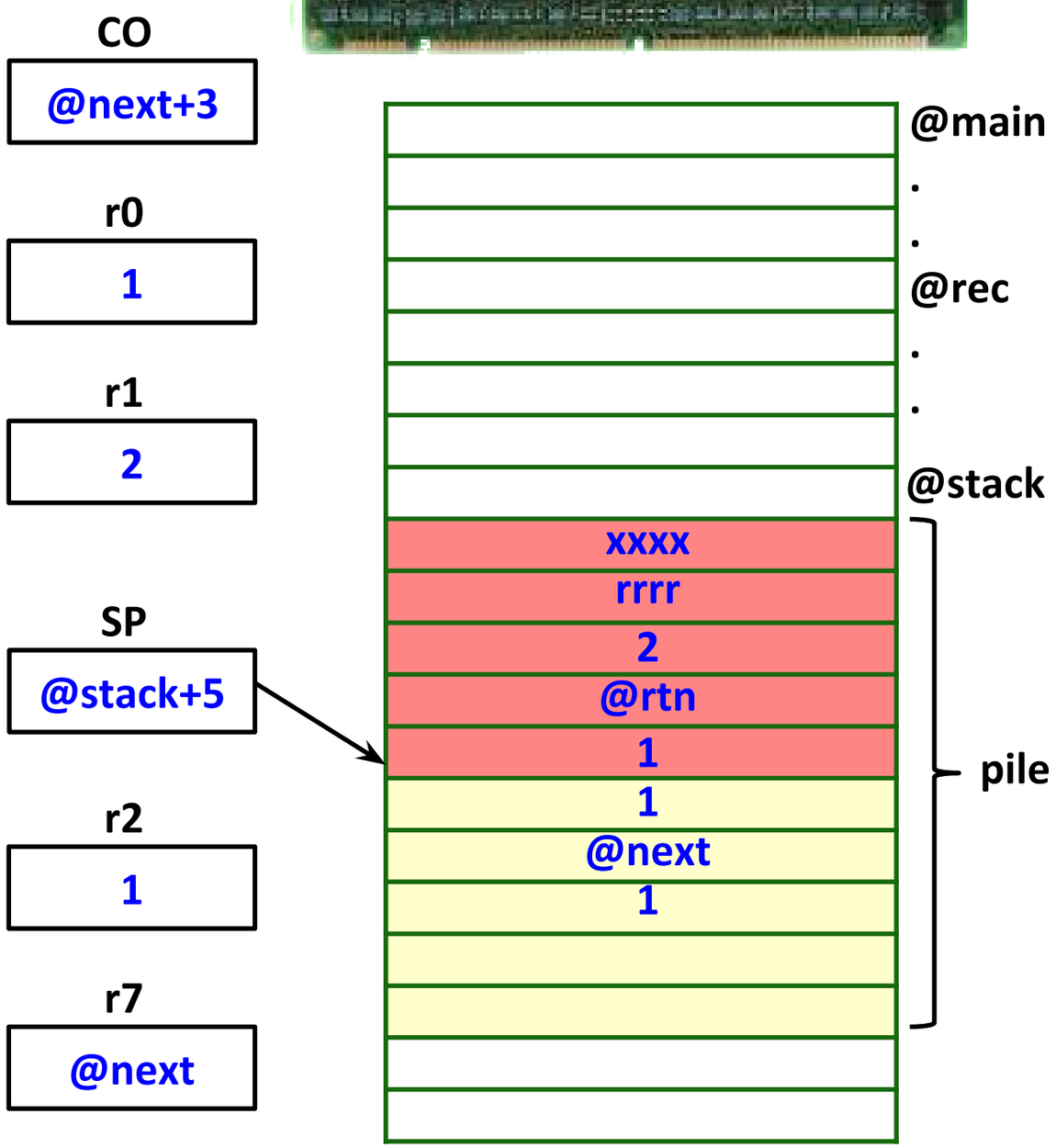
r7: @next



Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile

@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```

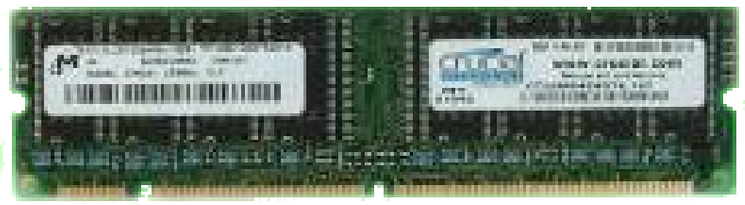


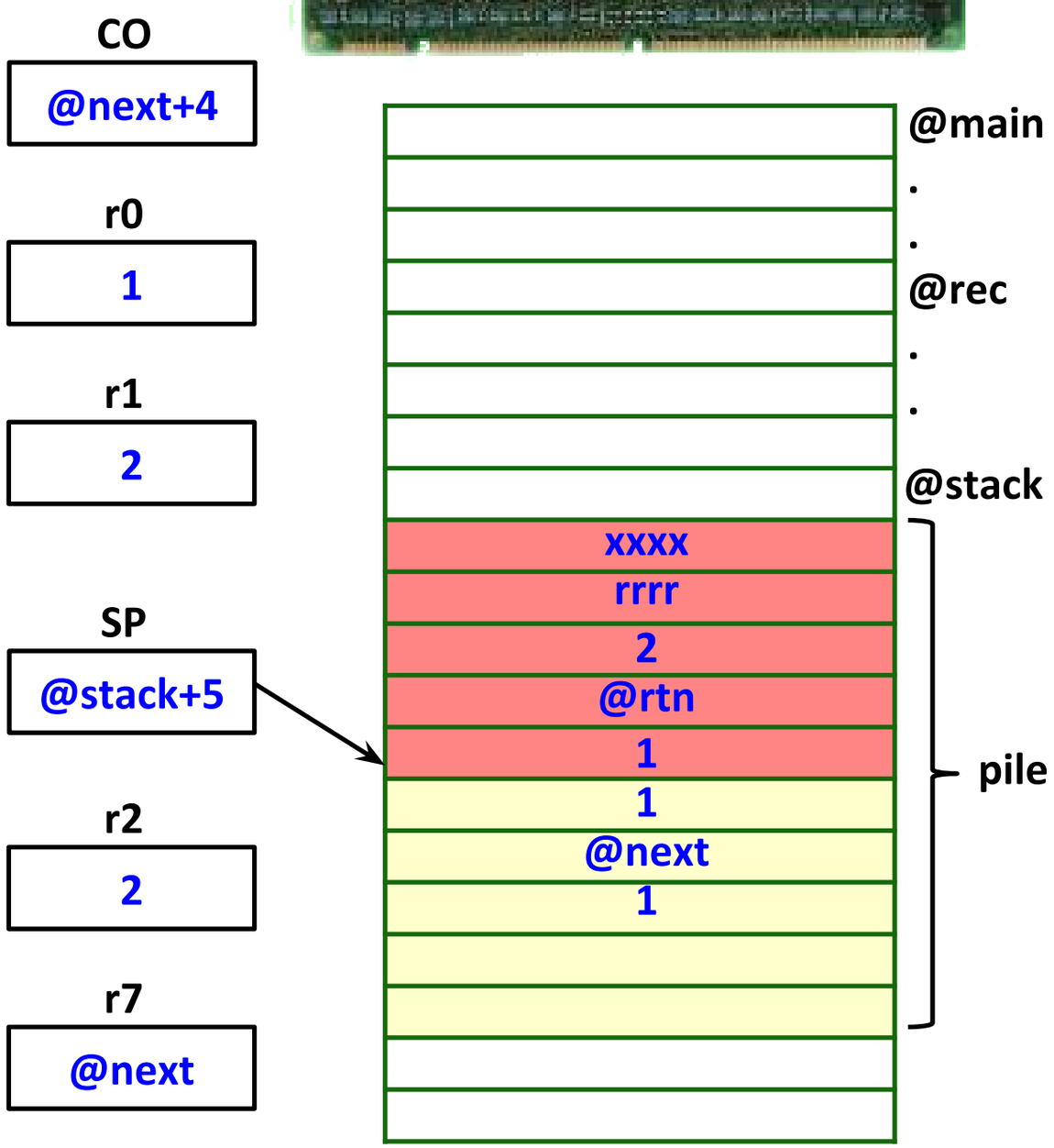


Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile
```

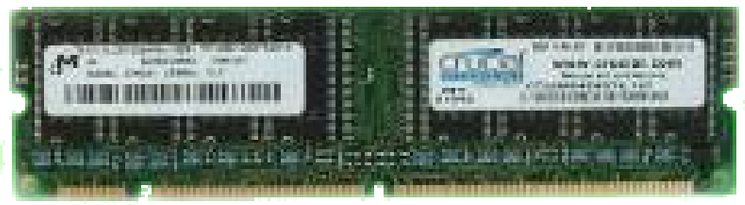
```
@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```





Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile

@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```



CO: @next+5

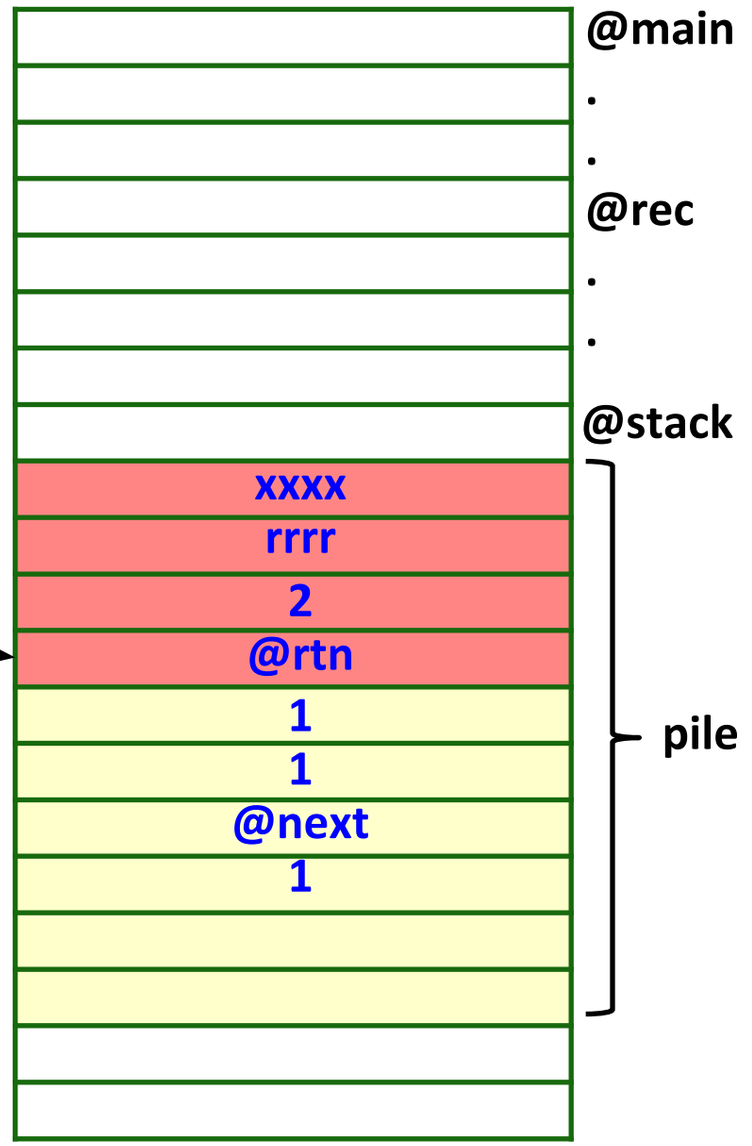
r0: 1

r1: 1

SP: @stack+4

r2: 2

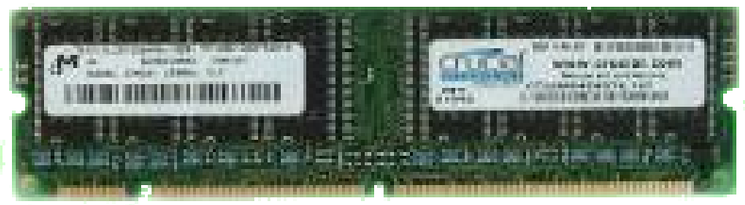
r7: @next

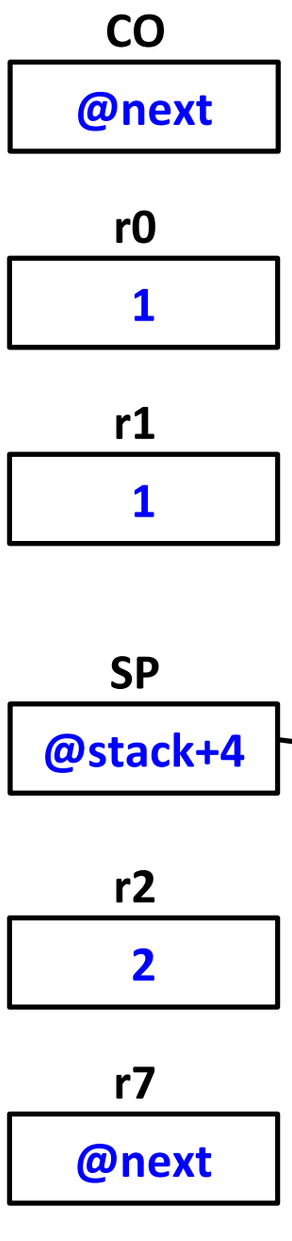


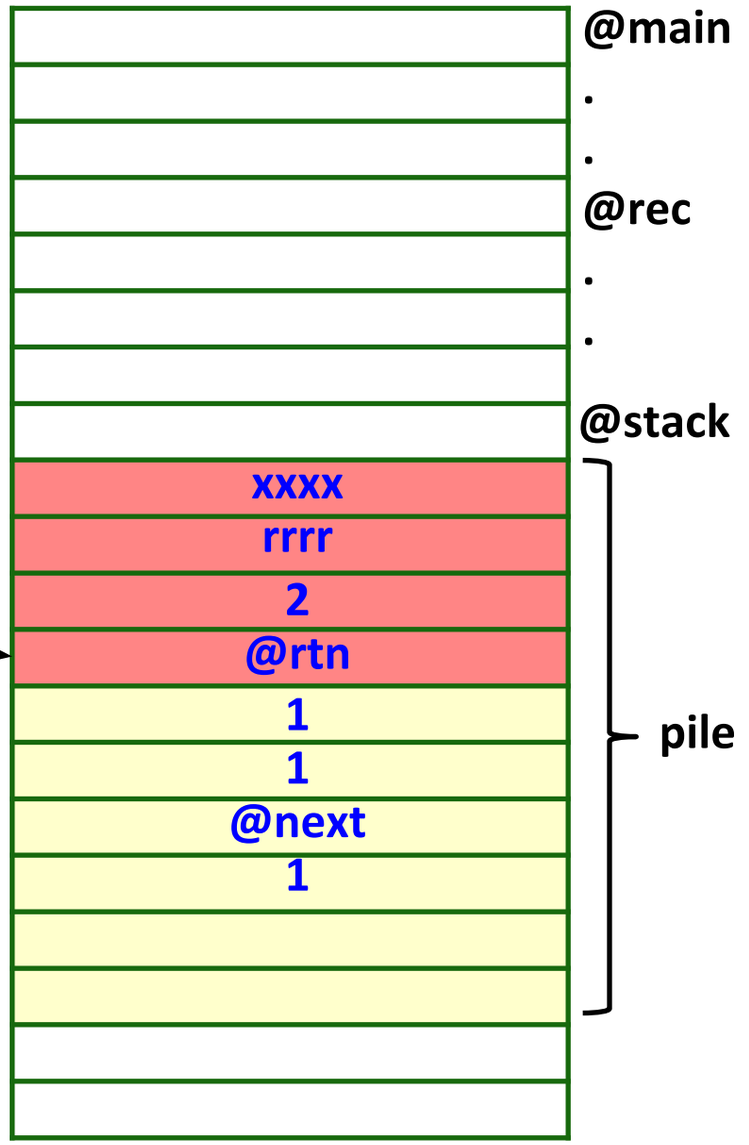
Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile
```

```
@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```

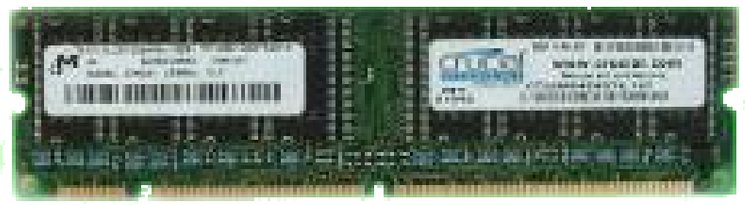


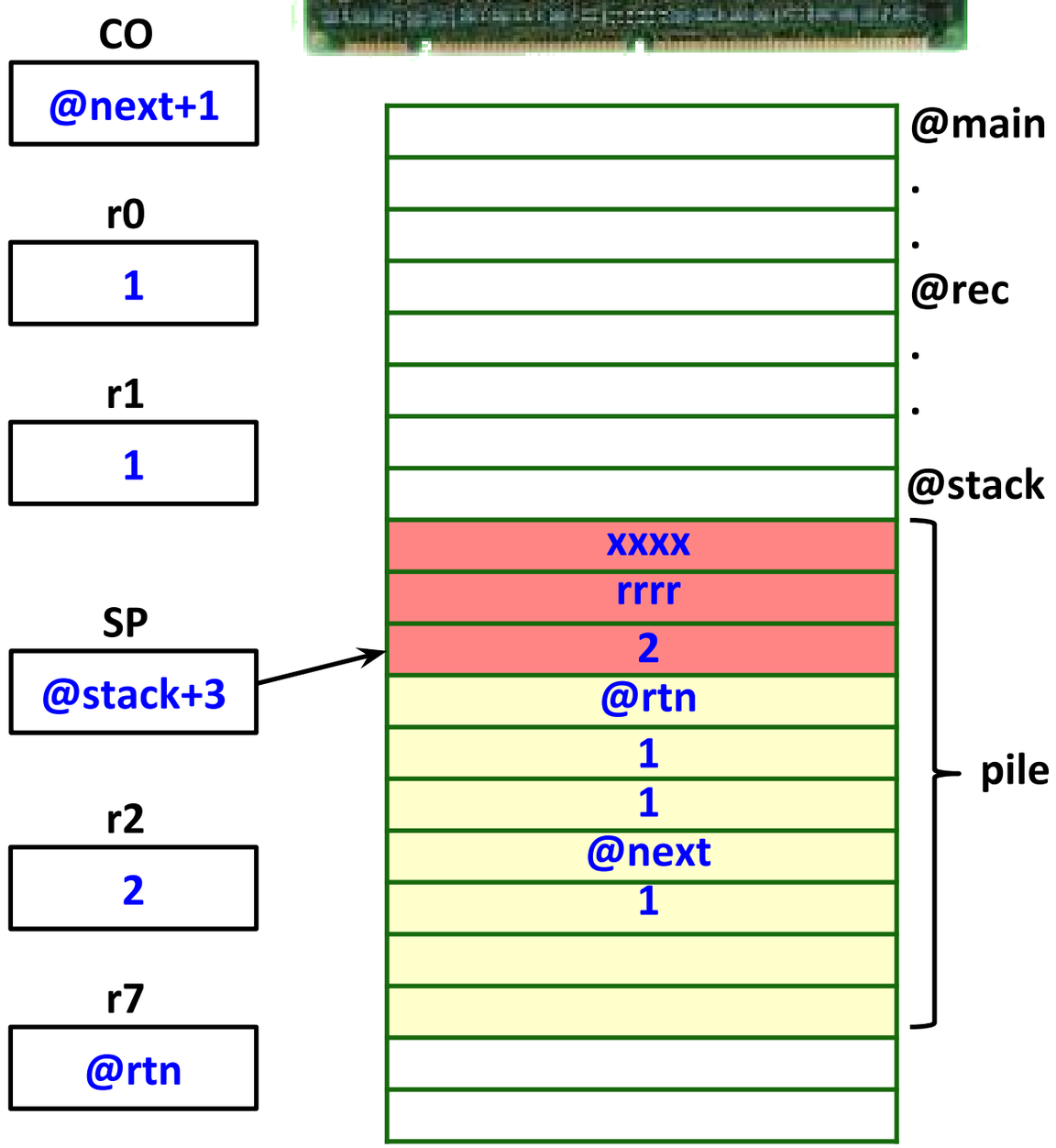




Fonction récursive

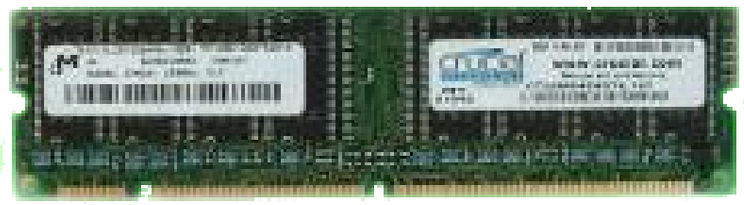
```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile

@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```

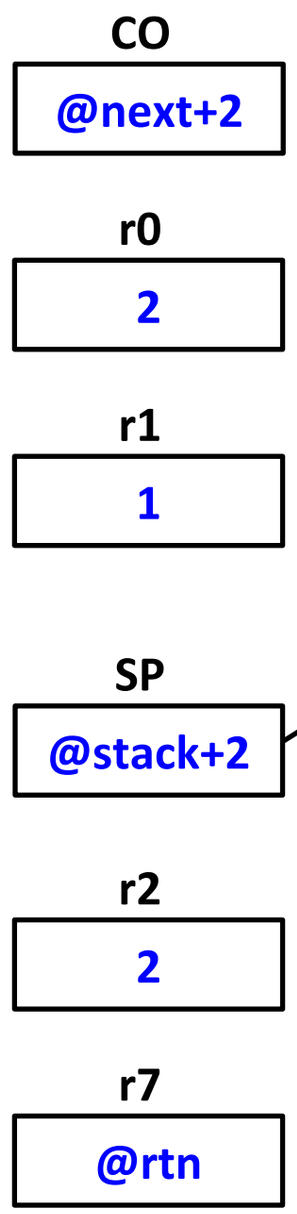


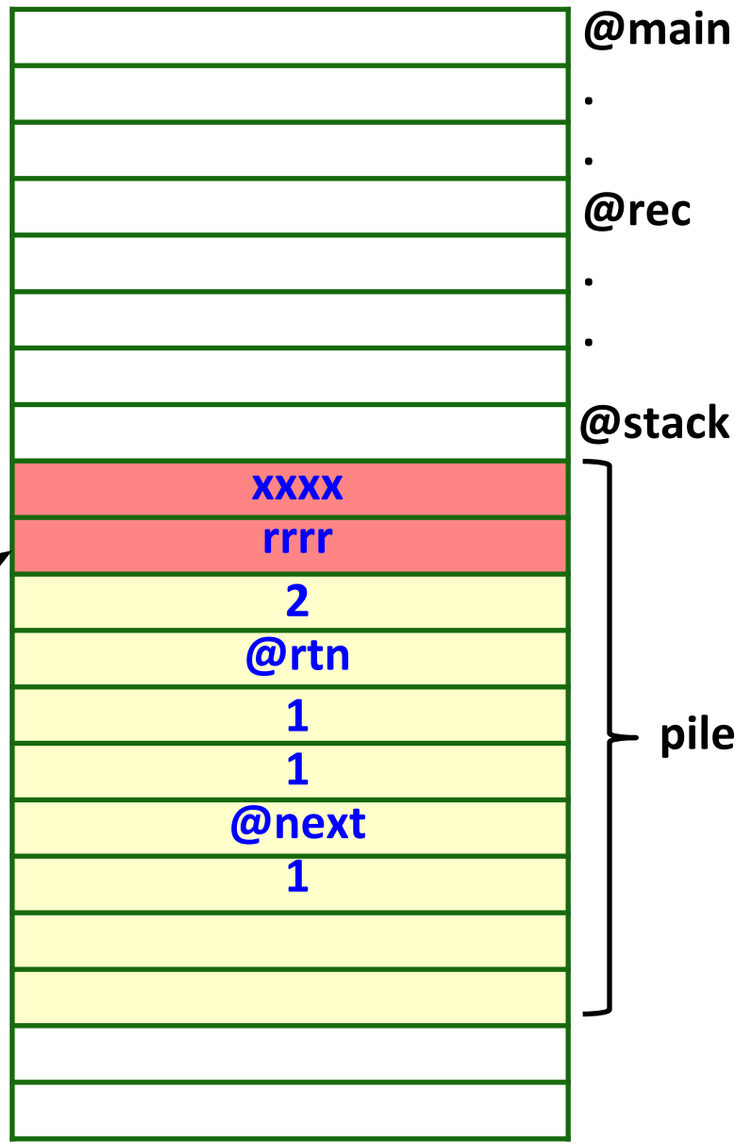


Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile

@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```



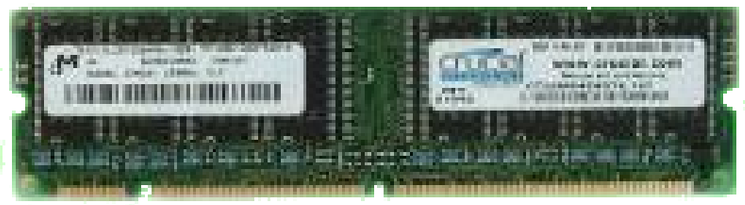


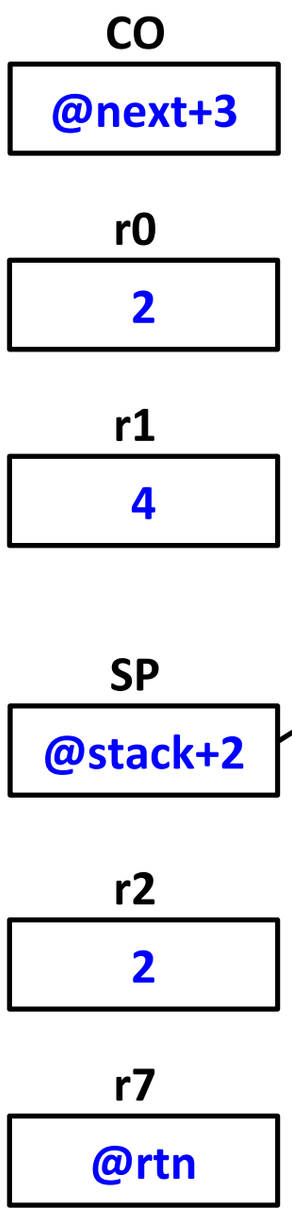


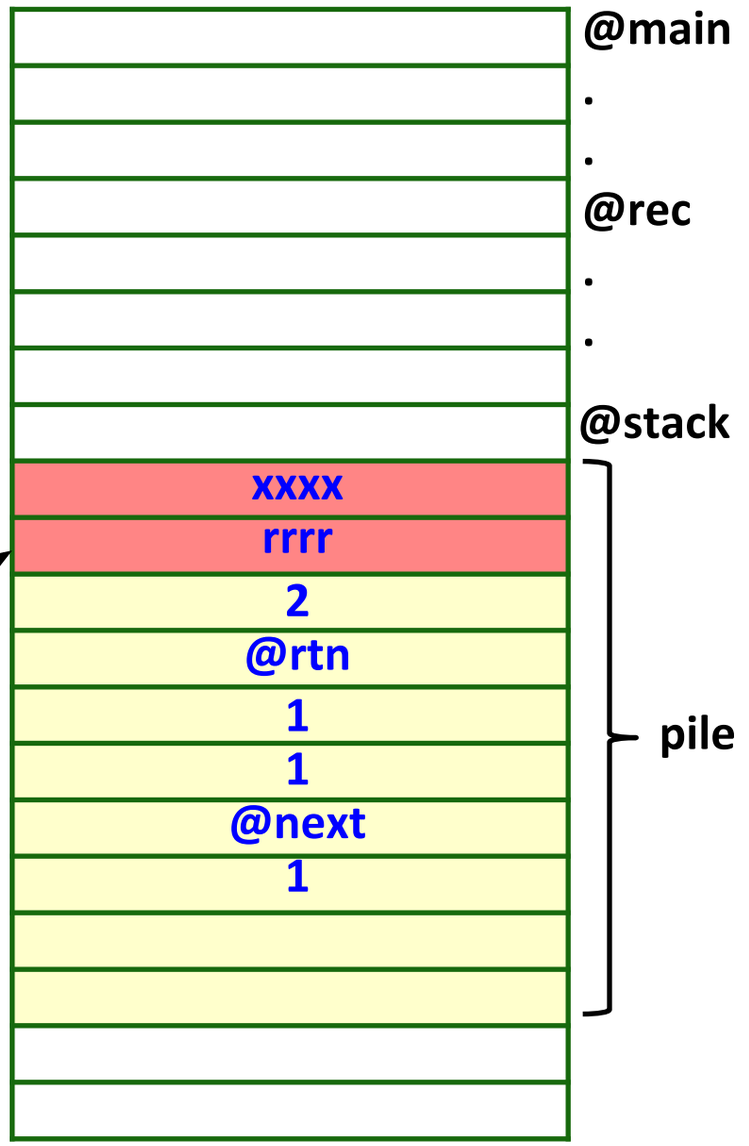
Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile
```

```
@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```



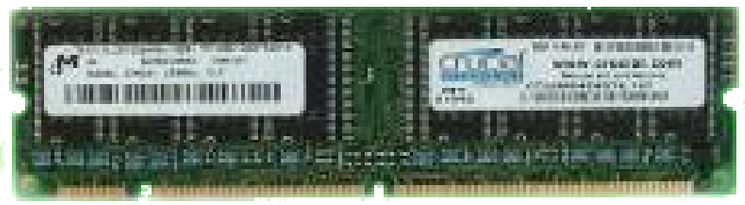


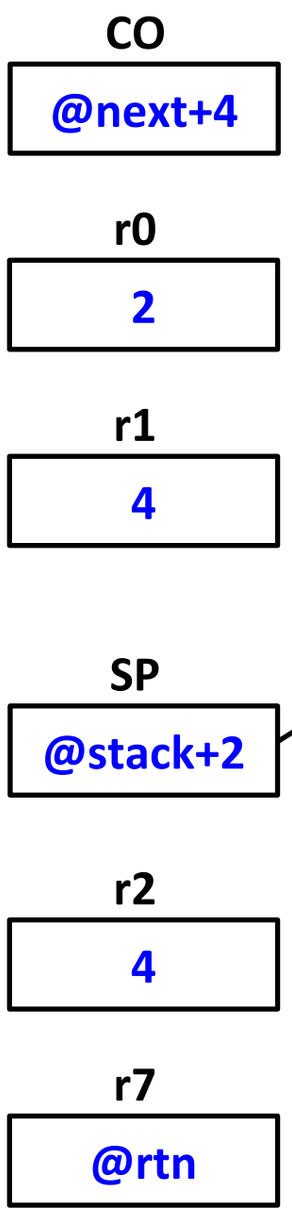


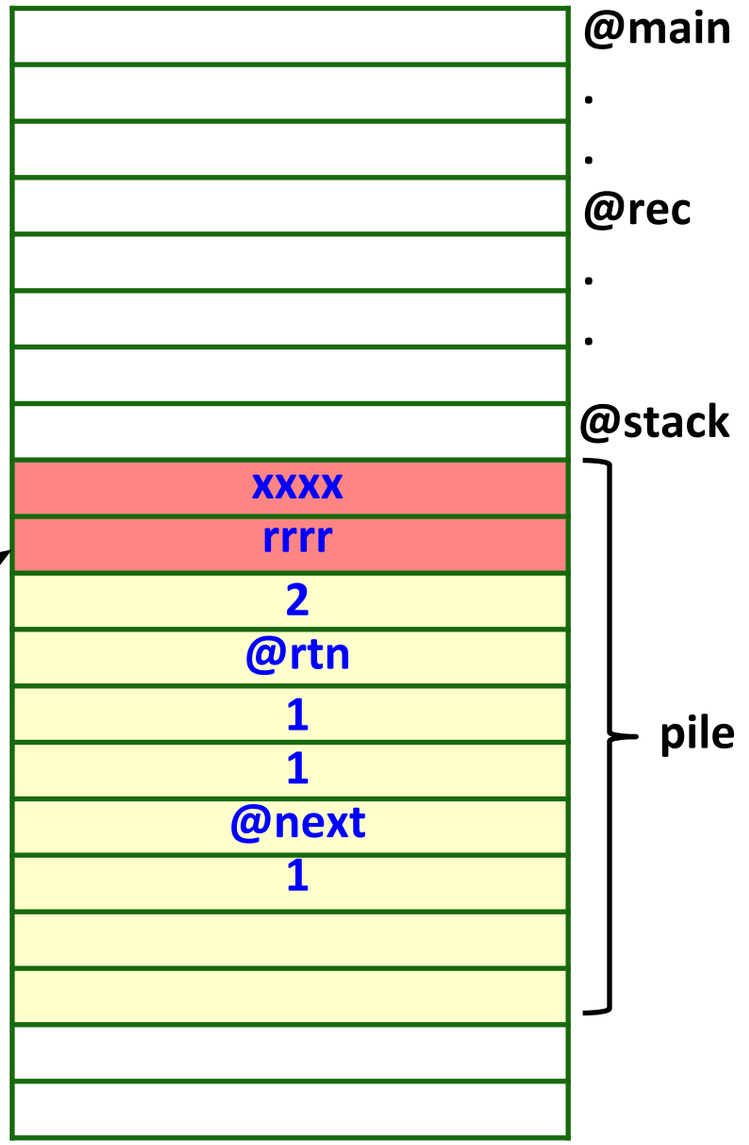
Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile
```

```
@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```

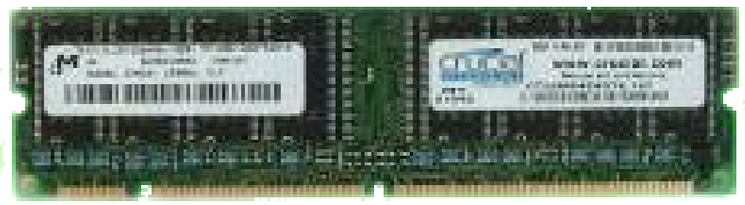






Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile

@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```



CO: @next+5

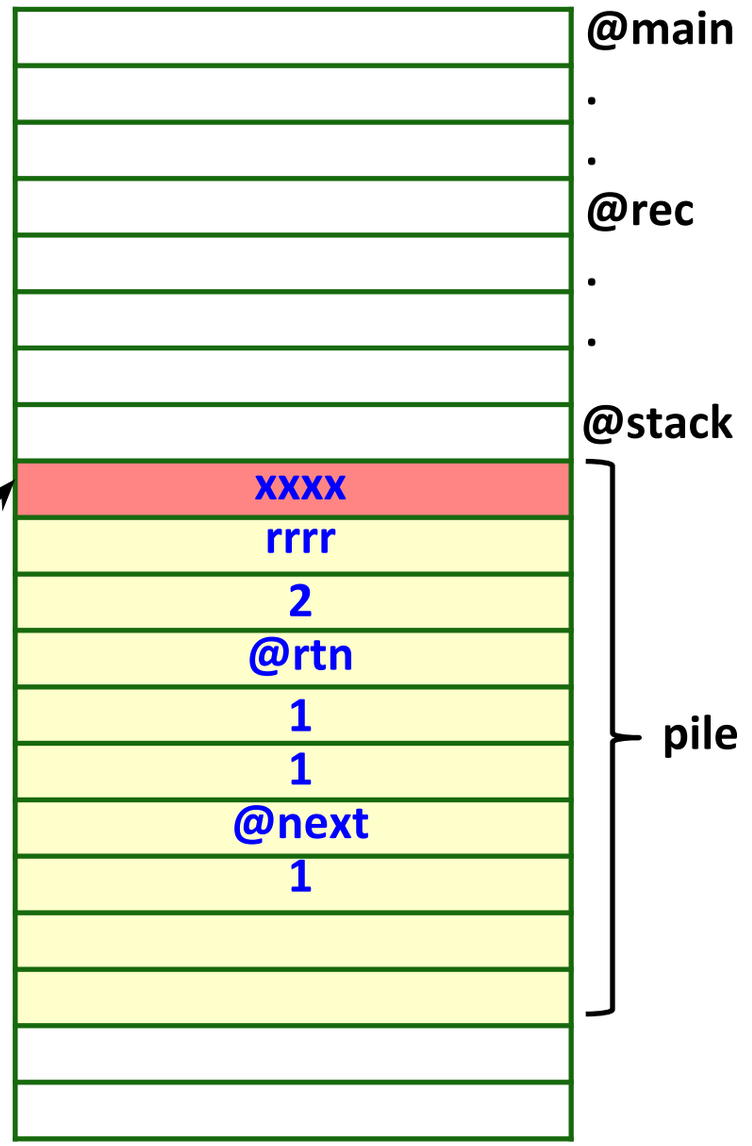
r0: 2

r1: rrrr

SP: @stack+1

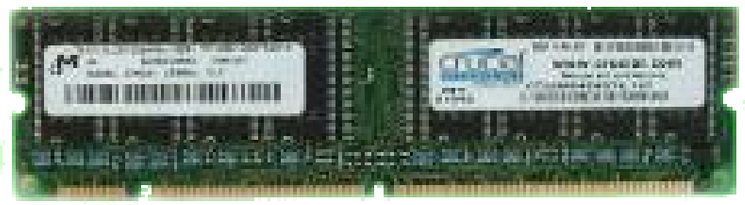
r2: 4

r7: @rtn



Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile

@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```



CO
@rtn

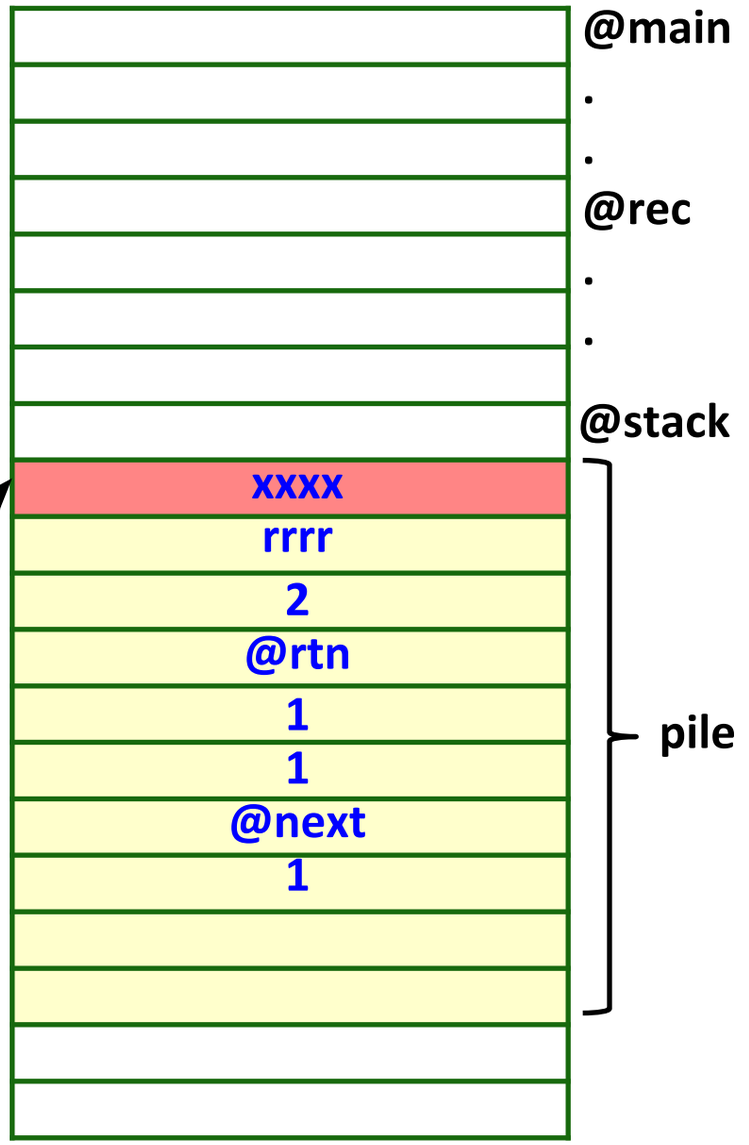
r0
2

r1
rrrr

SP
@stack+1

r2
4

r7
@rtn

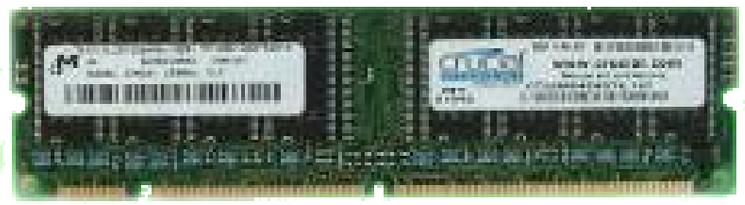


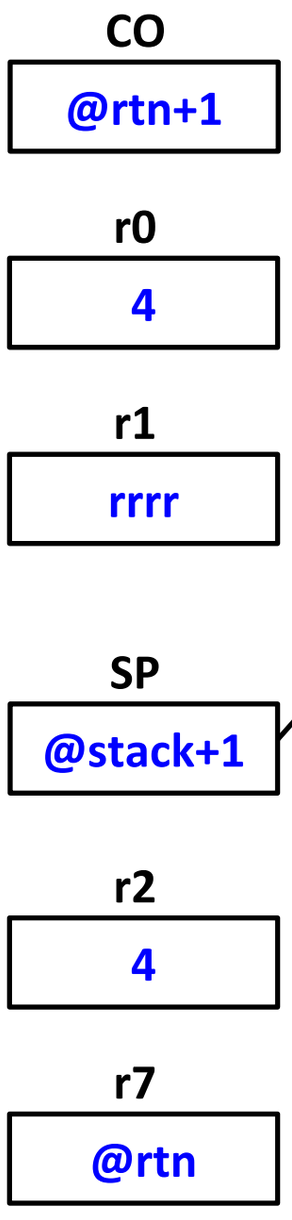
Fonction récursive

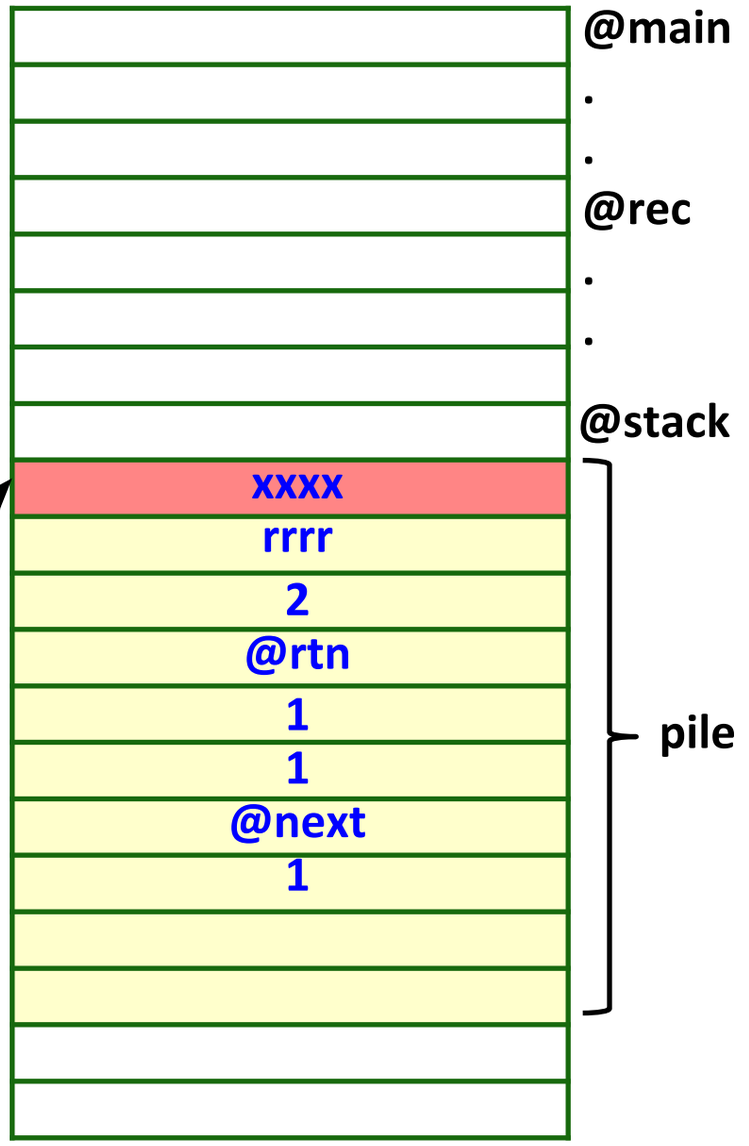
```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1
```

pile

```
@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```

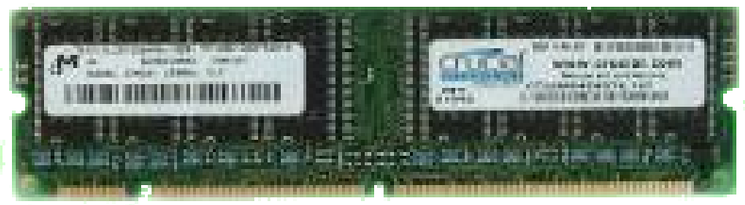


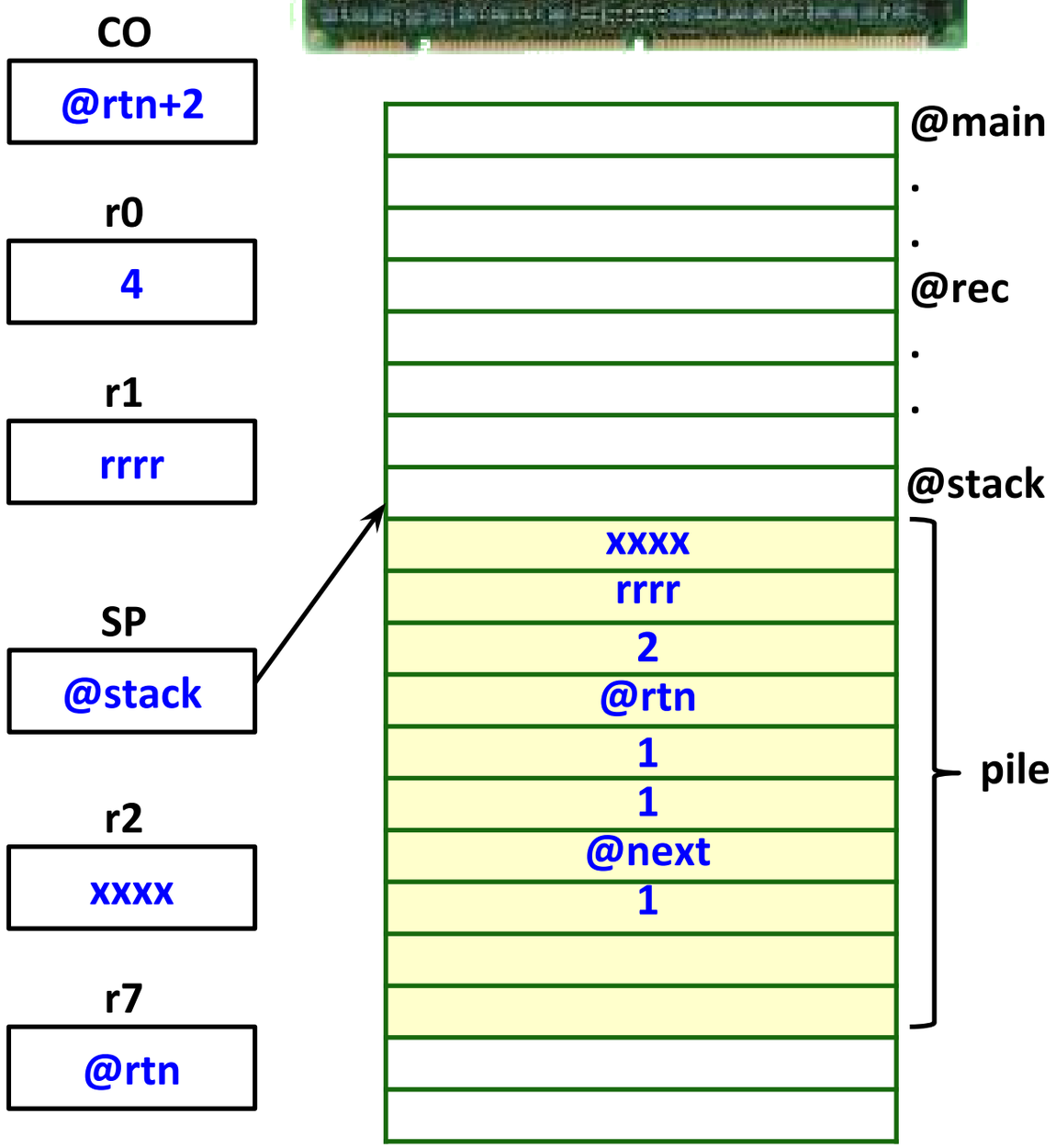




Fonction récursive

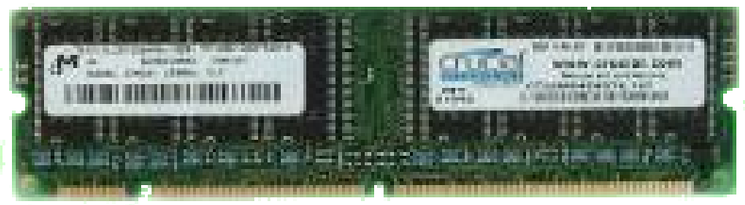
```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile

@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```

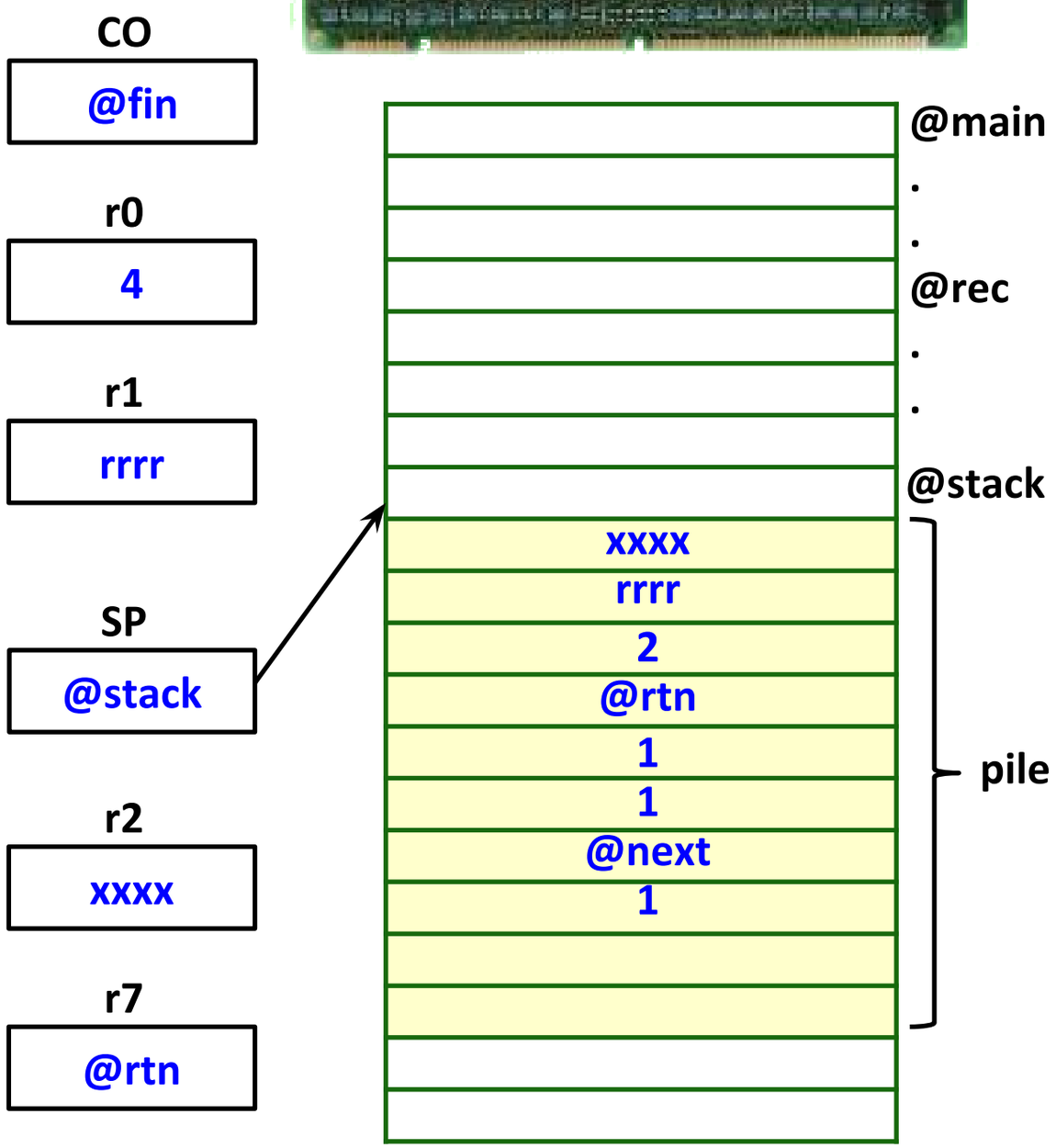




Fonction récursive

```
@main   mov sp, stack
        mov r0, #2
        push r2
        mov r7, rtn
        b    rec
@rtn    mov r0, r2
        pop r2
@fin    b    fin

@stack  rmw 1

        pile

@rec    push r1
        mov r1, #1
        cmp r0, #0
        beq  end
        push r0
        sub r0, r0, #1
        push r7
        mov r7, next
        b    rec
@next   pop r7
        pop r0
        add r1, r2, r0
@end    mov r2, r1
        pop r1
        b    r7
```





CentraleSupélec

université PARIS-SACLAY

LABORATOIRE DE RECHERCHE EN INFORMATIQUE

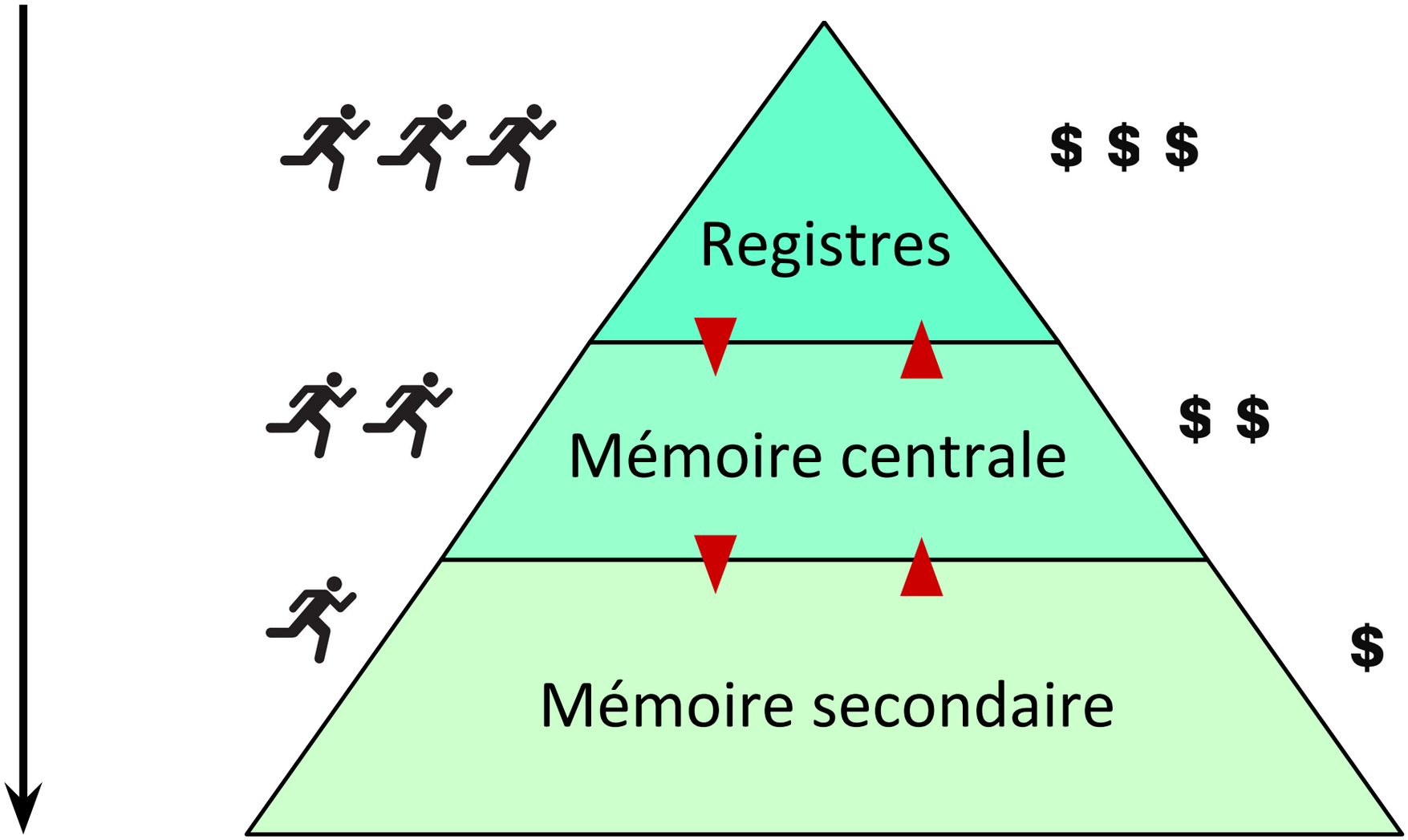
CHAPITRE V
Mémoire

Architecture des ordinateurs

Hiérarchie mémoire

- rapidité décroissante
- taille croissante
- coût décroissant

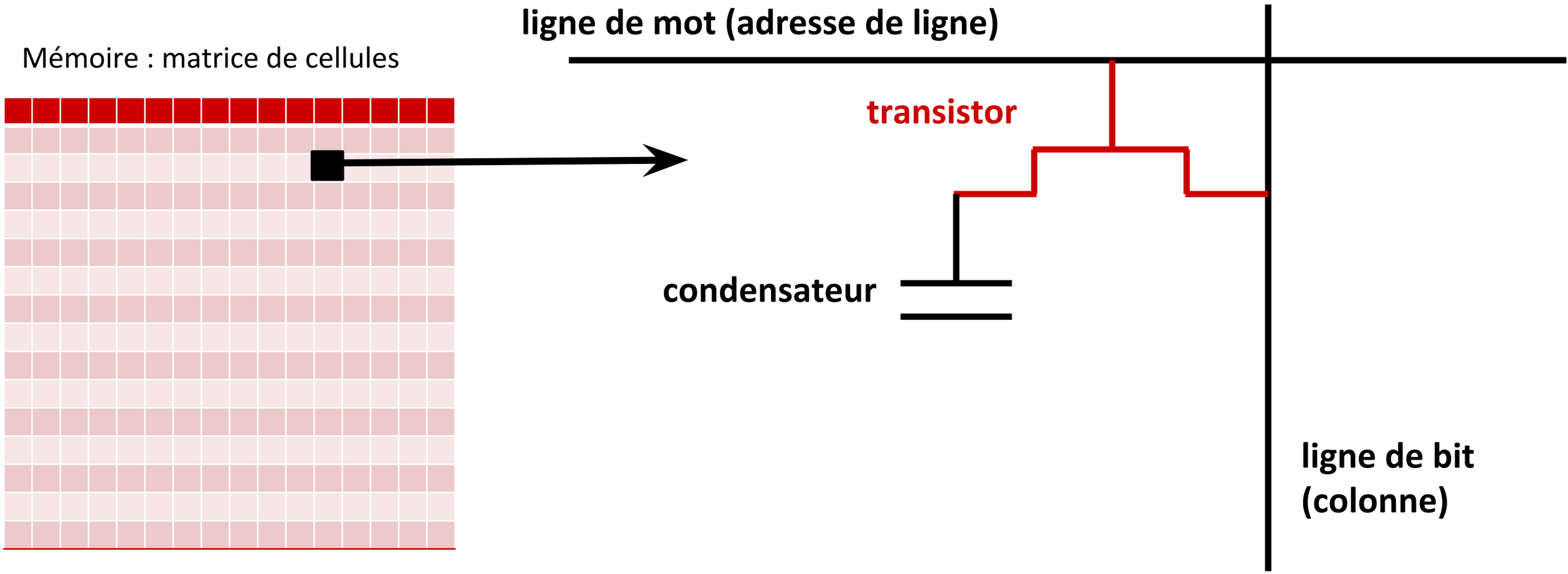
Processeur



La mémoire centrale

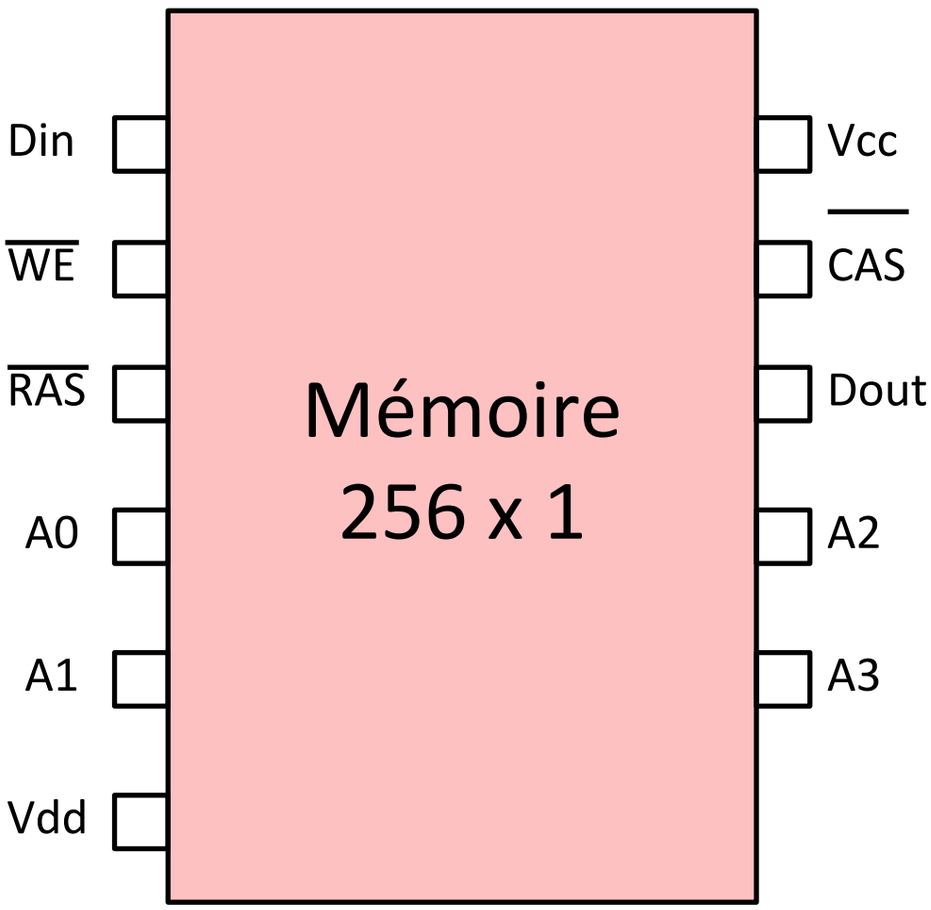
- La technologie utilisée pour réaliser la mémoire centrale est DRAM.
- DRAM = *Dynamic Random Access Memory*
  - Mémoire à accès direct (ou, aléatoire) dynamique.
- Mémoire à accès direct (RAM) : on accède directement à n'importe quel mot de mémoire par son adresse.
  - pour accéder au mot de mémoire à l'adresse x, il ne faut pas lire les mots de mémoire aux adresses 0,…., x-1.
- Mémoire dynamique : le contenu de la mémoire doit être lu et réécrit périodiquement (sinon il est perdu).
  - lire et réécrire la mémoire : rafraîchissement de la mémoire.
- La mémoire centrale est volatile.
  - en absence d'alimentation, tout son contenu est perdu!

DRAM – Cellule de mémoire (1ère génération)



- Mémoire : matrice de **cellules** (la matrice n'est pas nécessairement carrée).
- Chaque cellule de mémoire contient 1 bit.
- Le bit est mémorisé sous forme de charge électrique dans le condensateur.
  - condensateur chargé : bit 1, condensateur déchargé : bit 0.
- La charge dans un condensateur peut être gardée pendant plusieurs ms.
  - rafraîchissement de la mémoire : recharger les condensateurs périodiquement

DRAM – Puce de mémoire (lère génération)



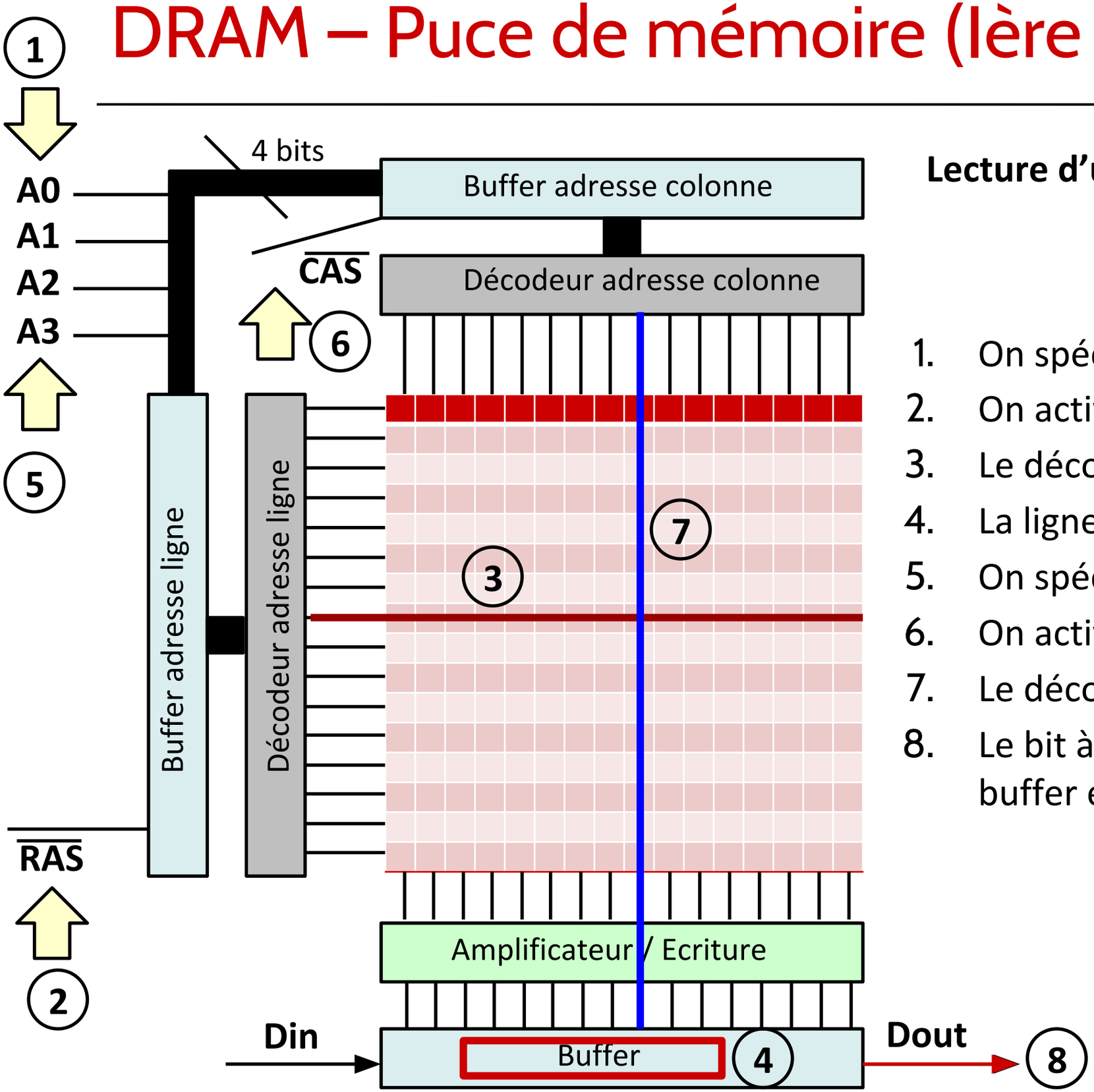
(mémoire déjà avancée car les adresses sont multiplexées.)

- La taille d'un mot de mémoire est 1 bit.

Exemple.

- Mémoire 256 x 1 : 256 cellules (*cells*) contenant 1 bit.
- Les cellules sont organisées dans une matrice 16x16.
- Pour accéder au contenu d'une cellule (1 bit), nous avons besoin de deux adresses.
  - adresse de la ligne de la matrice (4 bits).
  - adresse de la colonne de la matrice (4 bits).
- Les broches A0, A1, A2, A3 sont utilisées pour spécifier une adresse.
  - si ~~RAS~~ = 0 (*Row Access Strobe*), l'adresse sur ces broches est interprété comme adresse de ligne
  - si ~~CAS~~ = 0 (*Column Address Strobe*), l'adresse sur ces broches est interprété comme adresse de colonne.
- ~~WE~~ = 0 (*Write Enable*) : opération d'écriture.
  - ~~WE~~ = 1 : opération de lecture.
- Din : donnée en entrée (1 bit)
- Dout : donnée en sortie (1 bit)

DRAM – Puce de mémoire (1ère génération)



Lecture d'une cellule

1. On spécifie l'adresse de ligne.
2. On active le signal RAS.
3. Le décodeur sélectionne la ligne.
4. La ligne est envoyée au buffer de ligne.
5. On spécifie l'adresse de colonne.
6. On active le signal CAS.
7. Le décodeur sélectionne la colonne.
8. Le bit à la colonne sélectionné dans le buffer est envoyé sur la sortie.

DRAM – Puce de mémoire (Ière génération)

Puce de mémoire **Intel D2118-7.**

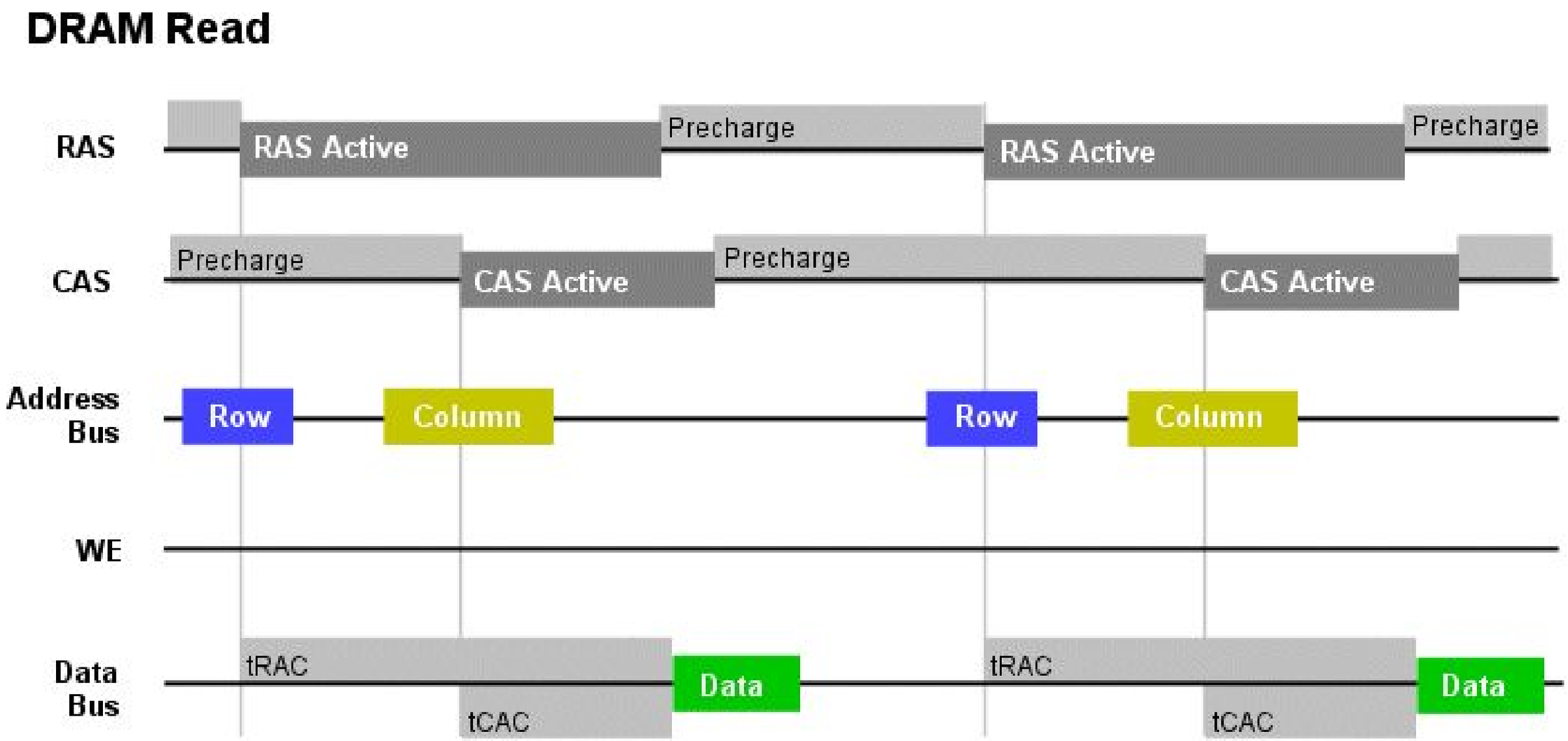
http://www.datasheets360.com/part/detail/d2118-7/5300631060238353796/

Attribut	Valeur
Status	Discontinued
Sub Category	DRAMs
Access mode	---
Access Time-max	150ns
Memory width	1
Number of words code	16K
Operating mode	---
Organization	16Kx1

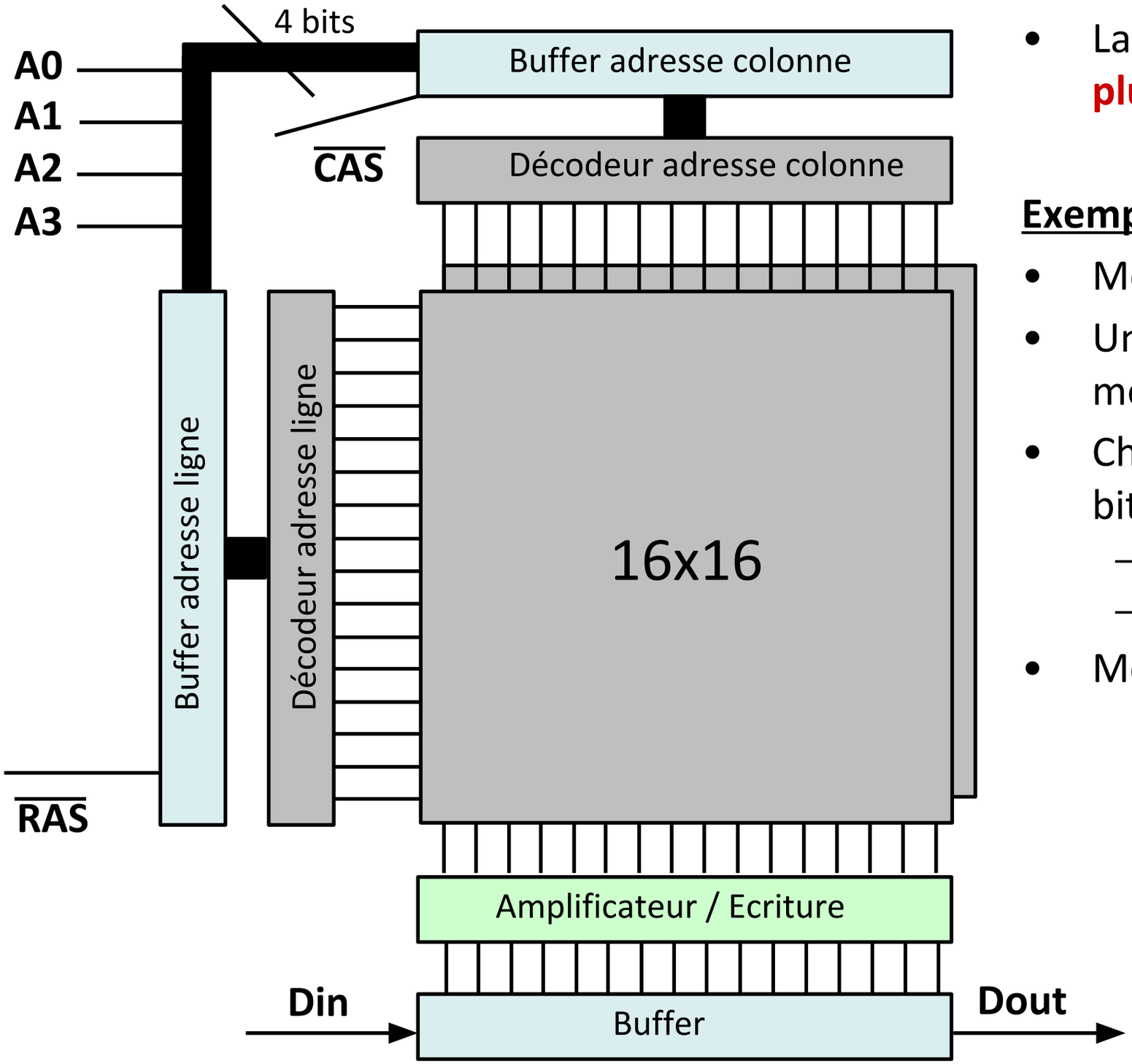
Taille de la mémoire : 16K bits = $2^4 * 2^{10}$ bits = $2^1 * 2^{10}$ bytes = **2 KB**

DRAM – Lecture (lère génération)

- Sortie d'une opération de lecture : un mot.



DRAM – Puce de mémoire (IIème génération)



- La mémoire consiste en **une banque** de **plusieurs matrices** de cellules.

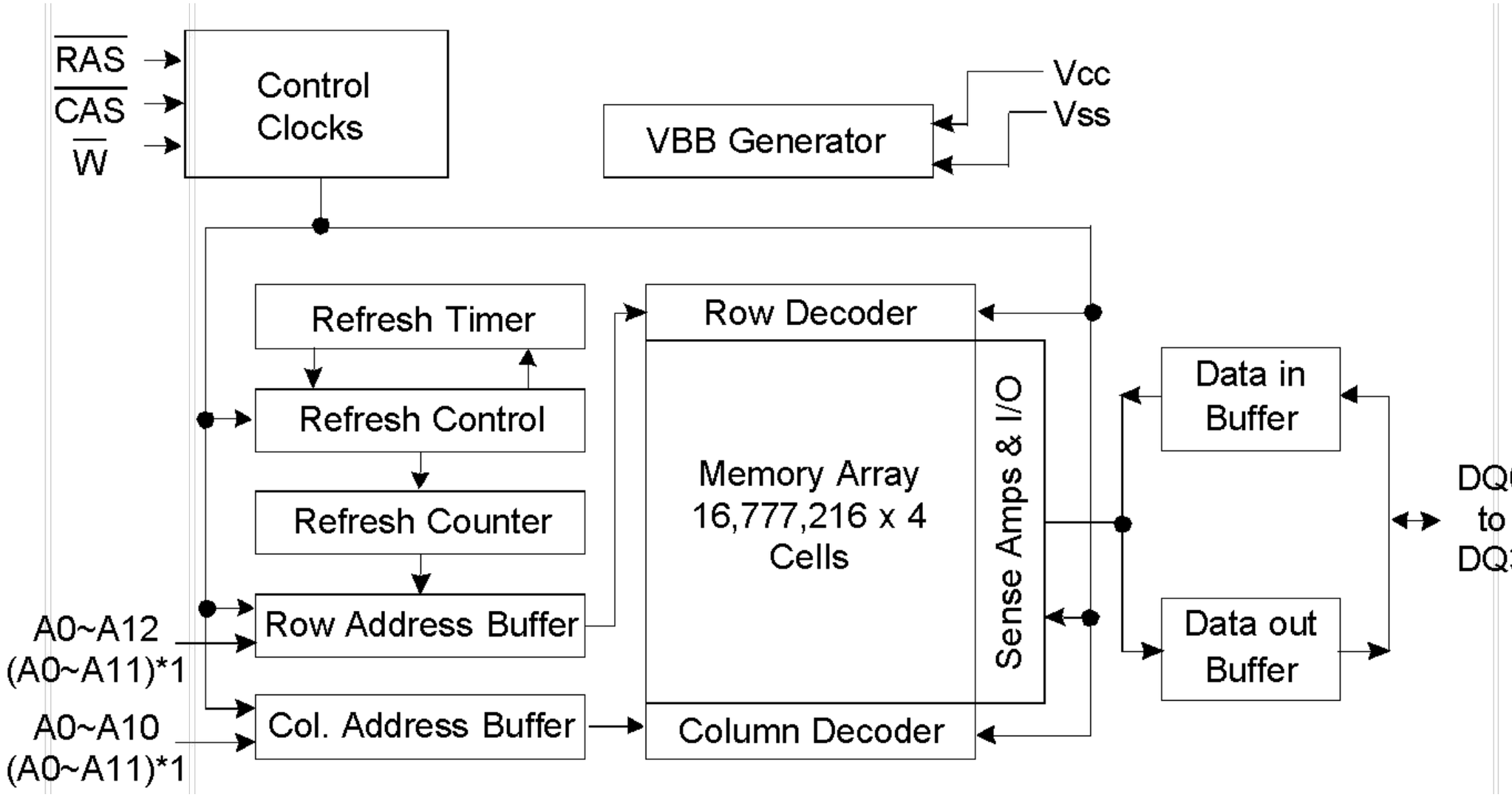
Exemple

- Mémoire 256 x 2
- Une banque de deux matrices de mémoire, chacune contenant 256 bits.
- Chaque adresse (i, j) correspond à deux bits
  - 1 bit sur la première matrice.
  - 1 bit sur la deuxième matrice.
- Mot de mémoire : 2 bits.

DRAM – Puce de mémoire (IIème génération)

Puce de mémoire **Samsung K4E660412D-JC50.**

http://www.datasheets360.com/pdf/-6286427797666065771



DRAM – Puce de mémoire (IIème génération)

Puce de mémoire **Samsung K4E660412D-JC50.**

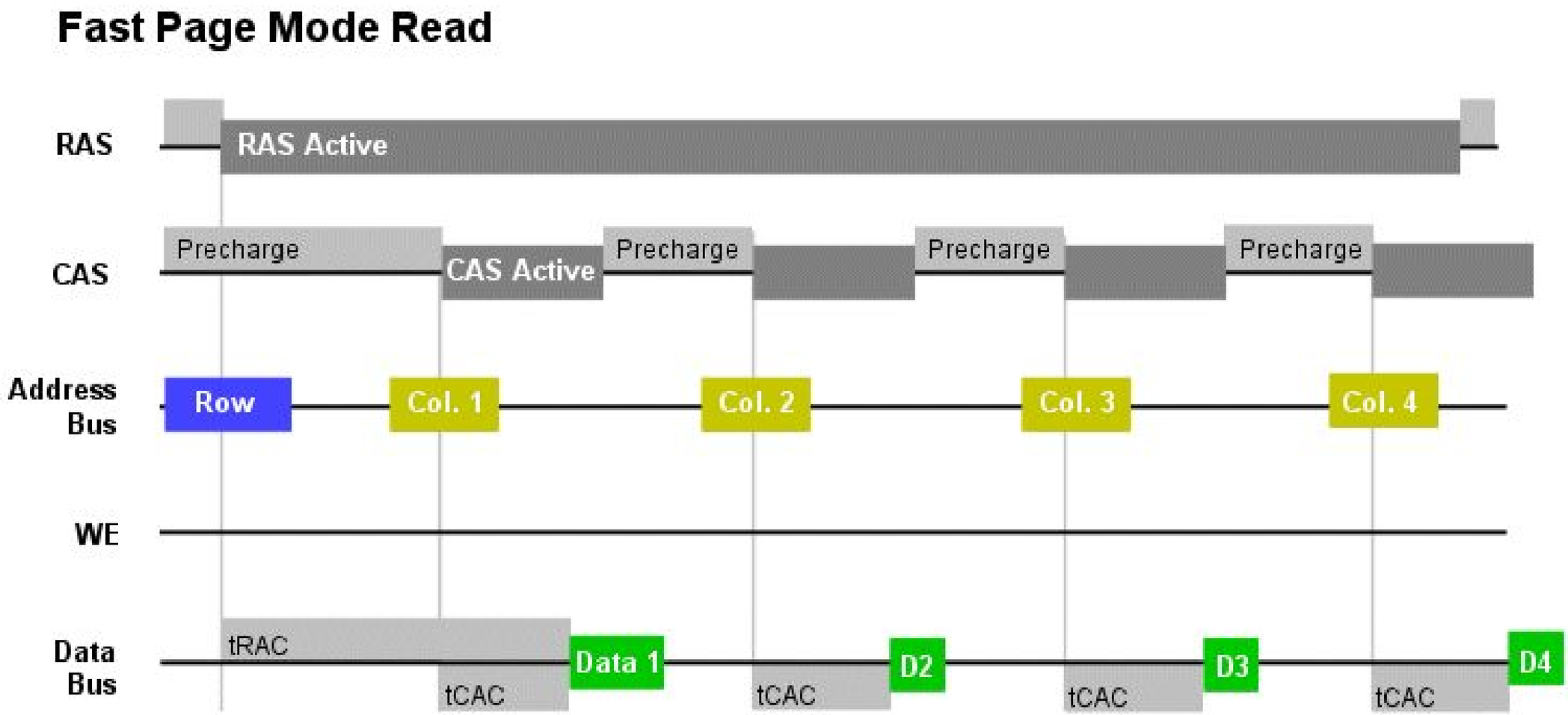
http://www.datasheets360.com/pdf/-6286427797666065771

Attribut	Valeur
Status	Discontinued
Sub Category	DRAMs
Access mode	FAST PAGE WITH EDO
Access Time-max	50ns
Memory width	4
Number of words code	16M
Operating mode	ASYNCHRONOUS
Organization	16Mx4

Taille de la mémoire : 16M * 4 bits = $2^4 * 2^{20} * 2^2$ bits
= $2^6 * 2^{20}$ bits = $2^3 * 2^{20}$ bytes = **8 MB**

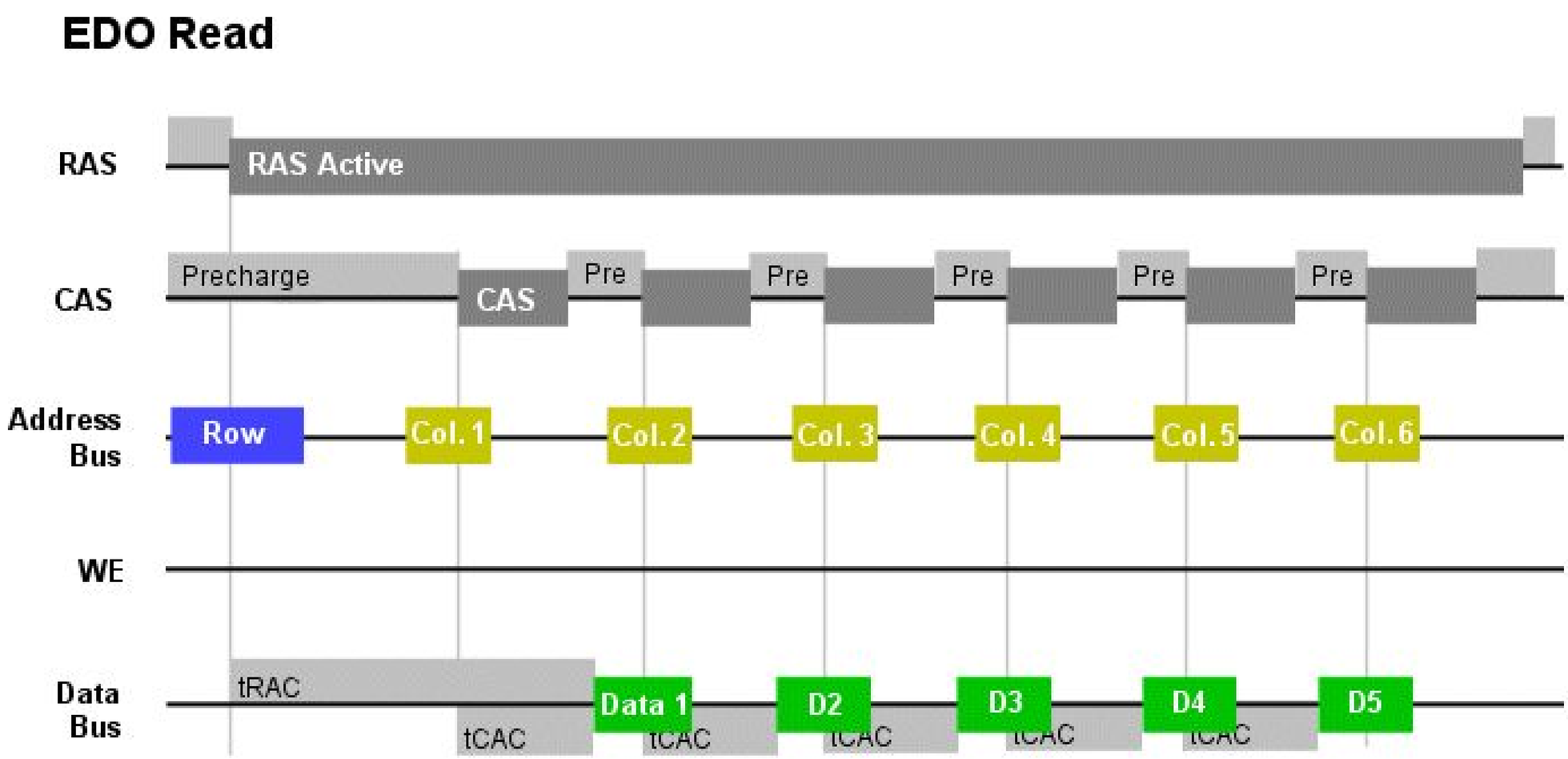
DRAM – IIème génération – Fast Page Mode (FPM)

- On donne un signal RAS et 4 signaux CAS.
  - On spécifie une adresse de ligne et on lit 4 colonnes sur cette ligne.
- Sortie d'une opération de lecture : 4 mots
- Il faut attendre le complètement d'une opération de lecture pour entamer la prochaine.

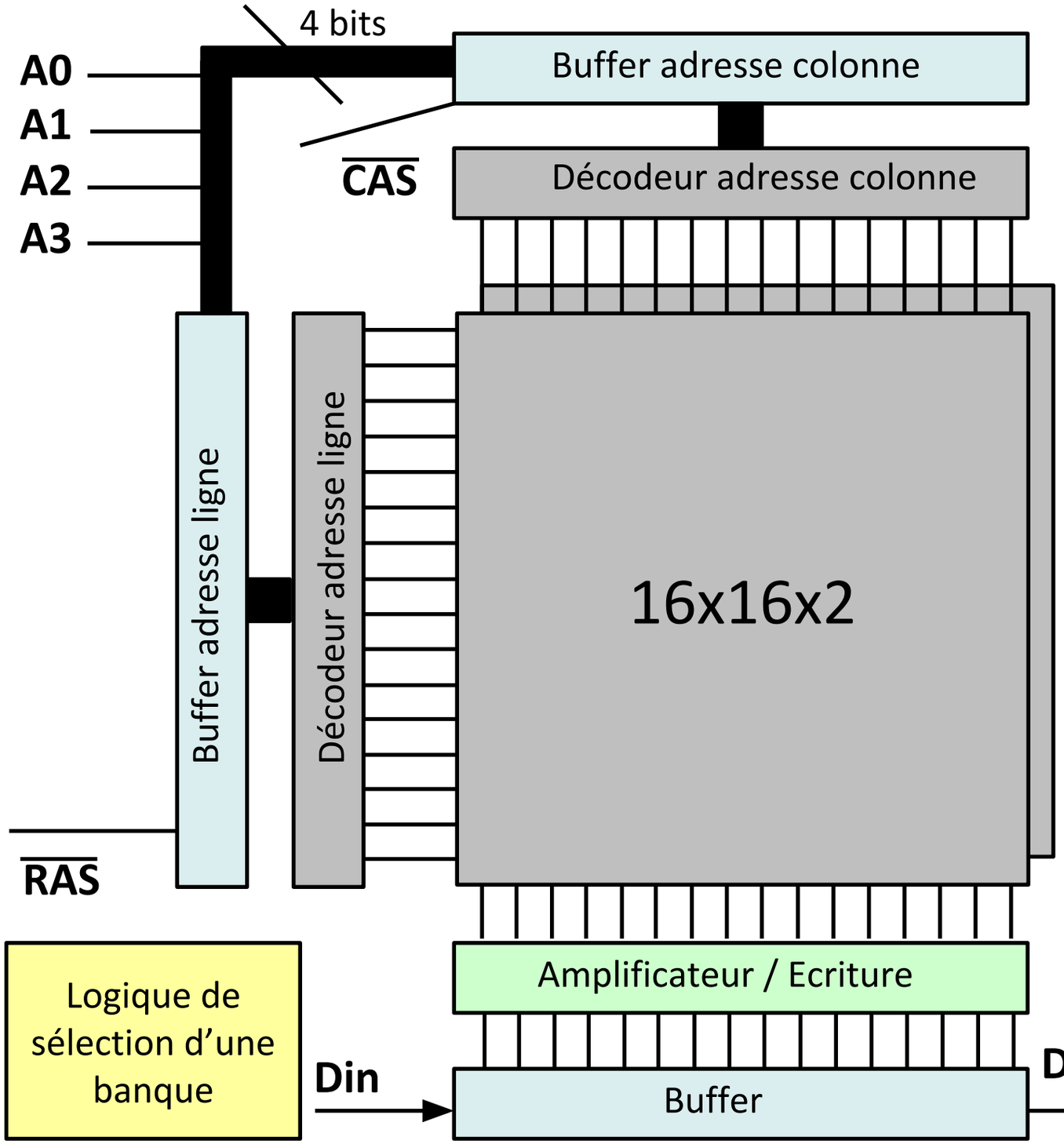


DRAM – IIème génération – EDO DRAM

- EDO : Extended Data Out.
- La sortie de la lecture en cours est enregistrée dans une mémoire tampon.
- On peut commencer la lecture suivante même avant la terminaison de la lecture précédente.



SDRAM – Puce de mémoire (IIIème génération)



- SDRAM : *Synchronous* DRAM
- La mémoire est **synchrone**
  - Le changement des signaux est réglé par une horloge.
- La mémoire consiste de **plusieurs banques** de **plusieurs matrices** de cellules.
- Seulement une banque est active à un moment donné.
- Avantage d'avoir deux banques : lorsqu'on attend des données d'une banque, on peut commencer la lecture d'une autre banque.

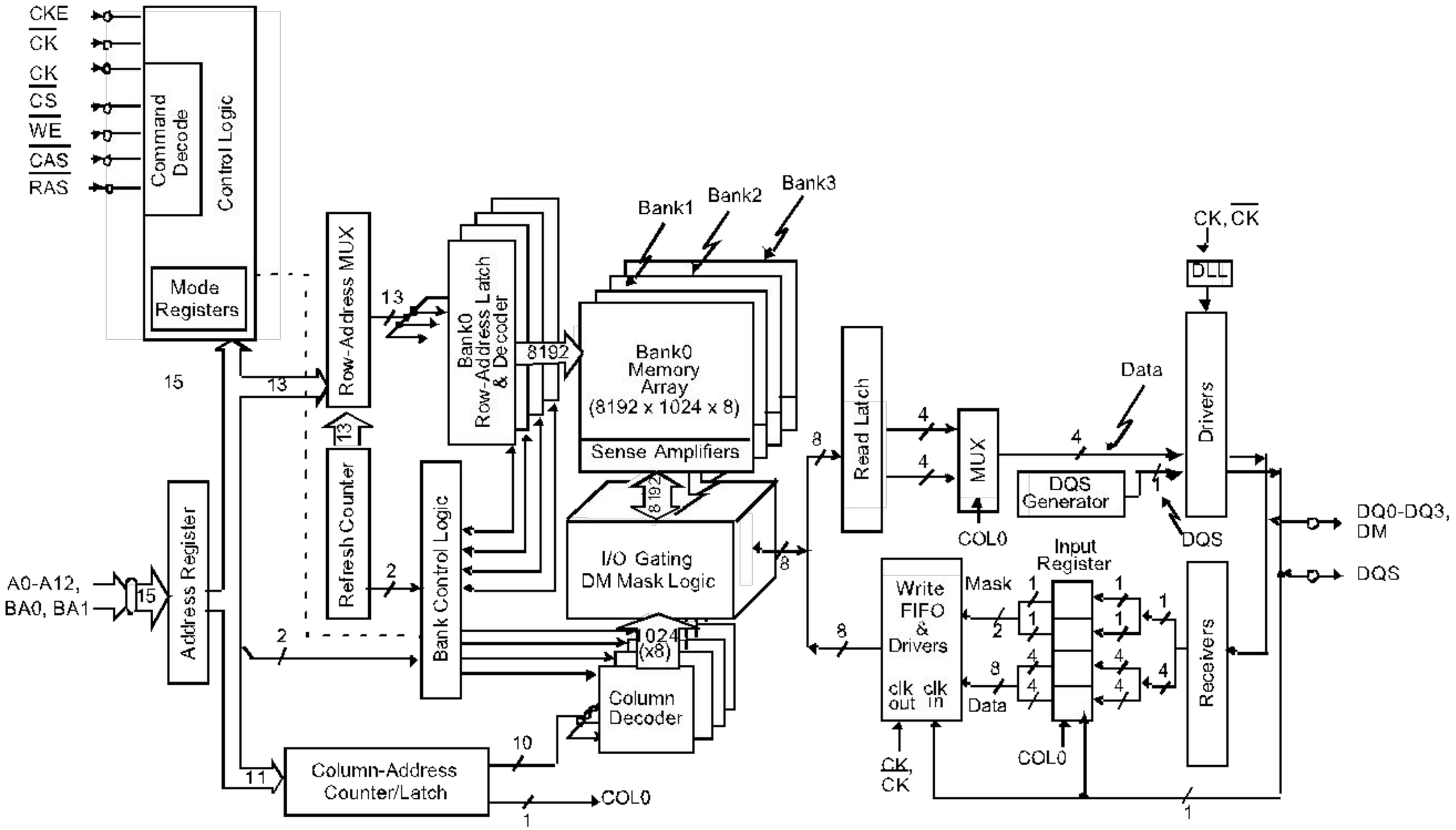
Exemple

- Mémoire 256 x 2 x 2.
- Deux banques de 256 mots de 2 bits.
- Mot de mémoire : 2 bits.

SDRAM – Puce de mémoire (IIIème géneration)

Puce de mémoire **Nanya NT5DS32M8BT-5T.**

http://www.datasheets360.com/pdf/4917439730698496913



SDRAM – Puce de mémoire (IIIème géneration)

Puce de mémoire **Nanya NT5DS32M8BT-5T.**

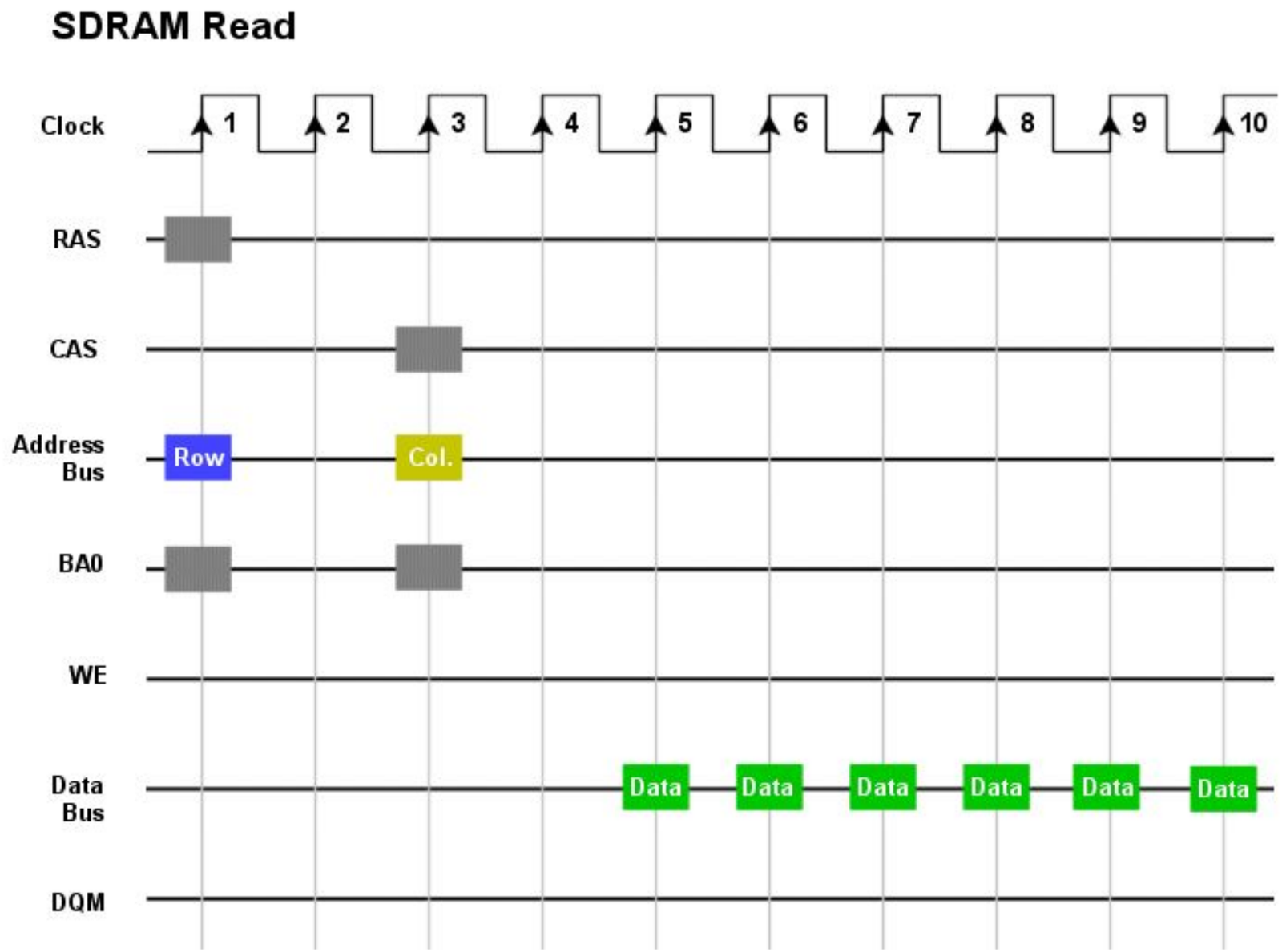
http://www.datasheets360.com/part/detail/nt5ds32m8bt-5t/4917439730698496913/

Attribut	Valeur
Status	Discontinued
Sub Category	DRAMs
Access mode	FOUR BANK PAGE BURST
Access Time-max	0.6500 ns
Memory width	8
Number of words code	32M
Operating mode	SYNCHRONOUS
Organization	32Mx8
Clock Frequency-Max (fCLK)	200 MHz

Taille de la mémoire : 32M * 8 bits = $2^5 * 2^{20} * 2^3$ bits = $2^8 * 2^{20}$ bits = $2^5 * 2^{20}$ bytes = **32 MB**

SDRAM – IIIème génération

- Une puce de mémoire SDRAM peut être programmée pour accéder à un nombre variables de colonnes (*burst mode*)



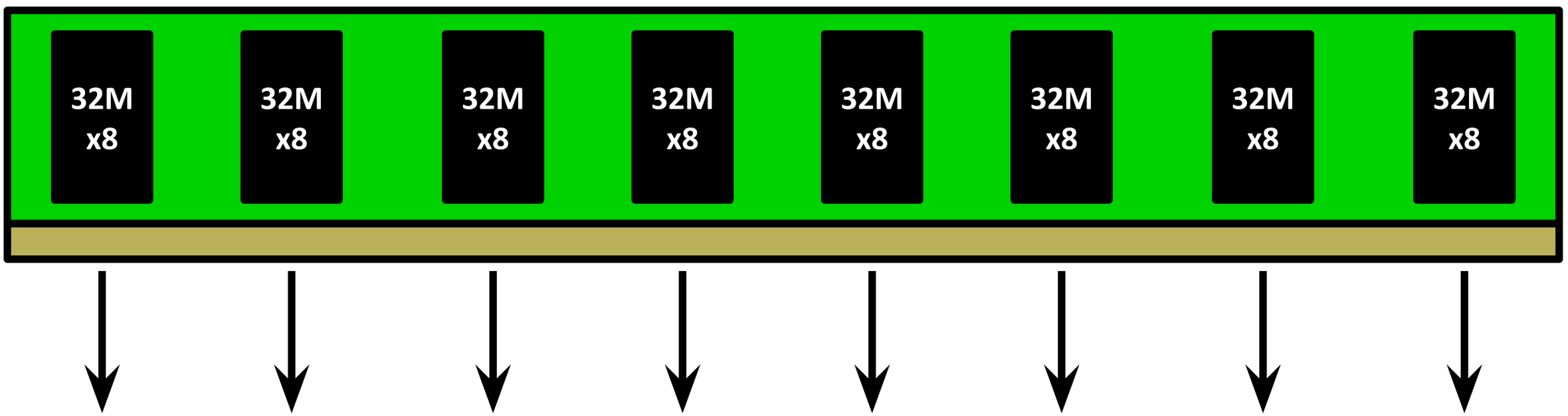
SDRAM – IIIème génération

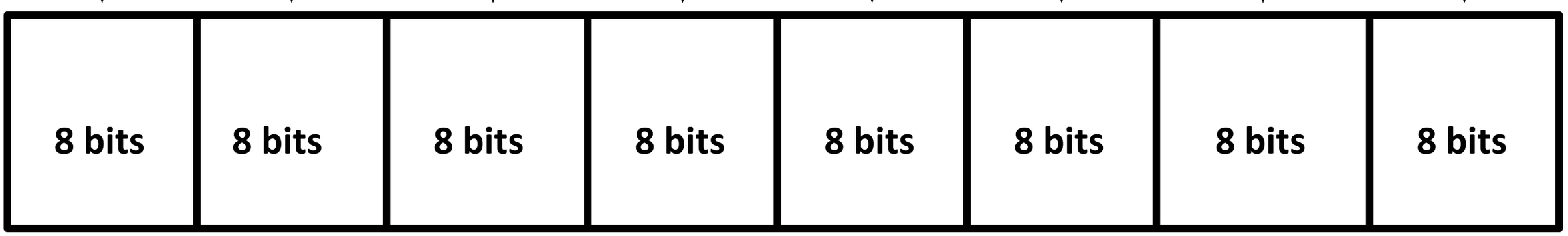
Double data rate SDRAM : les données sont transférées sur le front montant et sur le front descendant de l'horloge.

DDR Standard	Horloge du bus (MHz)	Fréquence interne (MHz)	Débit de données (MT/s)
DDR	100 – 200	100 – 200	200 – 400
DDR2	200 – 533,33	100 – 266,67	400 – 1066,67
DDR3	400 – 1066,67	100 – 266,67	800 – 2133,33
DDR4	1066,67 – 2133,33	133,33 – 266,67	2133,33 – 4266,67

source : https://en.wikipedia.org/wiki/DDR_SDRAM

La téchnologie DRAM – Module « 1-rank »

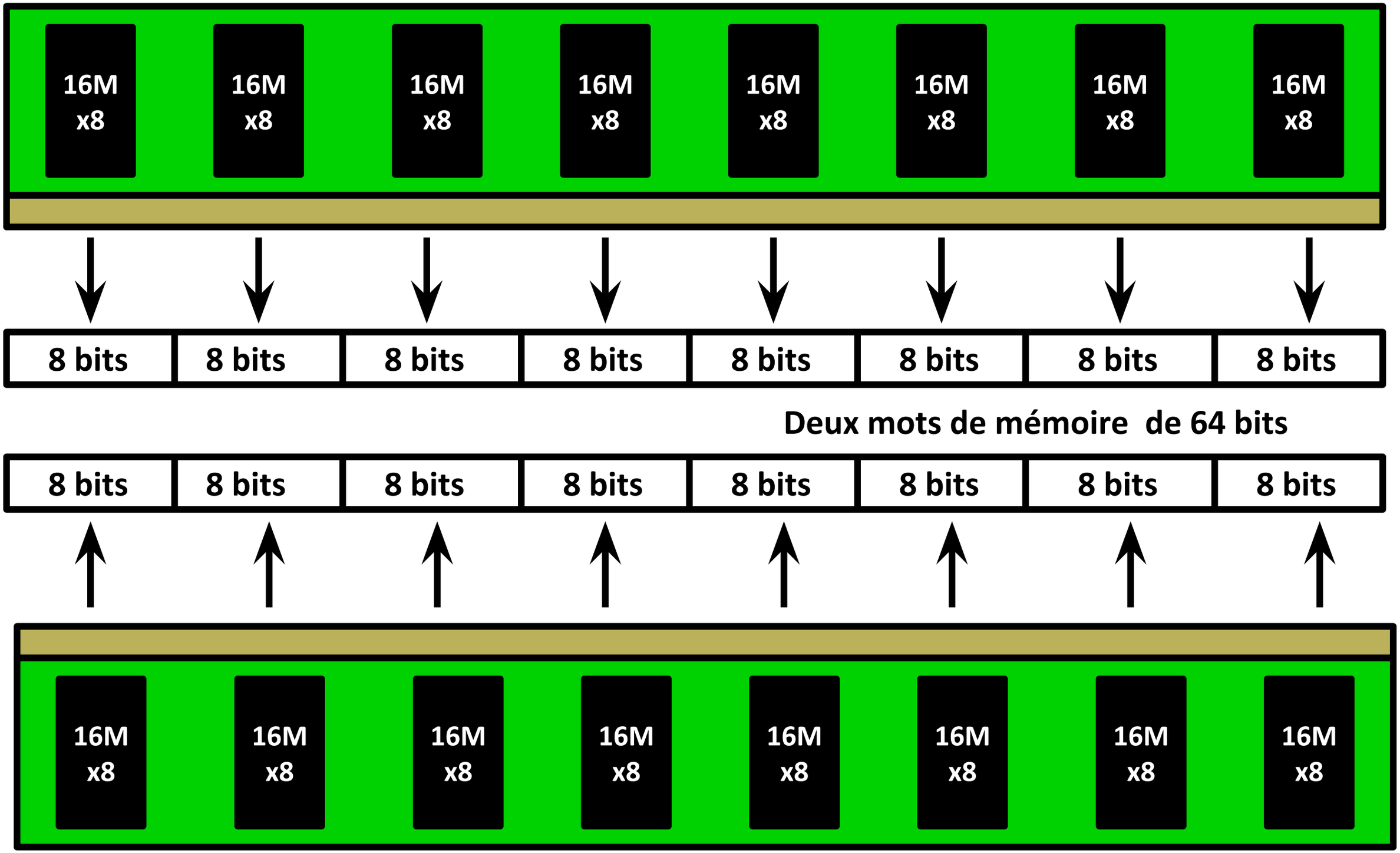




Mot de mémoire : 64 bits

- Le module contient 8 puces de 32MB.
- La taille du module est 256MB.

La technologie DRAM – Module « 2-rank »



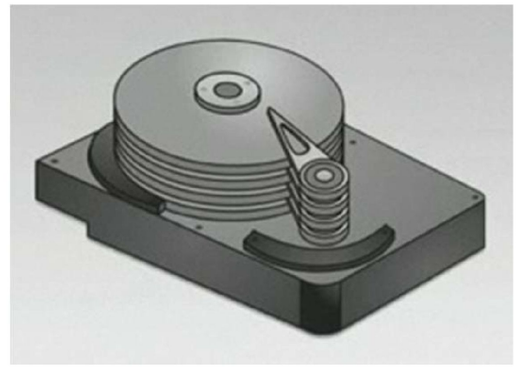
La mémoire secondaire

- Mémoire non-volatile.
- Mémoire plus lente que la mémoire principale.
- Mémoire beaucoup plus grande que la mémoire principale.
- Il était une fois …. le ruban magnétique (introduit par IBM en 1953).
  - Mémoire à accès séquentiel : pour lire une donnée, il faut parcourir toutes les données précédentes.
- Aujourd'hui : disques magnétiques, SSD.

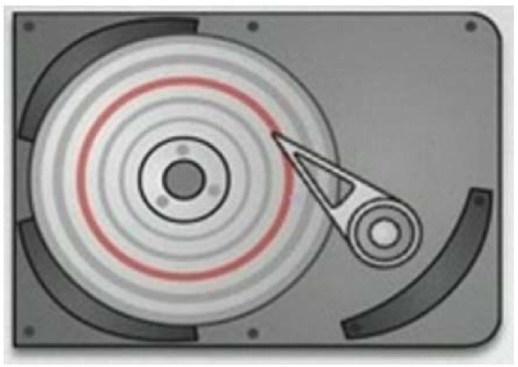
Disques magnétiques

- Il était une fois… RAMAC (*Random-Access Method of Accounting Control*)
  - premier disque magnétique (1955).
  - poids : une tonne (!)
  - cinquante disques d'aluminium.

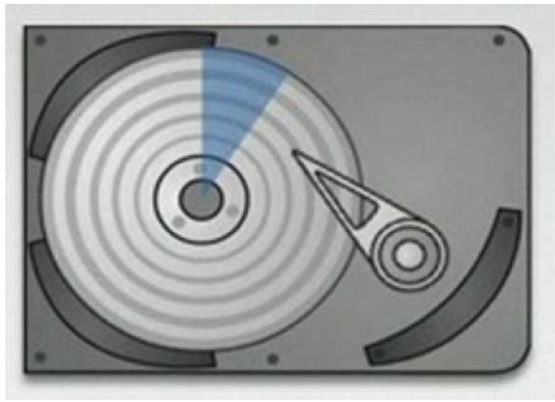
Disques magnétiques



- Un disque est composé de plusieurs **plateaux** (*platters*).
  - Chaque plateau consiste en deux **surfaces**.
  - Vélocité de rotation : 5400, 7200, 10000, 15000 rpm.
- Il y a une **tête** (*head*) de lecture/écriture pour chaque plateau.
  - les têtes ne peuvent pas se déplacer indépendamment.
- Le **bras du disque** (*spindle*) permet la rotation des plateaux .
- Chaque surface contient une collection de cercles concentriques, appelés **pistes** (*tracks*).
- Une piste est divisée en un certain nombre de secteurs (*sectors*).
  - taille fixe (typiquement 512 octets).



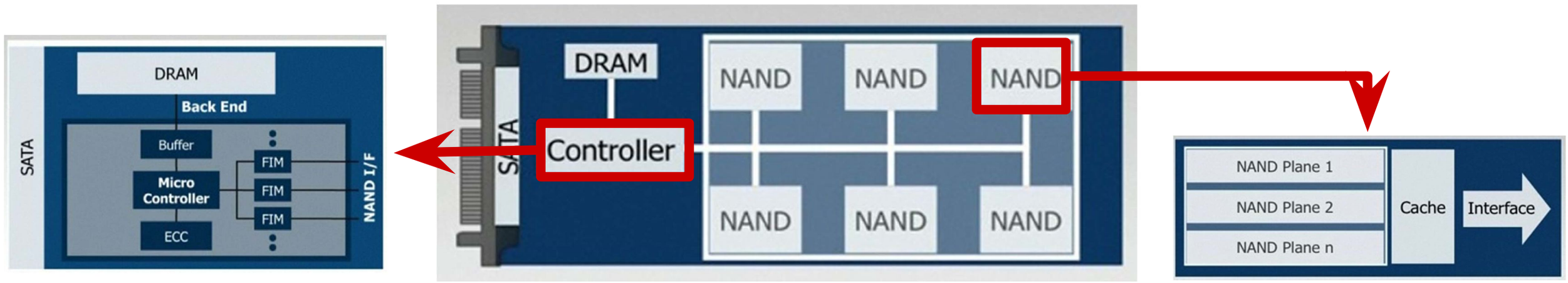
Piste



Secteur

Solid-State Drive (SSD)

- Un dispositif SSD contient un certain nombre de mémoires flash NAND.
- Chaque mémoire flash NAND contient des cellules NAND organisées en sous-ensembles.
  - dans une cellule NAND : 1 bit (*Single Level Cell*) ou plusieurs (*Multi Level Cell*).
- La mémorisation d'un bit est obtenue par des transistors qui forment des portes logiques NAND.
  - par défaut : une cellule contient un bit 1.
  - Opération de programmation (*program*) : mettre à zéro une cellule.
  - Opération d'effacement (*erase*) : remettre le bit à 1.
  - Chaque cycle *program/erase* (écriture) cause une usure de la cellule.



Avantages et inconvénients d'un SSD

Avantages :

- Rapide (pas de temps de positionnement)
- Résistant aux chocs.
- Pas de pièces mobiles.
  - silencieux
  - faible consommation d'énergie

Inconvénients :

- Plus cher qu'un disque magnétique.
- Capacité inférieure à un disque magnétique.
- Cycles d'écriture limités
  - 1 à 2M cycles d'écriture pour SSD MLC
  - ~5M cycles d'écriture pour SSD SLC

Caractéristiques

	Temps d'accès	Capacité	Coût de 1 Go
SRAM	10 ns	0,008 Go	5 000 €
DRAM (DDR4)	45 ns	32 Go	2,50 - 18 €
Flash (SSD)	34 400 ns	1 To	0,26 – 0,44 €
Disque dur	12 000 000 ns	8 To	0,030 €

- Mémoire DRAM : module 32GB CT32G4RFD4213 (DDR4-2133R)
- Flash SSD : Samsung 850 Evo 1TB
- Disque dur : Seagate Archive HDD 8To S-ATA III

Accélération : mémoire cache

- Le processeur accède souvent aux mêmes mots de mémoire principale.
- Idée : ajouter une mémoire plus rapide entre le processeur et la mémoire principale pour stocker les mots utilisés fréquemment.
  - cette mémoire est appelée mémoire cache.
- La mémoire cache est une mémoire SRAM.
  - La mémoire cache est donc plus rapide que la mémoire centrale (DRAM) ….
  - …mais plus petite (car sinon elle coûterait une fortune!).

- **Proximité temporelle**
  - Quand un processeur va chercher un mot en mémoire, il y a de fortes chances qu'il aura besoin du même mot peu après

  → Garder les mots utilisés récemment dans la mémoire cache
- **Proximité spatiale**
  - Quand un processeur va chercher un mot en mémoire, il y a de fortes chances qu'il ait besoin d'un mot voisin peu après

  → Ne pas stocker un mot isolé dans la mémoire cache, mais un *bloc* de mots contigus en mémoire

Principe du cache

adresse mot

contenu mot

Cache

adresse bloc

contenu bloc

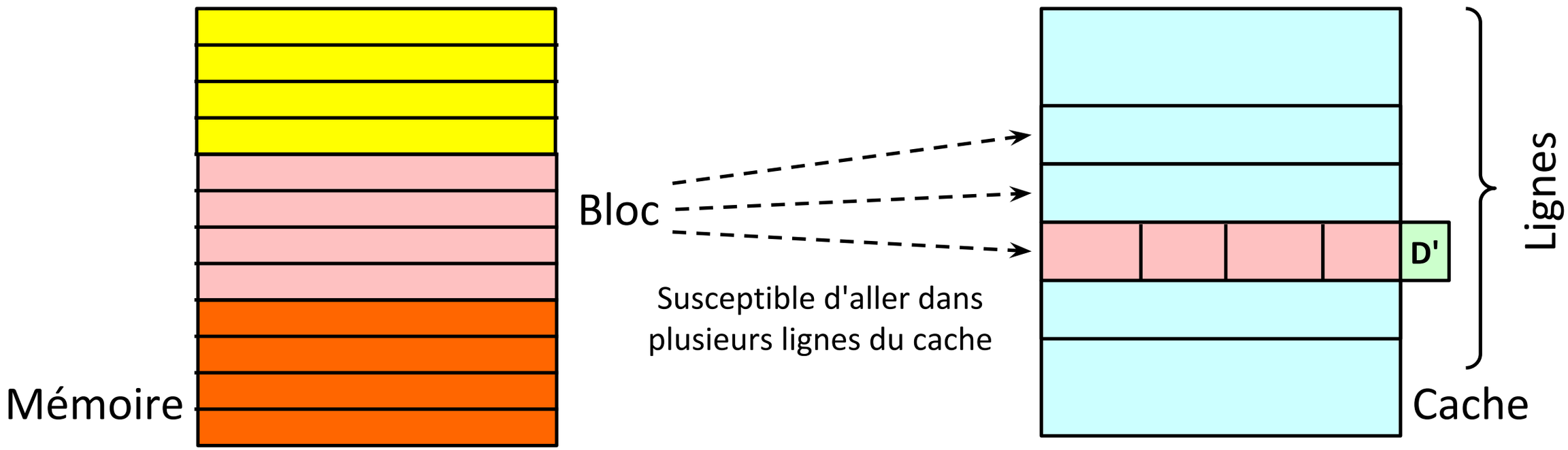
Mémoire centrale

si *miss*

Organisation d'un cache

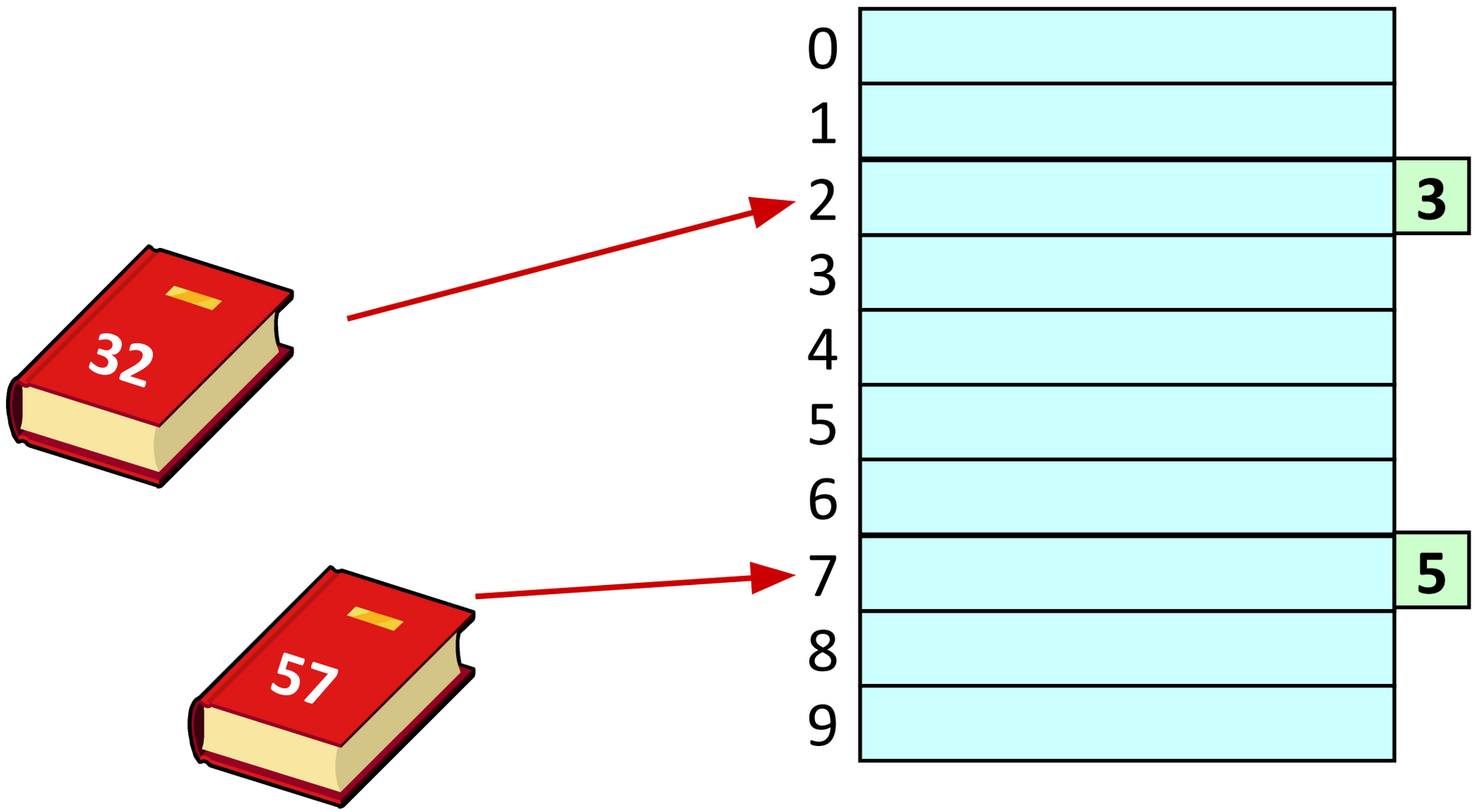
- La mémoire est vue "artificiellement" découpée en *blocs* de taille fixe.
- Un cache est organisé sous forme de N *lignes*, chaque ligne pouvant contenir un bloc (pas toujours le même)

- Chaque ligne possède un *descripteur* permettant de savoir quel est le bloc actuel qu'elle contient



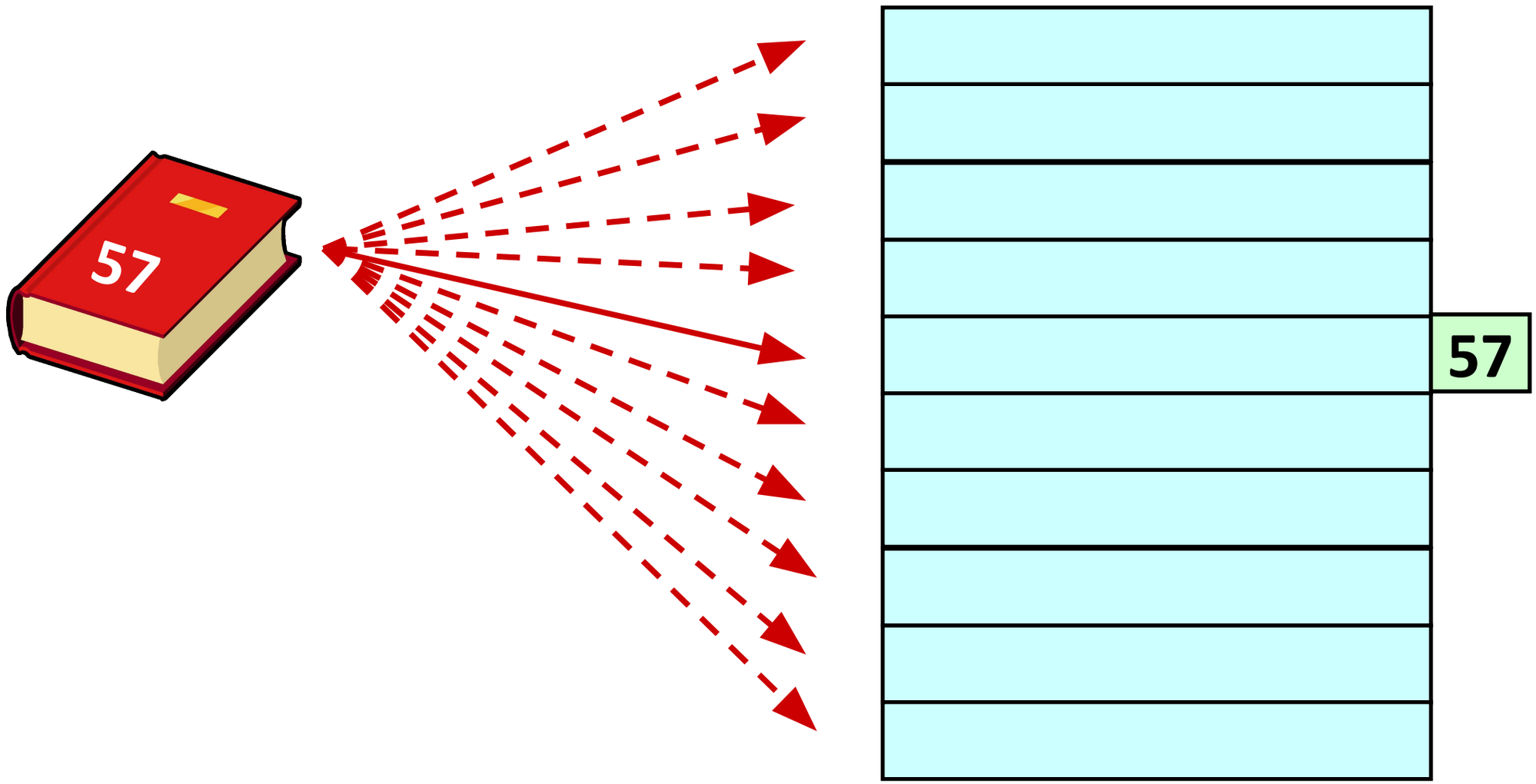
Organisation : directement adressé

Un bloc donné ne peut aller que dans une seule ligne



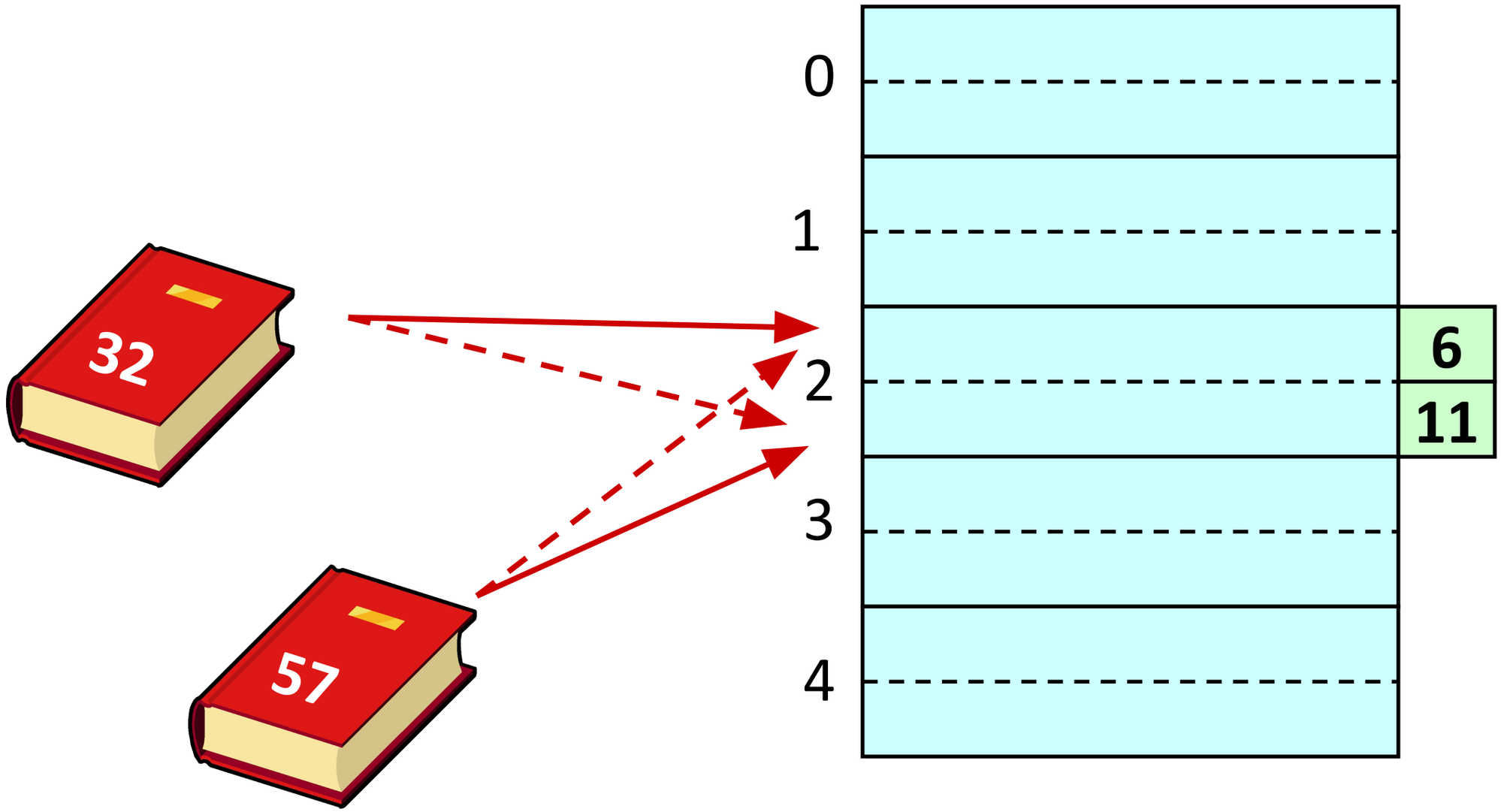
Organisation : purement associatif

Un bloc donné peut aller partout dans le cache



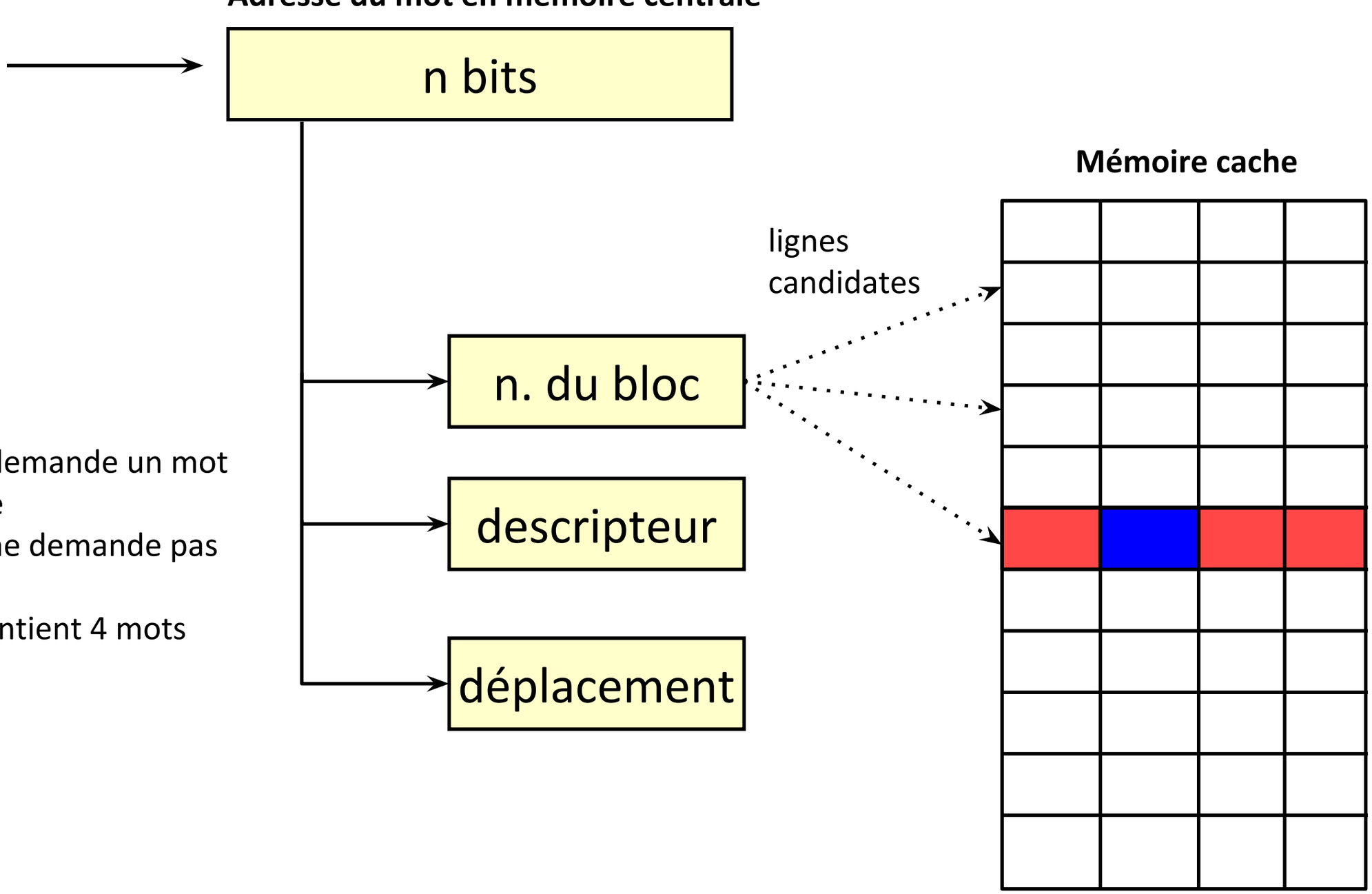
Organisation : associatif par sous-ensembles

Un bloc donné peut aller dans un sous-ensemble du cache



(two-way associative)

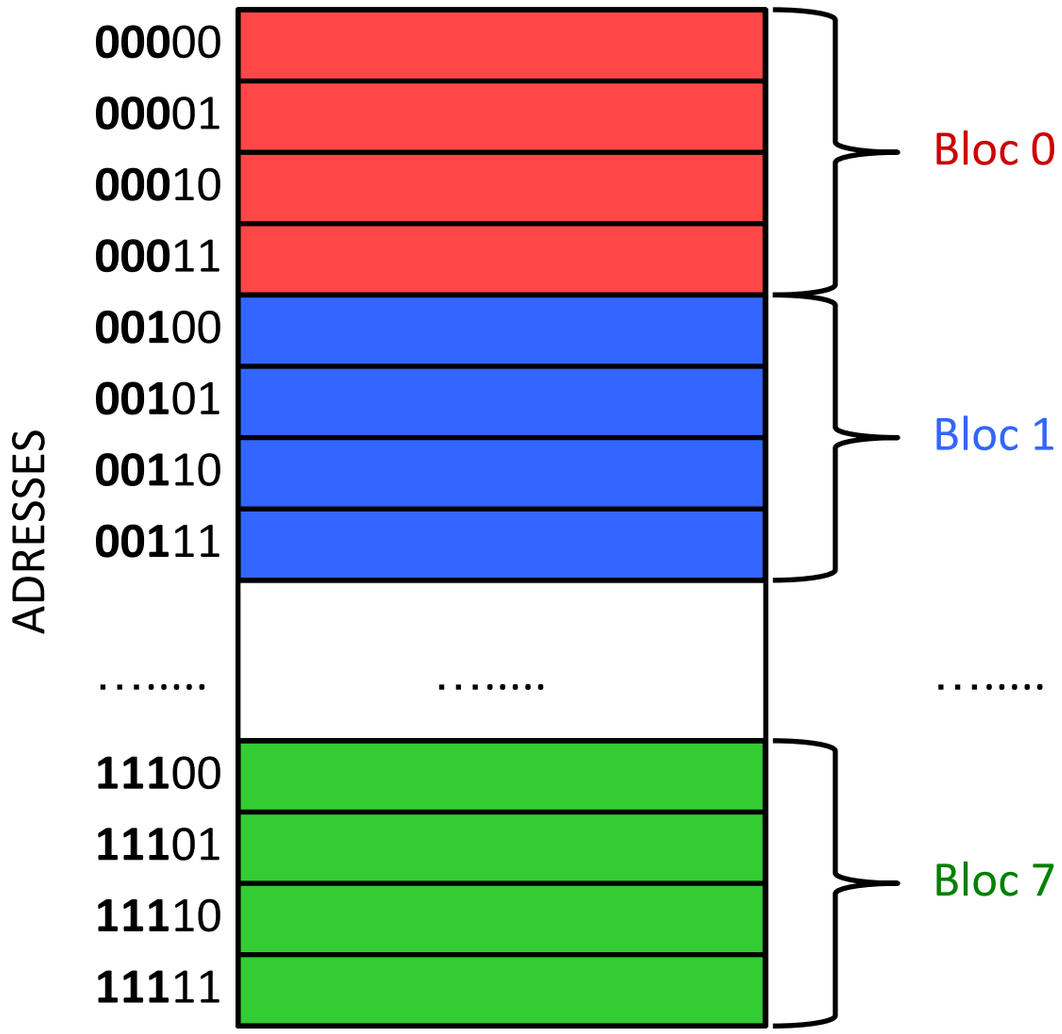
Recherche d'un mot de mémoire dans le cache



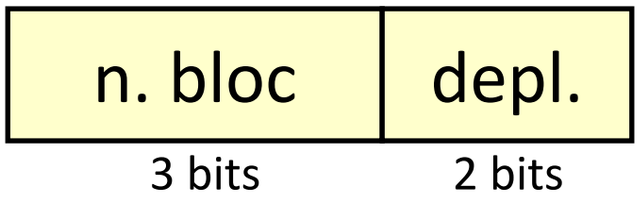
- Le processeur demande un mot par son adresse
- Le processeur ne demande pas un bloc!
- Chaque bloc contient 4 mots

Décomposition d'une adresse mémoire

- Mémoire centrale : 32 (2^5) mots.
- Une adresse de mémoire est codée sur 5 bits.
- Taille d'un bloc : 4 (2^2) mots.
- Nombre de blocs dans la mémoire centrale : 8 (2^3).

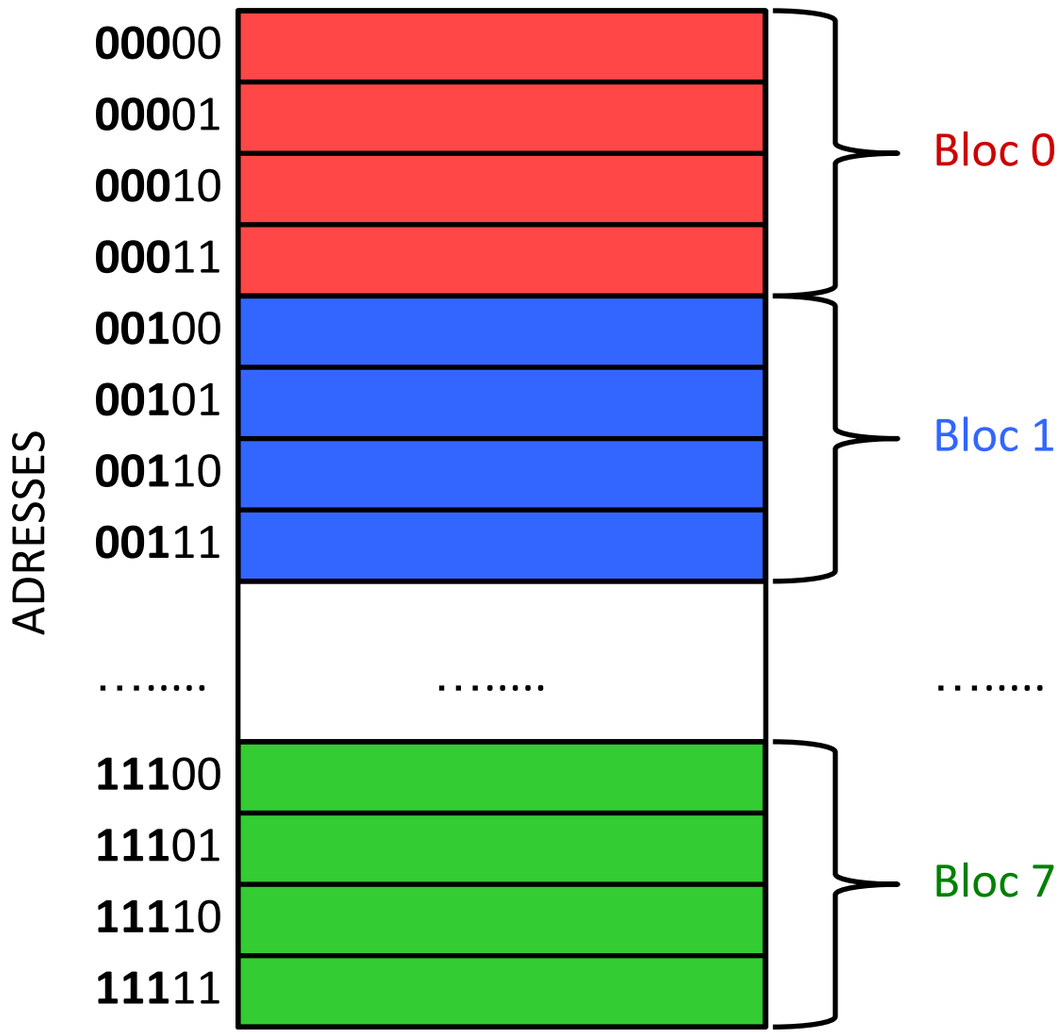


- Les trois bits de poids fort d'un mot indiquent le numéro de bloc d'appartenance d'un mot.
- Les deux derniers bits indiquent le déplacement dans le bloc pour atteindre le mot.



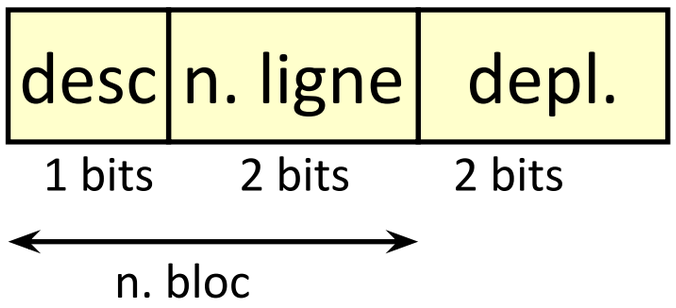
Décomposition – Cache directement adressé

- Mémoire centrale : 32 (2^5) mots.
- Une adresse de mémoire est codée sur 5 bits.
- Taille d'un bloc : 4 (2^2) mots.
- Nombre de blocs dans la mémoire centrale : 8 (2^3).



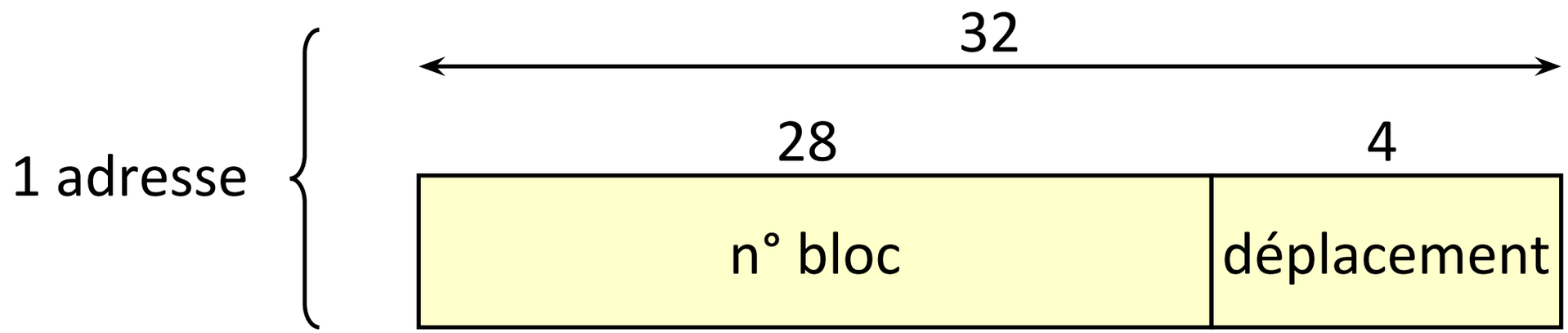
- Cache directement adressé avec 4 lignes.
- On peut caser 4 blocs au maximum.

Bloc 0**00** (0) et bloc 1**00** (4)	**Ligne 0**
Bloc 0**01** (1) et bloc 1**01** (5)	**Ligne 1**
Bloc 0**10** (2) et bloc 1**10** (6)	**Ligne 2**
Bloc 0**11** (3) et bloc 1**11** (7)	**Ligne 3**

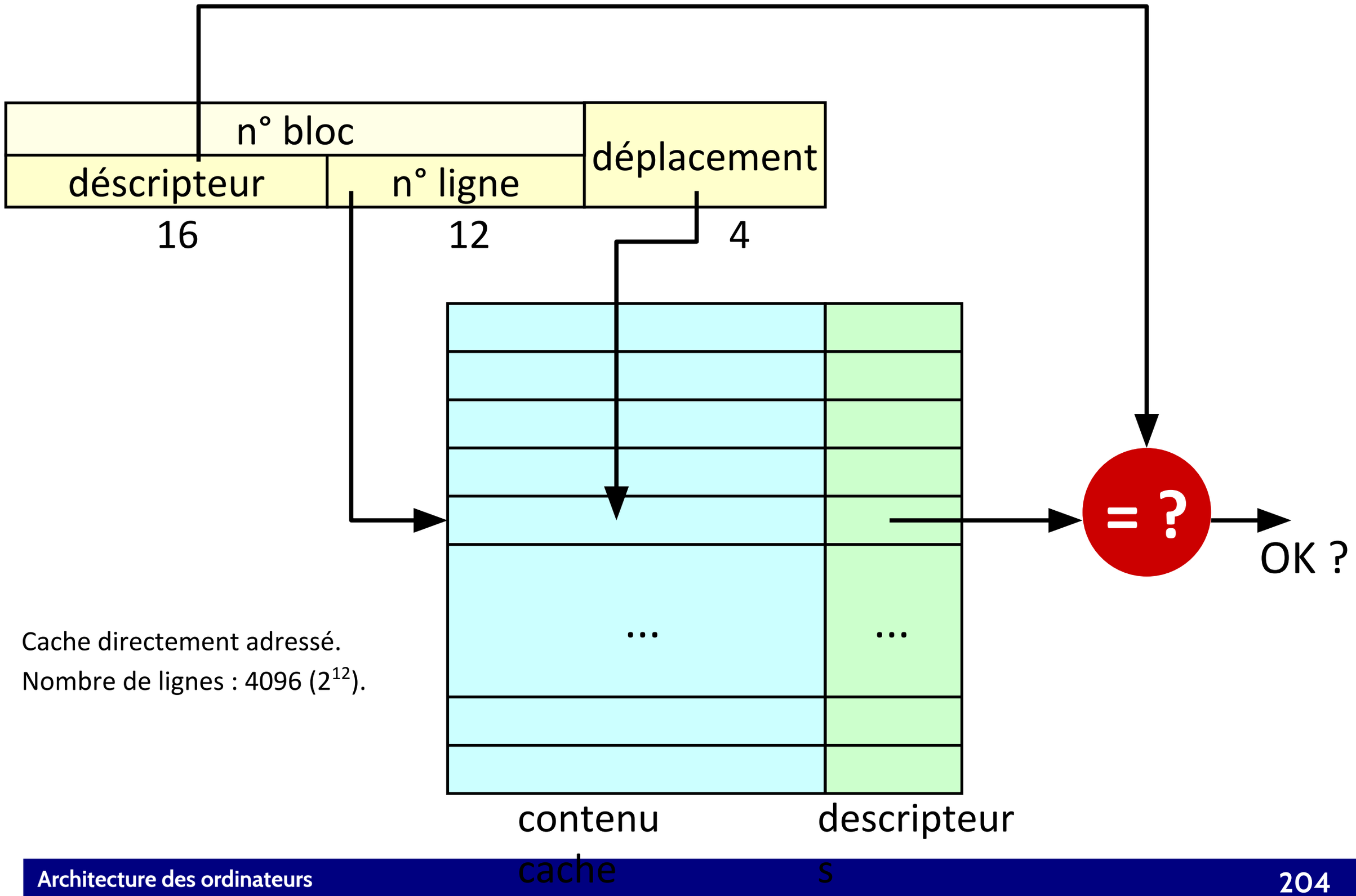


Exemple

- Mémoire centrale 4G (2^{32})
- Une adresse de mémoire est codée sur 32 bits.
- Taille d'un bloc : 16 (2^4) mots.
- Nombre de blocs dans la mémoire centrale : $2^{32}/2^4 = 2^{28}$
- Donc l'adresse sera composée de deux parties :
  - le numéro de bloc (sur 28 bits)
  - le déplacement dans un bloc (sur 4 bits)

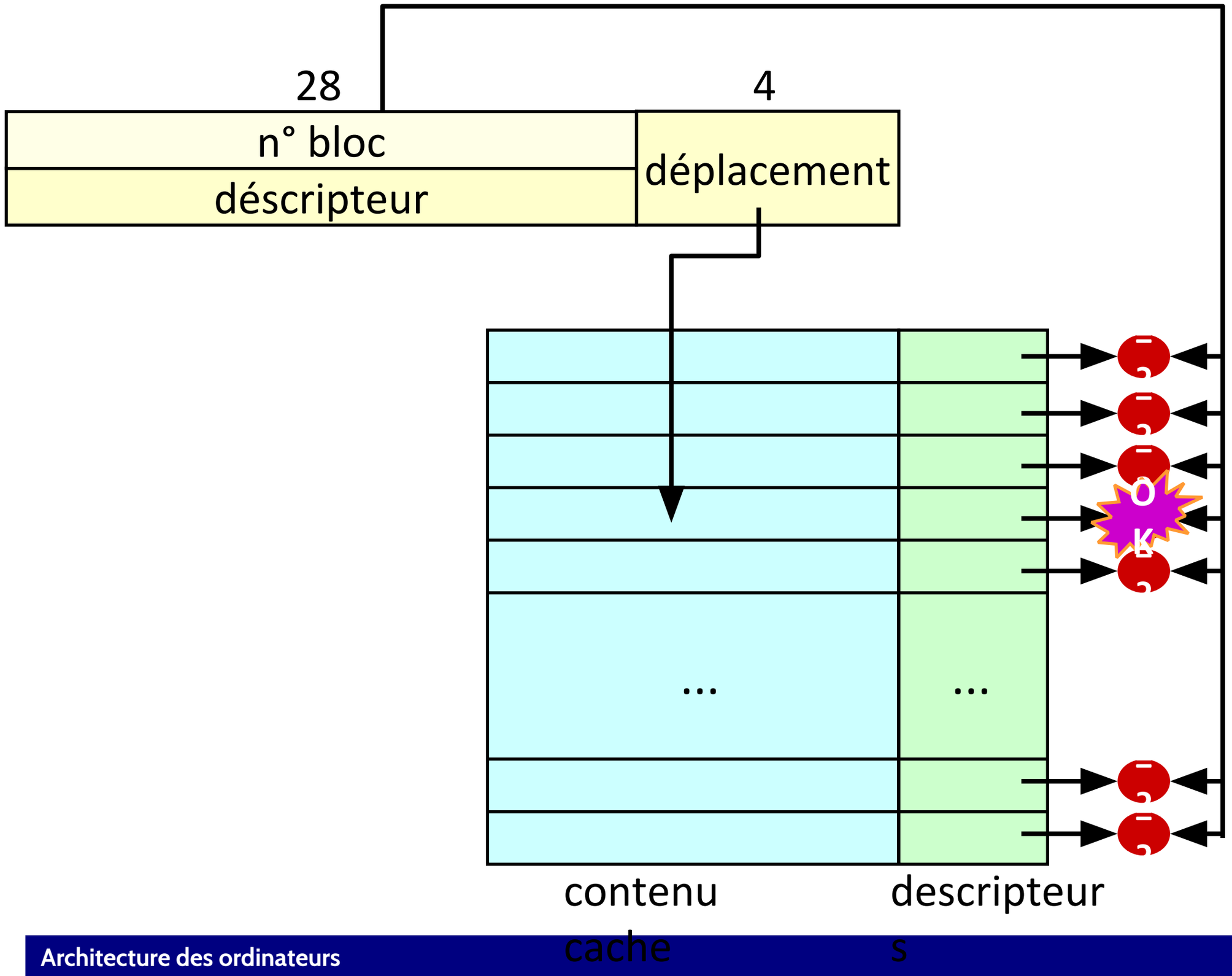


Recherche d'un mot – Cache directement adressé

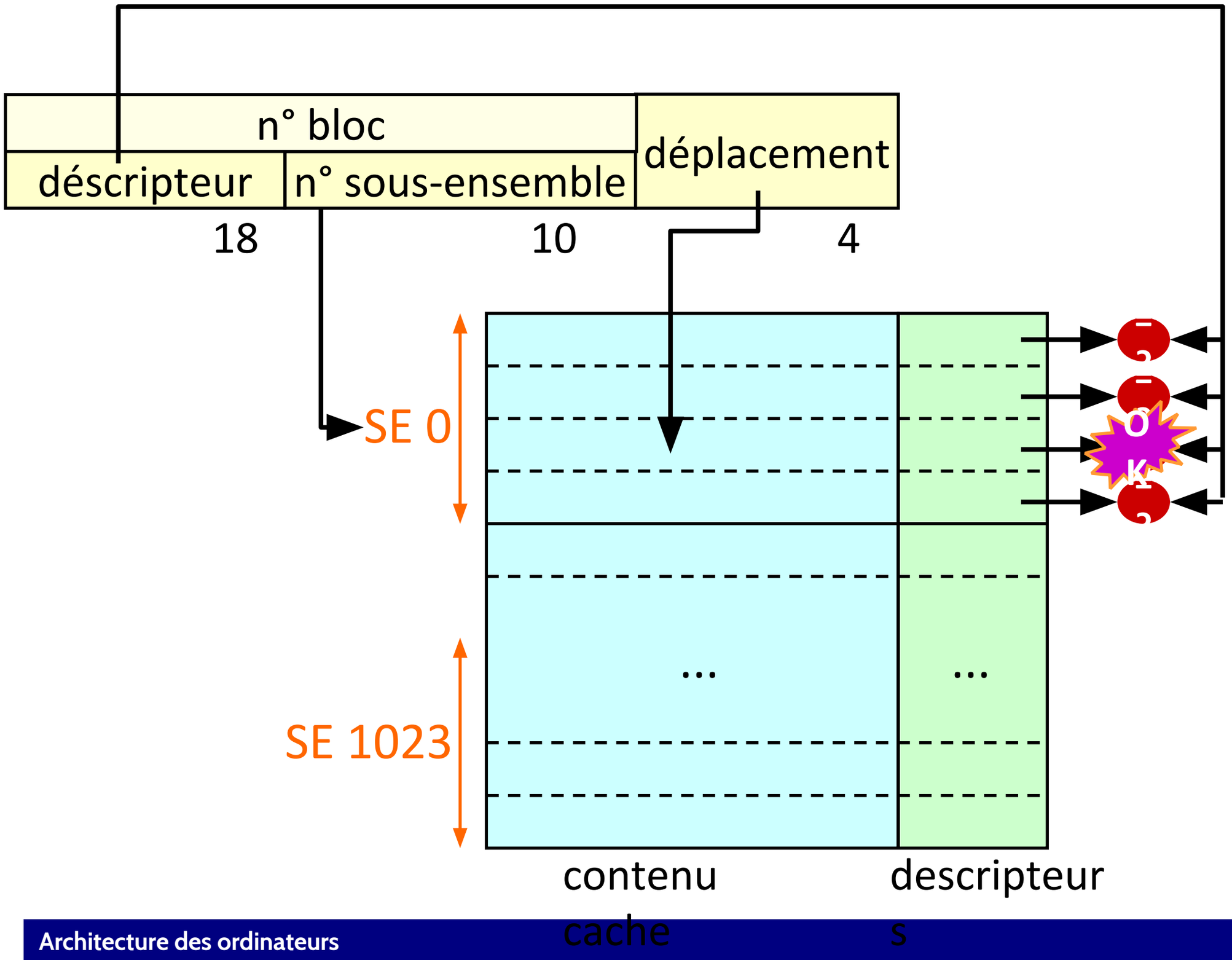


- Cache directement adressé.
- Nombre de lignes : 4096 (2^{12}).

Recherche d'un mot – Cache purement associatif



Recherche d'un mot – Cache associatif par SE



Algorithmes de remplacement

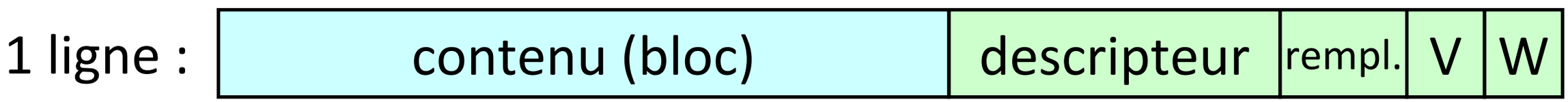
- Quand la mémoire cache est pleine il faut choisir le bloc à éliminer pour faire place à un nouveau bloc (sauf cache directement adressé).
- Algorithmes possibles :
  - *Random* : on élimine un bloc au hasard.
  - FIFO : on élimine le premier bloc qui a été ajouté à la mémoire cache.
  - LRU (*Least Recently Used*) : on élimine le bloc qui a été utilisé le moins récemment.
    - Nécessité d'ajouter des bits supplémentaires (*age bits*).
    - Age bits : compteur. A chaque fois qu'on utilise un bloc le compteur est remis à 0.

1 ligne dans la mémoire cache

valid bit : si la ligne contient des données valides (1 si la ligne n'est pas vide)

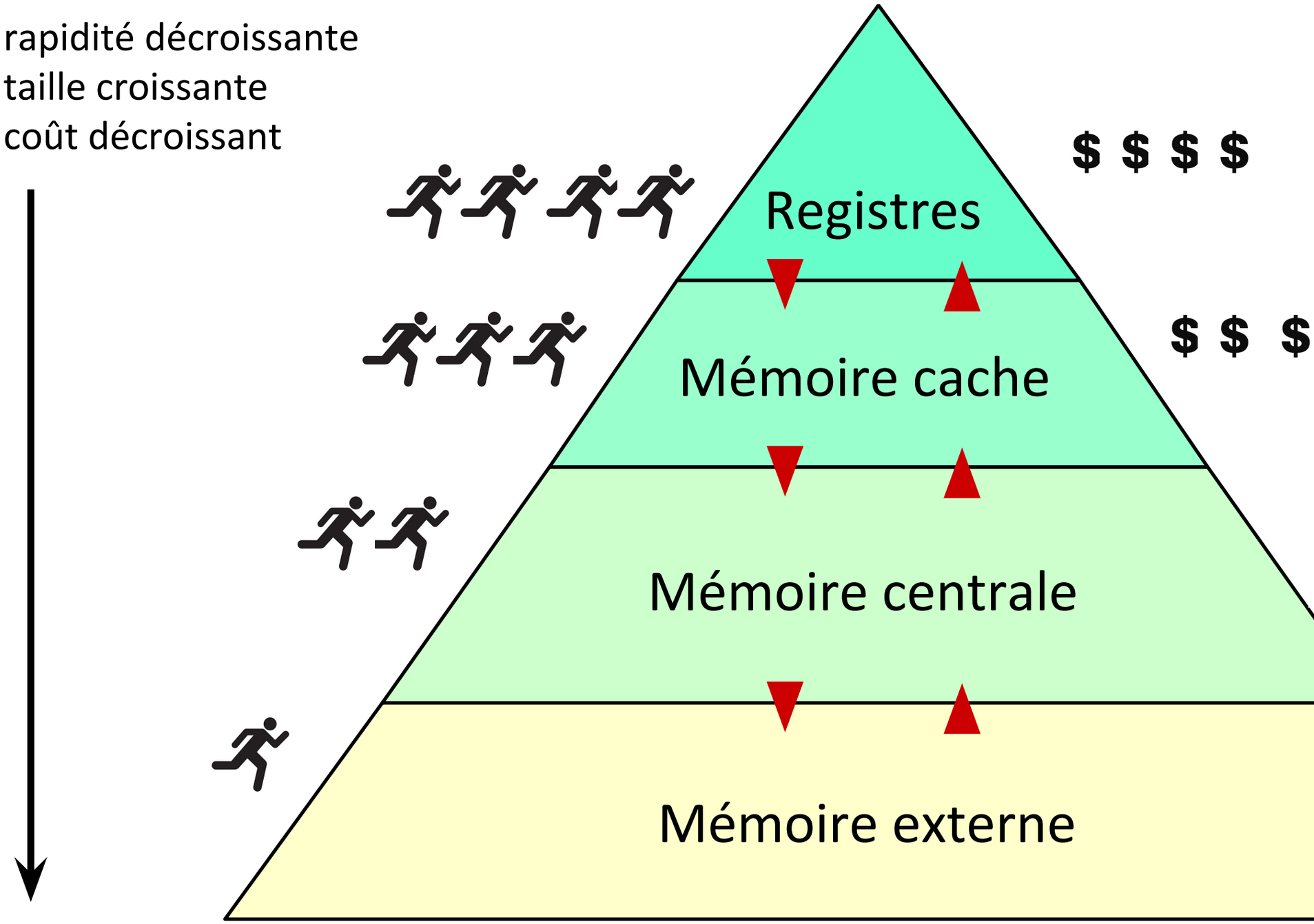
Stratégies d'écriture

- Lorsqu'on change le contenu d'un bloc dans la mémoire cache, il faut aussi changer le bloc correspondant en mémoire principale.
- Deux stratégies possibles : écriture immédiate ou écriture différée.
- Écriture immédiate (*write-through*).
  - Le bloc modifié est écrit immédiatement en mémoire principale.
  - **Inconvénient** : performance (un changement → deux écritures).
  - **Avantage** : cohérence entre cache et mémoire centrale
- Écriture différée (*write-back*).
  - Le bloc modifié est écrit en mémoire principale seulement lorsqu'il faut l'éliminer de la mémoire cache.
  - 1 bit W (*dirty bit*) pour chaque ligne.
  - **Avantage** : performance.
  - **Inconvénient** : le contenu du cache et celui de la mémoire centrale ne sont pas cohérents.



Hiérarchie mémoire

- rapidité décroissante
- taille croissante
- coût décroissant



CentraleSupélec

université PARIS-SACLAY

LABORATOIRE DE RECHERCHE EN INFORMATIQUE

CHAPITRE VI
Entrées/Sorties

Entrées/Sorties

- **Entrées/sorties – E/S** (*input*/*output – IO)* : échanges entre processeur et le monde externe (utilisateur).

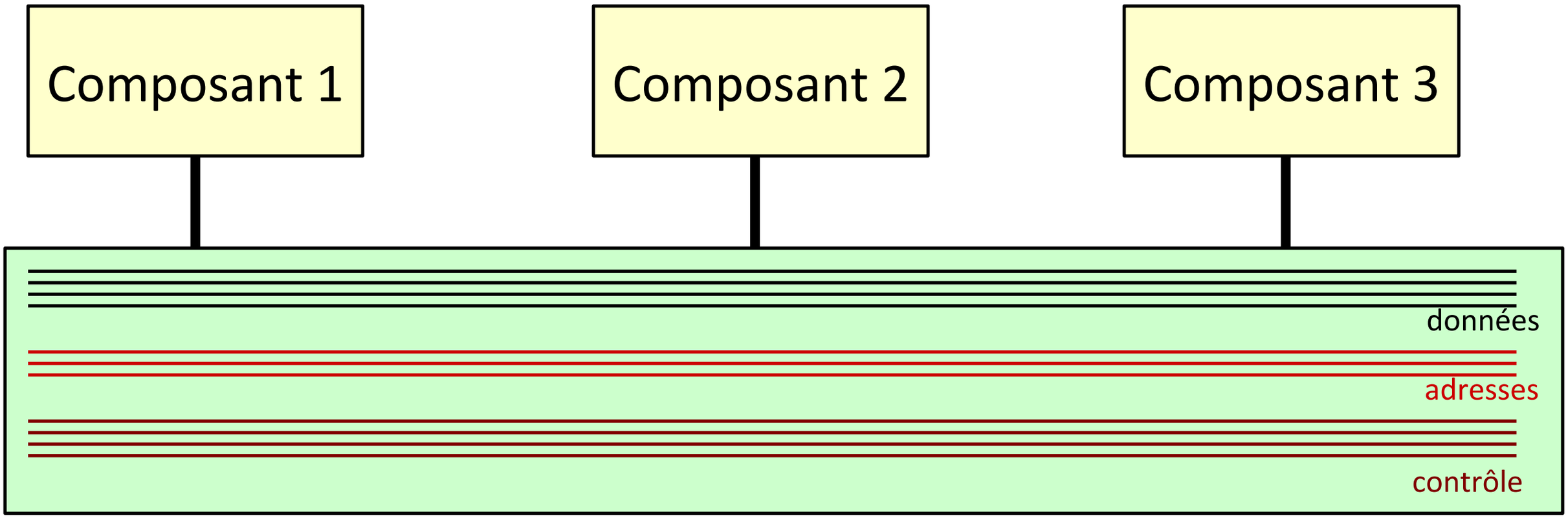
- **Périphériques** : dispositifs d'interface entre le processeur et l'utilisateur.
  - clavier, souris, écran, imprimante, disque dur ….

- Les échanges s'effectuent grâce à un **bus**.

- **Bus** (latin : *omnibus*) : dispositif permettant la transmission de données entre plusieurs composants.
  - fils, fibre optique....

- Un bus permet de relier des composants indépendants.
  - quand on change de clavier il ne faut pas changer de processeur...

Structure d'un bus



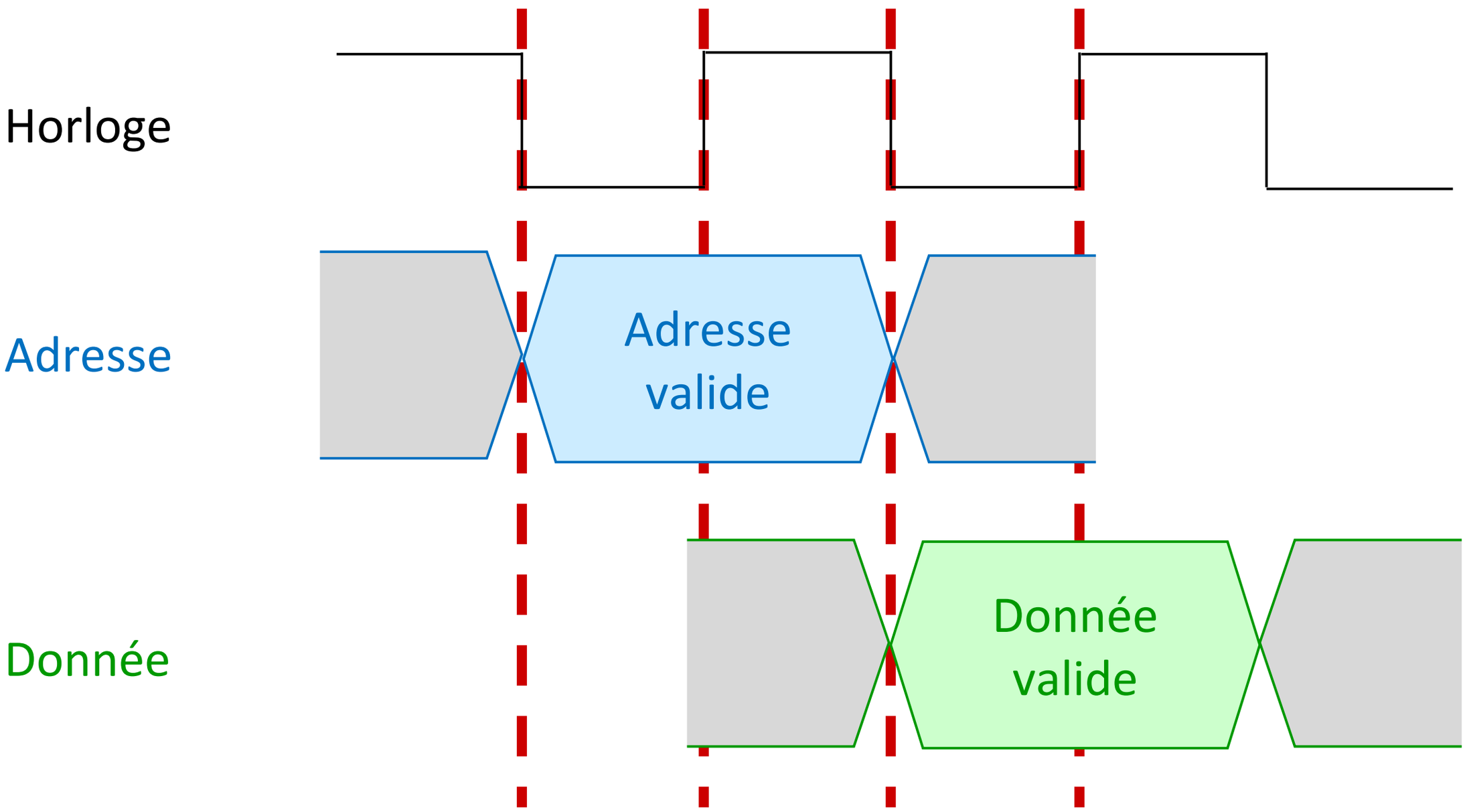
Un bus consiste typiquement de trois types de signaux :

- **Données** : les informations échangées pas les composants.
- **Adresses** : pour identifier un composant.
- **Contrôle** : signaux qui régissent la communication entre les composants.

- **Bus parallèle** : plusieurs signaux de données.
- **Bus série** : un seul signal de données.

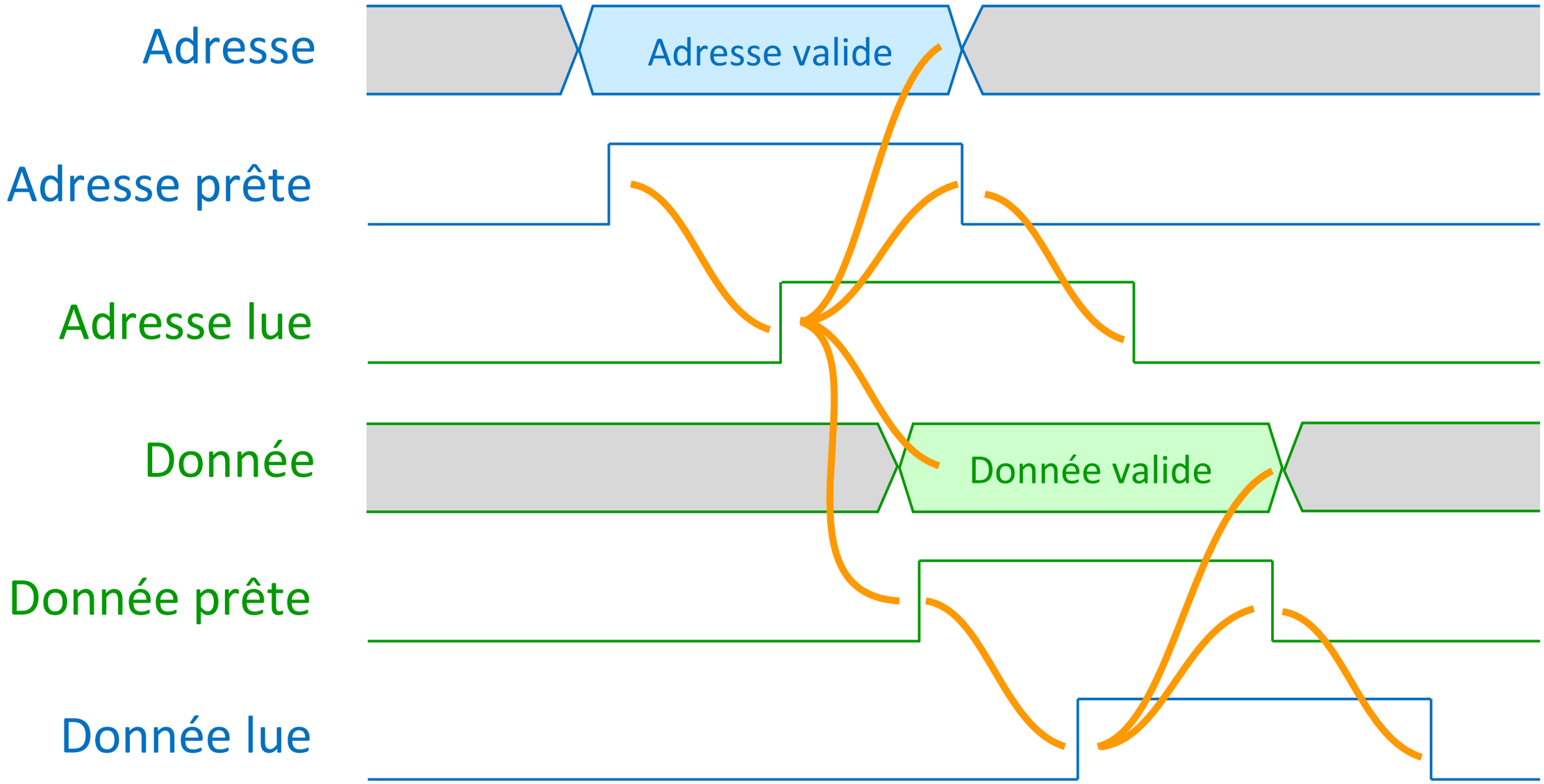
Bus synchrone

- Les signaux de contrôle incluent une horloge qui régit la communication.
- Exemple : Le processeur demande à la mémoire de lui fournir une donnée située à une certaine adresse



Bus asynchrone

- Différents signaux de contrôle et protocole de *handshaking*.
- Exemple : Le processeur demande au disque de lui fournir une donnée.



Bus synchrone Vs Bus asynchrone

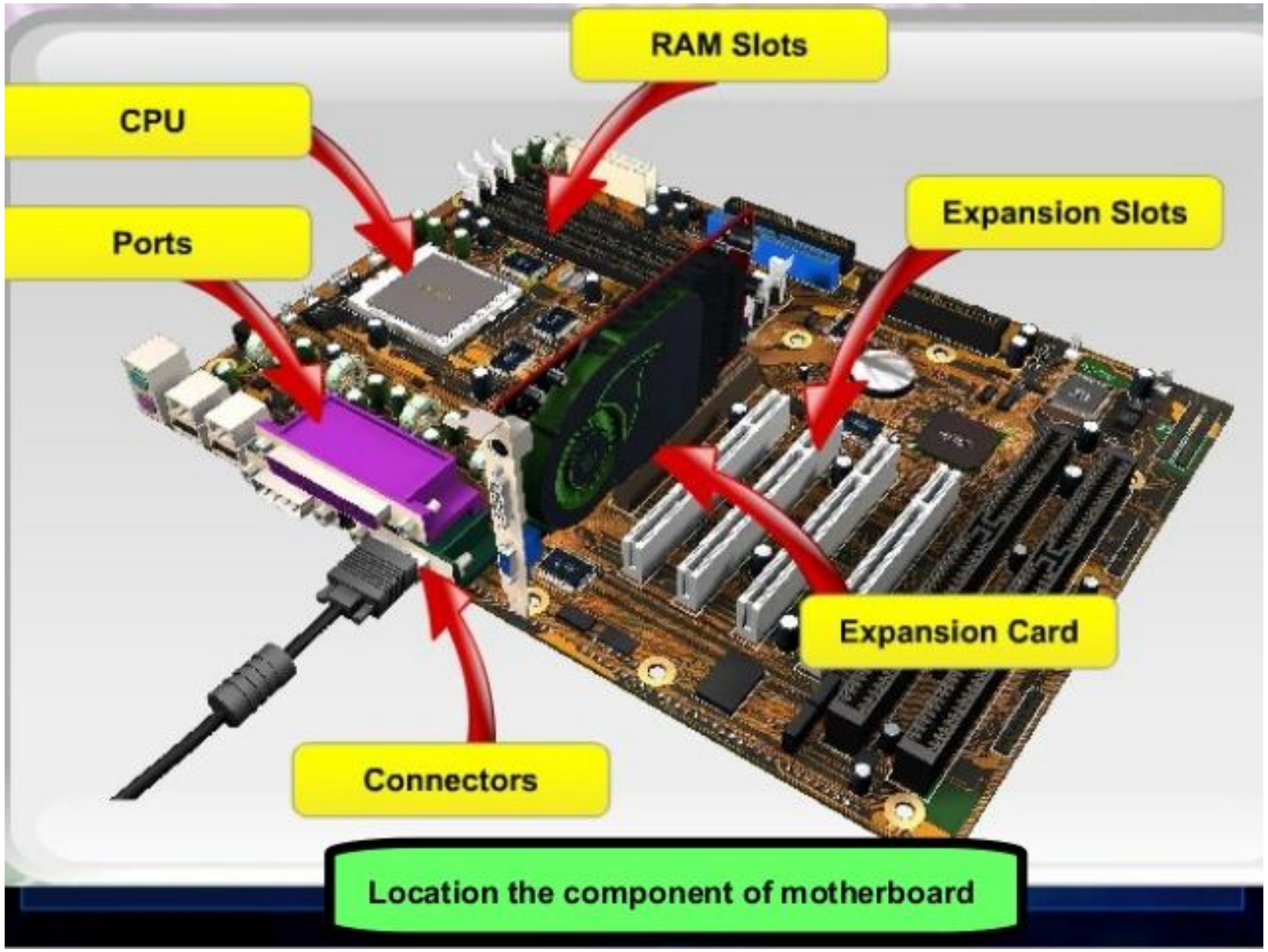
- **Bus synchrone**.
  - Avantages :
    - moins de signaux de contrôle.
    - rapide.
  - Désavantages :
    - tous les composants connectés au bus doivent utiliser la même fréquence d'horloge.
    - la longueur du bus doit être limitée.

- **Bus asynchrone.**
  - Avantages :
    - peut connecter des composants ayant une rapidité de fonctionnement différente.
    - sa longueur ne pose pas de problèmes.
  - Désavantages :
    - plus lent.
    - beaucoup de signaux de contrôle.

Standards de bus

- **Bus d'extension** (*expansion bus*).
  - liés à un connecteur d'extension (*slot*) permettant de connecter des composants (carte graphique, carte son, carte réseau).
  - différents standards : ISA, EISA, PCI, PCIe.
- **Interfaces de disque**.
  - utilisés pour connecter des disques.
  - différents standards : ATA, IDE, SCSI, SATA
- **Bus externes.**
  - utilisés pour connecter des composants externes (imprimantes, écrans…)
  - différents standards : LPT (imprimante), PS/2 (clavier/souris), USB.
- **Bus de communication.**
  - utilisés pour connecter deux ou plusieurs systèmes.
  - différents standards : LPT, RS232C, Ethernet.

Carte mère



source : https://www.slideshare.net/lishengshun90/motherboard-34369427

Bus partagé ou bus point à point

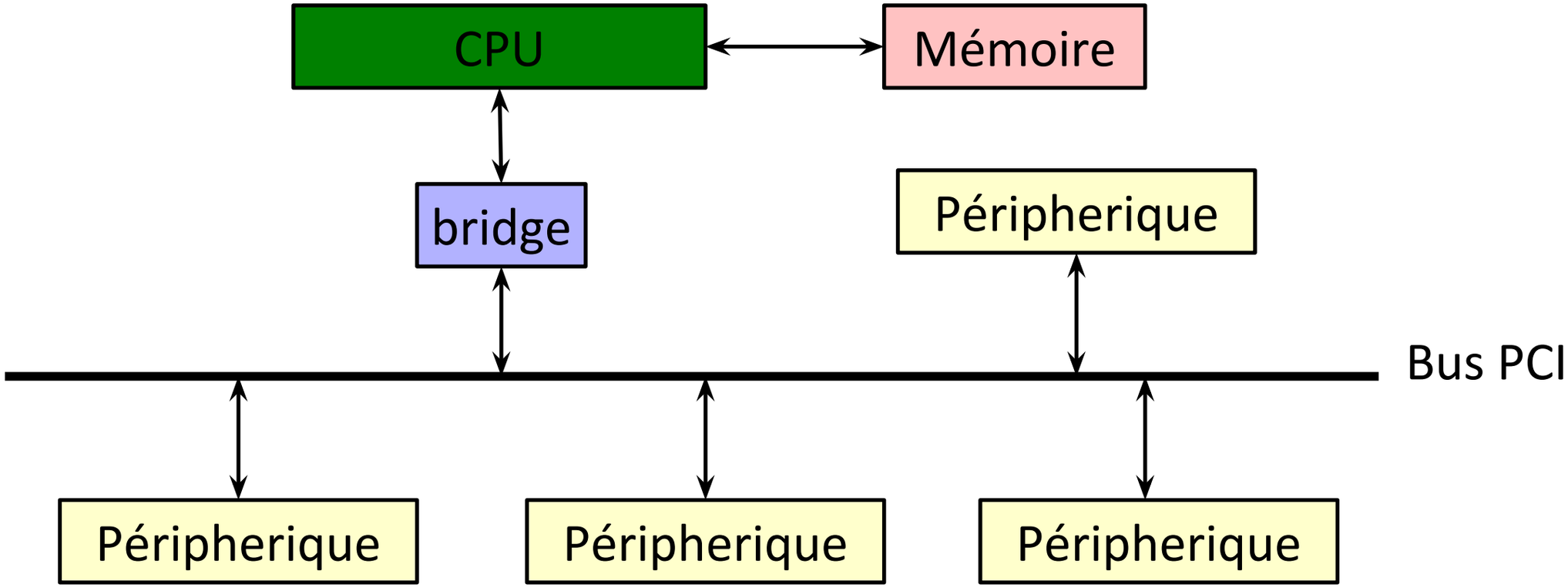
- **Bus partagé** : plusieurs composants sont connectés sur un même bus.
  - choix de préférence dans les ordinateurs anciens.
- **Bus point à point** : un bus relie deux composants.
  - choix de préférence dans les ordinateurs modernes.
- Dans un bus partagé :
  - un composant est le **maître** (*master*) du bus quand il initie un transfert de données.
  - un composant est **esclave** (*slave*) s'il ne reçoit que des données.
  - il faut un **mécanisme d'arbitrage** pour éviter que deux composants deviennent maîtres du bus au même temps (éviter les conflits).
  - **arbitrage centralisé** : un dispositif (**arbitre**) du bus décide qui peut devenir maître à un moment donné.
  - **arbitrage décentralisé** : les composants connectés au bus suivent un protocole pour décider qui peut devenir maître à un moment donné.
  - il faut assurer la **vivacité** : un composant ne doit pas attendre à l'infini pour pouvoir initier un transfert de données.
  - différents techniques (on ne les verra pas…).

Bus ISA et VL-Bus

- **Bus ISA** : *Industry Standard Architecture.*
  - créé en 1981 par IBM.
  - bus parallèle (8 ou 16 bits).
  - bus synchrone (4.77MHz - 6MHz - 8MHz).
  - bus partagé par au plus 6 composants.
  - survécut plusieurs années (compatibilité et rapidité satisfaisante avant les applications multimédias).
- **VL-Bus** : *Video Electronics Standards Association*.
  - créé en 1992 par VESA.
  - bus parallèle (32 bits).
  - bus synchrone (25 – 40 MHz).
  - bus partagé par au plus 3 composants.
  - aussi rapide que la CPU.
  - connexion directe avec la CPU….
  - ….mais limitation sur le nombre de composants.
  - utilisé surtout pour connecter des cartes graphiques.

PCI

- **PCI bus** : *Peripheral Component Interconnect*.
  - créé en 1992 par Intel.
  - bus parallèle (32 ou 64 bits).
  - bus synchrone (33MHz ou 66MHz).
  - bus partagé au plus par 5 composants.
  - utilise une puce (le **pont**, ou **bridge**) pour se connecter à la CPU.



PCI-X

- **PCI-X** : PCI eXtended.
  - similaire à PCI.
  - gestion des transferts de données différente.
  - plus rapide que PCI.

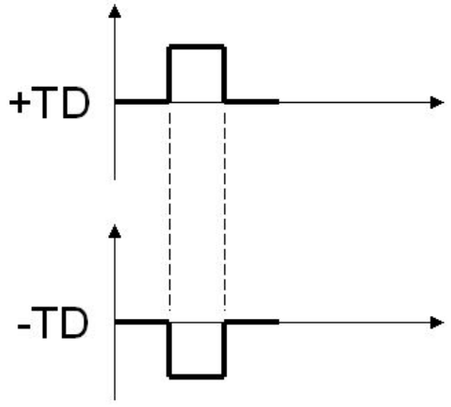
Bus	Vitesse du bus	Bande passante du bus	Nombre de composants
PCI 32-bit	33 MHz	132MB/s	4-5
PCI 32-bit	66 MHz	264 MB/s	1-2
PCI-X 32-bit	66 MHz	264 MB/s	4
PCI-X 32-bit	133 MHz	532 MB/s	1-2
PCI-X 32-bit	266 MHz	1.064 GB/s	1
PCI-X 32-bit	533 MHz	2.132 GB/s	1

pour les bus à 64 bits, il faut doubler les valeurs de la bande passante.

- Désavantages avec PCI et, en général, tous les bus parallèles.
  - Plus l'horloge est rapide, plus les interférences électromagnétiques sont fortes.
  - Les fils du bus n'ont pas la même longueur : délai de propagation (*time skew*).
  - Au final : un bus parallèle n'est pas plus rapide que un bus série (!)

Retour aux bus série

- Retour aux bus série (pensez à USB…).
  - transmission d'un bit à la fois.
- Plus simples à réaliser.
  - deux fils pour transmettre des données.
  - deux fils pour recevoir des données.
  - technique pour éliminer les interférences : **transmission différentielle.**

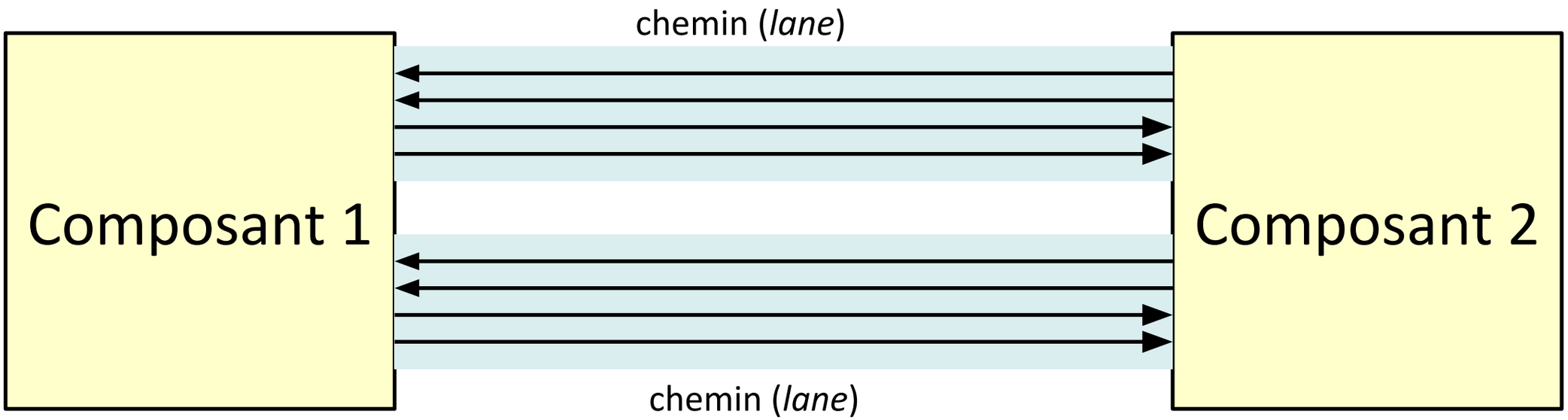


source : http://www.hardwaresecrets.com/everything-you-need-to-know-about-the-pci-express/2/

- Pas de délai de propagation : on peut incrémenter la fréquence de l'horloge.
- Communication **full-duplex**.

PCI Express

- **PCIe** : PCI Express.
  - créé en 2004 par Intel, Dell, HP et IBM.
  - Bus série.
  - Bus synchrone (2.5GHz – 16 GHz).
  - connexion point à point.



- PCIe x1, x2, x4, x8, x12, x16, x32 : nombre de chemins.
- Chaque « paquet » de données est envoyé sur plusieurs chemins.
- Chaque fil est indépendant des autres (!) : sinon on aurait une transmission parallèle.

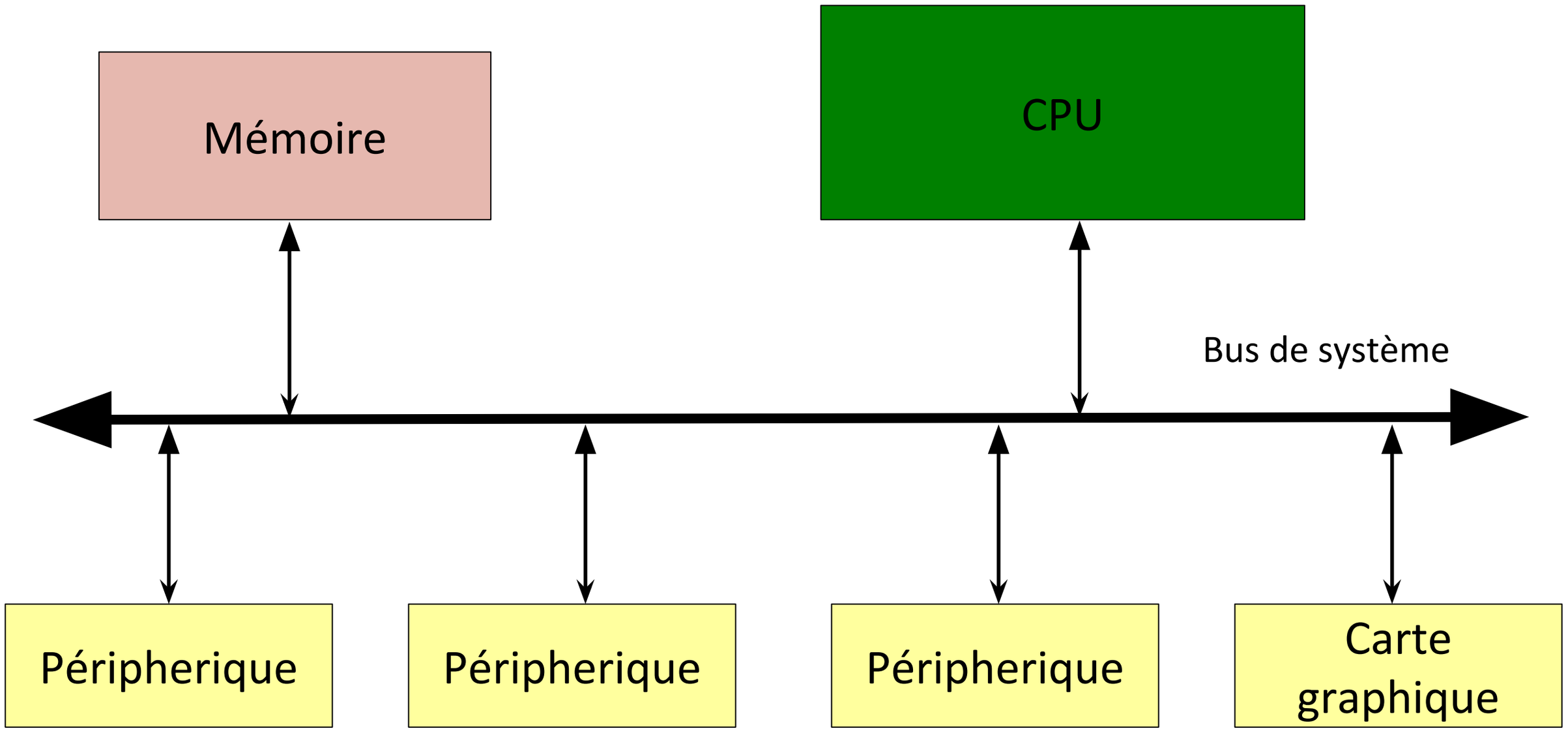
PCI Express

- Dans PCIe l'horloge est codée avec les données.
  - une partie des bits des données est utilisée pour l'horloge.
  - PCIe 1.0 et PCIe 2.0 utilisent le **codage 8b/10b**.
    - 8 bits de données sont codés sur 10 bits.
  - PCI 3.0, 4.0, 5.0 utilisent le codage **128b/130b**.
    - 128 bits de données sont codés sur 130.

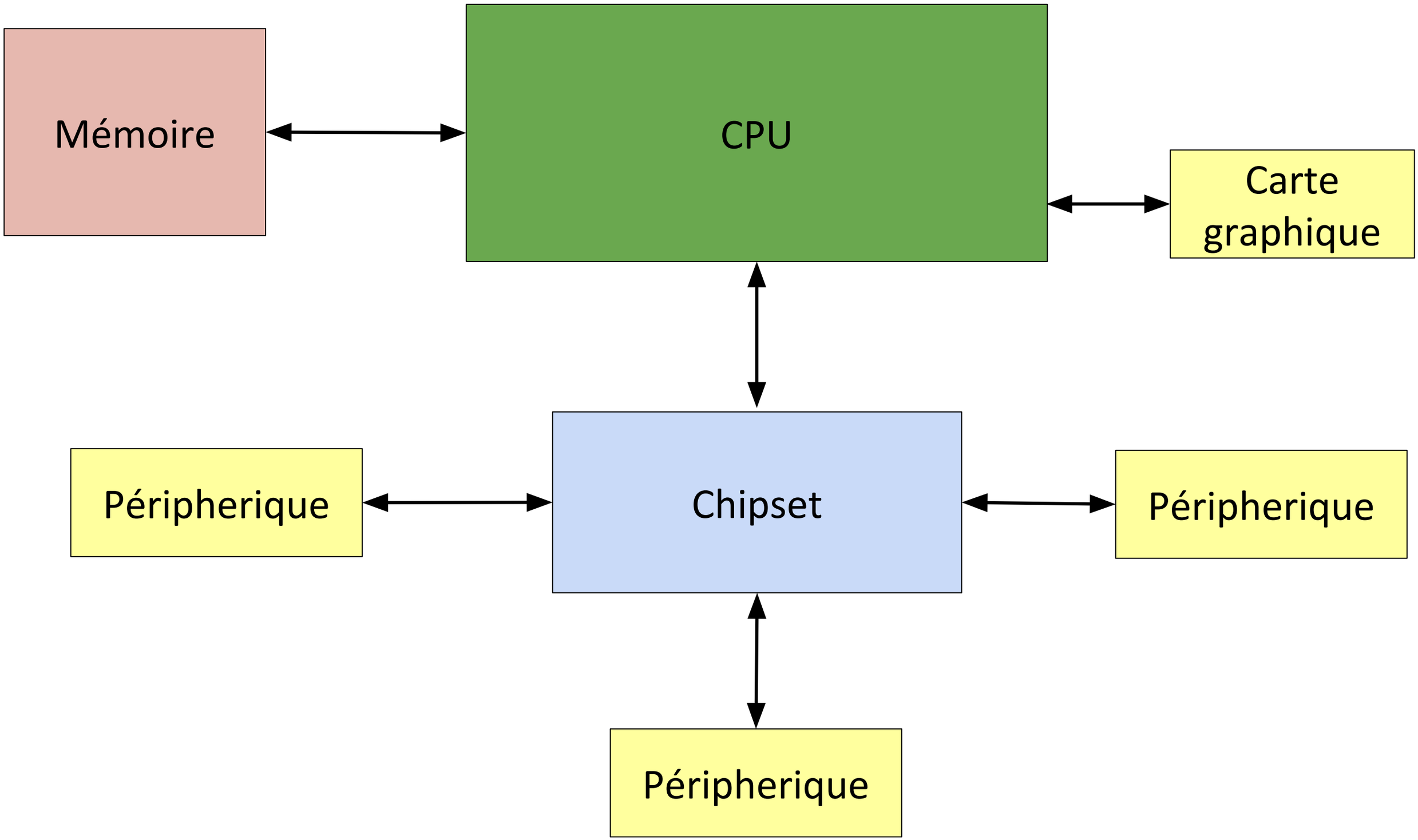
Bus	Codage	Horloge	Bande passante du bus (x1)	Année
PCIe 1.0	8b/10b	2.5 GHz	250 MB/s	2003
PCIe 2.0	8b/10b	5 GHz	500 MB/s	2007
PCIe 3.0	128b/130b	8 GHz	984.6 MB/s	2010
PCIe 4.0	128b/130b	16 GHz	1.969 GB/s	2017
PCIe 5.0	128b/130b	32 GHz (ou 25GHz?)	3,938 GB/s	2019

Avec PCIe 4.0 x16, nous obtenons une bande passante de 31.5 GB/s!

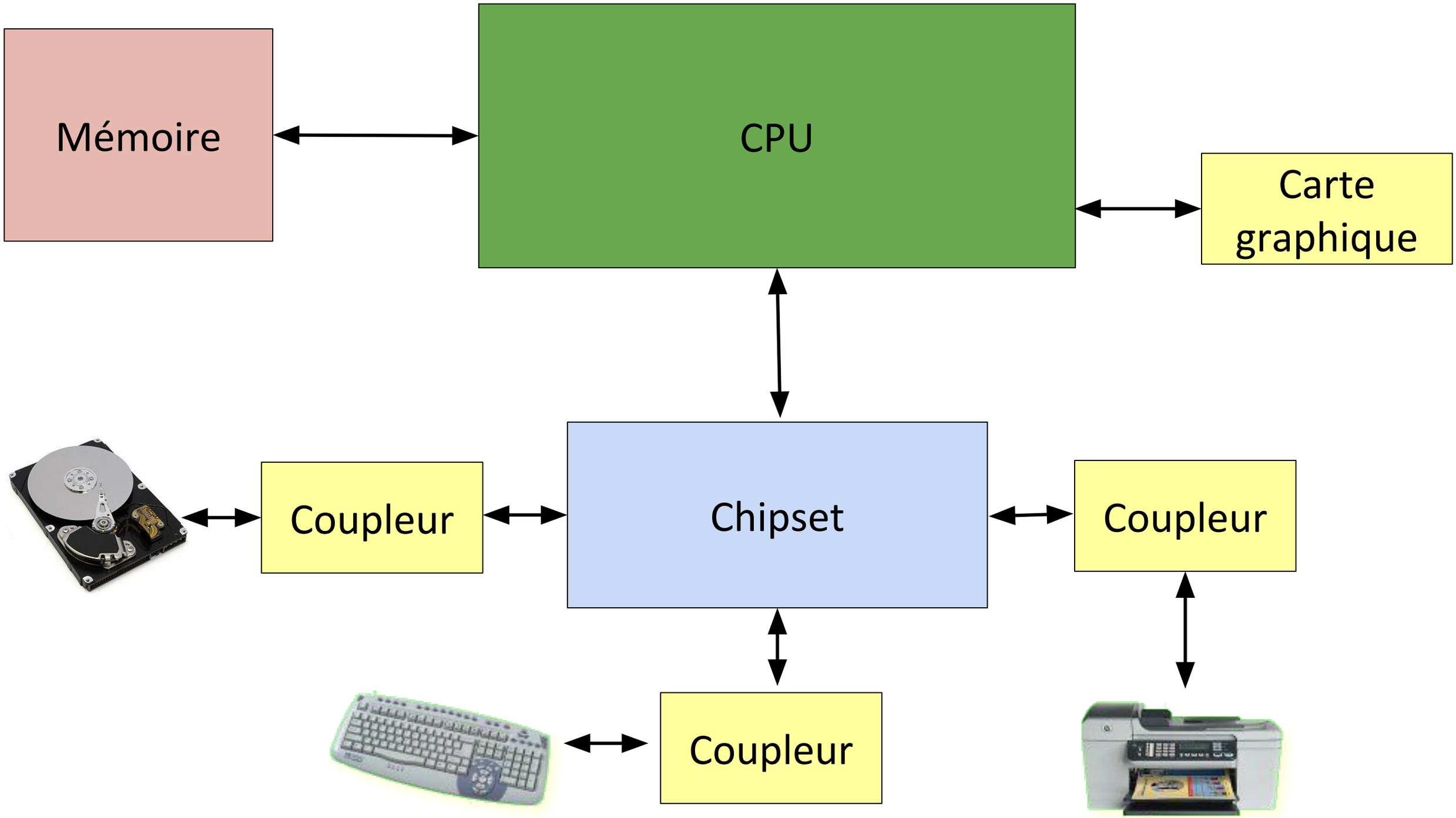
Structure d'un ordinateur ancien



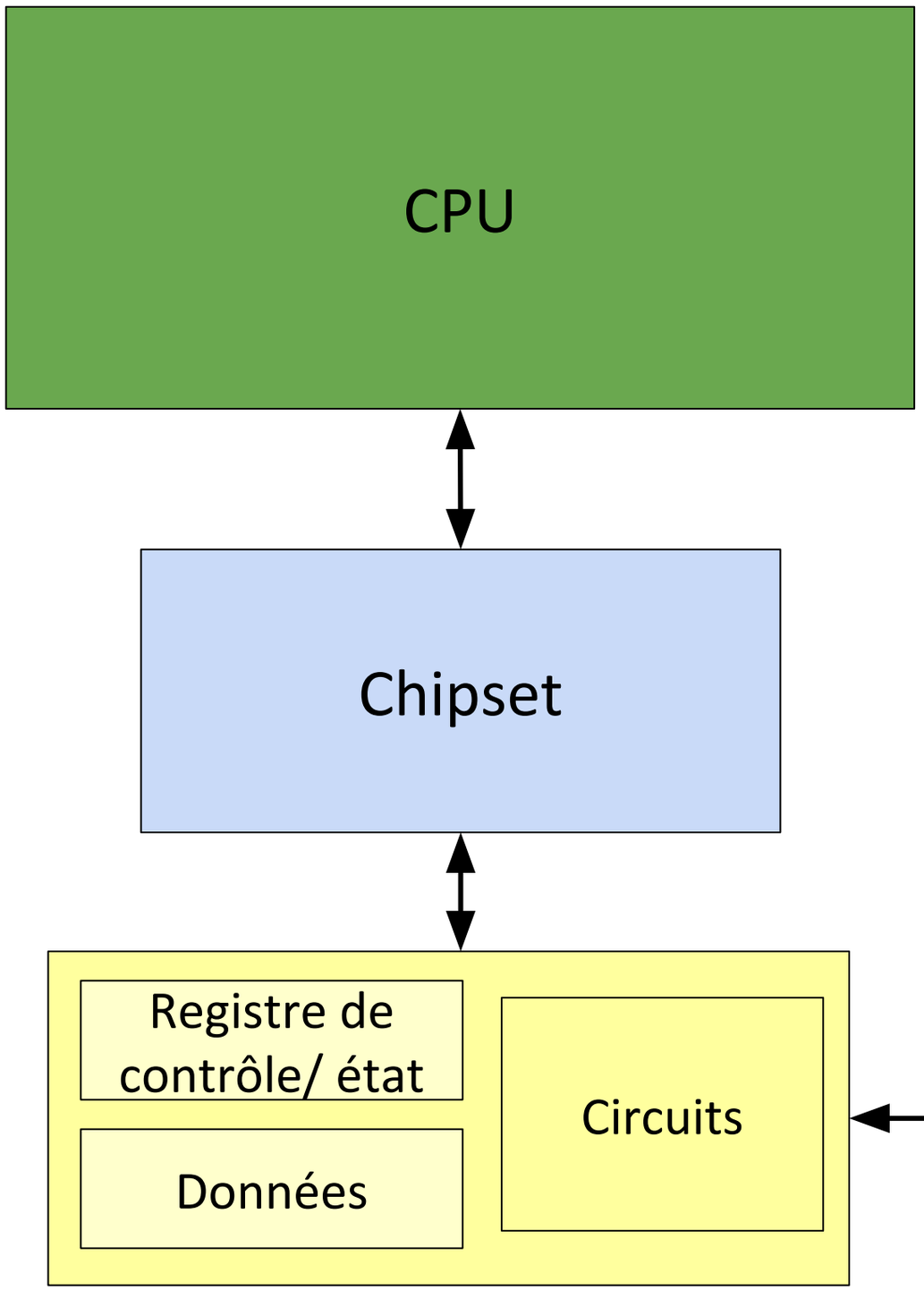
Structure d'un ordinateur moderne



Entrées/Sorties dans un ordinateur



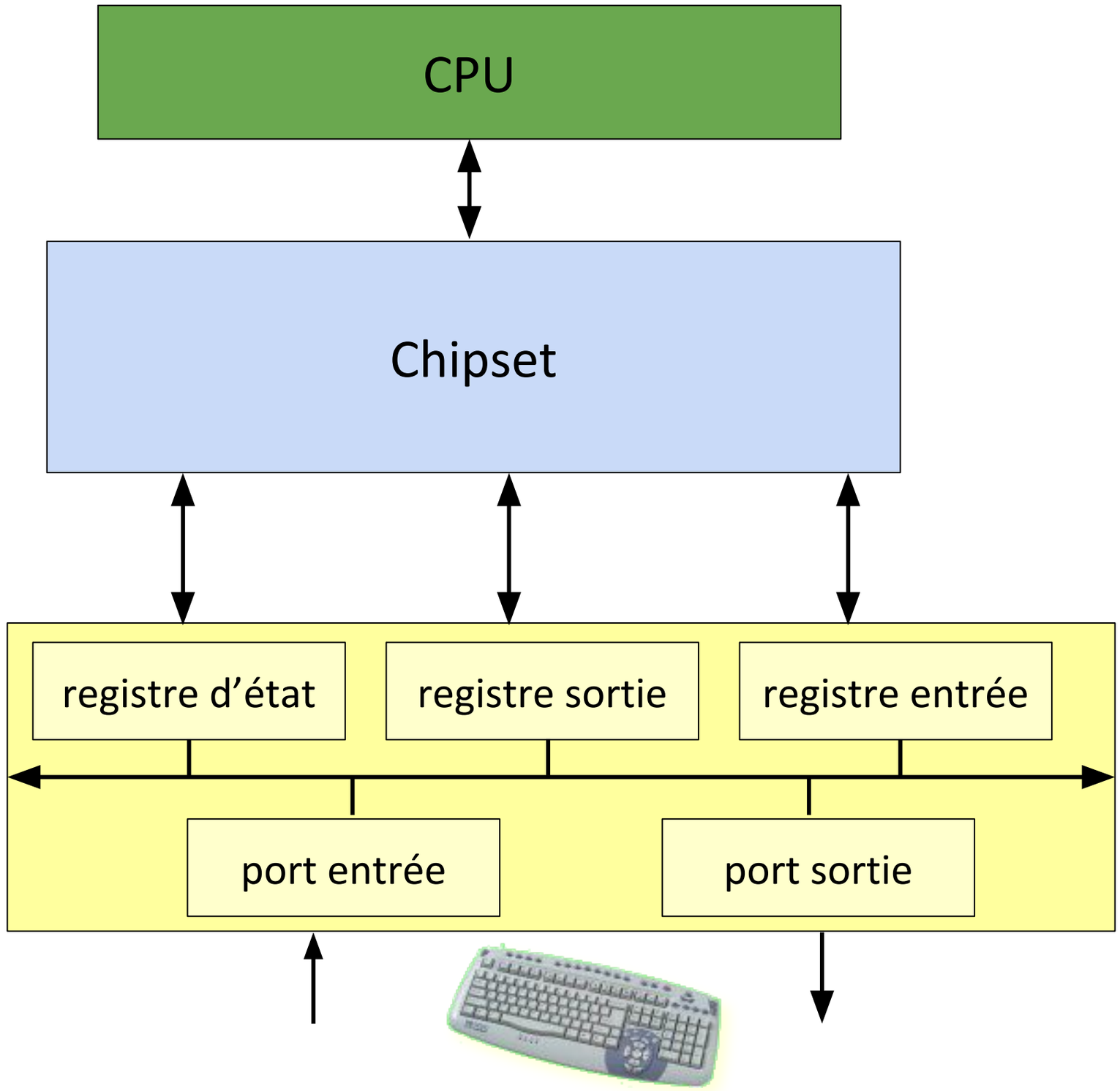
Entrées/Sorties dans un ordinateur



Entrées/Sorties mappées en mémoire (*Mémory-mapped I/O*) :

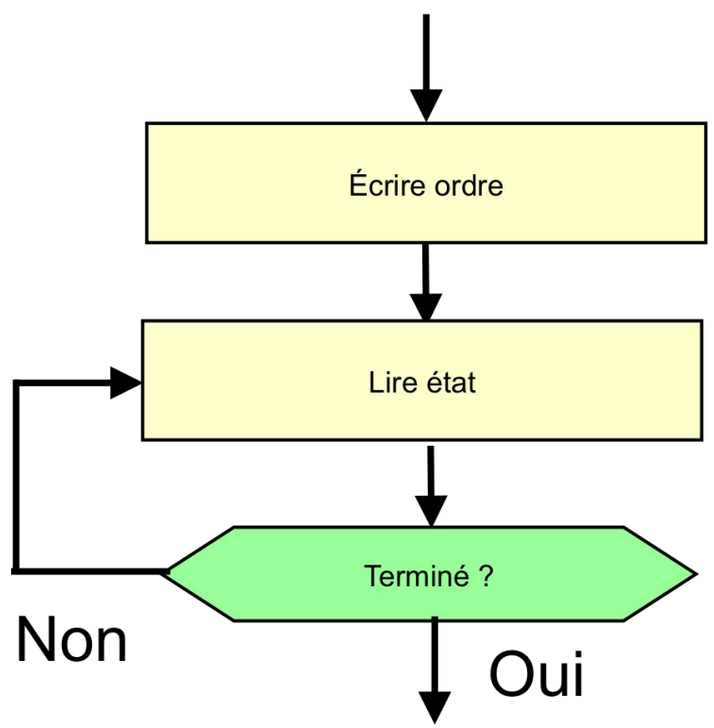
- Une partie des adresses mémoire indiquent des registres des coupleurs.
- Utilisation des instructions de lecture/écriture mémoire.

Exemple : clavier



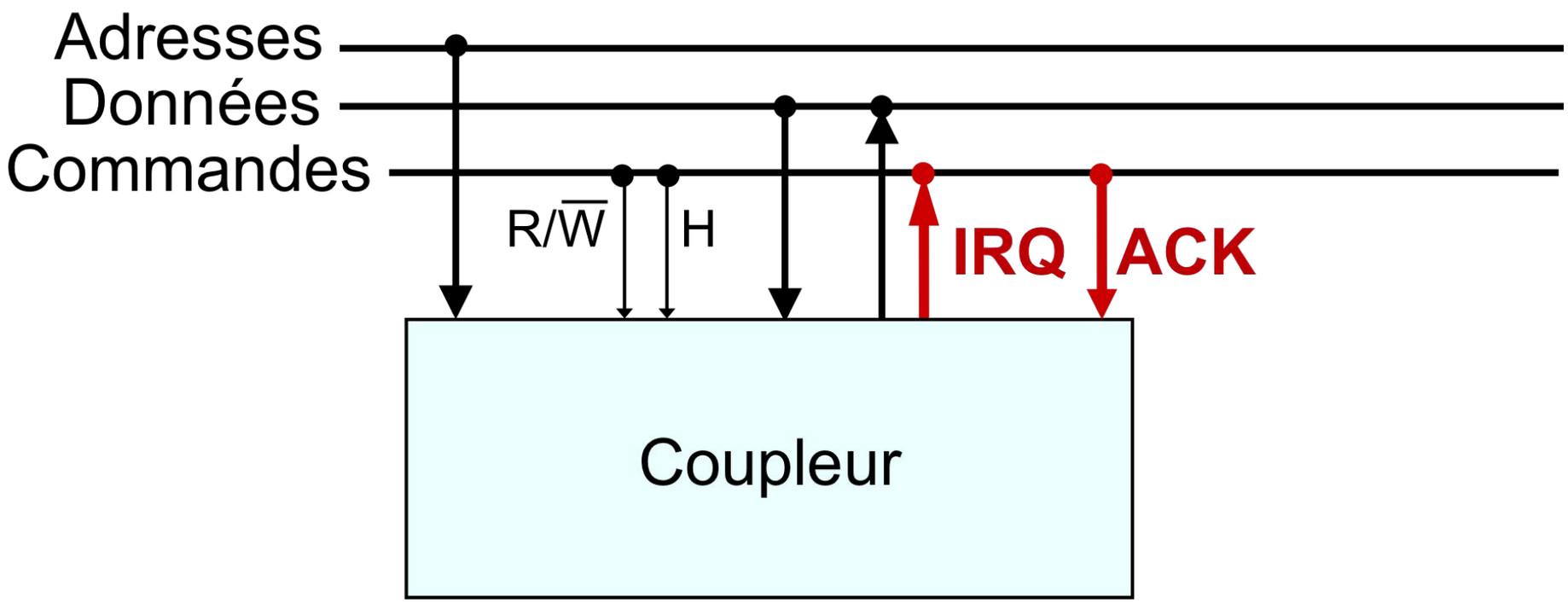
Interaction processeur – périphérique

- Un périphérique permet à l'utilisateur d'intéragir avec l'ordinateur.
  - clavier, souris….
- Les actions faites sur un périphérique donnent lieu à des événements.
  - appuyer sur une touche du clavier
- Un événement doit déclencher une réaction de l'ordinateur (du processeur)
  - visualiser sur l'écran le caractère correspondant à la touche appuyée.
- Le processeur ne sait pas quand un périphérique génère un événement.
- **Première solution** : attente active (polling)
- Le processeur contrôle périodiquement si le périphérique a généré un événement.
- Problème : le processeur perd son temps à attendre.



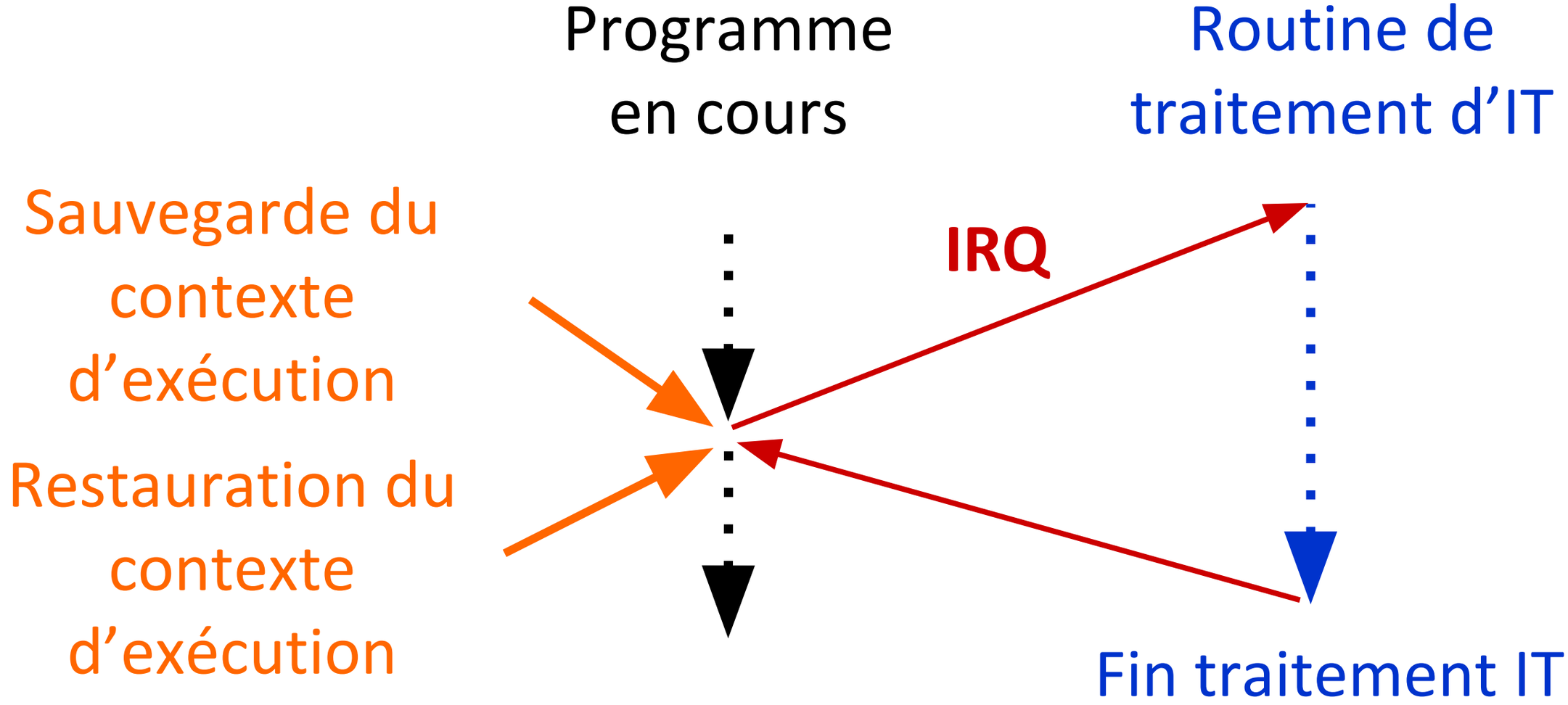
Interruptions

- Les interruptions sont un mécanisme permettant à un dispositif (tel qu'un périphérique) de forcer le processeur à interrompre l'exécution du programme en cours afin de s'occuper d'une autre tâche.
  - L'utilisateur appuie sur une touche, une interruption est déclenchée afin que le processeur lise le caractère et le visualise.
  - Le processeur demande des données au disque dur. Quand les données sont prêtes, le disque dur déclenche une interruption.
- Les interruptions sont implémentées au niveau matériel.



Prise en compte d'une interruption

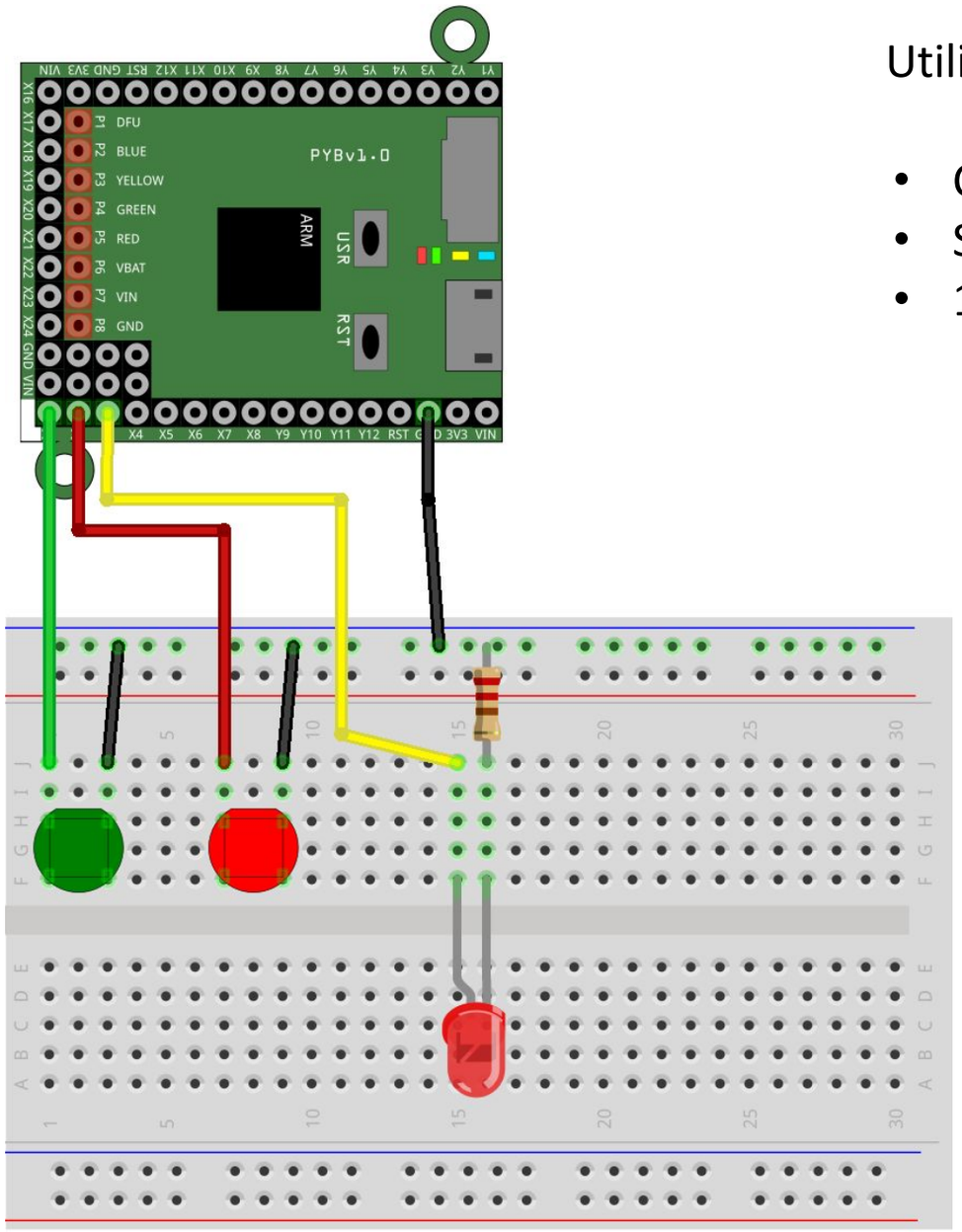
- A la fin de l'exécution de chaque instruction le processeur vérifie si le signal IRQ est activé.
- Si IRQ est activé et les interruptions ne sont pas **masquées**, il prend en compte l'interruption.
- Sauvegarde le **contexte d'exécution** (CO, registre état et autres registres utilisés par le programme en cours).
- Il charge le PC avec l'adresse de la routine de traitement d'interruption (**vecteur d'interruption**).



Timers

- Les **timers** sont des dispositifs qui permettent de compter des durées temporelles.
- Un timer permet de déterminer le lancement d'une tâche dans un délai précis, ou la répétition d'une tâche à intervalles réguliers.
- Le timer peut être programmé pour déclencher une interruption périodiquement
  - Lorsqu'une interruption se produit, une routine de traitement de l'interruption est appelée.

Pyboard



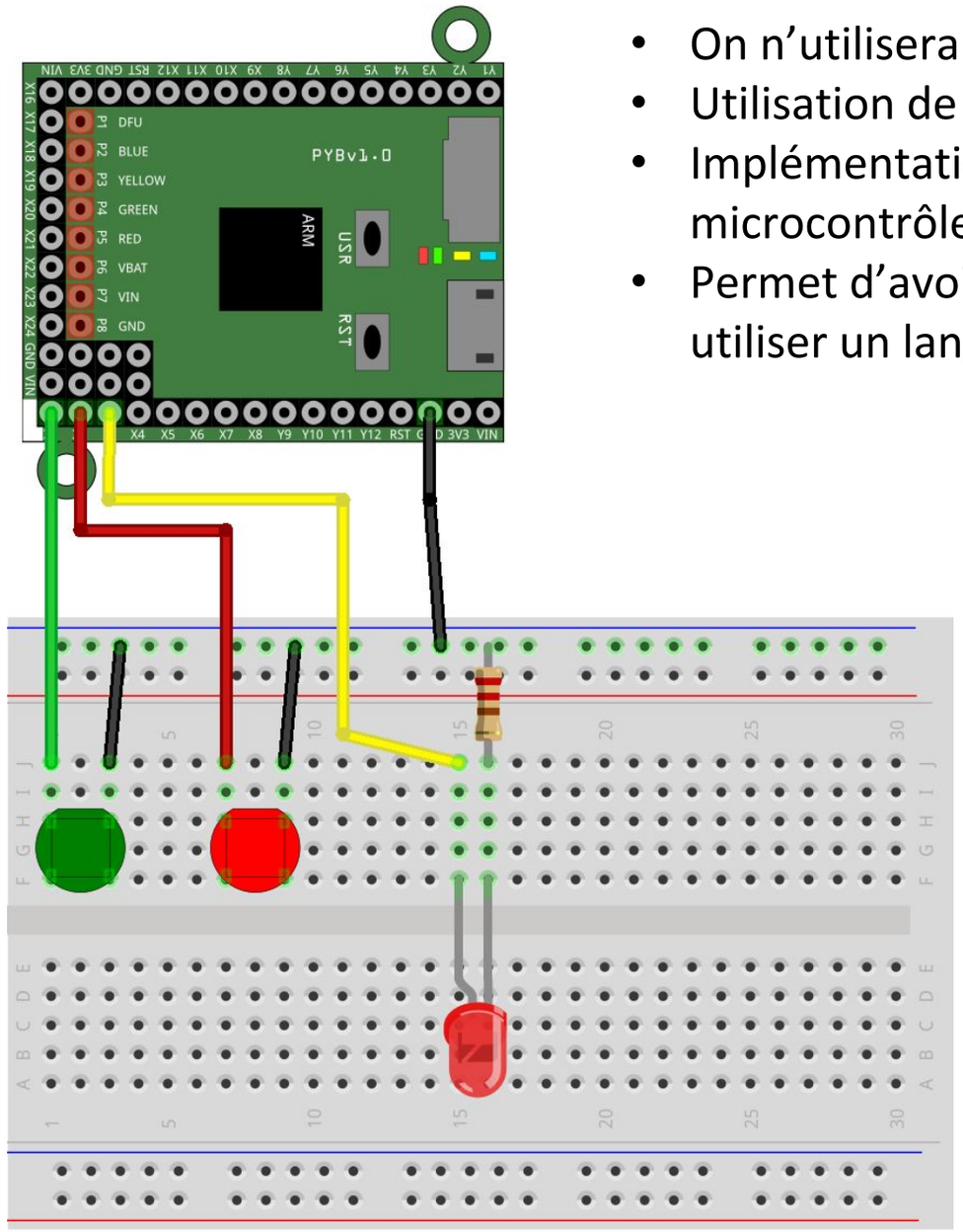
Utilise un microcontrôleur STM32F405RG

- CPU : ARM 32-bit 168 MHz.
- SRAM 192 KB.
- 1 MB mémoire Flash.

Exercices en faire en BE :

- Connecter deux boutons et une LED à la Pyboard.
- Ecrire un programme qui allume la LED lorsqu'on appuie sur le bouton vert et l'éteint lorsqu'on appuie sur le bouton rouge.
  - Par scrutation et par interruption…
- Faire clignoter la LED en utilisant un timer.

Pyboard – MicroPython



- On n'utilisera pas le langage d'assemblage!
- Utilisation de **MicroPython**.
- Implémentation d'un interpréteur Python (en C) qui s'exécute sur un microcontrôleur.
- Permet d'avoir le contrôle sur le matériel du microcontrôleur sans devoir utiliser un langage de bas niveau.

```
import pyb

bouton_vert = pyb.Pin("X1")
bouton_vert.init(pyb.Pin.IN, pyb.Pin.PULL_UP)
led.init(pyb.Pin.OUT_PP)

def interruption_vert(ligne):
  allume_led()

irq_vert = pyb.ExtInt(bouton_vert, pyb.ExtInt.IRQ_FALLING,
                      pyb.Pin.PULL_UP, interruption_vert)
```

CentraleSupélec

université PARIS-SACLAY

LABORATOIRE DE RECHERCHE EN INFORMATIQUE

CHAPITRE VII

Notions avancées sur les processeurs

CISC Vs. RISC

- Deux philosophies de réalisation d'un processeur.

- **CISC** : Complex Instruction Set Computer.
  - Une instruction peut exécuter plusieurs opérations à la fois.
  - Le jeu d'instructions d'un processeur CISC est très riche.
  - Idée : simplifier le codage en langage d'assemblage (combler le fossé sémantique).
  - Minimiser le nombre d'instructions d'un programme en langage d'assemblage.
  - Exemple de processeur CISC : Intel x86.

- **RISC** : Reduced Instruction Set Computer.
  - Le jeu d'instructions d'un processeur RISC est petit.
  - Idée : chaque instruction machine est exécutée en un seul cycle d'horloge.
  - Minimiser le nombre de cycles par instruction.
  - Exemple de processeur RISC : PowerPC

CISC Vs. RISC

- On veut multiplier deux valeurs a, b qui sont stockées en mémoire.
- Un processeur CISC aurait une instruction MULT a, b qui :
  - cherche les valeurs en mémoire.
  - les sauvegarde dans deux registres du processeur.
  - applique la multiplication.
  - sauvegarde le résultat en mémoire.
- Sur un processeur RISC, la même opération serait exécutée avec plusieurs instructions.
  - LOAD r1, a
  - LOAD r2, b
  - PROD r2, r1, r2
  - STORE r2, a

CISC Vs. RISC

- À première vue, l'approche CISC semble plus rapide.
  - une seule instruction.
  - moins de mémoire RAM nécessaire.
  - compilation d'un langage de haut niveau plus facile.
- Mais la structure d'un processeur CISC est plus complexe.
  - le séquenceur est généralement micro-programmé.
- L'instruction CISC nécessite de plusieurs cycles d'horloge.
- Chaque instruction RISC nécessite d'un seul cycle d'horloge.
  - même temps d'exécution qu'un programme CISC
- La structure d'un processeur RISC est plus simple.
  - le séquenceur peut être cablé (exécution plus rapide).
  - pipelining possible.
- Un programme RISC occupe plus de mémoire.
  - mais le coût de la mémoire ne cesse de baisser....

Caractéristiques d'un processeur RISC

- Chaque instruction est exécutée en un cycle d'horloge.

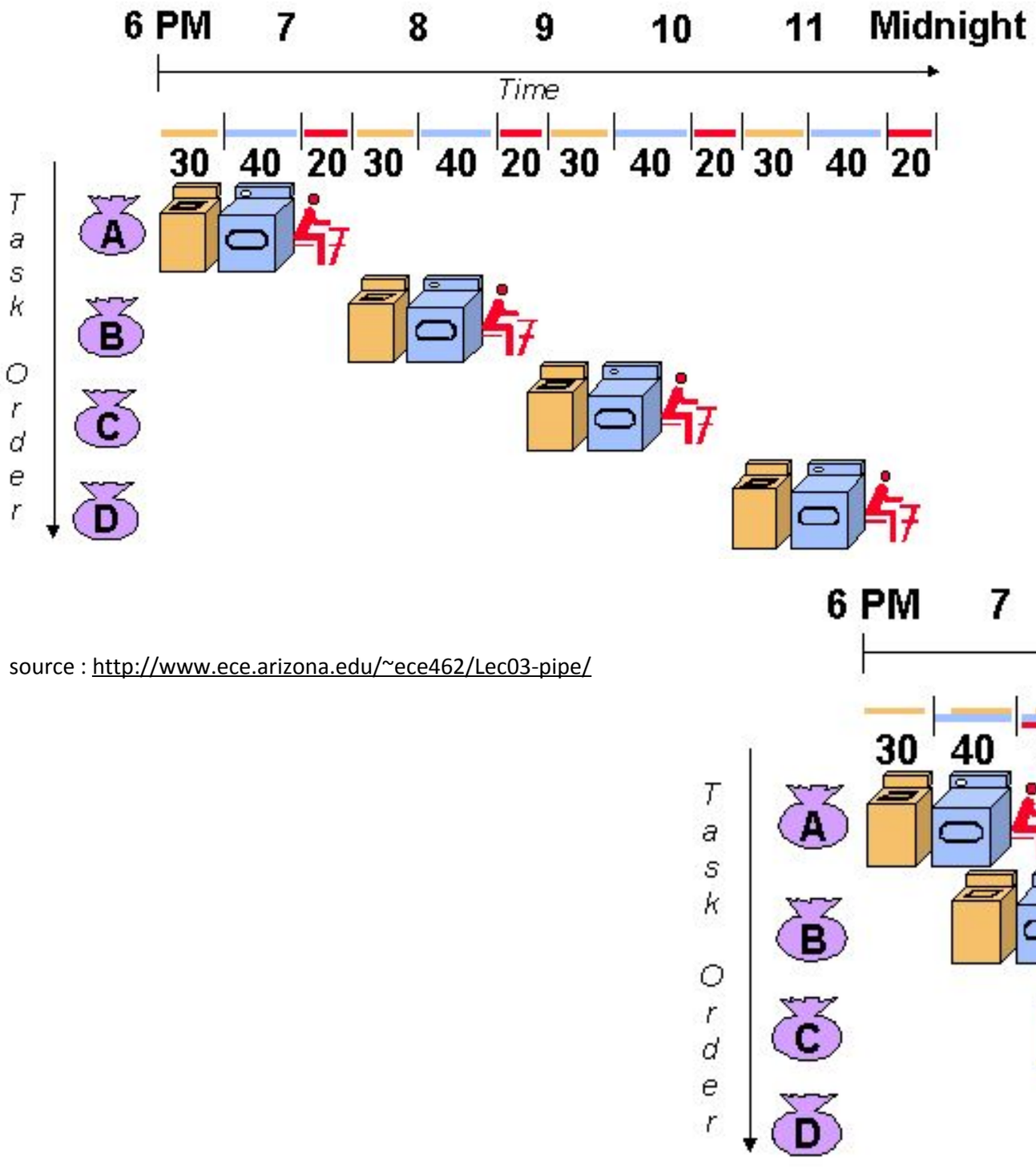
- Beaucoup de registres.
  - pour minimiser les interactions avec la mémoire.
  - dans un processeur CISC moins de registres.

- Format des instructions fixe.
  - le décodage est plus simple, donc plus rapide.
  - dans un processeur CISC les instructions ont un format variable.

- Modes d'adressage limités.

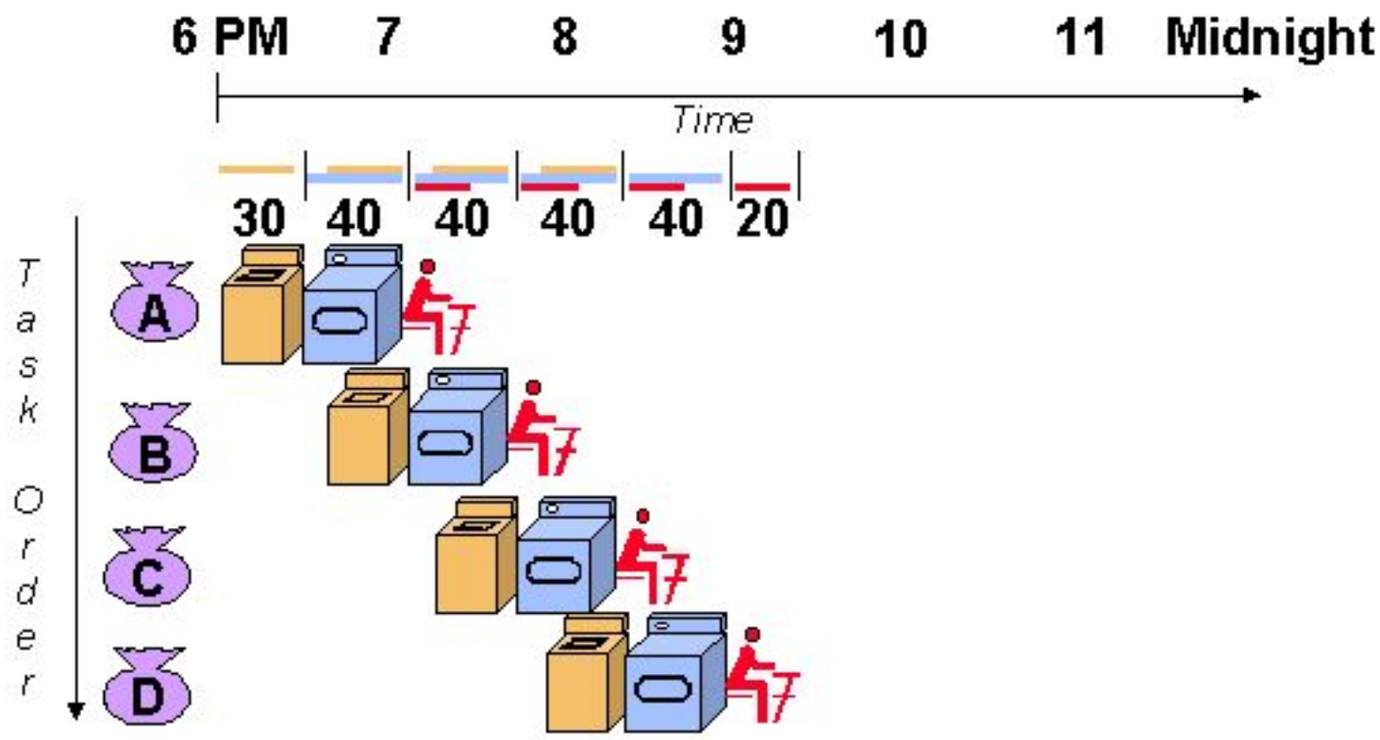
- Séquenceur cablé.

RISC – Pipelining



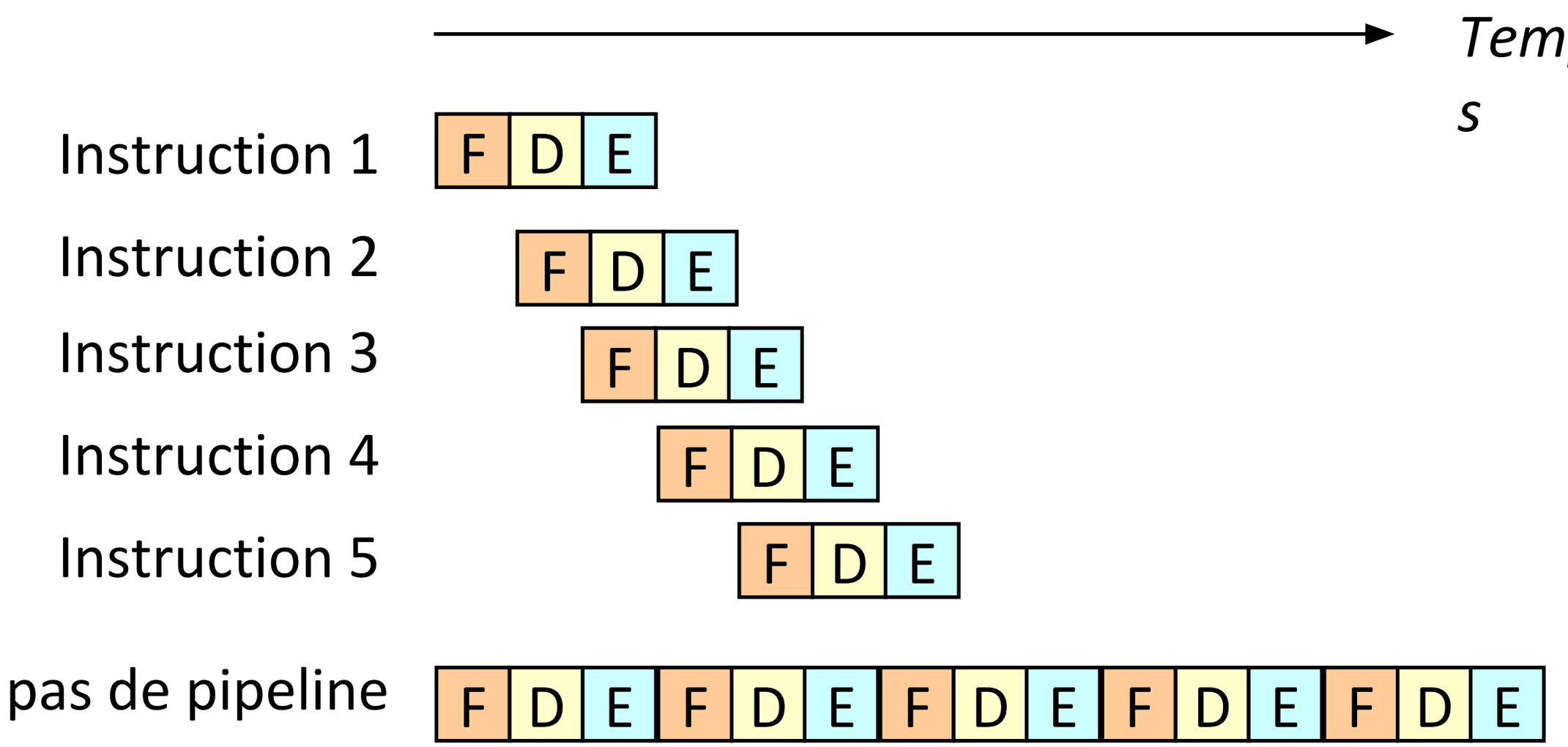
Analogie de la lessive

source : http://www.ece.arizona.edu/~ece462/Lec03-pipe/



RISC – Pipelining

- La phase de fetch est une source de ralentissement de la vitesse d'exécution d'une instruction.
- **Pipelining** : commencer le fetch de l'instruction suivante lorsqu'on décode l'instruction courante



RISC – Pipelining

…

.

b end

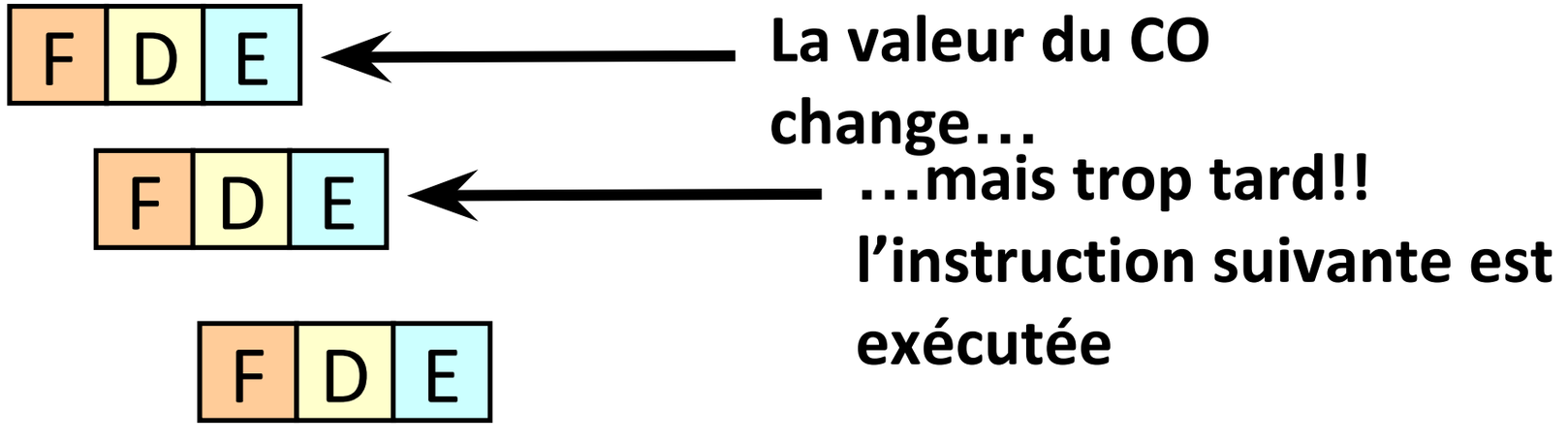
ldr r0, r1

mov r2, r0

ldr r1, r2

…

.



RISC – Pipelining

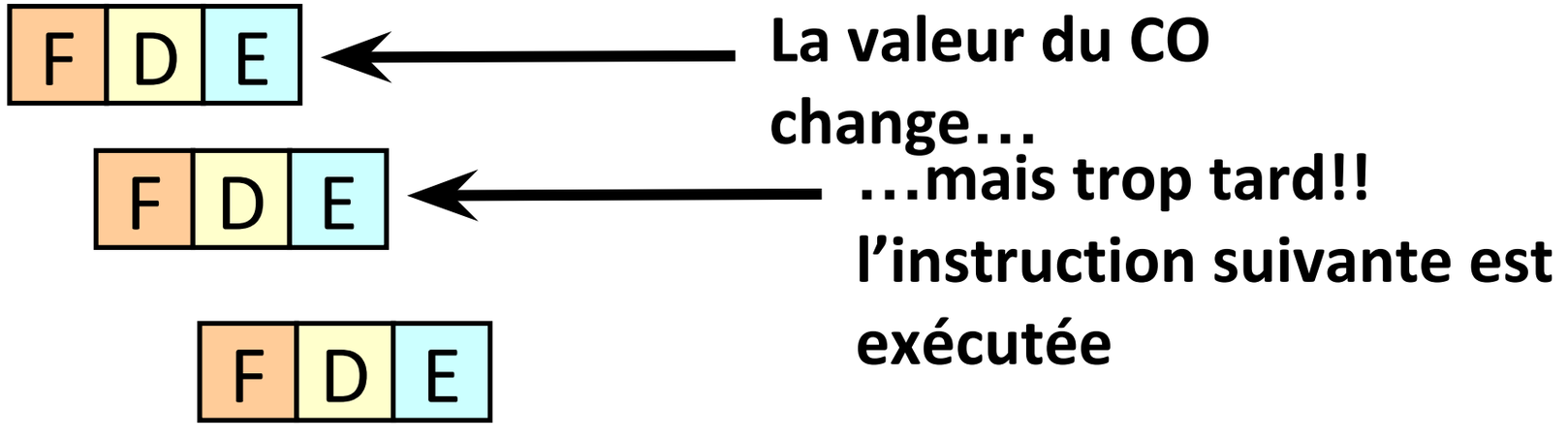
…

.

b end

ldr r0, r1

mov r2, r0



ldr r1, r2

…

.

- Une technique appelée « prédiction de branchement» (*branch prediction*) est utilisée.
- Le processeur fait un pari sur le résultat d'un branchement.
- Si son pari est correcte, l'exécution procède.
- Sinon il faut éviter que les instructions récupérées modifient les registres.
- Différentes techniques de branchement (statiques ou dynamiques).

RISC – Pipelining

…

ldr r0, r2

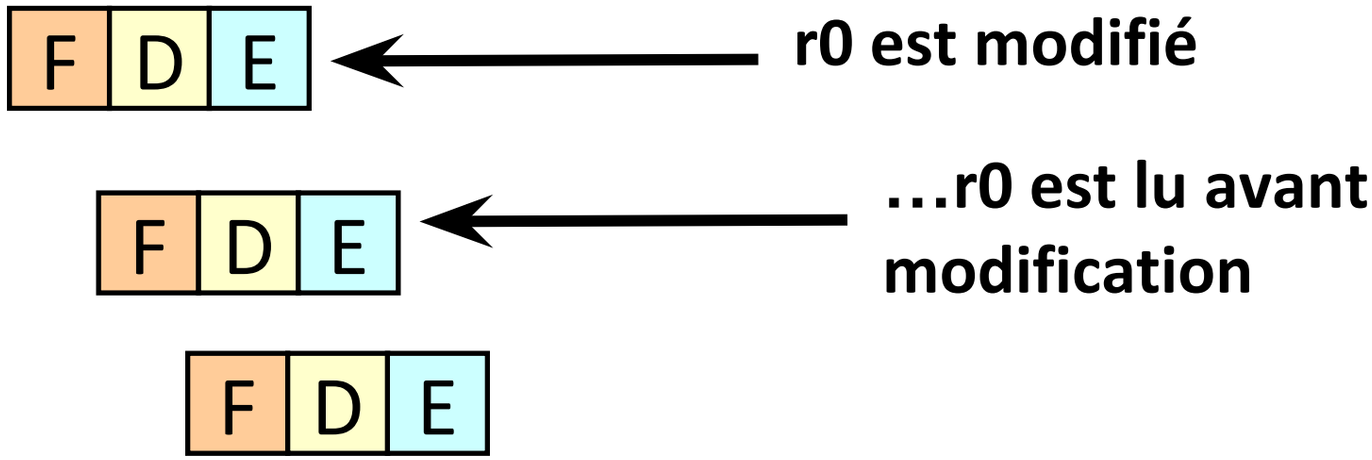
add r0, r1, r0

mov r5, r6

mov r3, r4

…

.



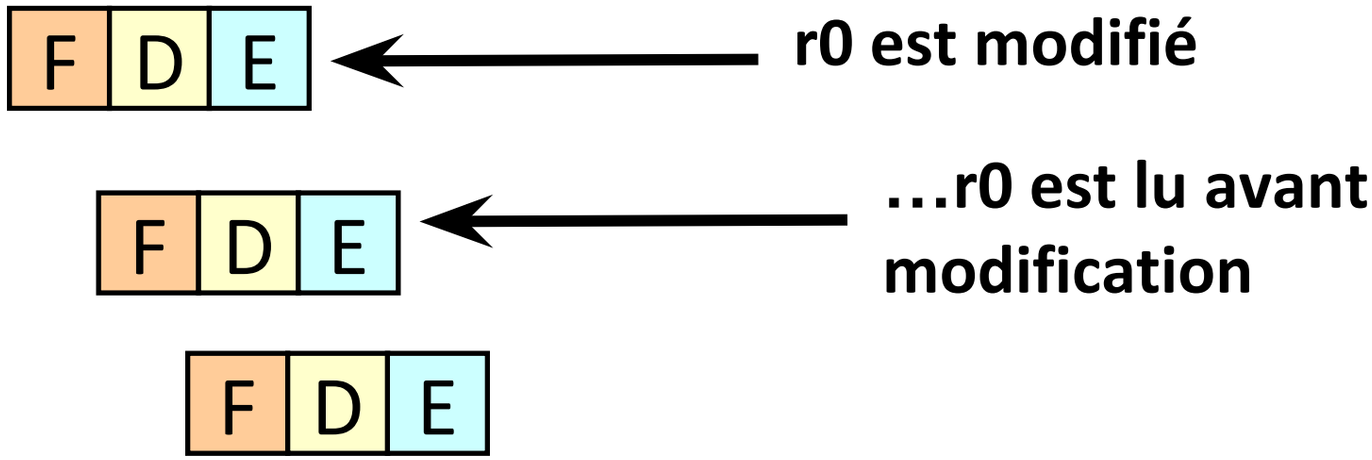
RISC – Pipelining

…

ldr r0, r2

mov r5, r6

mov r3, r4



add r0, r1, r0

…

- Une technique appelée « ordonnancement du code » (*code reordering*) est utilisée.
- C’est le compilateur qui s’en charge.
- Le compilateur peut détecter les dépendances dans le code.

Machine monoprocesseur

- Les ordinateurs anciens avaient seulement un processeur.

- Un processeur peut exécuter un seul programme à la fois.
  - le processeur a un seul compteur ordinal (CO).

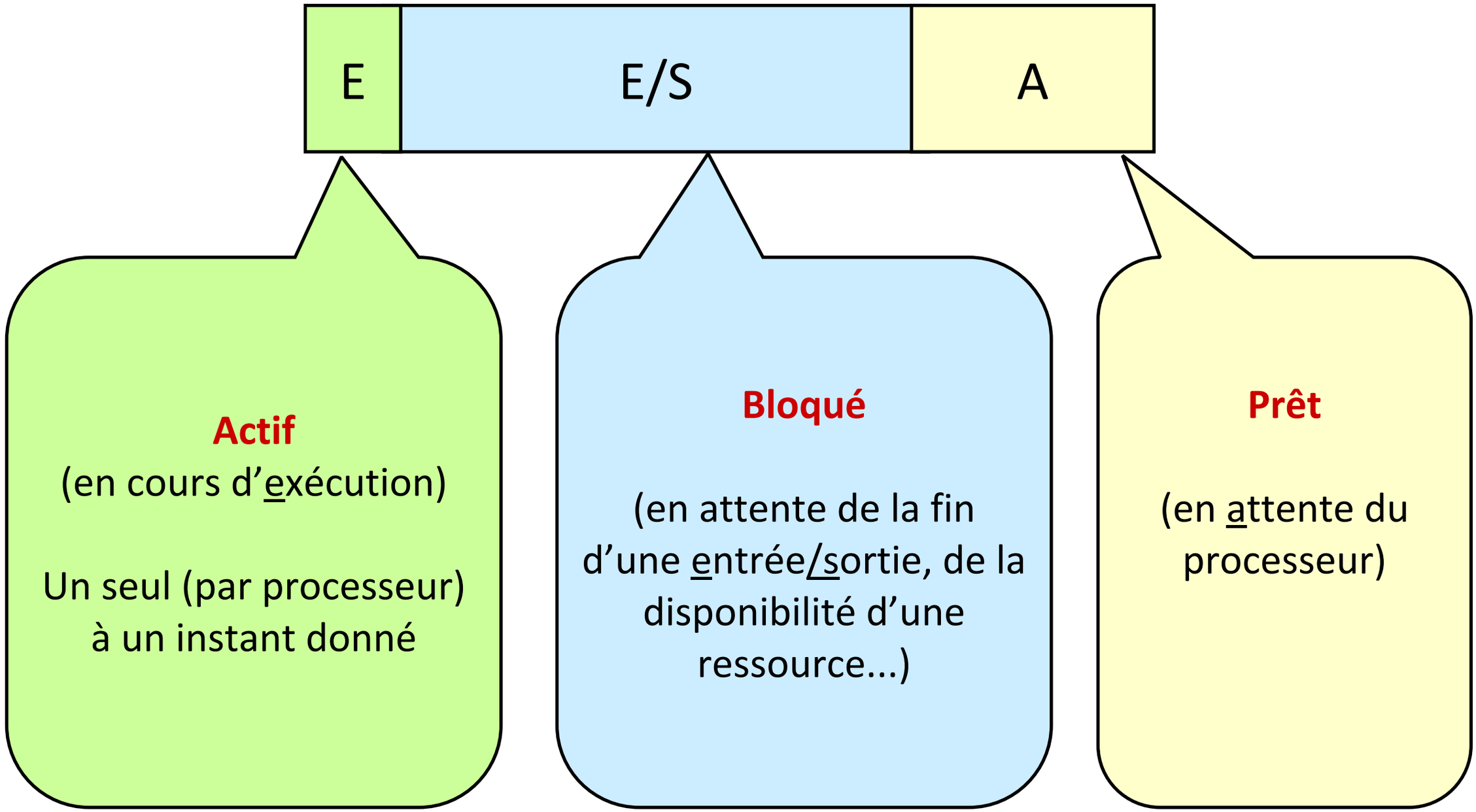
- Pourtant on peut lancer plusieurs applications même sur les ordinateurs monoprocesseur.
  - écrire un texte et écouter la musique.

- Comment un ordinateur monoprocesseur arrive à gérer plusieurs applications?

- La réponse est le **système d'exploitation**.

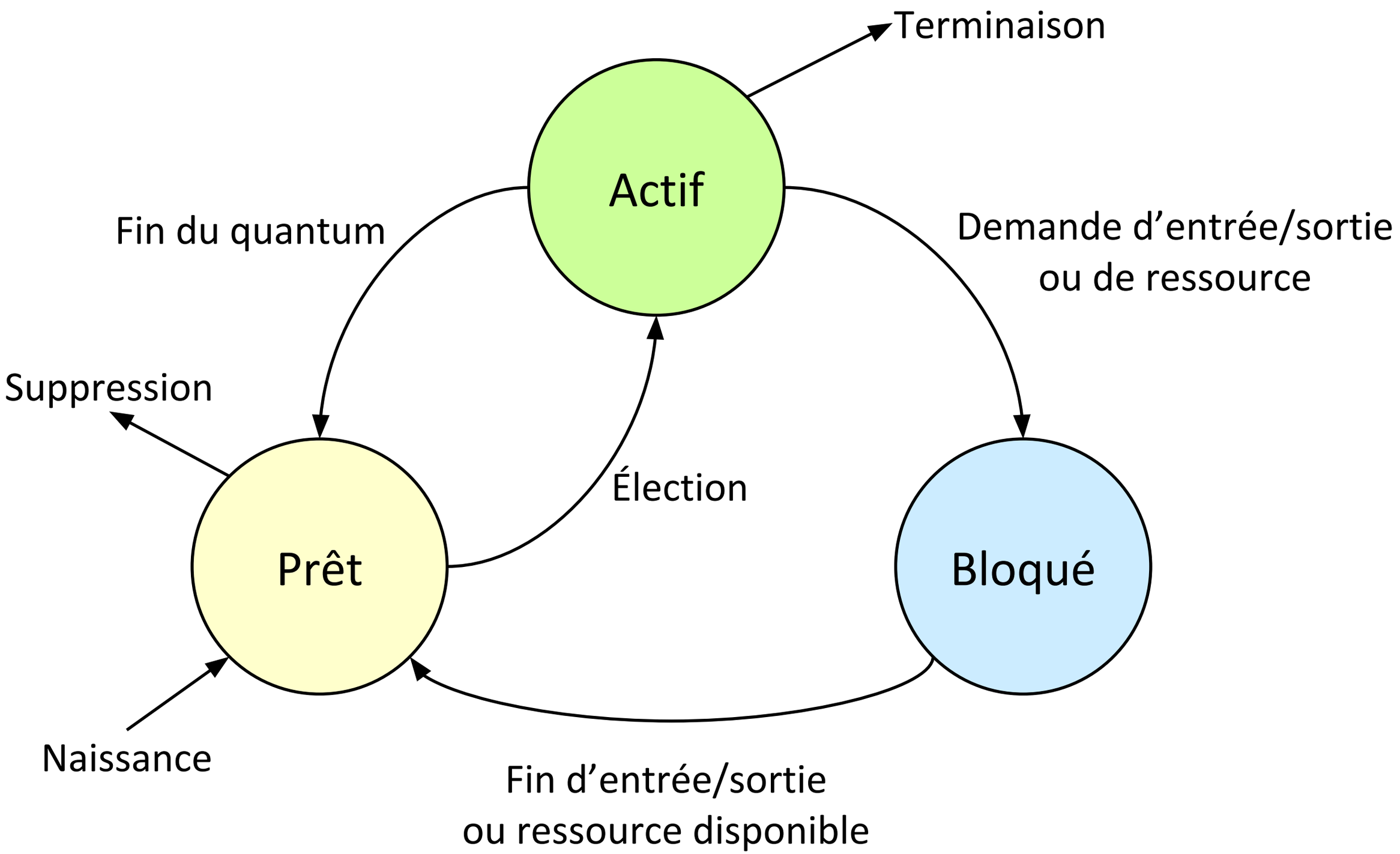
Machine monoprocesseur

Etats d'un processus (programme en exécution).

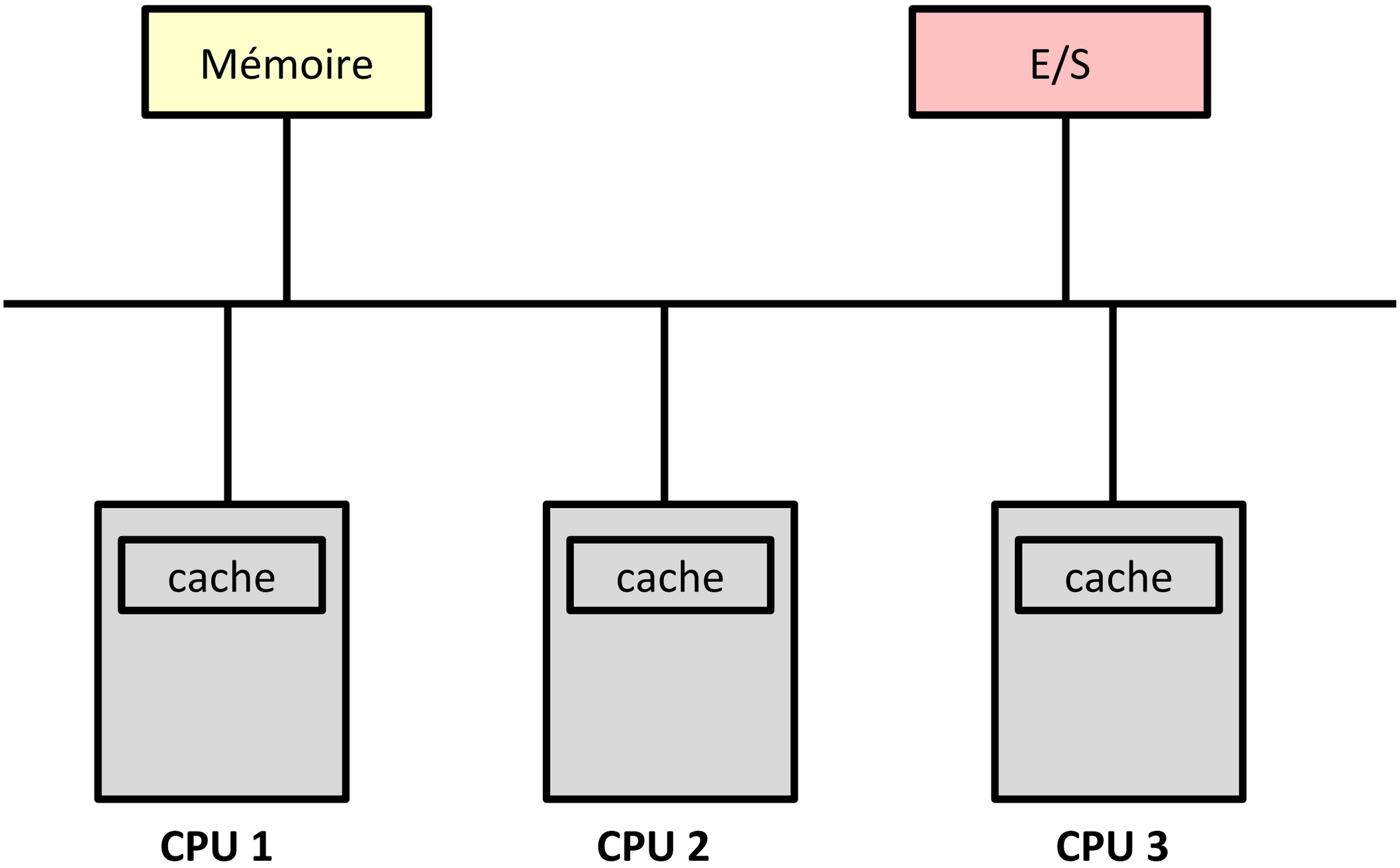


L'ordonnanceur (*scheduler*) gère les changements d'états

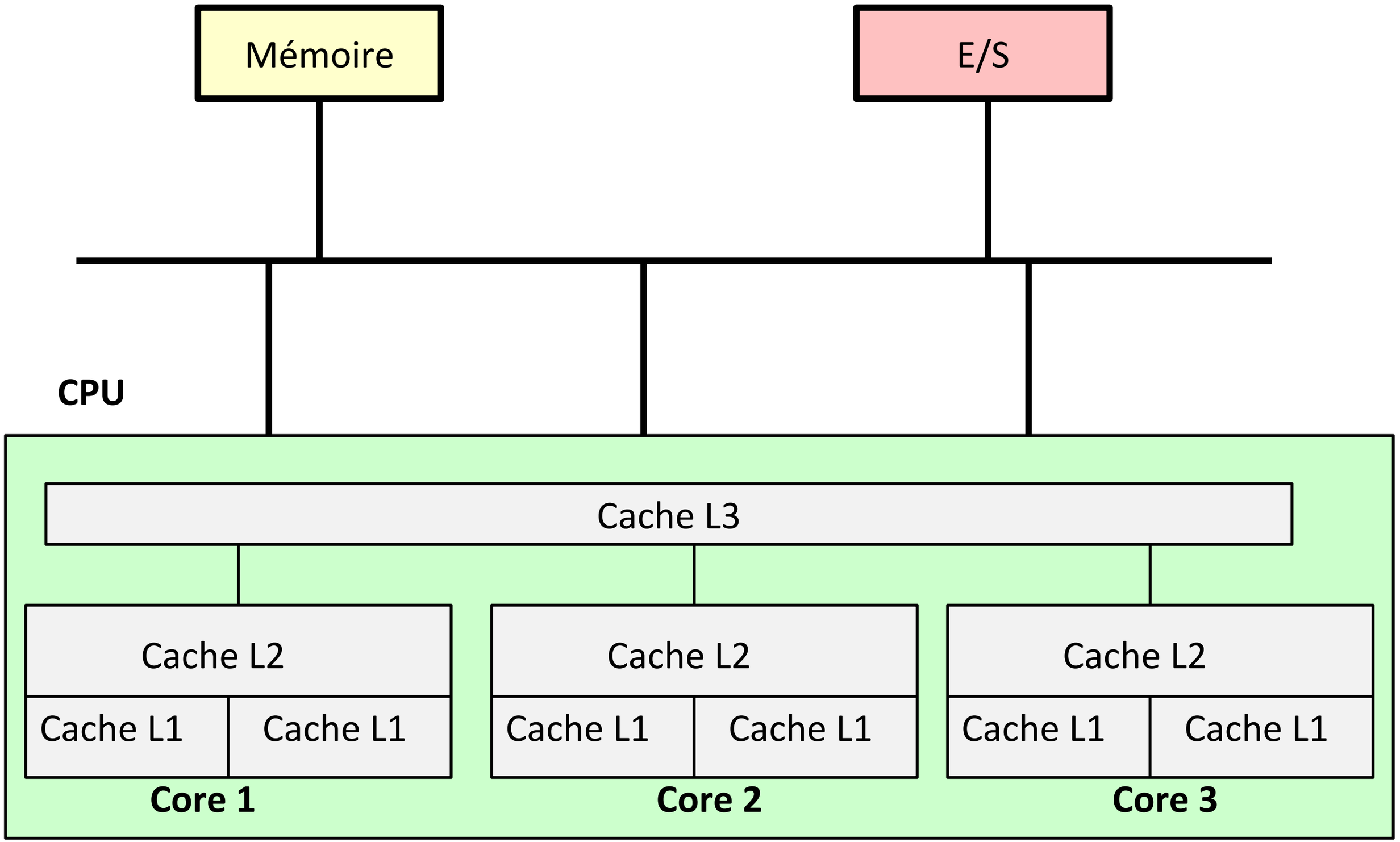
Machine monoprocesseur



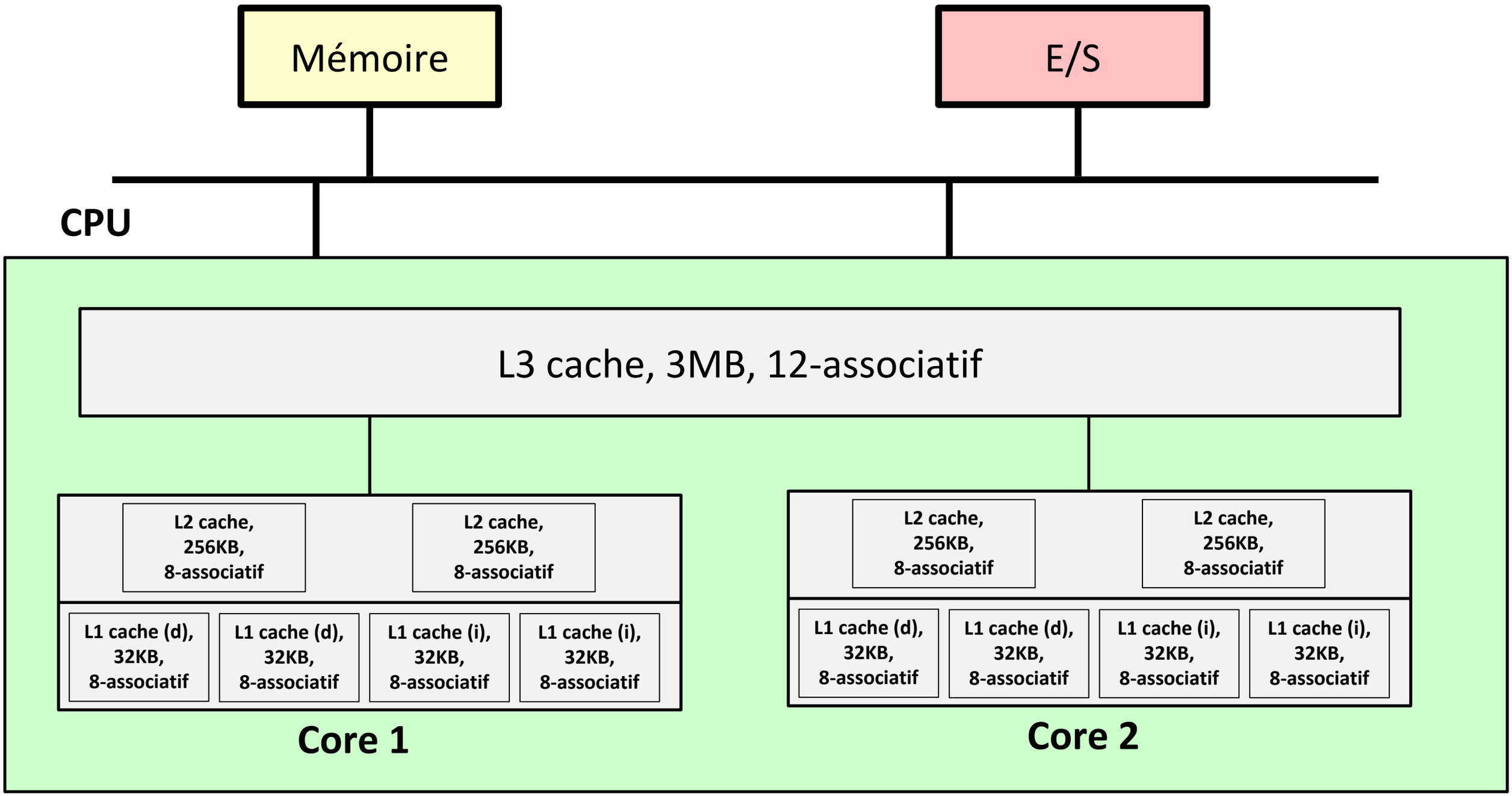
Machine multiprocesseur



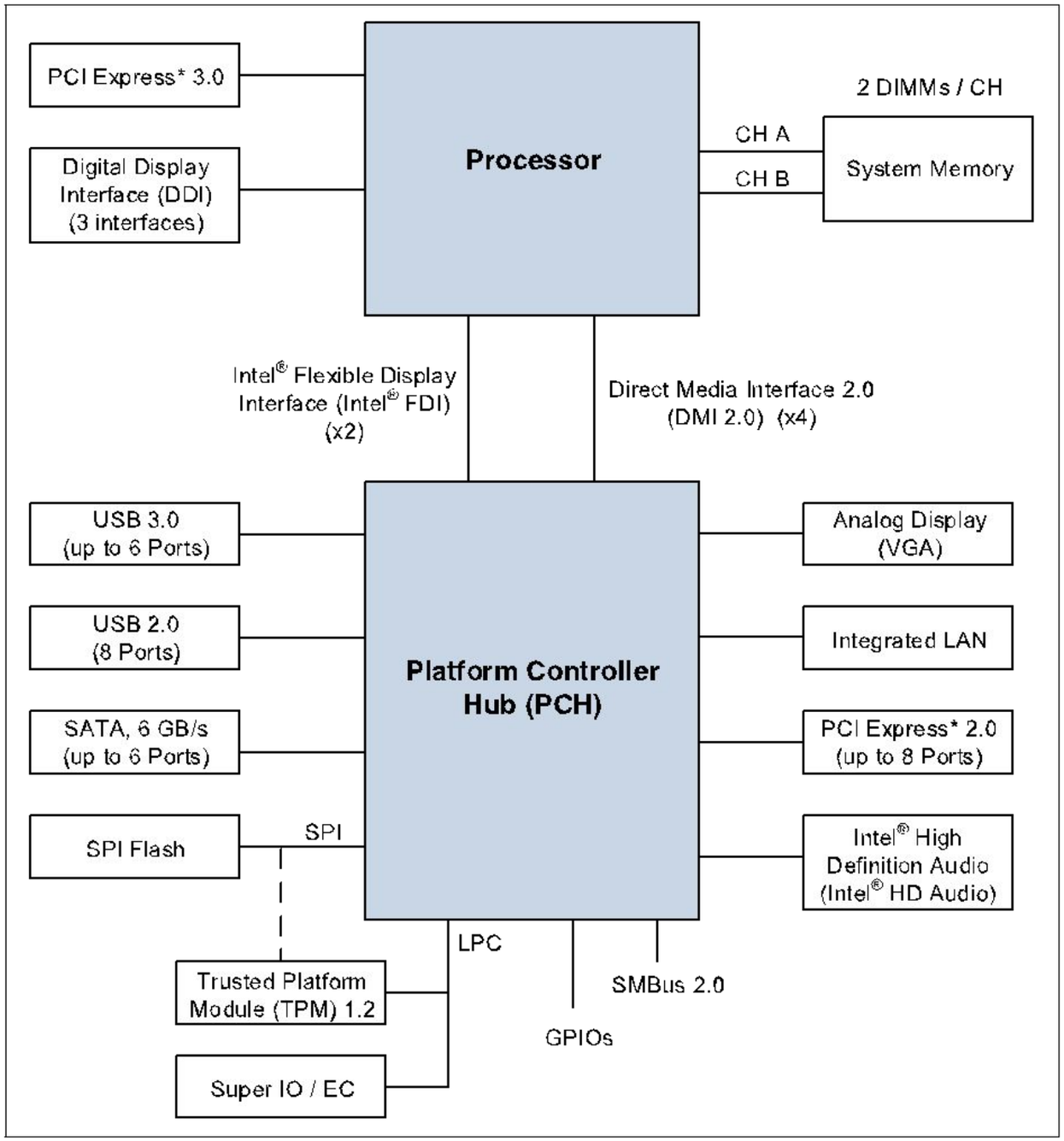
Processeur multi-cœur



Exemple : Intel Core i5-4278U (Haswell)

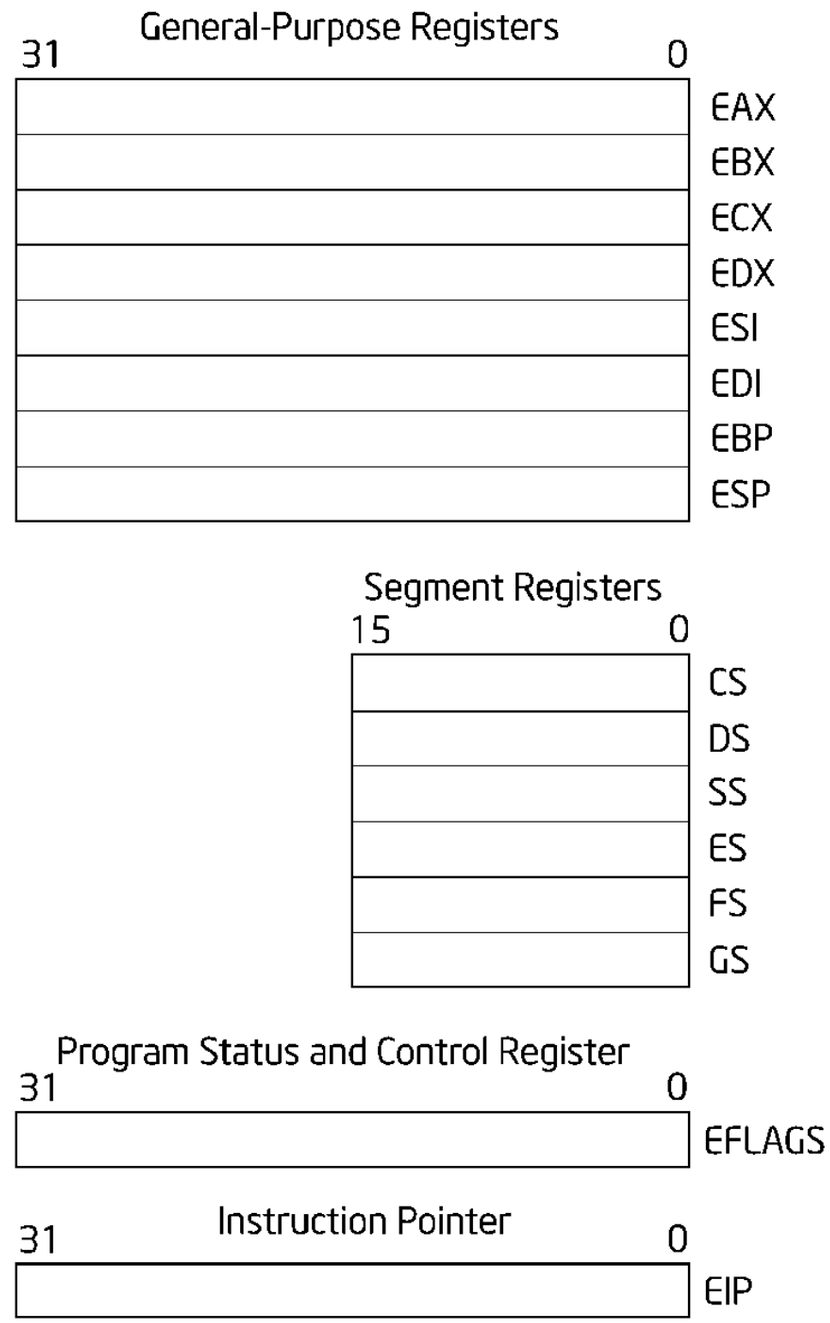


Exemple : Intel Core i5-4278U (Haswell)



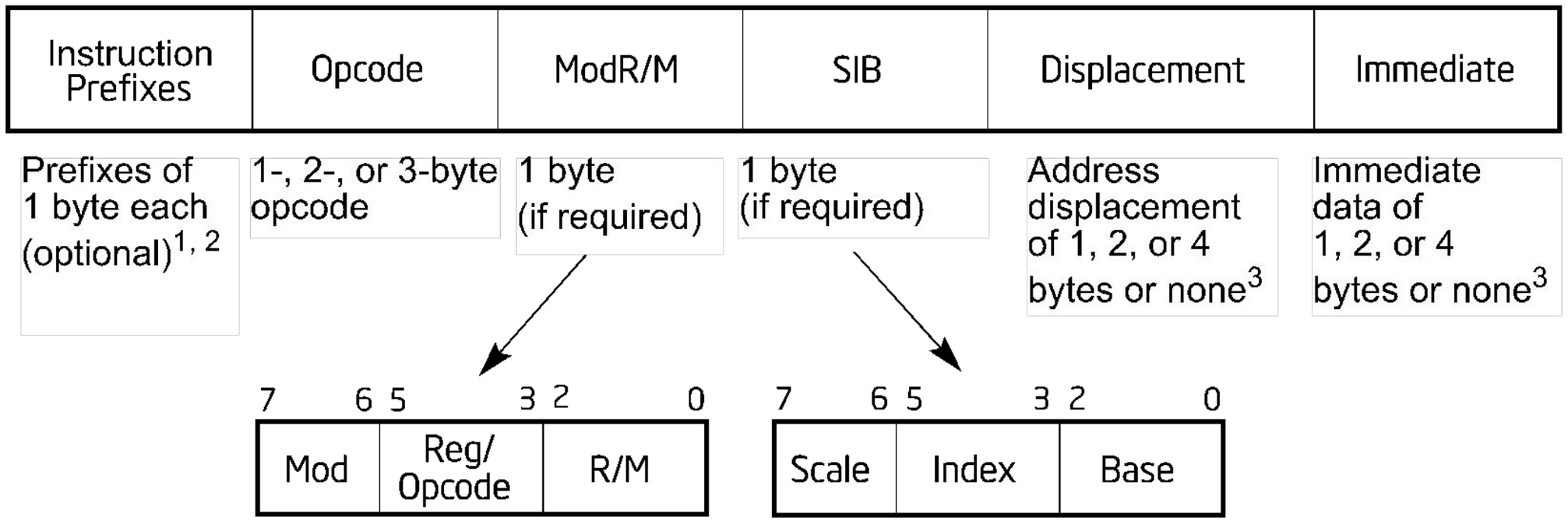
source: Desktop 4th Generation Intel® CoreTM Processor Family, Desktop Intel® Pentium® Processor Family, and Desktop Intel® Celeron® Processor Family, Datasheet – Volume 1 of 2

Exemple : Intel Core i5-4278U (Haswell)



source : Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4

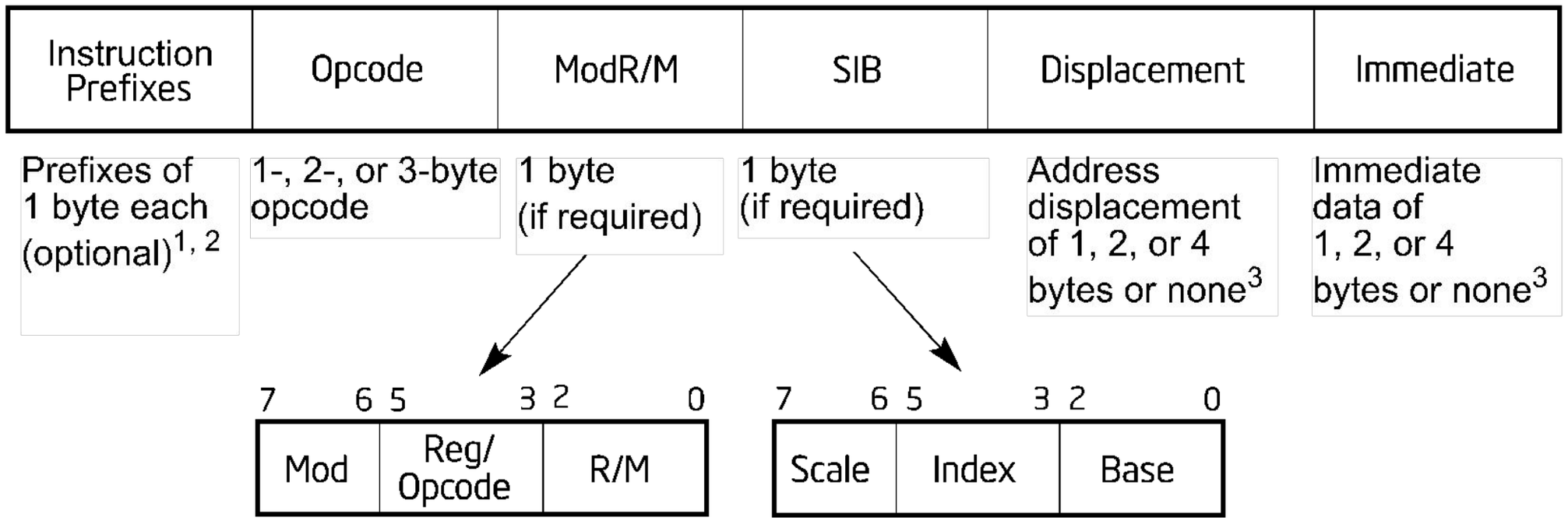
Exemple : Intel Core i5-4278U (Haswell)



source : Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4

- Prefix : Change le comportement d'une instruction.
  - Ex.: LOCK prefix → accès exclusif à la mémoire partagée
- Opcode : code opération.
- ModR/M : opérandes et modes d'adressages.
- SIB : mode d'adressage supplémentaire (indexé).
- Displacement : Adressage relatif
- Immediate : spécification d'une valeur constante.

Exemple : Intel Core i5-4278U (Haswell)



source : Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4

- Prefix : Change le comportement d'une instruction.
  - Ex.: LOCK prefix → accès exclusif à la mémoire partagée
- Opcode : code opération.
- ModR/M : opérandes et modes d'adressages.
- SIB : mode d'adressage supplémentaire (indexé).
- Displacement : Adressage relatif
- Immediate : spécification d'une valeur constante.

Exemple : Intel Core i5-4278U (Haswell)

```
int moyenne(int a, int b, int c, int d, int e, int f, int g, int h)
{
   return (a+b+c+d+e+f+g+h)/8;
}
```

```
_moyenne:   pushq    %rbp
            movq     %rsp, %rbp
            addl%esi, %edi
            addl%edx, %edi
            addl%ecx, %edi
            addl%r8d, %edi
            addl%r9d, %edi
            addl16(%rbp), %edi
            addl24(%rbp), %edi
            movl     %edi, %eax
            sarl $31, %eax
            shrl $29, %eax
            addl%edi, %eax
            sarl $3, %eax
            popq     %rbp
            retq
```

CentraleSupélec

université PARIS-SACLAY

LABORATOIRE DE RECHERCHE EN INFORMATIQUE

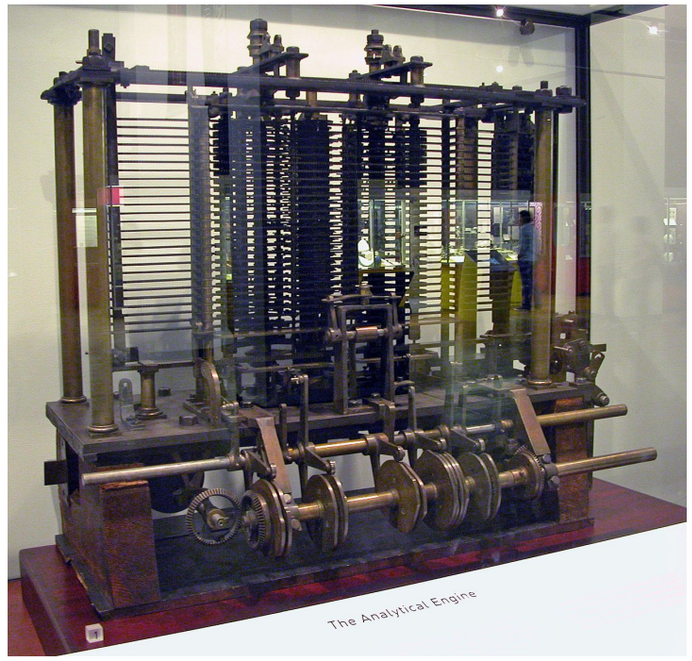
CHAPITRE VIII

Aperçu de l'histoire de l'informatique

Architecture des ordinateurs

Calculateurs mécaniques (1642 - 1945)

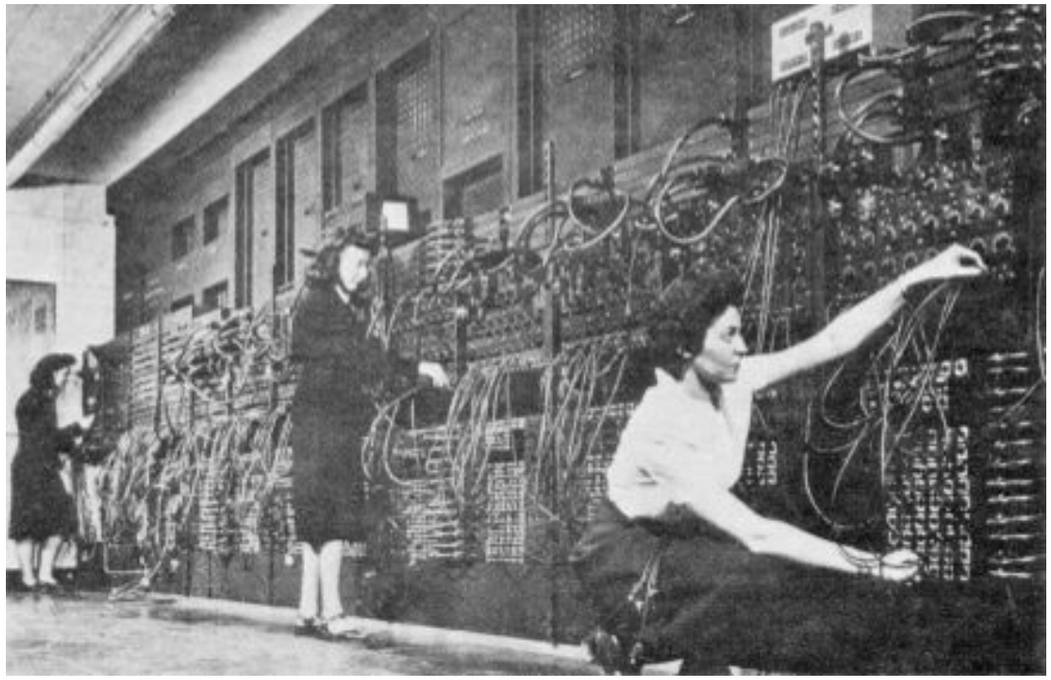
- Première calculette idée par Blaise Pascal (1642)
  - Seulement additions et soustractions.
- Machine analytique inventée par Charles Babbage (1834)
  - Premier ordinateur « general purpose »
- Quatre composants:
  - Un moulin, pour faire les calculs (le processeur)
  - Un magasin, pour stocker les résultats (la mémoire)
  - Lecteur de cartes perforées (dispositif d'entrée)
  - Imprimante (dispositif de sortie)
- Les programmes étaient écrites sur des cartes perforées
  - Ada Augusta Lovelace première programmeuse.



ENIAC, premier ordinateur (1946)

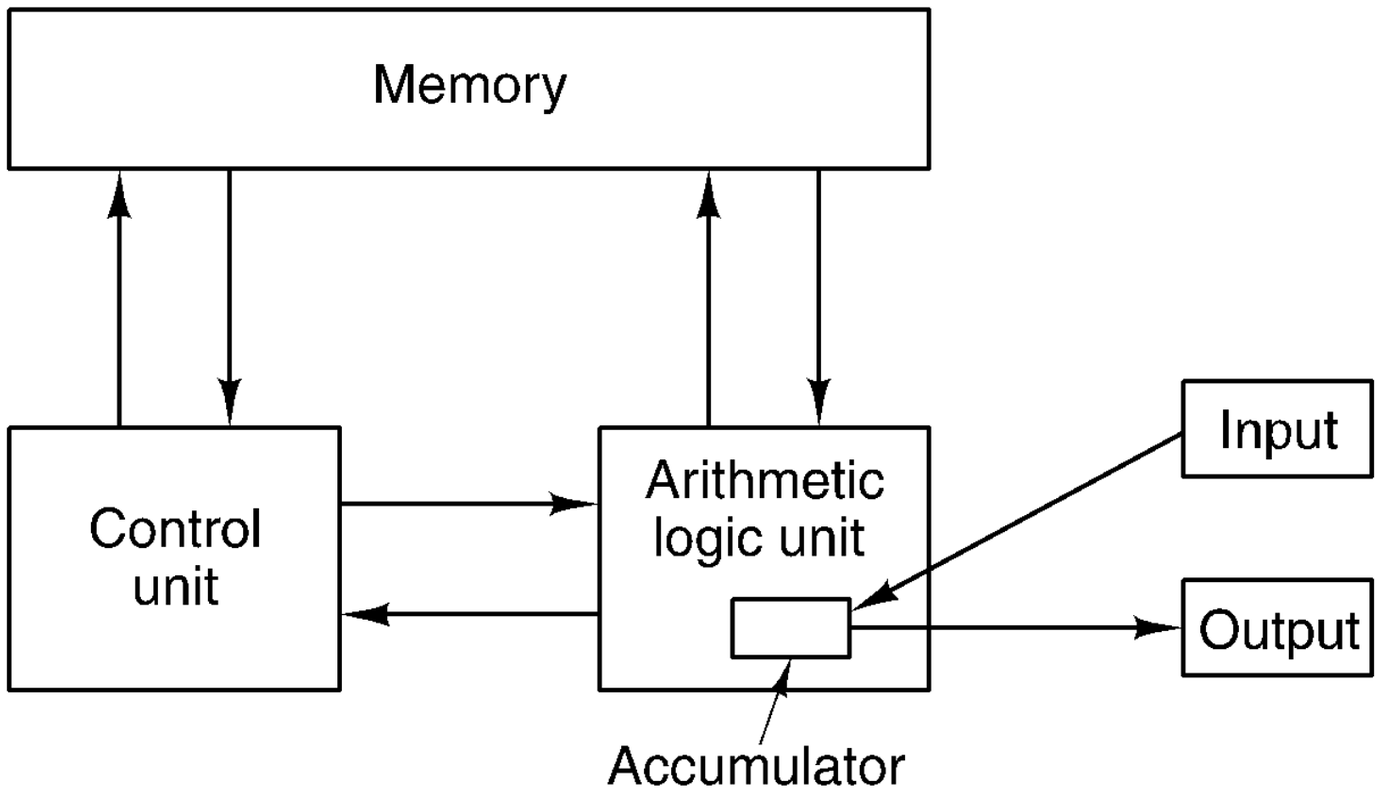
- Inventé par Eckert et Mauchley
- 18.000 tubes électroniques, 1.500 relais
- Formation d'un programme par des câbles.

Relais + tubes



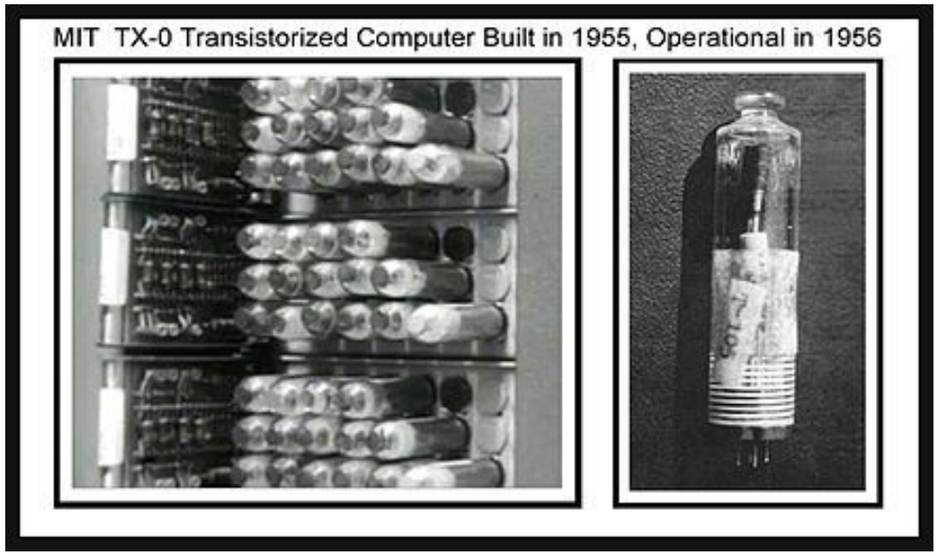
Histoire - EDVAC, archit. Von Neumann (1947)

- Représentation d'un programme sous forme digitale (et binaire).
- Mémorisation du programme.
- Architecture sur laquelle les ordinateurs modernes sont basés.

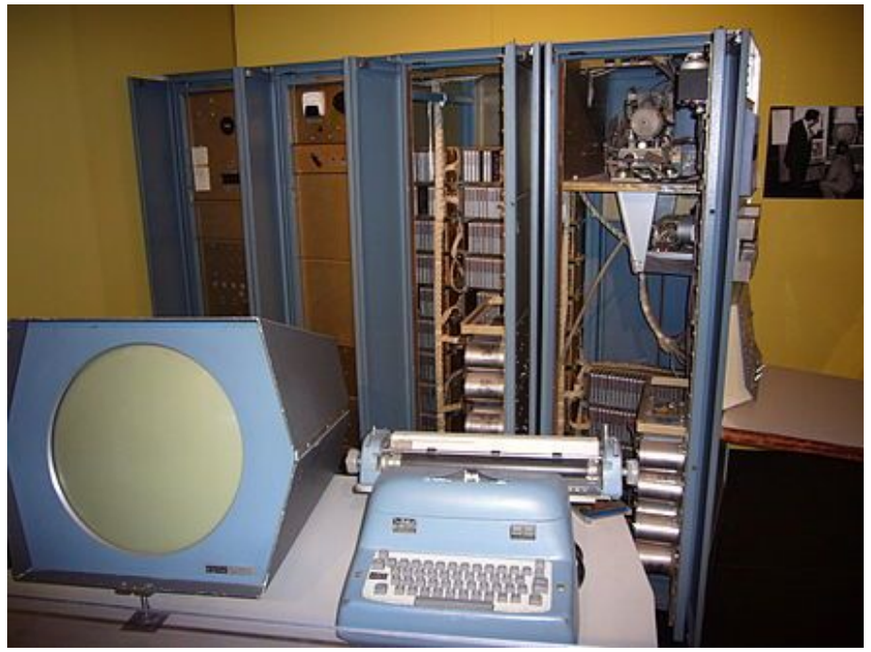


La génération transistors (1955 - 1960)

- **Transistor** : dispositif semi-conducteur utilisé comme un interrupteur dans les circuits logiques
- Inventé en 1948 : John Bardeen, Walter Brattain, William Shockley (Bell Labs).

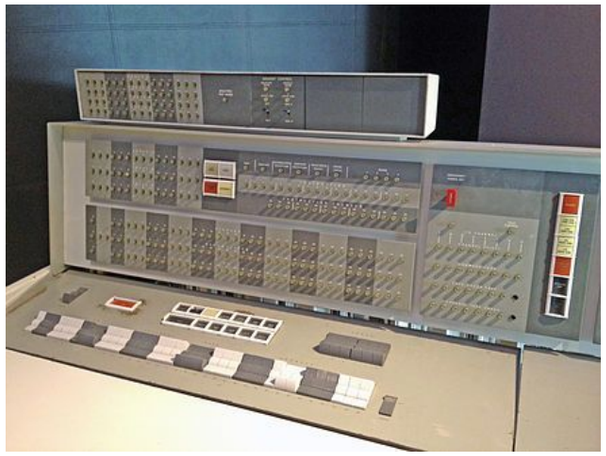


TX-0, premier ordinateur à transistors (MIT) - 1955



DEC PDP-1 (1960)

IBM 7090 (1959)



IBM 7094 (1962)

La génération transistors (1955 - 1965)

CDC 6600 (1964) – Premier superordinateur scientifique (Seymour Cray)

La génération circuits intégrés (depuis 1965)

- **Circuit intégré** : composant électronique intégrant des circuits électroniques sur une puce de matériel semi-conducteur (silicium)
  - Robert Noyce, 1958, Fairchild Semiconductor.
- Possibilité de placer des dizaines de transistors sur une seule puce.
  - ordinateurs moins encombrants.

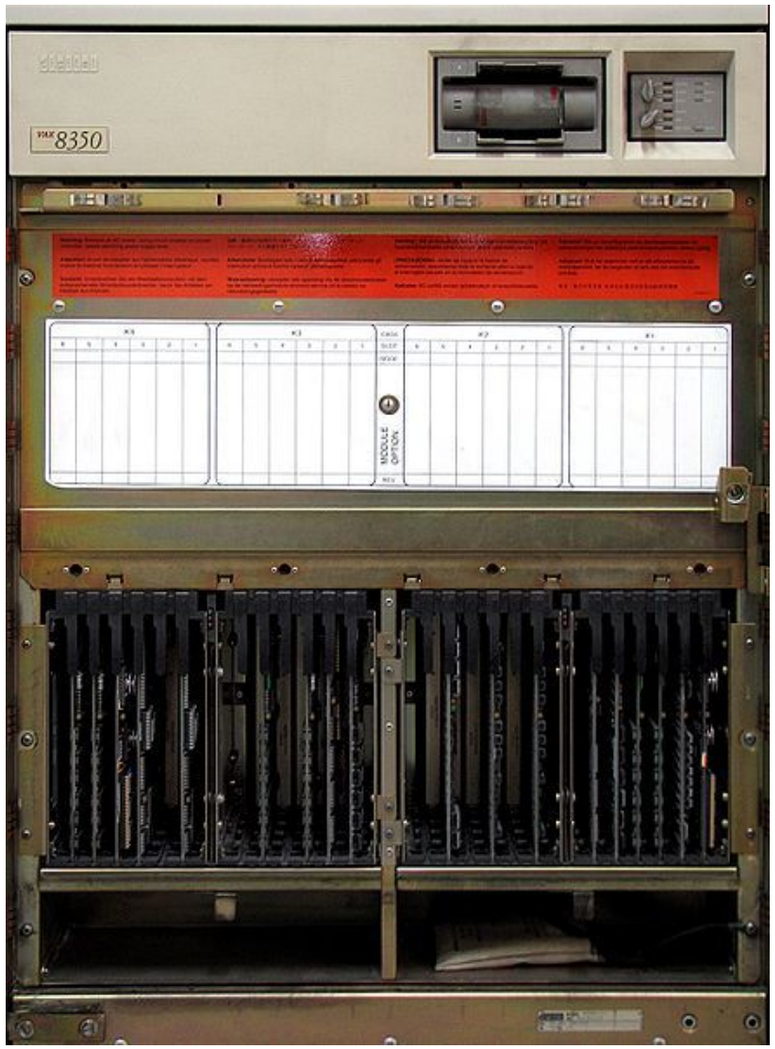
IBM 360 (1965) – première famille d'ordinateurs

- 13 modèles d'ordinateurs. Concept de **famille.**
- 14 000 exemplaires vendus.
- **Multiprogrammation** : plusieurs programmes en mémoire au même temps.
- Capacité d'émulation des anciens ordinateurs IBM.

La génération circuits intégrés (depuis 1965)

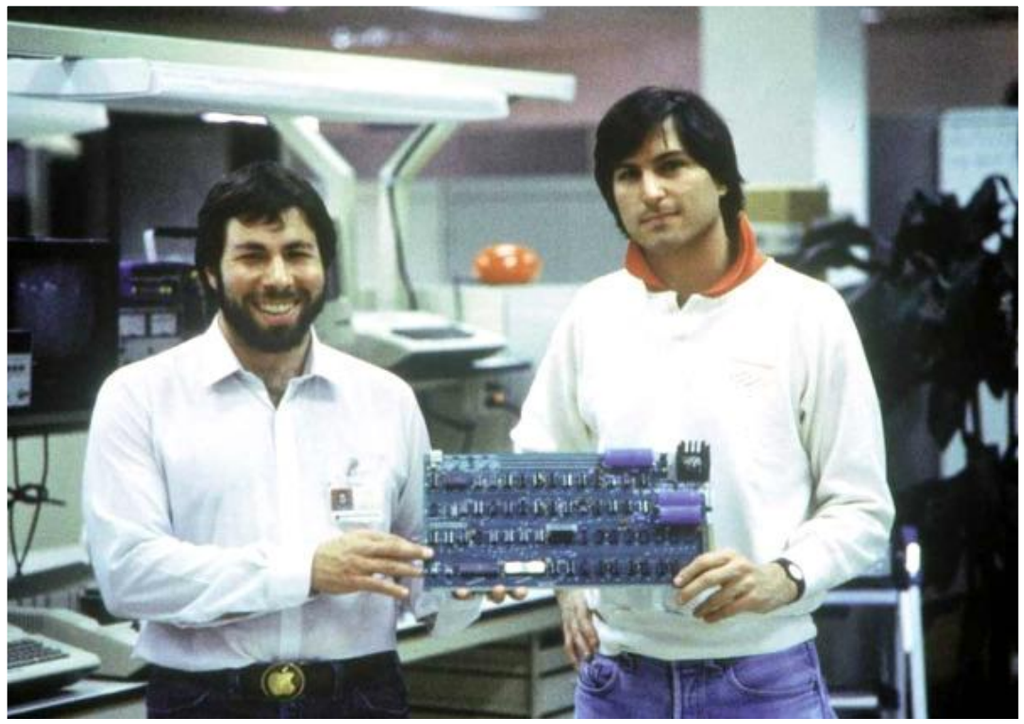
DEC PDP-8 (1965) – 50 000 unités vendues

DEC PDP-11 (1970) – 600 000 unités vendues



DEC VAX (1978) – Premier super mini-ordinateur (32 bits)

La génération circuits intégrés (depuis 1965)



La génération circuits intégrés (depuis 1965)

- **VLSI (Very Large Scale Integration) :** intégration de millions de transistors sur une puce unique (~1980).
  - ordinateurs plus petits et beaucoup moins chers.
- Concept de **micro-ordinateur** : simple, économique, mais puissant

- **IBM Personal Computer** : 12 Août 1981
- **CPU** : Intel 8080 4.77Mhz
- **Mémoire** : 16kB ~ 256 kB
- **Système d'exploitation :** Microsoft DOS
- **Prix :** $1 565 (~$5 000 aujourd'hui)
- 100 000 unités vendues seulement en 1981
- Schémas des circuits publiés
  - d'autres clones sont produits, beaucoup moins chers

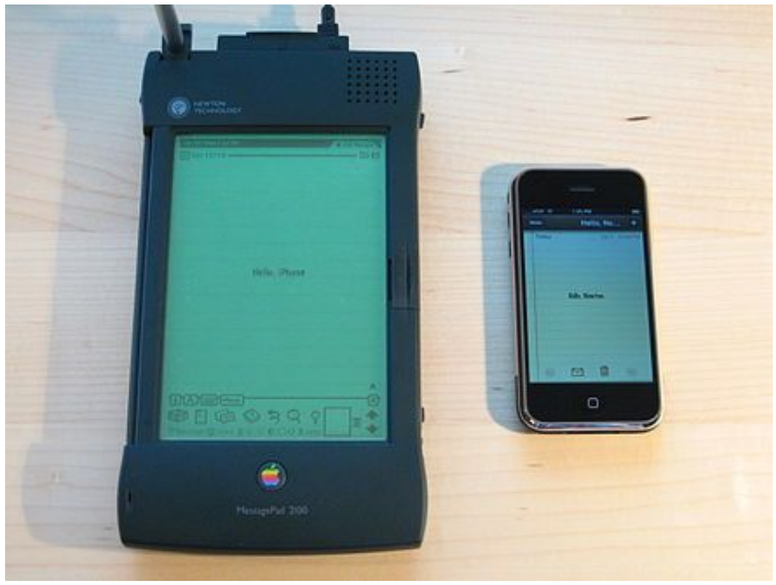
La génération circuits intégrés (depuis 1965)

Osborne 1 (1981) – Premier ordinateur portable (10,7 Kg)

Apple Lisa (1981) – Premier ordinateur doté d'une interface graphique.

Apple Macintosh (1984) – Successeur de Lisa, plus performant

Ubiquitous Computing

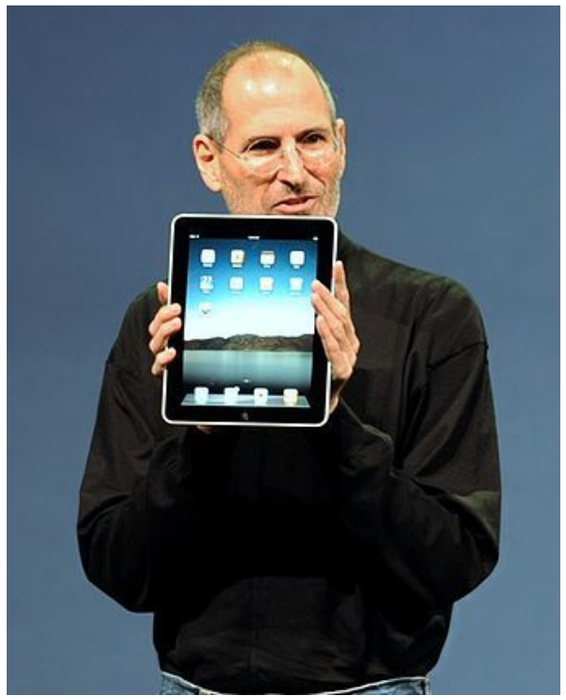


Apple Newton (1993) – Premier ordinateur de poche

Compaq iPAQ 3630 (2000)

HP iPAQ HW910

Apple iPhone (2007)



Apple iPad (2010)