CentraleSupélec

Architecture des Ordinateurs

H.Delebecque
B2-27
henri.delebecque@centralesupelec.fr

Introduction (1/2)

"Architecture", c'est quoi ?

Basée sur des standards

Décomposition hiérarchique

Conception en "couches"

Qu'allez-vous apprendre ?

3 B.E, 2 E.L

Introduction (2/2)

Cours dense, mais progressif

Présentation incrémentale

Arrêtez-moi dès que vous ne comprenez plus !

wwwdi.supelec.fr/architecture

Les pré-requisit

Plan

2- Logique, arithmétique
3- Structure d'un ordinateur
4- Conception d'un processeur
5- Mémoires
6- Entrées-Sorties, Bus
7- De Python au processeur

2. Logique, arithmétique

2.1 Notions de base

Binary Digit : le "bit"

Plus petite quantité d'information

2 Valeurs: 0 ou 1

→ Notation en binaire

2.1 Notions de base

Logique booléenne

→ 1 = vrai, 0 = faux

→ Opérateurs logiques:

Et, Ou, Non, Non Et, Non Ou

A	B	A And B	A Nand B	A Or B	A Nor B
0	0	0	1	0	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	0	1	0

2.2 Blocs logiques utiles

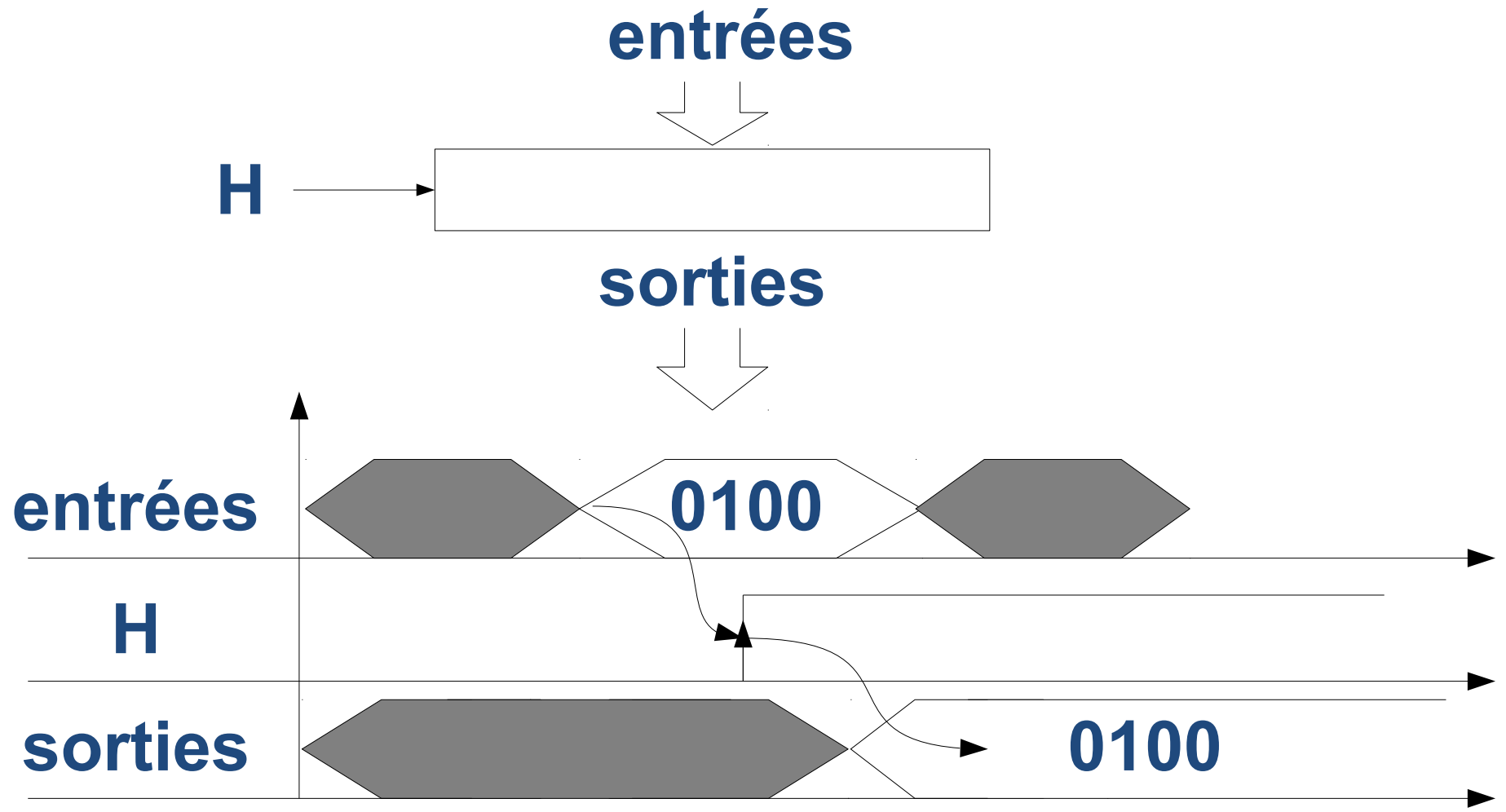
Vision externe seulement

Registre,

Mémoire,

Unité de calcul.

2.2.1 Registre

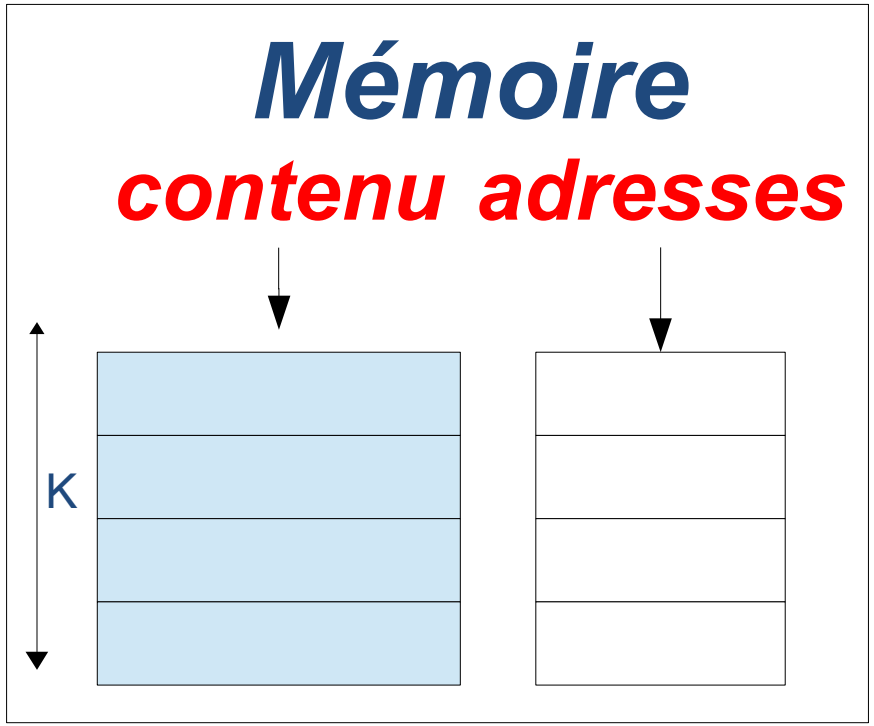


2.2.2 Mémoire

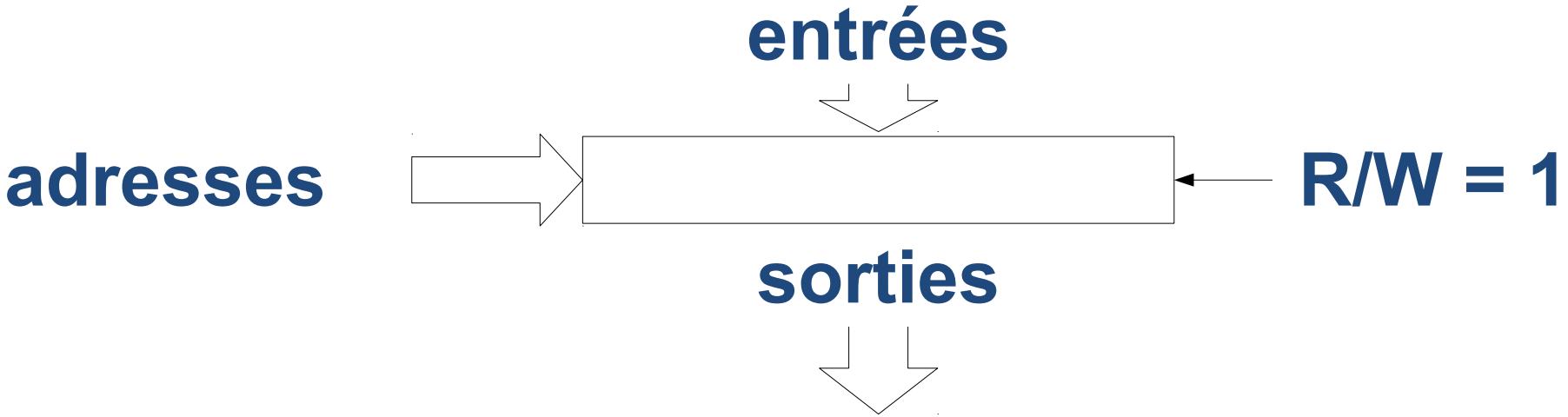
Analogie: meuble à tiroirs

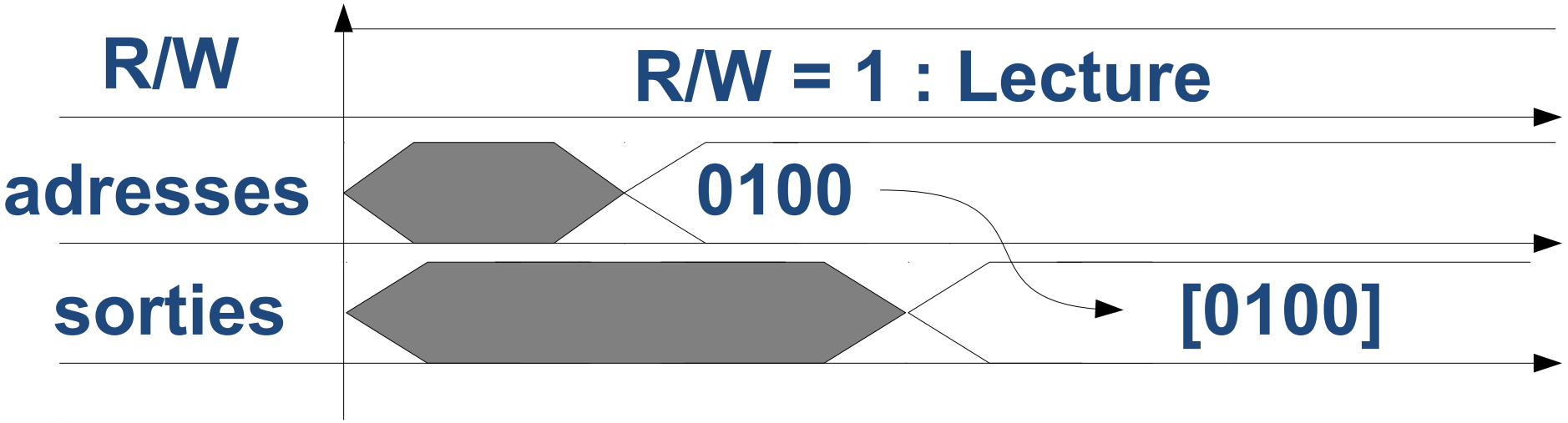
~ 1 mot ↔ une variable Python

Nombre de mots mémoire = K
Taille d'un mot usuel :
8, 16, 32 bits

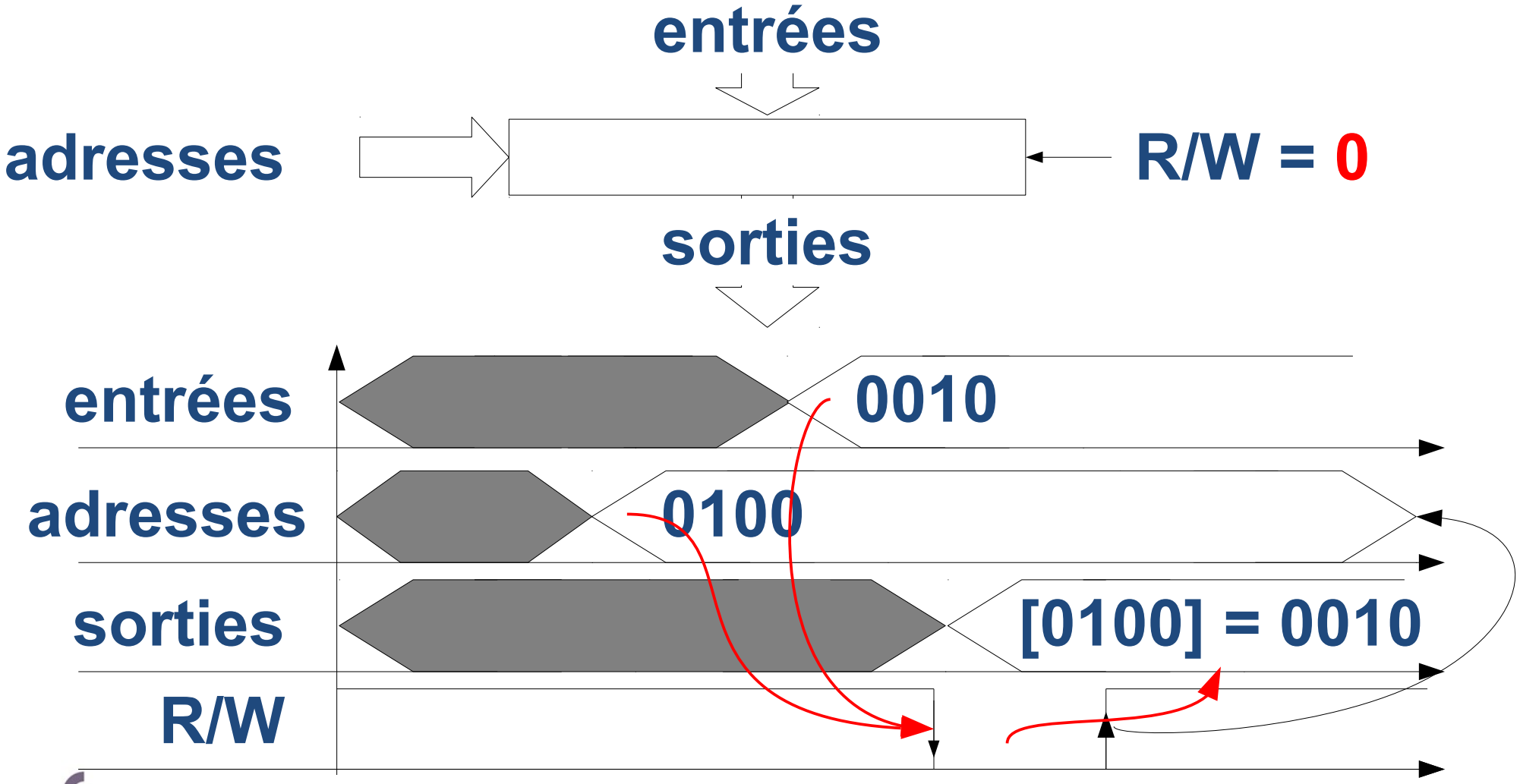


2.2.2 Mémoire en lecture

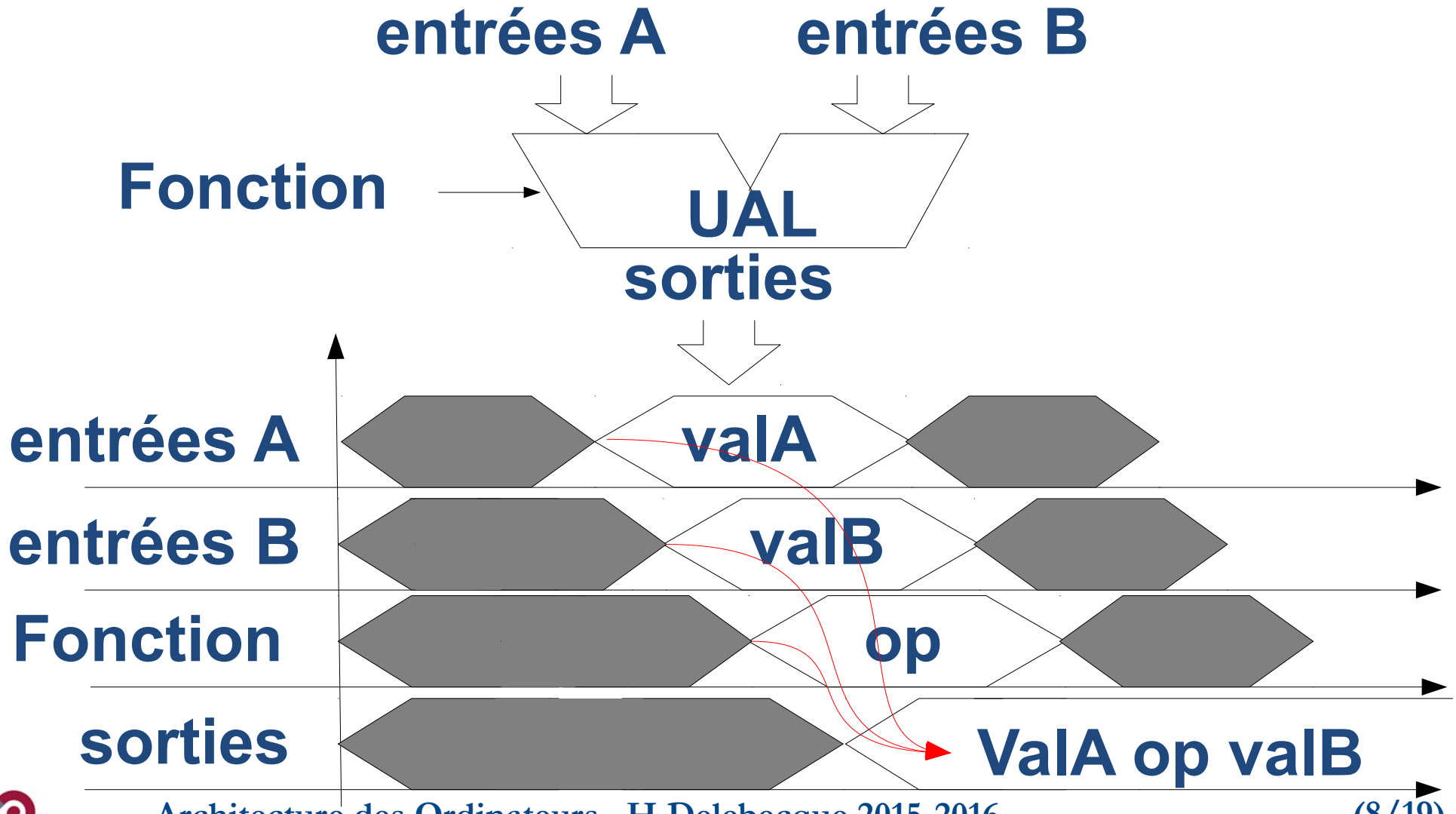




2.2.2 Mémoire en écriture



2.2.3 Bloc de calcul



2.3 Les entiers naturels $\mathbb{N}$

Notation binaire

→ base 2

→ un nombre de bits donné: n

→ un ensemble de valeurs fini !

Sur n bits, de 0 à 2^n-1

2.3 Les entiers naturels $\mathbb{N}$

Notation binaire délicate …

décimal	binaire
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

décimal	binaire
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

2.3 Les entiers naturels $\mathbb{N}$

Donc une base plus grande: 16

décimal	binaire	hexa	décimal	binaire	hexa
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

2.3 Les entiers naturels $\mathbb{N}$

Notation hexadécimale

$\longrightarrow$ un chiffre représente 4 bits

$\longrightarrow$ l'arithmétique reste la même

C'est juste un moyen de représenter les valeurs, de $\mathbb{N}$ comme de $\mathbb{Z}$!

2.4 Les entiers relatifs $\mathbb{Z}$

Des chiffres... et un **signe** !

→ le **complément à 2**

Si on veut calculer "-Val",

On calcule 2^n - Val

→ $\overline{\text{Val}}$ (inversion des bits) + 1

Toute valeur entre -2^{n-1} et $2^{n-1} - 1$

2.4 Les entiers relatifs $\mathbb{Z}$

Un peu de pratique !

Val décimale	Val	$\overline{\text{Val}}$	-Val
1	0001	1110	1111
3	0011	1100	1101
7	0111	1000	1001
8	1000	0111	1000

Bit de signe en poids fort

2.4 Les entiers relatifs $\mathbb{Z}$

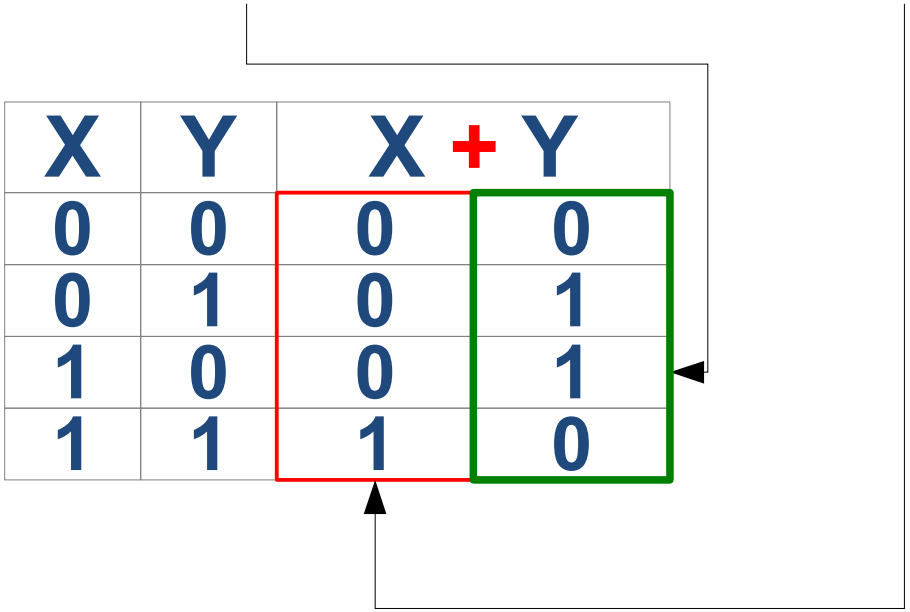
Attention:

Une valeur binaire ne fournit pas d'information sémantique

Val	Si non signée	Si signée
0111	7	7
1000	8	-8
1111	15	-1

2.5 Addition

Addition binaire mais classique :
Résultat: 1 bit et Retenue (Carry)



X	Y	X + Y	
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

2.5 Addition

Addition sur 4 bits, dans $\mathbb{N}$

X		Y		X + Y				
1	0001	1	0001		0	0	1	0
3	0011	1	0001		0	1	0	0
7	0111	1	0001		1	0	0	0
15	1111	1	0001	1	0	0	0	0

Retenue sortante

2.5 Addition

Addition sur 4 bits, dans $\mathbb{Z}$

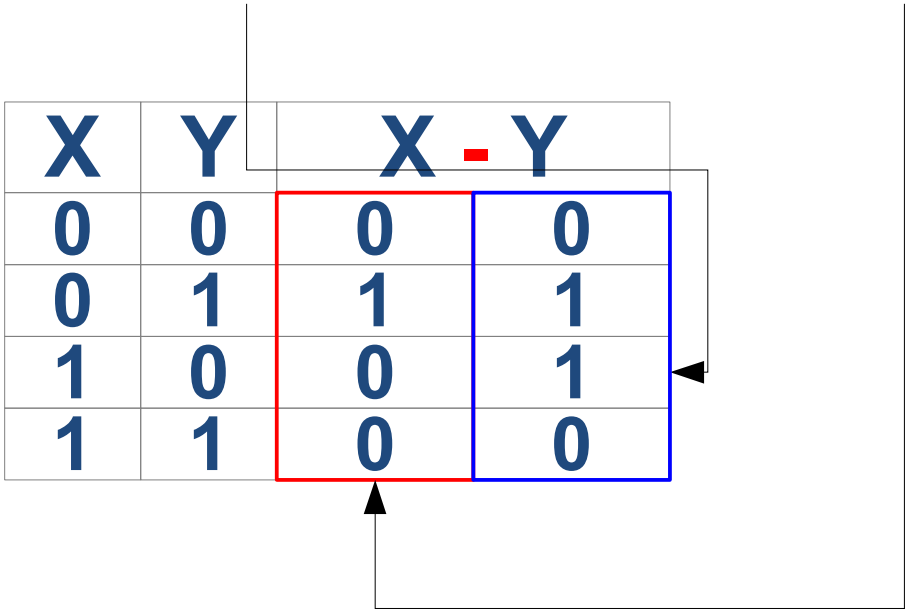
X		Y				X + Y			
1	0001	1	0001			0	0	1	0
-1	1111	1	0001		1	0	0	0	0
positif		positif			C	négatif			
7	0111	1	0001	1	0	1	0	0	0

Débordement (V)

retenue non significative

2.6 Soustraction

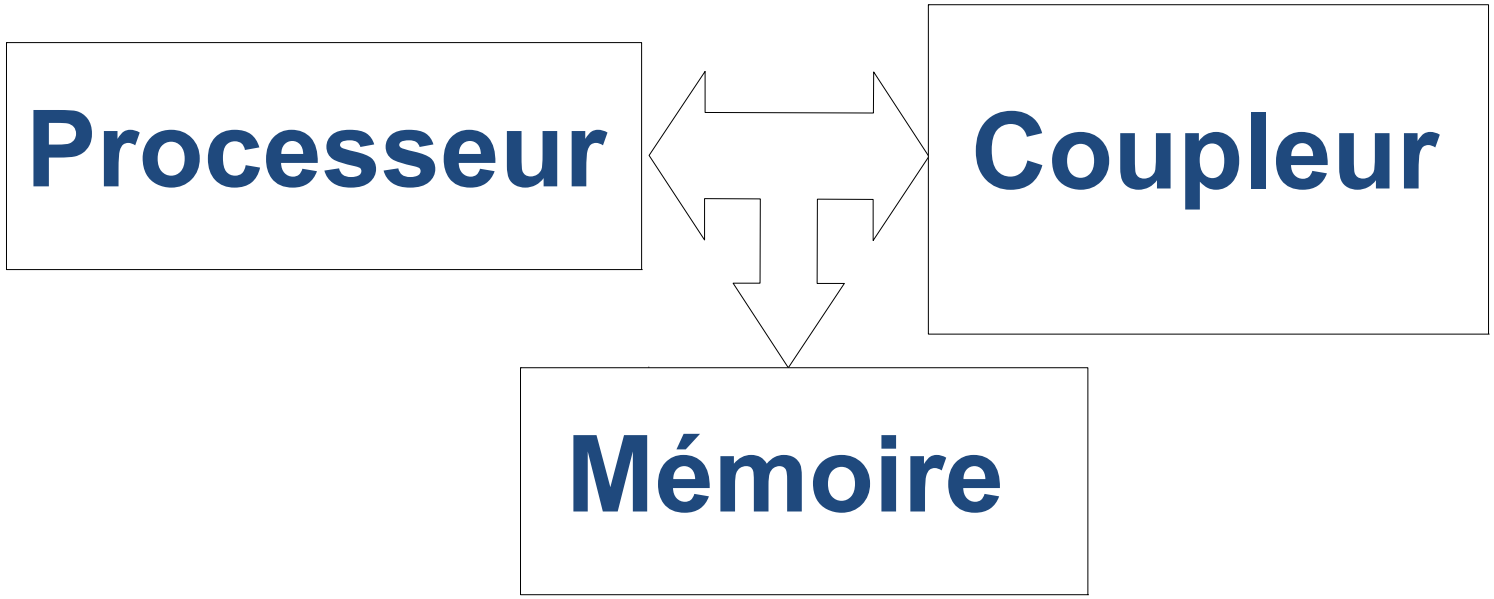
Soustraction binaire classique :
Résultat: 1 bit et le bit de signe



X	Y	X - Y	
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

3. Architecture Générale

3.1 Principes (1/7)



Les 4 éléments sont obligatoires !!

Deux exemples

Un système embarqué:

- très compact,
- très connecté au monde réel,
- autonome, tolérant aux pannes

Un ordinateur de calcul:

- coûteux, énorme, puissant,
- géré par d'autres ordinateurs.

3.1 Principes (2/7)

Le processeur (i.e. CPU)
Lit, stocke, calcule des **données**,
Contrôle les autres éléments,
Une réalisation **interne**,
Une spécification **externe**.

3.1 Principes (3/7)

La mémoire (au sens large)
Mémorise des **données**,
Mémorise les **instructions**.
De façon permanente ou non,
Avec différents moyens d'accès.

3.1 Principes (4/7)

Un coupleur (d'entrées-sorties)
Echange des **données**,
Avec le monde extérieur.
Dans les deux sens,
Données échangées avec la mémoire.

3.1 Principes (5/7)

Les bus (au sens large)
Transporte des informations,
Typées ou non.
1 émetteur, 1 à N récepteurs,
“Internes” ou “externes”.

3.1 Principes (6/7)

Architecture externe du CPU:

- **Ce que voit le programmeur**
  - **Registres (type, taille, usage,...)**
    - **“variables temporaires”**
  - **Instructions (rôle, usage,...)**

Valeurs des registres ↔ état CPU

3.1 Principes (7/7)

Architecture interne du CPU:

Ce qu'a défini le concepteur

Registres (“visibles” ou pas),

Unités de calcul (UAL)

Bus internes reliant ces modules

Séquenceur pilotant ces modules

4. Conception d'un CPU

Quelle architecture externe ?

→ Quels “utilisateurs” ?

→ Donc quels algorithmes ?

→ Quels types de données ?

→ Organisées comment ?

4.1 Modèle de Von Neuman

Defini dans les années 60

3 principes simples:

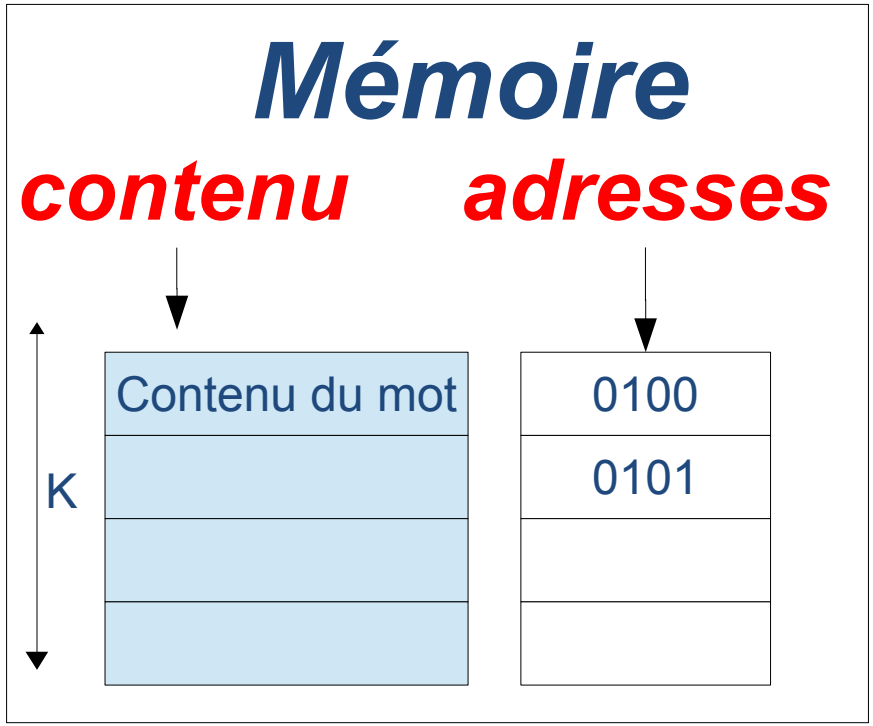
- mots mémoire banalisés,
- de taille constante
- des instructions en séquence

Modèle très (trop) simple

4.2 Codage d'une instruction

Notion de mémoire

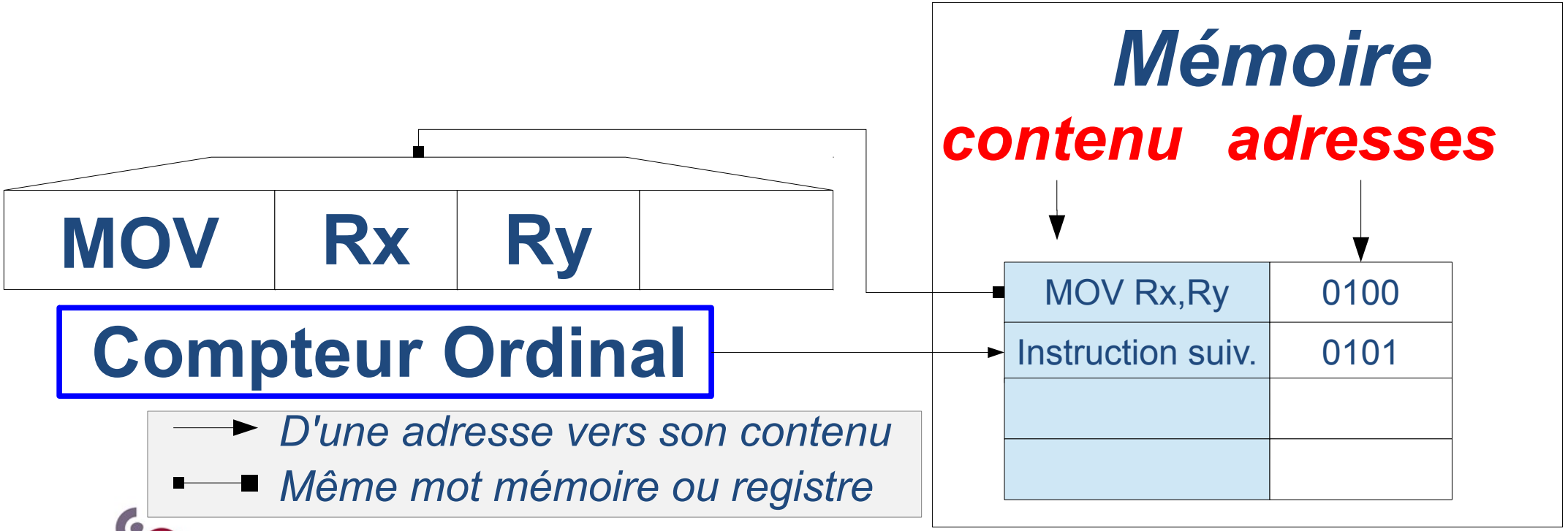
Analogie: meuble à tiroirs



Nombre de mots mémoire = K
Nombre de bits d'adresse = n
Donc : $2^n = K$

4.2 Codage d'une instruction

Une instruction machine c'est: 1 (ou plusieurs) mot mémoire



4.2 Codage d'une instruction

L'instruction spécifie

- le type d'opération (ADD,…)
  - → le “code-opération”
- où trouver le(s) argument(s)
- où ranger l'éventuel résultat

4.2 Codage d'une instruction

Pour spécifier un argument:

- on donne sa valeur (constante)

- on donne son adresse mémoire

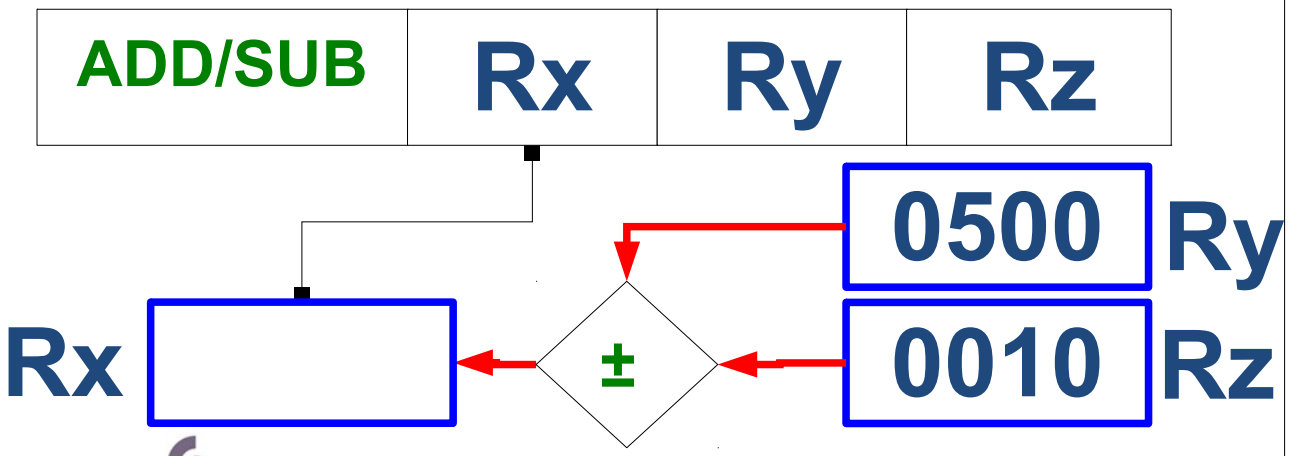
- on donne le n° de registre qui la contient

→ “mode d'adressage”

4.3.1 Mode Registre (1/2)

La fonction du registre dépend de l'instruction et de sa place

Ry ± Rz → Rx



Mémoire

op Rx,Ry,Rz	0100

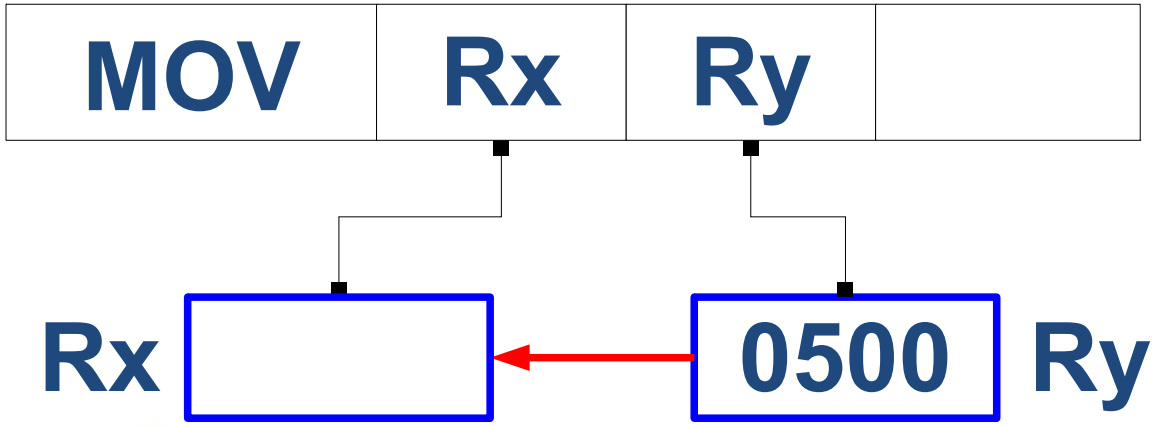
4.3.1 Mode Registre (2/2)

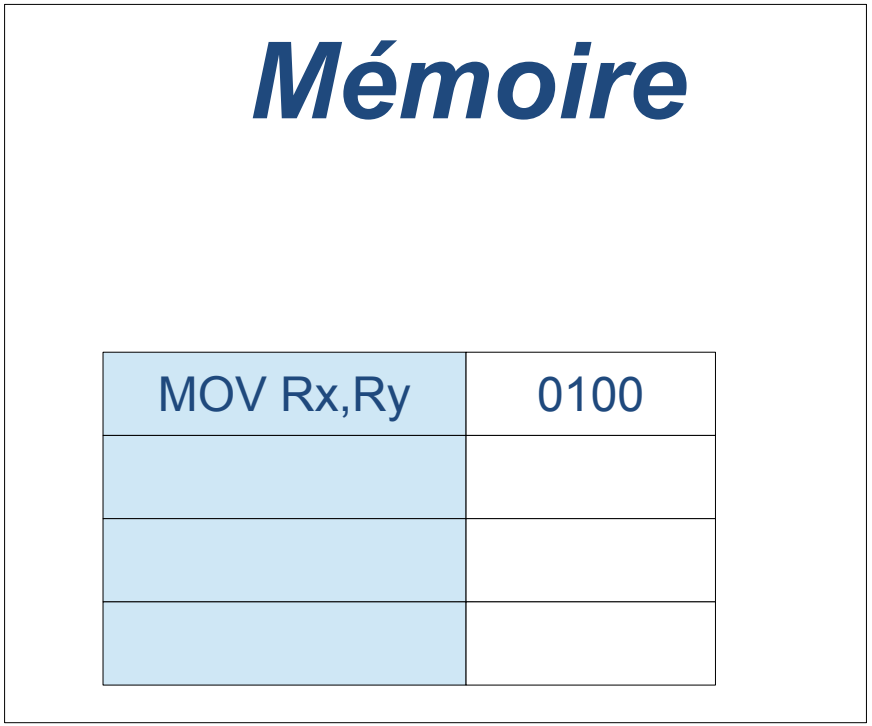
MOV entre registres: MOV Rx,Ry

Ry → Rx

Attention au sens

Ry reste intact

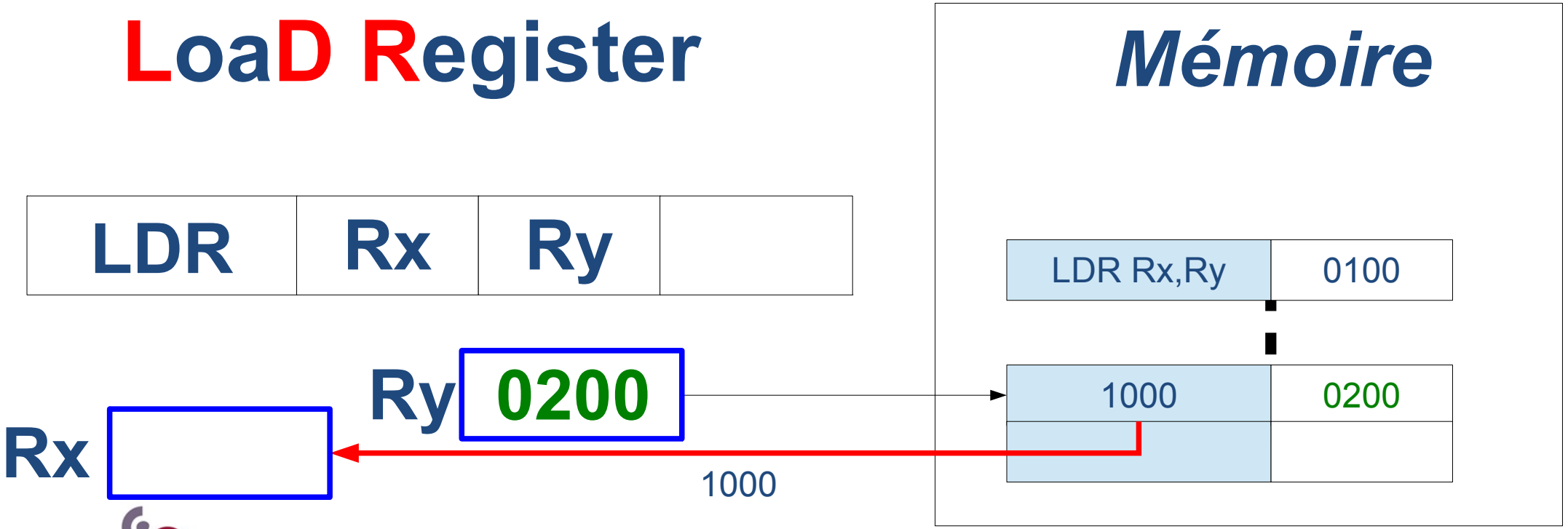




4.3.2 Mode Direct (1/3)

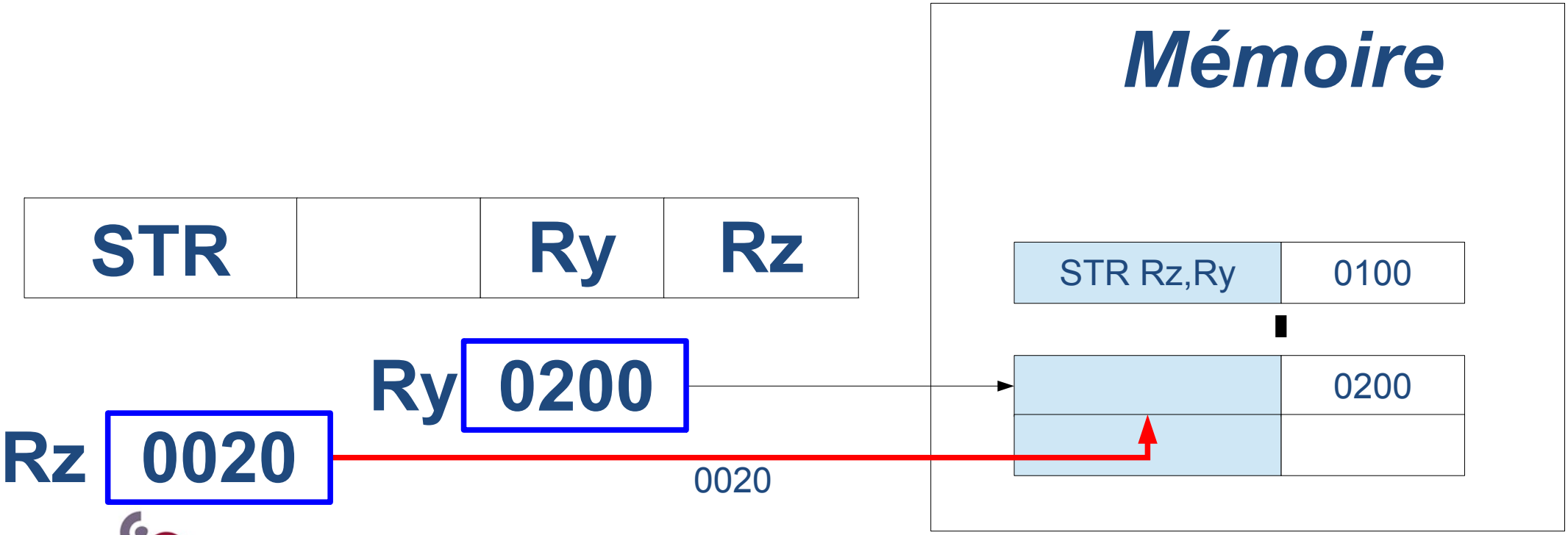
Si l'argument est la valeur à l'adresse contenue dans Ry

LoaD Register



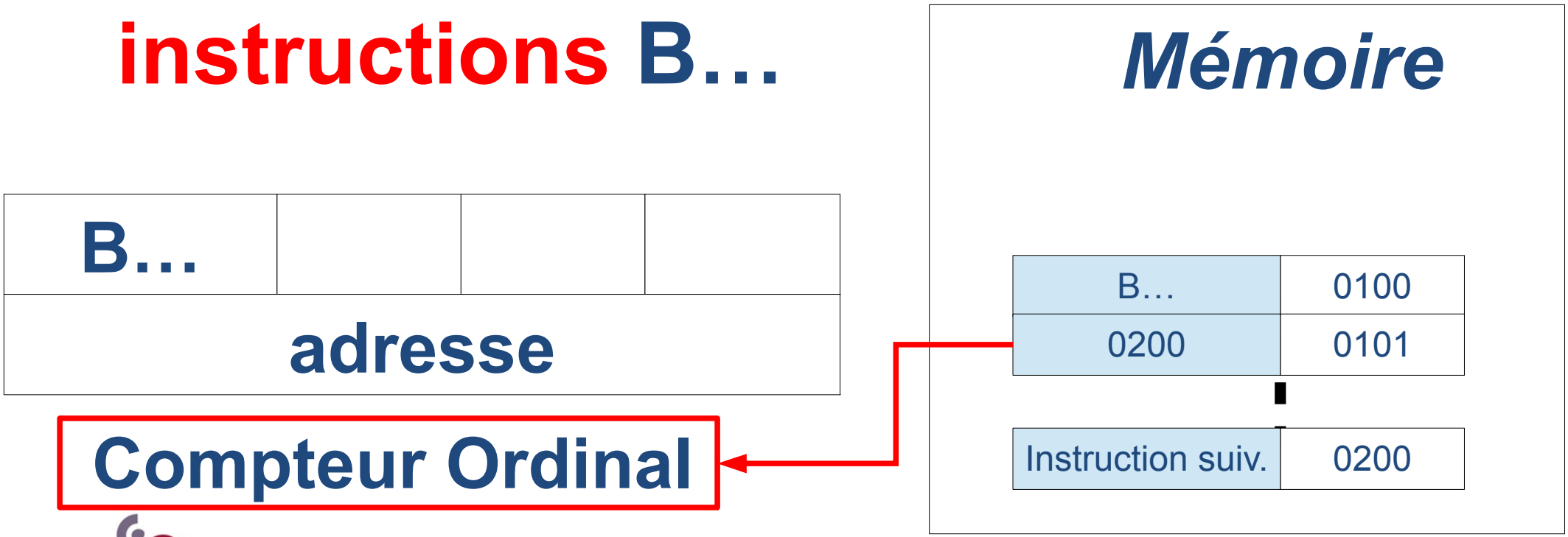
4.3.2 Mode Direct (2/3)

Si l'argument est à stocker à l'adresse contenue dans Ry



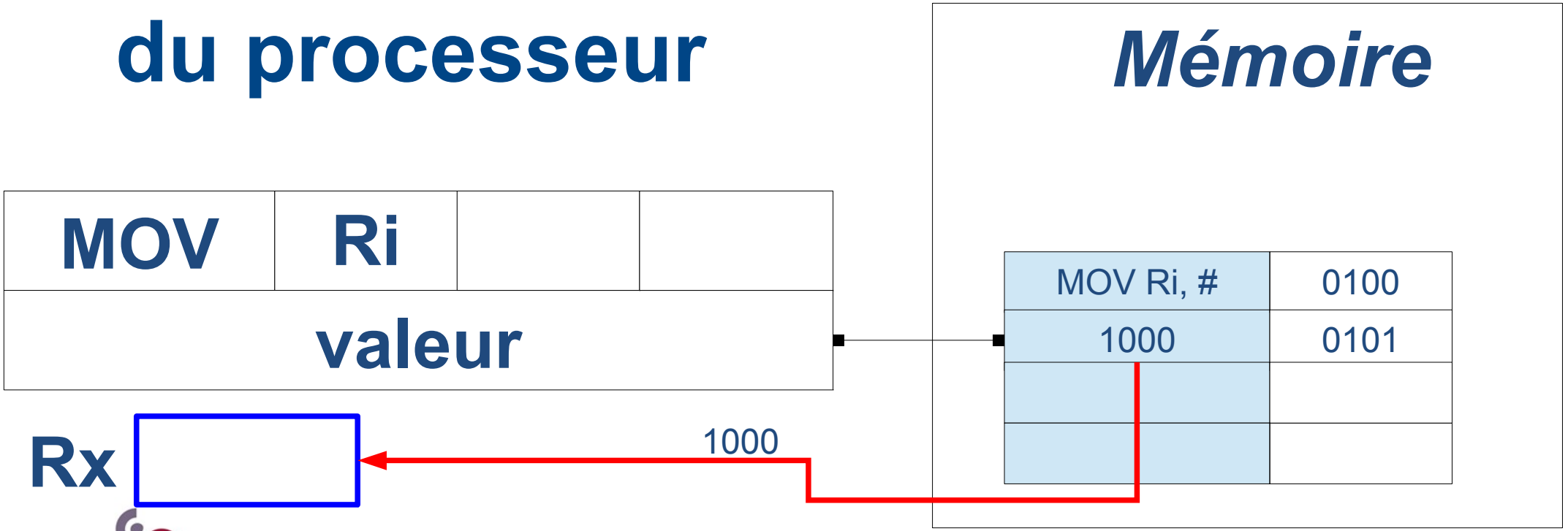
4.3.2 Mode “Direct” (3/3)

Si l'argument est l'adresse de la prochaine instruction pour des instructions B…



4.3.3 Mode immédiat

L'argument est une valeur signée ou non signée, Ri est le registre i du processeur



4.4 Architecture externe

Processeur RISC, type ARM

8 registres

16 instructions sur 1 ou 2 mots

Modes immédiat, registre, direct

De 1 à 3 arguments

4.4 Instructions (1/2)

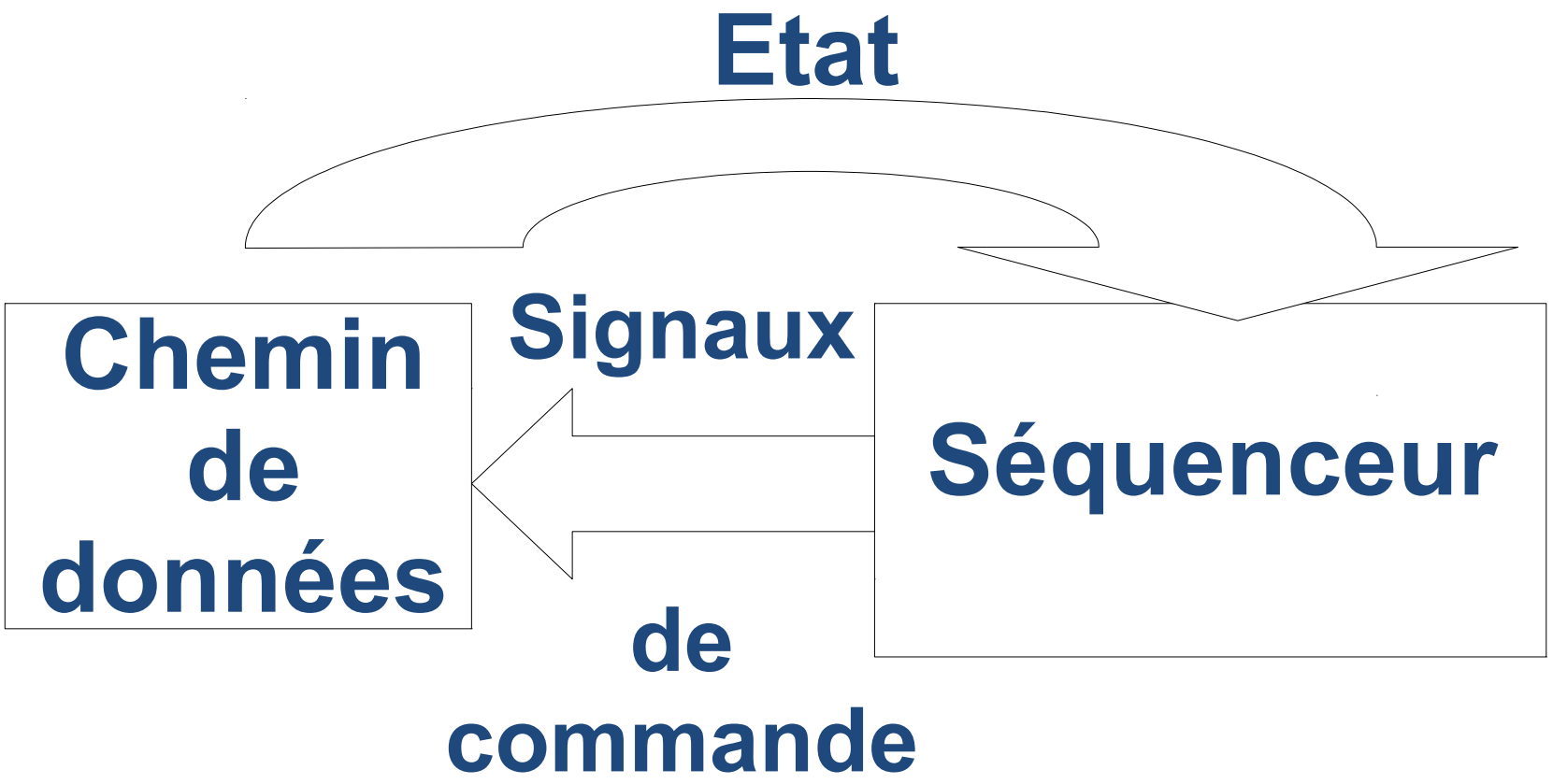
instruction	i	Rx	Ry	Rz	action
ldr rx,ry	0	dst	adr	-	[Ry] → Rx
str rz,ry	0	-	adr	src	Rz → [Ry]
mov rx,ry	0	dst	src	-	Ry → Rx
op rx,ry,rz	0	dst	arg1	arg2	Ry ± Rz → Rx
cmp ry,rz	0	-	arg1	arg2	Ry - Rz → flags
b… rz	0	-	-	arg2	Rz → CO (si flag)

Si vous ne comprenez pas tout, demandez !

4.4 Instructions (2/2)

instruction	i	Rx	Ry	Rz	action
mov rx,ry	0	dst	src	-	Ry → Rx
mov rx,#val	1	dst	-	-	[CO + 1] → Rx
op rx,ry,rz	0	dst	arg1	arg2	Ry ± Rz → Rx
op rx,ry,#val	1	dst	arg1	-	Ry ± [CO + 1] → Rx
cmp ry,rz	0	-	arg1	arg2	Ry - Rz → flags
cmp ry,#val	1	-	arg1	-	Ry - [CO + 1] → flags
b… rz	0	-		arg	Rz → CO (si flag)
b… val	1	-	arg	-	[CO + 1] → CO (si flag)

4.5 Conception Globale



4.5 Architecture Interne

Chemin de données:

Tout ce qui:

- mémorise,
- transporte,
- calcule des données

Doit recevoir des "ordres"

4.5 Architecture Interne

Séquenceur:

- commande le C.D.D,

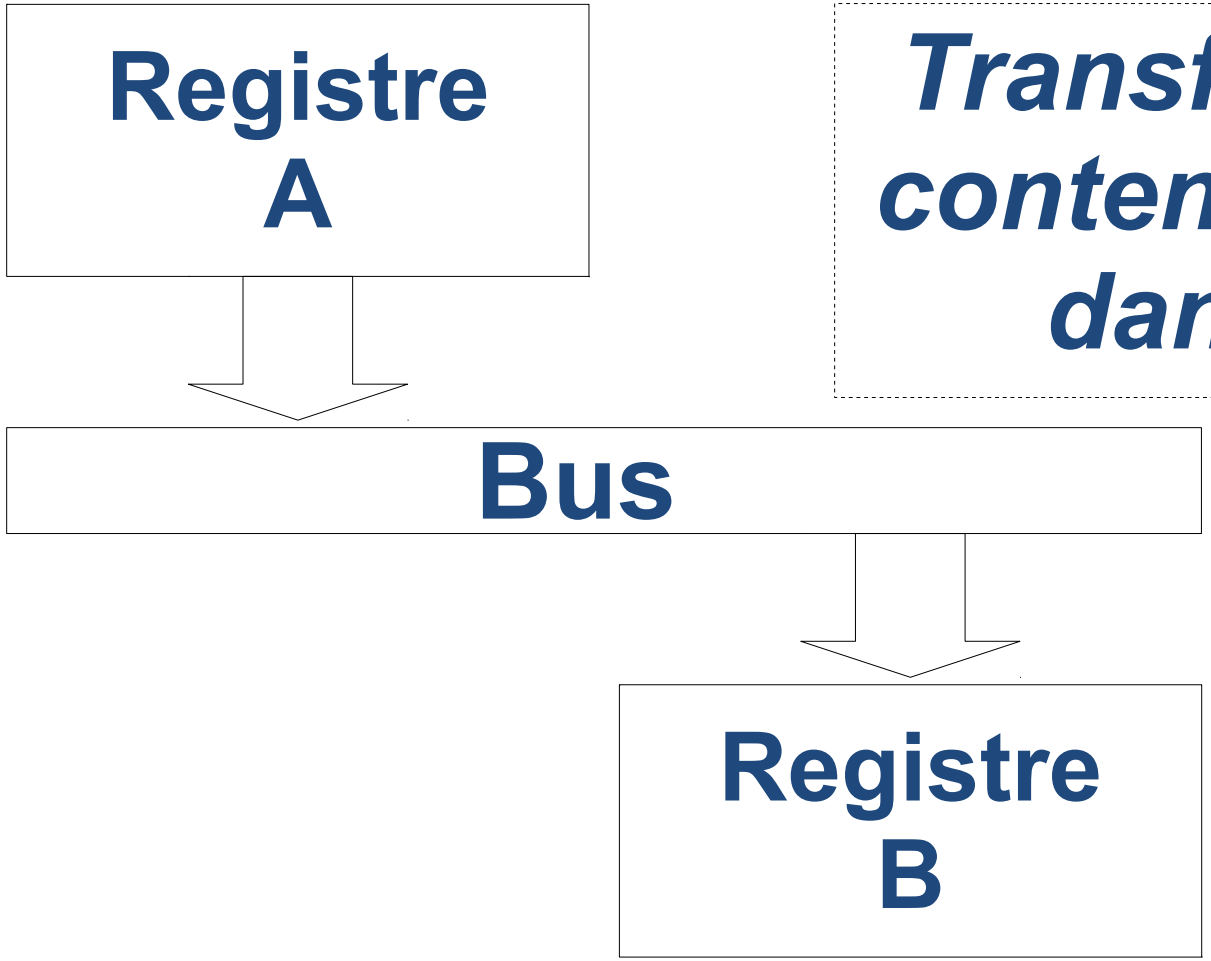
- selon l'instruction,

- en plusieurs cycles,

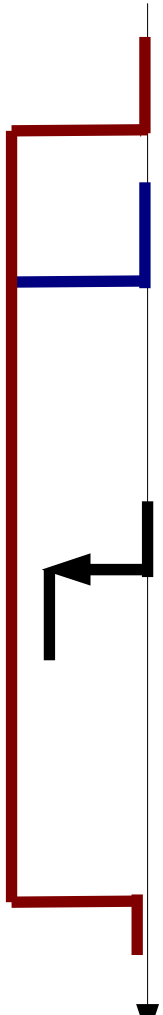
- en fonction de l'état du C.D.D.

Une analogie…

4.5 Exemple de CDD trivial



4.5 Le CDD en action



A branche ses sorties sur le bus

B branche ses entrées au bus

On attend qq **nanosecondes**

B **mémorise** ses entrées

On attend qq nanosecondes

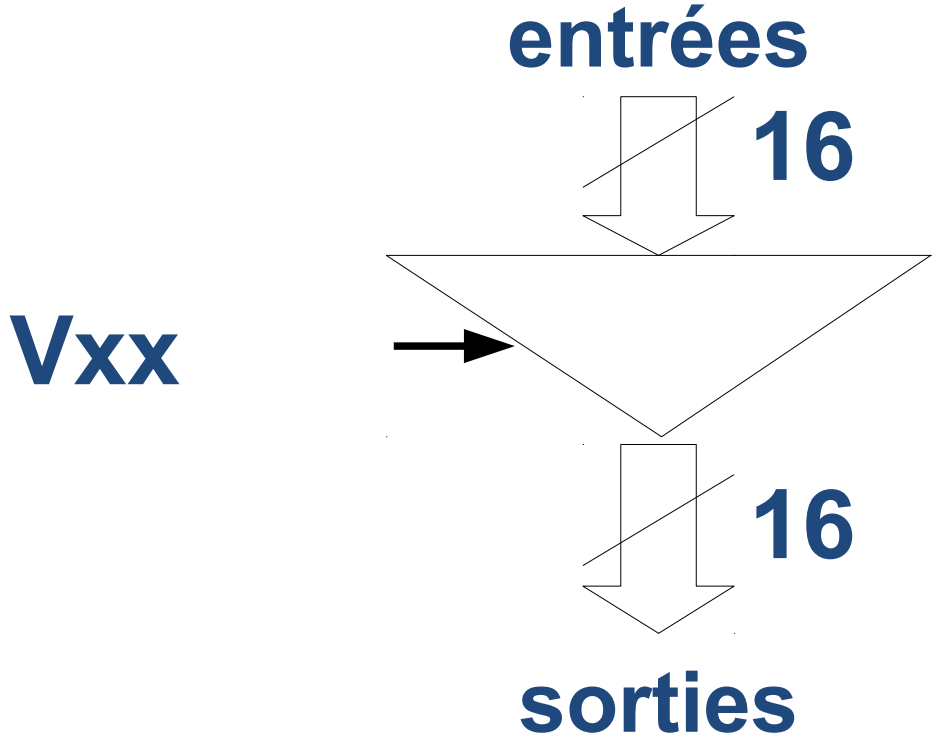
A et B libèrent le bus dans un ordre quelconque.

4.6 “Blocs de construction”

Des modules standards

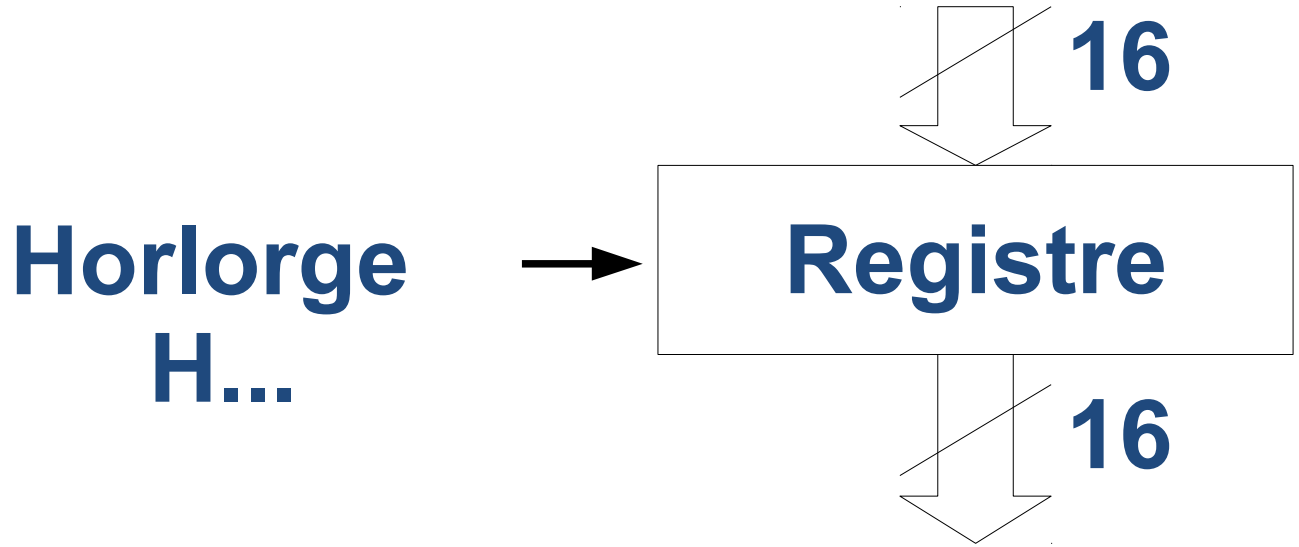
- Buffer 16 bits,
- Registre 16 bits,
- Bloc de registres 16 bits,
- Unité Arithmétique et Logique,
- Mémoire.

4.6.1 Buffer 16 Bits



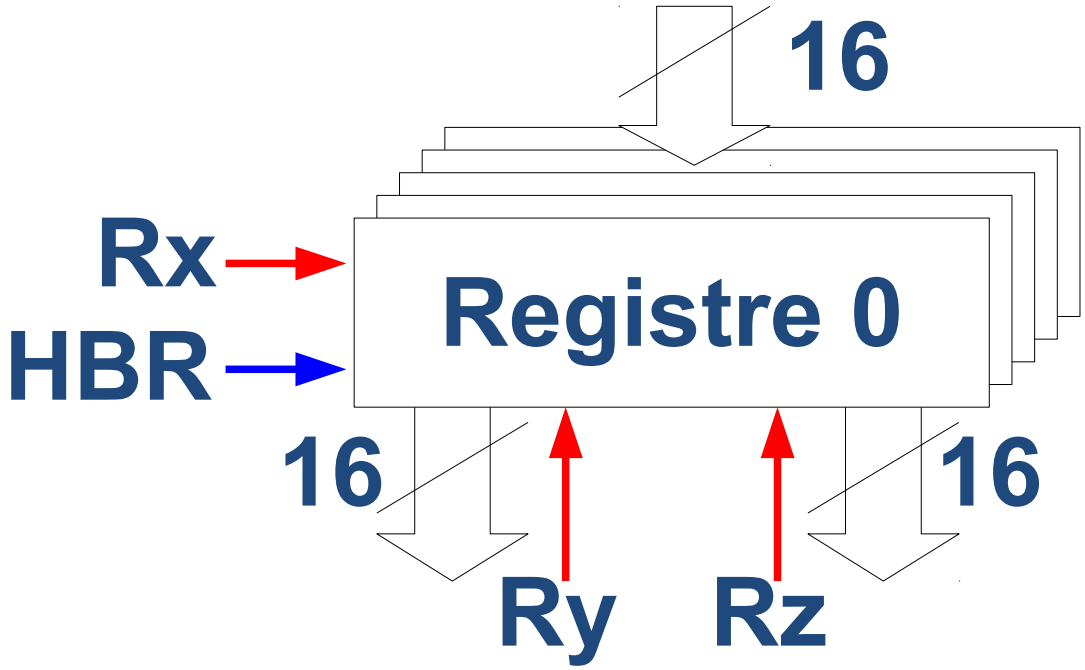
Les sorties fournissent les entrées si Vxx est à 1. Sinon, elles sont non connectées

4.6.2 Registre 16 Bits



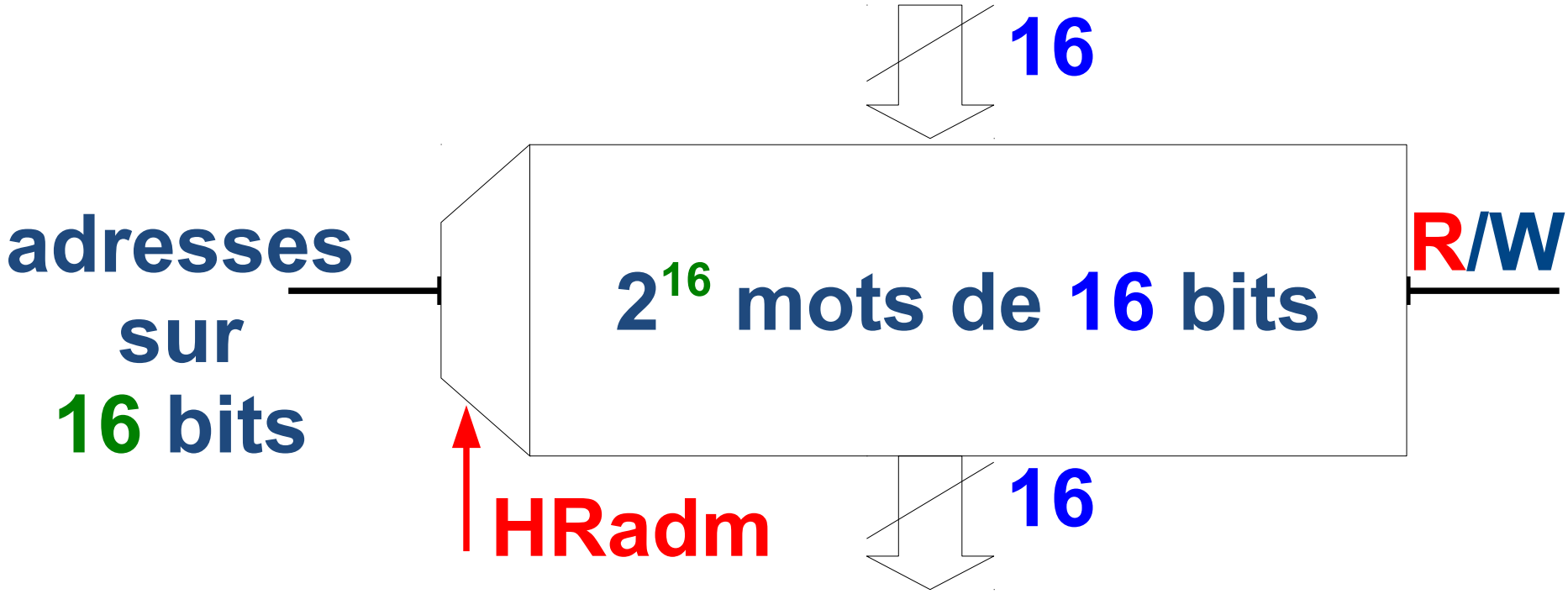
Voir 2.2.1 pour fonctionnement
Sorties toujours accessibles
→ H… reste inactif pendant la lecture

4.6.3 Bloc de registres 16 Bits



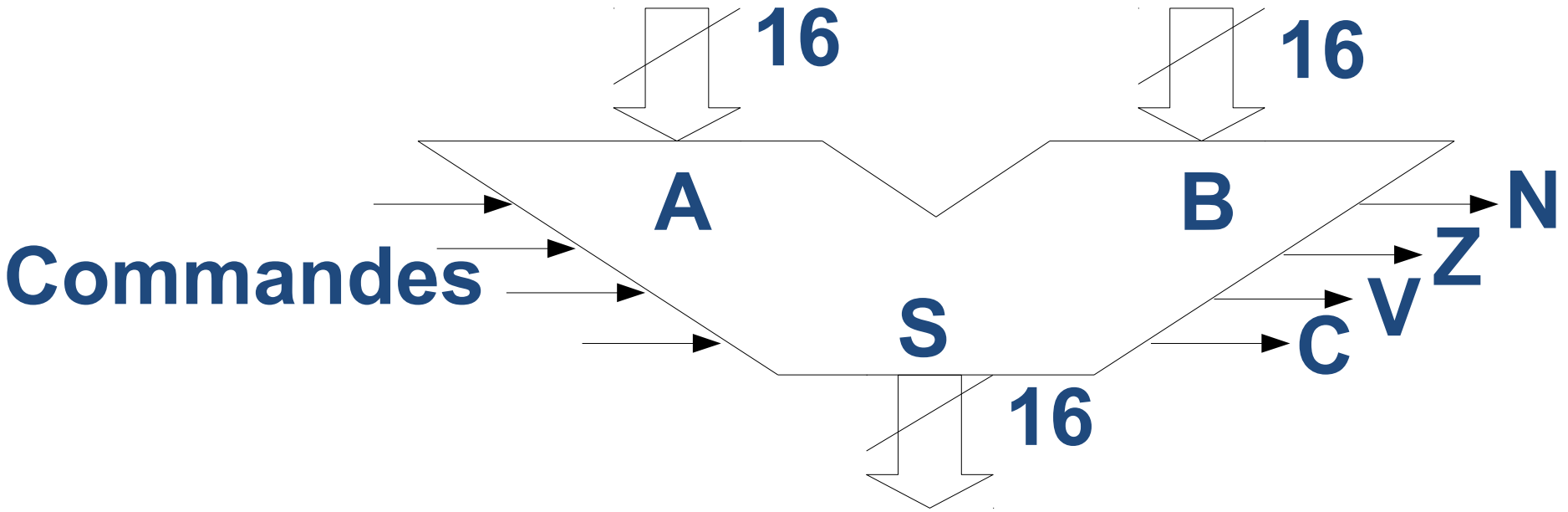
N registres dont 2 en lecture (choisis par Ry et Rz), et 1 en écriture (Rx)

4.6.4 Mémoire



Voir 2.2.2 pour fonctionnement
Sorties toujours accessibles,
Adr., Entrées et R/W stables durant l'accès

4.6.5 U.A.L



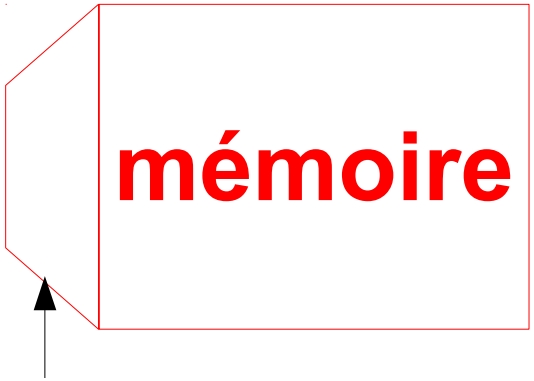
S et (N,Z,V,C) sont stables tant que les entrées (A, B, Commandes) le sont.

4.7 Conception CDD (1/15)

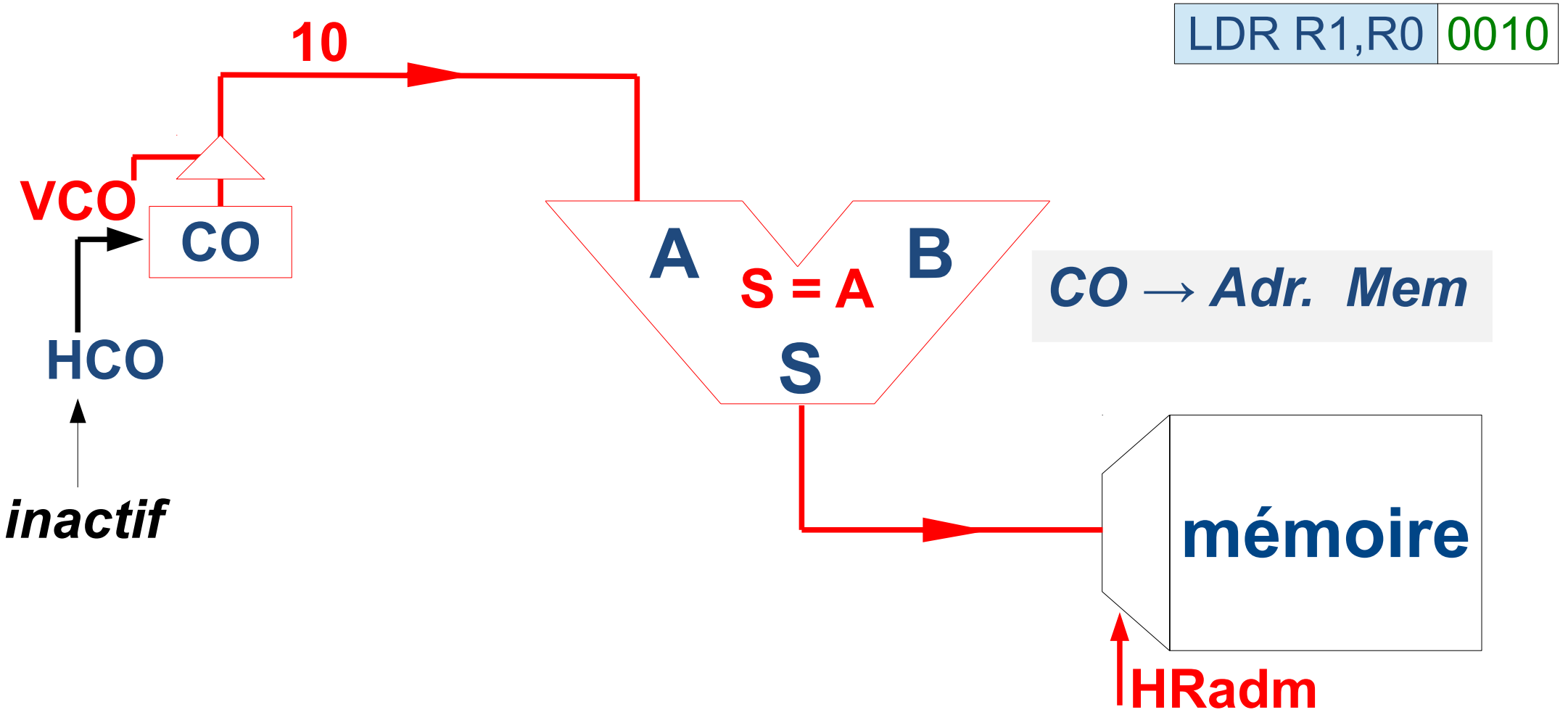
Affiché en rouge: éléments ajoutés ou importants
Affichés en vert: éléments activatés par cette étape
Les traits ont leur signification précédente

LDR R1,R0	0010

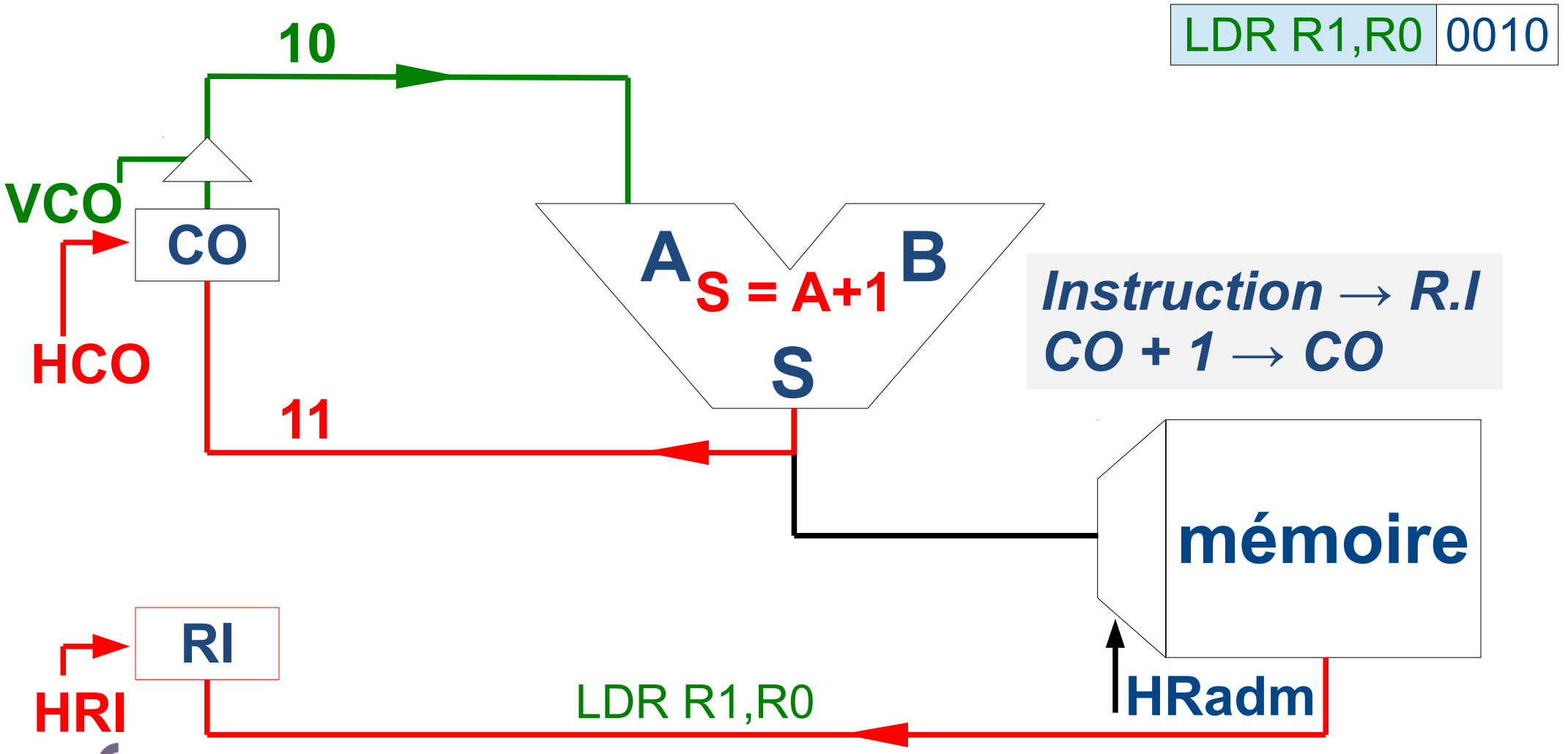
Commençons par l'exécution d'un LDR R1,R0, supposé placé à l'adresse 0010



4.7 Conception CDD (2/15)

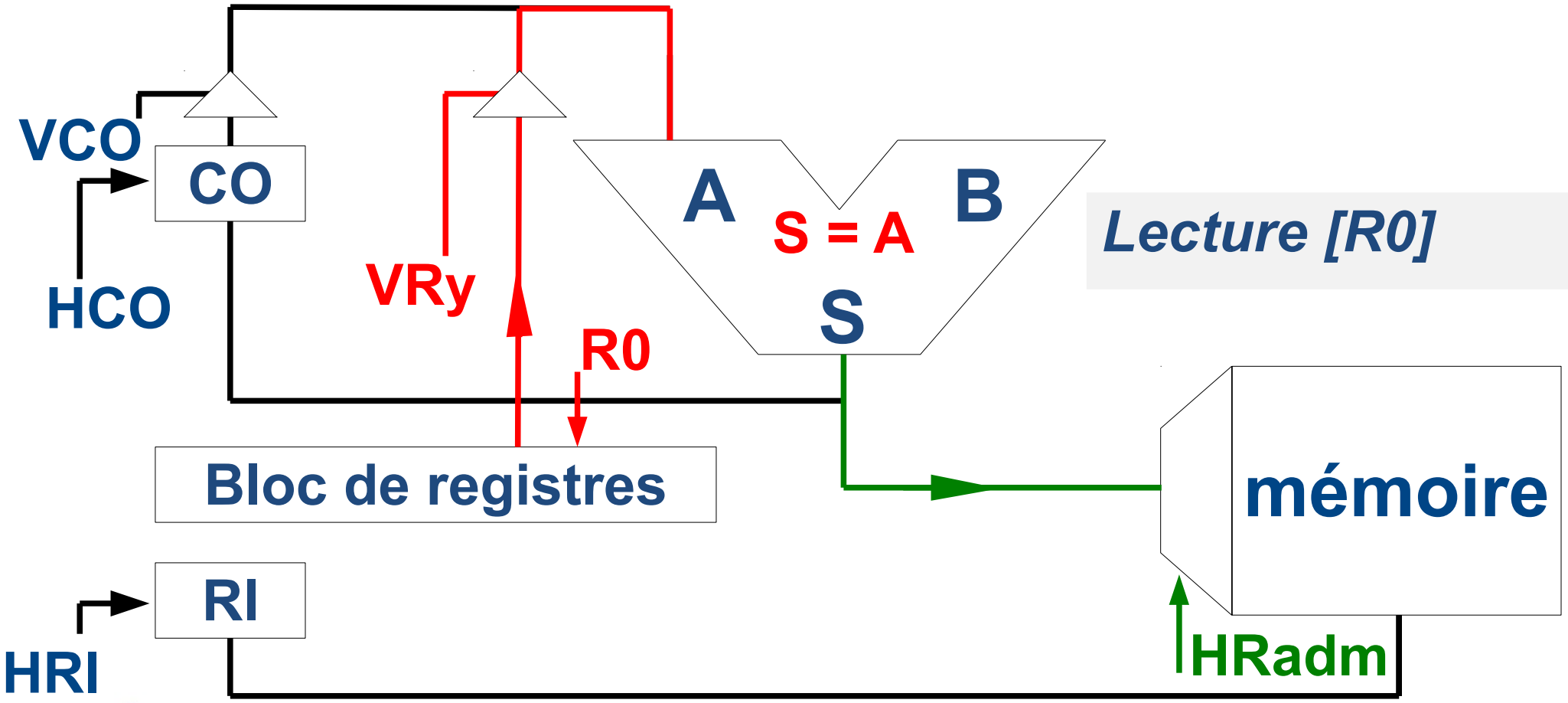


4.7 Conception CDD (3/15)



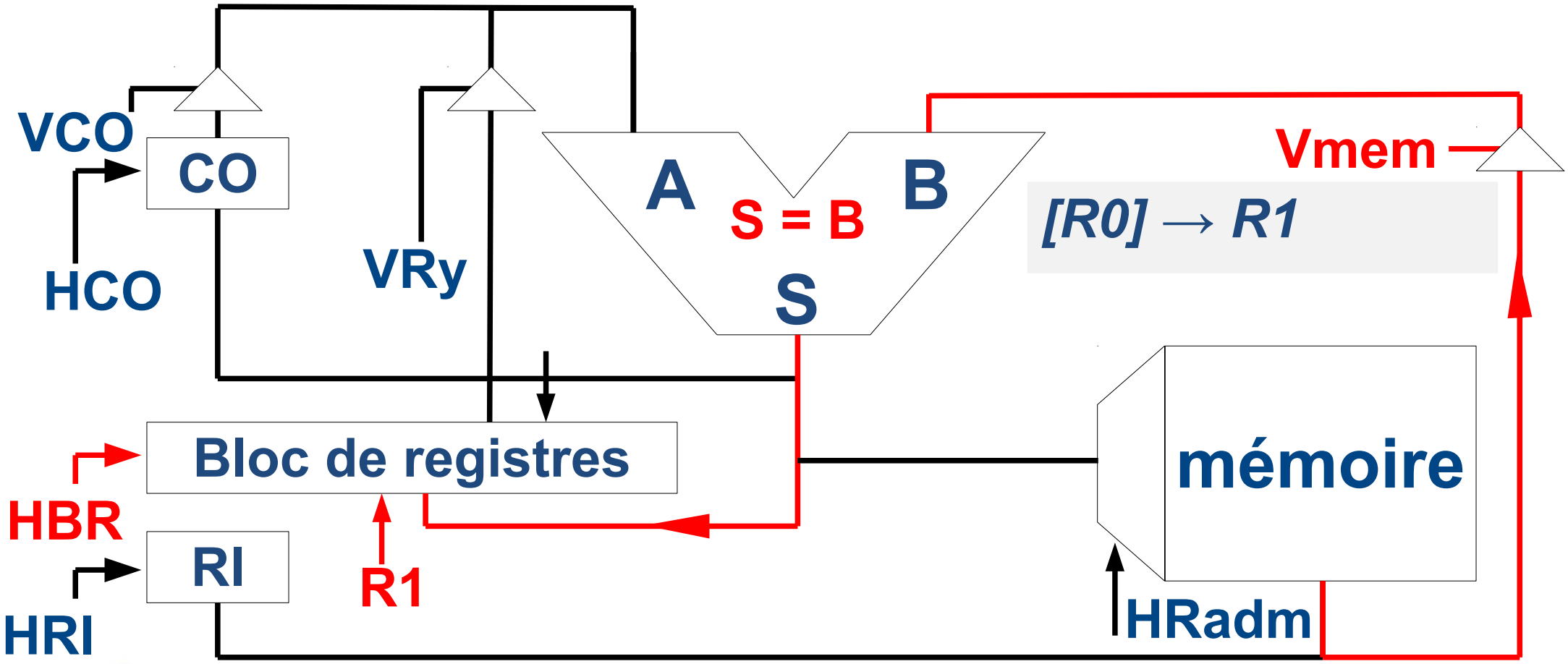
4.7 Conception CDD (4/15)

LDR R1,R0	0010

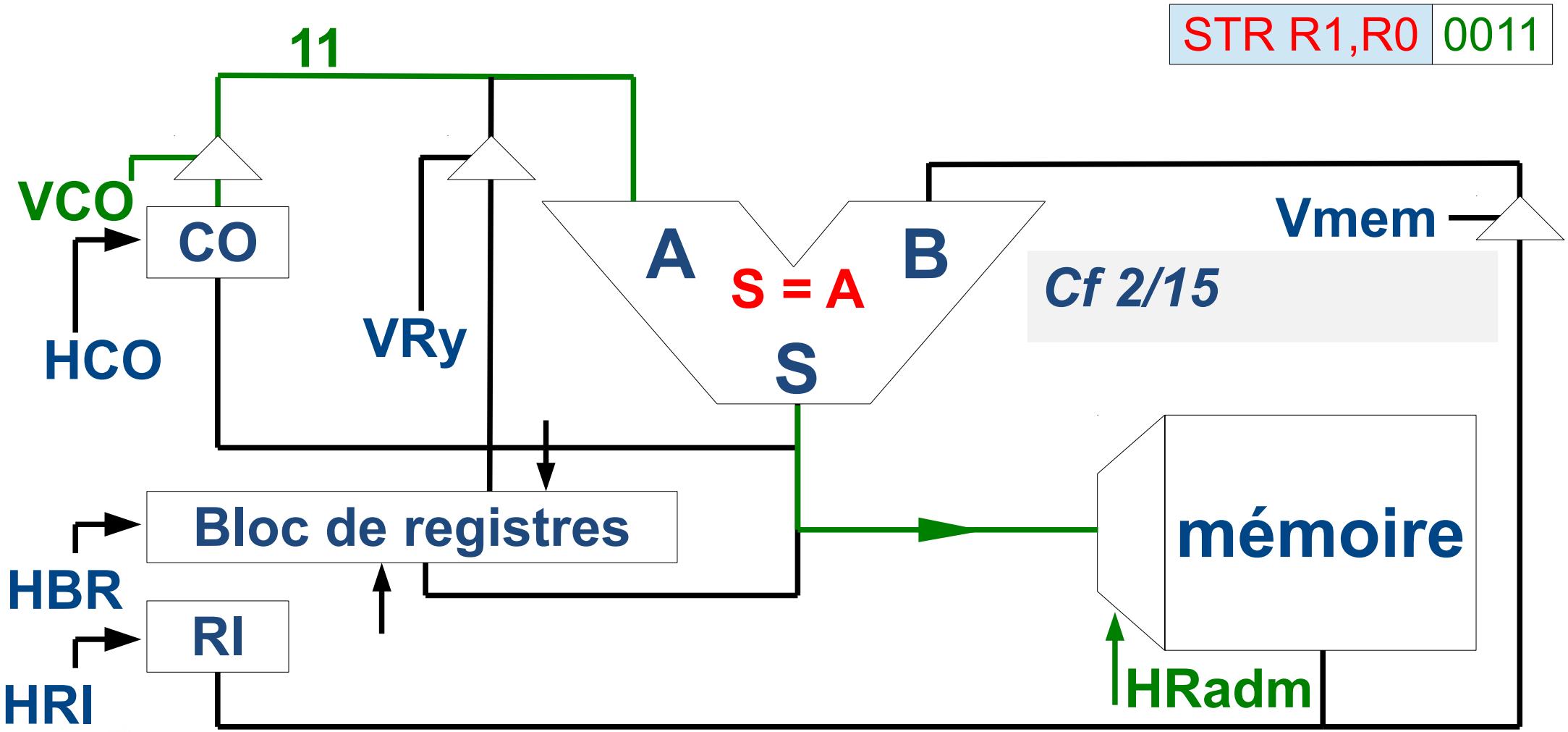


4.7 Conception CDD (5/15)

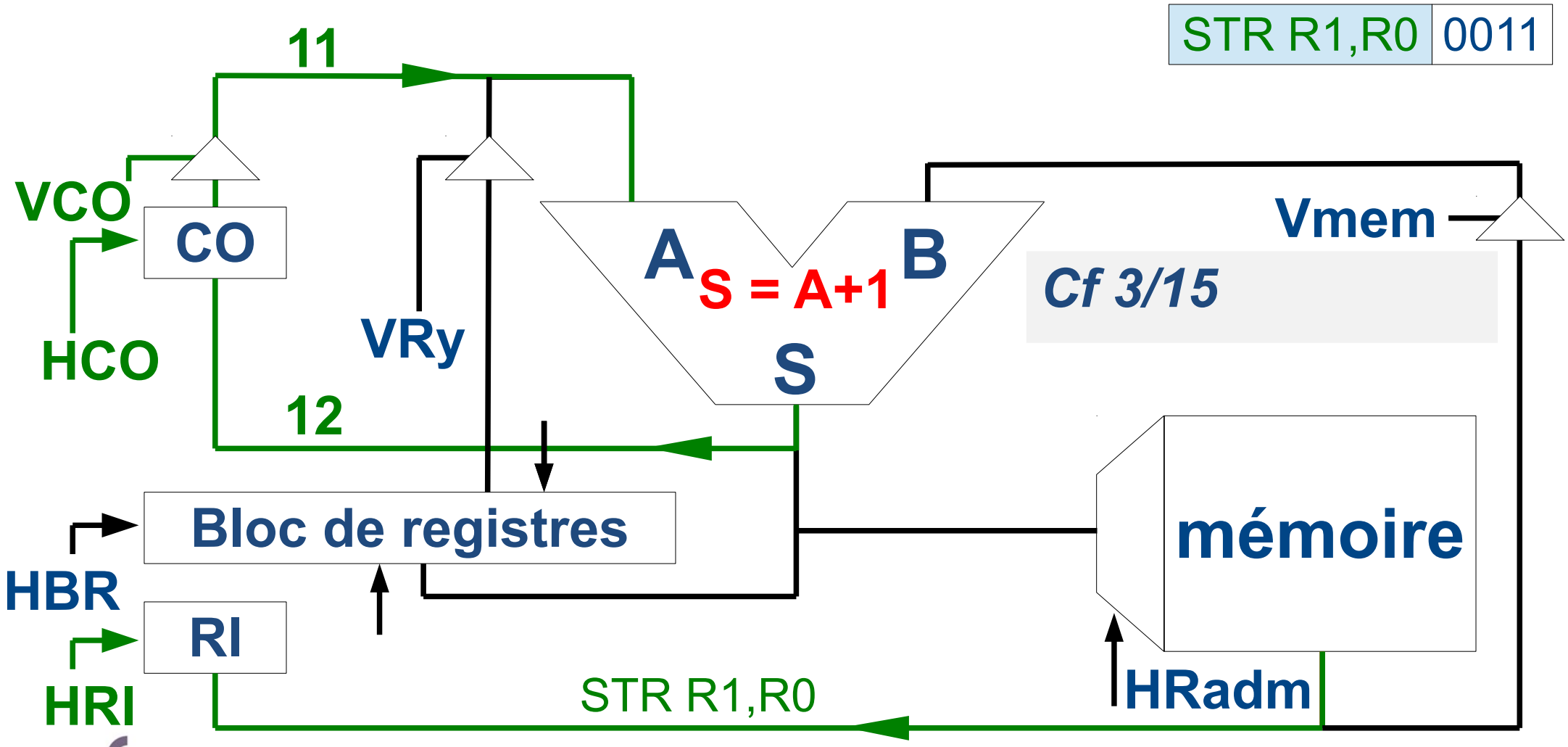
LDR R1,R0	0010



4.7 Conception CDD (6/15)

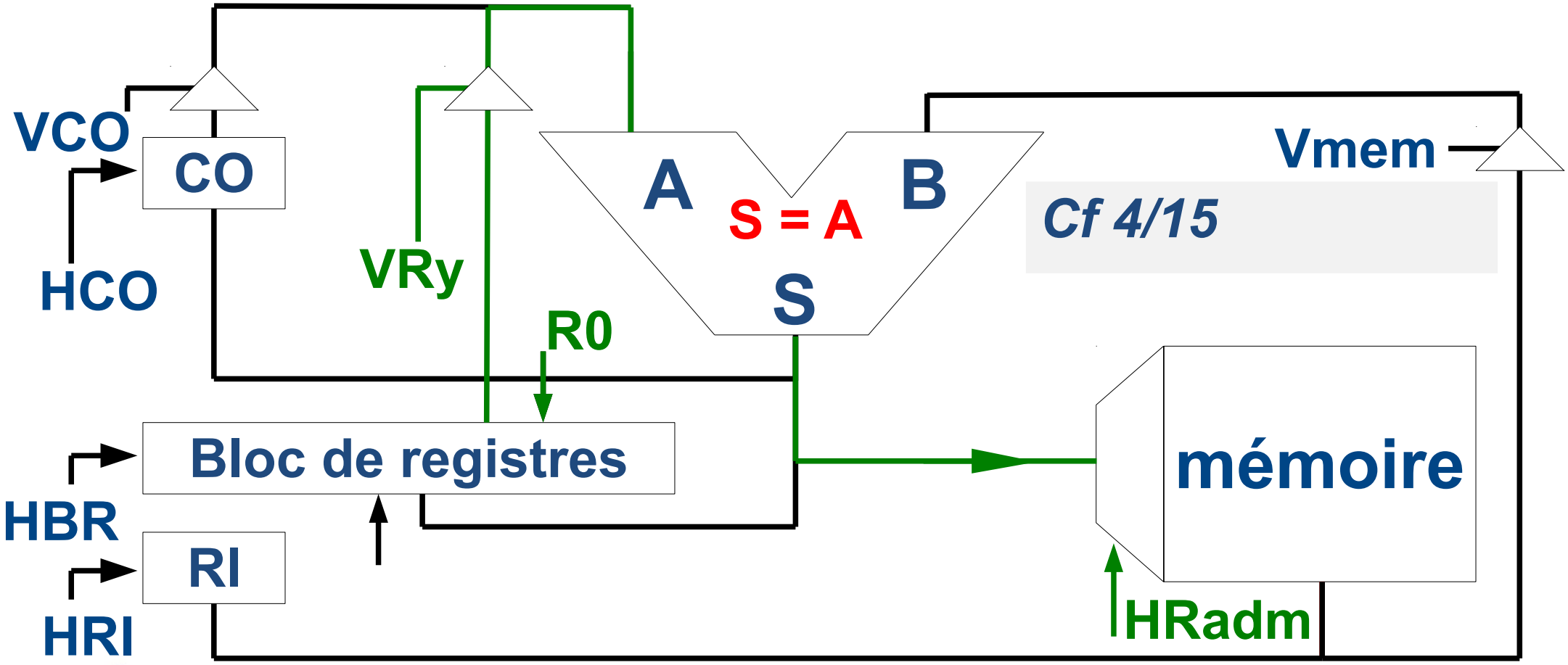


4.7 Conception CDD (7/15)

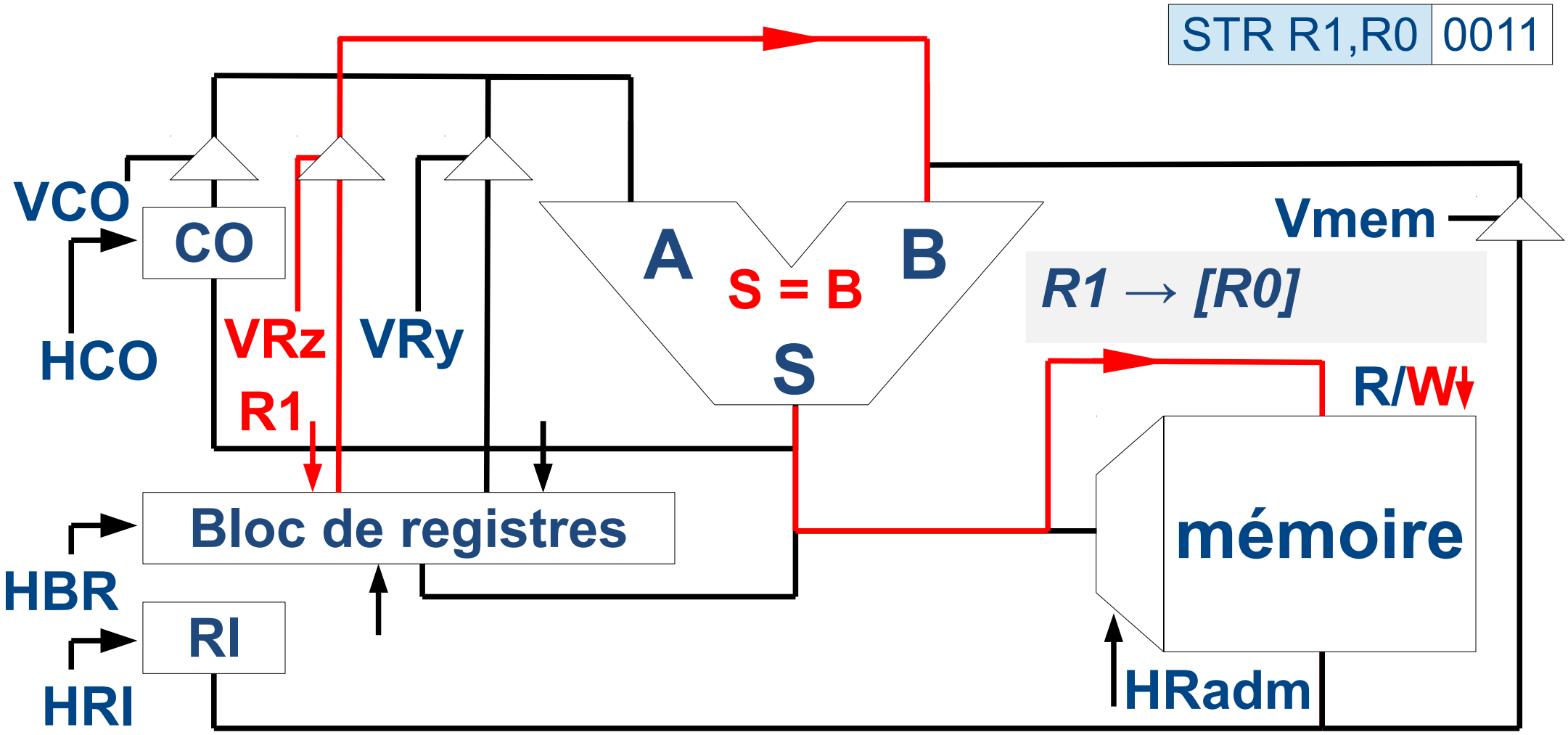


4.7 Conception CDD (8/15)

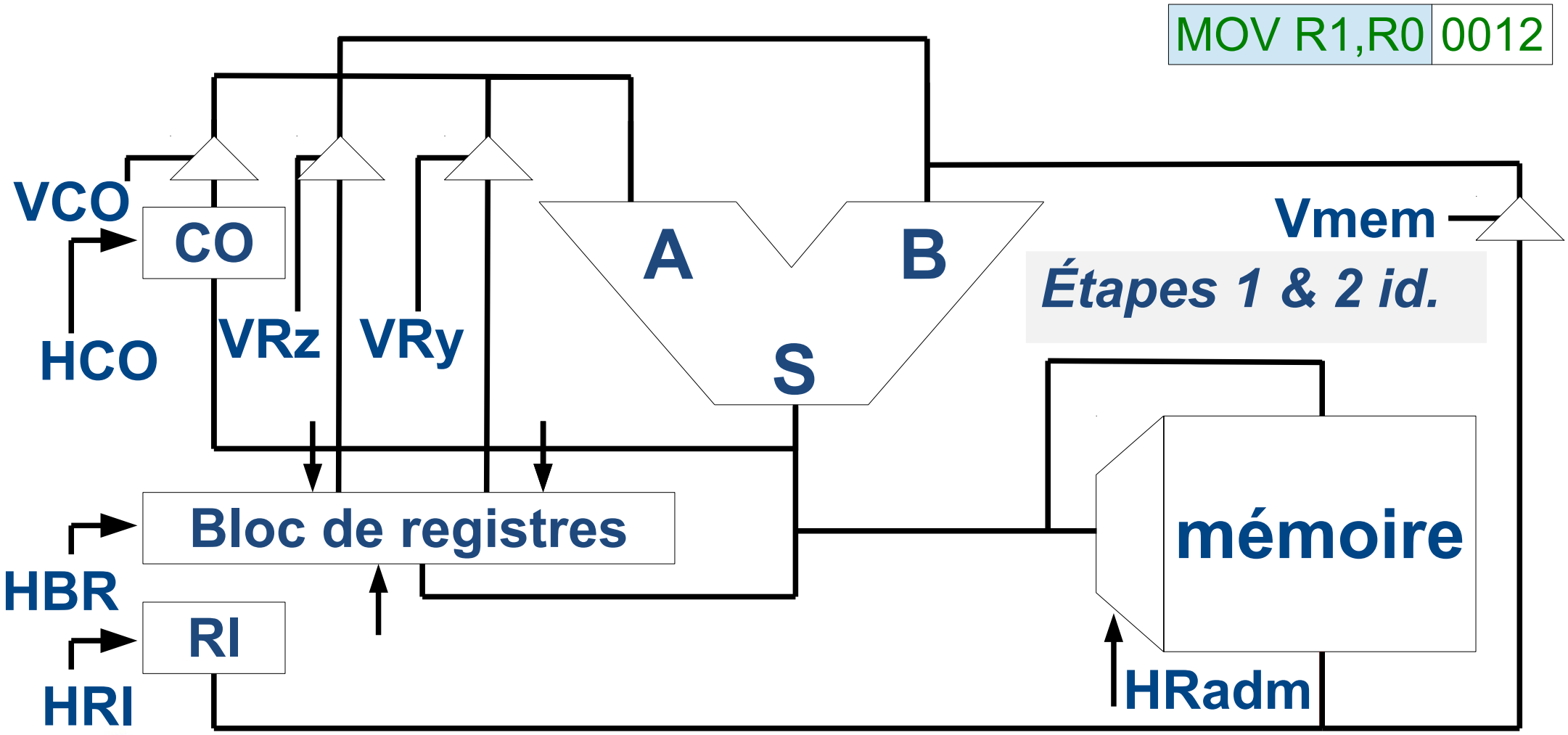
STR R1,R0	0011



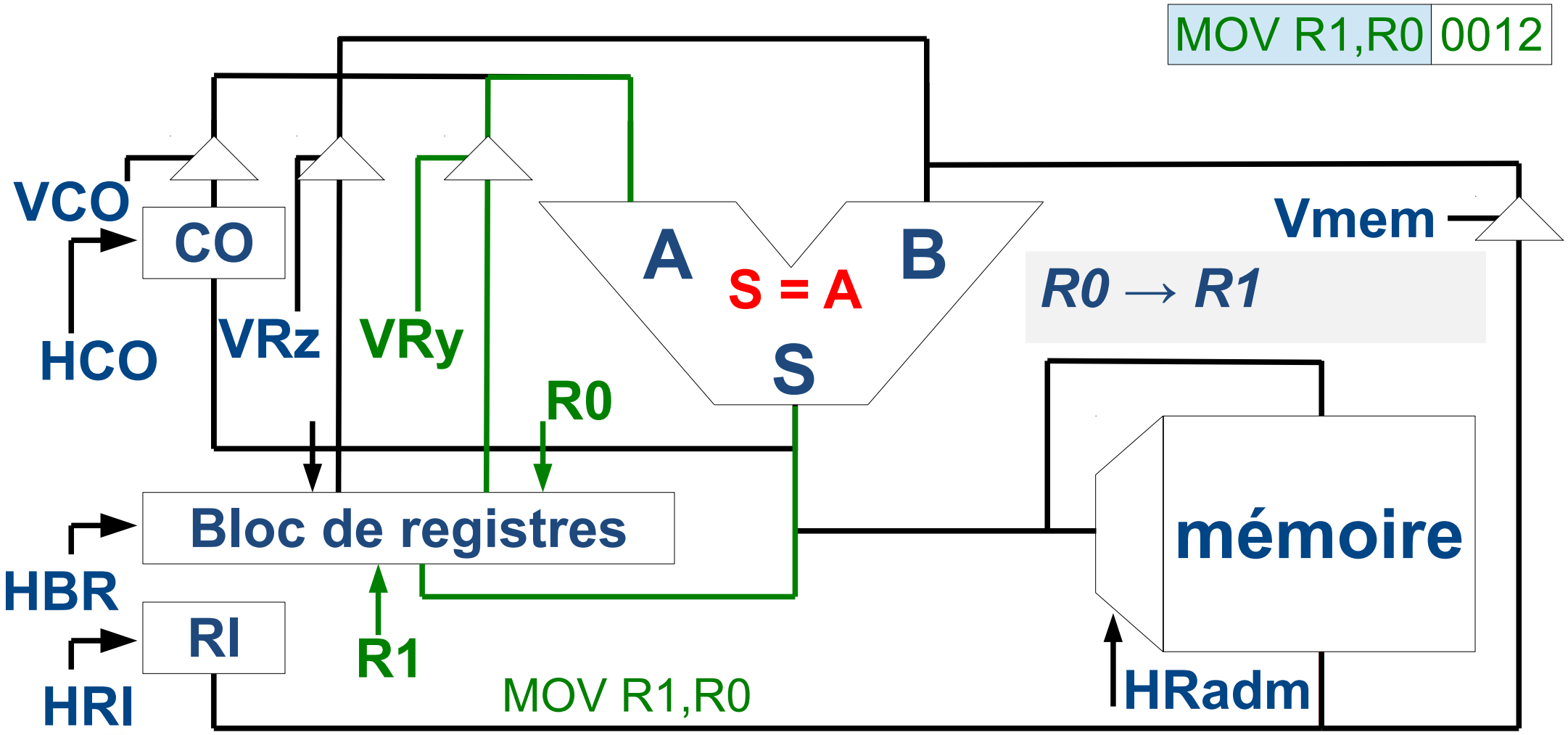
4.7 Conception CDD (9/15)



4.7 Conception CDD (10/15)

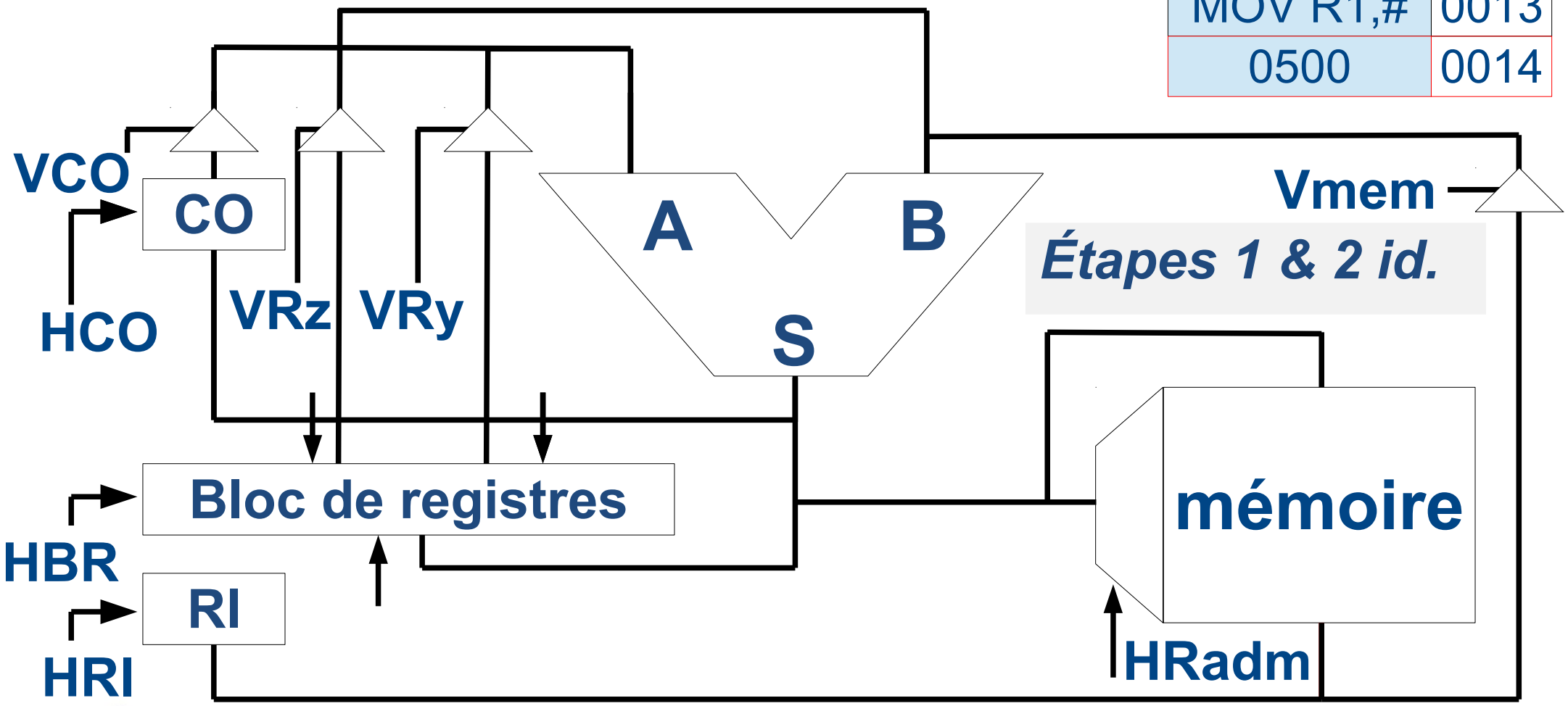


4.7 Conception CDD (11/15)

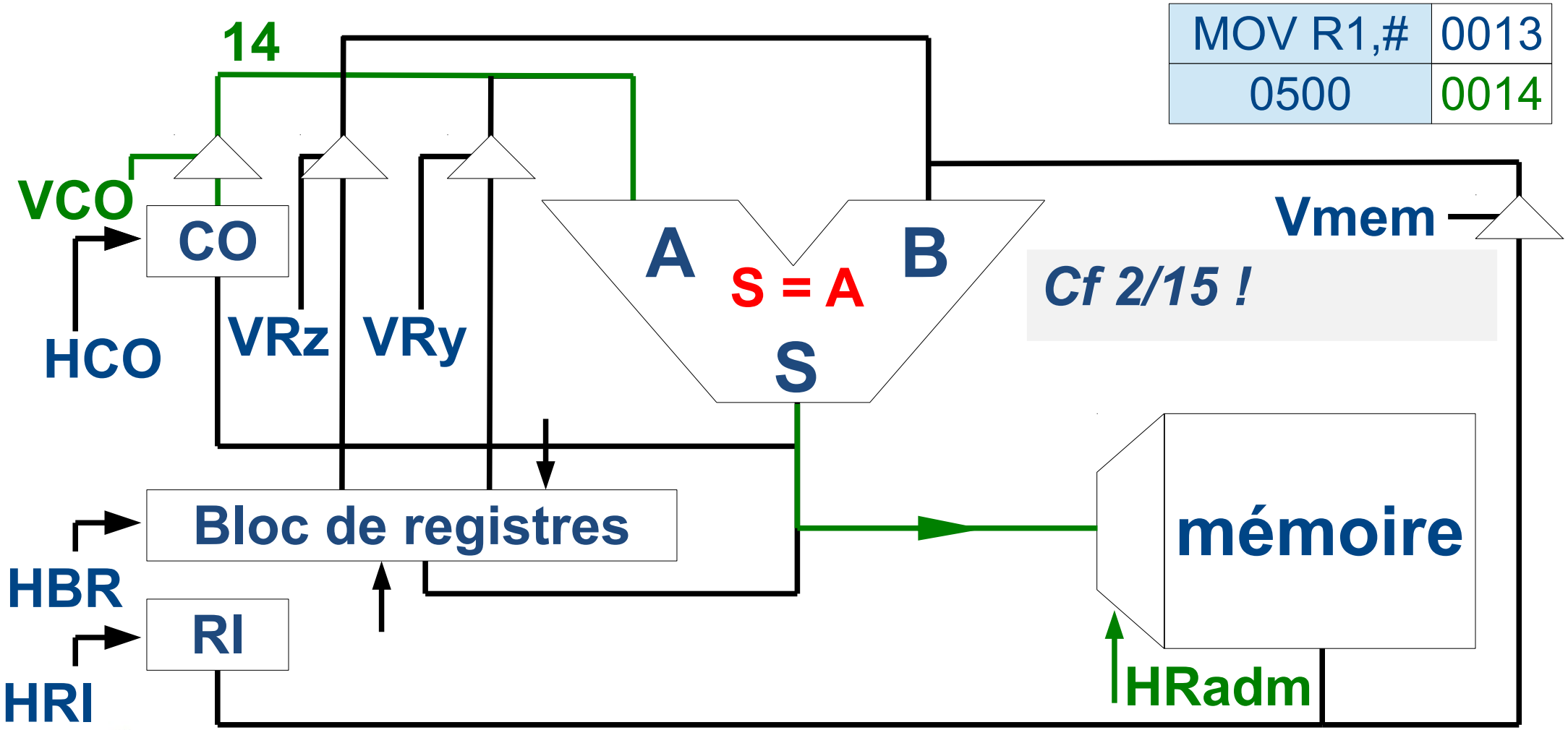


4.7 Conception CDD (12/15)

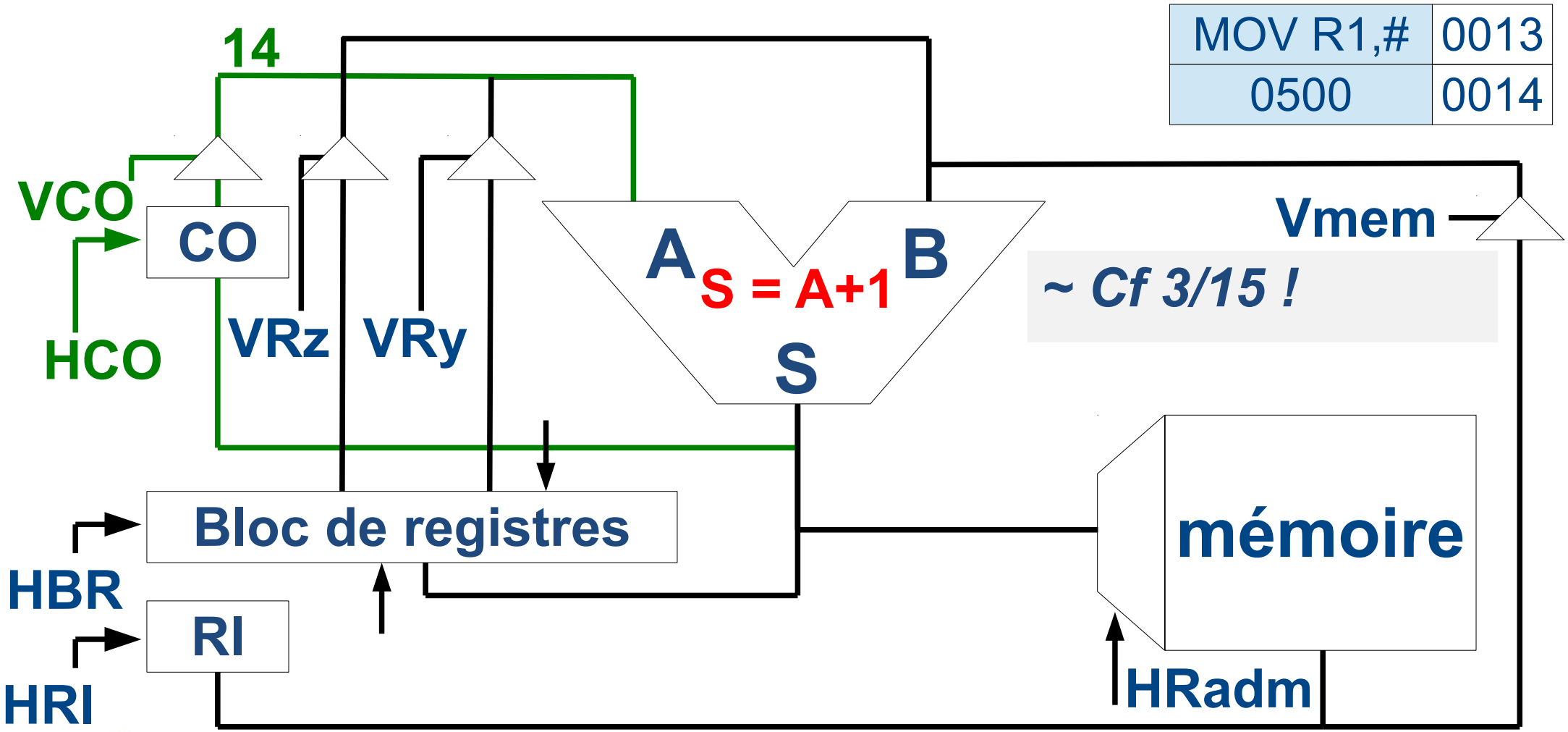
MOV R1,#	0013
0500	0014



4.7 Conception CDD (13/15)

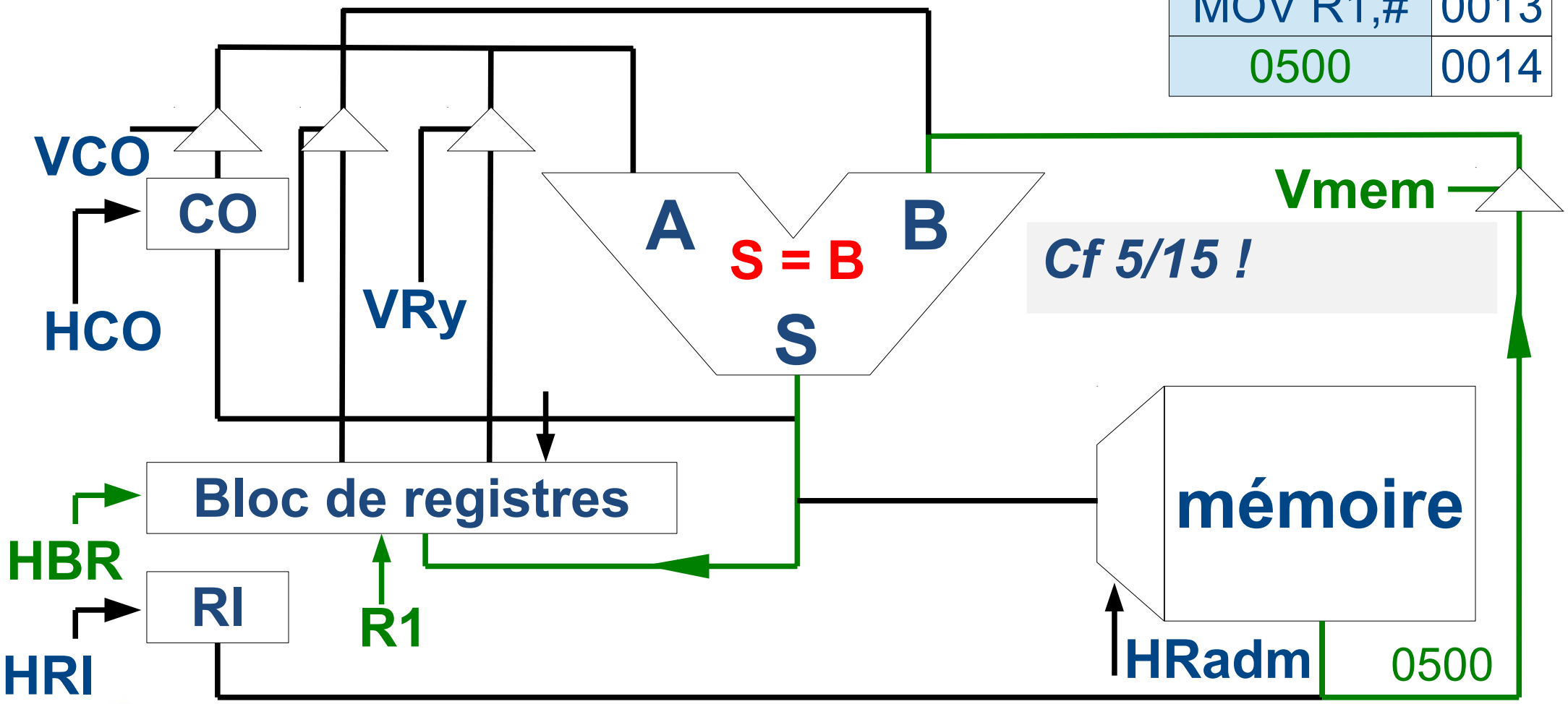


4.7 Conception CDD (14/15)



4.7 Conception CDD (15/15)

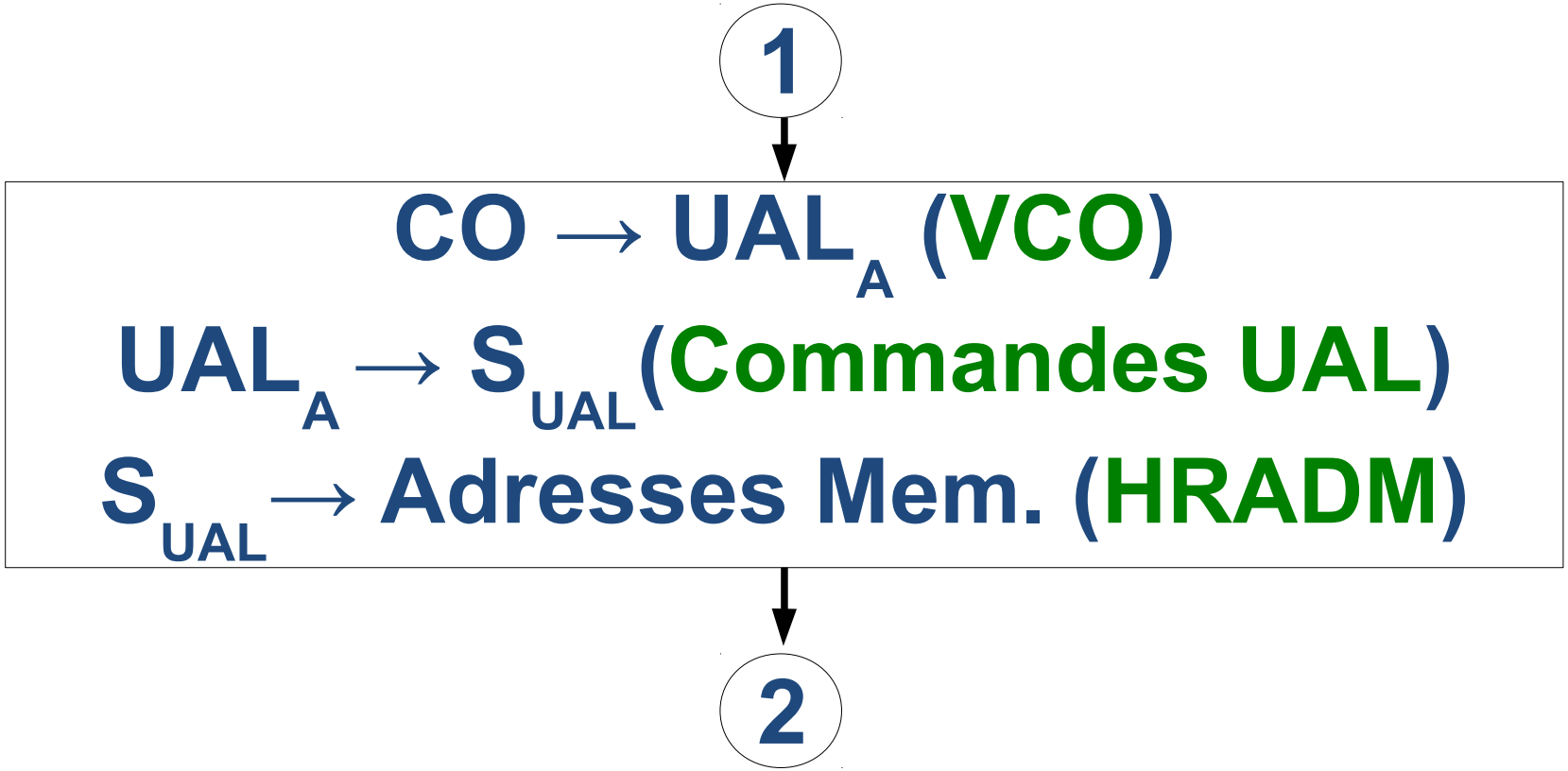
MOV R1,#	0013
0500	0014



4.8 Séquenceur

Pour exécuter toute instruction,
En plusieurs étapes,
Par des transferts entre modules,
En activant des signaux.
Diagramme d'états.

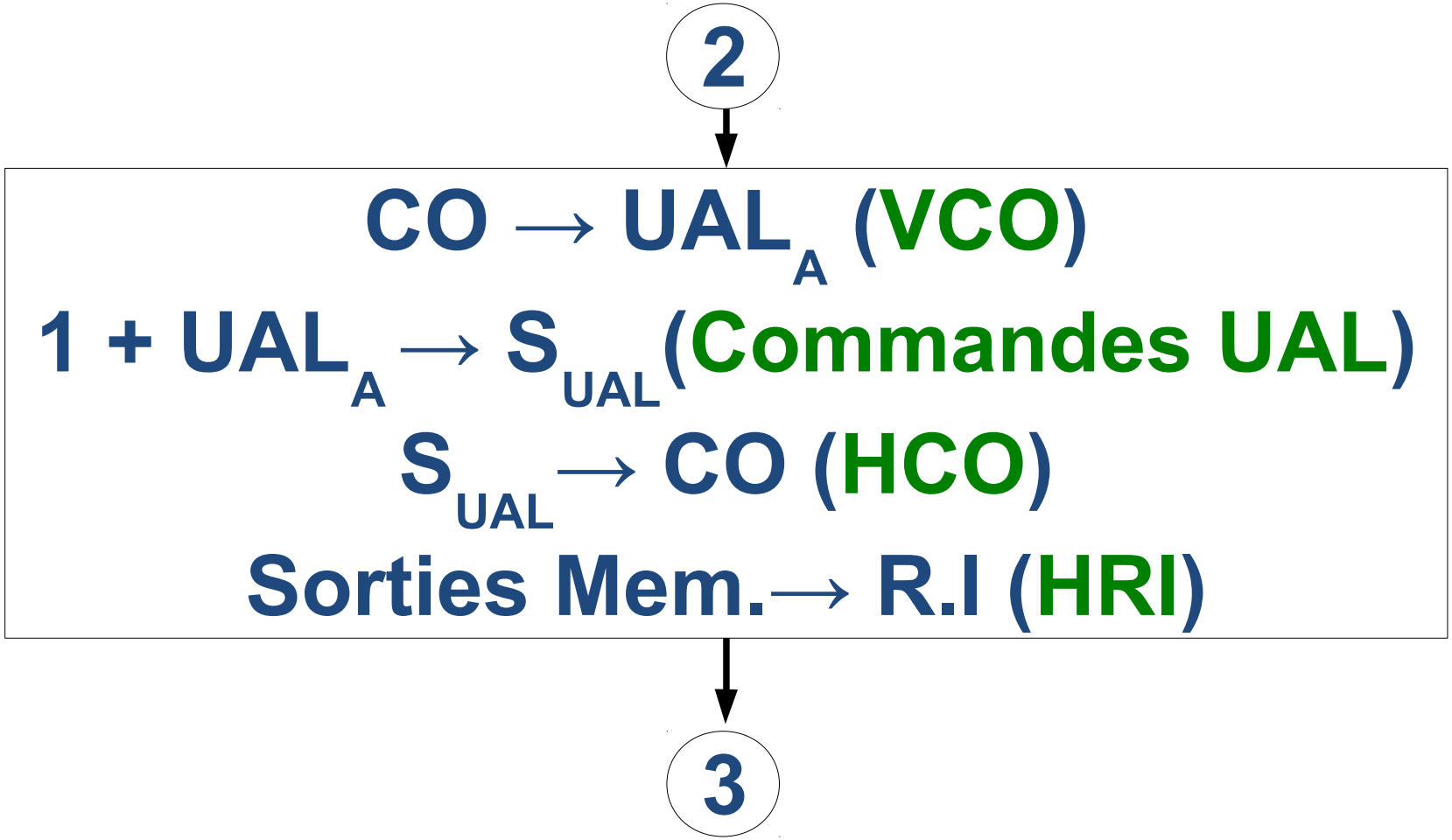
4.8.1 LDR R1,R0



2/15

T1

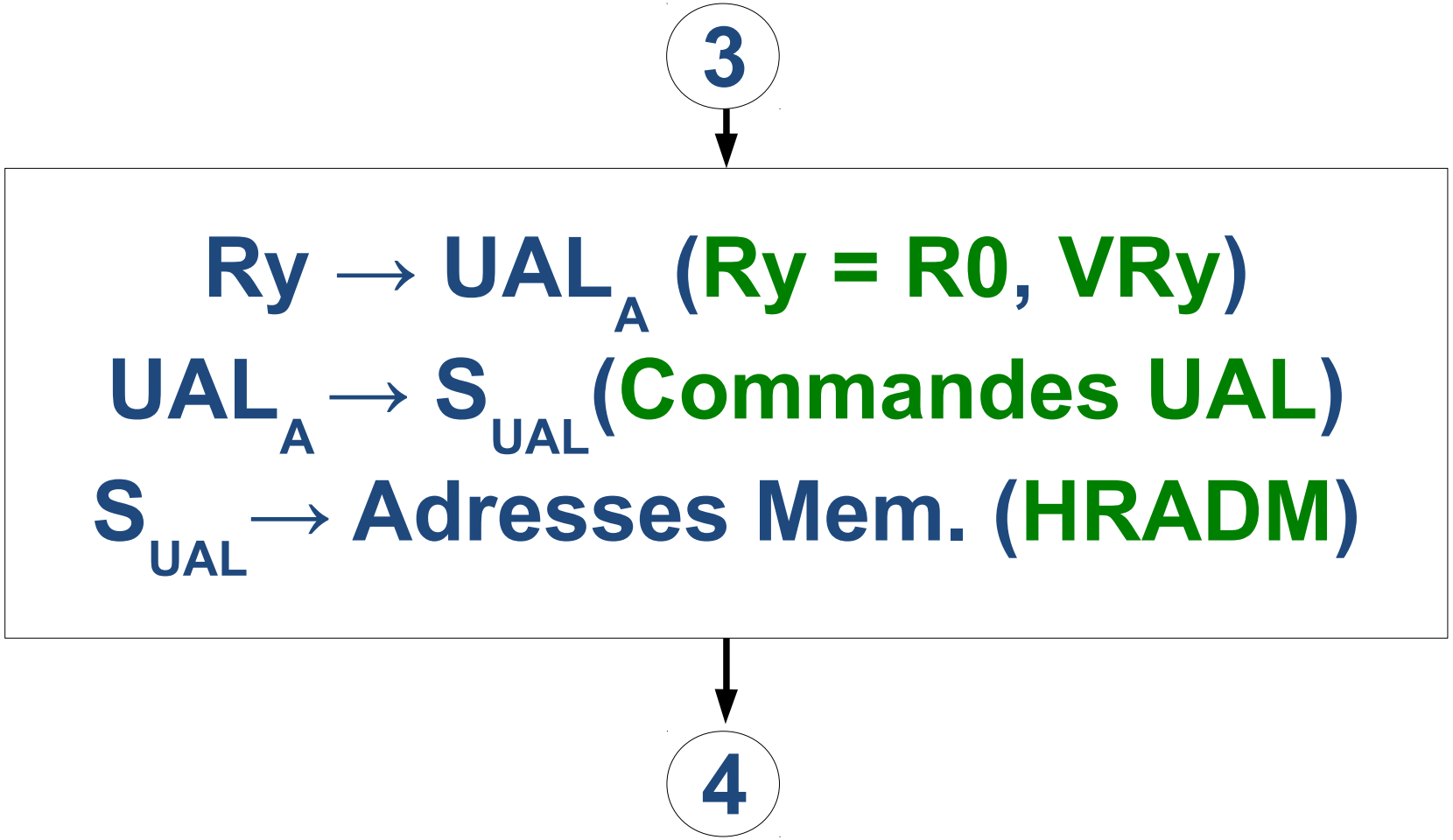
4.8.1 LDR R1,R0



3/15

T2

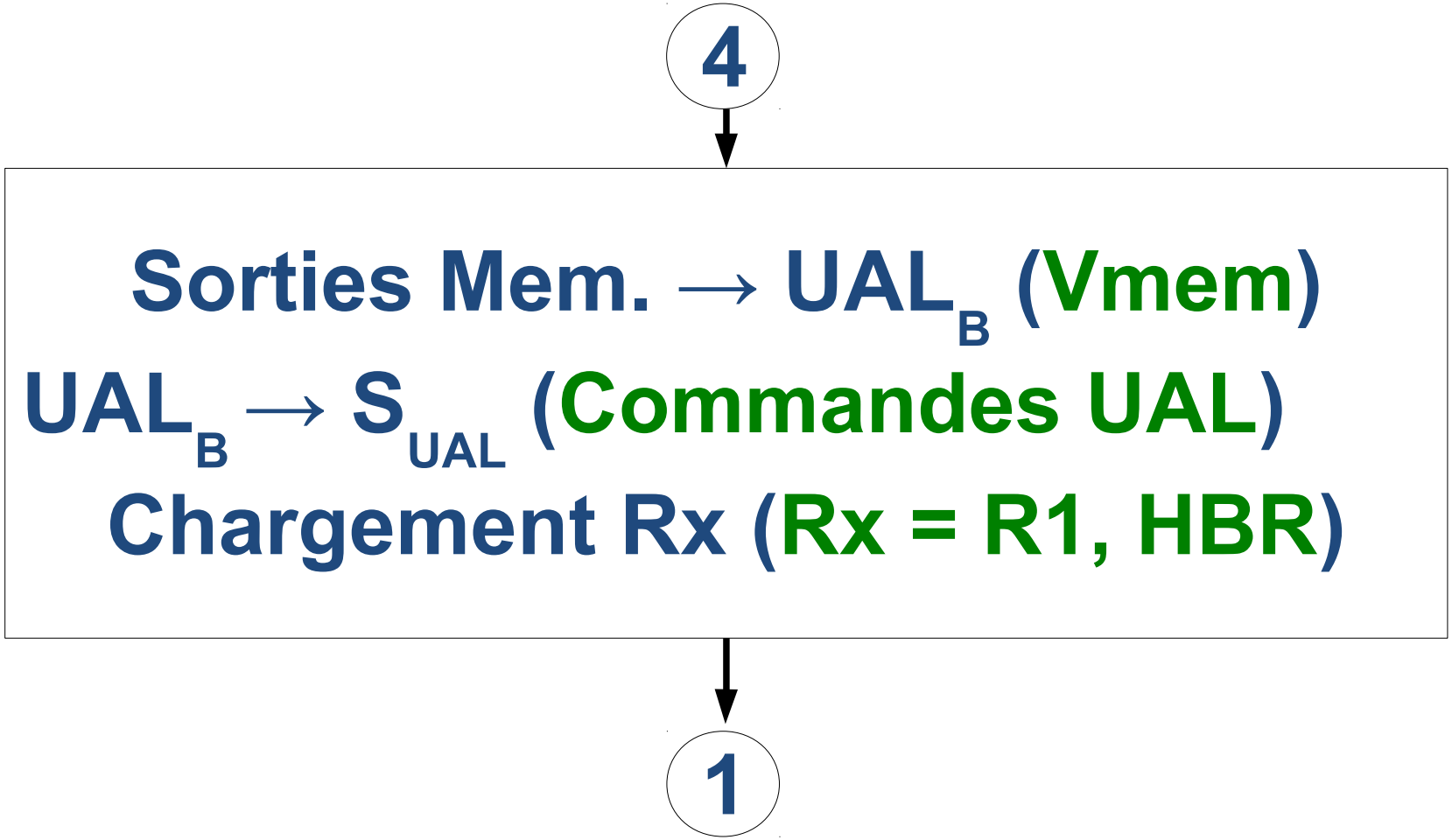
4.8.1 LDR R1,R0



4/15

T3

4.8.1 LDR R1,R0



5/15

T4

4.8.2 STR R1,R0

Pour un STR R1,R0,

- T1, T2 et T3 sont identiques

9/15

$Rz \rightarrow UAL_B$ (Rz = R1, VRz)

$UAL_B \rightarrow S_{UAL}$ (Commandes UAL)

Ecriture mémoire (R/W à 0)

T4

4.8.3 MOV R1,R0

- T1, T2 sont identiques,
- On ne fera rien en T3,

11/15

T4

$Ry \rightarrow UAL_A$ (Ry = R0, VRy)

$UAL_A \rightarrow S_{UAL}$ (Commandes UAL)

Chargement Rx (Rx = R1, HBR)

4.8.4 MOV R1, #0500

- T1, T2 sont identiques,
- T3 identique à T1,

13/15

$CO \rightarrow UAL_A$ (VCO)

$UAL_A \rightarrow S_{UAL}$ (Commandes UAL)

$S_{UAL} \rightarrow$ Adresses Mem. (HRADM)

T3

4.8.4 MOV R1, #0500

- T4 ressemble à T2,

14/15

$CO \rightarrow UAL_A$ (VCO)

$1 + UAL_A \rightarrow S_{UAL}$ (Commandes UAL)

$S_{UAL} \rightarrow CO$ (HCO)

T4

4.8.4 MOV R1, #0500

- On a besoin d'un T5 !
- Identique au T4 du LDR

15/15

Sorties Mem. → UAL_B (Vmem)

UAL_A → S_{UAL} (Commandes UAL)

Chargement Rx (Rx = R1, HBR)

T5

5. Mémoires

5.1 Typologie
5.2 La cellule
5.3 De la cellule à …
5.4 Mémoires de masse
5.5 Caches
5.6 Valeurs usuelles

5.1 Typologie

- Par localisation
  - interne/externe
- Par mode d'accès
- Par capacité d'écriture
- Par comportement hors-tension

5.1.1 Mode d'accès

Quel mot est accessible, compte tenu de l'accès précédent ?

Tous: **R**andom **A**ccess **M**emory

Celui écrit il y a le plus de temps:

First **I**n **F**irst **O**ut memory

(sans adresse)

5.1.1 Mode d'accès

Celui écrit il y a le moins de temps: **L**ast **I**n **F**irst **O**ut

Celui associé à la clé fournie: **C**ontent **A**ddressable **M**emory

Le mot d'adresse suivante: séquentiel, …

5.1.2 Capacité d'écriture

Read Only Memory : ROM

→ mémoire inscriptible : RAM !!

Programmable ROM,

Erasable PROM,

Electrically EPROM (“Flash”),

Flash → clés USB, “SSD”, …

5.1.4 Conservation info

Mémoire Volatile:
Perd son contenu si hors tension

	RAM	ROM
Volatile	Variable en mémoire	Instruction en mémoire
Non volatile	Disque, …	ROM, …

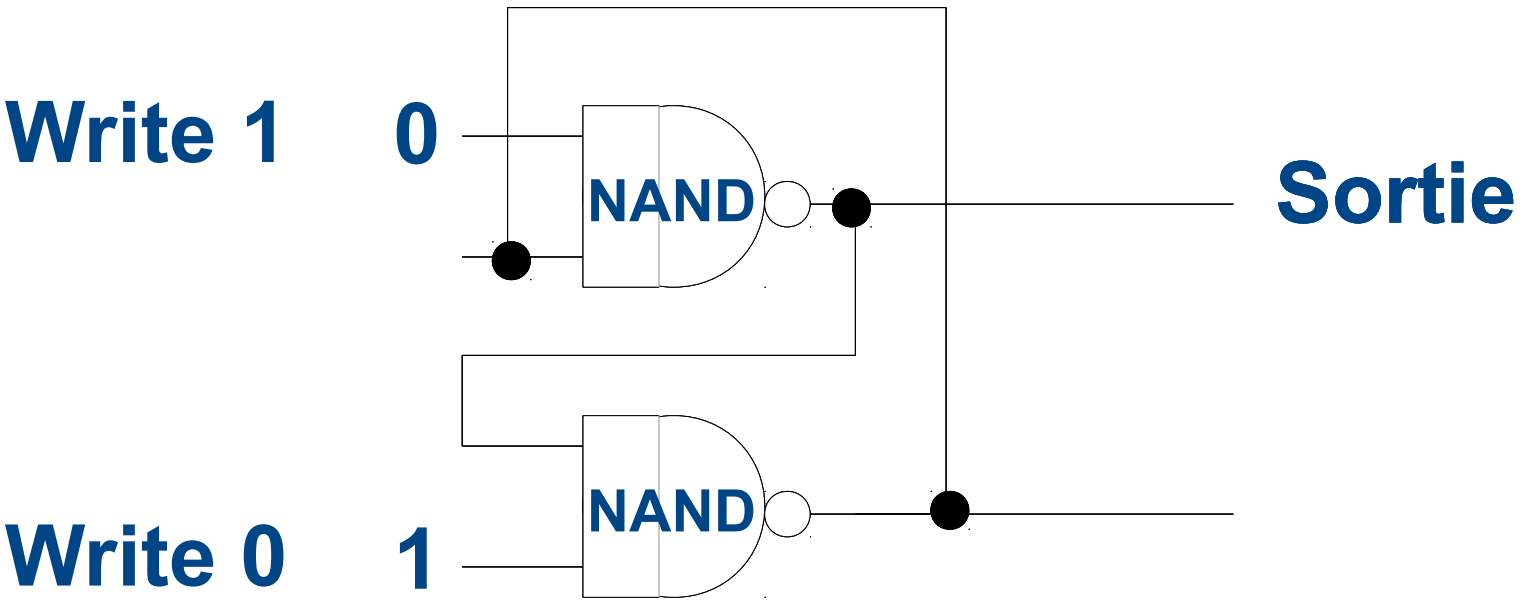
5.2 Cellule mémoire

Un “truc” propre à stocker 1 **bit**.

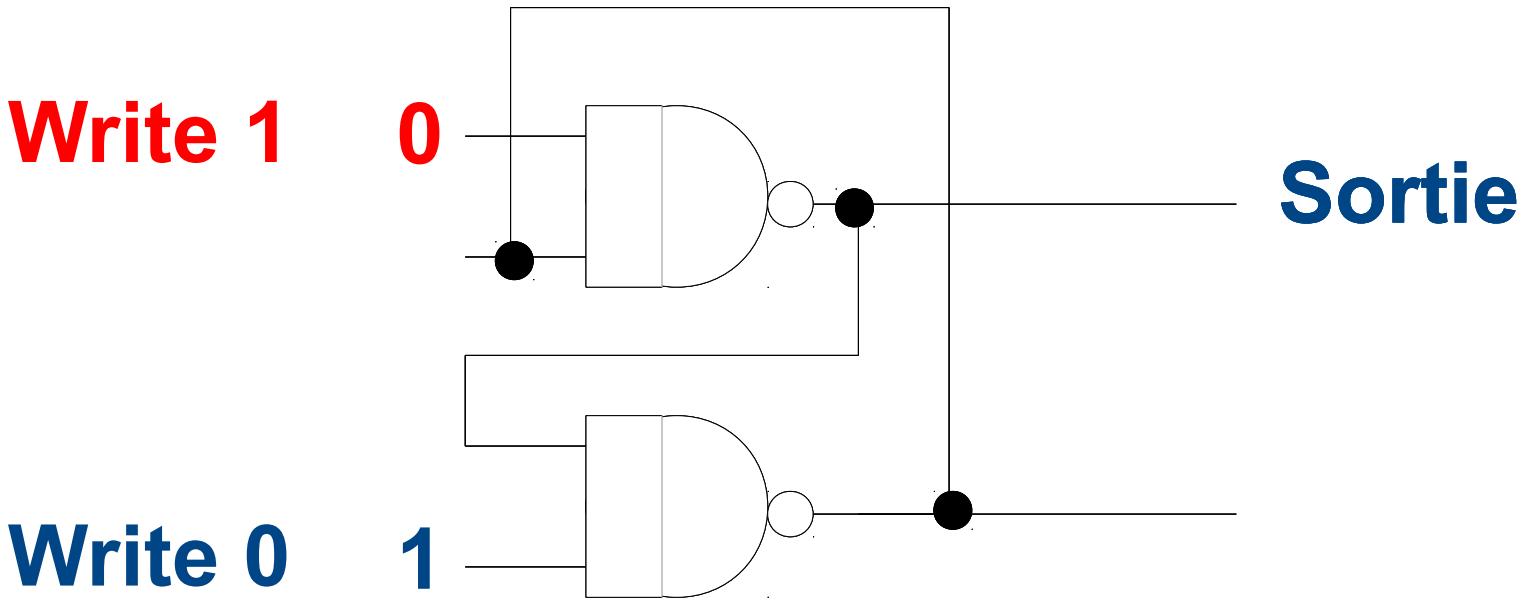
4 critères pour les comparer:

- temps d'accès
- densité (nb de cellules / puce)
- coût / cellule
- débit (≠ vitesse)

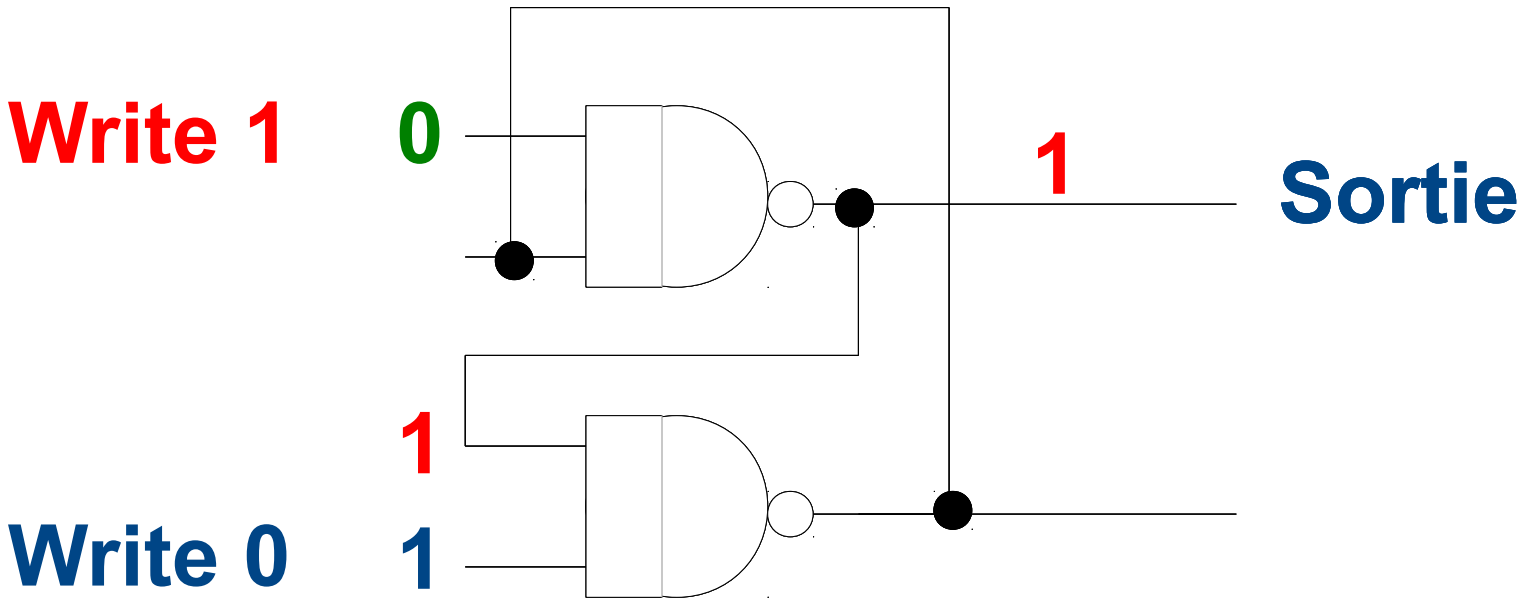
5.2.1 Bascule “RS”



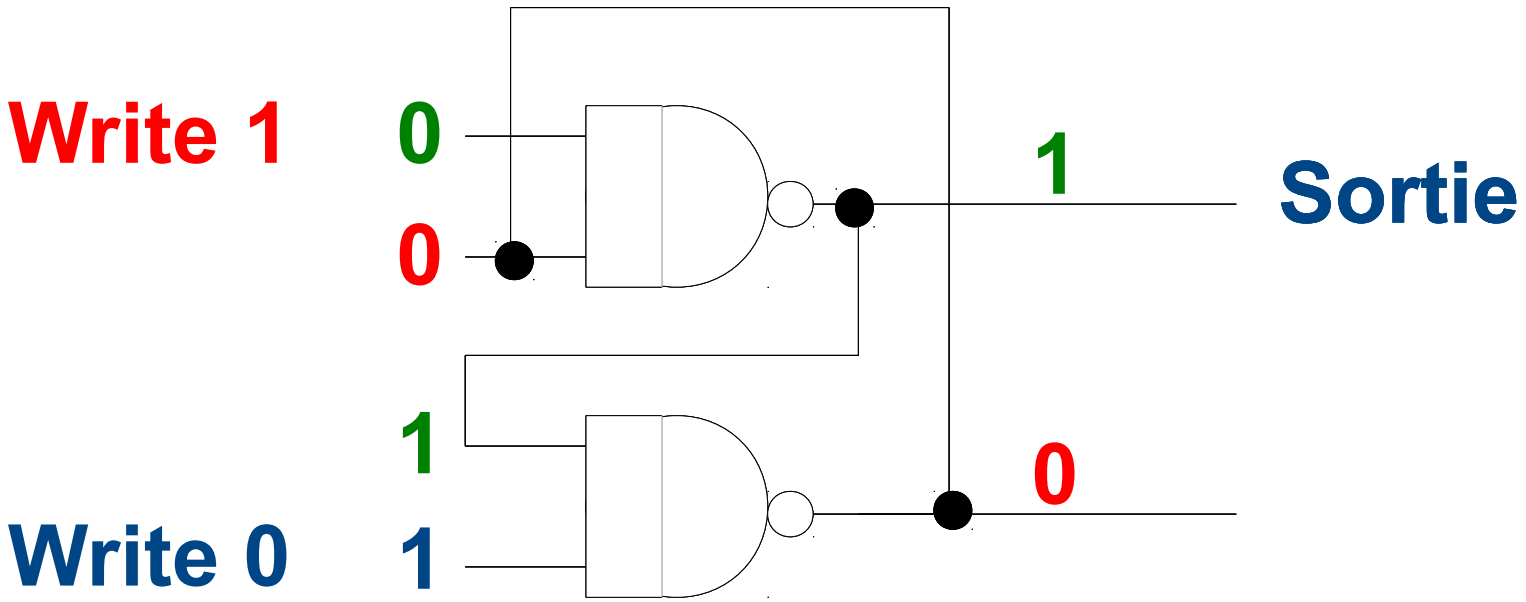
5.2.1 Bascule “RS”



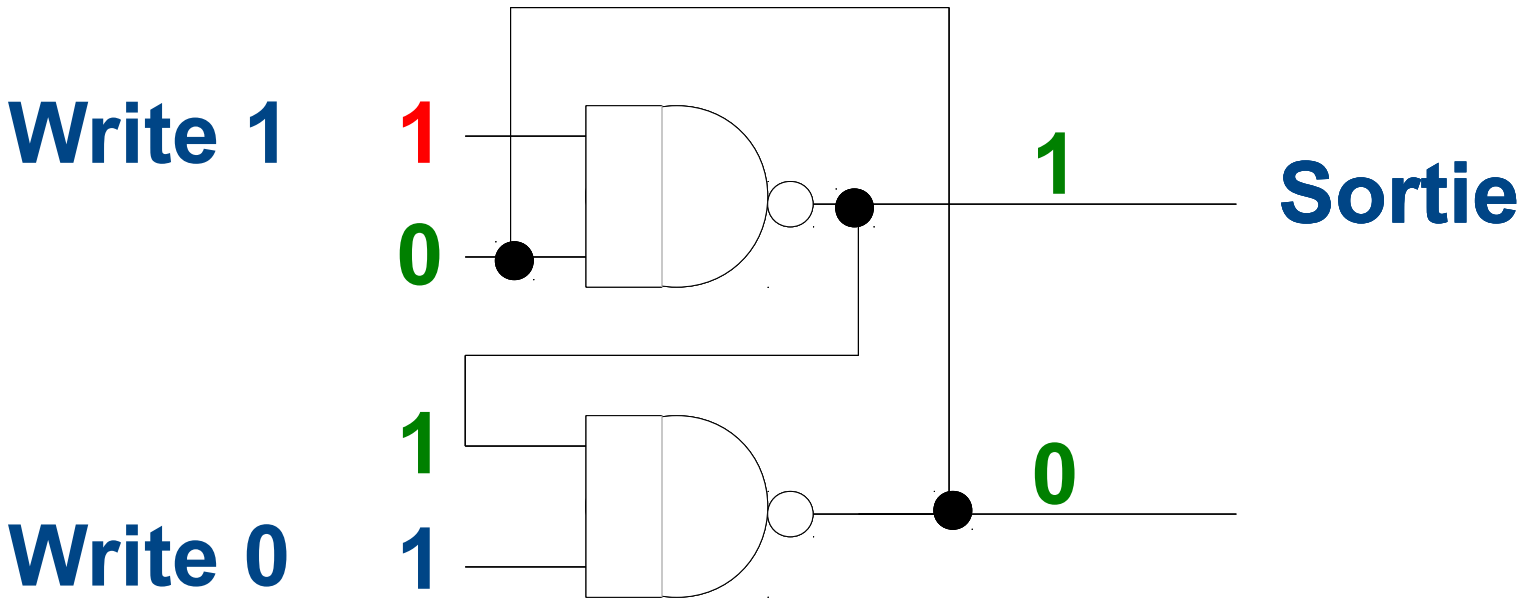
5.2.1 Bascule “RS”



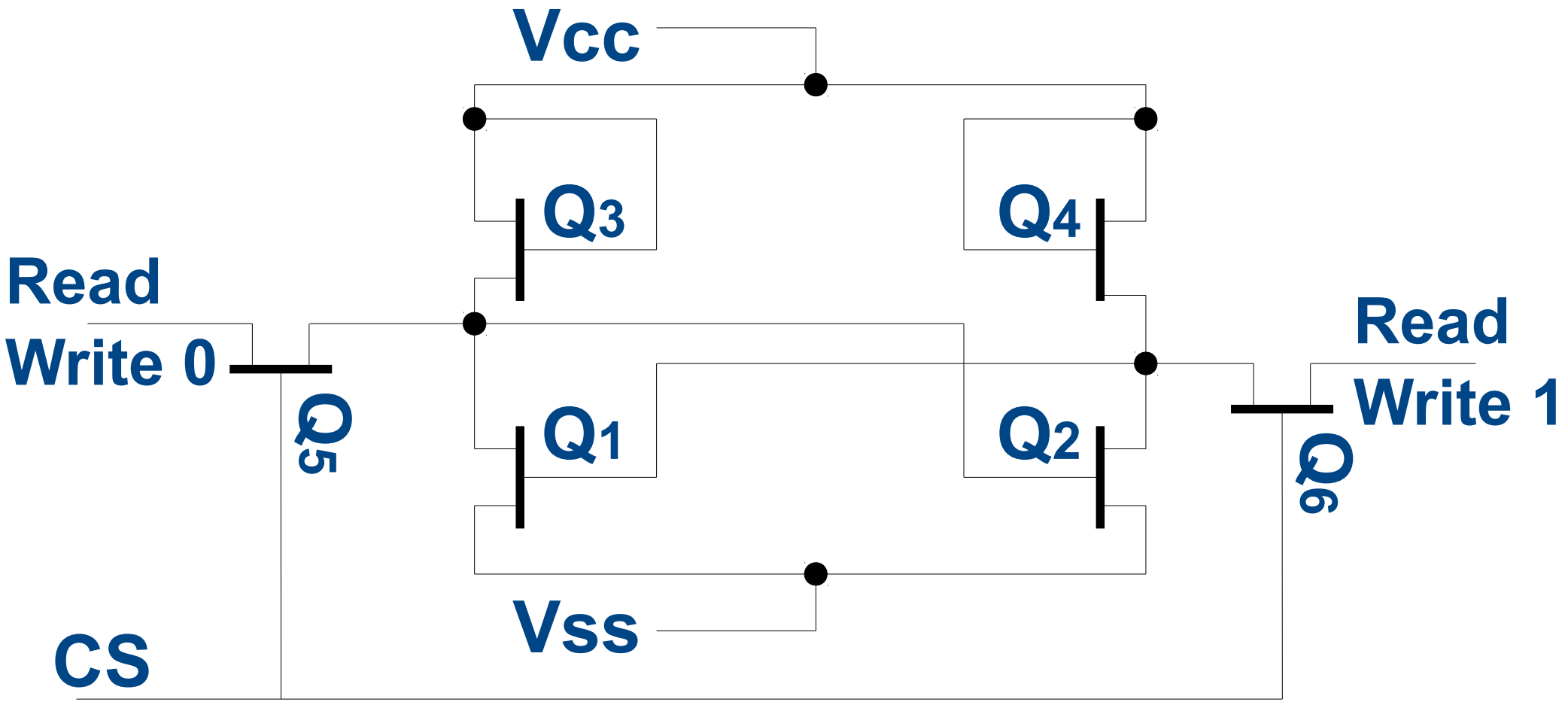
5.2.1 Bascule “RS”



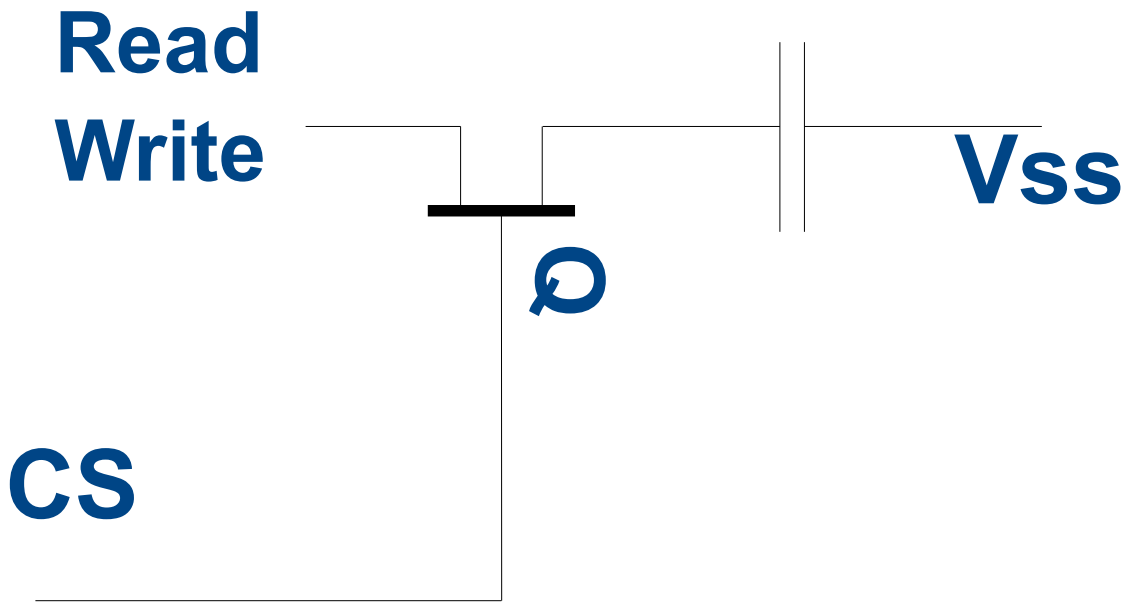
5.2.1 Bascule “RS”



5.2.1 Cellule statique



5.2.2 Cellule dynamique



5.2.2 Comportement

Le "0" or "1" est stocké dans une capacité parasite,
Qui a un courant de fuite.
→ rafraîchissement régulier
La lecture est destructrice,
→ obligation de ré-écriture
→ **Perte de bande passante**.

5.2.2 Comportement

Temps d'accès:
- sélection de la cellule,
- sa lecture ou son écriture.
Temps de cycle (~ 2 * $t_{accès}$)

temps pour l'accès,
+ temps pour la ré-écriture.

5.3 De la cellule à …

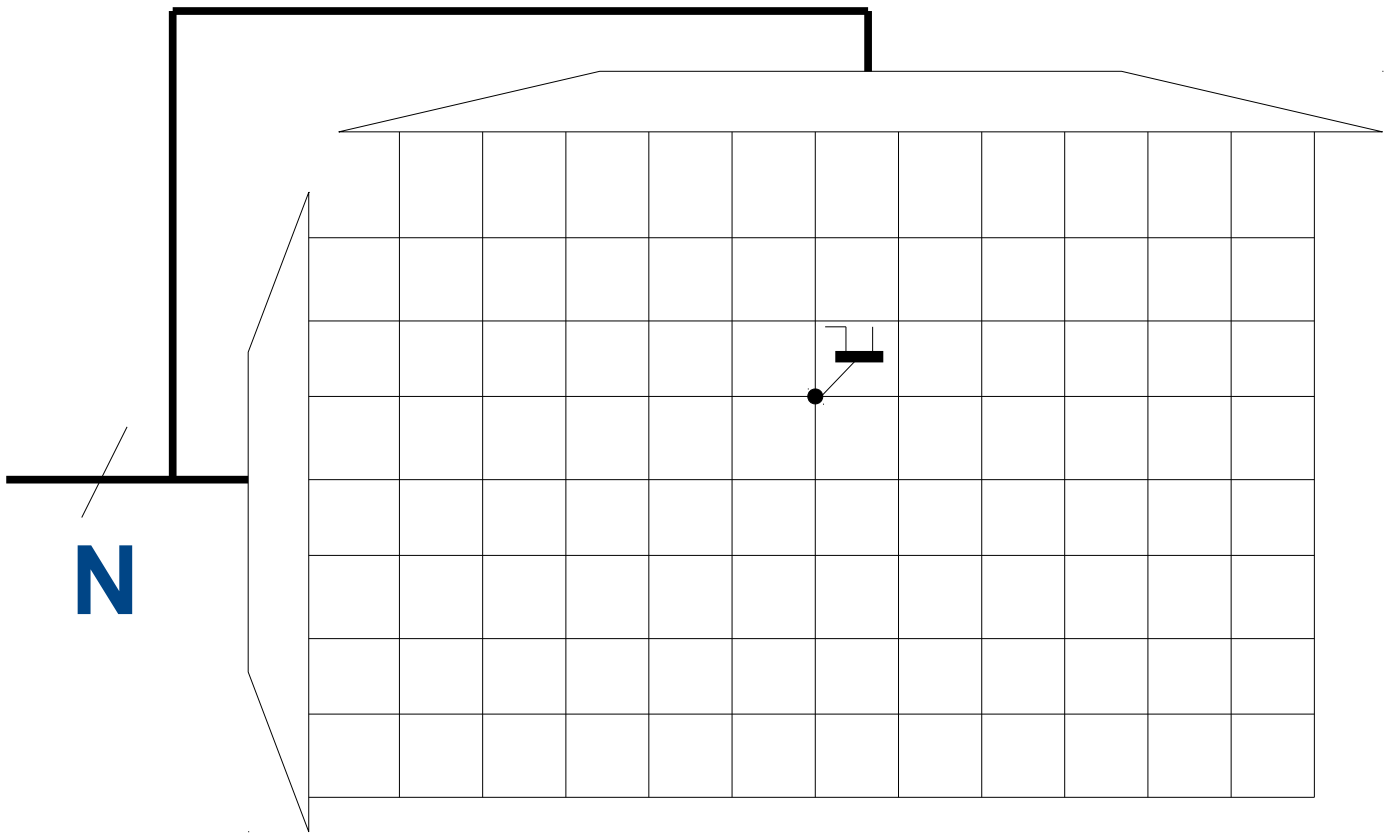
Dans un boîtier, on peut avoir:

- 1 mot de N cellules,
- N mots de 1 cellule.

Dans une "barrette",

- K mots de 8 bits,
- K/2 mots de 16 bits, …

5.3.1 Matrice de cellules



N lignes,
N colonnes
L'adresse est fournie en 2 temps

5.3.2 Adressage d'unités

Plusieurs unités (barrettes),
→ Répartition de l'espace d'adresses total ?
→ Comment attribuer les morceaux aux unités ?
Deux stratégies

5.3.2 Adressage d'unités

Exemple:

Un espace total de 256 K

$8 = 2^3$, $16 = 2^4$, $256 = 2^8$, $1K = 2^{10}$

8 unités (bancs) → 64K chacune

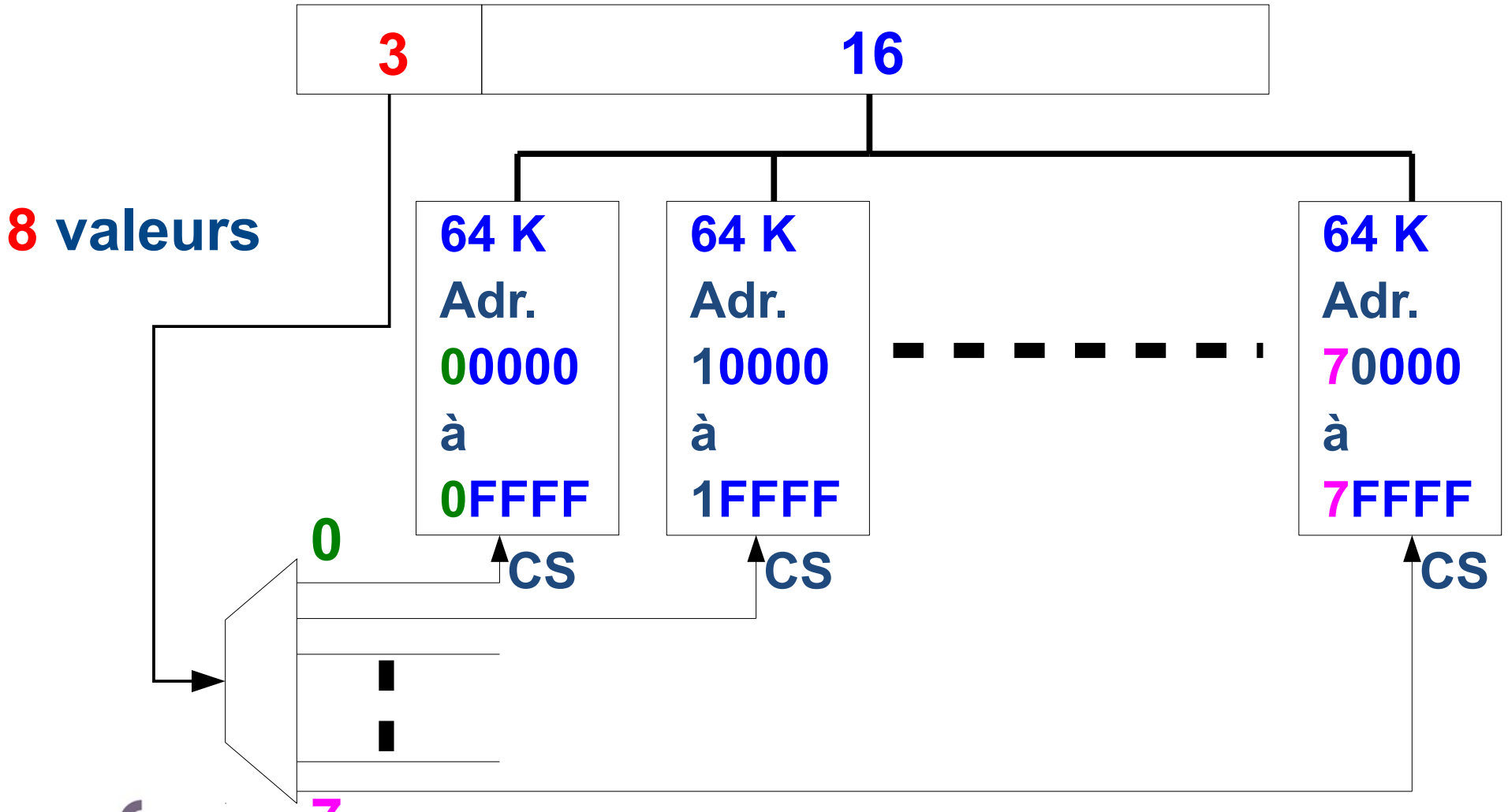
256 (2^8) = 8 (2^3) * 64 (2^5)

Valeurs irréalistes, mais simples.

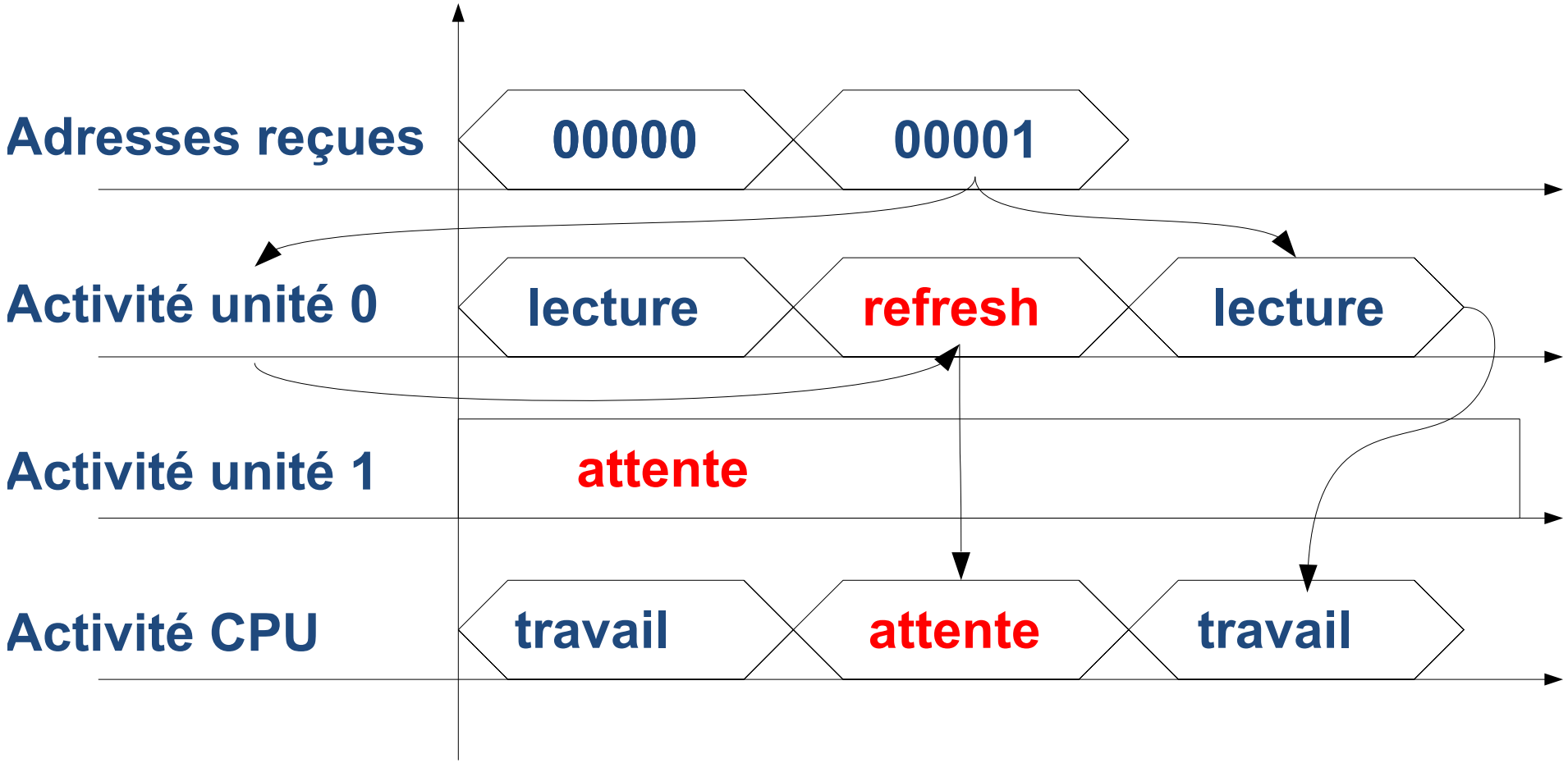
5.3.3 Adressage linéaire

Adresse fournie par le CPU:

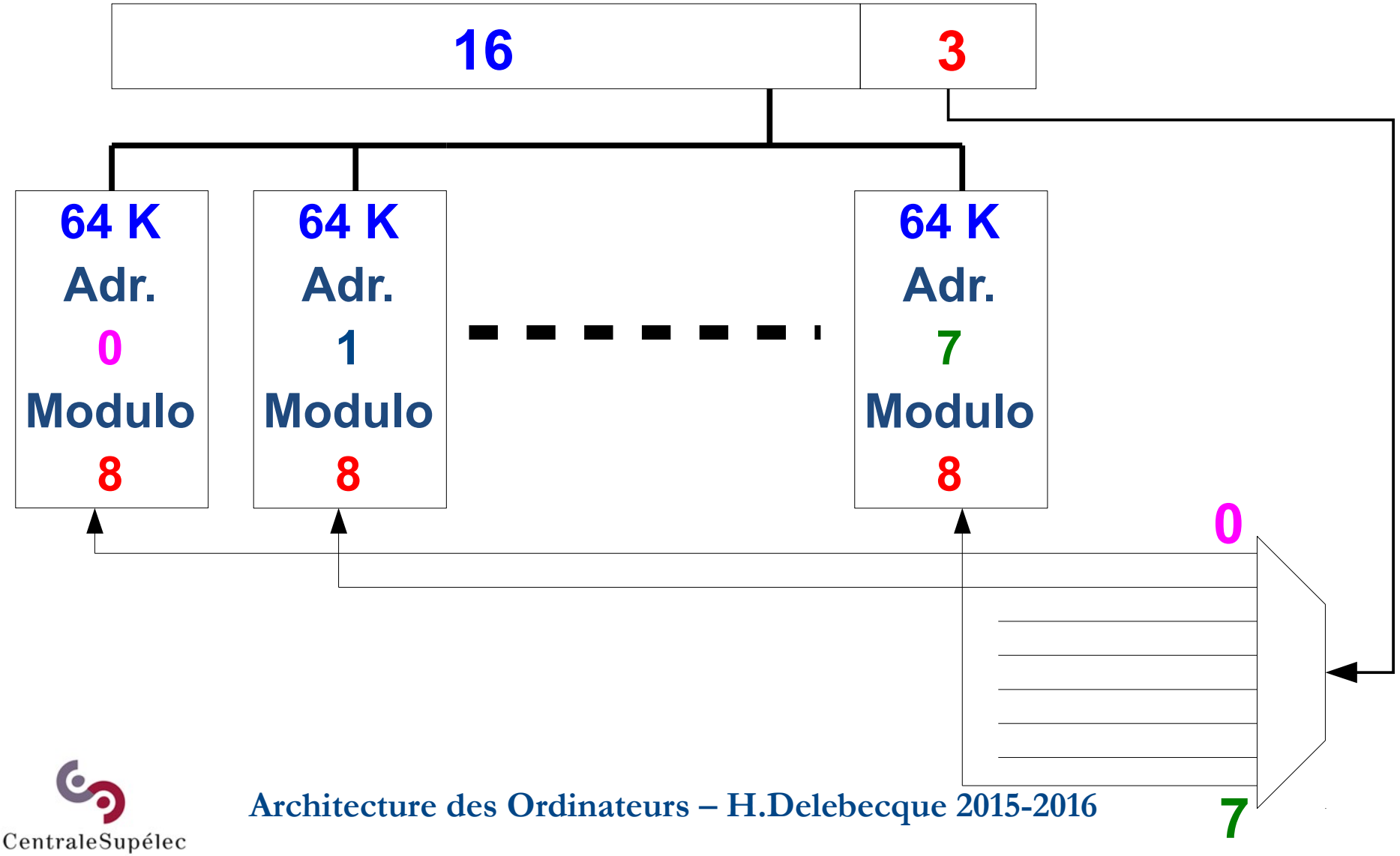
5.3.3 Adressage linéaire



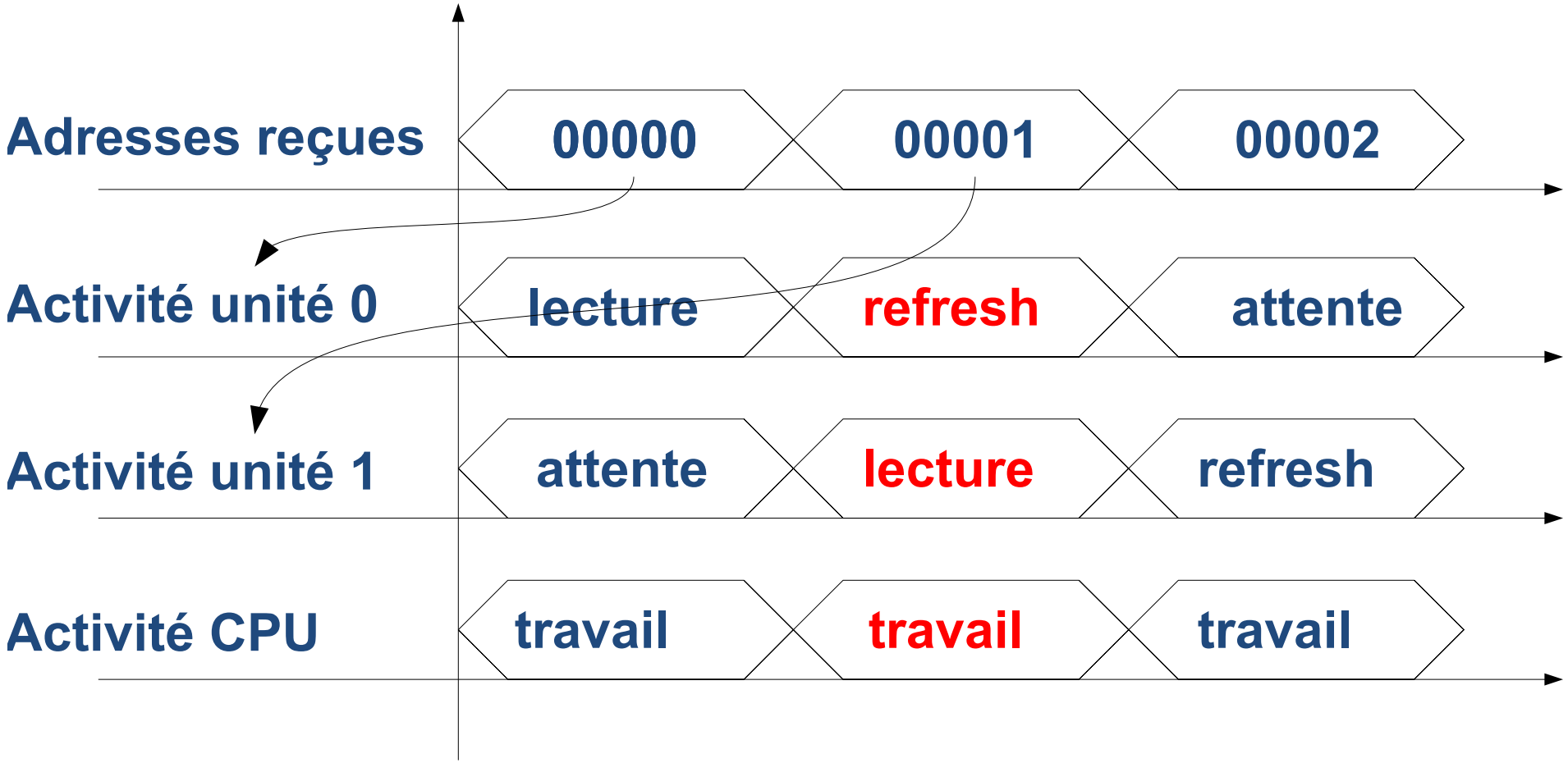
5.3.3 Adressage linéaire



5.3.4 Adressage entrelacé



5.3.4 Adressage entrelacé



5.4 Mémoires de masse

Pour conserver vos fichiers

Différentes technologies pour:

- Forte densité
- Faible coût à l'octet
- Lentes (relativement)

Accès par **nom du fichier**

5.4.1 Réalisations

Avec une **ROM**

→ petite taille, écriture difficile

RAM non volatile

→ petite taille, pas de versions

Mémoire “de masse” sans fichier

→ pas de version, pas de nom

5.4.2 Fichiers

Est-ce une nécessité ?

Oui, dès que vous avez besoin de

- Stockage “permanent”

Ou de

- “Partage” entre utilisateurs

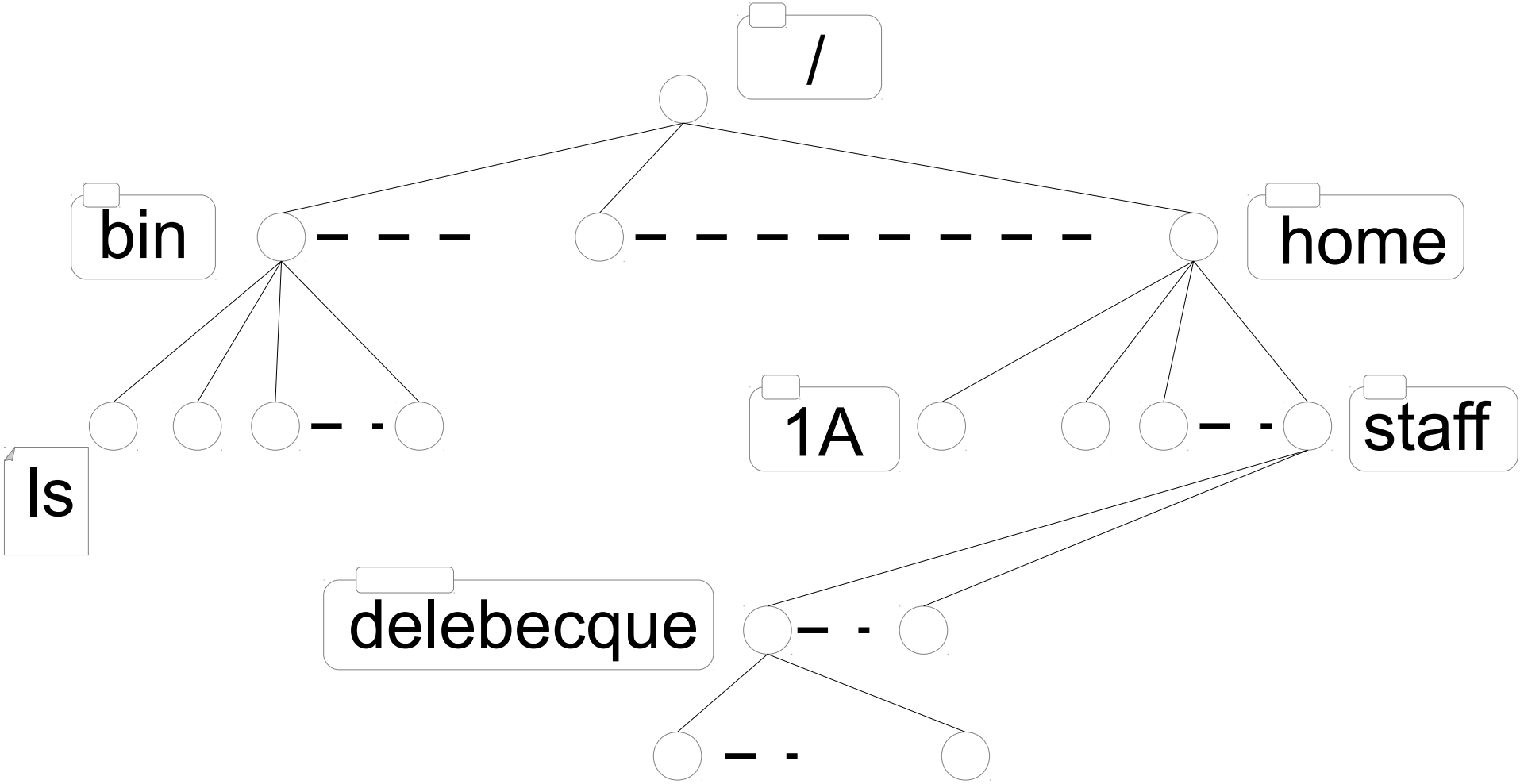
Ou d'un

- Gros volume d'information

5.4.2 Système de fichiers

→ hiérarchie de “répertoires”
→ un “répertoire racine”
→ qui contient toutes les autres
→ le tout formant un arbre
→ une définition universelle

5.4.2 Système de fichiers



5.5 Caches

- Principes
- Schéma général
- Lois sur les adresses
- Une organisation simple

5.5.1 Principes

Augmenter les performances :

Une mémoire centrale rapide ?

Grosse mémoire → Coût !

Idée :

Une grosse mémoire lente et

Une petite mémoire rapide

5.5.1 Principes

Tous les mots en mémoire lente,

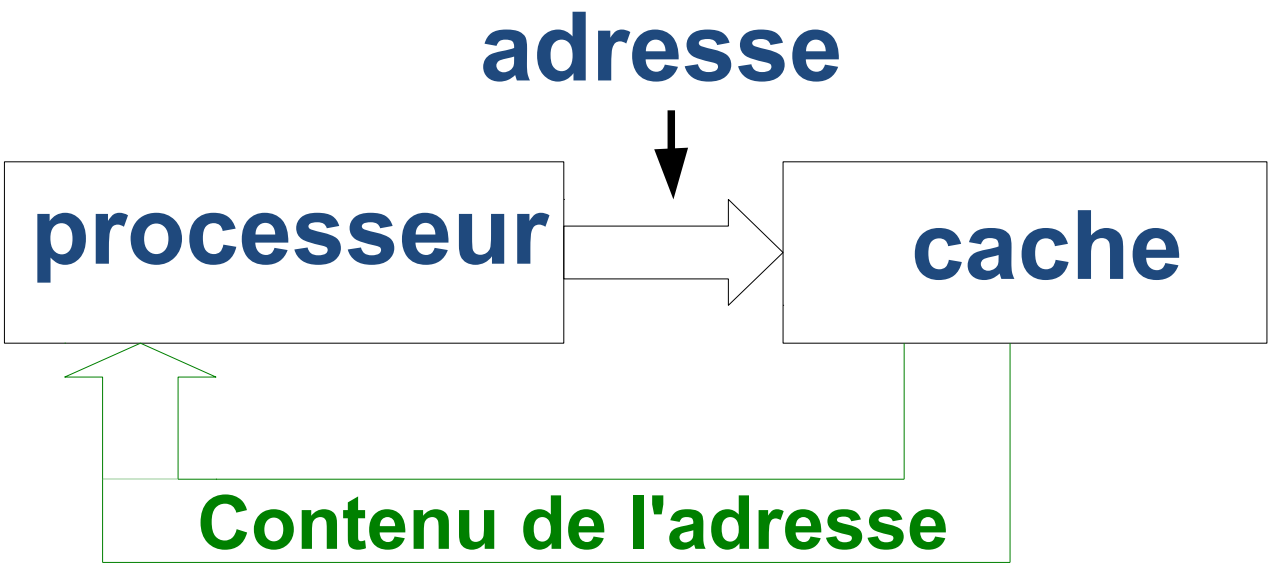
Juste les mots utiles en rapide.

→ Comment **prévoir** quels mots vont être utiles ?

→ Comment savoir s'ils sont là ?

→ Comment savoir où ils sont ?

5.5.2 Schéma général

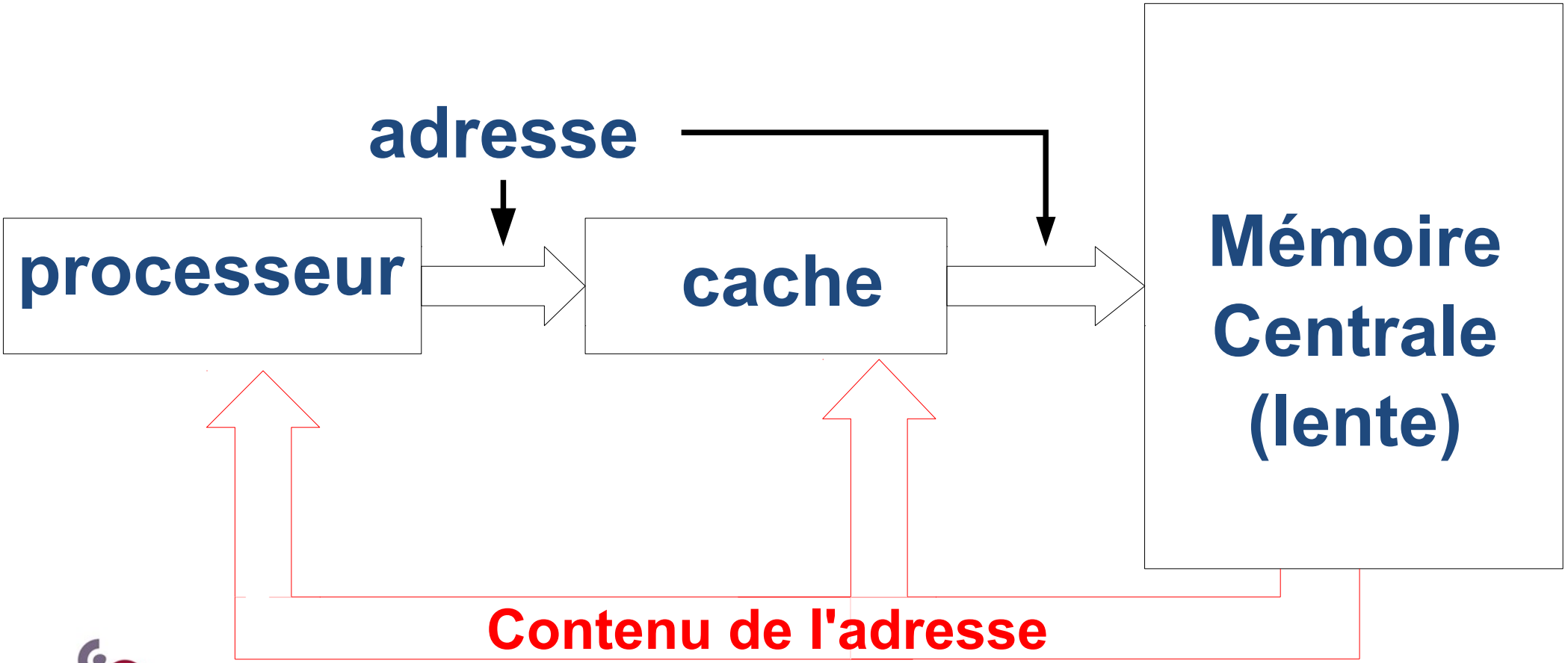


Mémoire Centrale (lente)

Si le cache contient l'adresse

5.5.2 Schéma général

Si le cache ne contient pas l'adresse



5.5.3 Lois des adresses

Un processeur ne produit pas les adresses aléatoirement

Quand il produit une adresse, il y a de fortes chances pour qu'il en demande une proche bientôt

Pourquoi ?

→ **Localité spatiale**

5.5.3 Lois des adresses

Quand il produit une adresse, il y a de fortes chances pour qu'il la redemande bientôt

Pourquoi ?

→ **Localité temporelle**

5.5.4 Organisation

Un cache est très petit.

Il ne contient pas tous les mots.

→ Quels mots stocker ?

→ Comment savoir si un mot est là ?

→ Comment le retrouver ?

→ Quel mot retirer s'il est plein ?

5.5.4 Organisation

L'analogie du **bibliothécaire**

Le client donne le n° du livre

Les livres sont sur des étagères,

Et le bibliothécaire est fatigué d'aller de son bureau aux étagères.

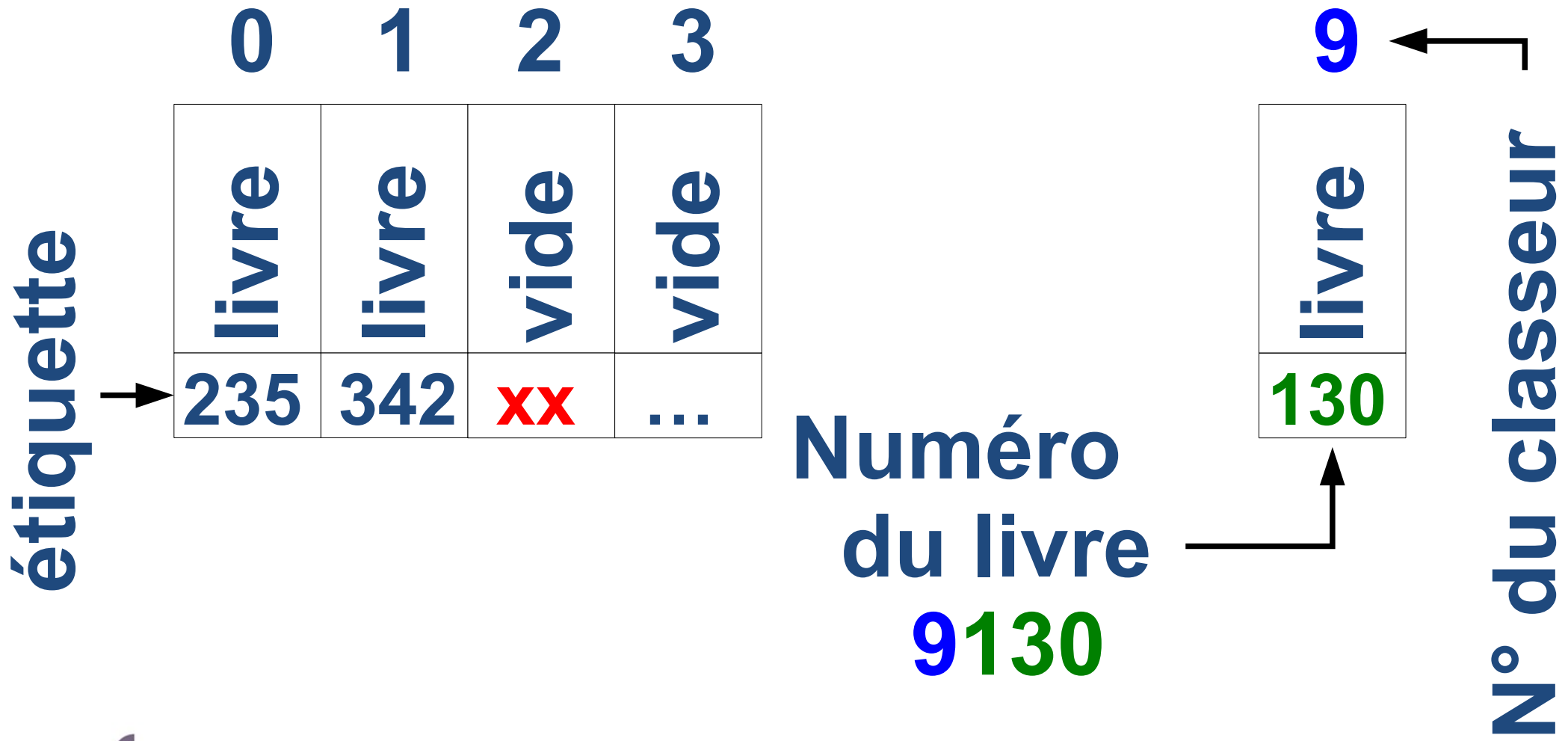
5.5.4 Organisation

Le bibliothécaire place 10 classeurs sur son bureau.

Un livre est placé dans le classeur selon le chiffre des milliers de son numéro.

Les autres chiffres sont reportés sur une étiquette du classeur.

5.5.4 Organisation



5.5.4 Organisation

Supposons:

Un cache de 128 blocs of 16 (2^4) mots (donc de taille 2K),

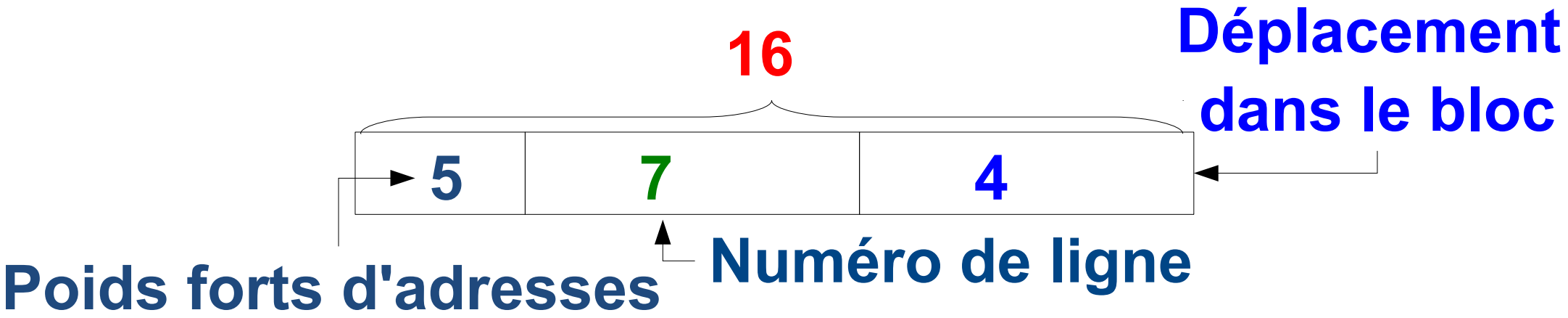
Une mémoire de 2^16 mots (donc de 64K)

Taille d'un mot quelconque,

Les valeurs sont simplistes.

5.5.4 Organisation simple

On décide que l'adresse de 16 bits est découpée en un n° de ligne rangée parmi 2^7 lignes dans le cache.



5.5.4 Organisation simple

Supposons :

Le cache initialement vide,

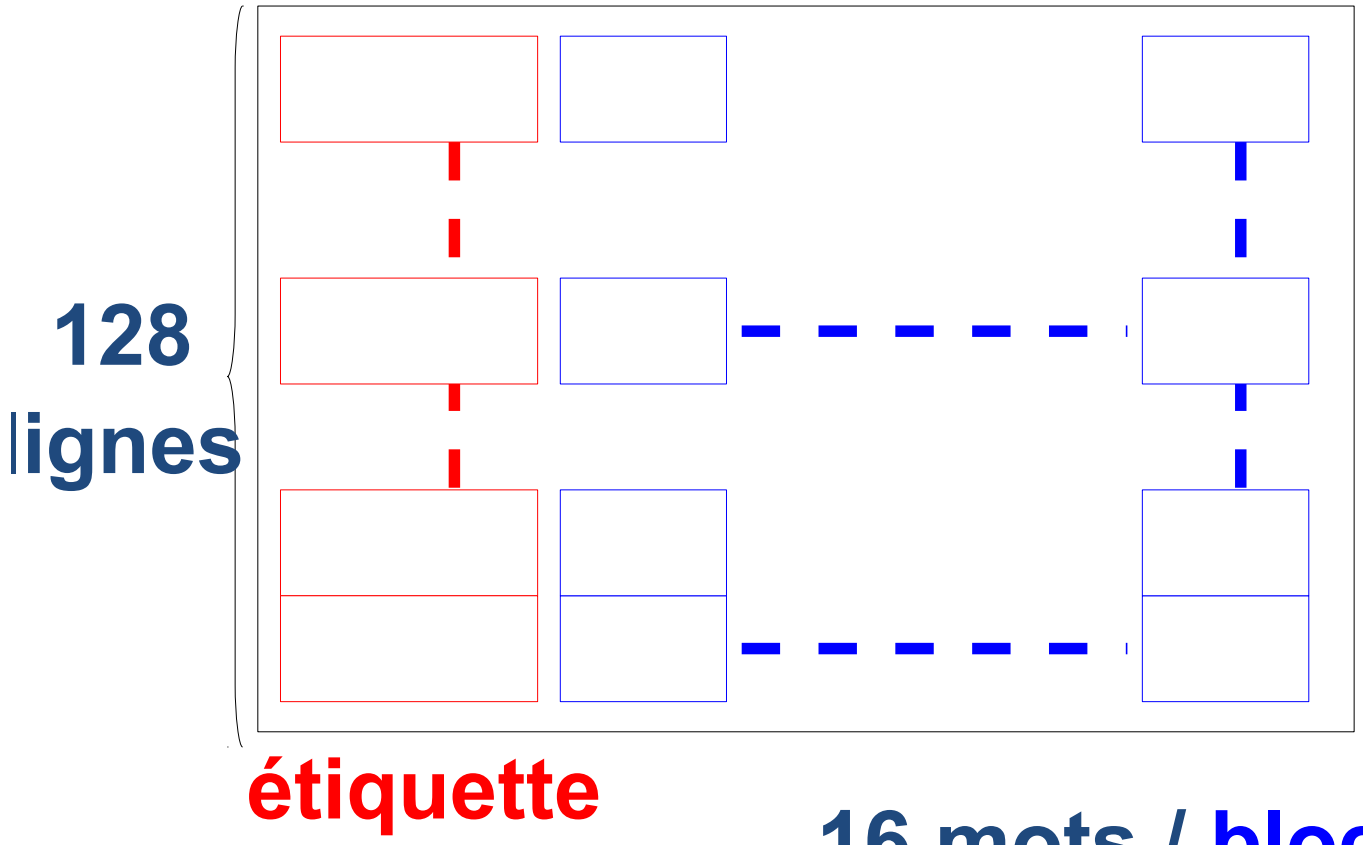
Et le processeur produisant les adresses de 0 à FFFF.

Comment le cache se remplit-il ?

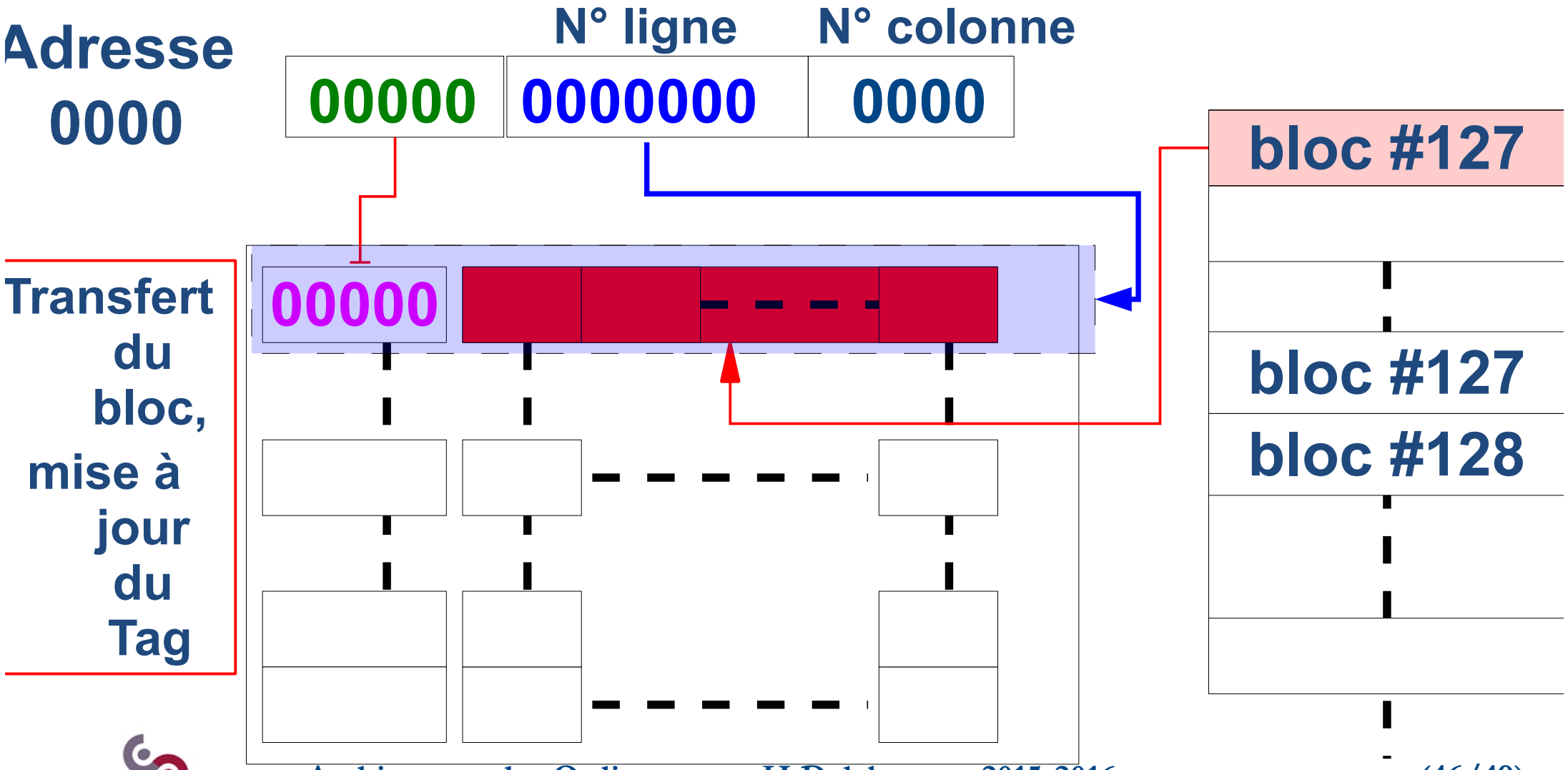
5.5.4 Organisation simple

Contenu du cache

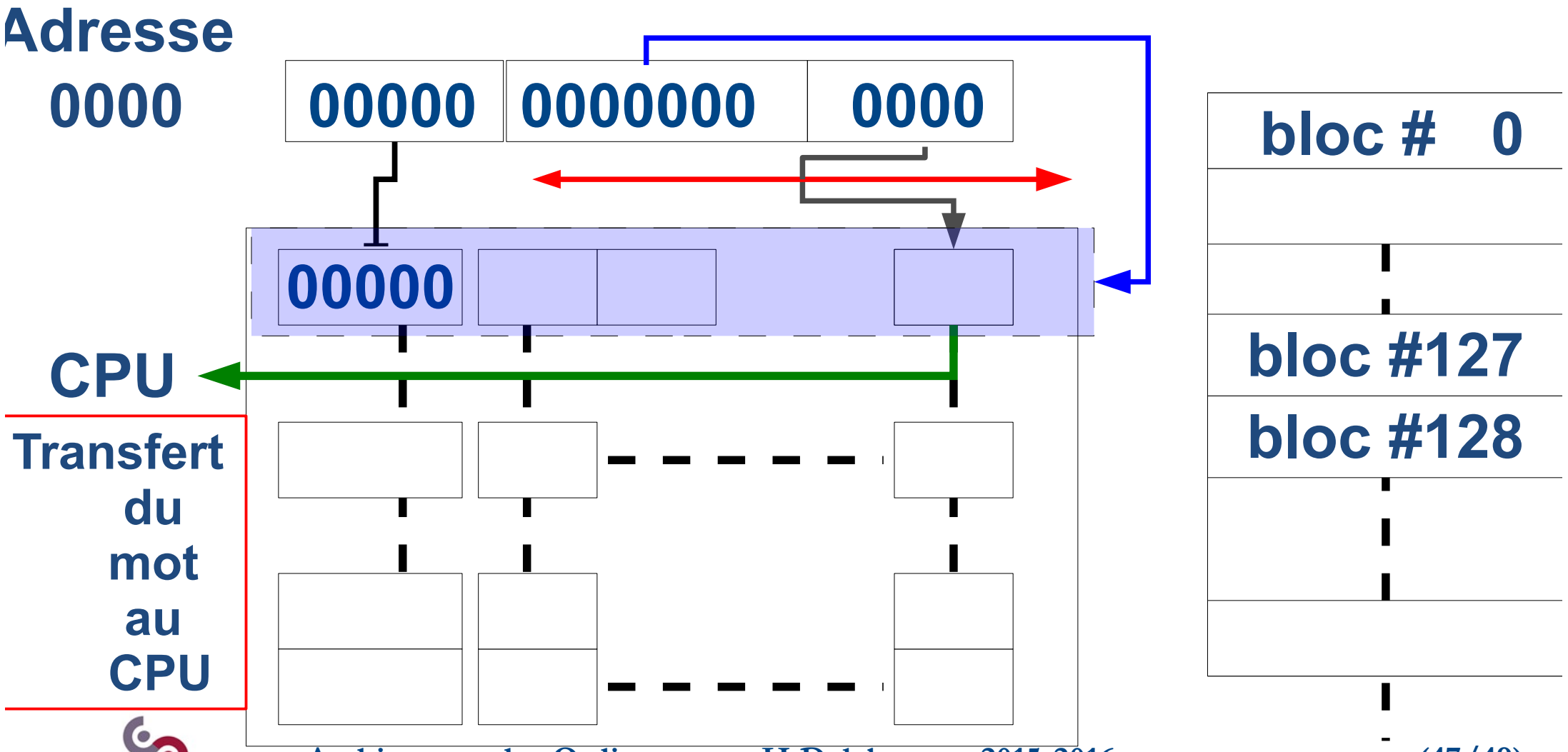
mémoire



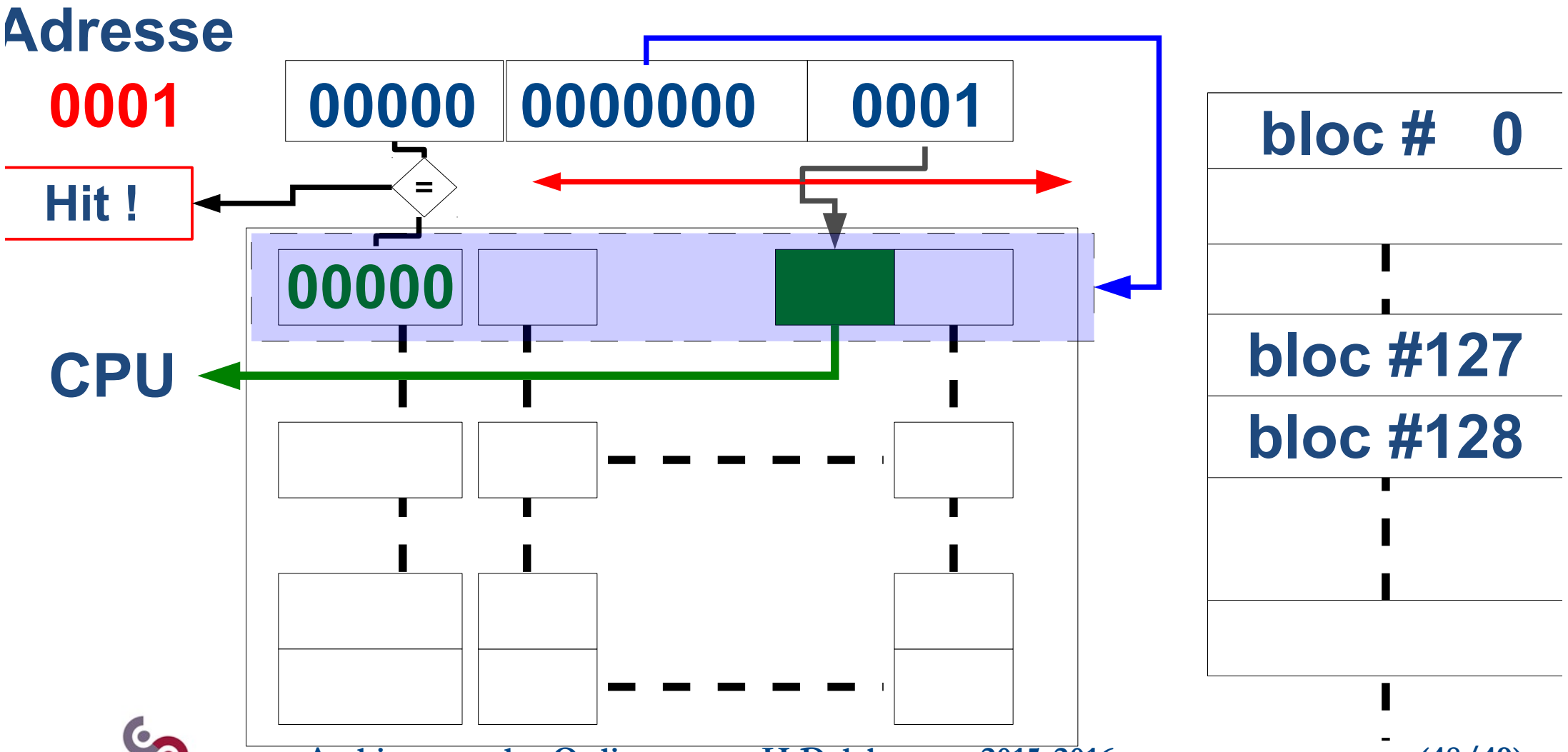
5.5.4 Organisation simple



5.5.4 Organisation simple



5.5.4 Organisation simple



5.6 Valeurs usuelles

Type	Technologie	Taccès typ.	Taille typ.	Coût/octet
Registre	dynamique	< ns	10 to 256 W	?
Cache	statique	1 ns	qq M	1.10^{-5} €
Centrale	dynamique	10-100 ns	qq G	6.10^{-8}€
Disque dur	magnétique	1-10 ms	qq T	2.10^{-10}€
DVD,Blu-Ray	optique	10 ms	< 30G	6.10^{-10}€

6. Entrées-Sorties

6.1 Organisation générale
6.2 Lien processeur-coupleur
6.3 Interruptions
6.4 Communication entre mémoire et coupleur
6.5 Liaison coupleur-périphérique

6.1 Organisation générale

Objectif:
Permettre la communication avec l'extérieur via des périphériques:

- **d'IHM (souris, écran, ...)**
- **de stockage (disques, …)**
- **ad-hoc (capteurs, moteurs, ...)**

6.1 Organisation générale

Entrée-sortie logique:

- demandée par l'utilisateur,
- indépendante du matériel,
- bufferisée ou pas.

Entrée-sortie physique:

- réellement réalisée (ou pas),
- pour plusieurs E/S logiques ?

6.1 Organisation générale

Un coupleur :

- **Entre le CPU et le périphérique**

Pour :

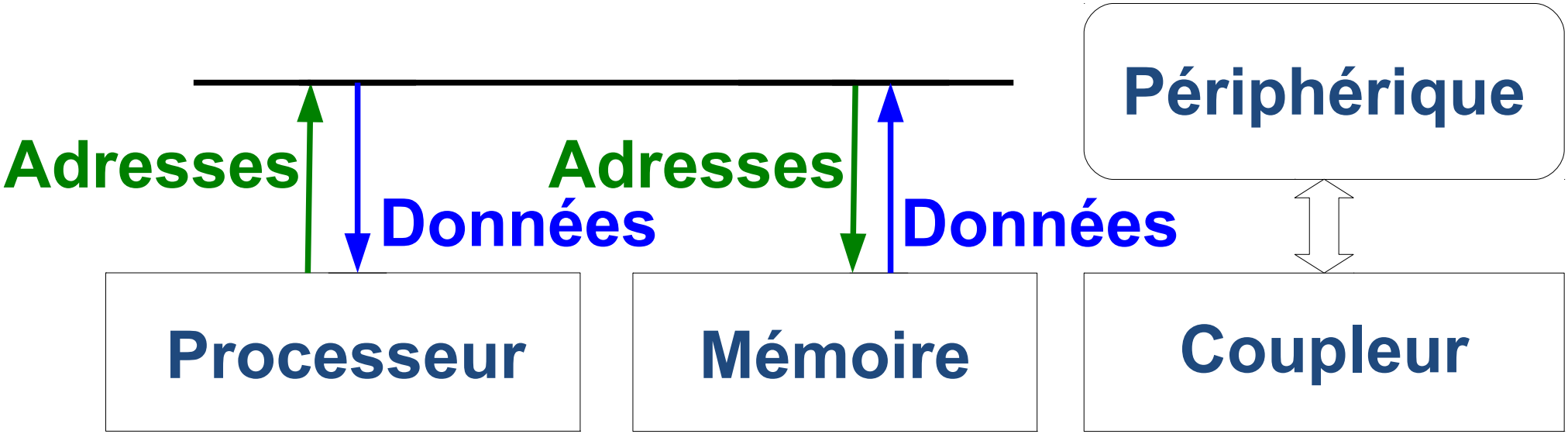
- **Le partage entre périphériques**
- **Faire les adaptations**
- **S'appuyant sur des normes**

6.2 Lien CPU-Contrôleur

Plusieurs architectures:

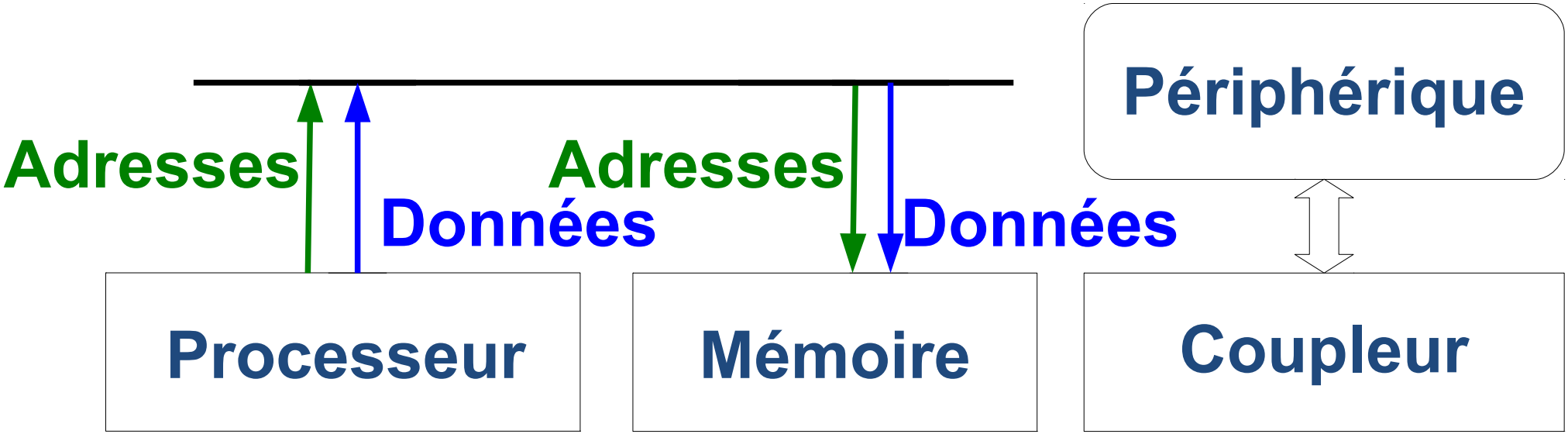
- un bus d'E/S dédié pour les ordinateurs chargés en E/S,
- avec un processeur d'E/S pour les gérer, au besoin
- un bus unique sinon

6.2.1 Bus d'E/S dédié



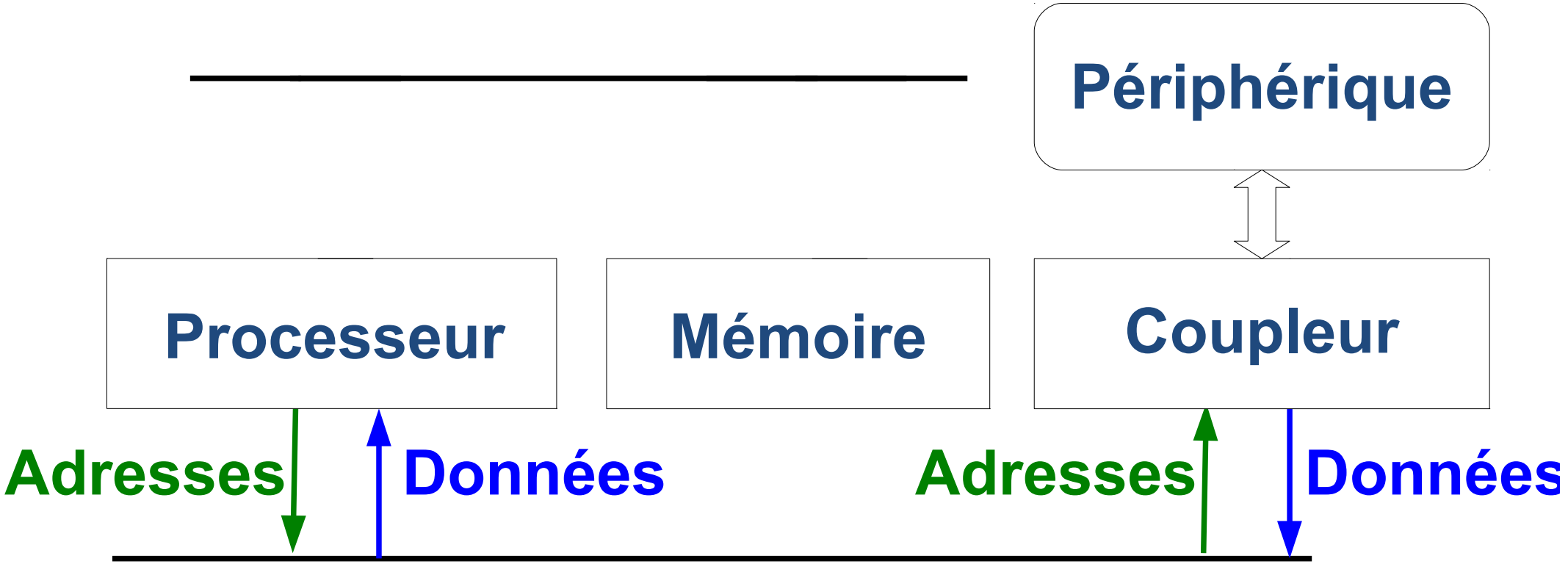
Le processeur lit en mémoire

6.2.1 Bus d'E/S dédié



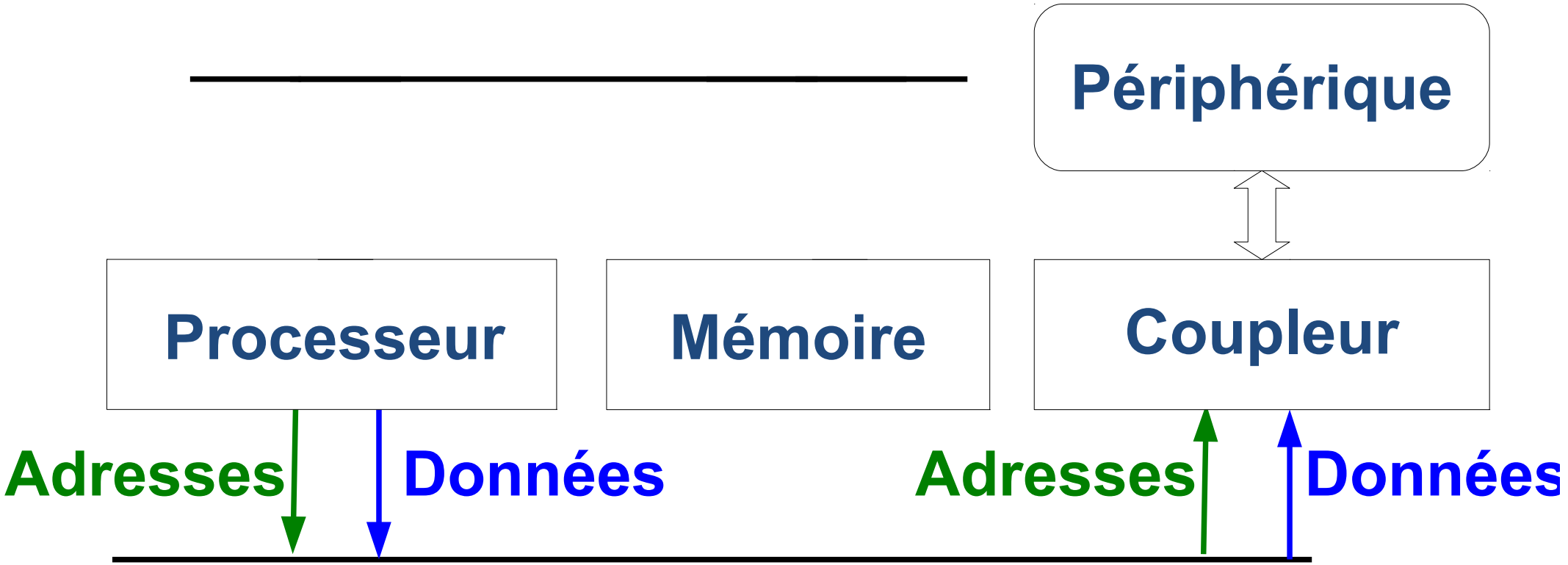
Le processeur écrit en mémoire

6.2.1 Bus d'E/S dédié



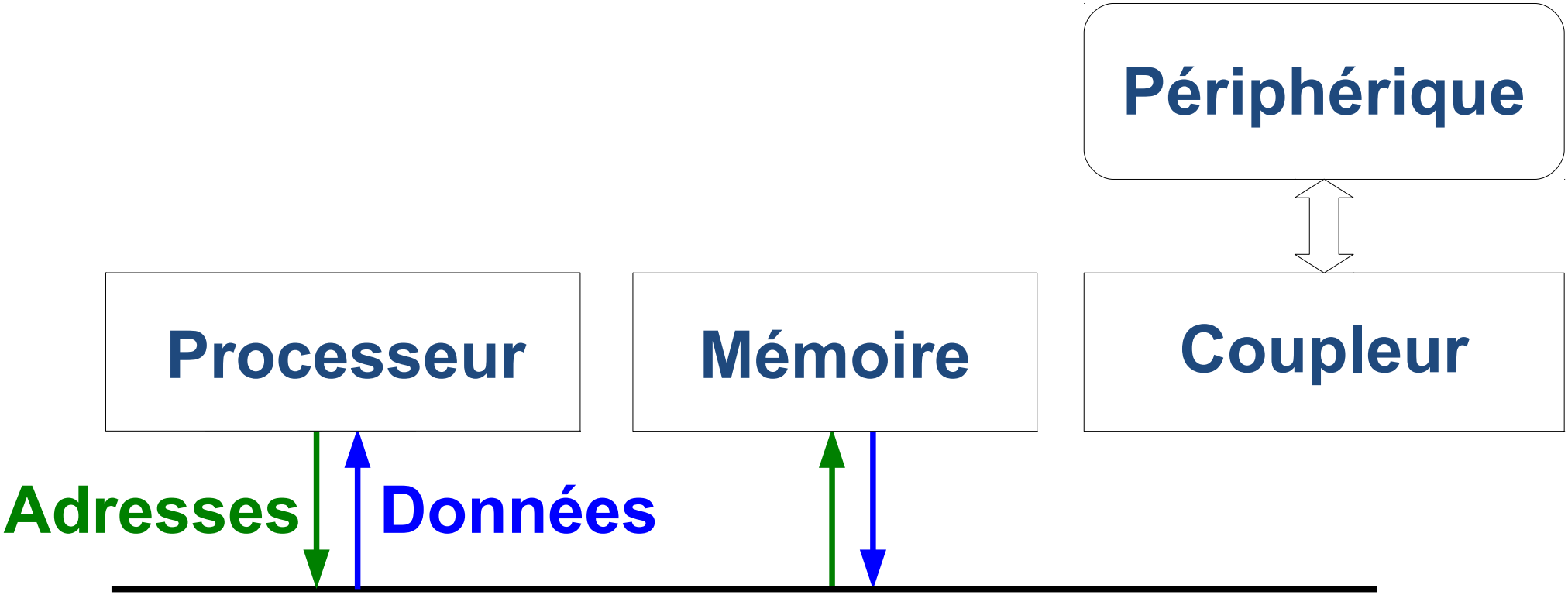
Le processeur lit dans le coupleur

6.2.1 Bus d'E/S dédié



Le processeur écrit dans le coupleur

6.2.2 Bus commun



Le processeur lit en mémoire

6.3 Interruptions

Analogie avec le monde réel:

Vous attendez quelqu'un, tout en regardant un match ou un film.

Problème:

Vous ne voulez manquer ni l'un, ni l'autre.

6.3 Interruptions

Scrutation:
Régulièrement, vous mettez la TV en pause, allez ouvrir la porte, puis relancez à la TV.

Problèmes :

- **la période “d'échantillonage”**
- **la perte de temps**

6.3 Interruptions

1ère amélioration:
Vous installez une sonnette
Quand elle sonne:
Vous mettez sur “Pause”,
Vous discutez avec le visiteur,
Vous relancez la TV

6.3 Interruptions

Vous avez (aussi) un mobile

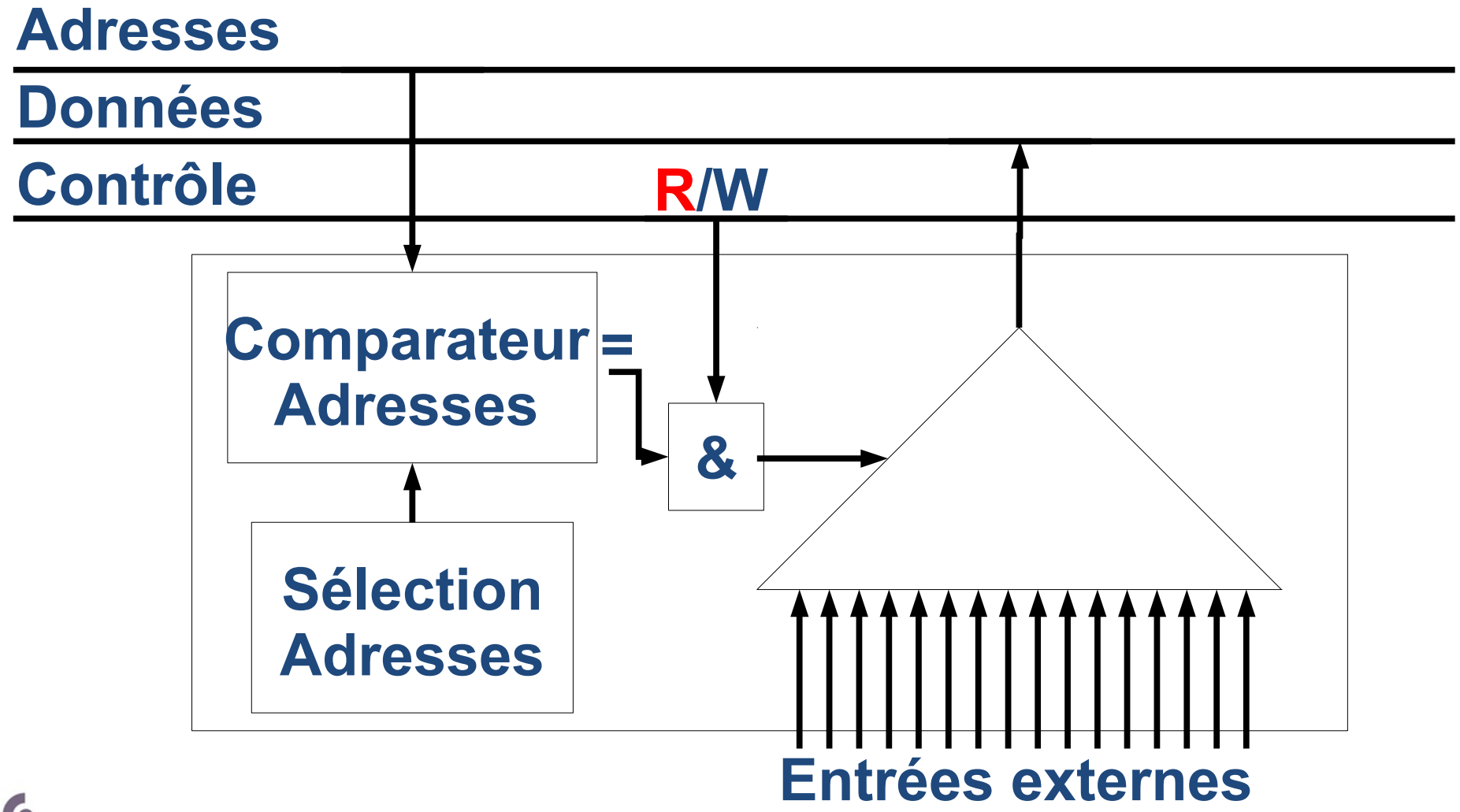
2ème amélioration:

S'il sonne pendant que vous parlez au visiteur,

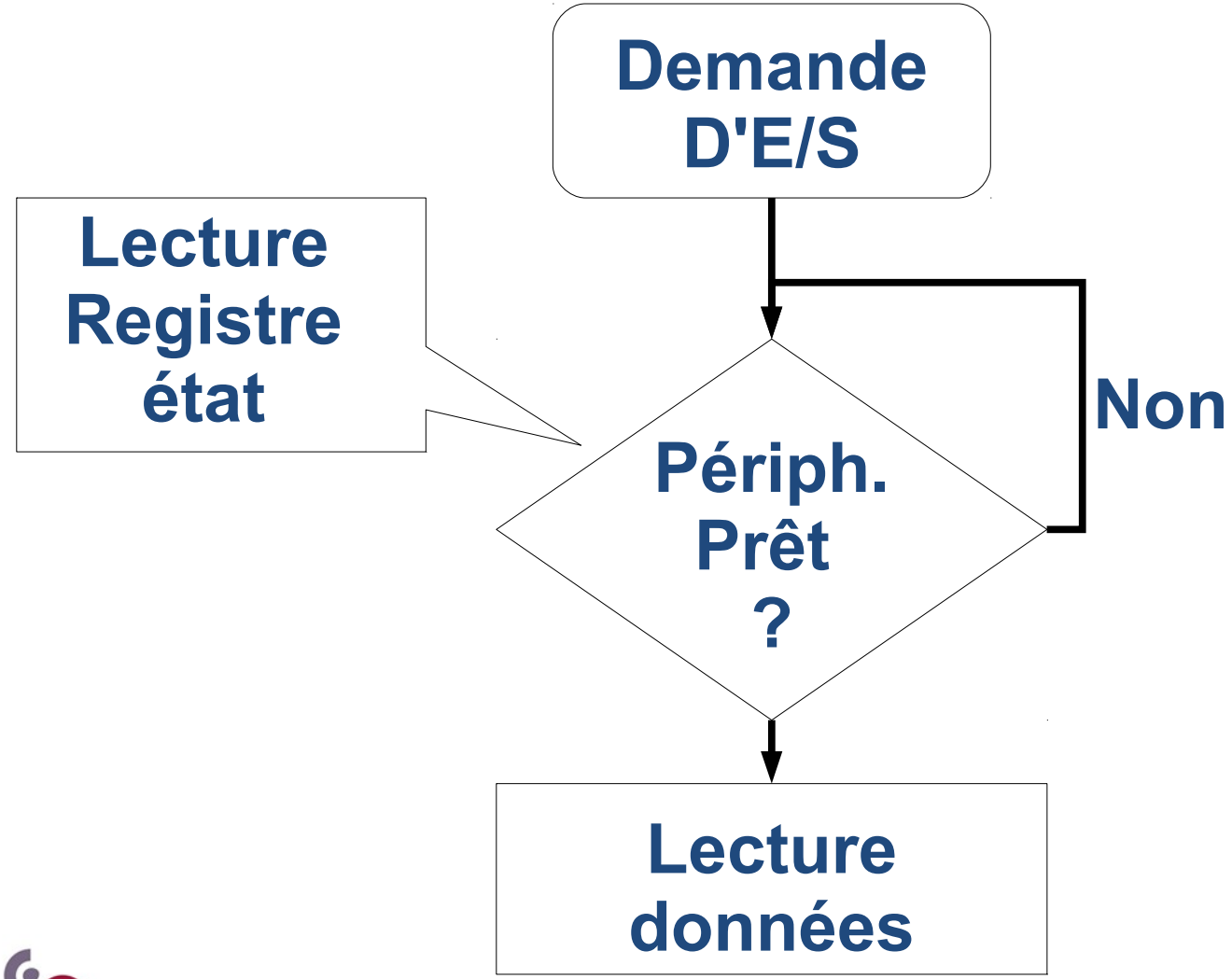
Vous prenez l'appel,

Sans être trop long !

6.3.1 Coupleur d'entrée



6.3.2 Scrutation



6.3.3 Généralités

Elles sont asynchrones

→ Le programme interrompu ne sait pas quand elles arrivent

→ il ne peut sauvegarder son "état" avant une interruption

→ Le S.P de traitement doit laisser le processeur intact

6.3.3 Généralités

Matériel nécessaire :

- un signal **IRQ** vers le CPU,
- un signal d'acquittement vers le coupleur,
- Une capacité **minimale** de sauvegarde de l'état CPU

6.3.4 Gestion

Partiellement par le CPU

- Les registres **vitaux** sont sauvegardés en pile
- les interruptions de même niveau **masquées**
- le SP de traitement est **appelé**

6.3.4.3 Appel du SP

Qui impose l'adresse du SP ?

→ pas le programme interrompu

→ pas le processeur, qui doit pourtant la trouver

→ un mot d'adresse connu

Le “vecteur” d'interruption

6.4 Transfert de Données

Analogie: Envoi d'une lettre.

Vous prenez un caractère,

Vous allez le porter à destination,

Vous revenez chez vous pour le suivant

Et ainsi de suite…

6.4.1 E/S par programme

Le processeur assure le transfert

Pour chaque mot,

- il le lit (mémoire ou coupleur)
- il l'écrit (inverse)
- il incrémente l'adresse (mém.)
- il décrémente un compteur

6.4.1 E/S par programme

Inconvénients:

Le temps processeur gâché,

Une IT par E/S élémentaire pour chaque coupleur.

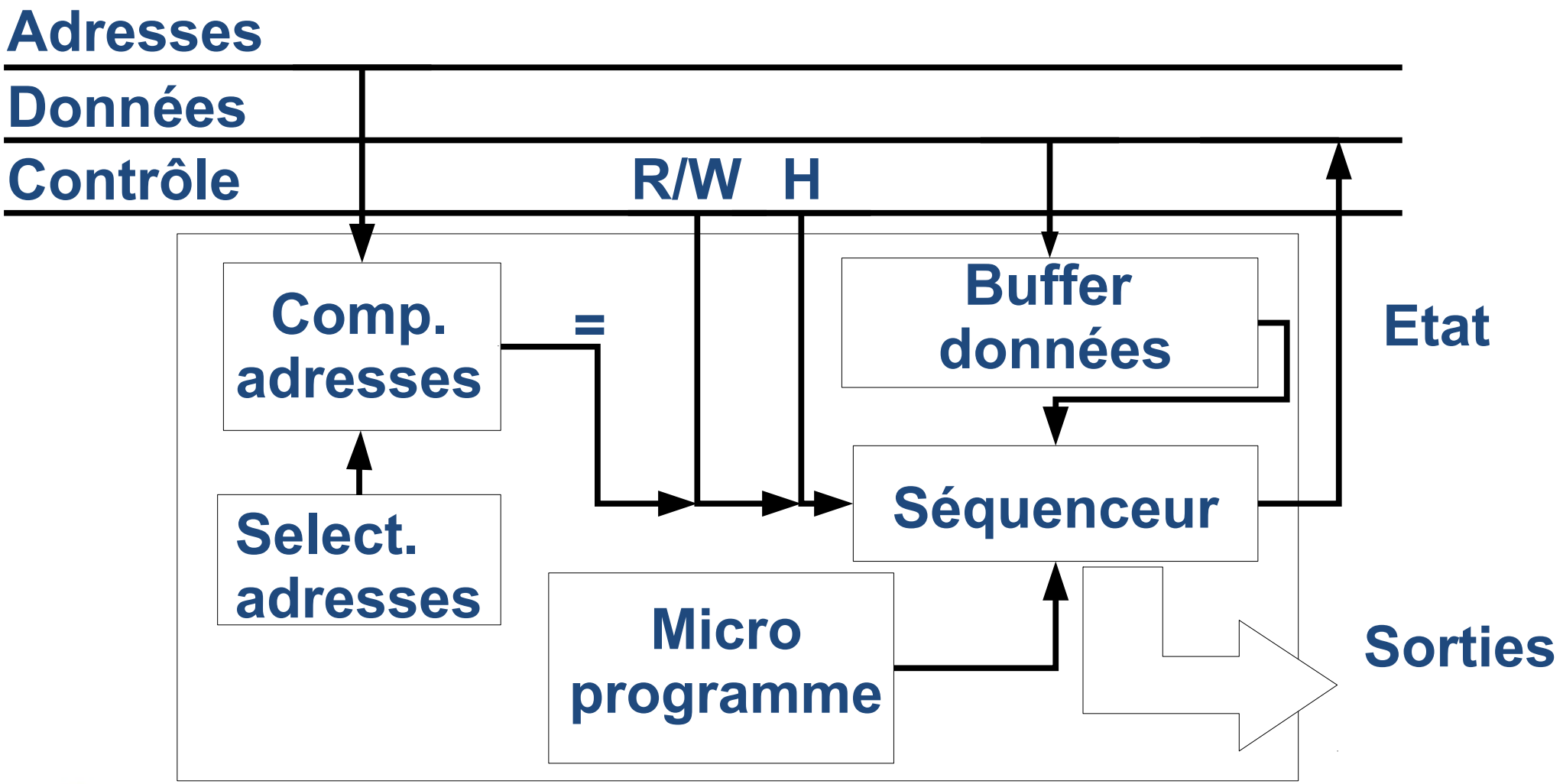
Chaque E/S est spécifique

Avantages: simple et pas cher

6.4.2 Groupe d'E/S

- Le processeur envoie un “buffer”
- Le coupleur le traite tout seul
- Il émet une IT par buffer
- Toutes les E/S sont gérées de la même manière (via un **driver**).
- La charge CPU de transfert reste la même

6.4.2 Groupe d'E/S



6.4.3 Rôles du driver

Initialisation de l'E/S globale:

- le CPU envoie les données
- Il autorise les IRQ du coupleur

Gestion des interruptions:

- traite la fin de l'E/S élémentaire,
- s'il y en a une autre, il la lance

6.4.3 Rôles du driver

- S'il n'y a plus d'E/S élémentaire, interdit les IRQ du coupleur.

→ fin de l'E/S globale

Selon le périphérique, d'autres points d'entrées existent:

- test, reset, formattage,...

6.4.4 D.M.A

On décharge le CPU du transfert des données.

Le coupleur lit ou écrit tout seul

→ Le CPU lui fournit l'adresse de départ (à l'initialisation)

→ Il émet les adresses sur le bus du "CPU"

6.4.4 Architecture

Le DMA doit pouvoir:

→ émettre l'adresse d'un mot

→ envoyer ou recevoir celui-ci

→ décompter les mots

→ enchaîner seul ces opérations

Il consomme des accès mémoire

6.5 Lien coupleur-périph.

Dépend :

- du type de périphérique,
- de la distance entre unité centrale et périphérique

Standardisée ou pas

6.5.1 Régulation de flux

Une paire “producteur-consommateur”
Débits moyens supposés égaux
Mais des “pointes” ponctuelles
→ dispositif “amortisseur”
→ arrêt/relance du plus rapide

6.5.2 Types de liaisons

Parallèle

→ transfert par octet

→ liaison rapide, faible distance

Série

→ transfert par bit (groupés)

→ liaison simple, bon marché

7. De Python au CPU

7.1 Bases
7.2 Interprète, compilateur
7.3 Exemples de traduction
7.4 Implantation en mémoire

7.1 Bases

Expression / programme

Syntaxe / sémantique

Langue naturelle / langage de programmation

Traduction / évaluation

Langage typé / non-typé

7.2 Interprète-Compilateur

Soit un programme à exécuter

Interprété :

Traduit, évalué, la valeur affichée

Compilé:

Traduit (et stocké dans un fichier)

Puis executé, la valeur affichée

7.2.1 Traduction

→ Analyse les caractères

→ Isole les instructions et “mots”

→ Vérifie la syntaxe

→ Retrouve les variables

→ Repère les “mot-clés”

→ Traduit les instructions

Assembleur / Compilateur

7.2.2 Evaluation

→ D'une forme “interne”

→ Qui référence des variables

→ Et des fonctions traduites

→ avec des arguments

→ qui seront évalués

→ Retourne une valeur

7.2.3 Exécution

→ D'un fichier “exécutable”
→ Résultat d'une compilation
→ Stocké sur un disque
→ **Chargé** en mémoire
→ Avec d'éventuels arguments
→ Effectuant des E/S

7.2.4 Impression

→ Dans le cas interprété

→ Inverse d'une traduction

→ D'une forme interne, vers une chaîne de caractères

→ En plus des éventuelles E/S

7.3 Quelques exemples

7.3.1 Gestion des variables
7.3.2 Affectation simple
7.3.3 Affectation par expression
7.3.4 Condition
7.3.5 Boucle

7.3.1 Variable

Déclaration **implicite** en Python,

→ une table des variables

→ clé d'accès: le nom

→ données: (adresse, type)

→ évaluation = lecture mémoire

→ affectation = écriture mémoire

7.3.2 Affectation simple

Ex: *var = 10*

→ évaluation de la partie droite

→ constante (pré-évaluée)

→ variable (lecture mémoire)

→ stockage en mémoire

À l'adresse associée à la variable

7.3.3 Affectation complexe

Ex: *var = facto(3)*

→ évaluation de la partie droite

→ évaluation des arguments,

→ “liaison” des valeurs

→ “application” de la fonction

→ “retour” de la valeur

→ stockage en mémoire

7.3.4 Condition

```
if (var == 0) :
    resultat = 1
else :
    resultat = var * facto(var – 1)
```

→ évaluation de la condition

→ calcul de la “bonne” valeur

→ retour de celle-ci

7.3.5 Boucle

```
while (var > 0) :
    resultat = resultat * var
    var = var - 1
```

→ si la condition est vraie

→ évaluation du corps

→ saut en début de boucle

→ sinon, saut après le corps

7.4 Implantation mémoire

Variables: statiques

Listes, Objets:

→ créés dynamiquement

Pile:

→ comportement dynamique

→ adresse de début statique

7.4.1 Listes et Objets

Demandés par le programmeur

→ Durées de vie variables

→ Tailles variables

→ Nombre très variable

Alloués dans une zone spéciale

Le “heap” ou “tas”

7.4.2 Pile

Réservée par le système

→ une zone mémoire,

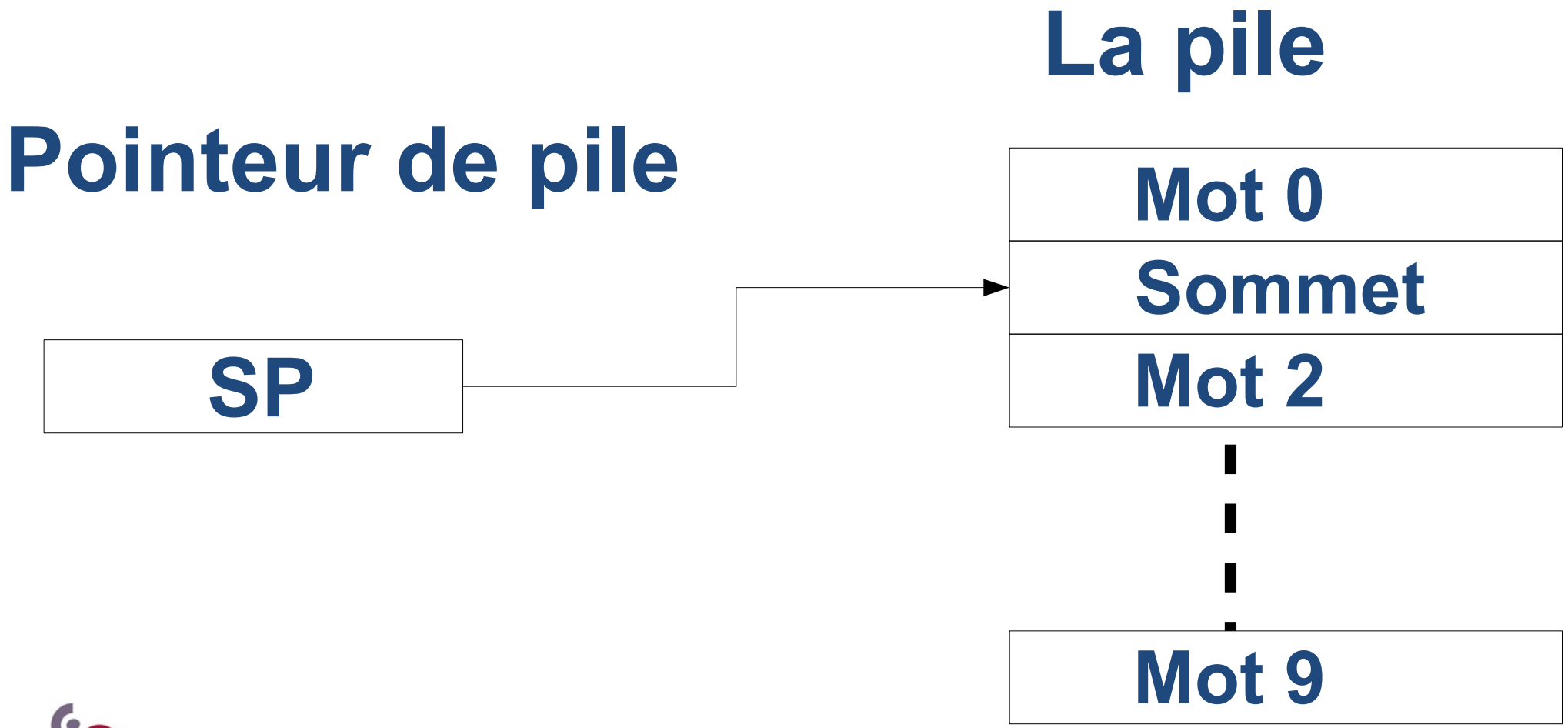
→ un sommet de pile

→ accès FIFO

Indispensable pour la récursivité

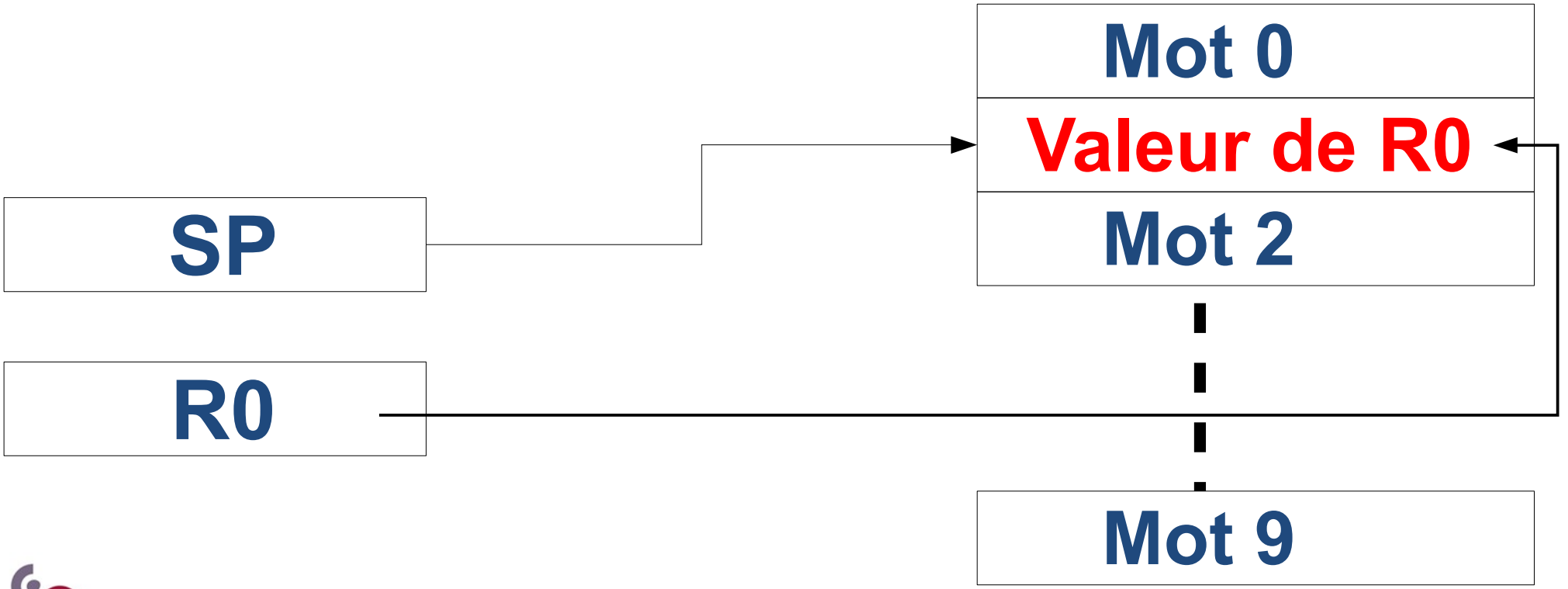
Très pratique pour les fonctions

7.4.2.1 Définition



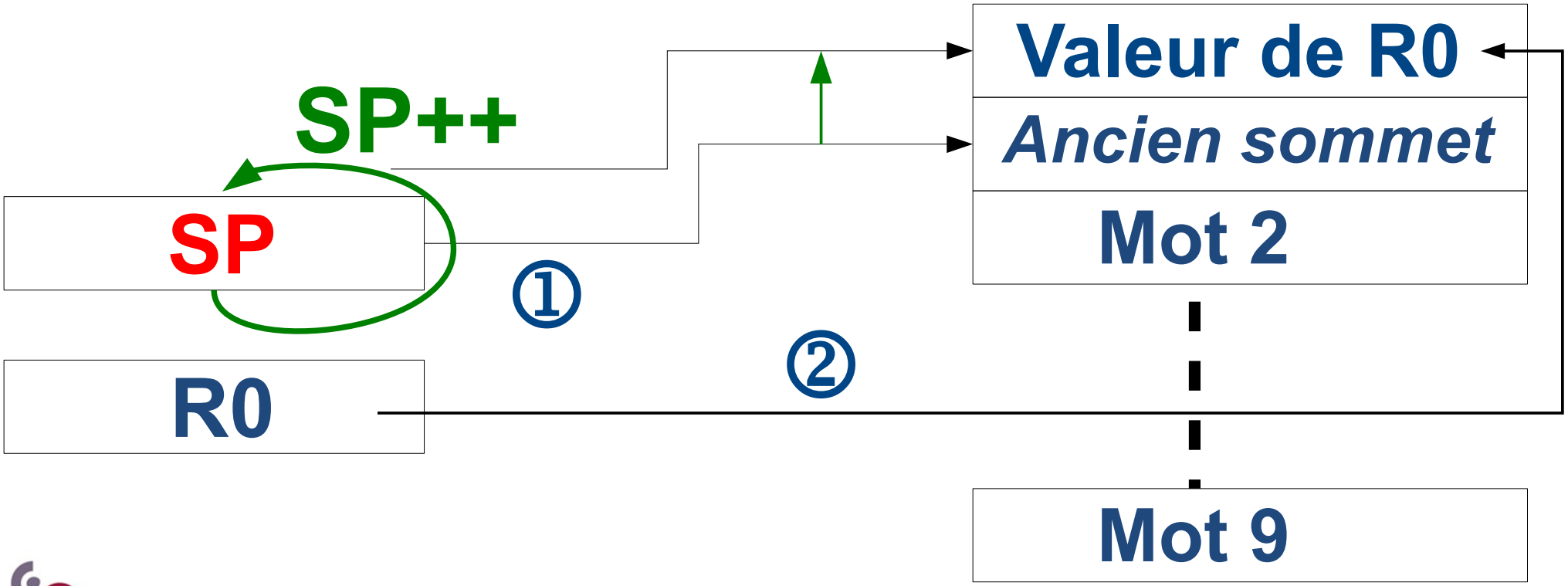
7.4.2.2 Empilage (faux)

PUSH R0 ↔ STR R0, SP ?



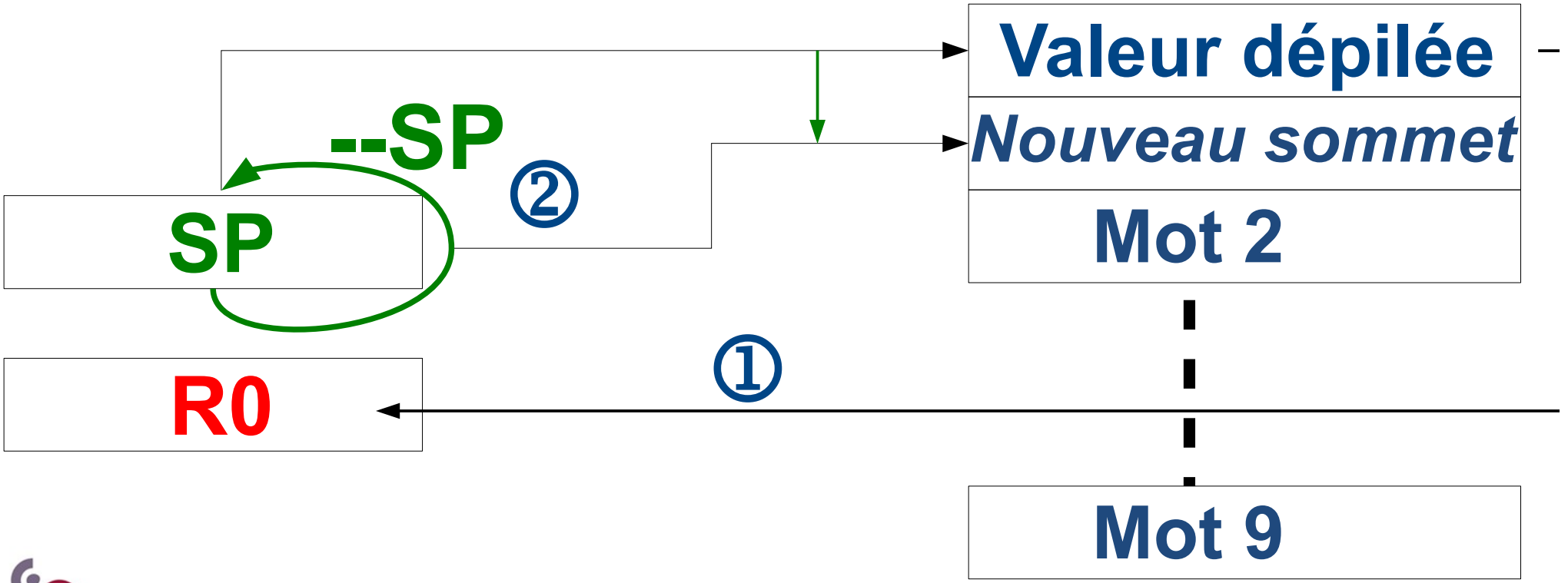
7.4.2.2 Empilage

ADD SP,SP, #1
STR R0, SP

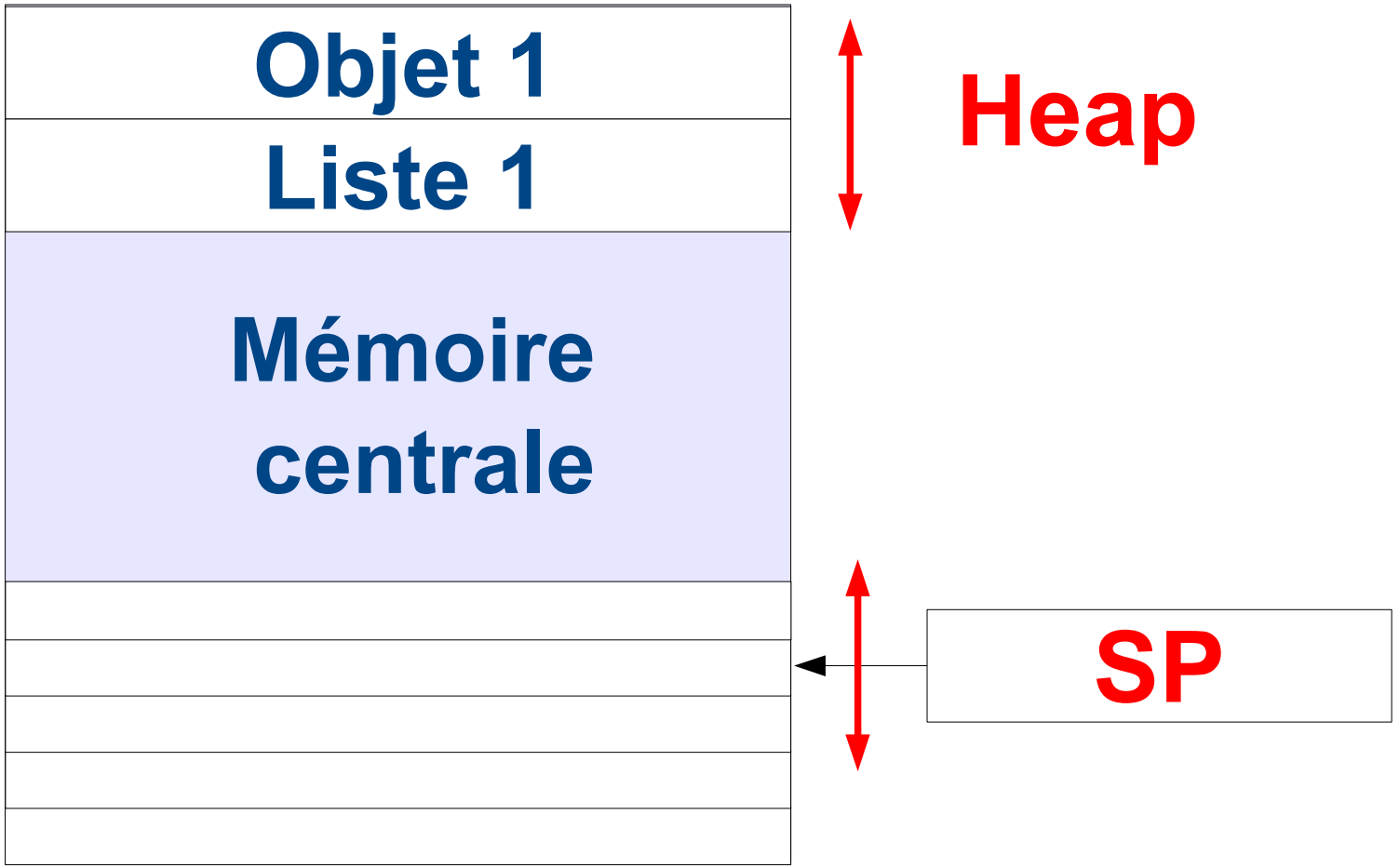


7.4.2.2 Dépilage (pop)

LDR R0,SP
SUB SP, SP, #1



7.4.3 Pile et Heap



8. Pour aller plus loin

8.1 Mode privilégié
8.2 Appels système
8.3 Pipeline
8.4 Architectures alternatives

8.1 Mode privilégié

Pour limiter au code système :

L'accès à certaines instructions

Certaines fonctionnalités

Lui donner des droits d'accès privilégiés

Dans l'intérêt des utilisateurs

8.2 Appels système

Pour autoriser un code normal :

À passer en mode privilégié

→ via des fonctions système

→ Appelées normalement

→ ayant un code fiable

Pour des opérations sensibles

Tout en étant sûr du résultat

8.2 Appels système

Pour :

Lancer une entrée-sortie,

Accéder à une ressource,

→ Demander de la mémoire,

→ Exécuter un programme,

→ Accéder aux données système

8.3 Pipeline

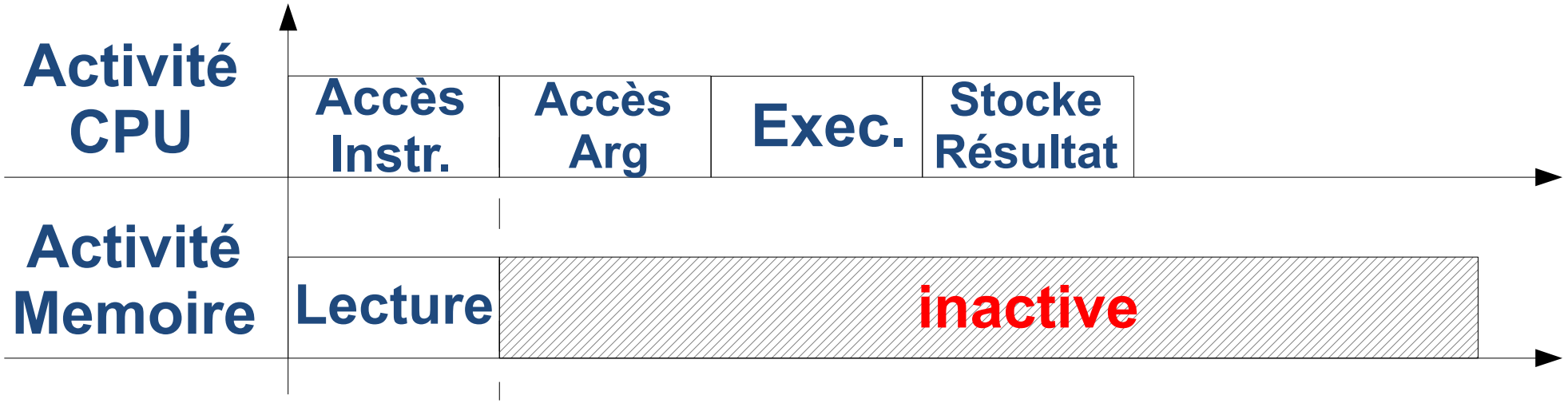
L'accélération par les caches est limitée par le coût et la technologie !

Quoi d'autre ?

Une meilleure utilisation de la mémoire !

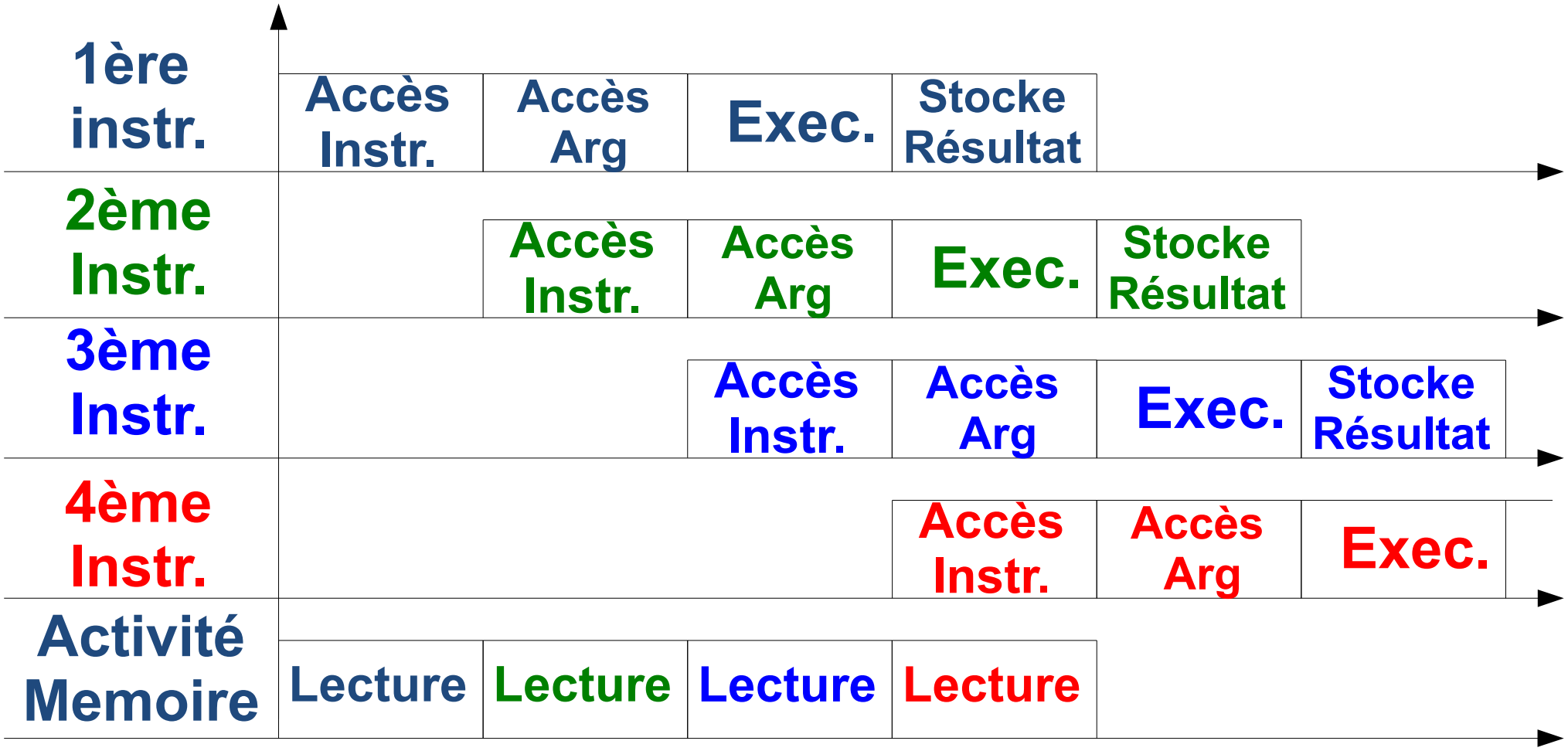
8.3 Pipeline

Découpons l'exécution de toute instruction en 4 étapes…



4 ticks d'horloge / instructions

8.3.1 Principe du Pipeline



8.3.1 Principe du Pipeline

On découpe l'exécution en N

→ N instructions en parallèle

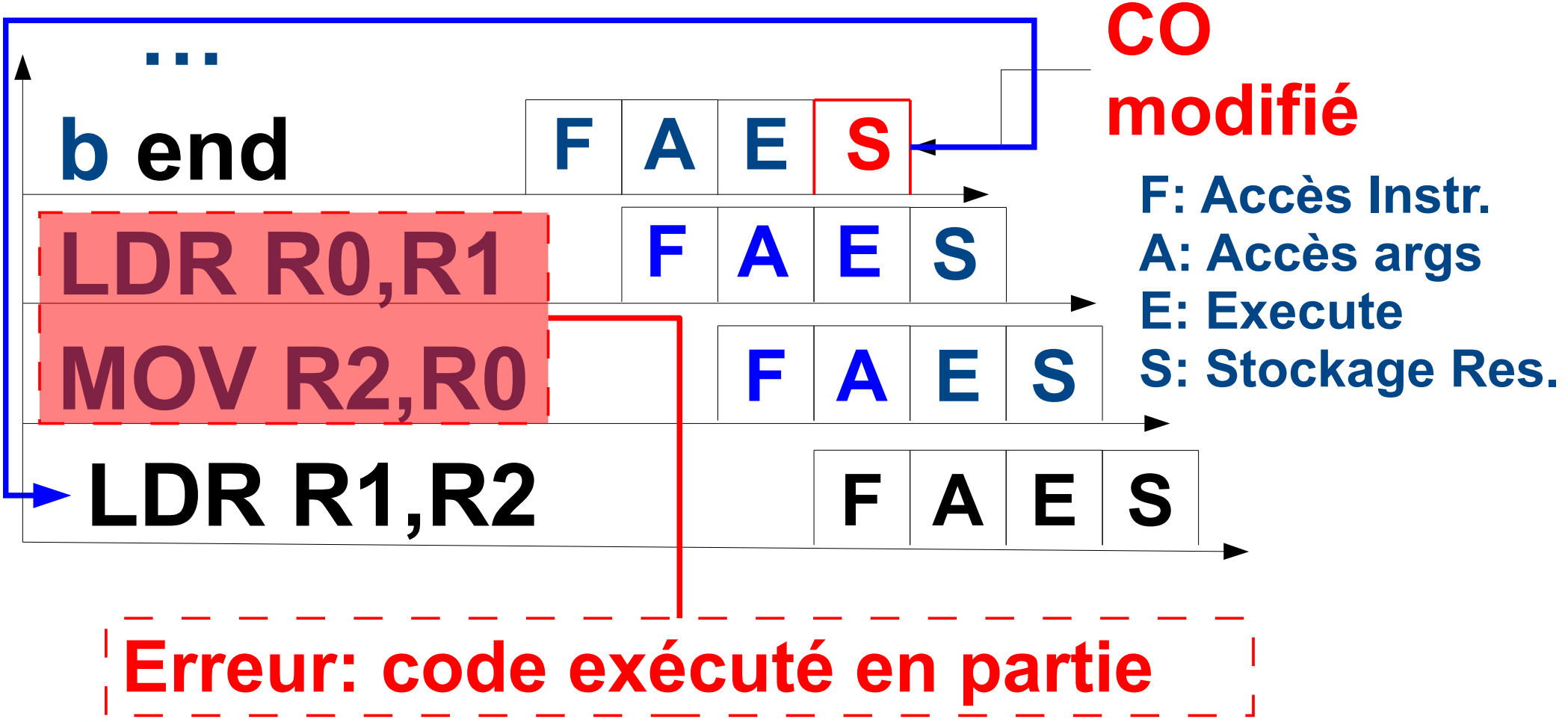
→ 1 fin d'instruction / période

Nb d'éxécutions d'instruction * N

Avec un faible surcoût matériel

Si **toutes** les instructions sont sur **registres** !!

8.3.2 Pb de Pipeline (1/3)



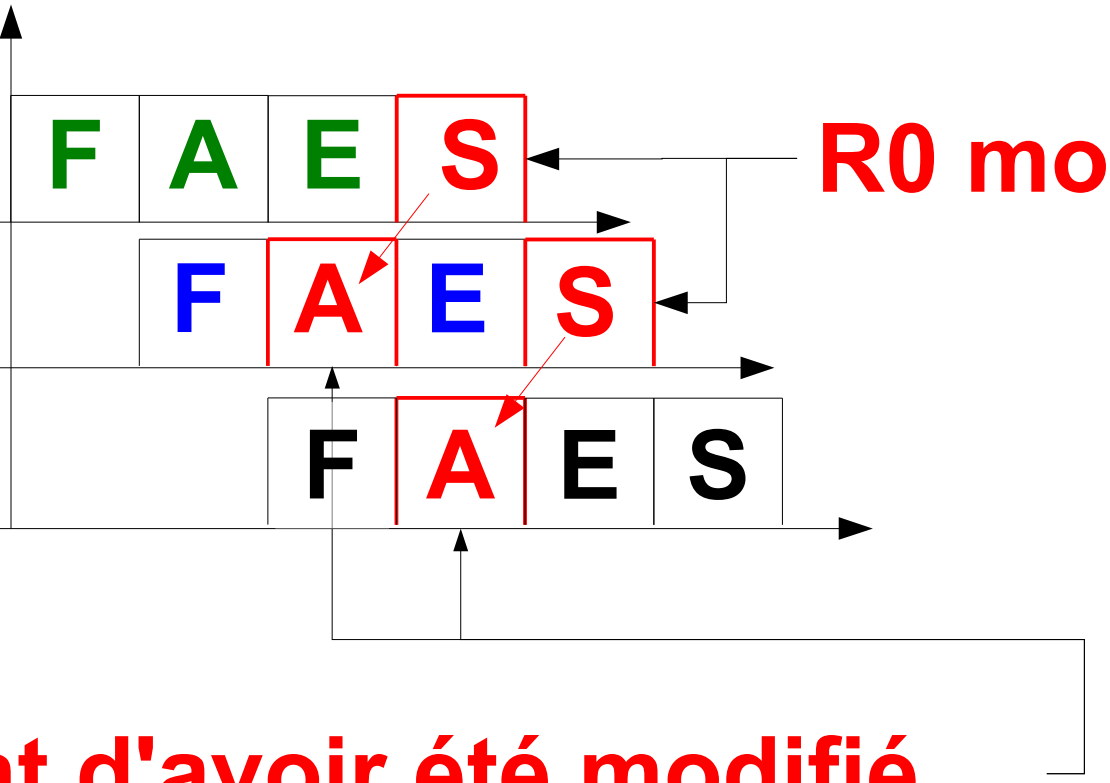
8.3.2 Pb de Pipeline (2/3)

...

LDR R0,R2

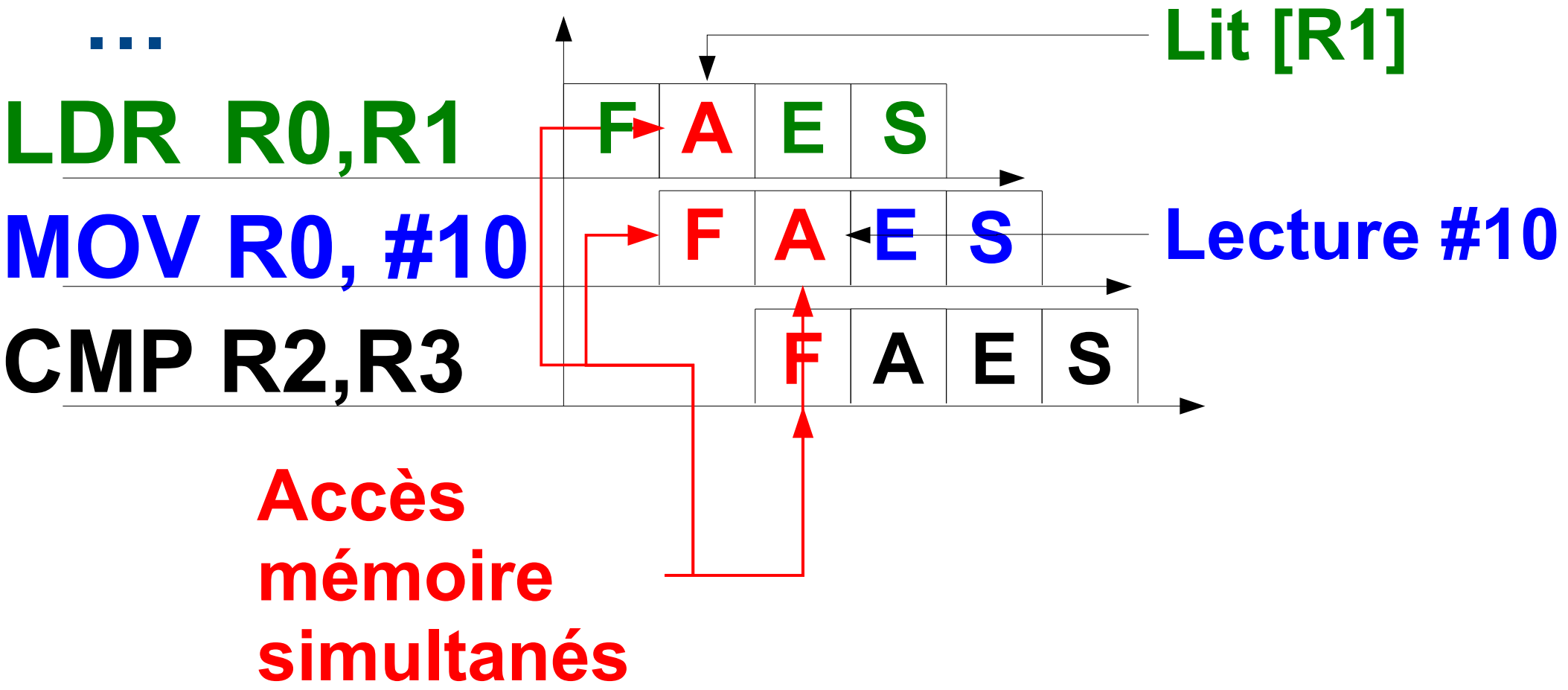
ADD R0,R1,R0

STR R0,R2



R0 est lu avant d'avoir été modifié

8.3.2 Pb de Pipeline (3/3)



8.3.3 Solutions (1/3)

Minimiser les accès mémoire

→ A quelques instructions

→ modèle “Load-and-Store”

→ beaucoup de registres

Les registres restent partagés !

8.3.3 Solutions (2/3)

Pour supprimer les conflits:

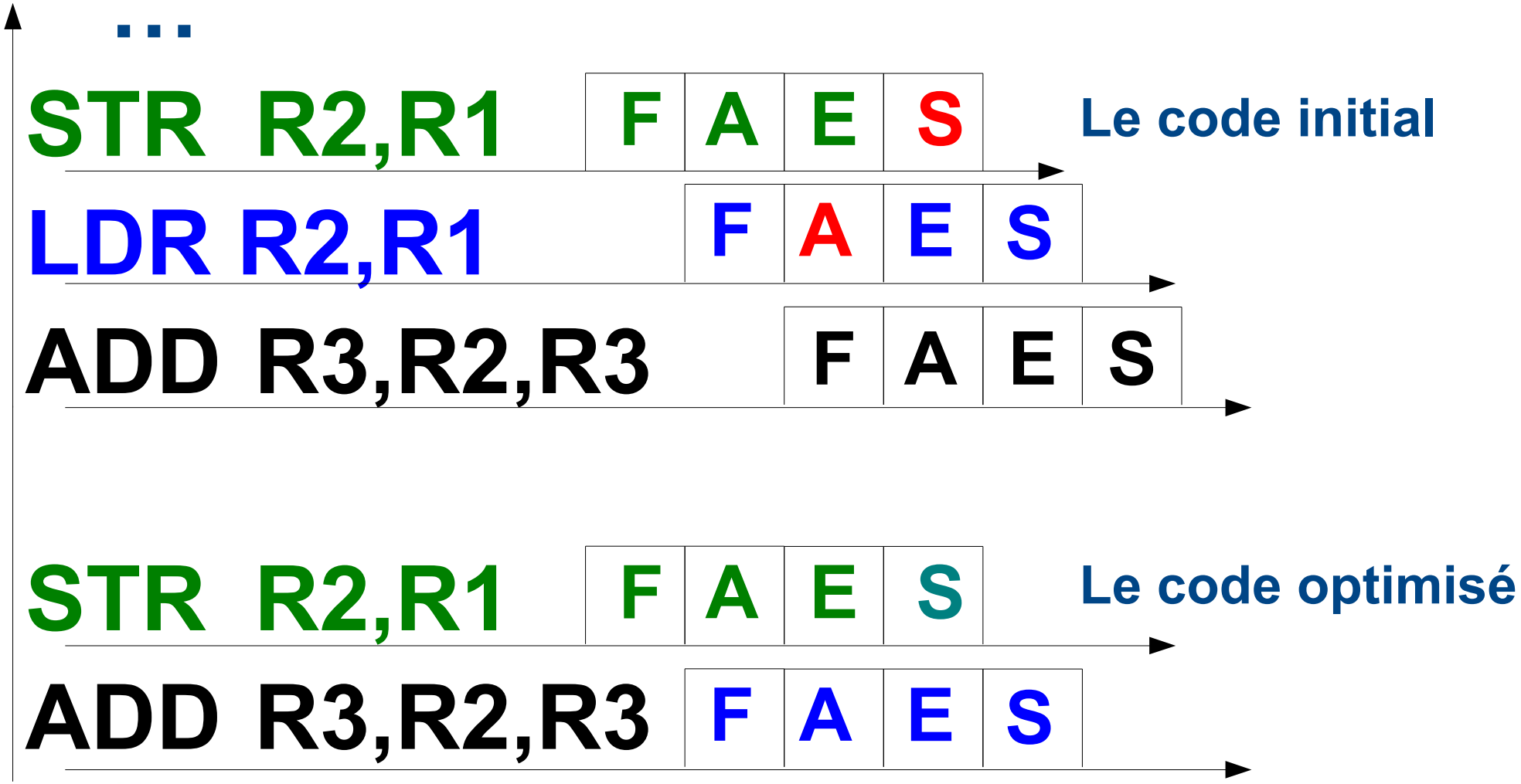
→ on demande au programmeur

→ l'assembleur insère des NOP

→ le compilateur déplace des instructions

→ le processeur ré-ordonne les instructions à l'exécution

8.3.3 Solutions (3/3)



8.4 Architectures autres

CISC vs RISC
Architecture Harvard
Multi-coeurs

8.4.1 CISC

Complex Instructions Set CPU

Les mémoires étaient lentes…

→ Fréquence CPU basse

→ Plus de temps pour exécuter

→ instructions très riches

→ et de taille variable

8.4.1 CISC

Typiquement:

Arguments souvent en mémoire

Modes d'adressage complexes

→ 3 modes d'adressage !

→ grandes instructions

Instruction de boucle, par ex.

8.4.1 RISC

Reduced Instruction Set CPU

- → Instructions simples
- → modèle LOAD and STORE
- → Moins d'accès mémoire
- → Plus de registres
- → Fréquence d'horloge élevée

8.4.1 CISC / RISC

CISC	RISC
Microprocesseurs 8 bits	
68X00 (Motorola)	MIPS (Berkeley)
386, IAPX-432 (Intel)	Alpha (DEC)
Vax (DEC)	Sparc (Sun)
Pentium (Intel)	PowerPC (IBM)
	ARM

8.4.1 RISC / CISC

IBM POWER5+	Intel-Xeon 5160
64 bits	32/64 bits
~ 2.3 GHz	~ 3 GHz
L1: 64k (I) / 32k (D)	32k (I) / 32k (D)
L2: 2 M	4 M
L3: 36 M (externe)	Aucun
276 million portes	291 million

8.4.2 Architecture Harvard

Dès que Instructions et Données
- sont stockées séparément
- sont traitées différemment

Exemple :
Caches d'instruction et de données séparés

8.4.3 Multi-coeurs

Un CPU ne suffit plus ?

Prenez un double-coeur, octo-coeurs,… !

Performances / nb coeurs cste ?

Problèmes spécifiques de ces architectures ?

8.4.3 Problèmes

La mémoire centrale est partagée

Comme les ressources

→ Comment gérer le partage ?

→ Quel est le coût du partage ?

→ Comment les caches des coeurs coopèrent-ils ?

8.4.3 Rendement

Le système va-t-il répartir la charge entre les coeurs ?
Mes applications courantes ont besoin de combien de coeurs ?

8.4.3 Rendement

Unix est fait pour gérer beaucoup de petites **tâches**.

Il divise un gros traitement en tâches séparées,

Confiées à des coeurs différents.

Grâce à une bonne communication entre tâches.

8.4.4 Benchmarks

L'architecture employée:

Intel Core2 Duo CPU E6550

Dual core,

2.33GHz (bus 1.33 GHz),

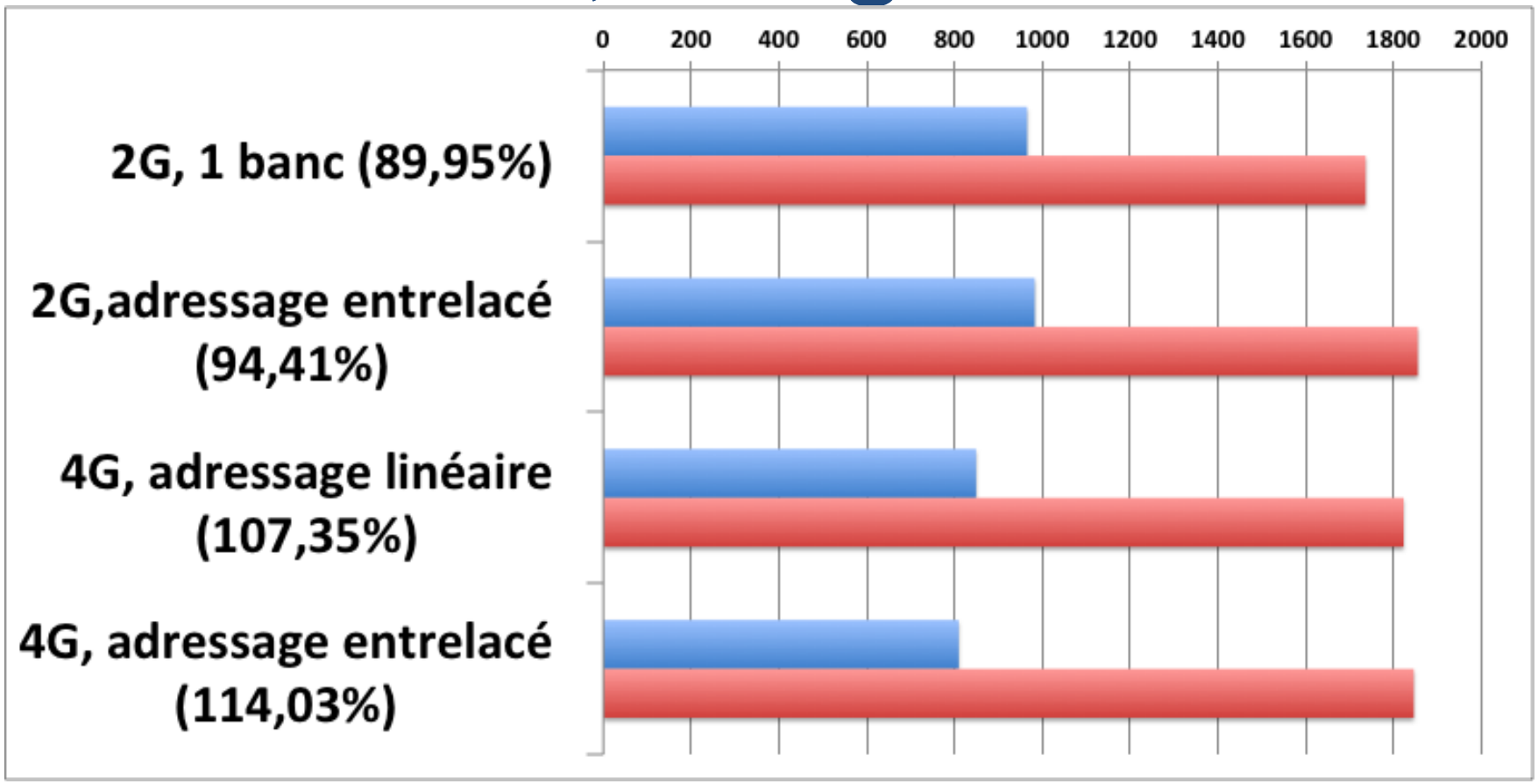
Cache de niveau 1: 2 x 64K

Cache de niveau 2: 4 MB

x86-64

8.4.4 Résultats globaux

Bleu: 1 coeur, Rouge: 2 coeurs



8.4.4 Résultats globaux

Conclusions:

2 coeurs pas toujours > 2 * CPU

2 demi-bancs mémoire: mieux qu'un grand

2 coeurs meilleurs avec plus de mémoire, adressage entrelacé