

Information, Calcul et Communication

Module 3 : Systèmes

La question de ce module

- ▶ Comment **fonctionne** et de quoi est **fait** un ordinateur capable de traiter de **l'information** avec des **algorithmes** ?

Les réponses de ce module

Comment fonctionne et de quoi est fait un ordinateur capable traiter de l'information avec des algorithmes ?

A base de trois technologies :

- ▶ Des **transistors** (pour le processeur et la mémoire vive)
☞ Leçons 1 (Architecture) & 2 (Hiérarchie)
- ▶ Des **disques** et autres Flash (pour les mémoires mortes)
☞ Leçons 2 (Hiérarchie) & 3 (Stockage)
- ▶ Des **réseaux** (pour les communications entre machines et utilisateurs)
☞ Leçon 4 (Réseaux)

Information, Calcul et Communication

Architecture des ordinateurs à programmes enregistrés

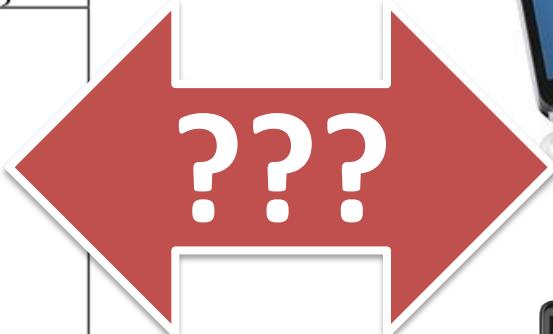
Paolo lenne, Willy Zwaenepoel,
Anastasia Ailamaki, Phil Janson

La première question de cette leçon

- Maintenant qu'on a développé des algorithmes, comment peut-on **construire des systèmes pour les exécuter** ?

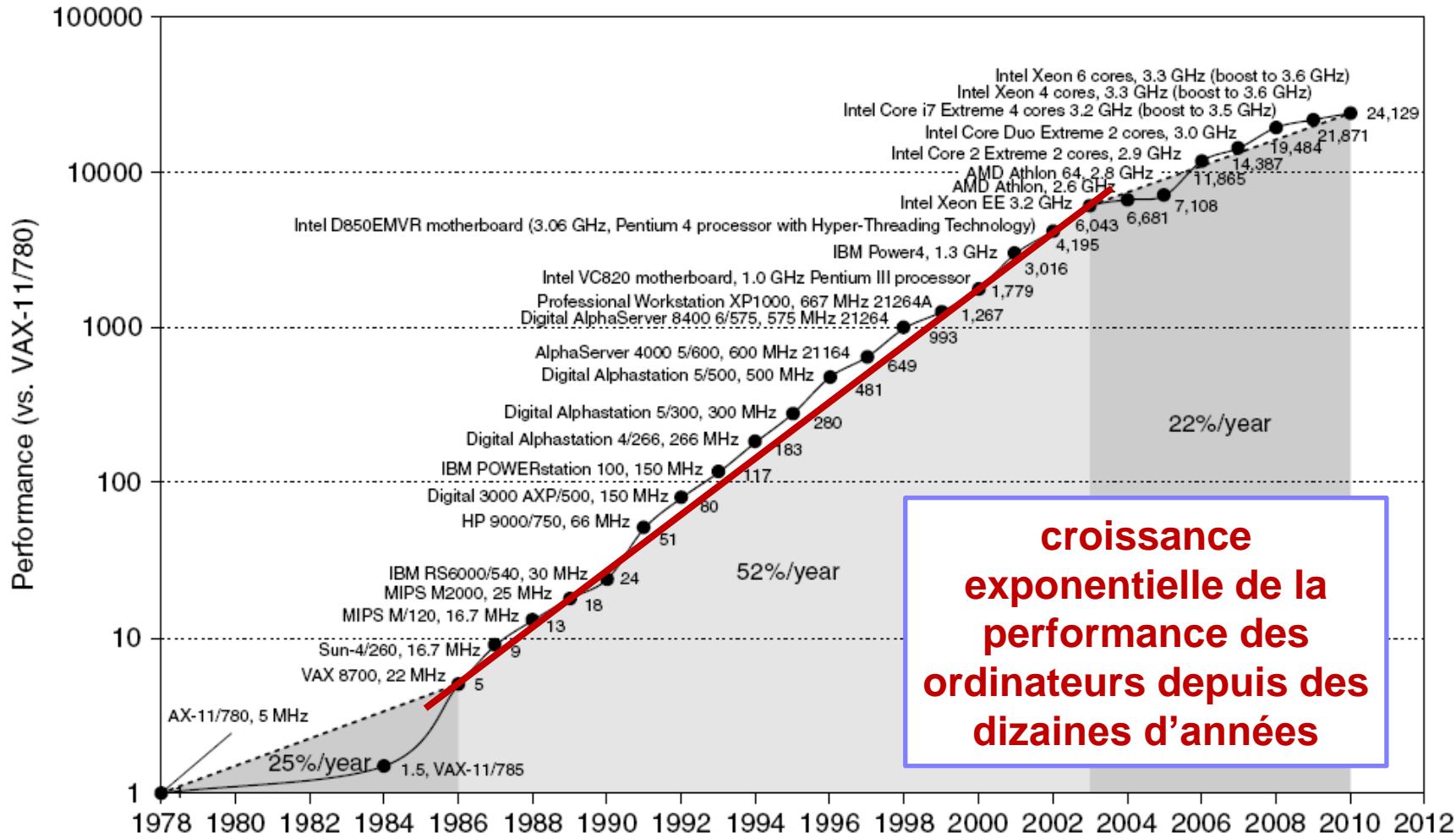
☞ **Algorithme :**

Second degré
entrée : b, c
sortie : $\{x \in \mathbb{R} : x^2 + bx + c = 0\}$
$\Delta \leftarrow b^2 - 4c$
<i>Si</i> $\Delta < 0$
afficher \emptyset
<i>Sinon</i>
<i>Si</i> $\Delta = 0$
$x \leftarrow -\frac{b}{2}$
afficher x
<i>Sinon</i>
$x_1 \leftarrow \frac{-b - \sqrt{\Delta}}{2},$
$x_2 \leftarrow \frac{-b + \sqrt{\Delta}}{2}$
afficher x_1 et x_2



Le deuxième question de cette leçon

- ▶ Comment peut-on rendre ces systèmes **plus rapides** ?

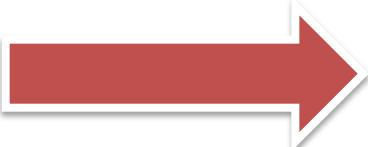


Des algorithmes aux ordinateurs

somme des premiers
 n entiers

entrée : n
sortie : m

```
s ← 0
tant que  $n > 0$ 
    s ← s + n
    n ← n - 1
m ← s
```



somme des premiers
 n entiers

entrée : $r1$
sortie : $r2$

```
1: charge r3, 0
2: cont_neg r1, 6
3: somme r3, r3, r1
4: somme r1, r1, -1
5: continue 2
6: charge r2, r3
```



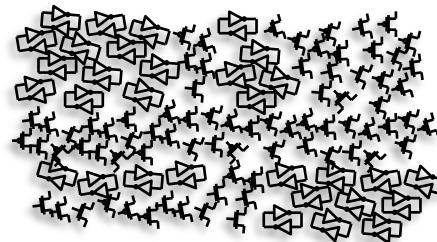
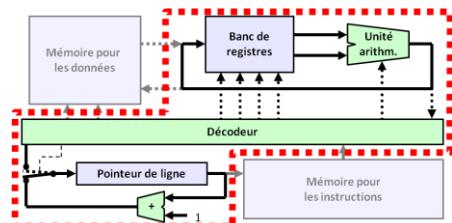
somme des premiers
 n entiers

entrée : $r1$
sortie : $r2$

```
1: 0100010010111010100
2: 01011000111000000101
3: 1110101101010010010010
4: 111010110101000010011
5: 0001100101010010101
6: 0100010110010111001
```

Logiciel

Matériel



Des algorithmes aux ordinateurs

0
somme des premiers
 n entiers
entrée : n
sortie : m
 $s \leftarrow 0$
tant que $n > 0$
 $s \leftarrow s + n$

On va partir
des algorithmes
qu'on a étudié...

1
somme des premiers
 n entiers
entrée : $r1$
sortie : $r2$
1: charge r3, 0
2: cont_neg r1, 6
3: somme r3, r3, r1
...

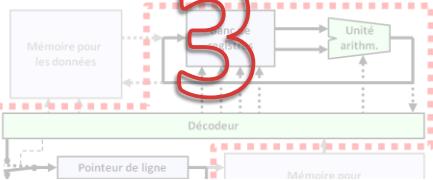
...pour les réécrire
d'une façon plus
formelle...

2
somme des premiers
 n entiers
entrée : $r1$
sortie : $r2$
1: 0100010010111010100
2: 0101100011100000101
3: 1110101110101001001010

...et les rendre
compréhensibles
à une machine

Logiciel

Matériel



En parallèle, on va créer
une machine abstraite...



...pour la réaliser
avec des transistors



Un algorithme

**somme des premiers
 n entiers**

entrée : n

sortie : m

$s \leftarrow 0$

tant que $n > 0$

$s \leftarrow s + n$

$n \leftarrow n - 1$

$m \leftarrow s$

- ▶ Pour exprimer l'idée d'un algorithme, dans le Module 1 on a utilisé une sorte de « langage » intuitif, parfois proche du langage naturel

Essayons de le réécrire avec moins de liberté

On a besoin de mémoriser des valeurs

on va se restreindre à des symboles comme **r1, r2, r3,...**

**somme des premiers
n entiers**

entrée : *n*

sortie : *m*

$s \leftarrow 0$

tant que $n > 0$

$s \leftarrow s + n$

$n \leftarrow n - 1$

$m \leftarrow s$

Les « registres »

- ▶ On appelle les variables « registres »
- ▶ On les représente par **r1, r2, r3, ...**
- ▶ On remplace tous les noms arbitraires de nos variables par ces nouveaux noms
 - n → **r1**
 - m → **r2**
 - s → **r3**

Etape 1.1

**somme des premiers
n entiers**

entrée : ***n***

sortie : ***m***

***s* ← 0**

tant que *n* > 0

s* ← *s* + *n

***n* ← *n* – 1**

m* ← *s



**somme des premiers
n entiers**

entrée : ***r1***

sortie : ***r2***

***r3* ← 0**

tant que *r1* > 0

r3* ← *r3* + *r1

***r1* ← *r1* – 1**

r2* ← *r3

Continuons...

**somme des premiers
 n entiers**

entrée : r1
sortie : r2

$r3 \leftarrow 0$

tant que $r1 > 0$

$r3 \leftarrow r3 + r1$

$r1 \leftarrow r1 - 1$

$r2 \leftarrow r3$

On a besoin
d'assigner des valeurs
à ces registres

On va écrire cela
de façon très régulière

p.ex.,

« **charge r3, 0** »
pour signifier $r3 \leftarrow 0$

ou

« **charge r2, r3** »
pour signifier $r2 \leftarrow r3$

Etape 1.2

**somme des premiers
 n entiers**

entrée : r1

sortie : r2

$r3 \leftarrow 0$

tant que $r1 > 0$

$r3 \leftarrow r3 + r1$

$r1 \leftarrow r1 - 1$

$r2 \leftarrow r3$



**somme des premiers
 n entiers**

entrée : r1

sortie : r2

charge r3, 0

tant que $r1 > 0$

$r3 \leftarrow r3 + r1$

$r1 \leftarrow r1 - 1$

charge r2, r3

Continuons...

somme des premiers
 n entiers

entrée : n
sortie : m

charge r3, 0

tant que $n > 0$

$r3 \leftarrow r3 + r1$

$r2 \leftarrow r2 - 1$

charge r2, r3

On a besoin d'assigner des nouvelles valeurs à ces registres suite à des opérations arithmétiques

on va écrire cela
De façon très régulière
p.ex.,
« somme r3, r3, r1 »
pour signifier $r3 \leftarrow r3 + r1$

Etape 1.3

**somme des premiers
 n entiers**

entrée : r1

sortie : r2

charge r3, 0

tant que $r1 > 0$

$r3 \leftarrow r3 + r1$

$r1 \leftarrow r1 - 1$

charge r2, r3



**somme des premiers
 n entiers**

entrée : r1

sortie : r2

charge r3, 0

tant que $r1 > 0$

somme r3, r3, r1

somme r1, r1, -1

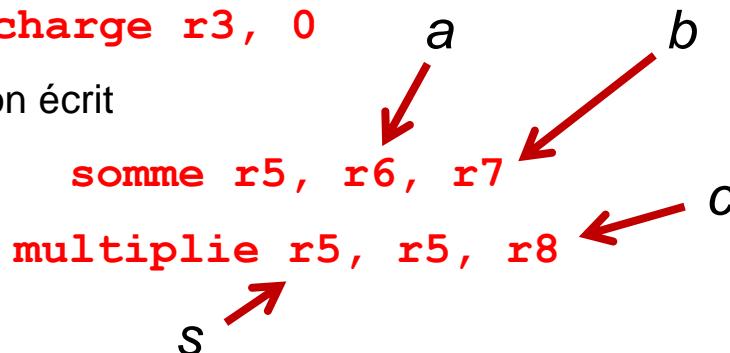
charge r2, r3

Les opérations ou « instructions »

- ▶ On définit un **nombre limité** d'opérations; p.ex.
 - **charge** pour l'assignation
 - **somme** pour l'addition
 - **soustrait** pour la soustraction
- ▶ Toutes les opérations ont **un résultat et** opèrent sur **une ou deux valeurs ou opérandes**, jamais plus
- ▶ Les opérandes sont **soit des registres soit des constantes** (p.ex., 73)
- ▶ On écrit ces opérations ainsi

somme destination , operande1 , operande2

- Au lieu d'écrire $s \leftarrow s + n$ on écrit **somme r3, r3, r1**
- Au lieu d'écrire $s \leftarrow 0$ on écrit **charge r3, 0**
- Au lieu d'écrire $s \leftarrow c (a + b)$ on écrit



Continuons...

**somme des premiers
 n entiers**

entrée : r1
sortie : r2

charge r3, 0

tant que $r1 > 0$

~~somme r3, r3, r1~~
~~somme r1, r1, -1~~

charge r2, r3

Peut-être peut-on faire quelques modifications...

somme des premiers
 n entiers

entrée : n

sortie : m

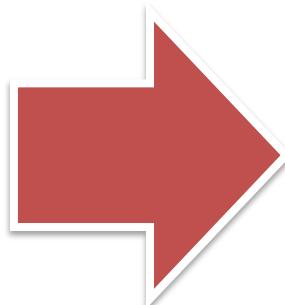
$s \leftarrow 0$

tant que $n > 0$

$s \leftarrow s + n$

$n \leftarrow n - 1$

$m \leftarrow s$



somme des premiers
 n entiers

entrée : n

sortie : m

$s \leftarrow 0$

si $n \leq 0$ continue ici

sinon

$s \leftarrow s + n$

$n \leftarrow n - 1$

continue ici

$m \leftarrow s$

C'est un peu plus « tordu »
mais il est facile de se convaincre
que c'est **exactement la même chose**

Attention: Une transformation un peu tordue

somme des premiers
 n entiers

entrée : r1

sortie : r2

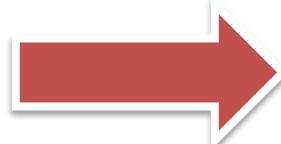
charge r3, 0

tant que $r1 > 0$

 somme r3, r3, r1

 somme r1, r1, -1

charge r2, r3



somme des premiers
 n entiers

entrée : r1

sortie : r2

charge r3, 0

→ si $r1 \leq 0$ continue

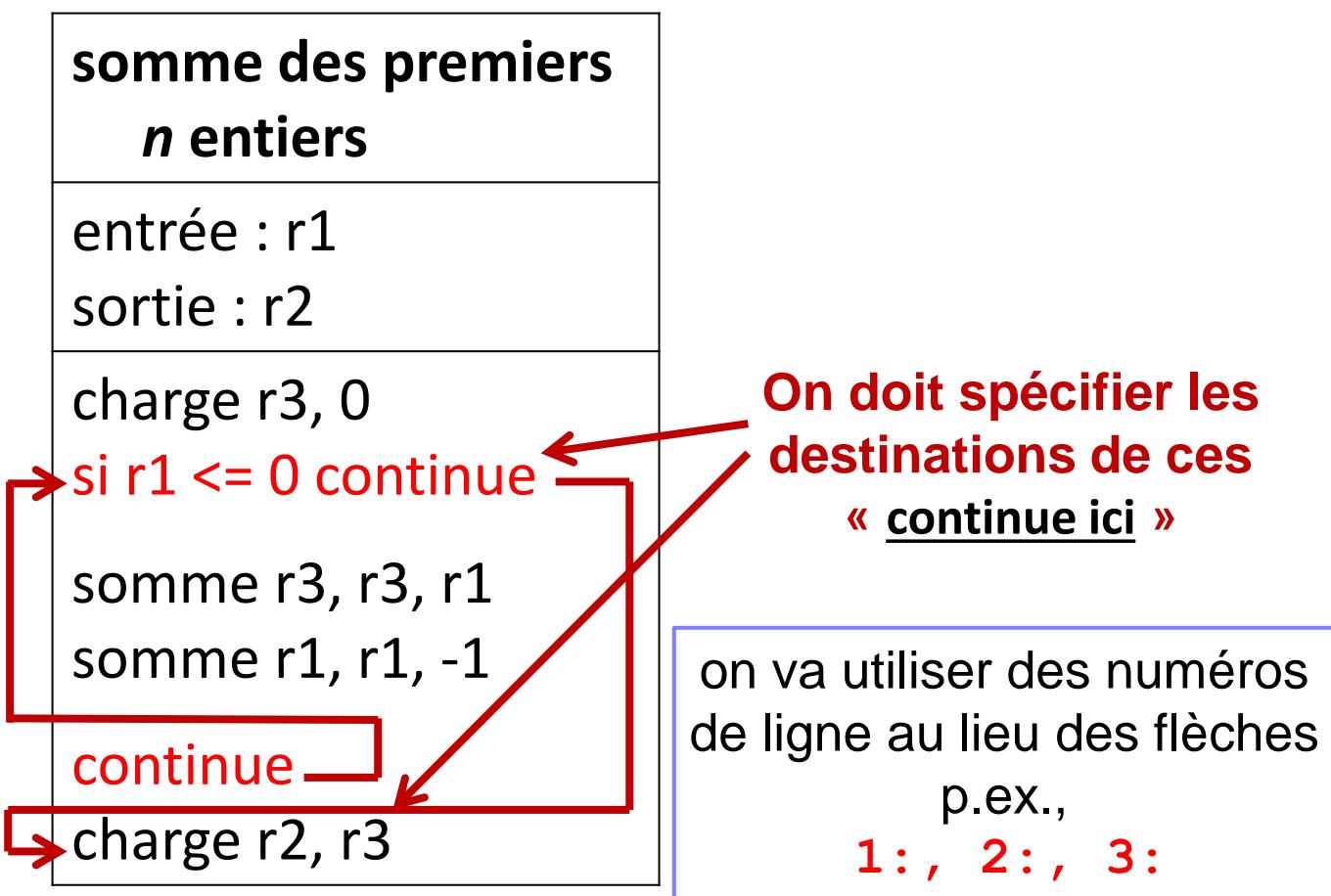
 somme r3, r3, r1

 somme r1, r1, -1

 continue

→ charge r2, r3

Introduisons les numéros de ligne



Etape 1.4

**somme des premiers
n entiers**

entrée : r1

sortie : r2

charge r3, 0

si r1 <= 0 continue

somme r3, r3, r1

somme r1, r1, -1

continue

charge r2, r3



**somme des premiers
n entiers**

entrée : r1

sortie : r2

1: charge r3, 0

2: si r1 <= 0 continue 6

3: somme r3, r3, r1

4: somme r1, r1, -1

5: continue 2

6: charge r2, r3

L'instruction de saut

On a besoin de spécifier
que l'exécution ne
continue pas à la ligne
suivante

on introduit une
nouvelle
action/instruction
« **continue 2** »

somme des premiers
 n entiers

entrée : r1

sortie : r2

1: charge r3, 0

2: si r1 <= 0 continue 6

3: somme r3, r3, r1

4: somme r1, r1, -1

5: continue 2

6: charge r2, r3

L'instruction de saut conditionnel

somme des premiers
 n entiers

entrée : r1
sortie : r2

1: charge r3, 0
2: si r1 <= 0 continue 6
3: somme r3, r3, r1
4: somme r1, r1, -1
5: continue 2
6: charge r2, r3

On doit pouvoir réagir à
des conditions

on introduit des
actions/instructions
conditionnelles p. ex.,
« **cont_ppe r1, 0, 6** »
pour signifier
si $r1 \leq 0$ continue à la ligne 6

L' étape 1.5

**somme des premiers
 n entiers**

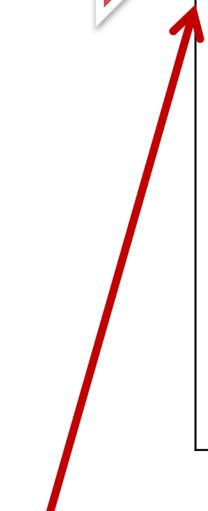
entrée : r1
sortie : r2

- 1: charge r3, 0
- 2: si $r1 \leq 0$ continue 6
- 3: somme r3, r3, r1
- 4: somme r1, r1, -1
- 5: continue 2
- 6: charge r2, r3

**somme des premiers
 n entiers**

entrée : r1
sortie : r2

- 1: charge r3, 0
- 2: cont_ppe r1, 0, 6
- 3: somme r3, r3, r1
- 4: somme r1, r1, -1
- 5: continue 2
- 6: charge r2, r3



C'est ce qu'on appelle un programme en langage “**Assembleur**”

Les « instructions »

- ▶ On définit un **nombre limité** d'instructions; p.ex.
 - **charge** pour l'assignation
 - **somme** pour l'addition
 - ...
- ▶ Toutes ces instructions
 - ont **un seul résultat** ...
 - et opèrent sur **une ou deux opérandes** ...
 - qui sont **soit des registres soit des constantes**
- ▶ On écrit ces instructions
opération destination , operande1 , operande2

Les instructions « de saut »

- ▶ La famille des instructions du type **continue** spécifie que la prochaine instruction à exécuter **n'est pas à la ligne suivante** mais à la ligne spécifiée dans l'instruction
- ▶ On définit un **nombre limité** de tests possibles; p.ex.
 - **continue 7** pour le saut vers la ligne 7 sans conditions
 - **cont_egal r1, r2, 14** pour le saut vers la ligne 14 si **r1 = r2**
 - **cont_neg r1, 31** pour le saut vers la ligne 31 si **r1 < 0**
- ▶ Toutes ces instructions ont **jusqu'à deux opérandes et une ligne de destination**
 - Au lieu d'écrire continue ici ↗ on écrit **continue 2**
 - Au lieu d'écrire si $n < 0$ continue ici ↗ on écrit **cont_neg r1, 6**

Petit résumé...

- ▶ On utilise des « **registres** » comme **r1, r2, r3**, etc.
- ▶ On écrit les programmes comme des séquences d'« **instructions** »
 - des instructions de chargement de registres
 - des instructions arithmétiques sur les registres
 - des instructions de saut à d'autres instructions
- ▶ On utilise un jeu restreint d'instructions préalablement définies

Des algorithmes aux ordinateurs: Etape 2

somme des premiers
n entiers

entrée : n
sortie : m

$s \leftarrow 0$

Tant que $n > 0$

- $s \leftarrow s + n$
- $n \leftarrow n - 1$

$m \leftarrow s$

somme des premiers
n entiers

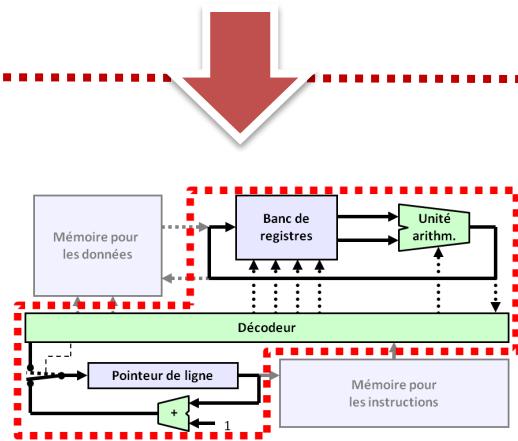
entrée : $r1$
sortie : $r2$

```

1: charge r3, 0
2: cont_neg r1, 6
3: somme r3, r3, r1
4: somme r1, r1, -1
5: continue 2
6: charge r2, r3

```

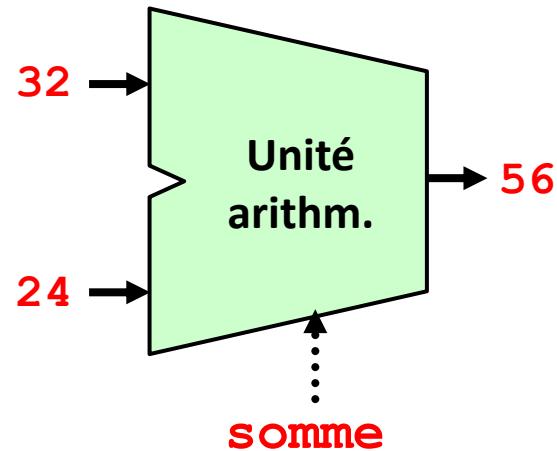
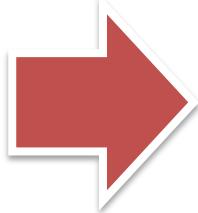
Logiciel
Matériel



De quoi a-t-on besoin pour le calcul?

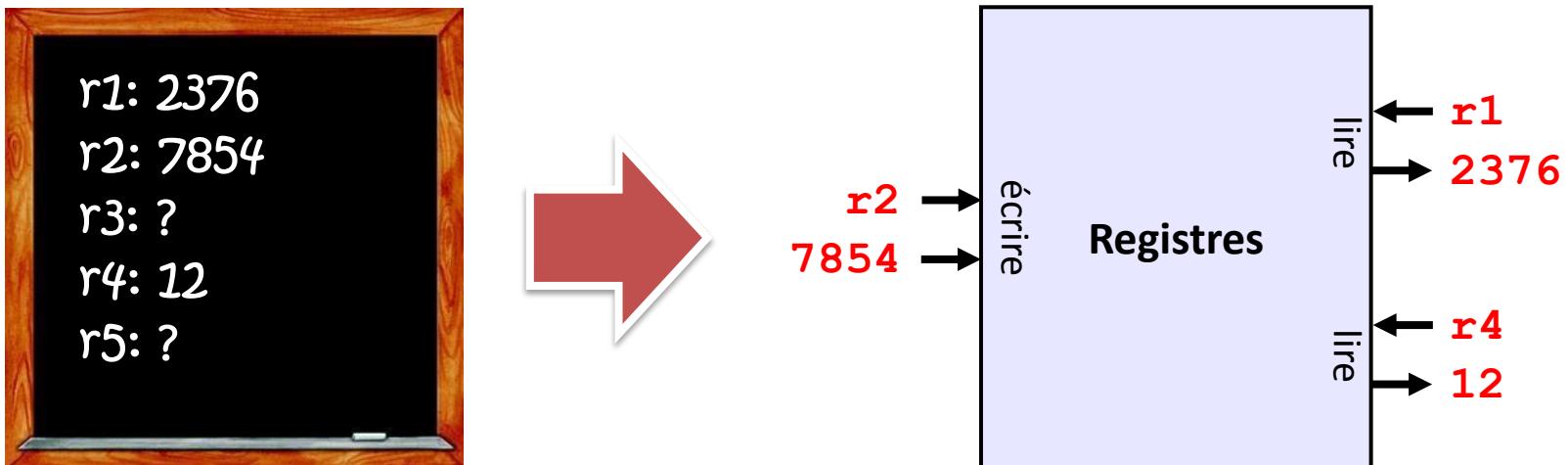
- ▶ L'unité arithmétique effectue les instructions arithmétiques

$$\begin{array}{r} 32 + \\ 24 = \\ \hline 56 \end{array}$$

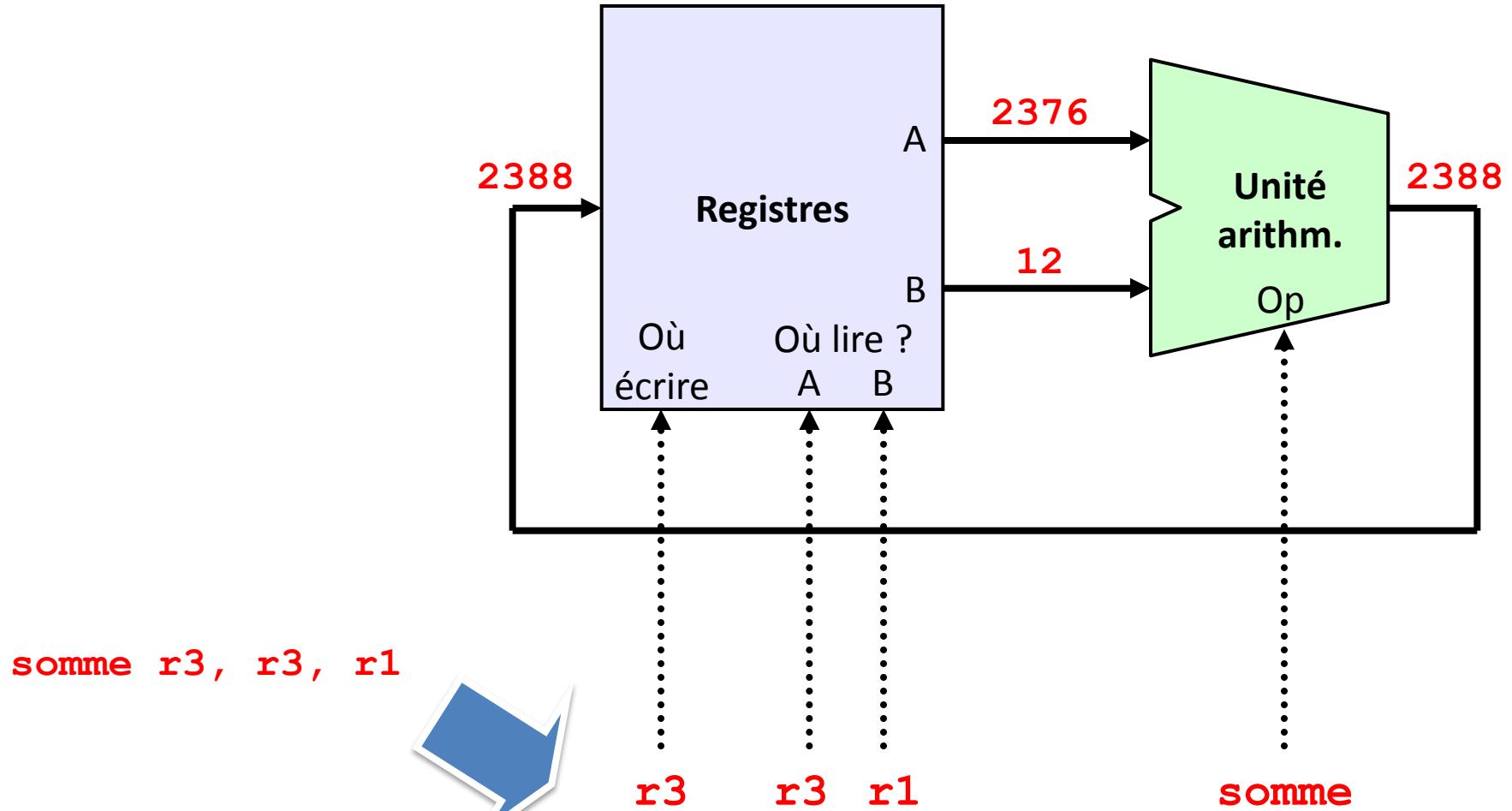


De quoi a-t-on besoin pour le calcul ?

- ▶ Les **registres** mémorisent les opérandes et les résultats

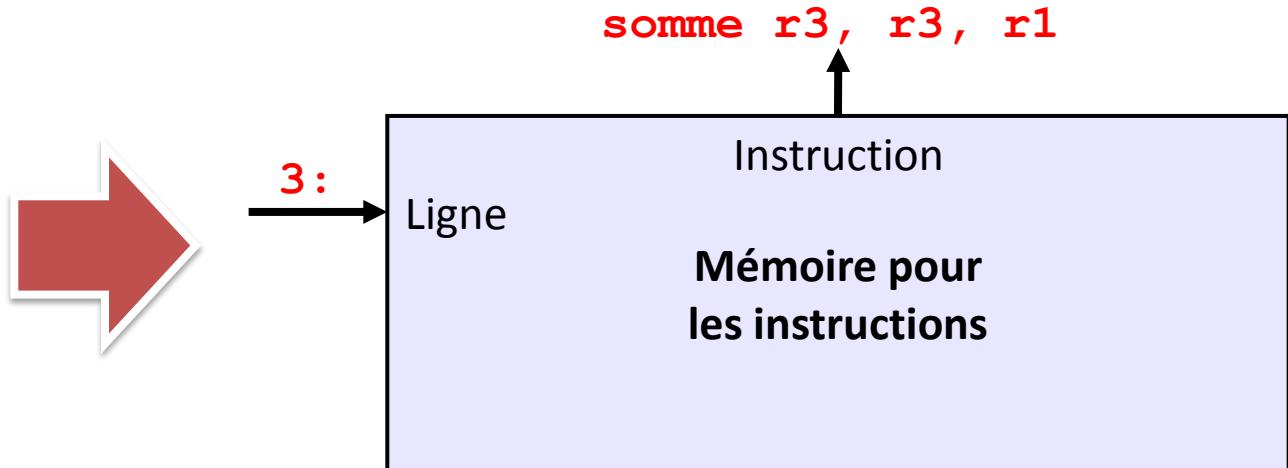
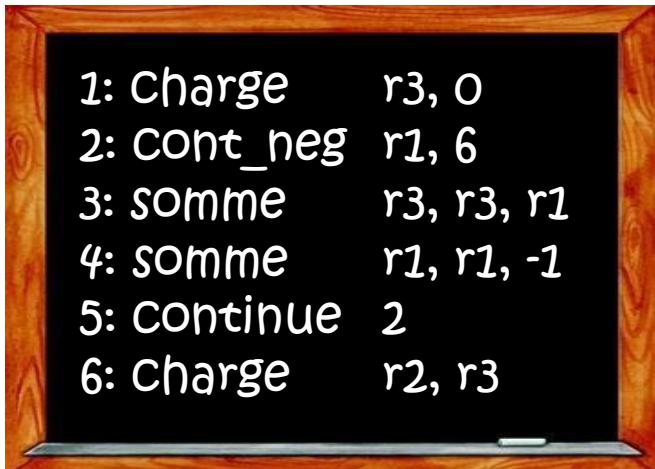


Le circuit pour le calcul



De quoi a-t-on encore besoin ?

- ▶ Notre algorithme ou programme doit être enregistré quelque part

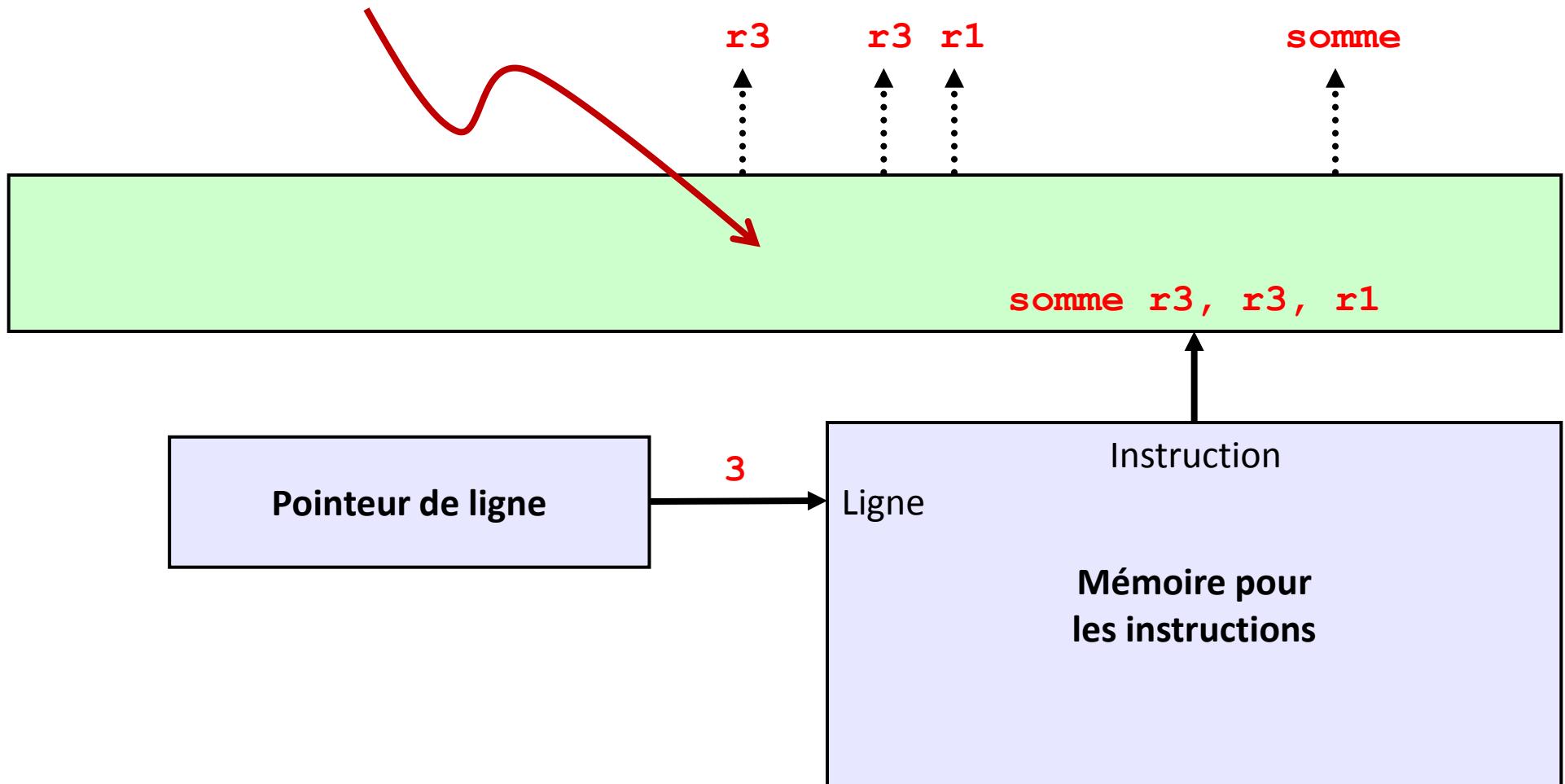


- ▶ Il faut pouvoir contrôler où on en est



Pour contrôler où on en est

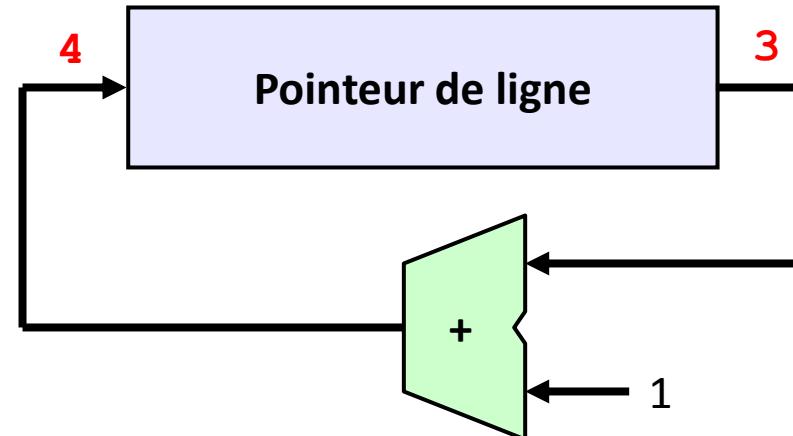
Un circuit assez simple qui répartit les éléments qui constituent une instruction



Pour contrôler où on en est

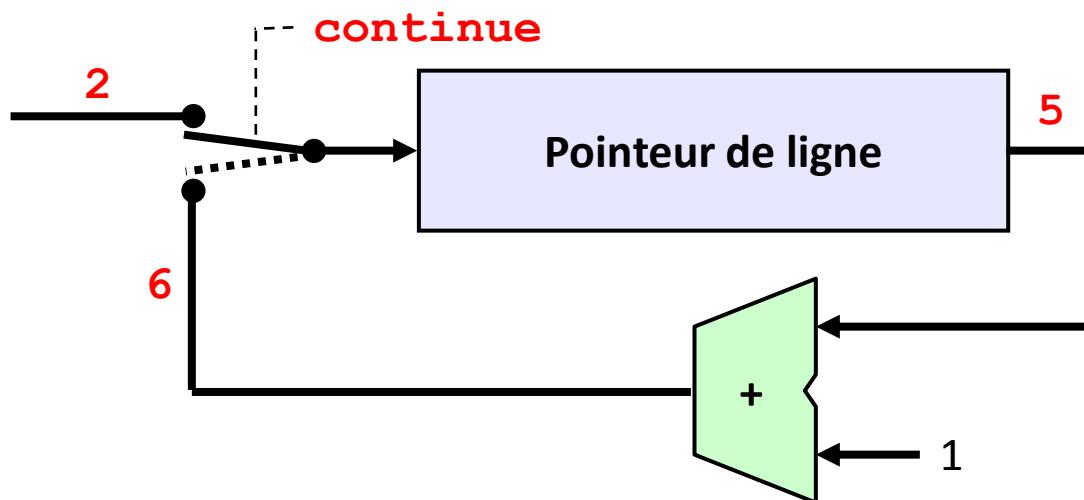
- ▶ Normalement on veut passer d'une ligne à la suivante

3: somme r3, r3, r1

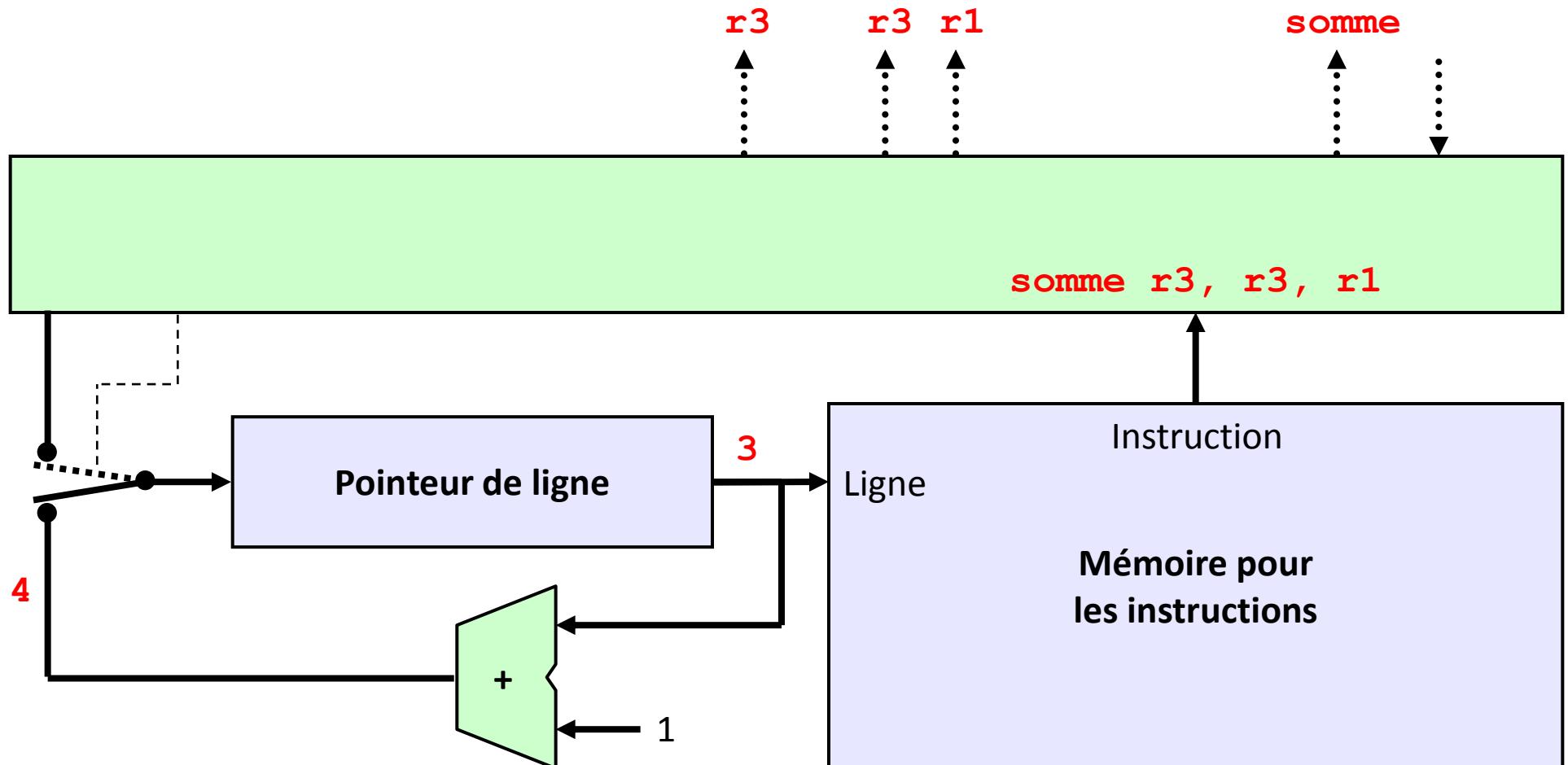


- ▶ Si on a une instruction **continue**
on veut imposer une autre ligne à lire prochainement

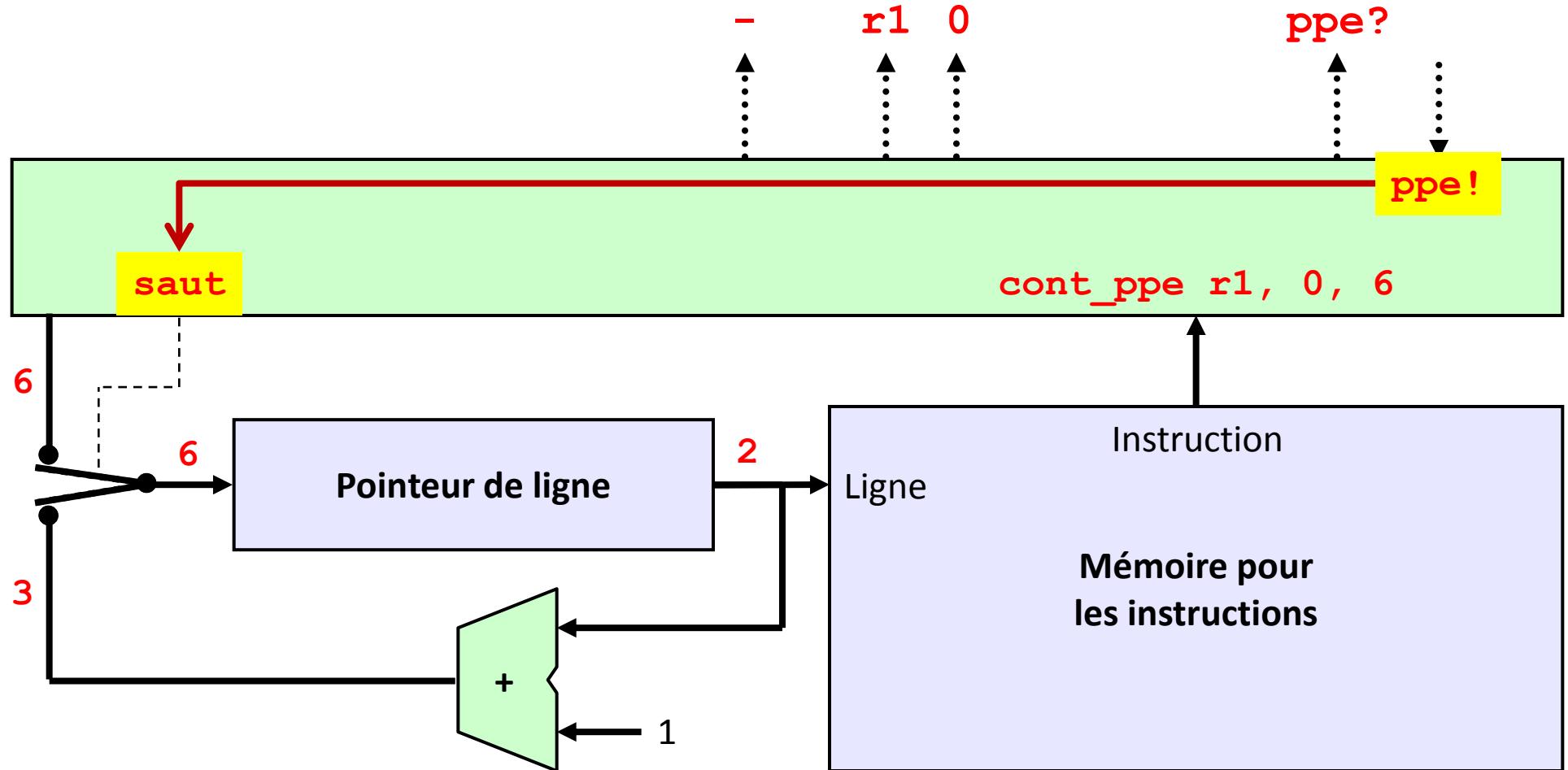
5: continue 2



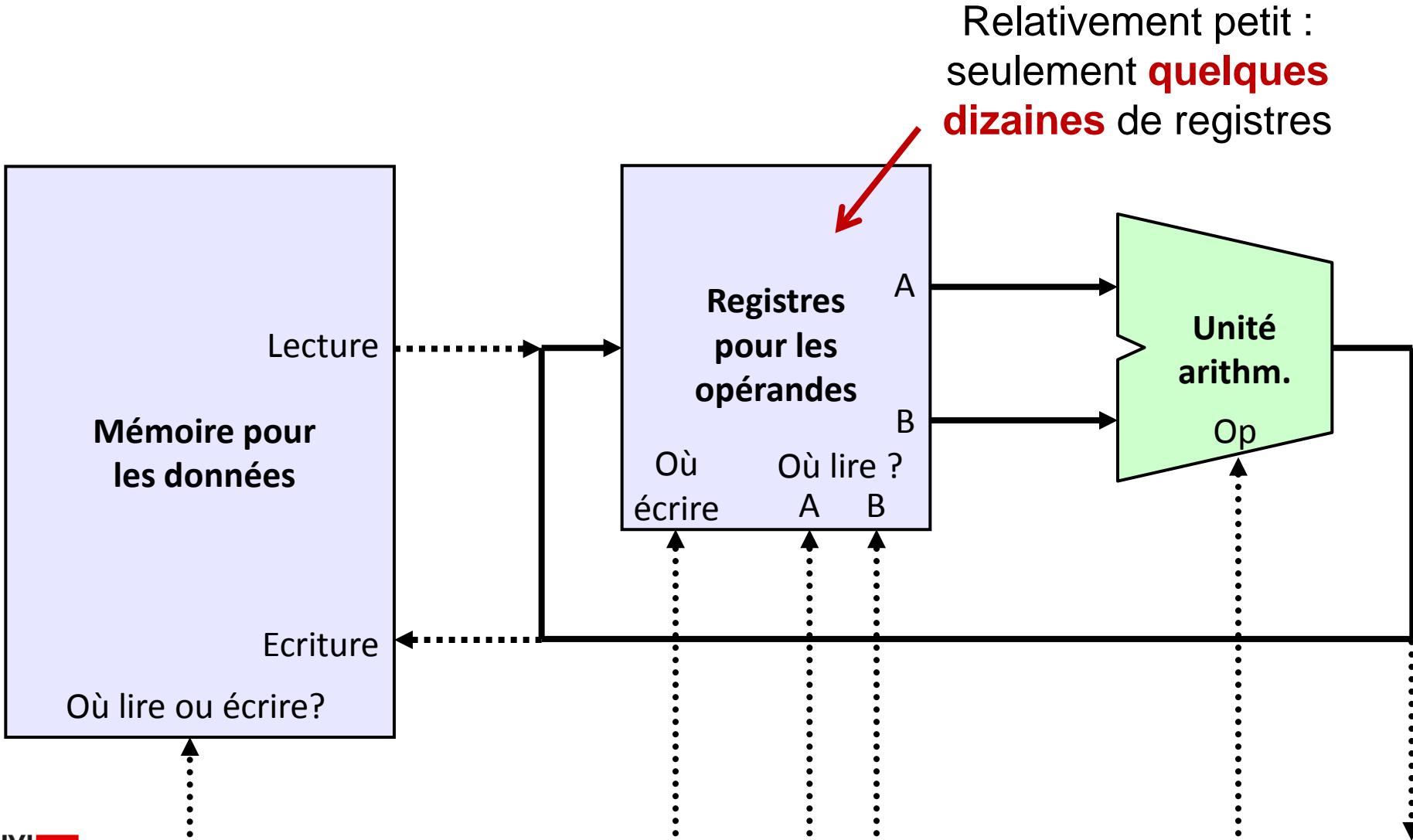
Le circuit pour le contrôle séquentiel



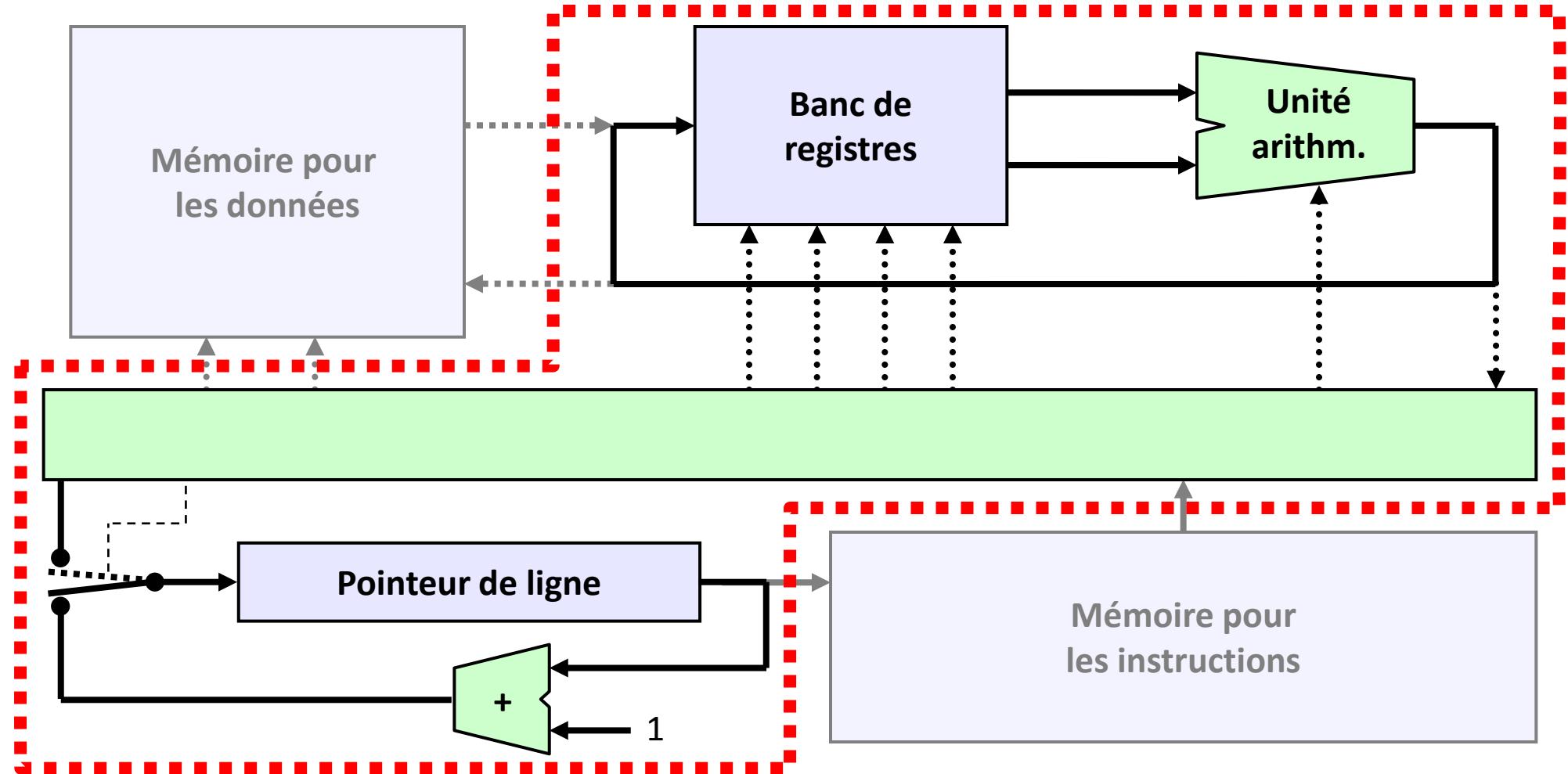
Le circuit pour le contrôle en cas de saut



Une mémoire pour avoir plus de données



Un processeur complet



Des algorithmes aux ordinateurs: Etape 3

**somme des premiers
n entiers**

entrée : n
sortie : m

```

s ← 0
Tant que  $n > 0$ 
     $s \leftarrow s + n$ 
     $n \leftarrow n - 1$ 
 $m \leftarrow s$ 

```

**somme des premiers
n entiers**

entrée : $r1$
sortie : $r2$

```

1: charge r3, 0
2: cont_neg r1, 6
3: somme r3, r3, r1
4: somme r1, r1, -1
5: continue 2
6: charge r2, r3

```

**somme des premiers
n entiers**

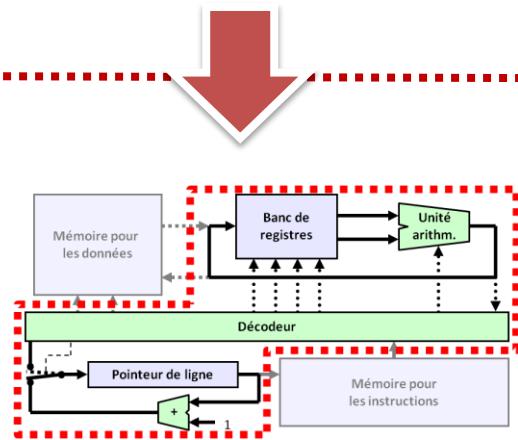
entrée : $r1$
sortie : $r2$

```

1: 0100010010111010100
2: 0101100011100000101
3: 1110101101010010010
4: 1110101101000010011
5: 0001100101010010101
6: 0100010110010111001

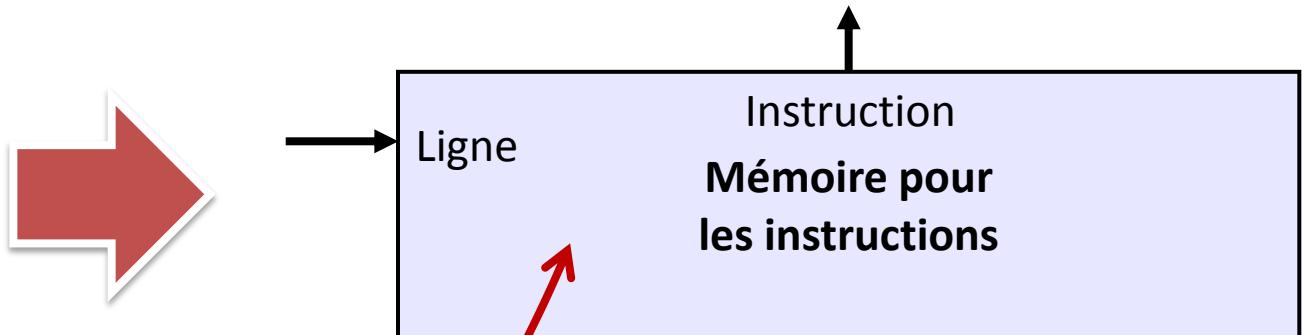
```

Logiciel
Matériel

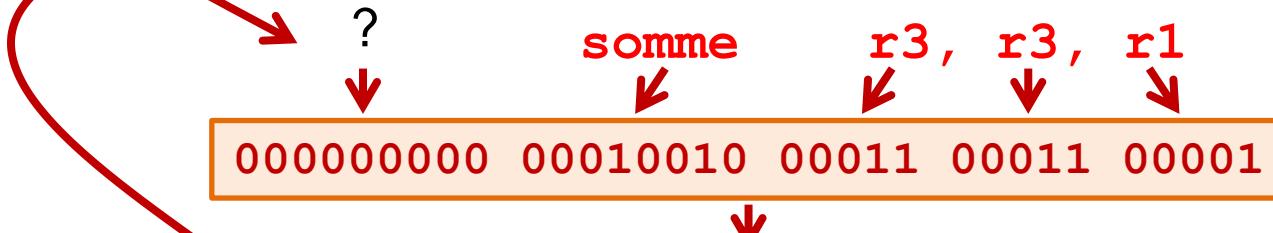


Comment encoder les instructions ?

```
1: charge    r3, 0
2: cont_neg  r1, 6
3: somme     r3, r3, r1
4: somme     r1, r1, -1
5: continue   2
6: charge    r2, r3
```



- ▶ On peut inventer un encodage simple (v. module 1, leçon 4) :
 - Quelques bits pour identifier l'opération (p.ex., si on a 256 opérations possibles, 8 bits sont suffisants)
 - Quelques bits pour identifier les registres (p.ex., si on a 32 registres, 3 x 5 bits = 15 bits)
 - Et ainsi de suite pour le reste...
- ▶ Peut-être peut-on s'en sortir avec 32 ou 64 bits comme pour un entier typique



La valeur **592,993** représente l'instruction **somme r3, r3, r1**

Encoder les instructions

somme des premiers
 n entiers

entrée : $r1$
sortie : $r2$

```
1: charge    r3, 0
2: cont_neg  r1, 6
3: somme     r3, r3, r1
4: somme     r1, r1, -1
5: continue   2
6: charge    r2, r3
```



somme des premiers
 n entiers

entrée : $r1$
sortie : $r2$

```
1: 0100010010111010100
2: 0101100011100000101
3: 1110101101010010010
4: 1110101101000010011
5: 0001100101010010101
6: 0100010110010111001
```

Langage assembleur

Langage machine en binaire

Des algorithmes aux ordinateurs: Etape 4

**somme des premiers
n entiers**

entrée : n
sortie : m

```
s ← 0
Tant que  $n > 0$ 
     $s ← s + n$ 
     $n ← n - 1$ 
 $m ← s$ 
```

**somme des premiers
n entiers**

entrée : $r1$
sortie : $r2$

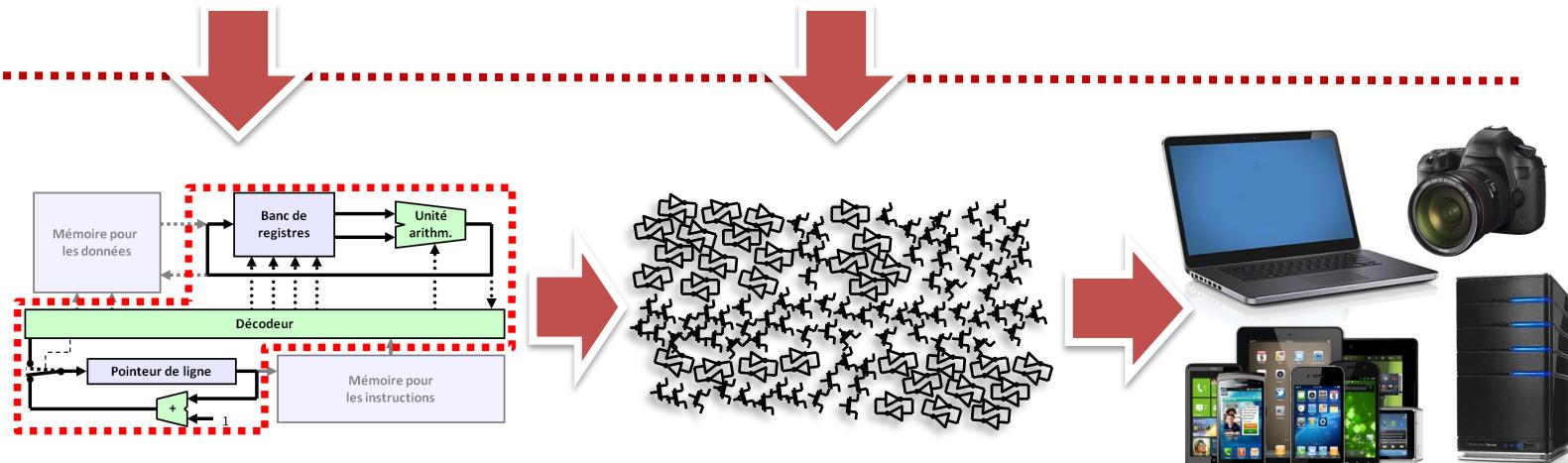
```
1: charge r3, 0
2: cont_neg r1, 6
3: somme r3, r3, r1
4: somme r1, r1, -1
5: continue 2
6: charge r2, r3
```

**somme des premiers
n entiers**

entrée : $r1$
sortie : $r2$

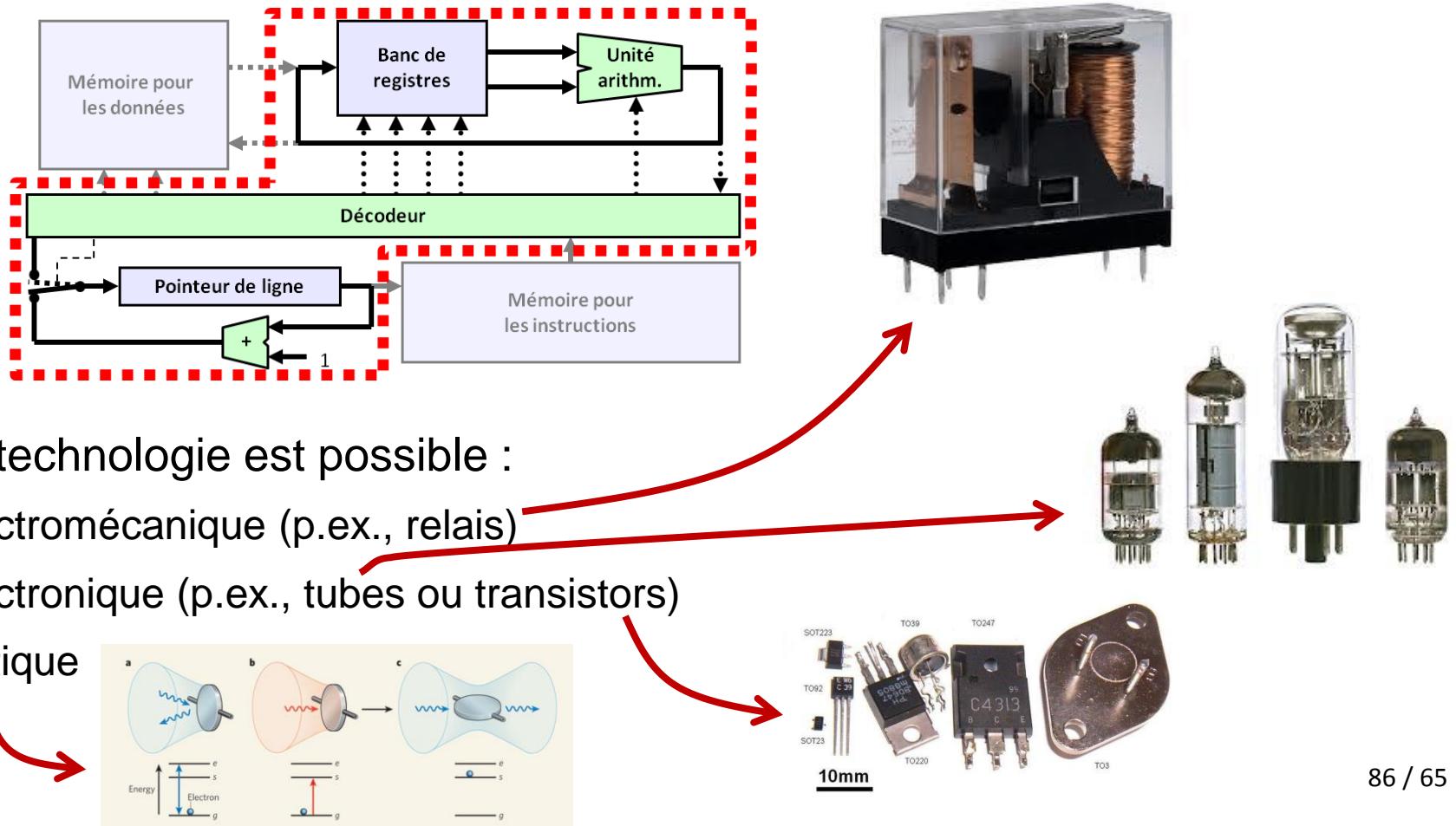
```
1: 0100010010111010100
2: 0101100011100000101
3: 11101011101010010010
4: 11101011101000010011
5: 0001100101010010101
6: 0100010110010111001
```

Logiciel
Matériel



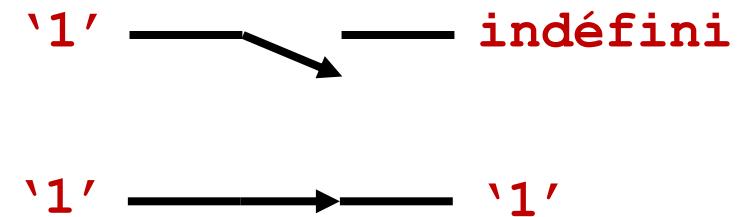
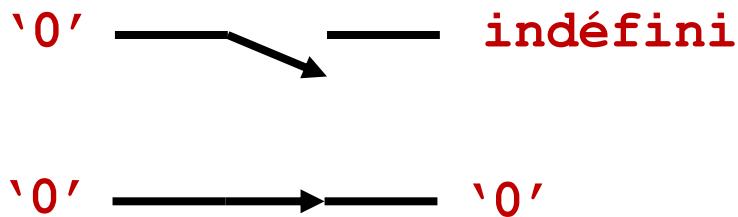
Jusque-là, pas de choix technologique...

- ▶ Notre machine est parfaitement abstraite et totalement indépendante d'un choix d'implémentation
- ▶ Même l'encodage binaire n'est nullement une nécessité



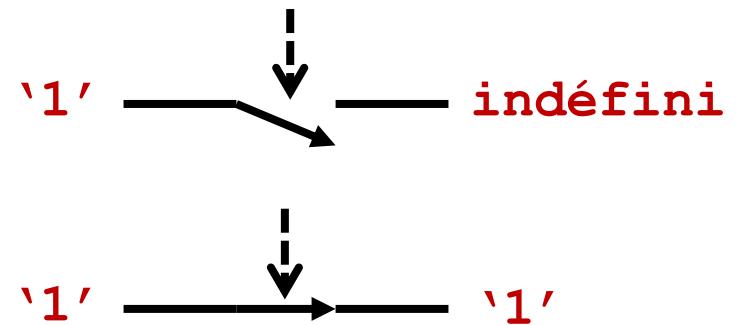
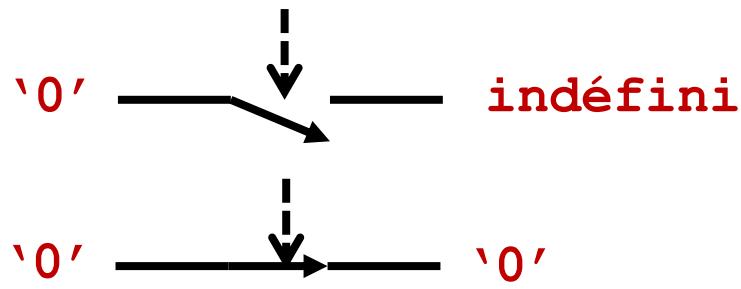
Un interrupteur

Ne propage rien s'il est ouvert

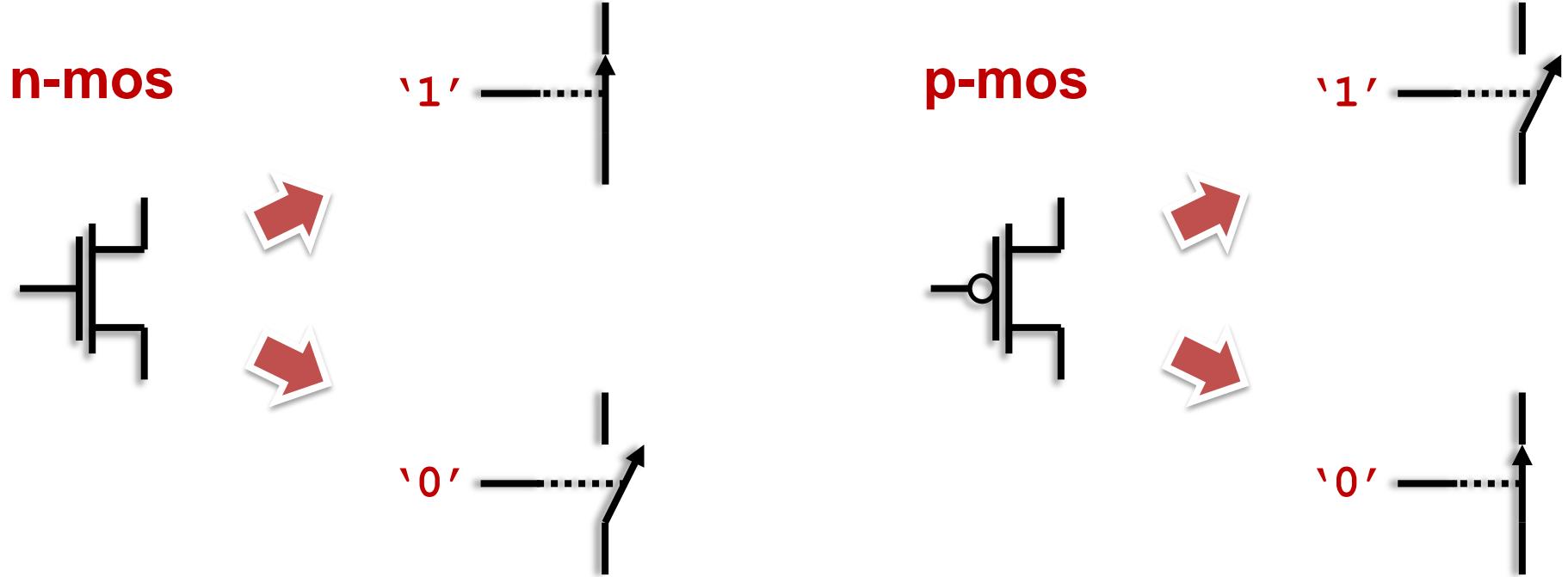


Propage son entrée s'il est fermé

Un transistor = un interrupteur contrôlé

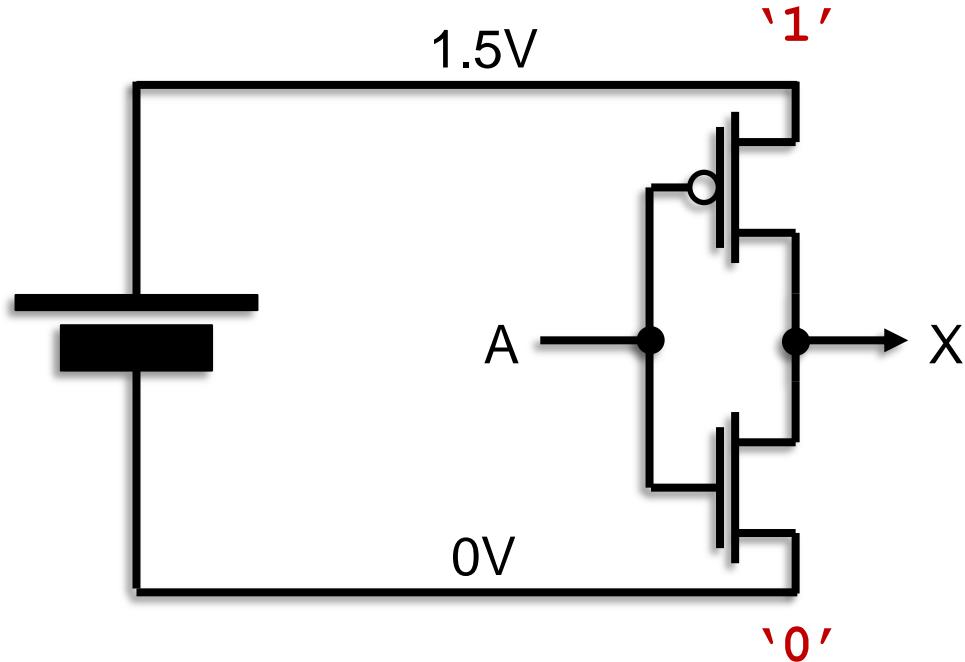


Les transistors, des interrupteurs contrôlés



- ▶ Ils ne coûtent presque rien : un transistor pour faire un processeur moderne coûte entre 10^{-5} et 10^{-4} centimes (CHF, USD, EUR...)

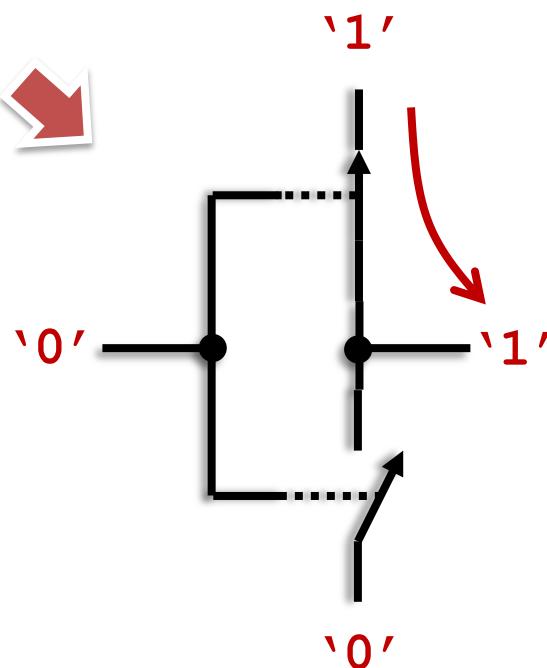
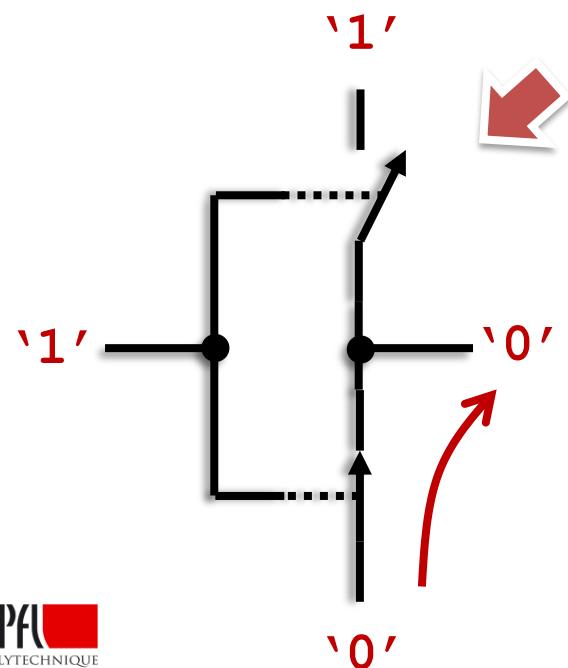
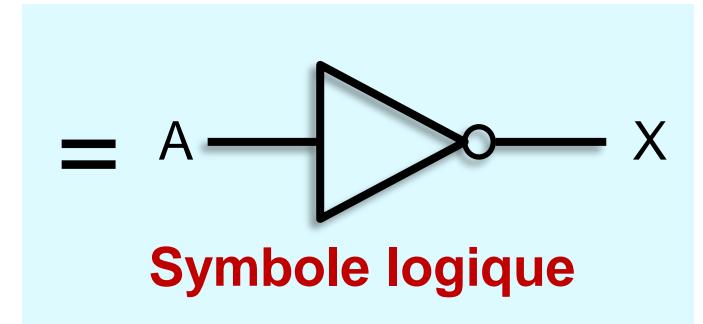
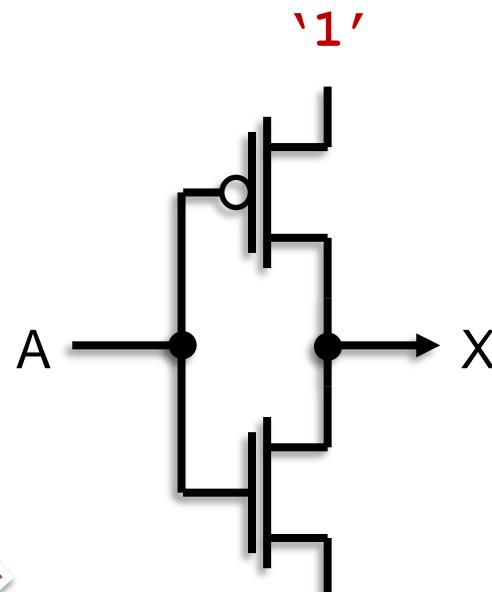
Un inverseur (en CMOS)



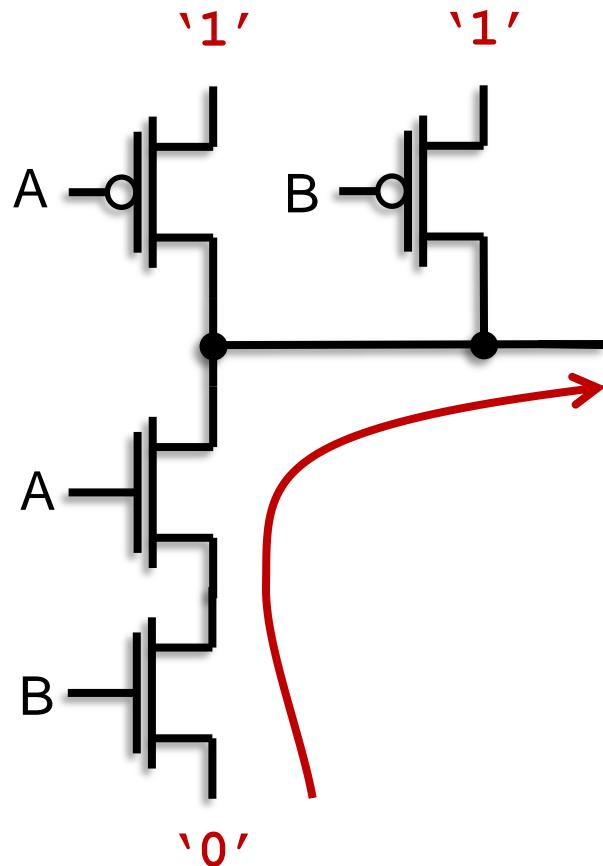
Un inverseur (en CMOS)

Table de la vérité

A	X
0	1
1	0



Un circuit « et » avec sortie inversée



A	B	A et B inversé
0	0	1
0	1	1
1	0	1
1	1	0

- ▶ La seule façon d'obtenir un '0' est de mettre deux '1' aux entrées A et B : la sortie est à '0' seulement si A **et** B sont à '1'

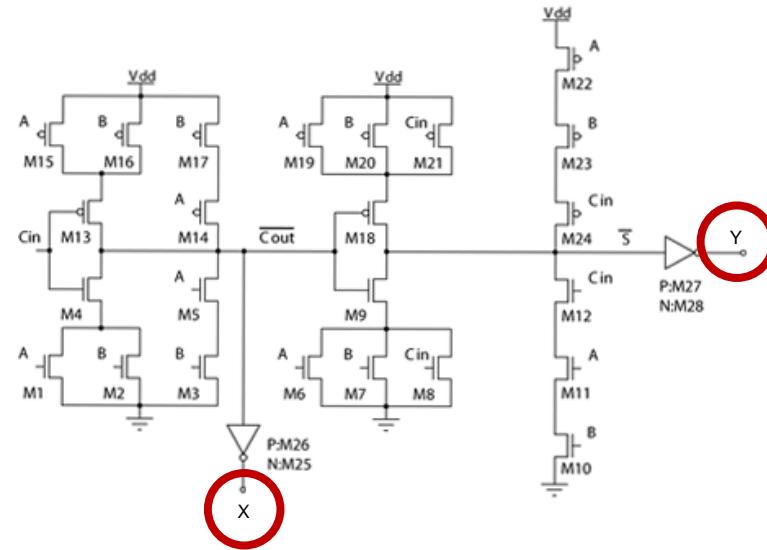
La table de vérité d'un circuit

- ▶ Résume la fonction d'un circuit
- ▶ Par exemple, pour un circuit avec 2 entrées et 1 sortie

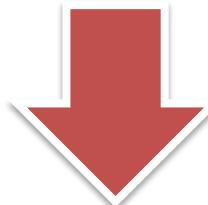
entrée 1	entrée 2	sortie
0	0	?
0	1	?
1	0	?
1	1	?

On peut réaliser n'importe quelle fonction !

A	B	C	XY
0	0	0	00
0	0	1	01
0	1	0	01
0	1	1	10
1	0	0	01
1	0	1	10
1	1	0	10
1	1	1	11

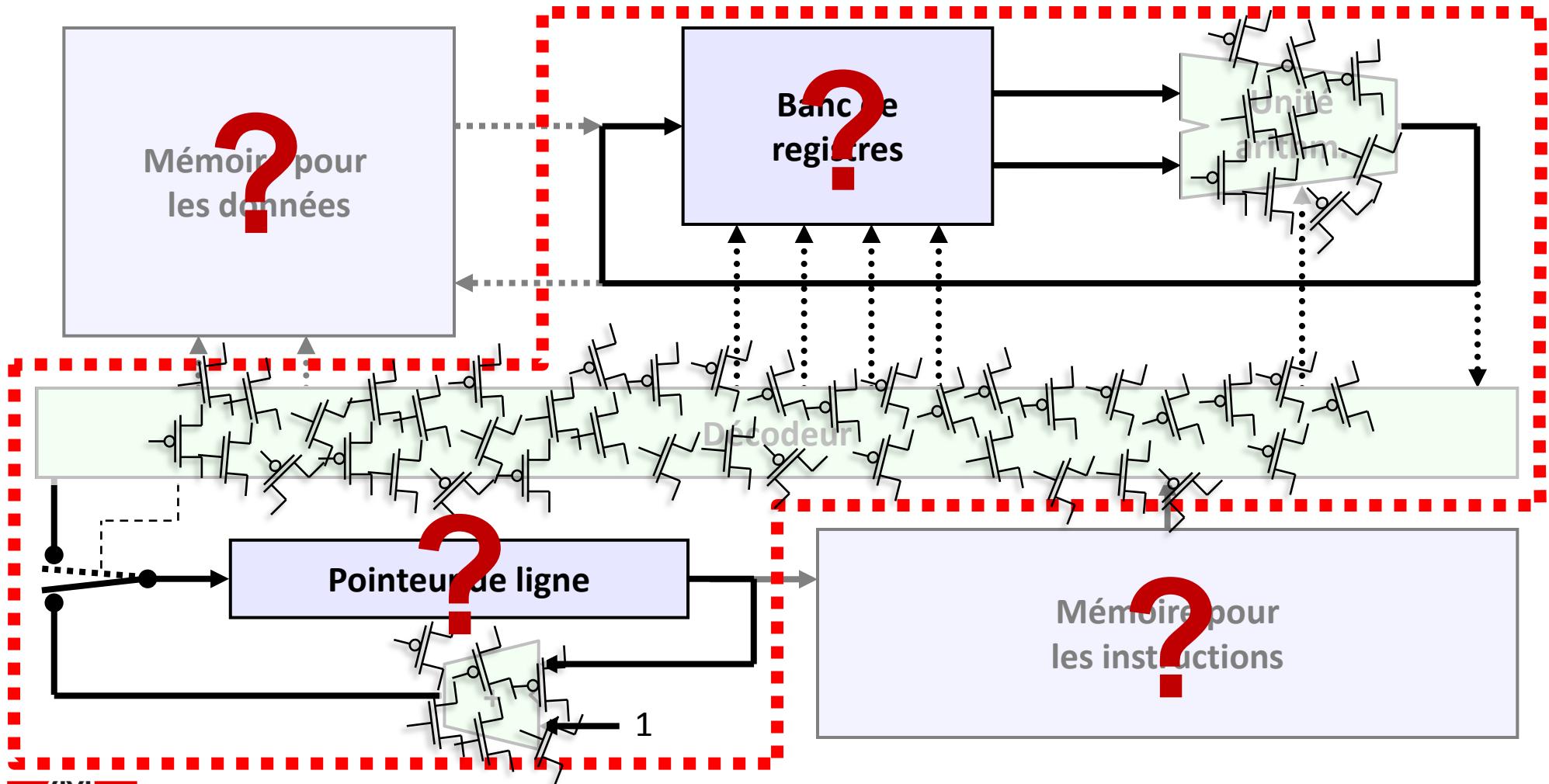


Les sorties XY sont la somme
(en représentation binaire)
des trois bits A, B et C à l'entrée



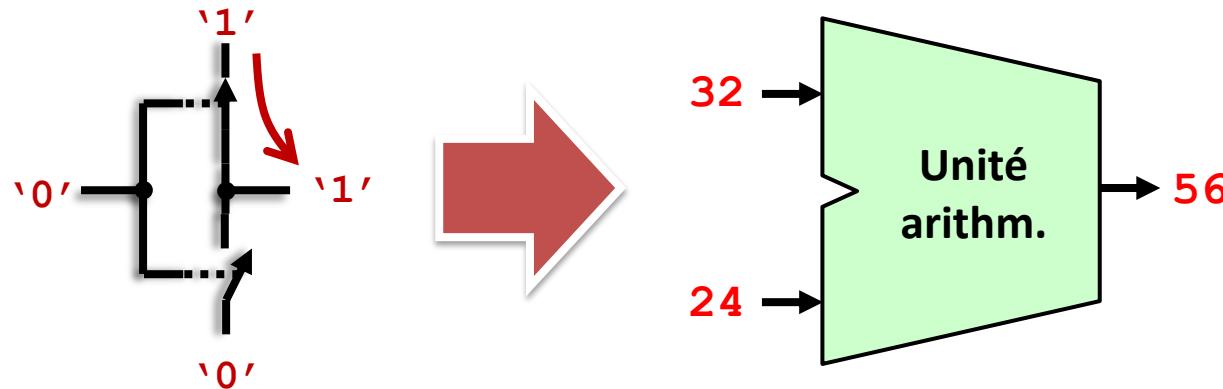
Addition !

Et notre processeur, donc ?

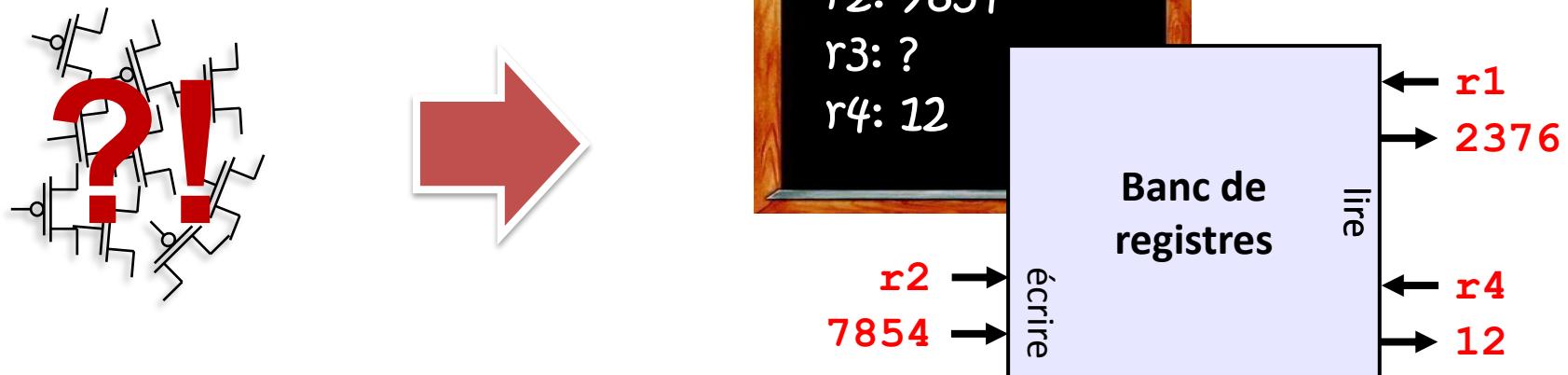


Peut-on aussi mémoriser l'information ?

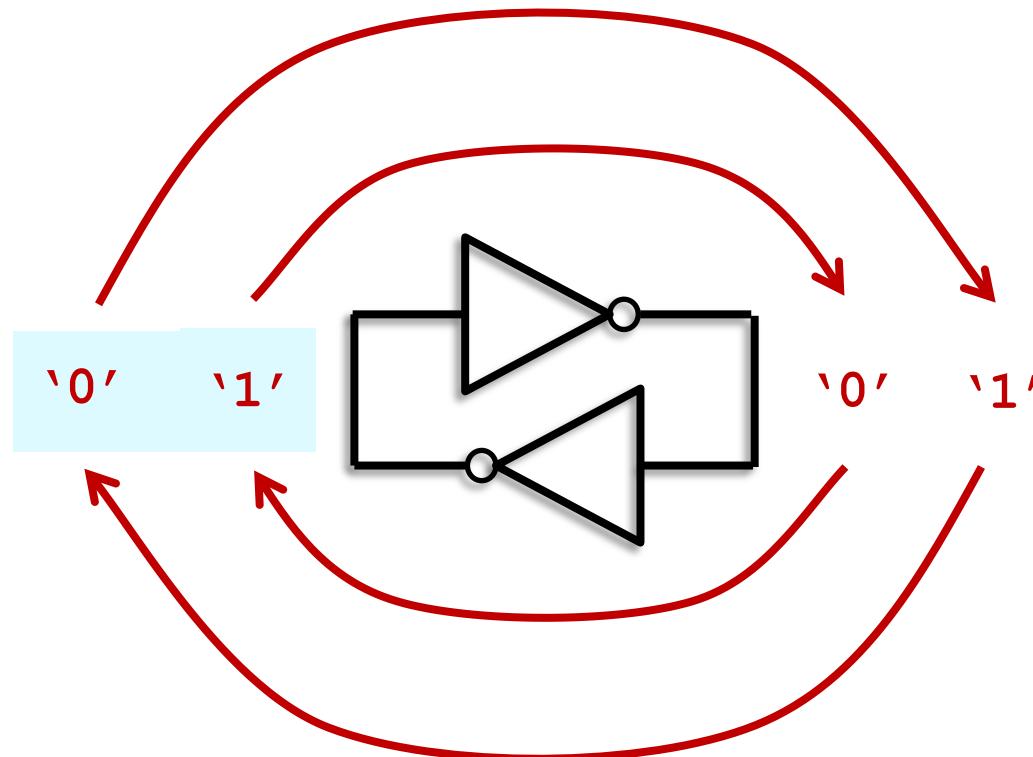
- ▶ Pour les calculs, tout va bien :



- ▶ Mais pour mémoriser ?!

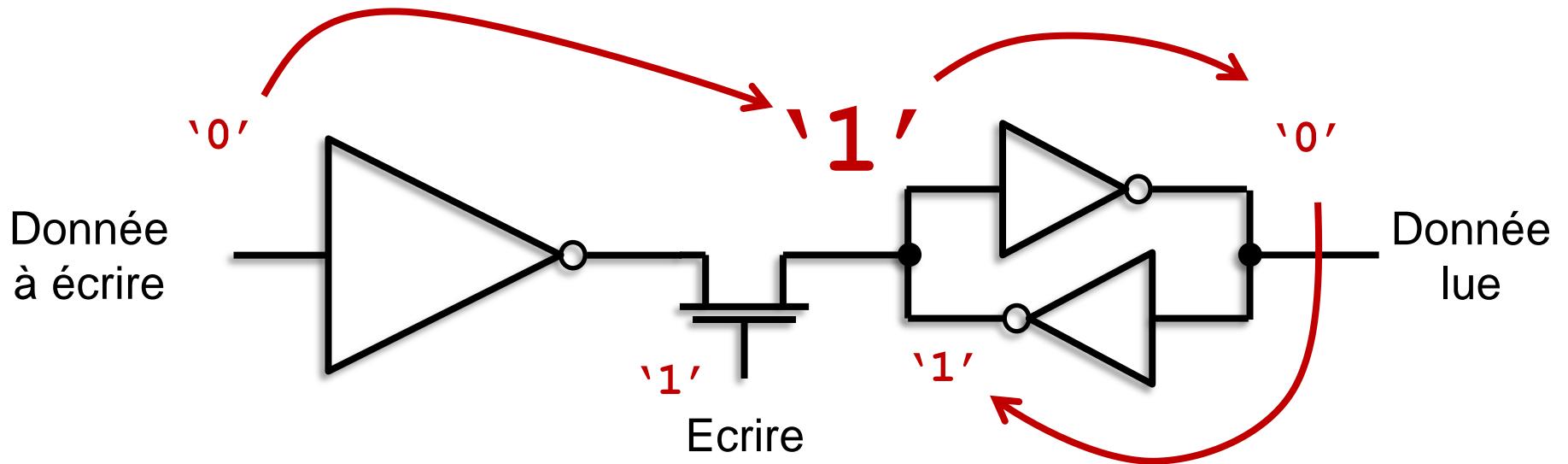


Un circuit assez particulier

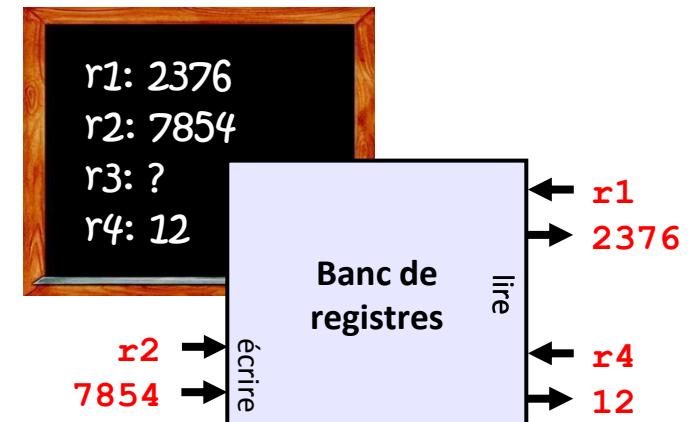


- Un circuit « bistable » c.à.d. qui peut être dans un parmi deux états parfaitement stables → **un élément mémoire !**

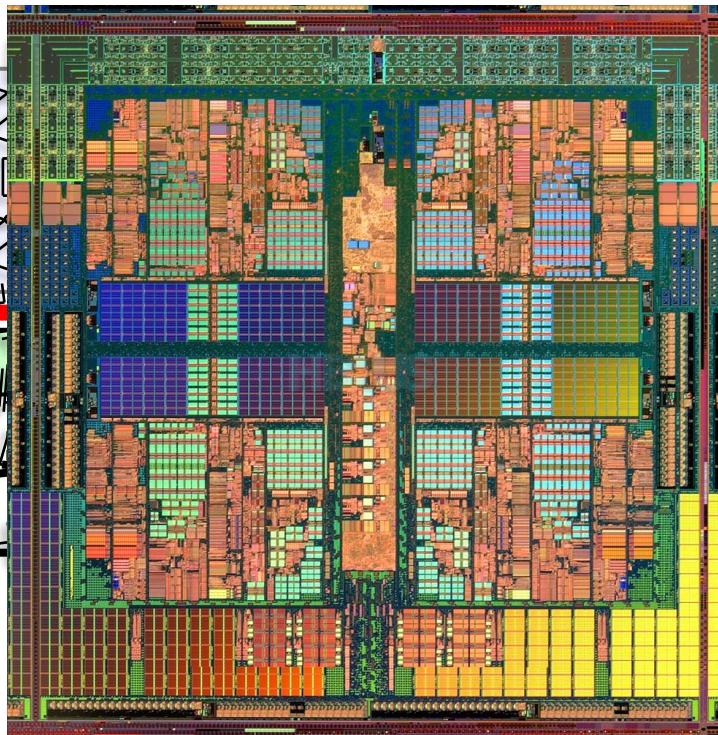
Comment écrire cette mémoire ?



- ▶ Avec quelques transistors on a un parfait circuit mémoire pour notre processeur

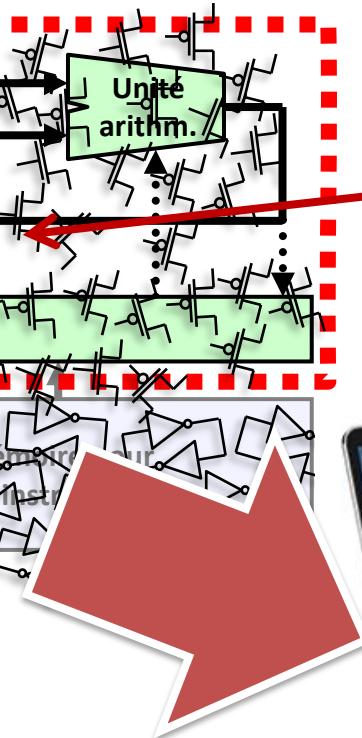


On a atteint notre but !



Architecture du
processeur

Circuit électronique



Circuit intégré VLSI
(aujourd'hui autour de
 $10^8\text{-}10^9$ transistors)



Des algorithmes aux ordinateurs

**somme des premiers
n entiers**

entrée : n
sortie : m

```

s ← 0
tant que  $n > 0$ 
     $s \leftarrow s + n$ 
     $n \leftarrow n - 1$ 
     $m \leftarrow s$ 

```

**somme des premiers
n entiers**

entrée : n
sortie : m

```

s ← 0
si  $n \leq 0$  continue ici
sinon
     $s \leftarrow s + n$ 
     $n \leftarrow n - 1$ 
    continue ici

```

$m \leftarrow s$

**somme des premiers
n entiers**

entrée : $r1$
sortie : $r2$

```

1: charge r3, 0
2: cont_neg r1, 6
3: somme r3, r3, r1
4: somme r1, r1, -1
5: continue 2
6: charge r2, r3

```

**somme des premiers
n entiers**

entrée : $r1$
sortie : $r2$

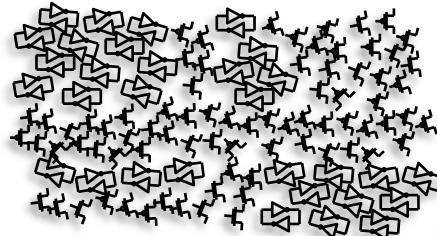
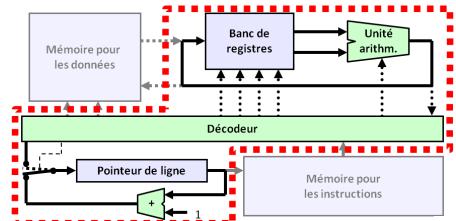
```

1: 0100010010111010100
2: 0101100011100000101
3: 1110101101010010010
4: 1110101101000010011
5: 0001100101010010101
6: 0100010110010111001

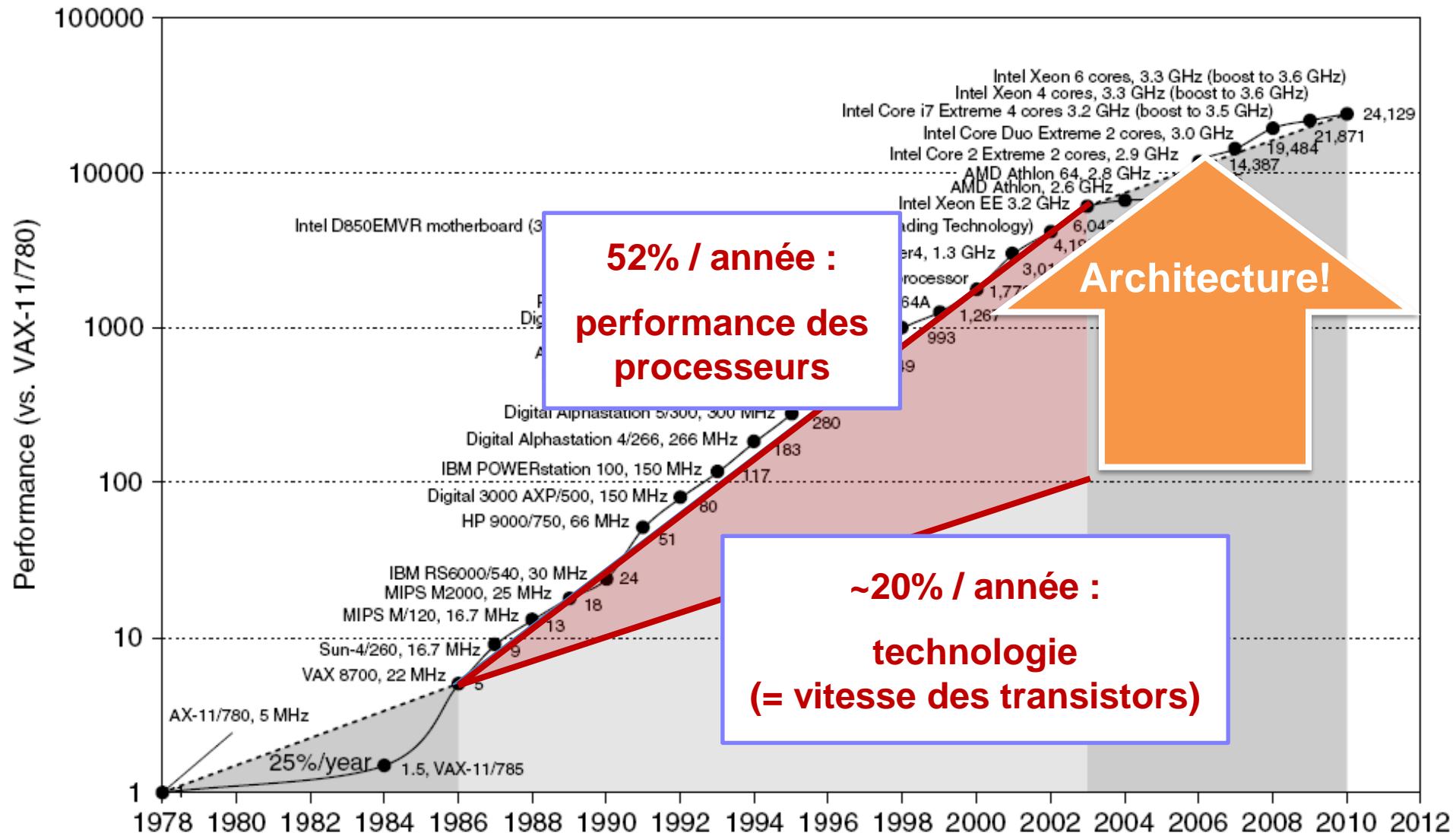
```

Logiciel

Matériel



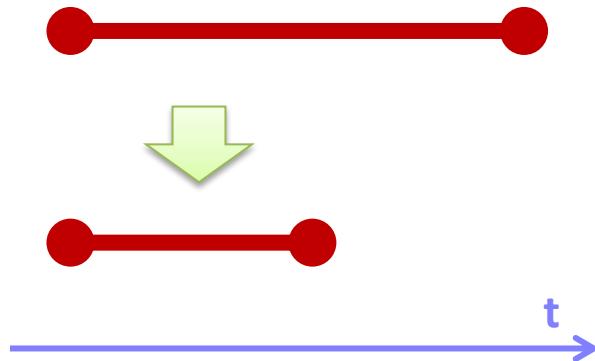
La croissance de la performance



Augmenter la performance ?

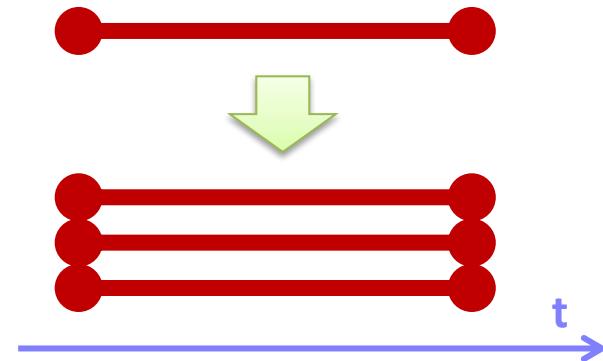
= Réduire le délai

temps d'attente pour obtenir un résultat



= Augmenter le débit

nombre de résultats dans l'unité de temps



Deux exemples simples d'amélioration de la performance :

1. Au niveau du circuit

Réduire le délai d'un additionneur

2. Au niveau de la structure du processeur

Augmenter le débit d'instructions

Faire des sommes est facile...

$$\begin{array}{r} A \quad 0111010101100011010 + \\ B \quad 1011100010111001011 = \\ \hline 111000111100011010 \\ 10010111000011100101 \end{array}$$

Retenue

Sommes élémentaires

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 10 = 1 \cdot 2^1 + 0 \cdot 2^0 = 2_{10}$$

Faire un circuit est au

A
B

Somme

$$0 + 0 + 0 = 0$$

$$0 + 0 + 1 = 1$$

$$0 + 1 + 0 = 1$$

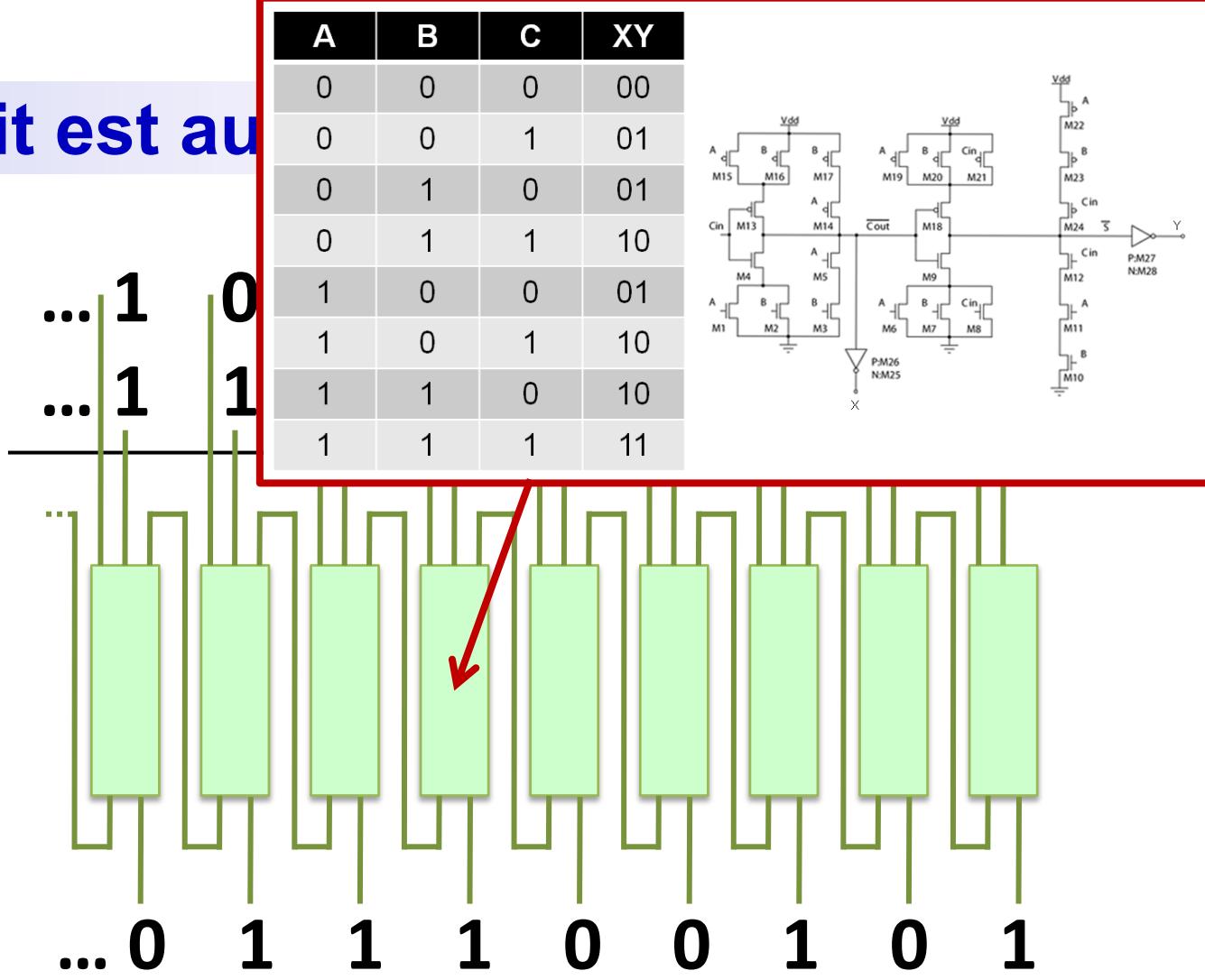
$$0 + 1 + 1 = 10$$

$$1 + 0 + 0 = 1$$

$$1 + 0 + 1 = 10$$

$$1 + 1 + 0 = 10$$

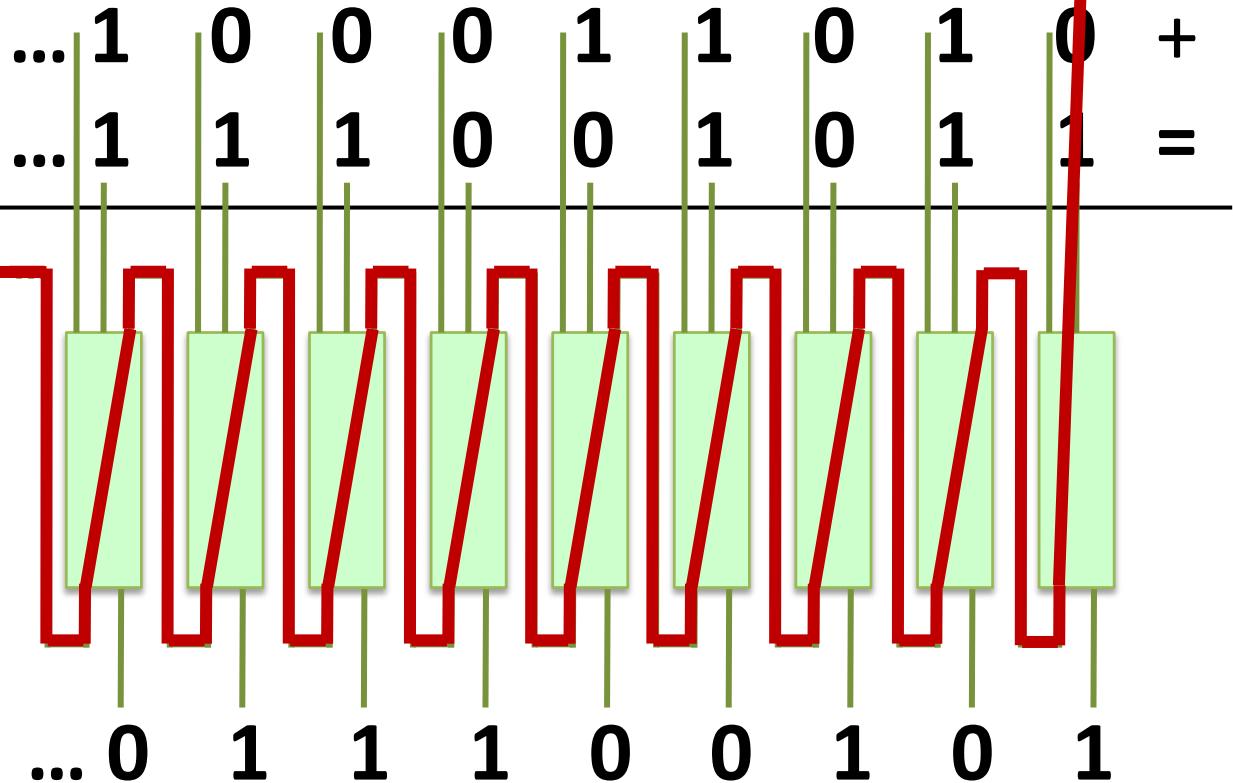
$$1 + 1 + 1 = 11$$



Mais ce circuit est lent !

A

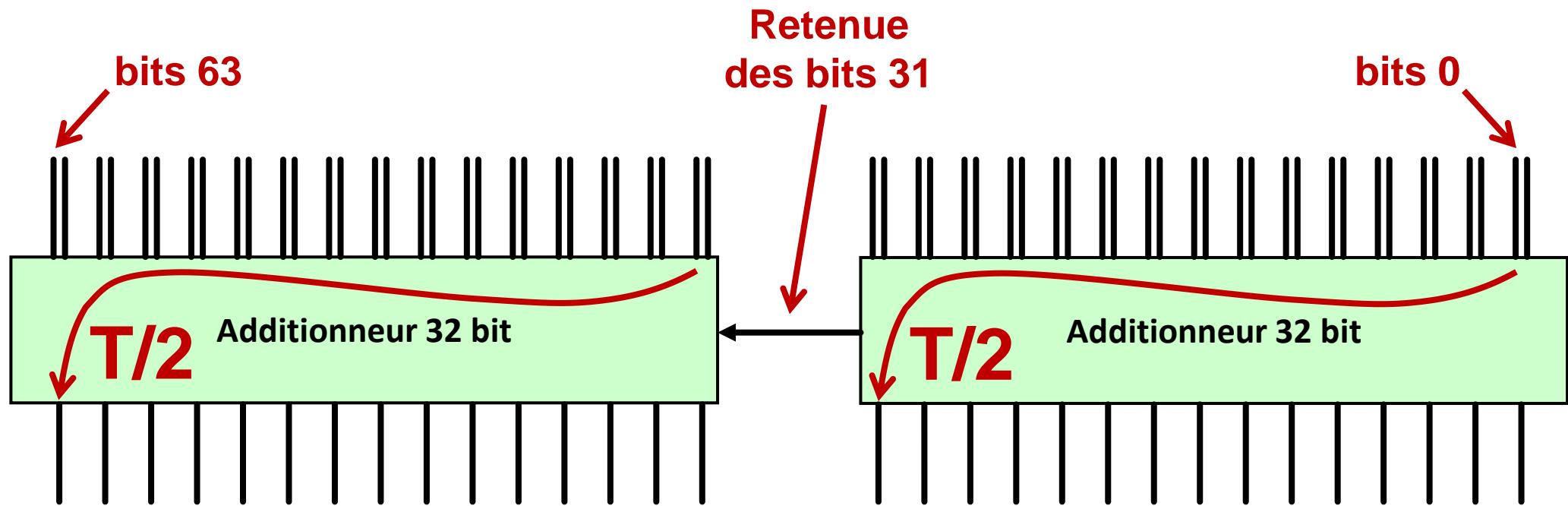
B



- ▶ La propagation de la retenue est un aspect **fondamental** de la somme !

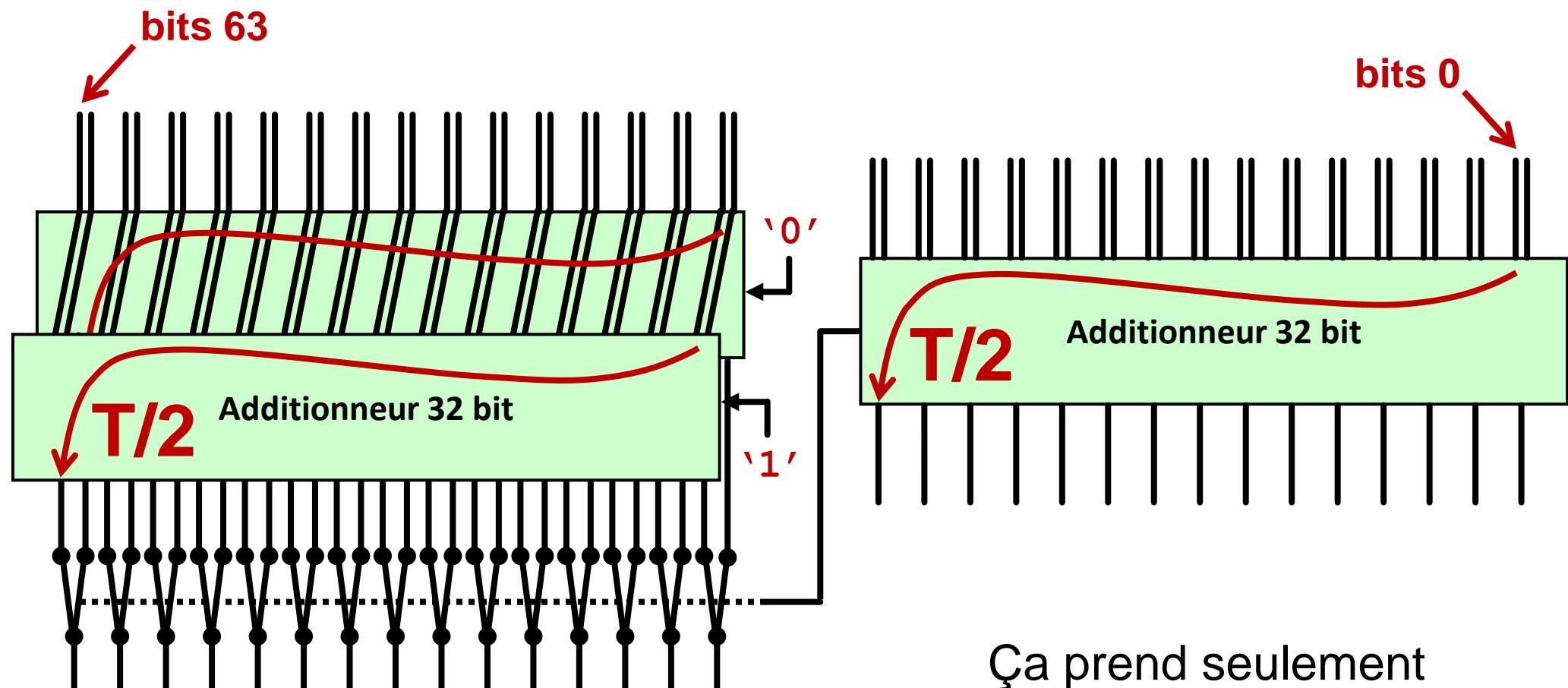
- ▶ A la base, le **délai** d'un additionneur est donc **proportionnel au nombre de bits à additionner**

Peut-on faire mieux ?



On n'a **rien** gagné...

Peut-on faire mieux ?



Ça prend seulement
la moitié du temps !

Le génie informatique (1)

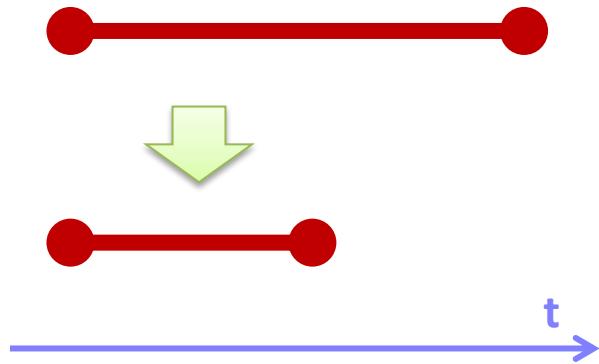
- ▶ On peut profondément changer la performance du circuit sans en changer la fonctionnalité
- ▶ On peut investir plus de transistors et plus d'énergie pour obtenir des circuits très rapides
- ▶ On peut ralentir les circuits pour épargner de l'énergie

Ceci est un exemple de **synthèse logique** qui est une des branches de l'ingénierie informatique (**Computer Engineering**)

Augmenter la performance ?

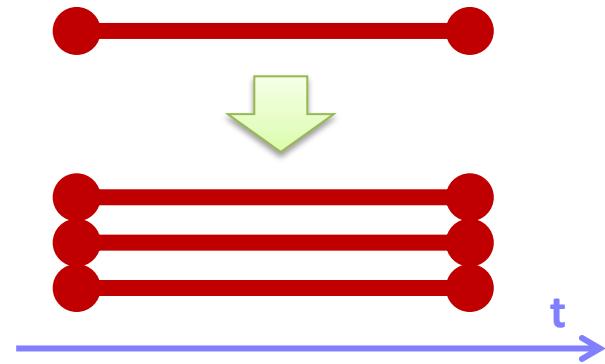
= Réduire le délai

temps d'attente pour obtenir un résultat



= Augmenter le débit

nombre de résultats dans l'unité de temps



Deux exemples simples d'amélioration de la performance :

1. Au niveau du circuit

Réduire le délai d'un additionneur

2. Au niveau de la structure du processeur

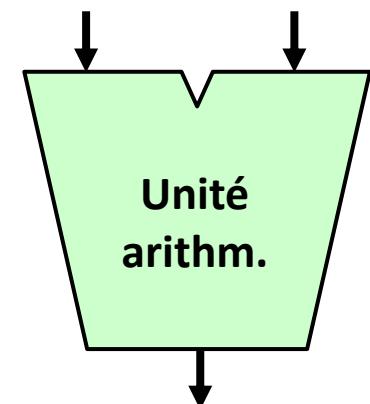
Augmenter le débit d'instructions

Notre processeur

```
103: charge      r1, 0
104: charge      r2, -21
105: somme       r3, r7, r4
106: multiplie   r2, r5, r9
107: soustrais   r8, r7, r9
108: charge      r9, r4
109: somme       r3, r2, r1
110: soustrais   r5, r3, r4
111: charge      r2, r3
112: somme       r1, r2, -1
113: somme       r8, r1, -1
114: divise      r4, r1, r7
115: charge      r2, r4
```

On exécute approximativement
une instruction par cycle

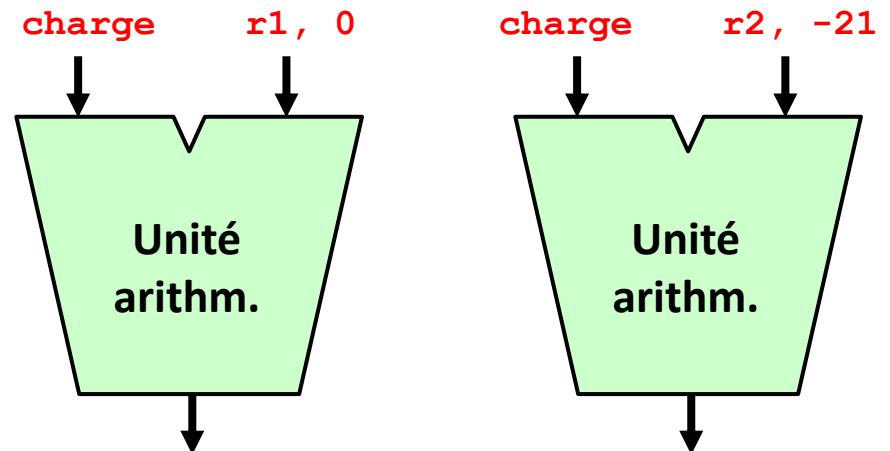
Comment faire mieux ?



Doubler le débit de notre processeur

```
103: charge      r1, 0
104: charge      r2, -21
105: somme       r3, r7, r4
106: multiplie   r2, r5, r9
107: soustrais   r8, r7, r9
108: charge      r9, r4
109: somme       r3, r2, r1
110: soustrais   r5, r3, r4
111: charge      r2, r3
112: somme       r1, r2, -1
113: somme       r8, r1, -1
114: divise      r4, r1, r7
115: charge      r2, r4
```

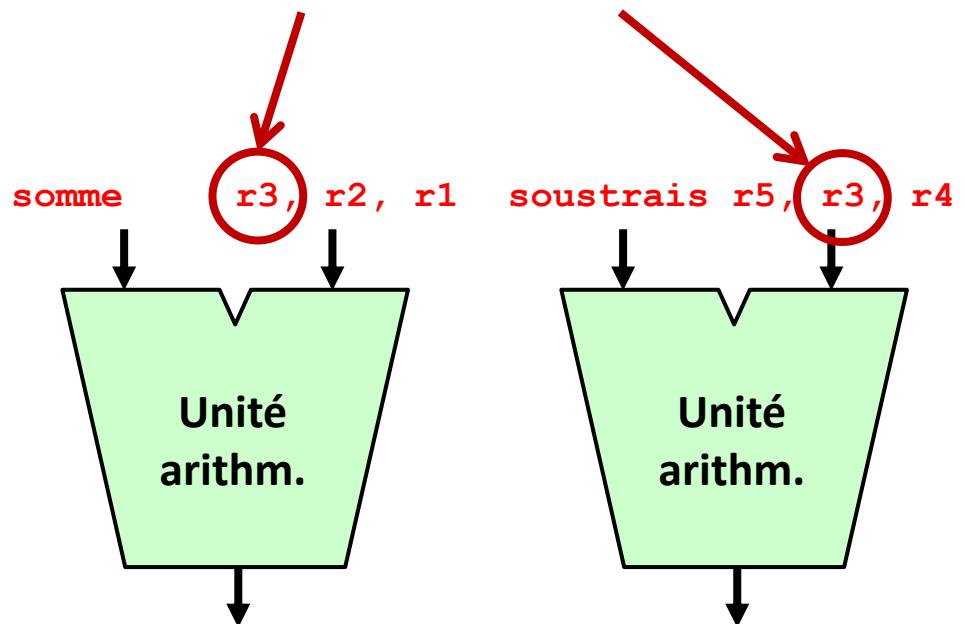
On exécute maintenant approximativement
deux instructions par cycle !



Doubler le débit de notre processeur

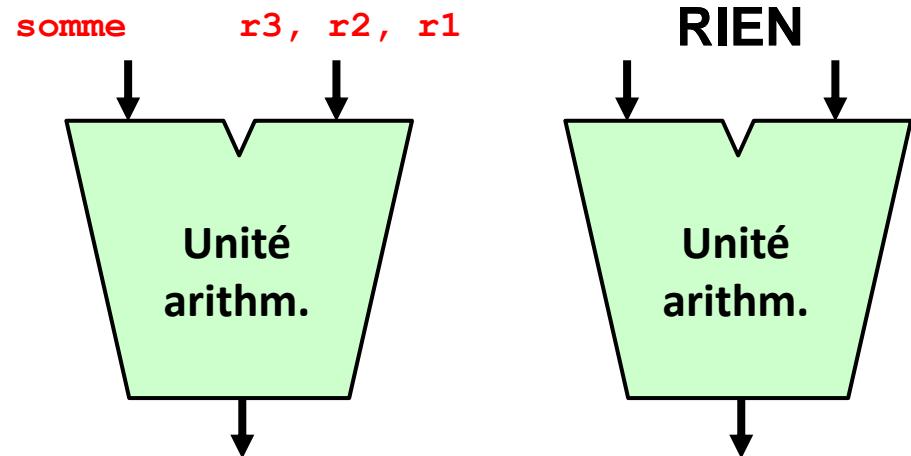
```
103: charge    r1, 0
104: charge    r2, -21
105: somme     r3, r7, r4
106: multiplie r2, r5, r9
107: soustrais r8, r7, r9
108: charge    r9, r4
109: somme     r3, r2, r1
110: soustrais r5, r3, r4
111: charge    r2, r3
112: somme     r1, r2, -1
113: somme     r8, r1, -1
114: divise    r4, r1, r7
115: charge    r2, r4
```

Le problème est que la deuxième instruction a besoin d'une valeur calculée par la première !
Si on ne fait pas attention,
le résultat sera faux !



Doubler le débit de notre processeur

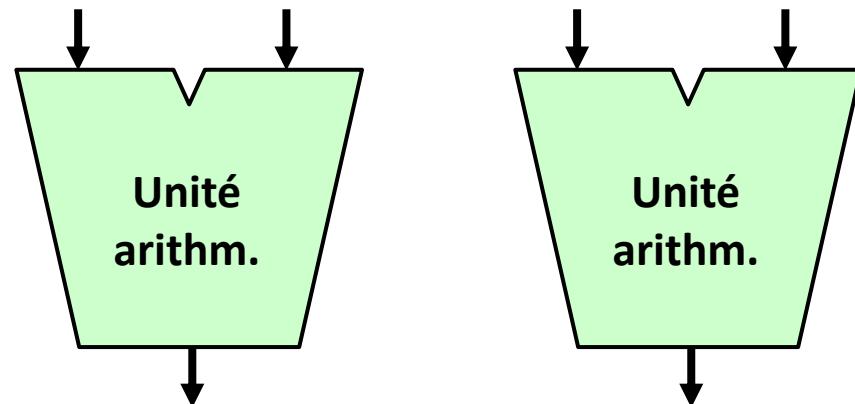
```
103: charge      r1, 0
104: charge      r2, -21
105: somme       r3, r7, r4
106: multiplie   r2, r5, r9
107: soustrais   r8, r7, r9
108: charge      r9, r4
109: somme       r3, r2, r1
110: soustrais   r5, r3, r4
111: charge      r2, r3
112: somme       r1, r2, -1
113: somme       r8, r1, -1
114: divise      r4, r1, r7
115: charge      r2, r4
```



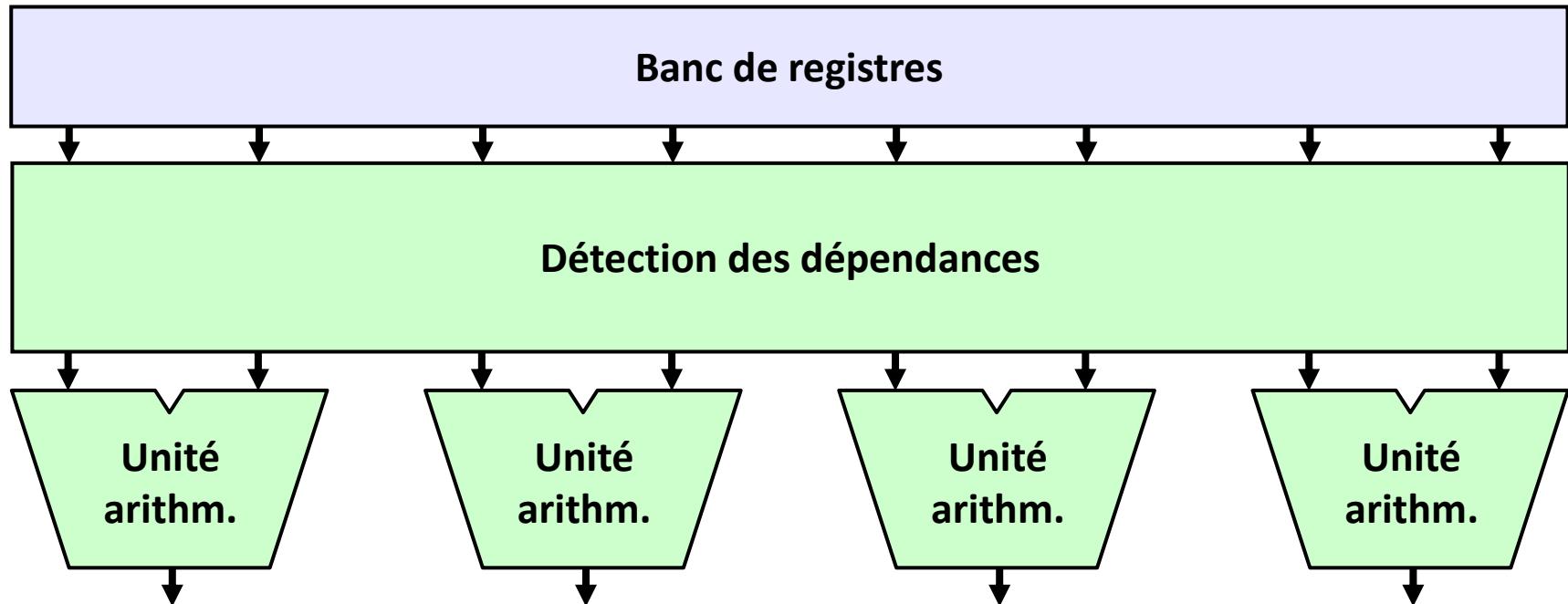
~~Doubler le débit de notre processeur~~

```
103: charge    r1, 0
104: charge    r2, -21
105: somme     r3, r7, r4
106: multiplie r2, r5, r9
107: soustrais r8, r7, r9
108: charge    r9, r4
109: somme     r3, r2, r1
110: soustrais r5, r3, r4
111: charge    r2, r3
112: somme     r1, r2, -1
113: somme     r8, r1, -1
114: divise    r4, r1, r7
115: charge    r2, r4
```

On exécute maintenant approximativement entre **une et deux instructions** par cycle et le résultat est correct !



Un processeur “superscalaire”



- ▶ Tous les processeurs modernes pour les ordinateurs portables et les serveurs sont de ce type
- ▶ De plus, ils réordonnancent les instructions et en exécutent avant que ce soit sûr qu'elles doivent être exécutées (p.ex. après une instruction comme **cont_neg**)

Le génie informatique

- ▶ On peut modifier la structure du système pour exécuter les programmes plus rapidement
- ▶ On peut ajouter des ressources aux processeurs pour les rendre beaucoup plus rapides
- ▶ On peut utiliser des processeurs très élémentaires pour les rendre économiques et peu gourmands en énergie

On vient de voir un exemple d'**architecture des ordinateurs**, qui est une autre des branches du génie informatique (ou **Computer Engineering**)