

# Information, Calcul et Communication

## Module 3 : Systèmes

# Information, Calcul et Communication

## Ordinateur à programme enregistré

Paolo lenne

# La première question de cette leçon

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

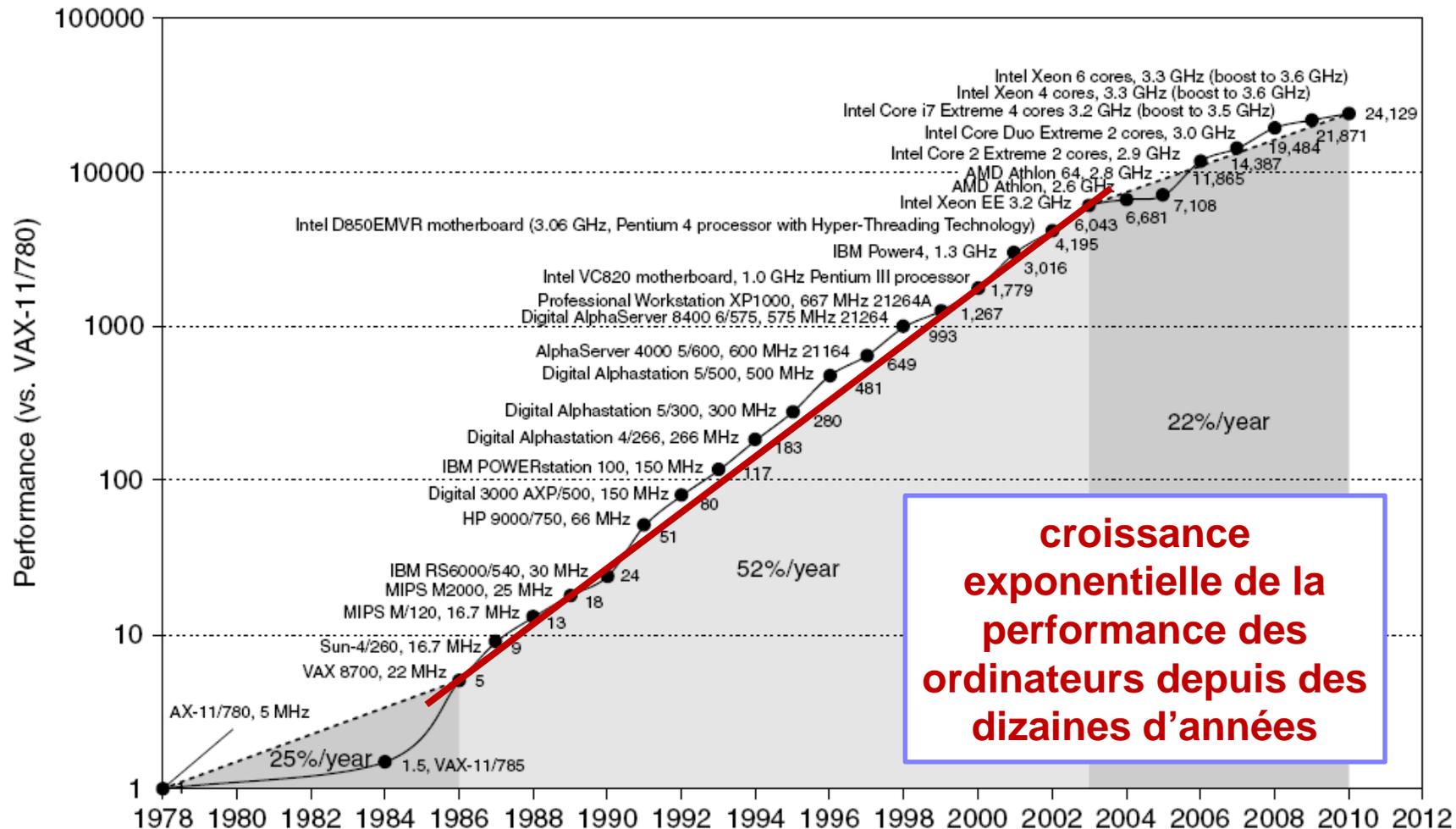
👉 **Algorithme :**

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



# Le deuxième question de cette leçon

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



**croissance exponentielle de la performance des ordinateurs depuis des dizaines d'années**

# 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: 01000100101111010100  
2: 01011000111100000101  
3: 1110101101010010010

...et les rendre  
compréhensibles  
à une machine

Logiciel

Matériel



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



...pour la réaliser  
avec des transistors



# Comment écrit-on un algorithme ?

**somme des premiers  
 $n$  entiers**

entrée :  $n$  *entier positif*  
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
  - Au lieu d'écrire  $n$  on écrit **r1**
  - Au lieu d'écrire  $m$  on écrit **r2**
  - Au lieu d'écrire  $s$  on écrit **r3**

# Continuons...

**somme des premiers  
 $n$  entiers**

entrée :  $n$   
sortie :  $m$

$s \leftarrow 0$

**tant que**  $n > 0$

$s \leftarrow s + n$

$n \leftarrow n - 1$

$m \leftarrow s$

**On a besoin d'assigner  
des nouvelles valeurs à  
ces symboles suite à des  
opérations**

on va écrire cela  
d'une façon très régulière

P.ex.,

« **somme  $r1, r1, r3$**  »  
pour signifier  $r1 \leftarrow r1 + r3$

# Les opérations ou « instructions »

- ▶ On définit un **nombre limité** d'opérations; p.ex.
  - **charge** pour l'assignation (= affectation)
  - **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

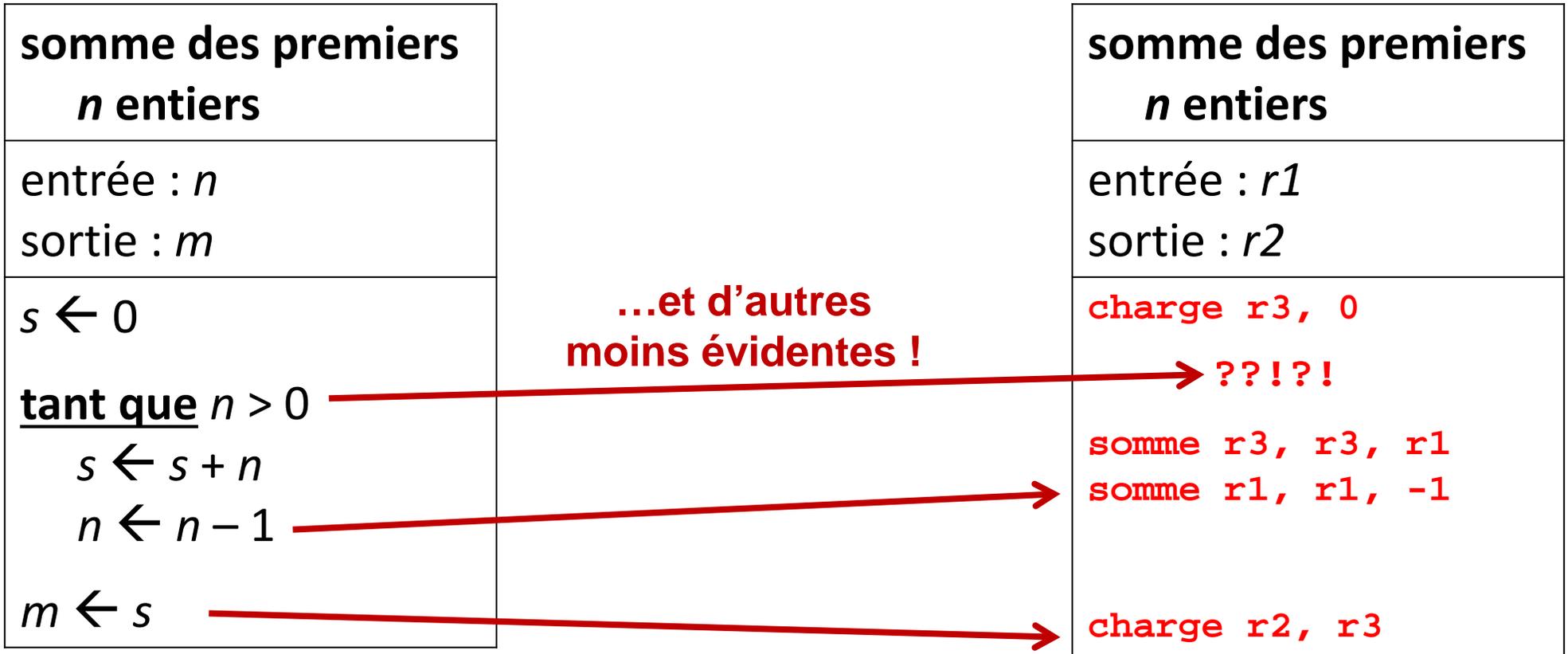
- Au lieu d'écrire  $s \leftarrow s + n$  on écrit **somme r3, r3, r1**
- Au lieu d'écrire  $s \leftarrow 0$  on écrit **charge r3, 0**
- Au lieu d'écrire  $s \leftarrow c (a + b)$  on écrit

**somme r5, r6, r7**  
**multiplie r5, r5, r8**

Diagram illustrating the mapping of variables to registers in assembly instructions:

- Variable **a** points to register **r6** in the **somme** instruction.
- Variable **b** points to register **r7** in the **somme** instruction.
- Variable **c** points to register **r8** in the **multiplie** instruction.
- Variable **s** points to register **r5** in both the **somme** and **multiplie** instructions.

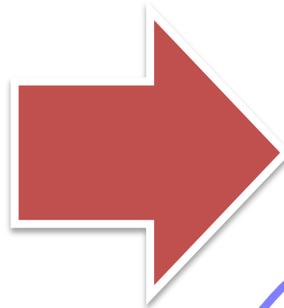
# Un langage un peu moins vague ?



**Quelques transformations  
sont très simples...**

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

<b>somme des premiers <math>n</math> entiers</b>
entrée : $n$ <i>entier positif</i> sortie : $m$
$s \leftarrow 0$ <u><math>\text{tant que } n &gt; 0</math></u> $s \leftarrow s + n$ $n \leftarrow n - 1$ $m \leftarrow s$

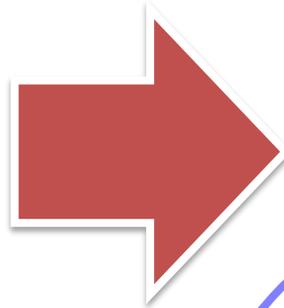


<b>somme des premiers <math>n</math> entiers</b>
entrée : $n$ <i>entier positif</i> sortie : $m$
$s \leftarrow 0$ <u><math>\text{tant que } n &gt; 0</math></u> $s \leftarrow s + n$ $n \leftarrow n - 1$ $m \leftarrow s$

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

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

<b>somme des premiers <math>n</math> entiers</b>
entrée : $n$ <i>entier positif</i> sortie : $m$
$s \leftarrow 0$ <u>tant que</u> $n > 0$ $s \leftarrow s + n$ $n \leftarrow n - 1$ $m \leftarrow s$



<b>somme des premiers <math>n</math> entiers</b>
entrée : $n$ <i>entier positif</i> sortie : $m$
$s \leftarrow 0$ <u>si</u> $n = 0$ <u>continue ici</u>  $s \leftarrow s + n$ $n \leftarrow n - 1$  <u>continue ici</u>  $m \leftarrow s$

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

# Quelques nouvelles instructions

	<b>somme des premiers <math>n</math> entiers</b>
	entrée : $n$ <i>entier positif</i> sortie : $m$
1:	$s \leftarrow 0$
2:	<u>si <math>n = 0</math> continue ici</u>
3:	<u>sinon</u>
4:	$s \leftarrow s + n$
5:	$n \leftarrow n - 1$
6:	<u>continue ici</u>
7:	<del><math>m \leftarrow s</math></del>

**On doit spécifier les destinations de ces « continue ici »**

au lieu des flèches, on peut utiliser les numéros de ligne: **1, 2, 3,...**

# Quelques nouvelles instructions

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

on introduit une  
nouvelle  
action/instruction  
« **continue 2** »  
pour signifier  
continue ici ↪

1:

somme des premiers  
 $n$  entiers

entrée :  $n$  entier positif  
sortie :  $m$

$s \leftarrow 0$

2:

si  $n = 0$  continue ici

3:

sinon

4:

$s \leftarrow s + n$

5:

$n \leftarrow n - 1$

6:

continue ici

7:

$m \leftarrow s$

# Quelques nouvelles instructions

	<b>somme des premiers <math>n</math> entiers</b>
	entrée : $n$ <i>entier positif</i> sortie : $m$
1:	$s \leftarrow 0$
2:	<u>si</u> $n = 0$ <u>continue ici</u>
3:	<u>sinon</u>
4:	$s \leftarrow s + n$
5:	$n \leftarrow n - 1$
6:	<u>continue ici</u>
7:	$m \leftarrow s$

On doit pouvoir réagir à des conditions

on introduit des actions/instructions conditionnelles comme « **cont\_nul r1, 7** » pour signifier **si r1 est nul continue ici**

# 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** est égal à **r2**
  - **cont\_nul r1, 12** pour le saut vers la ligne 12 si **r1** est nul
  - **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\_nul r1, 6**

# Un langage un peu moins vague !

somme des premiers  
 $n$  entiers

entrée :  $n$   
sortie :  $m$

$s \leftarrow 0$

si  $n = 0$  continue ici

sinon

$s \leftarrow s + n$

$n \leftarrow n - 1$

continue ici

$m \leftarrow s$

Maintenant,  
tout marche !

somme des premiers  
 $n$  entiers

entrée :  $r1$   
sortie :  $r2$

1: charge  $r3, 0$

2: cont\_nul  $r1, 6$

3: somme  $r3, r3, r1$

4: somme  $r1, r1, -1$

5: continue 2

6: charge  $r2, r3$

Et celui-là, où est-il passé ?!

# Petit résumé...

- ▶ On écrit nos programmes comme des séquences d'actions appelées « **instructions** »
- ▶ La plupart de ces actions indiquent quelles valeurs donner à des variables à la suite d'opérations (p.ex., mathématiques comme **somme**)
- ▶ On utilise seulement un jeu restreint d'opérations préalablement définies (p.ex., on pourrait ne pas avoir de soustraction si on a l'opération d'addition **somme** et l'opération pour trouver l'opposé **oppose**)
- ▶ On utilise seulement des variables comme **r1**, **r2**, **r3**, etc.—on les appelle « **registres** »
- ▶ Certaines actions indiquent un *branchement* (p.ex., **continue**, si ce branchement est inconditionnel, ou **cont\_nul**, si il est à suivre seulement dans certains cas)—on les appelle « **instructions de saut** »

# Est-ce que cela nous rapproche du but ?

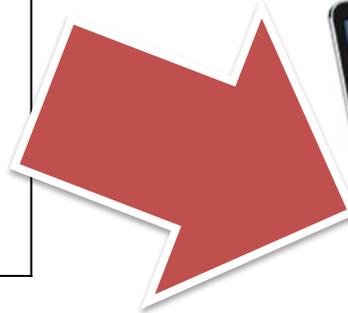
somme des premiers  
 $n$  entiers

entrée :  $r1$  entier positif

sortie :  $r2$

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

Langage assembleur



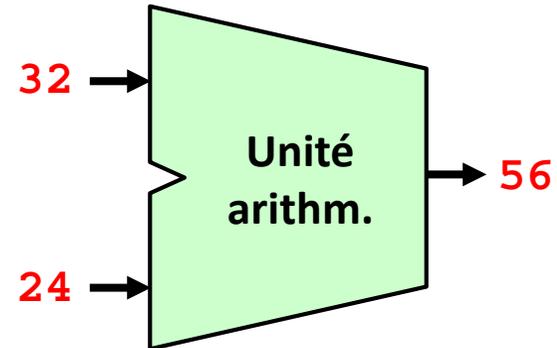
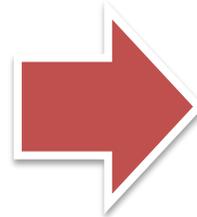
# Essayons de créer une telle machine...



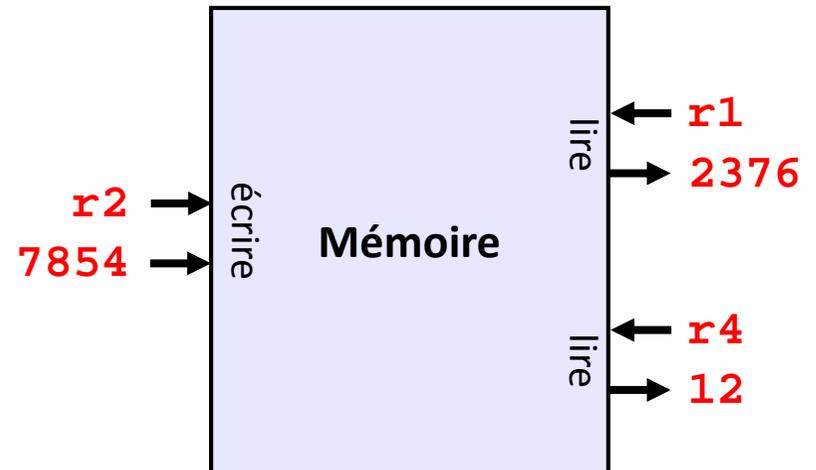
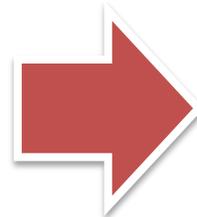
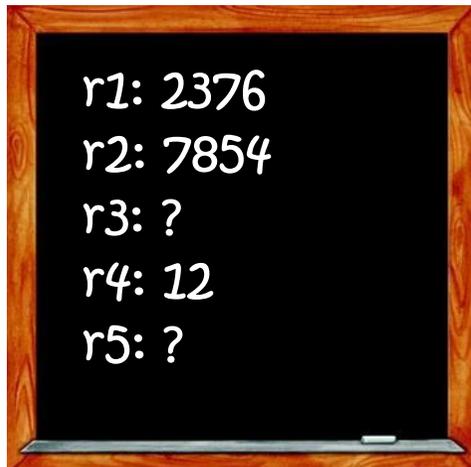
# De quoi a-t-on besoin ?

- Pour les calculs (somme, etc.) on a besoin d'un objet capable de les effectuer

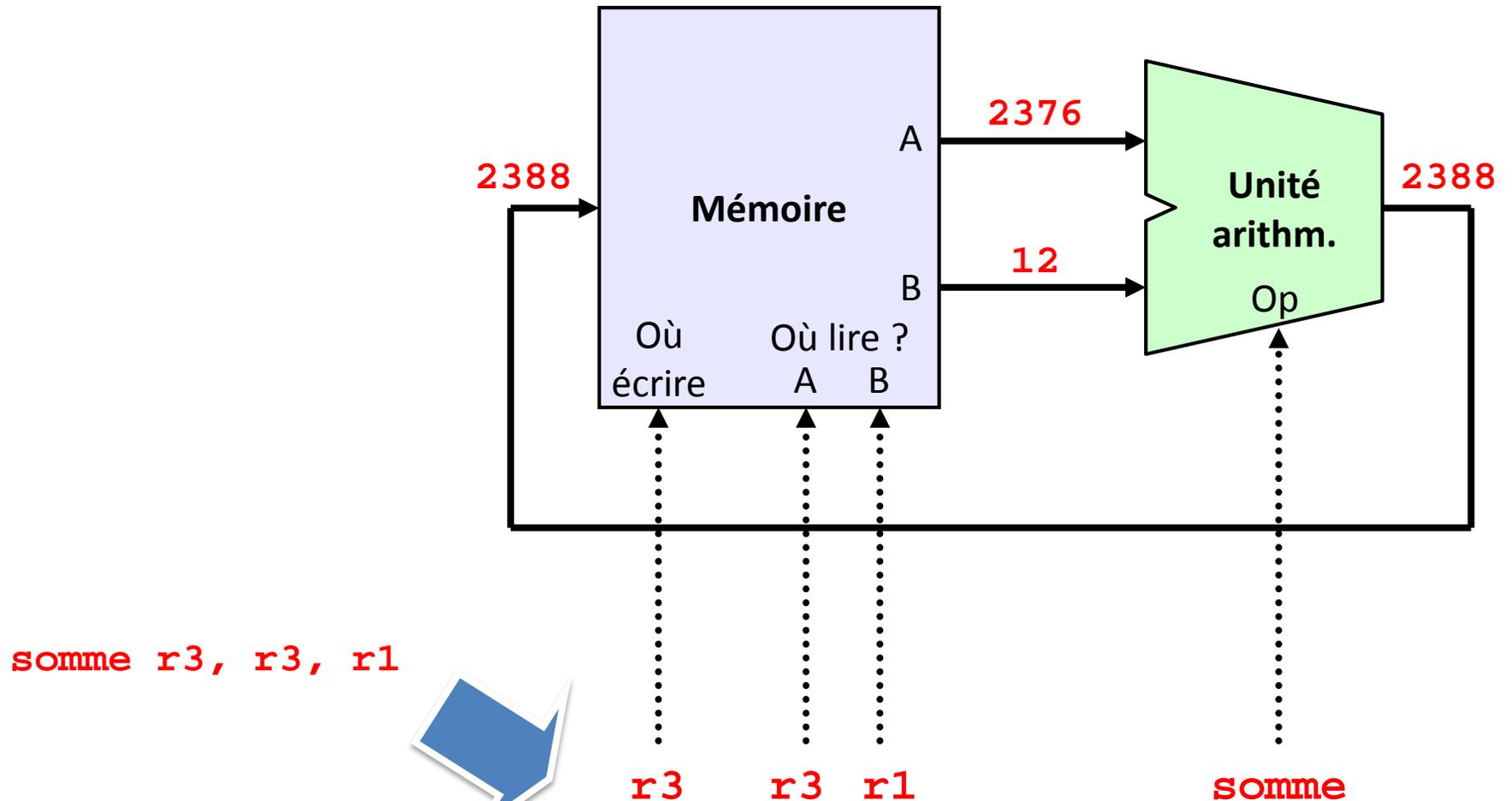
$$\begin{array}{r} 32 + \\ 24 = \\ \hline 56 \end{array}$$



- Les calculs ont besoin d'opérandes et il faut les mémoriser quelque part



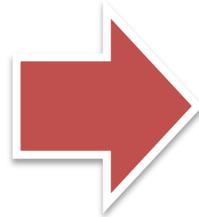
# Le circuit pour les données



# De quoi a-t-on encore besoin ?

- ▶ On doit mettre notre programme en assembleur quelque part

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

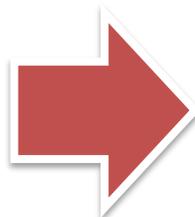


3:



- ▶ Il faut savoir où on en est

```
prochaine instruction
4:
```

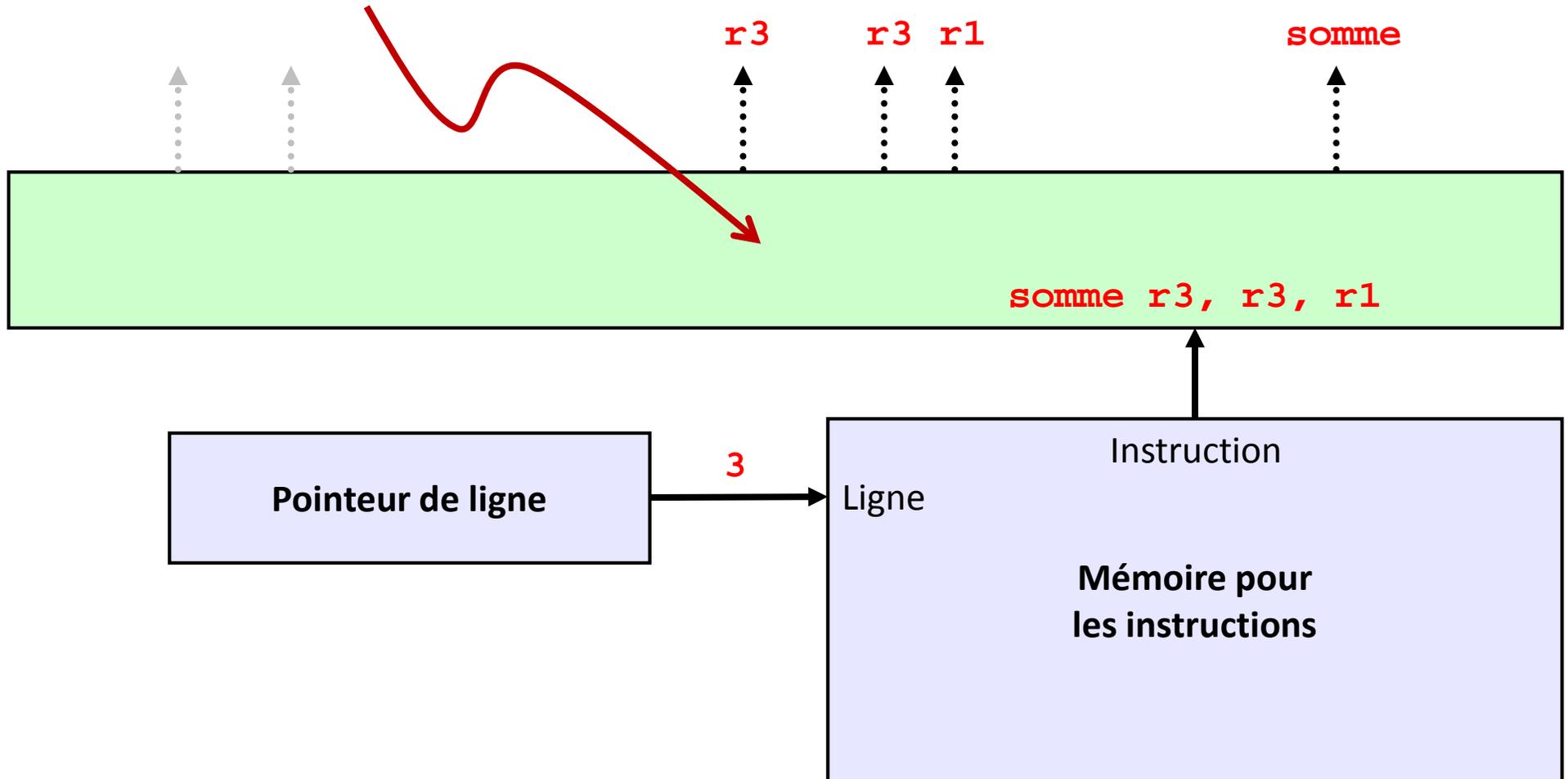


4:



# Un première partie pour contrôler le tout...

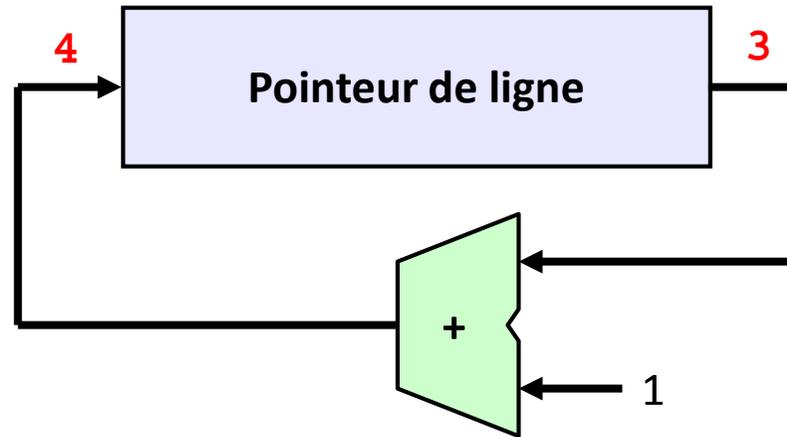
Un circuit assez simple qui répartit les éléments qui constituent une instruction



# De quoi a-t-on encore besoin ?

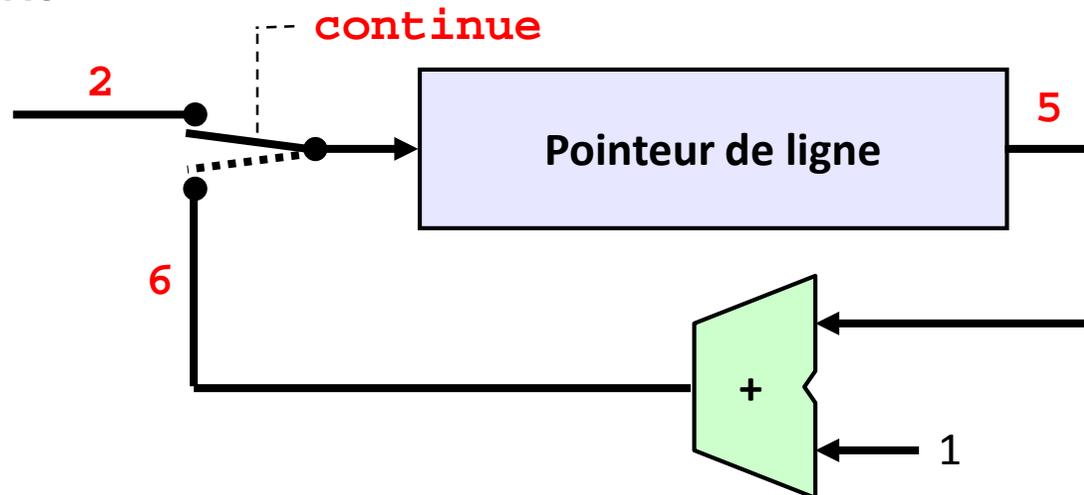
- ▶ Normalement on veut passer d'une ligne à la suivante

3: somme  $r3, r3, r1$

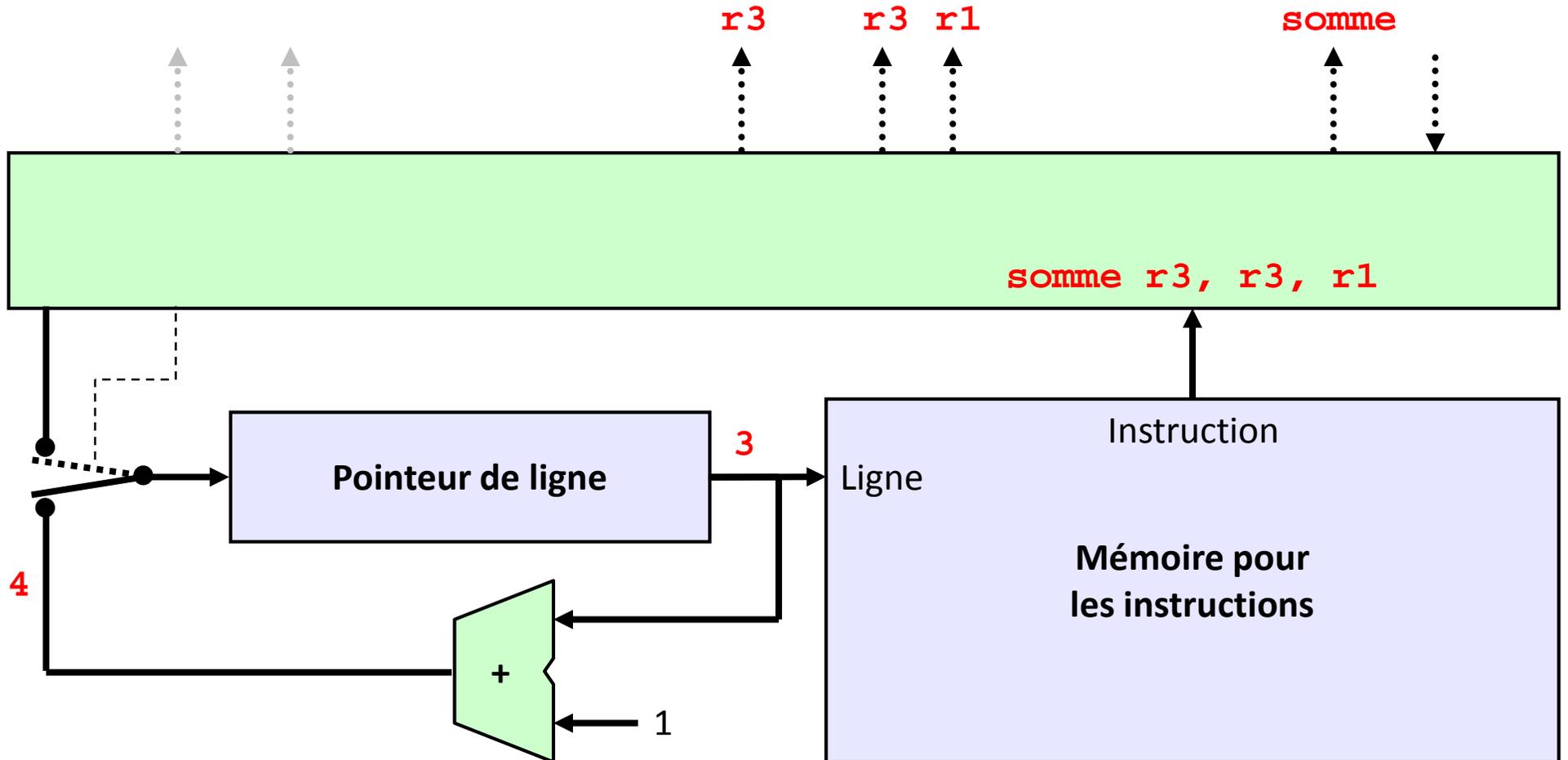


- ▶ Si on a une instruction **continue**, on veut imposer une autre ligne à lire prochainement

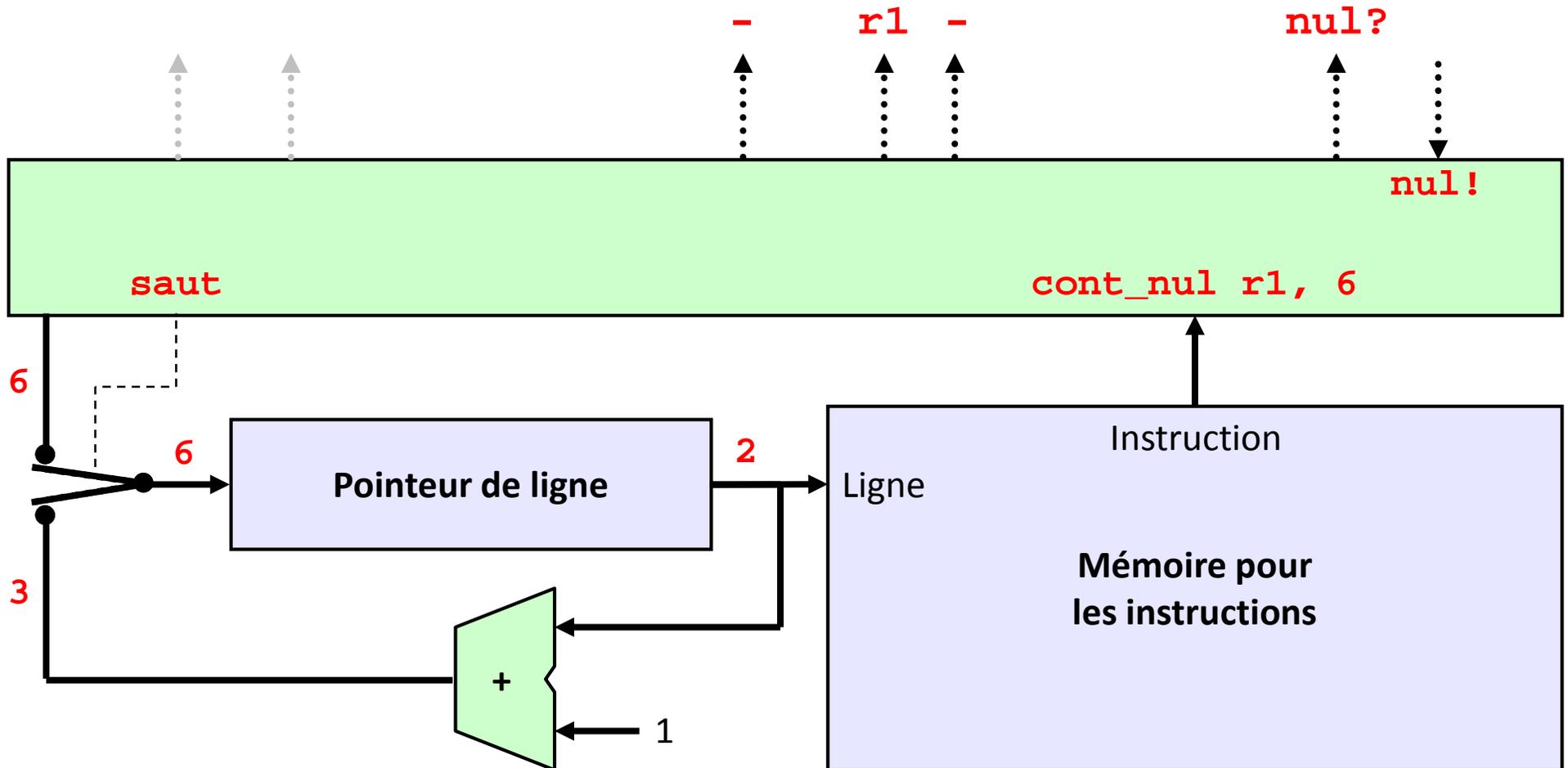
5: continue 2



# Le circuit pour le contrôle

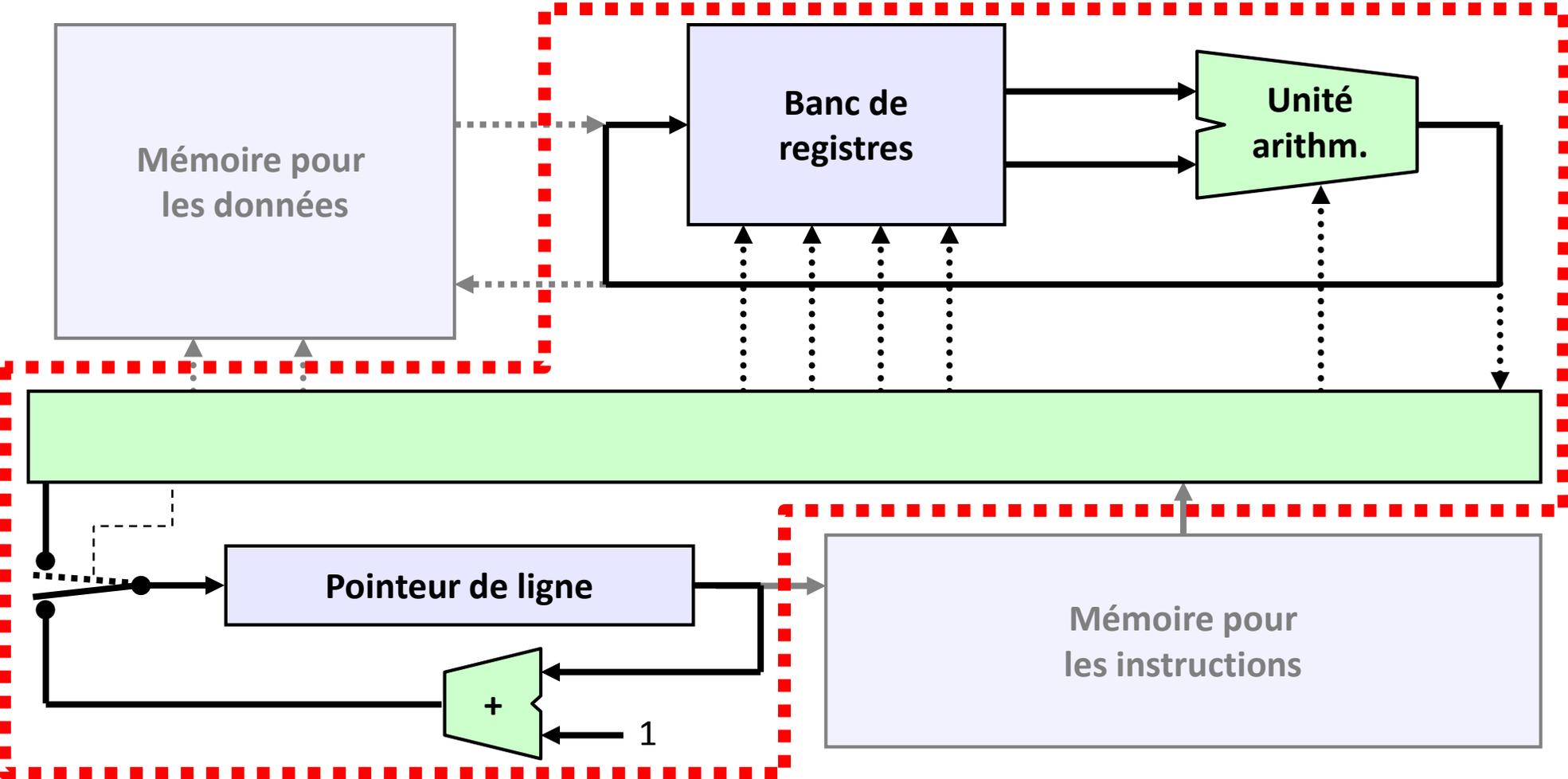


# En cas de saut ?



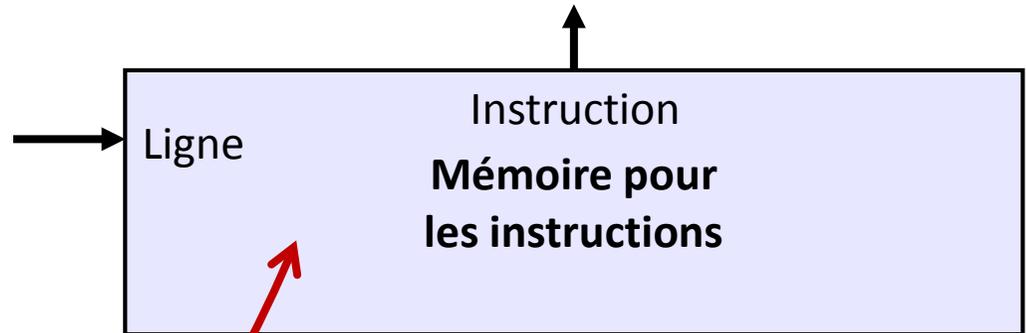
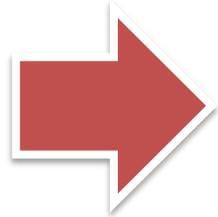


# Un processeur !

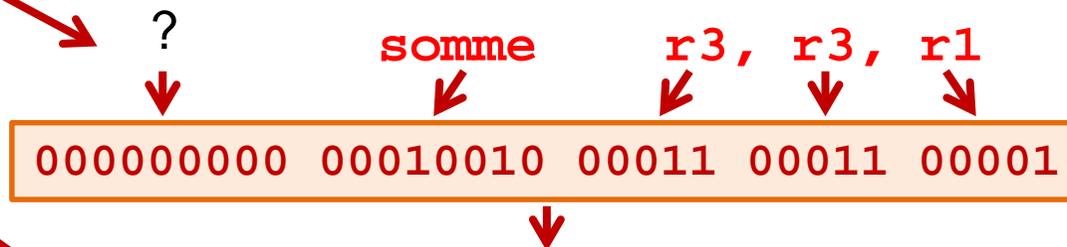


# Comment encoder les instructions ?

```
1: charge    r3, 0
2: cont_nul  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

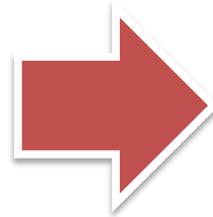


La valeur **592,993** représente l'instruction **somme r3, r3, r1**

# Encoder les instructions

somme des premiers <i>n</i> entiers	
entrée : <i>r1</i> sortie : <i>r2</i>	
1:	<code>charge r3, 0</code>
2:	<code>cont_nul r1, 6</code>
3:	<code>somme r3, r3, r1</code>
4:	<code>somme r1, r1, -1</code>
5:	<code>continue 2</code>
6:	<code>charge r2, r3</code>

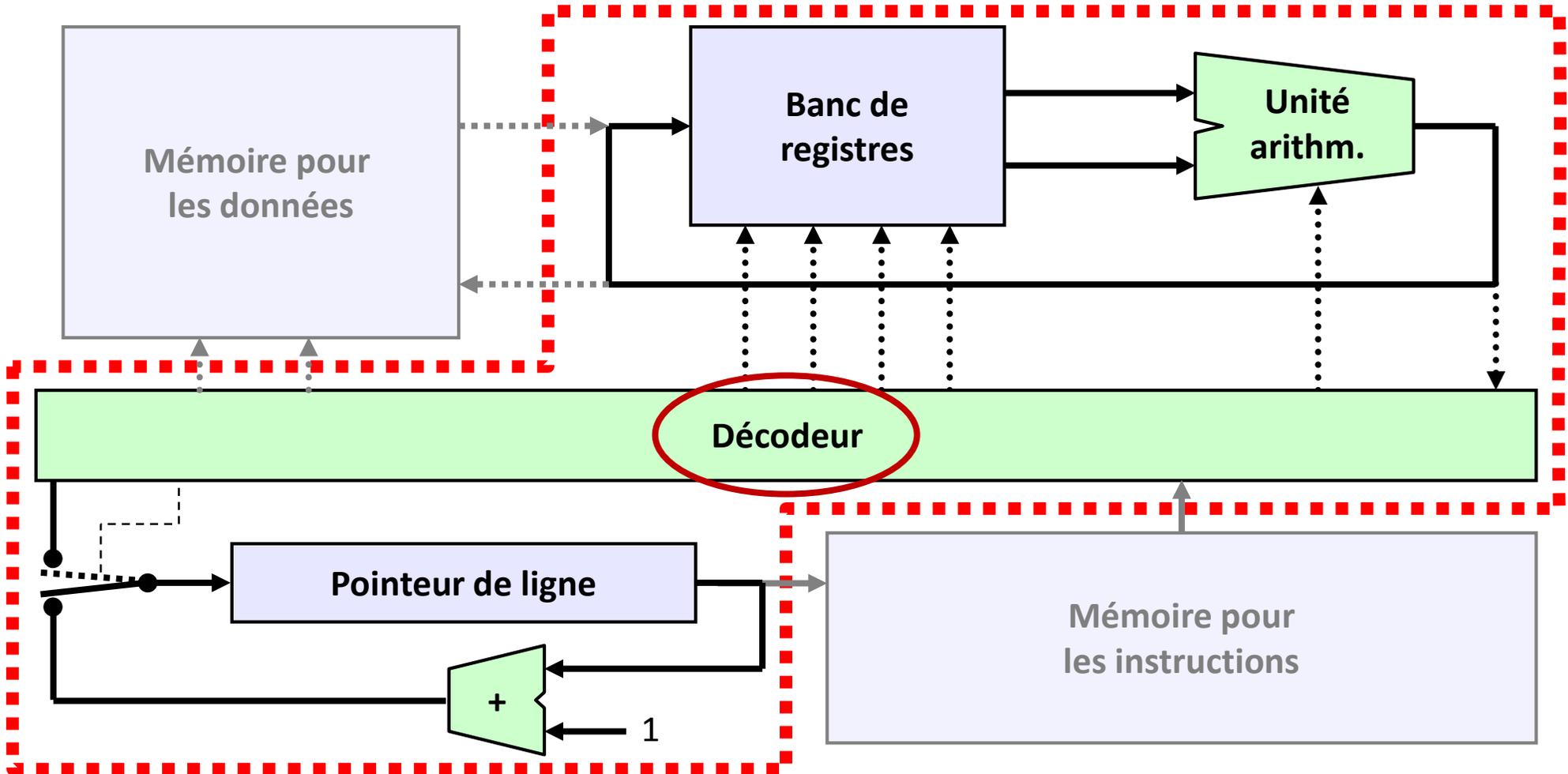
Langage assembleur



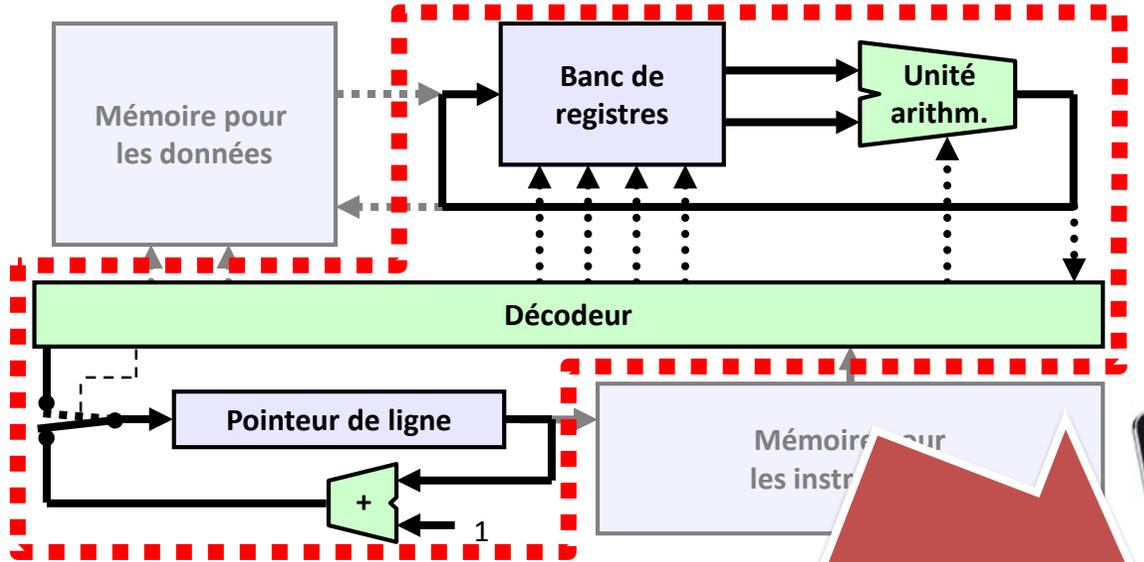
somme des premiers <i>n</i> entiers	
entrée : <i>r1</i> sortie : <i>r2</i>	
1:	<code>0100010010111010100</code>
2:	<code>0101100011100000101</code>
3:	<code>1110101101010010010</code>
4:	<code>1110101101000010011</code>
5:	<code>0001100101010010101</code>
6:	<code>0100010110010111001</code>

Langage machine en binaire

# Un processeur !



# On s'en approche...

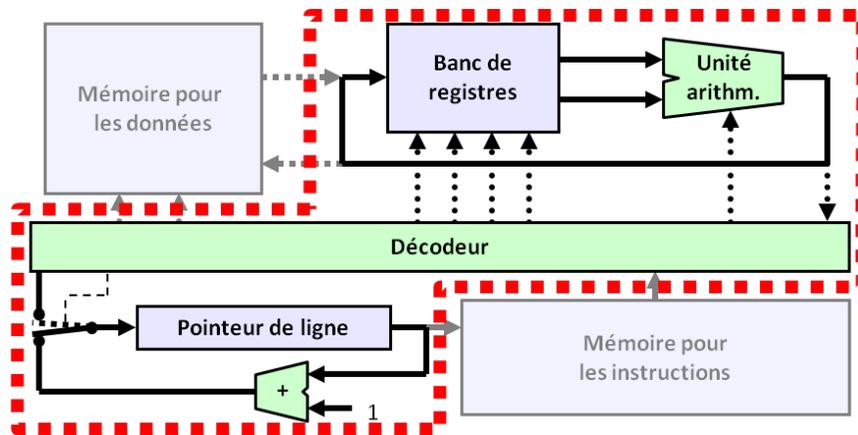


Architecture du processeur

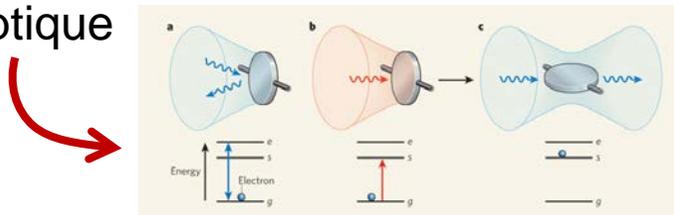


# Jusque-là, pas de choix technologique...

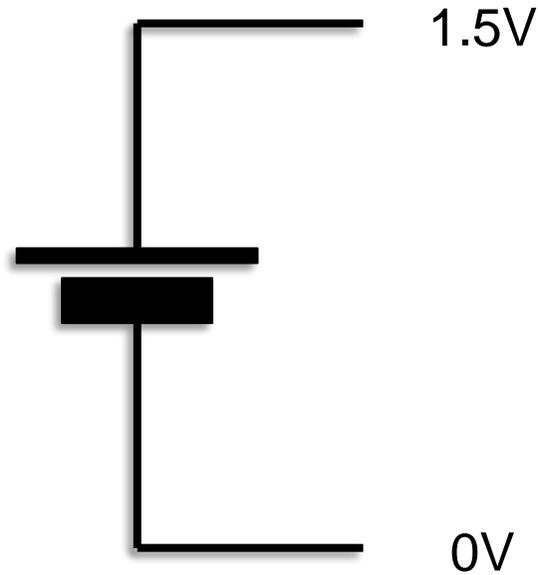
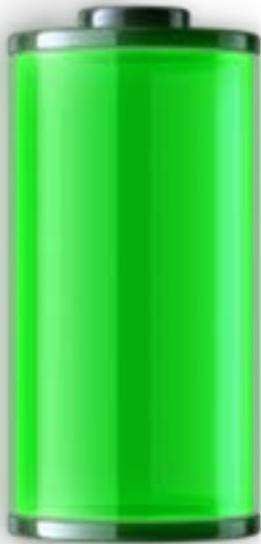
- ▶ Notre machine est parfaitement abstraite et totalement indépendante d'un choix d'implémentation
- ▶ Même l'encodage binaire n'est nullement une nécessité



- ▶ Toute technologie est possible :
  - Electromécanique (p.ex., relais)
  - Electronique (p.ex., tubes ou transistors)
  - Optique



# Une pile, deux niveaux de tension



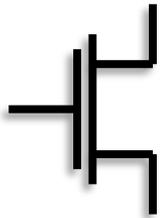
' 1 '

' 0 '

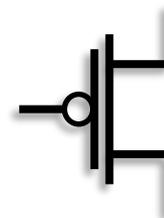
► Très bien adaptés à représenter l'information de manière binaire

# Les transistors, des interrupteurs contrôlés

**n-mos**

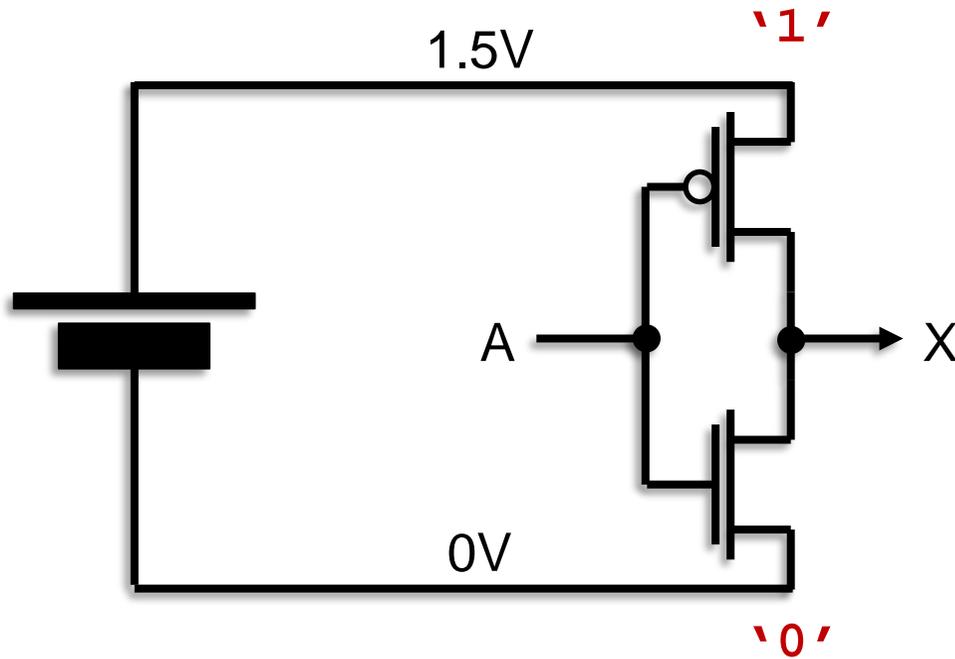


**p-mos**



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

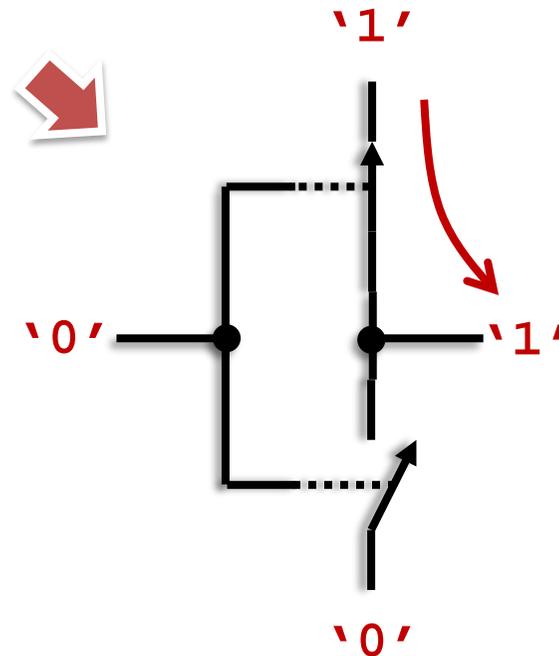
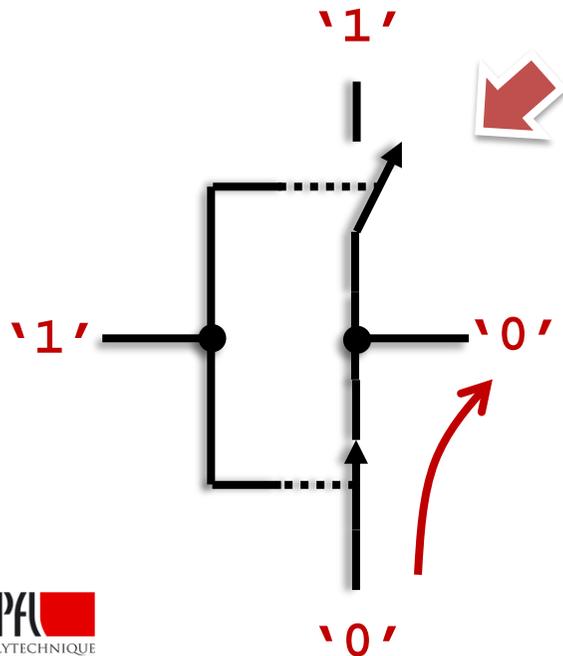
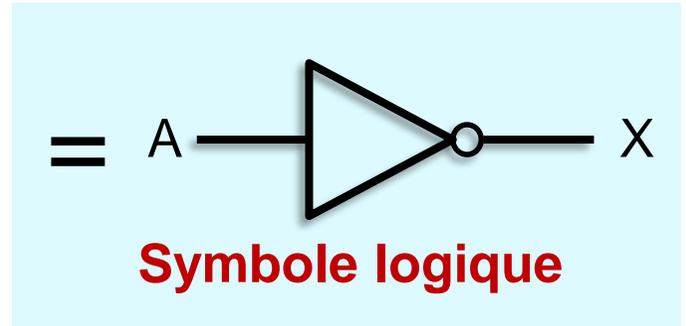
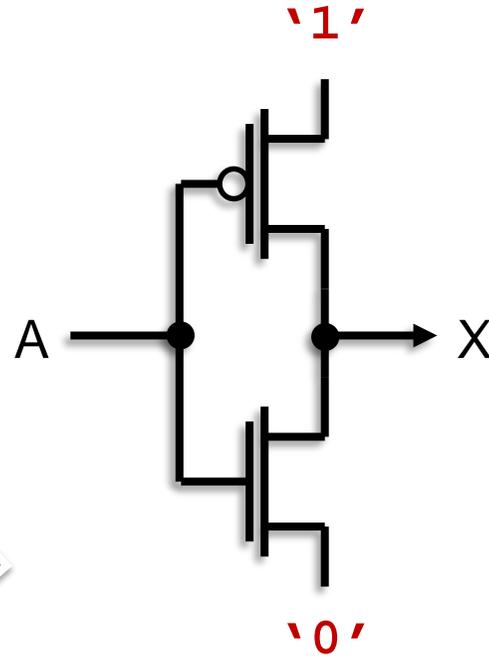
# Un inverseur (en CMOS)



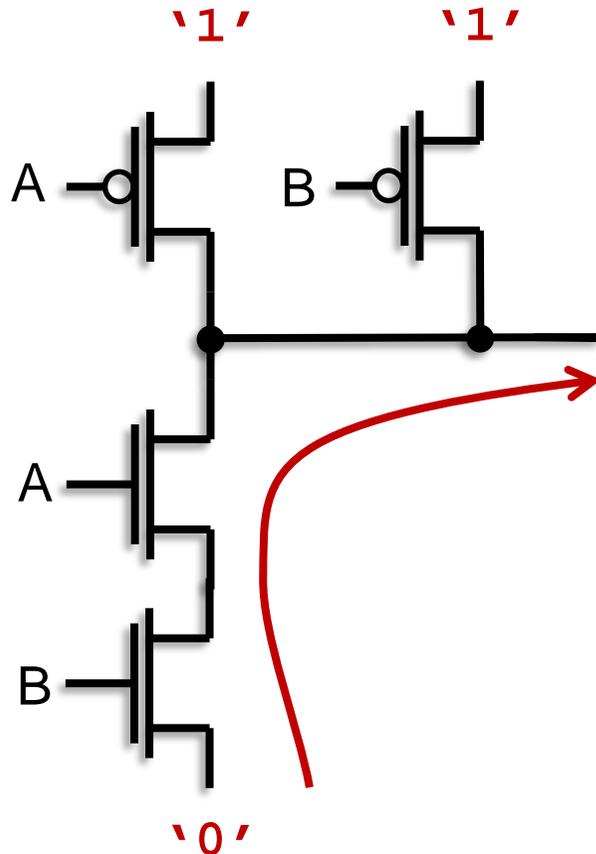
# Un inverseur (en CMOS)

Table de la vérité

A	X
0	1
1	0



# Un circuit « et » avec sortie inversée

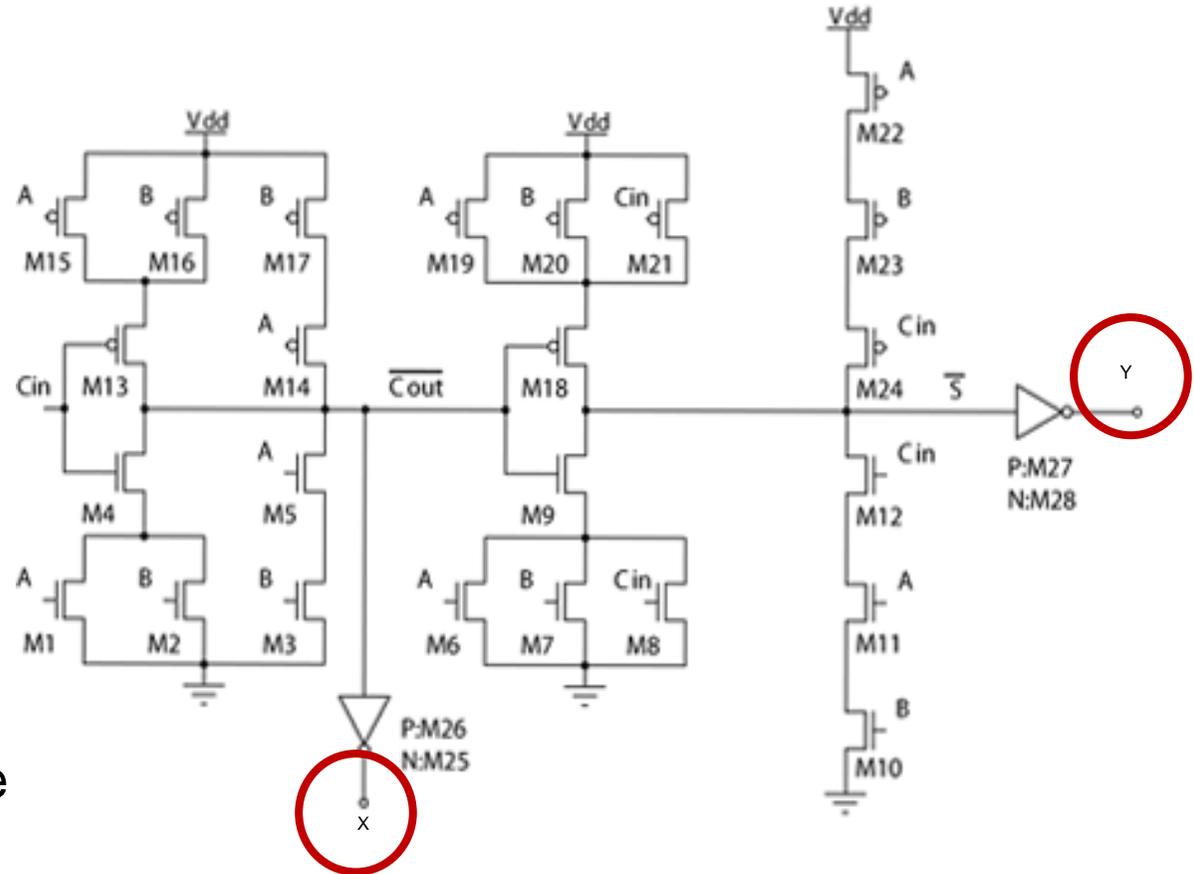


A	B	A et B inversé
0	0	1
0	1	1
1	0	1
1	1	0

- ▶ 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'

# 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

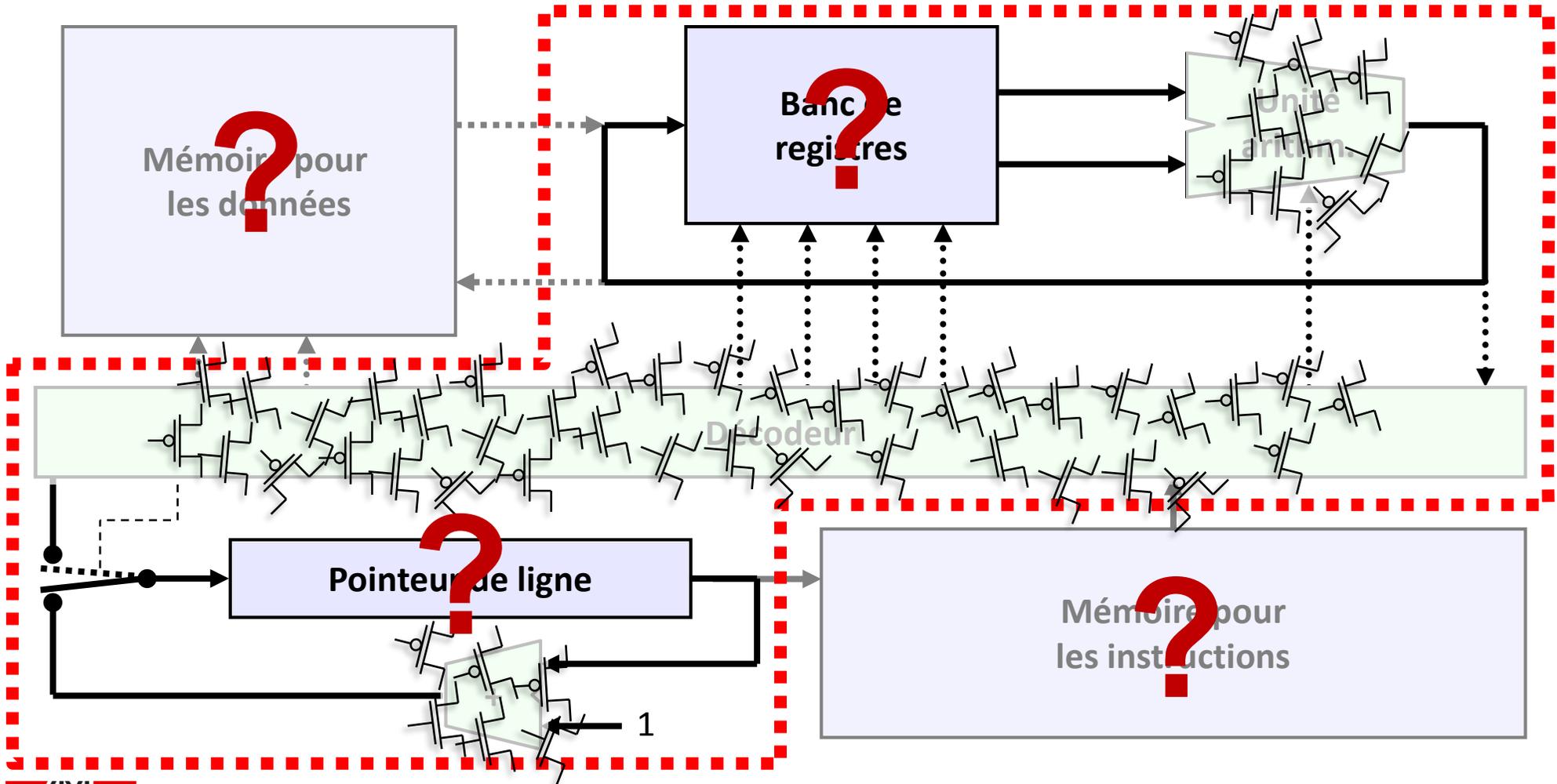


Les sorties XY sont la somme  
(en représentation binaire)  
des trois bits A, B et C à l'entrée



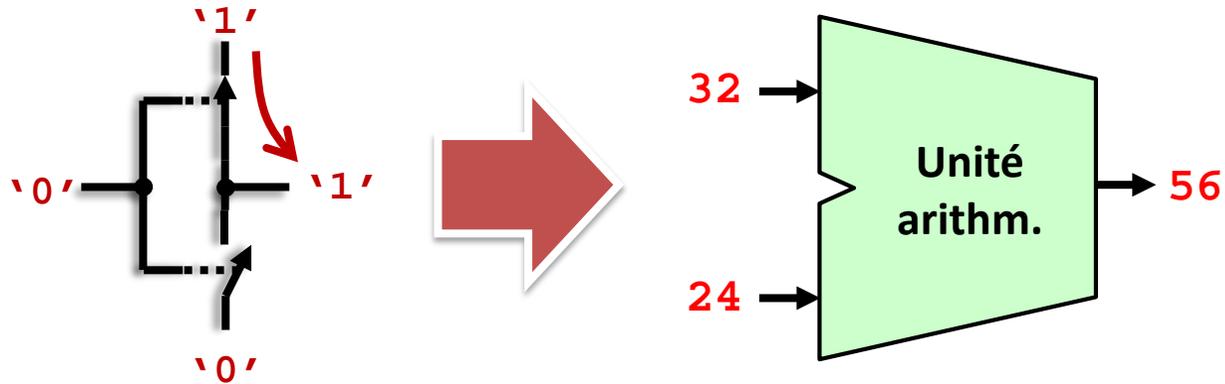
## Addition !

# Et notre processeur, donc ?

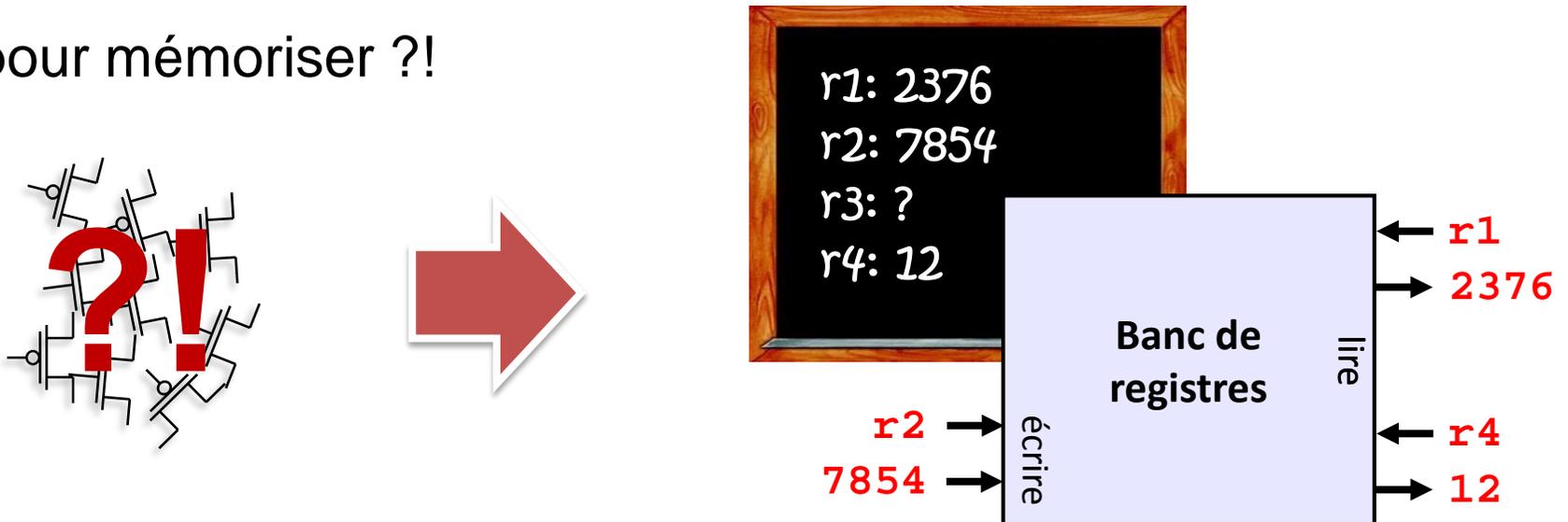


# Peut-on aussi mémoriser l'information ?

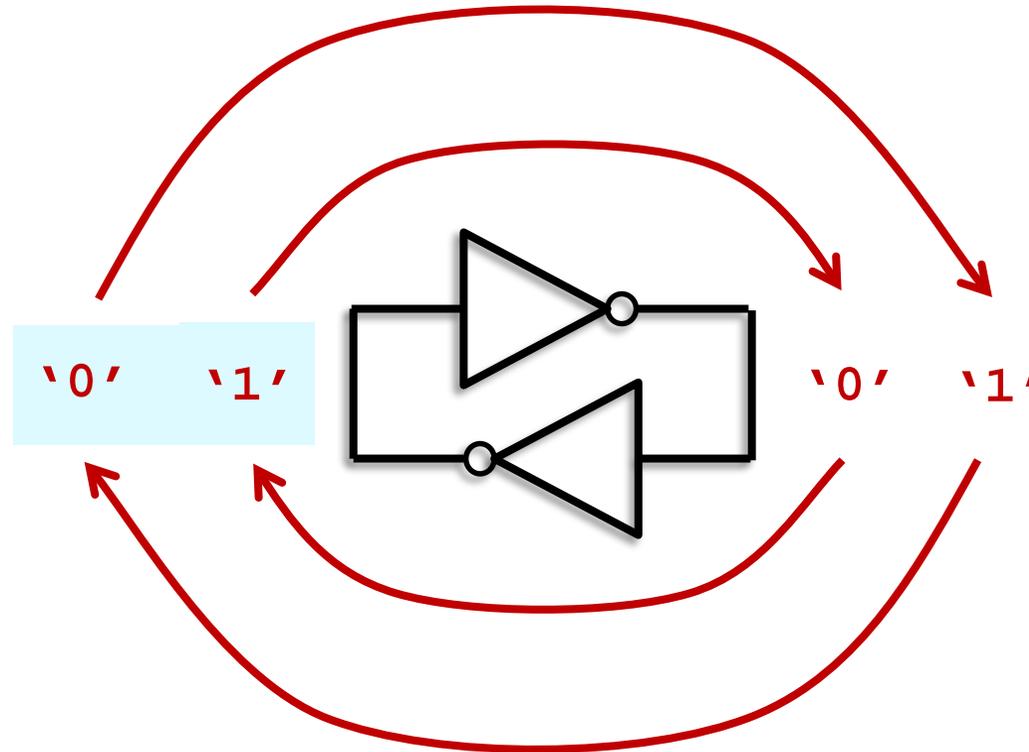
- Pour les calculs, tout va bien :



- Mais pour mémoriser ?!

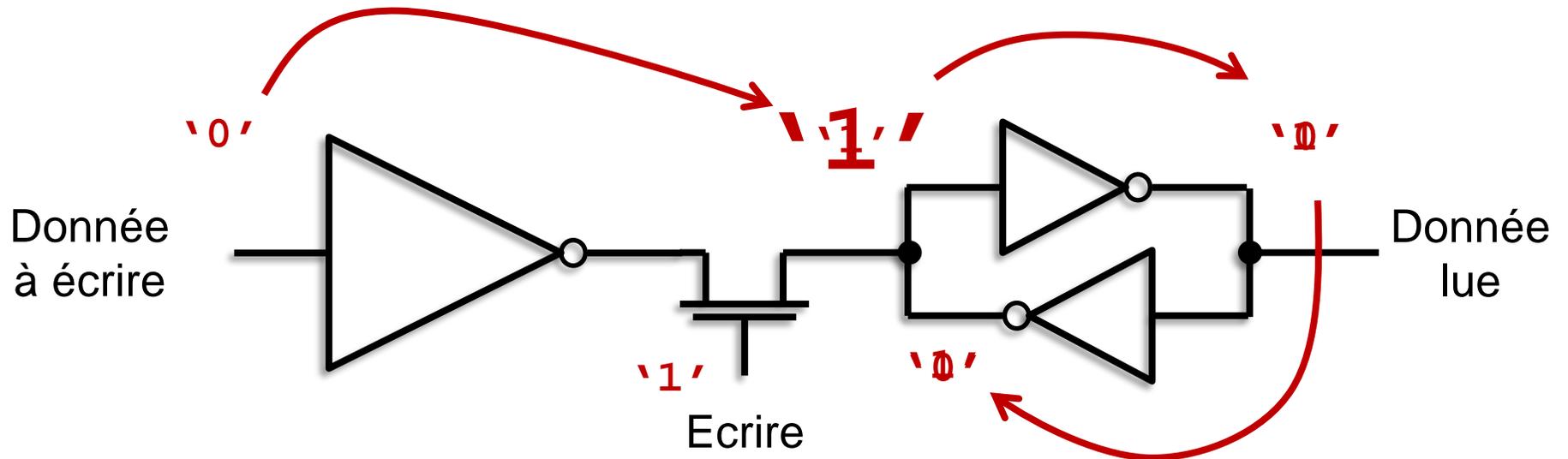


# Un circuit assez particulier

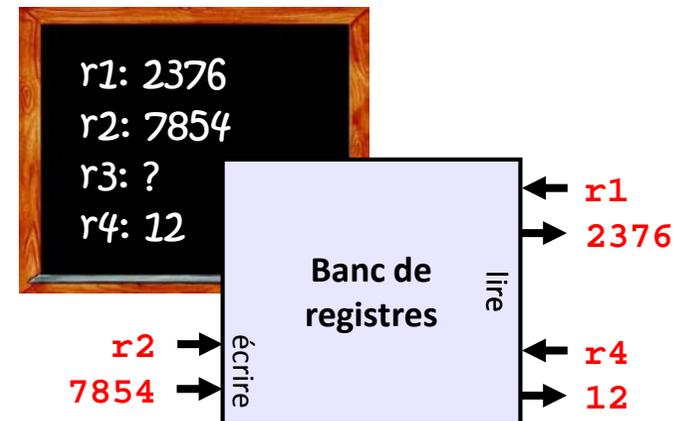


- ▶ Un circuit « bistable » c.à.d. qui peut être dans un parmi deux états parfaitement stables → **un élément mémoire !**

# Comment écrire cette mémoire ?

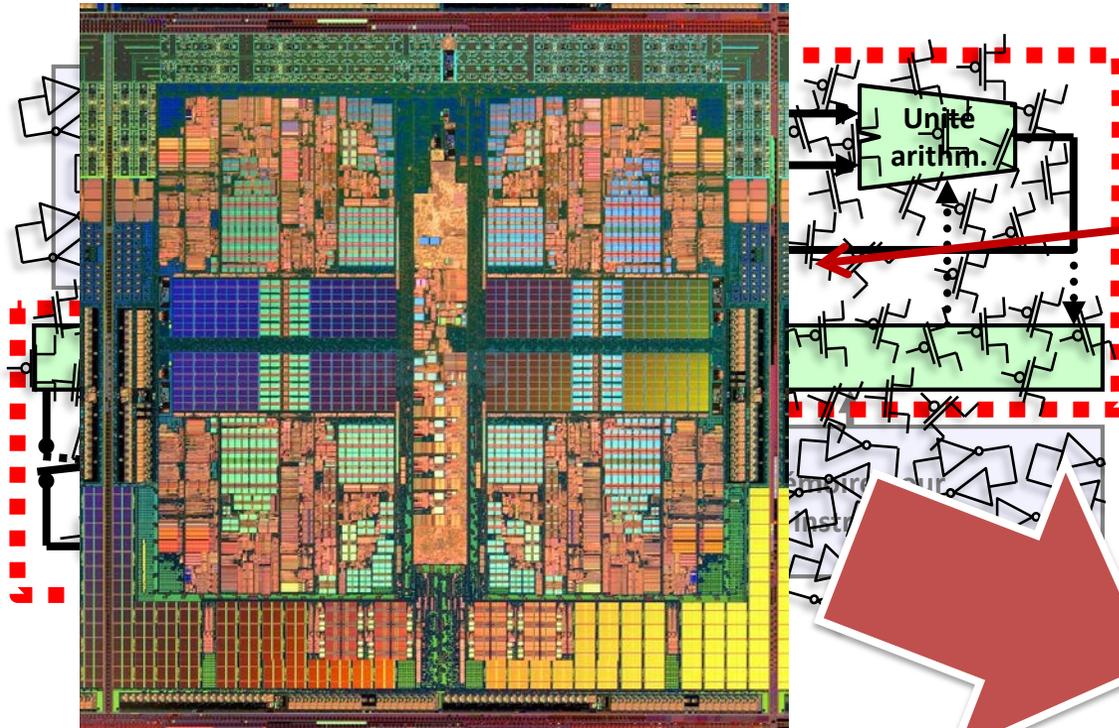


- ▶ Avec quelques transistors on a un parfait circuit mémoire pour notre processeur





# On a atteint notre but !



**Circuit intégré VLSI  
(aujourd'hui autour de  
 $10^8$ - $10^9$  transistors)**

**Architecture du  
processeur**

**Circuit électronique**



# Des algorithmes aux ordinateurs

**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**

$m \leftarrow s$



**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**

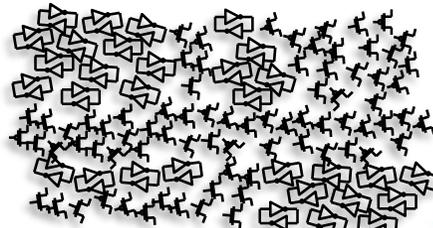
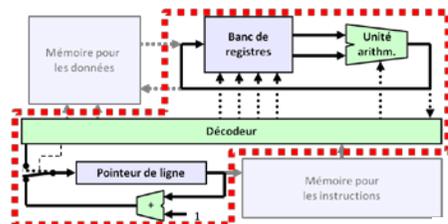
entrée :  $r1$   
sortie :  $r2$

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



**Logiciel**

**Matériel**

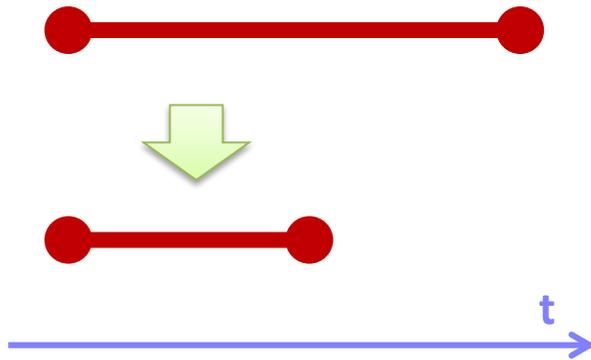




# Augmenter la performance ?

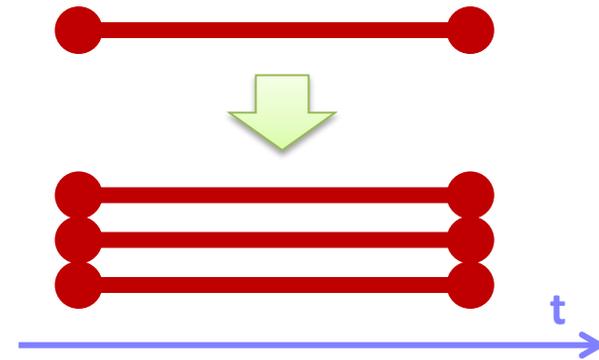
**= Réduire le délai**

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

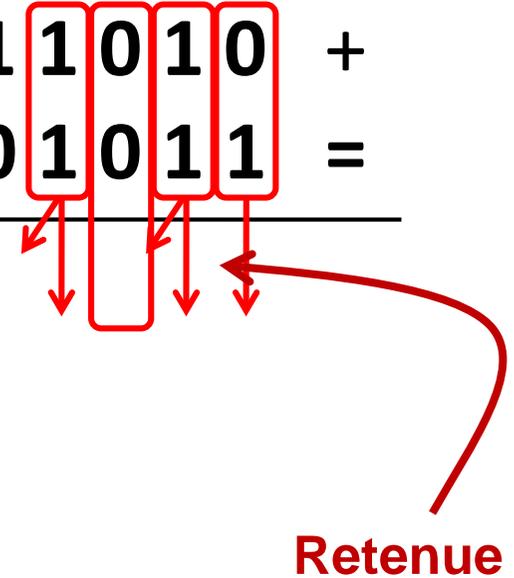
**Réduire le délai** d'un additionneur

2. Au niveau de la structure du processeur

**Augmenter le débit** d'instructions

# Faire des sommes est facile...

$$\begin{array}{r} A \quad 01110101011000110100 \\ B \quad 1011100010111001011 \\ \hline \end{array} + =$$



## 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

... 1 0  
... 1 1

## Somme

$$0 + 0 \neq 0 = 0$$

$$0 + 0 \neq 1 = 1$$

$$0 + 0 \neq 0 = 1$$

$$0 + 1 \neq 10 = 10$$

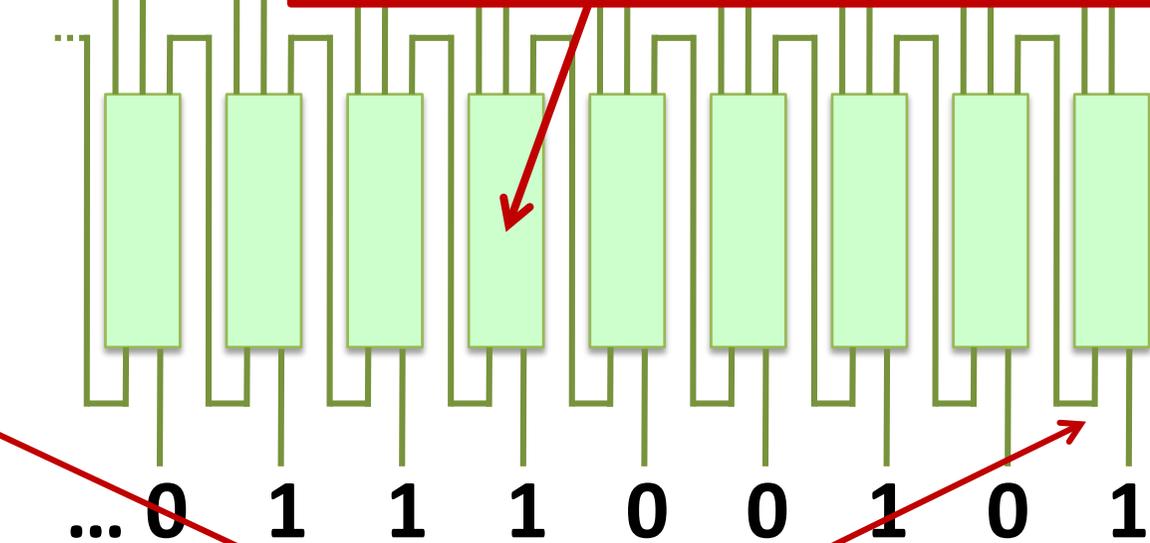
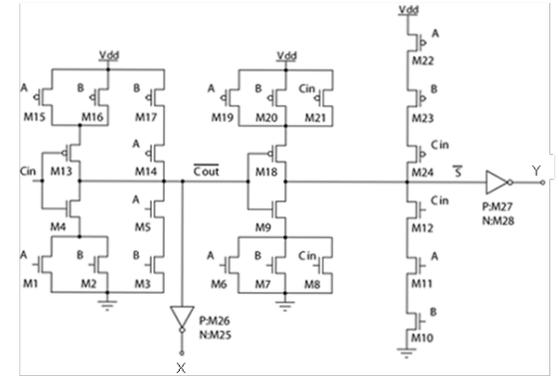
$$1 + 0 + 0 = 1$$

$$1 + 0 + 1 = 10$$

$$1 + 1 + 0 = 10$$

$$1 + 1 + 1 = 11$$

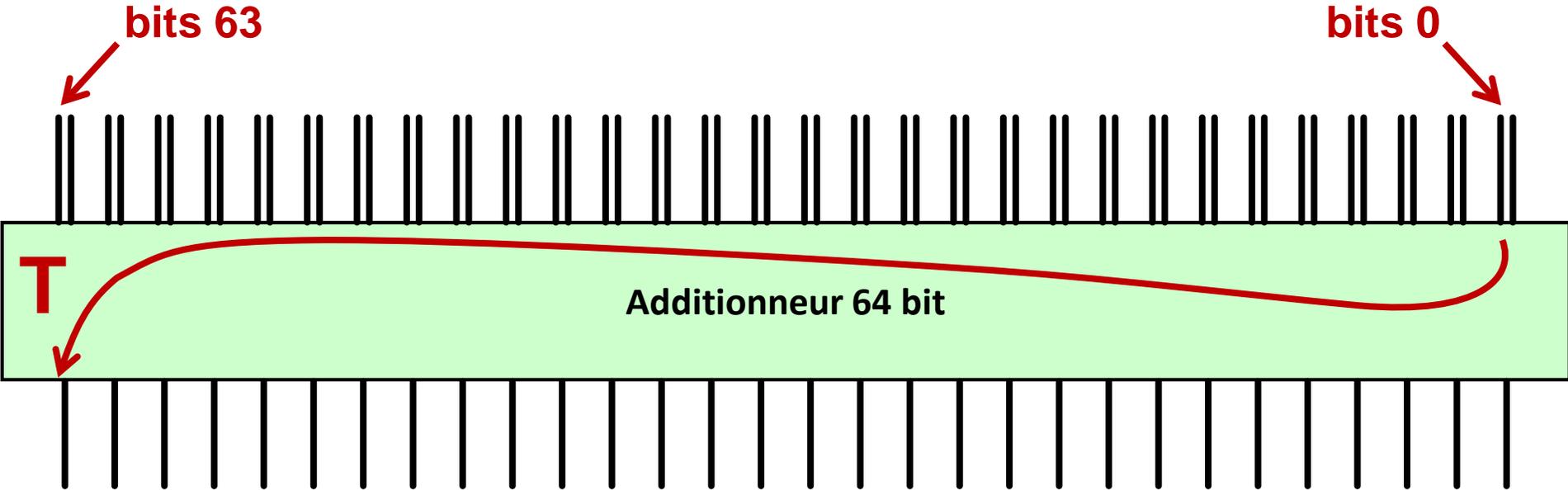
A	B	C	XY
0	0	0	00
0	0	1	01
0	1	0	01
0	1	1	10
1	0	0	01
1	0	1	10
1	1	0	10
1	1	1	11



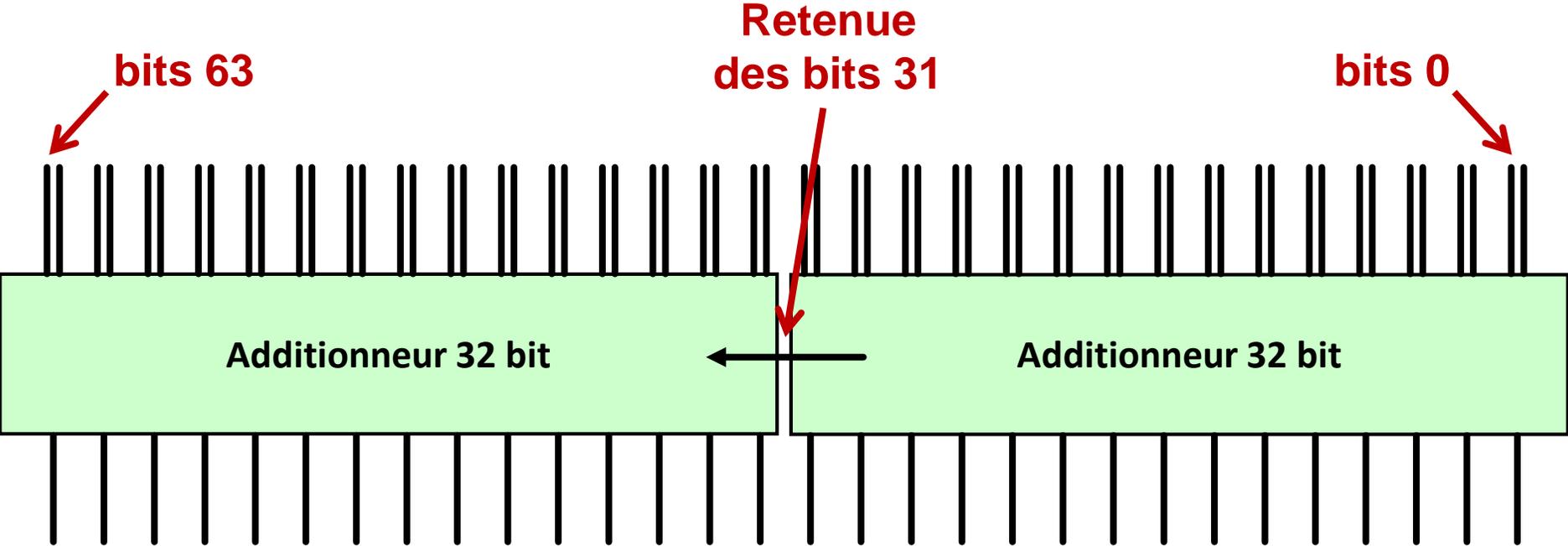
**Il faut ajouter aussi la retenue**



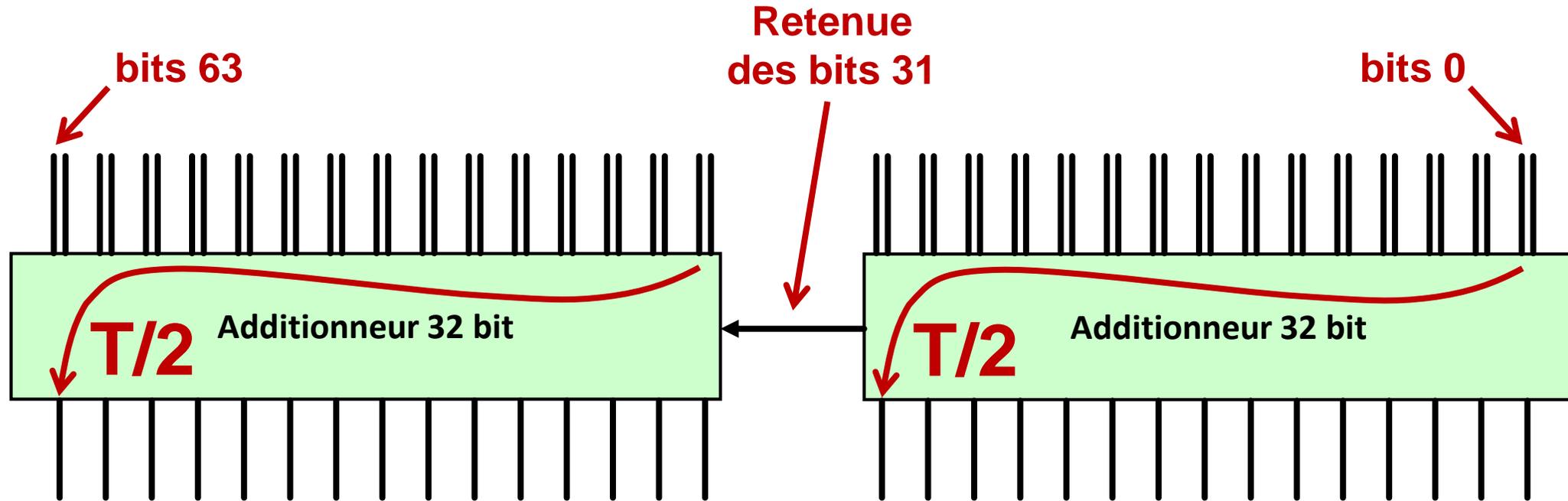
# Peut-on faire mieux ?



# Peut-on faire mieux ?

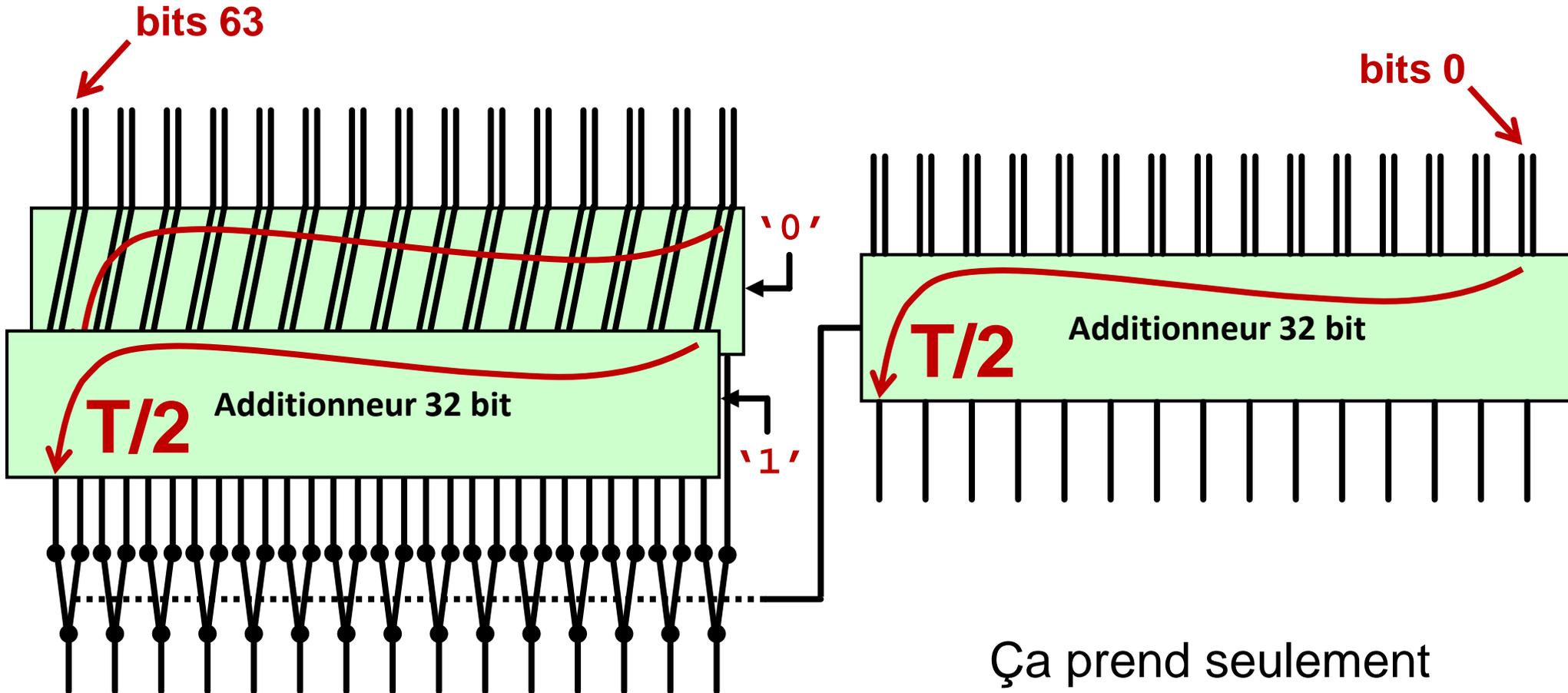


# Peut-on faire mieux ?



On n'a **rien** gagné...

# Peut-on faire mieux ?



Ça prend seulement  
**la moitié du temps !**

# Le génie informatique

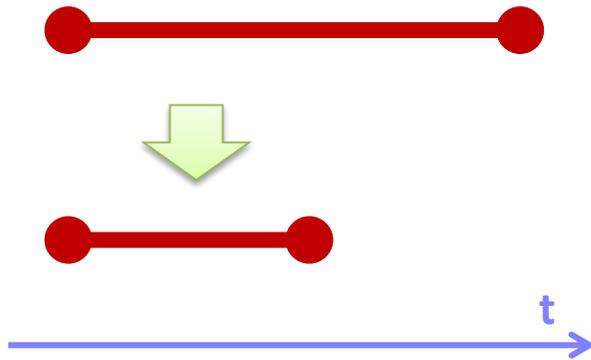
- ▶ On peut changer profondément le circuit sans en changer la fonctionnalité
- ▶ On peut investir plus de transistors et plus d'énergie pour obtenir des circuits très rapides
- ▶ On peut ralentir les circuits pour épargner de l'énergie

On vient de voir un exemple de **synthèse logique**, qui est une des branches du génie informatique (ou **Computer Engineering**)

# Augmenter la performance ?

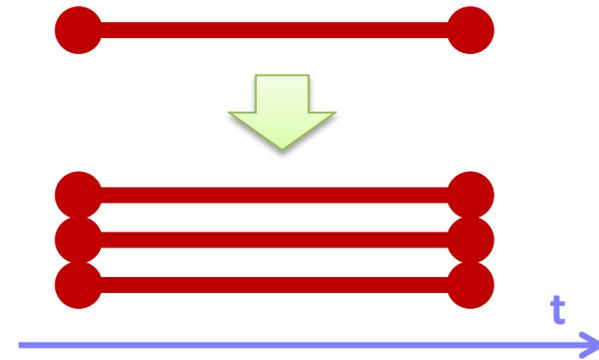
**= Réduire le délai**

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

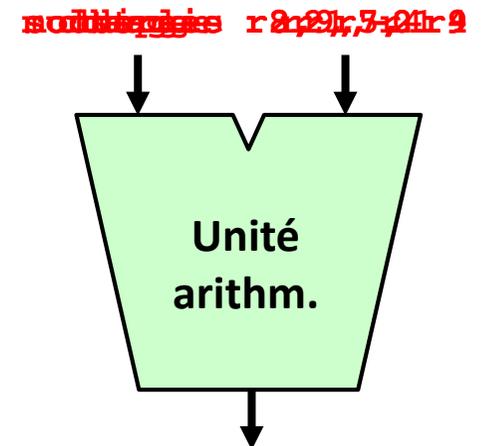
**Augmenter le débit** d'instructions

# 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 ?

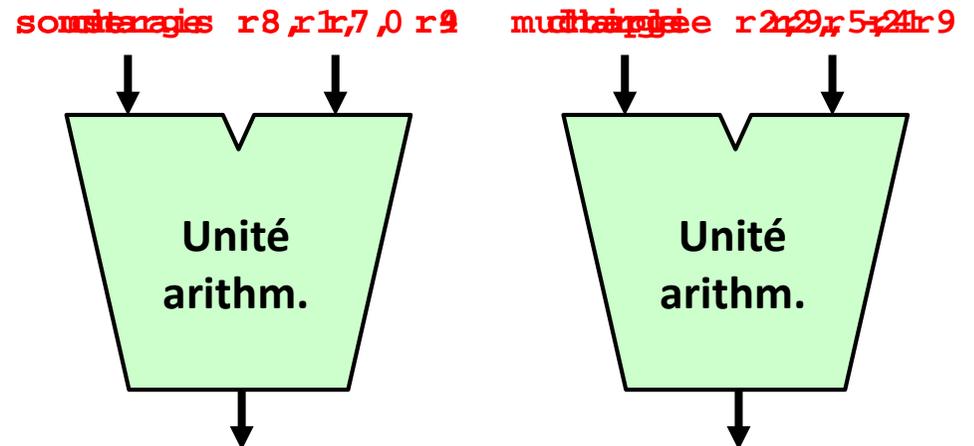


# Doubler le débit de notre processeur

```
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 !

**Problèmes ?!**



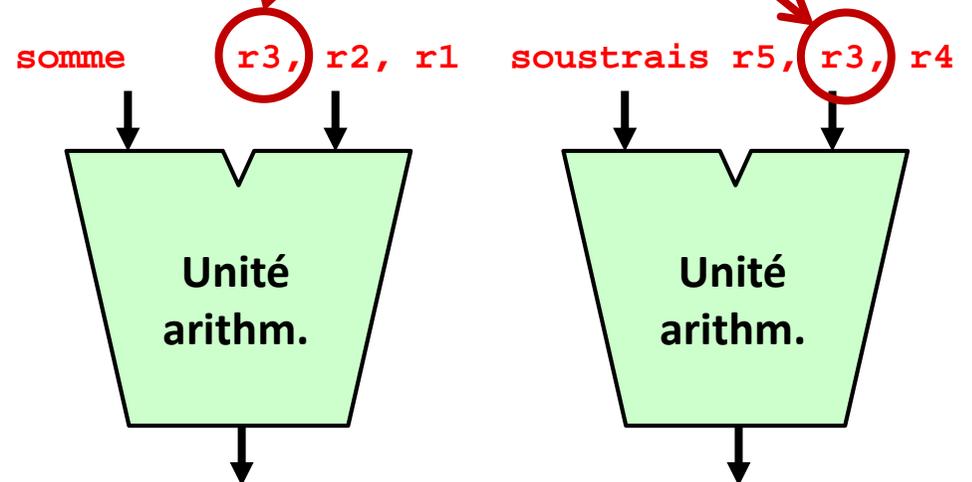
# Doubler le débit de notre processeur

```
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 !  
On exécute maintenant approximativement **deux instructions** par cycle !  
Si on ne fait pas attention,

**le résultat sera faux !**

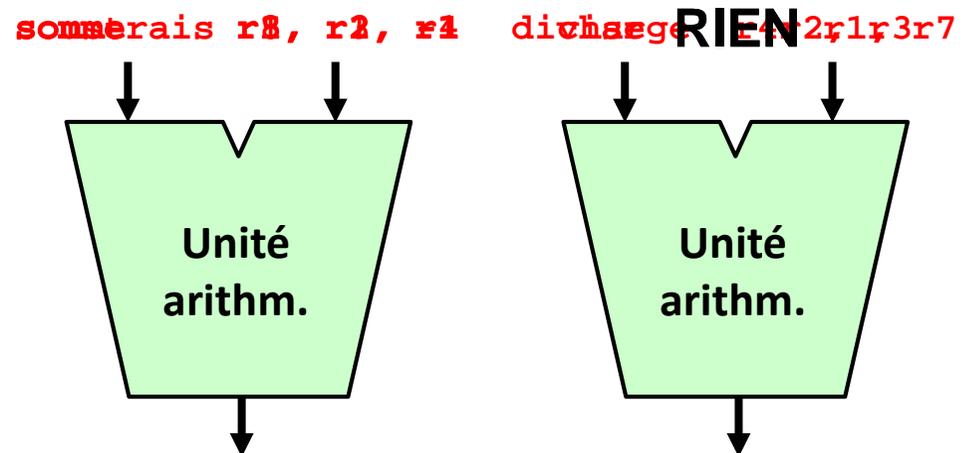
**Problèmes ?!**



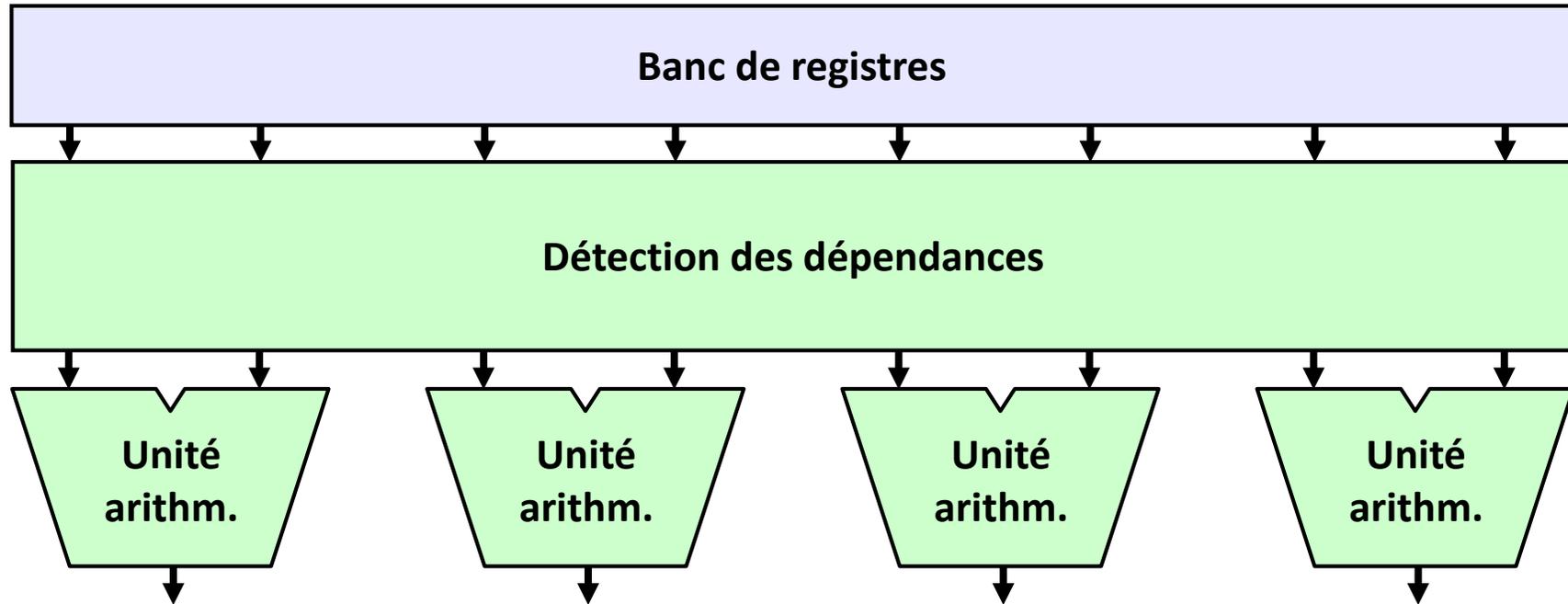
# ~~Double~~ le débit de notre processeur

```
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\_nul**)

# 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**)

# Des algorithmes aux ordinateurs

somme des premiers $n$ entiers
entrée : $n$ sortie : $m$
$s \leftarrow 0$
<b>tant que</b> $n > 0$
$s \leftarrow s + n$
$n \leftarrow n - 1$
$m \leftarrow s$



## Langages de programmation

C, Java,  
Python, Perl,  
PHP, Scala,  
etc.



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

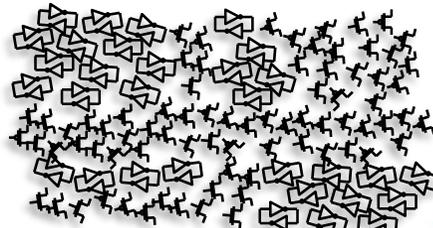
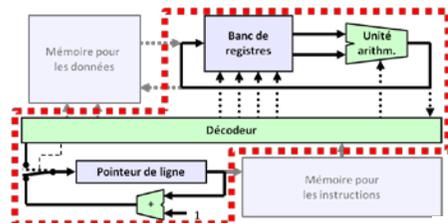


somme des premiers $n$ entiers
entrée : $r1$ sortie : $r2$
1: 0100010010111010100
2: 0101100011100000101
3: 1110101101010010010
4: 1110101101000010011
5: 0001100101010010101
6: 0100010110010111001



Logiciel

Matériel



Ordinateur à  
programme  
enregistré