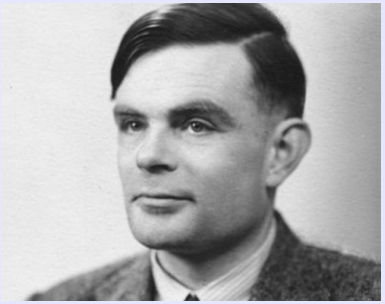
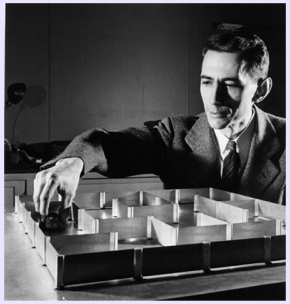
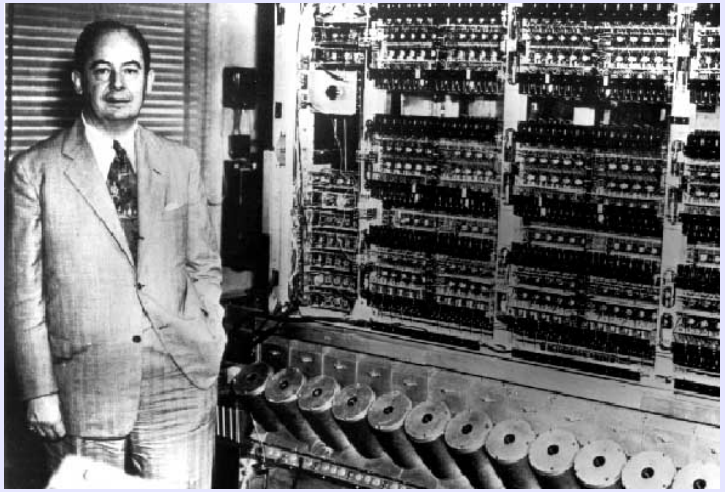




# Information, Calcul et Communication

## Module 3 : Systèmes

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

# Information, Calcul et Communication
# Ordinateur à programme enregistré

Prs. P. Iene, W. Zwaenepoel, A. Ailamaki & P. Janson

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

# La question de ce module

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

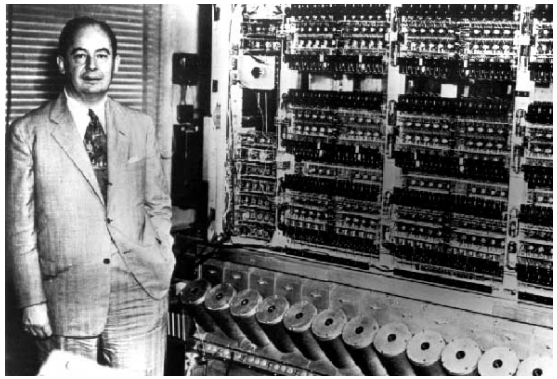
# Les réponses de ce module

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 3 (Réseaux)

# Architecture de von Neumann



unité de contrôle

unité arithmétique et logique (ALU)

**Unité centrale de traitement** (CPU)

*instructions*

*données*

**Mémoire centrale** (RAM/ROM)

**Unités d'E/S**

**Contrôleur de périphériques**

**Unités périphériques**

# Objectifs du cours d'aujourd'hui

Les objectifs de cette leçon sont de :

- expliquer comment l'on peut effectivement construire des machines pouvant exécuter des programmes (= traductions d'algorithmes)
- présenter avec quelle technologie les ordinateurs actuels sont construits
- présenter les deux principes permettant d'augmenter la rapidité de calcul de tels ordinateurs

# La première question de cette leçon

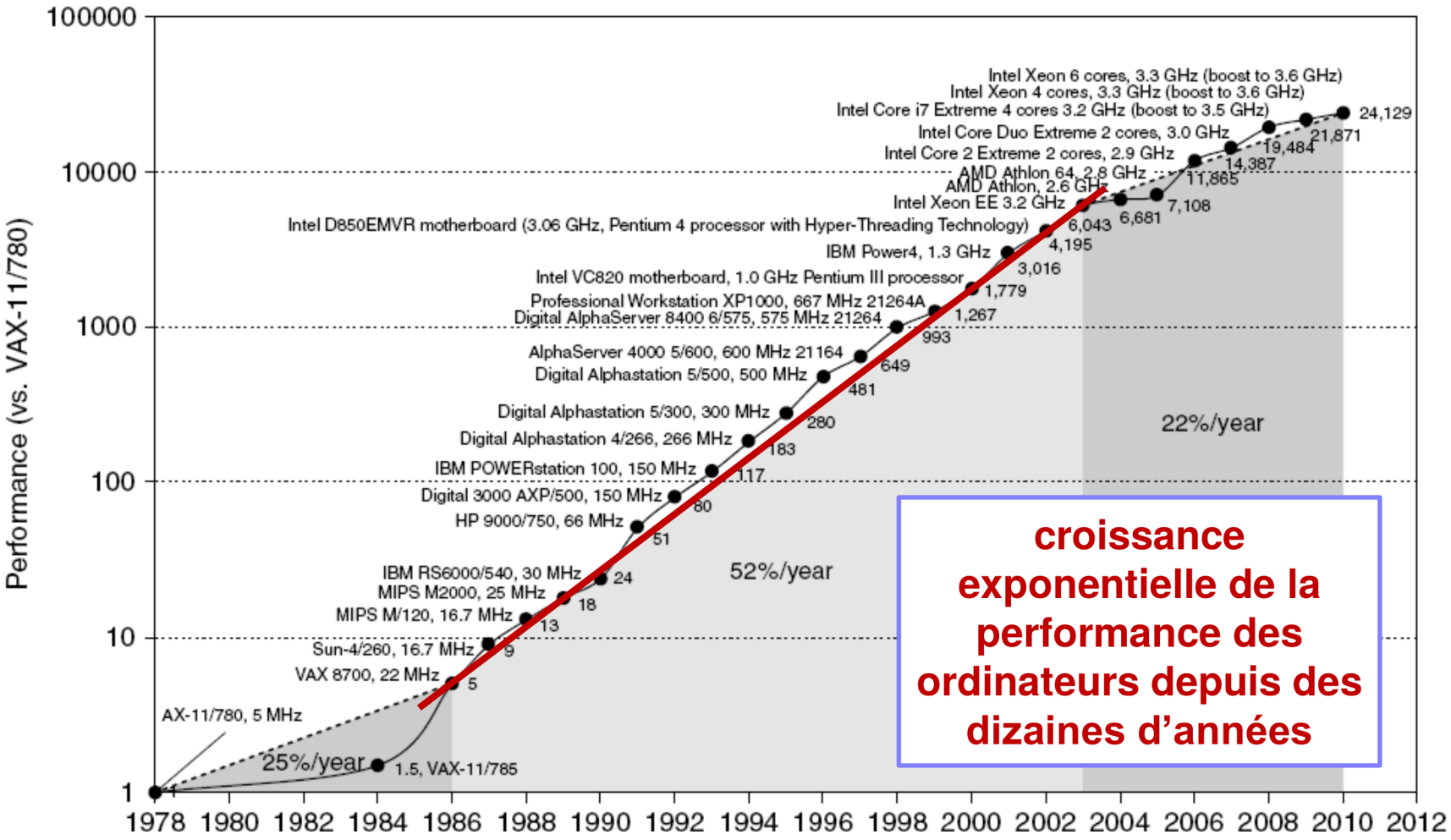
- Maintenant que l'on a développé des algorithmes, comment peut-on **construire des systèmes pour les exécuter** ?

☞ **Algorithme** :

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

# La deuxième question de cette leçon

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




Source: Hennessy & Patterson, © MK 2011

# 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 : *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
```

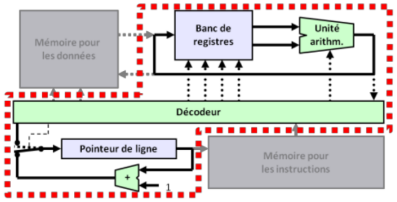
**somme des premiers *n* entiers**

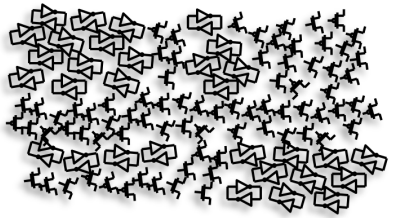
entrée : *r1*
sortie : *r2*

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

## Logiciel

## Matériel



# Des algorithmes aux ordinateurs

# Un algorithme

Pour exprimer les algorithmes, nous avons dans le Module 1 utilisé un langage assez intuitif, proches des mathématiques et du langage naturel.

Considérons un exemple :

| **somme** |
| --- |
| entrée : $n$<br>sortie : $S(n)$ |
| $s \leftarrow 0$<br>***Tant que*** $n > 0$<br>  $s \leftarrow s + n$<br>  $n \leftarrow n - 1$<br>***sortir :*** $s$ |

# Ecriture plus contrainte

Pour *simplifier* en vue de construire une machine, essayons de réécrire cet algorithme de façon plus précise, avec **moins de libertés**
(tout en gardant quelque chose de plus pratique/expressif qu'une table de transition d'une machine de Turing !)

**Etape 1 :** donner un sens concret à « **Sortir** » : affecter une variable

**Etape 2 :** restriction des noms de variables : $r1$, ..., $r6$

| **somme** |
|---|
| entrée : $nr1$<br>sortie : $m = r2 = S(nr1)$ |
| $sr3 \leftarrow 0$<br>***Tant que*** $nr1 > 0$<br>$\quad sr3 \leftarrow sr3 + nr1$<br>$\quad nr1 \leftarrow nr1 - 1$<br>***Sortir :*** $sm \leftarrow sr2 \leftarrow r3$ |

# Les registres

- On a besoin de mémoriser des valeurs
- Ces valeurs seront stockées dans ce qui est appelé des « **registres** » :
  réalisation concrète dans notre machine de la notion de variable
- On les représente simplement par $r1$, ..., $r6$
- Nous essayerons d'en avoir qu'un petit nombre limité
  (de l'ordre de la dizaine)
- Pour de plus grosses données (tableaux, listes, ...) : mémoire *externe* : voir la leçon de la semaine prochaine

# Ecriture plus contrainte (suite)

**Etape 3 :** identifions chacune des opérations nécessaires à notre machine en les nommant

**3.1 :** Par exemple l'affectation : `charge`
**3.2 :** Par exemple les opérations arithmétiques : `somme`

| **somme** |
|---|
| entrée : $r1$<br>sortie : $r2$ |
| `charge` $r3, 0$<br>***Tant que*** $r1 > 0$<br>$r3 \leftarrow r3 +$ $r1$ `somme` $r3, r3, r1$<br>$r1 \leftarrow r1 - 1$ `somme` $r1, r1, -1$<br>`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 (contenus de) **registres**, soit des **constantes**
- On écrit ces opérations ainsi :

```
somme destination, operande1, operande2
```

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

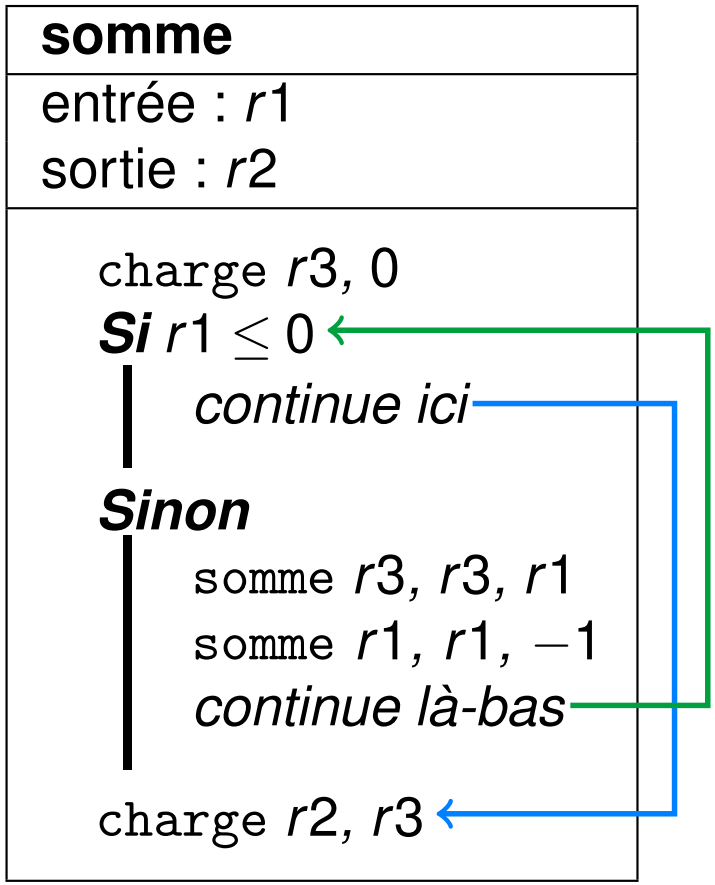
```
somme r5, r6, r7
multiplie r5, r5, r8
```

# Ecriture plus contrainte (suite)

**Etape 4 :** réduisons les structures de contrôle à **une seule** : le branchement conditionnel

- ☞ Mais comment faire des boucles ?
- ▶ en ayant des **sauts** dans le programme :

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



# Numérotation des lignes

Pour spécifier les endroits des sauts (conditionnels ou non) :
on **numérote** les lignes

On a donc introduit des instructions supplémentaires :

- les **sauts conditionnels** : saute à la ligne indiquée si une condition est vérifiée ;

  par exemple `cont_ppe`

- un **saut inconditionnel** : saute à la ligne indiquée
  `continue`

# Langage « Assembleur »

**somme**

entrée : $n$
sortie : $S(n)$

$s \leftarrow 0$
***Tant que*** $n > 0$
  $s \leftarrow s + n$
  $n \leftarrow n - 1$
***sortir :*** $s$

**somme**

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` $r2$, $r3$

# Exemple réél de langage assembleur

```
int somme(int n)
{
  int s(0);
  while (n > 0) {
    s += n;
    --n;
  }
  return s;
}
```

→

```
    movl    $0, -4(%rbp)
.L3:
    cmpl    $0, -20(%rbp)
    jle    .L2
    movl    -20(%rbp), %eax
    addl    %eax, -4(%rbp)
    subl    $1, -20(%rbp)
    jmp    .L3
.L2:
    movl    -4(%rbp), %eax
```

```
g++ -S somme.cc -o somme.a
```

« `movl` » c'est « `charge` »,
« `-4(%rbp)` » c'est « `r3` », et « `-20(%rbp)` » c'est « `r1` »,
« `cont_ppe` » s'écrit en fait sur deux lignes avec « `cmpl` » et « `jle` », etc.

(voir aussi `http://gcc.godbolt.org/`)

# Résumé à ce stade

- 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 quelques variables comme `r1`, `r2`, `r3`, etc. — on les appelle « **registres** »
- Certaines actions indiquent la séquence (p.ex., `continue`, si cette séquence est à suivre toujours, ou `cont_egal`, si elle est à suivre seulement dans certains cas) — on les appelle « **instructions de saut** »

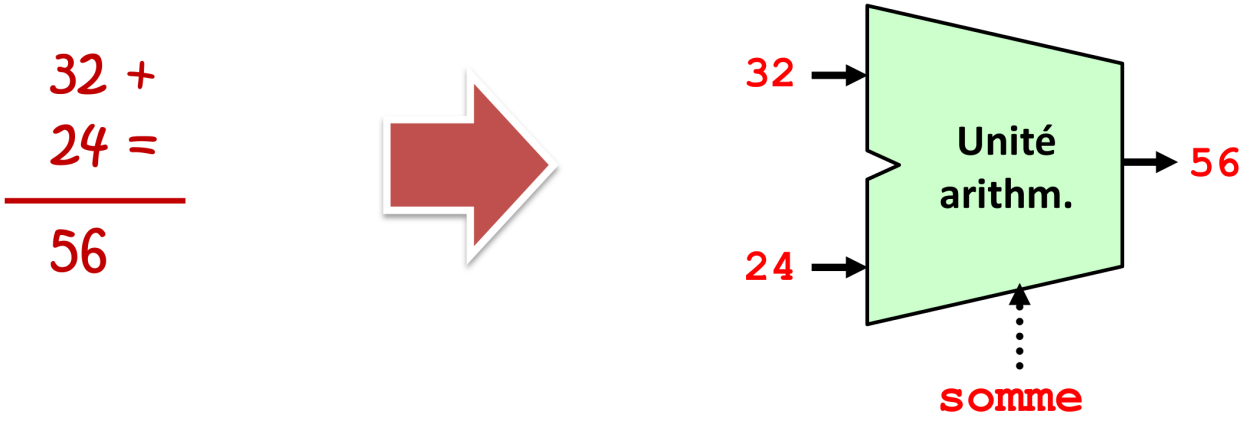
# Essayons de créer une telle machine…

☞ **Algorithme** :

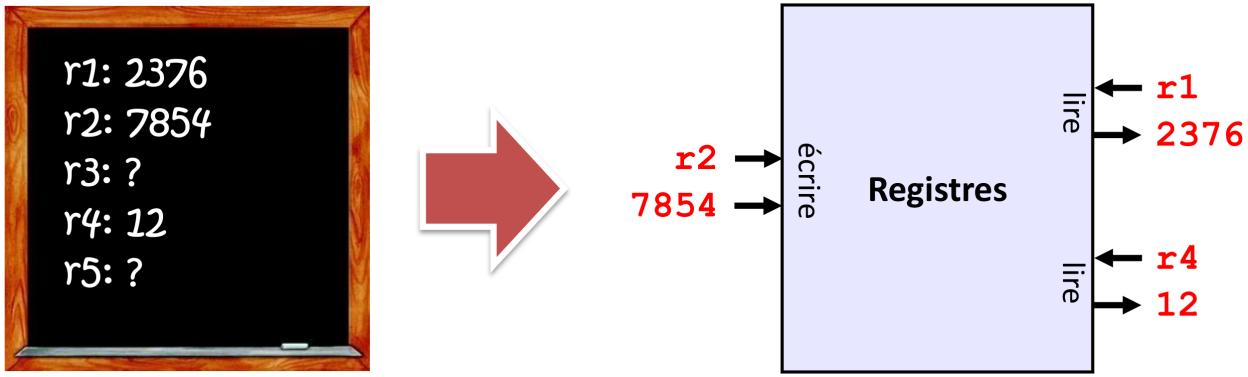
| **Second degré** |
|---|
| entrée : $b$, $c$<br>sortie : $\{x \in \mathbb{R} : x^2 + b\,x + c = 0\}$ |
| $\Delta \leftarrow b^2 - 4\,c$<br>***Si*** $\Delta < 0$<br>  *afficher* $\emptyset$<br>***Sinon***<br>  ***Si*** $\Delta = 0$<br>  $x \leftarrow -\frac{b}{2}$<br>  *afficher* $x$<br>  ***Sinon***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

# De quoi a-t-on besoin ?

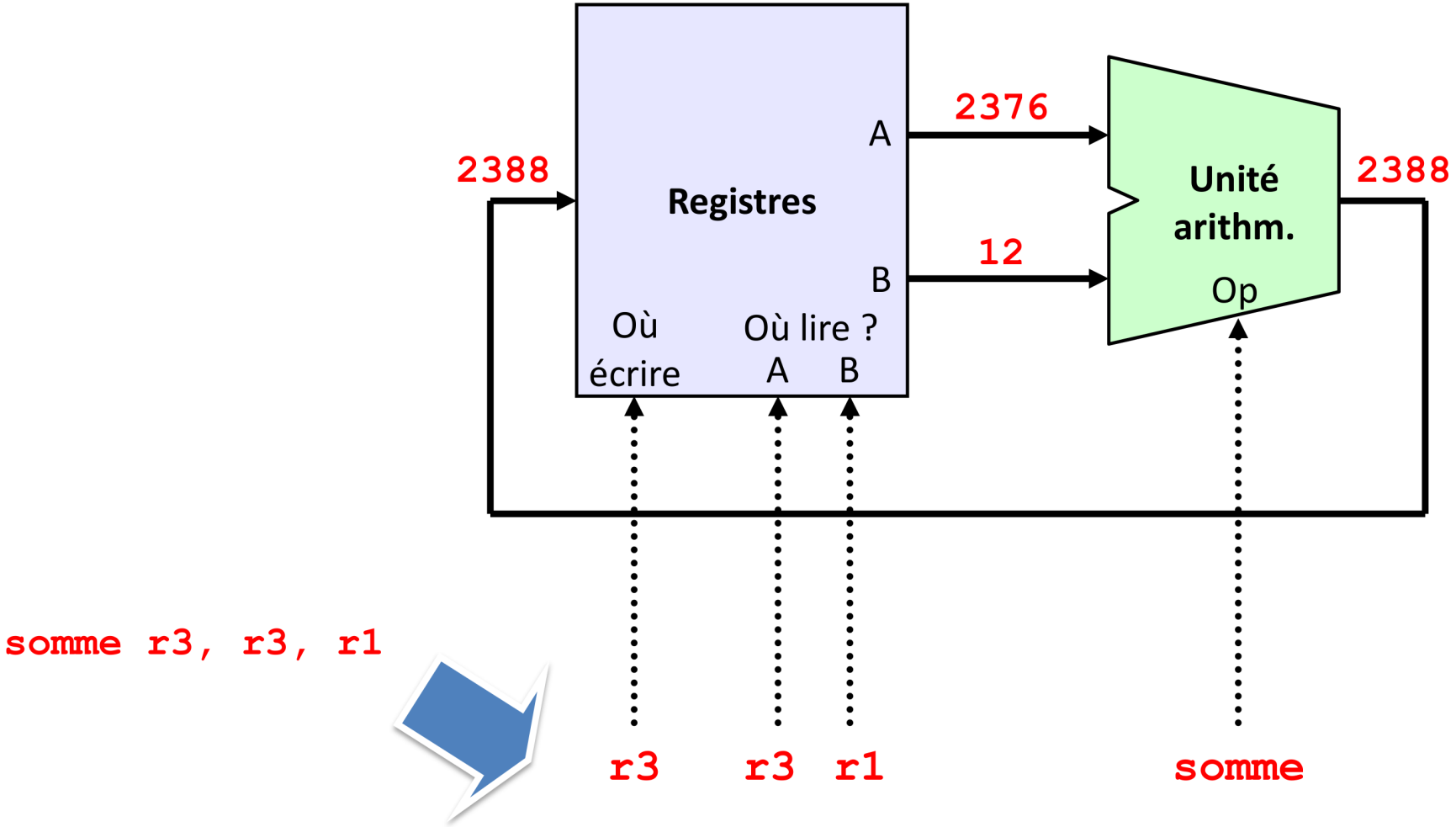
- L'**unité arithmétique** effecture les opérations arithmétiques




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

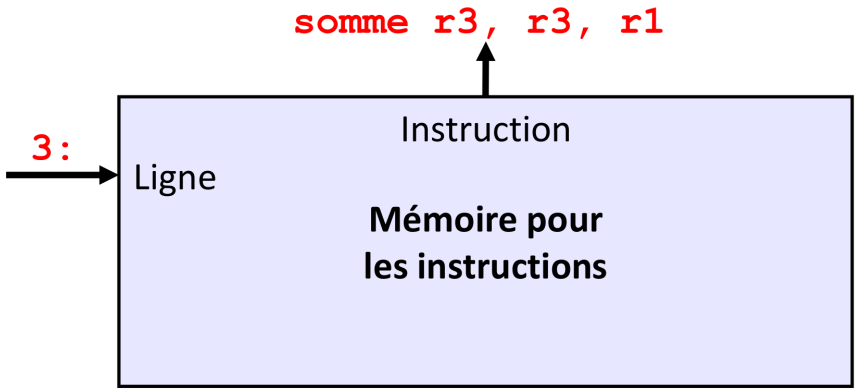


# Le circuit pour les *calculs*



# De quoi a-t-on encore besoin ?

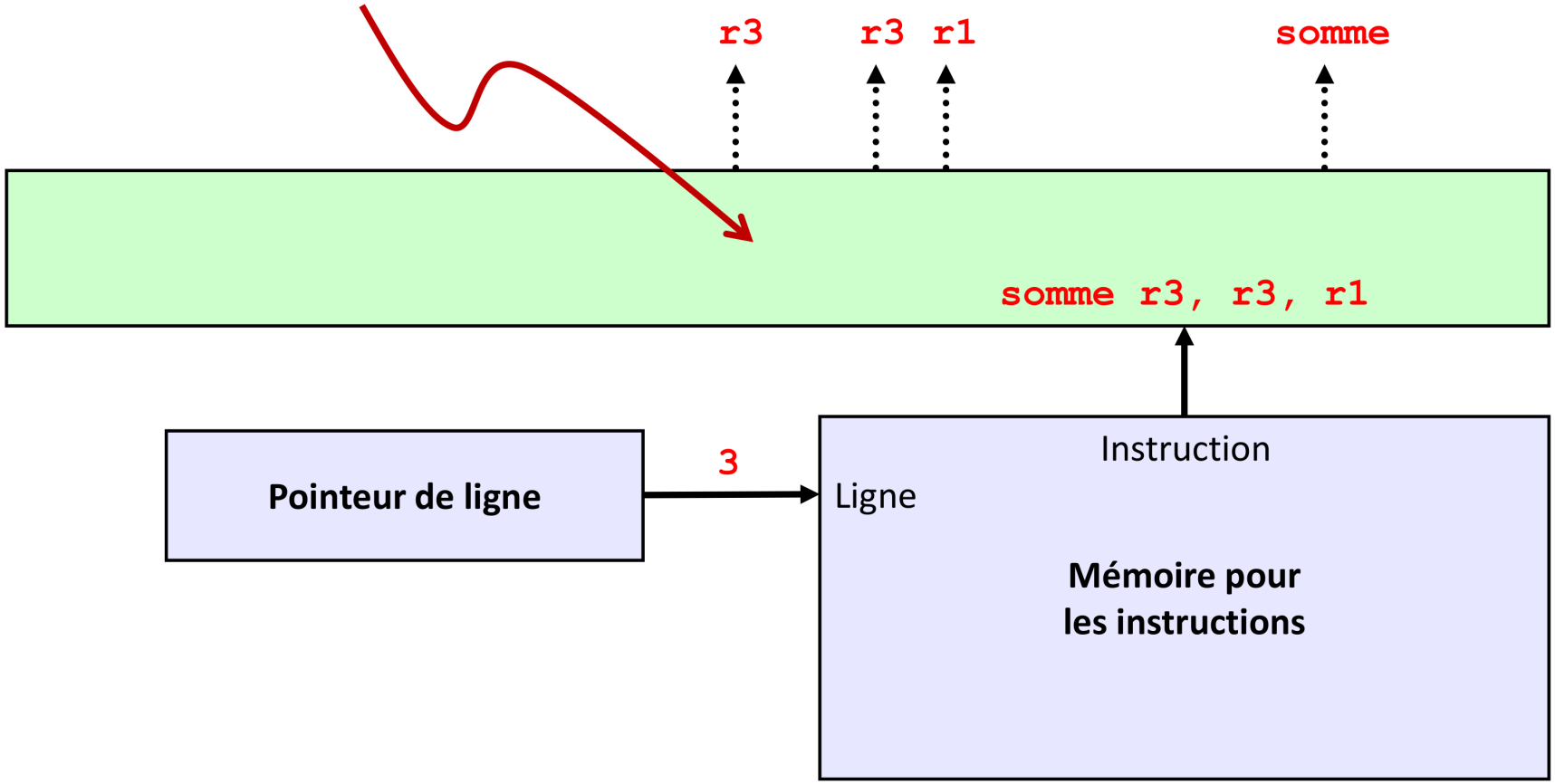
- Le programme doit être enregistrer quelque part


- Il faut pouvoir contrôler où on en est


# Une première partie pour contrôler le tout...

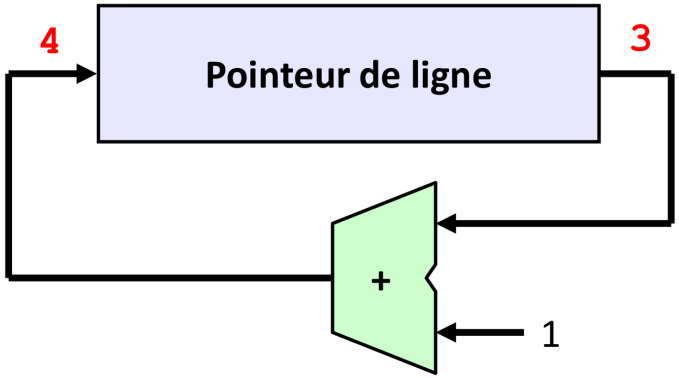
Un circuit assez simple qui répartit les éléments qui constituent une instruction



# En cas de saut ?

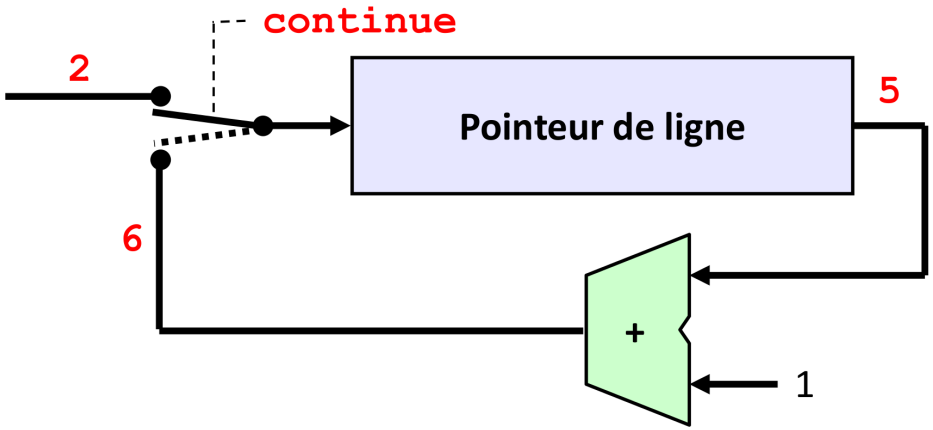
- La plupart du temps on passe simplement à la ligne suivante

```
3: somme    r3, r3, r1
```

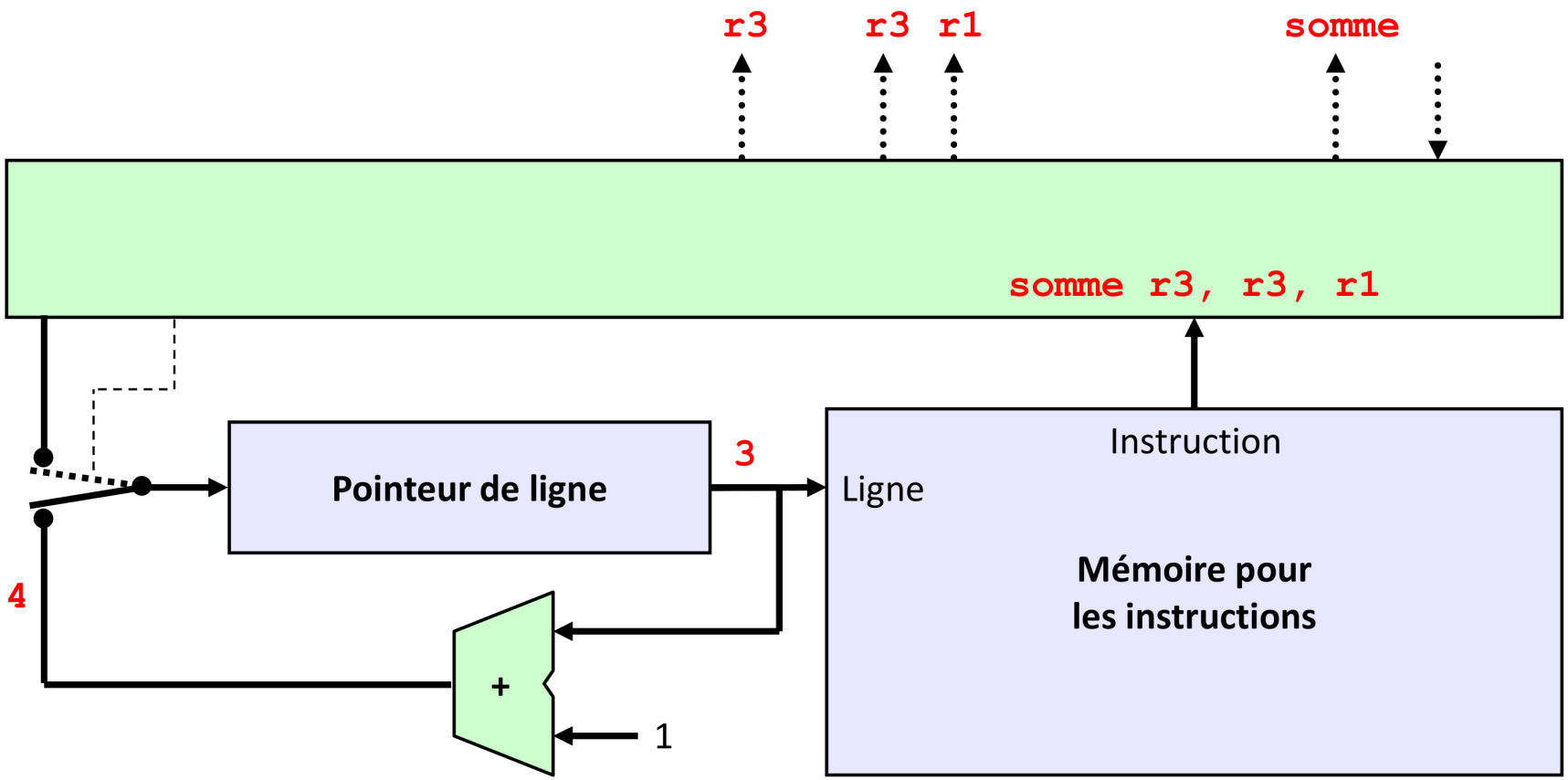



- Si on a une instruction de saut (p.ex. `continue`), on veut imposer une autre ligne

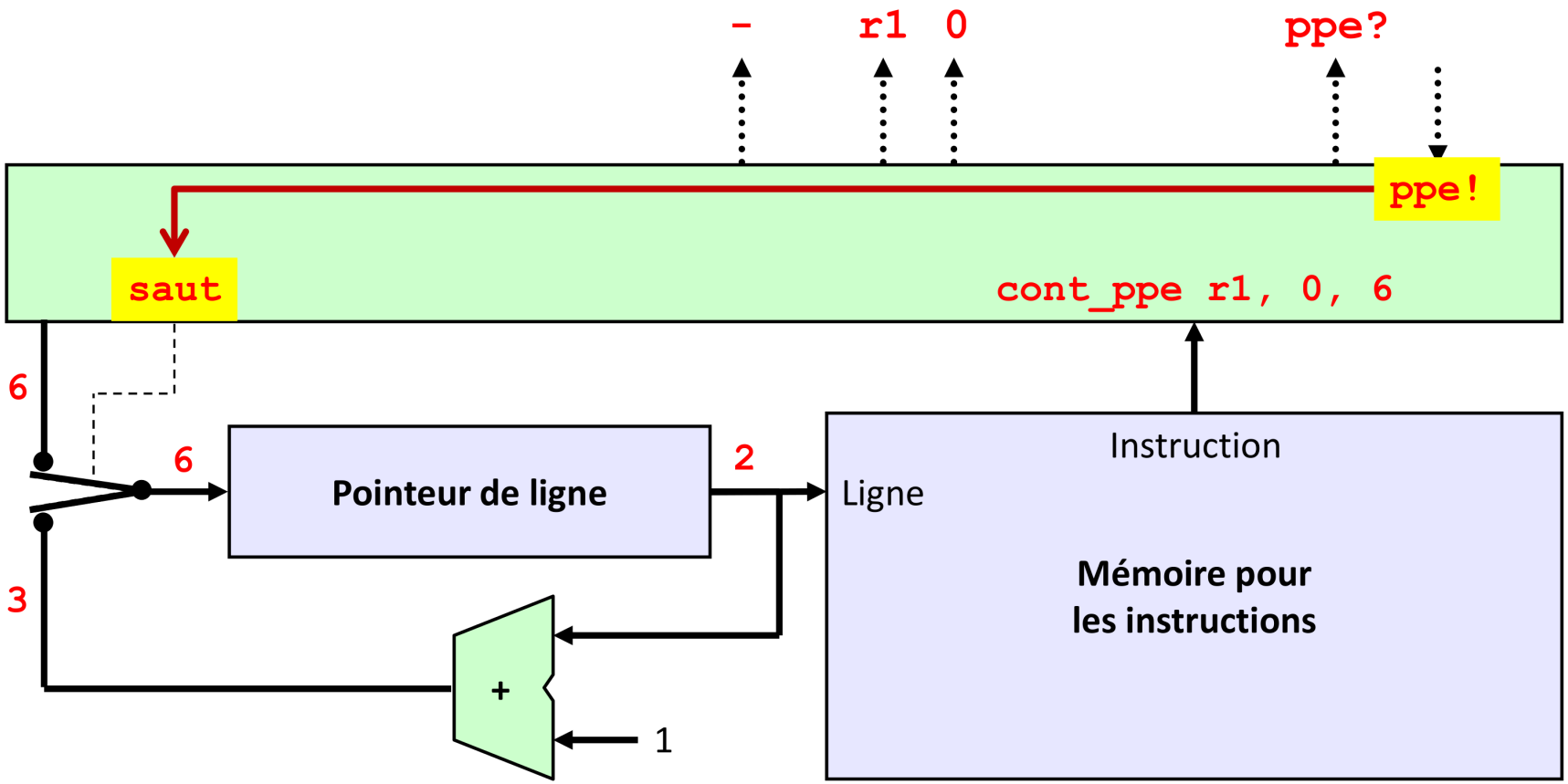
```
5: continue 2
```



# Le circuit pour le contrôle

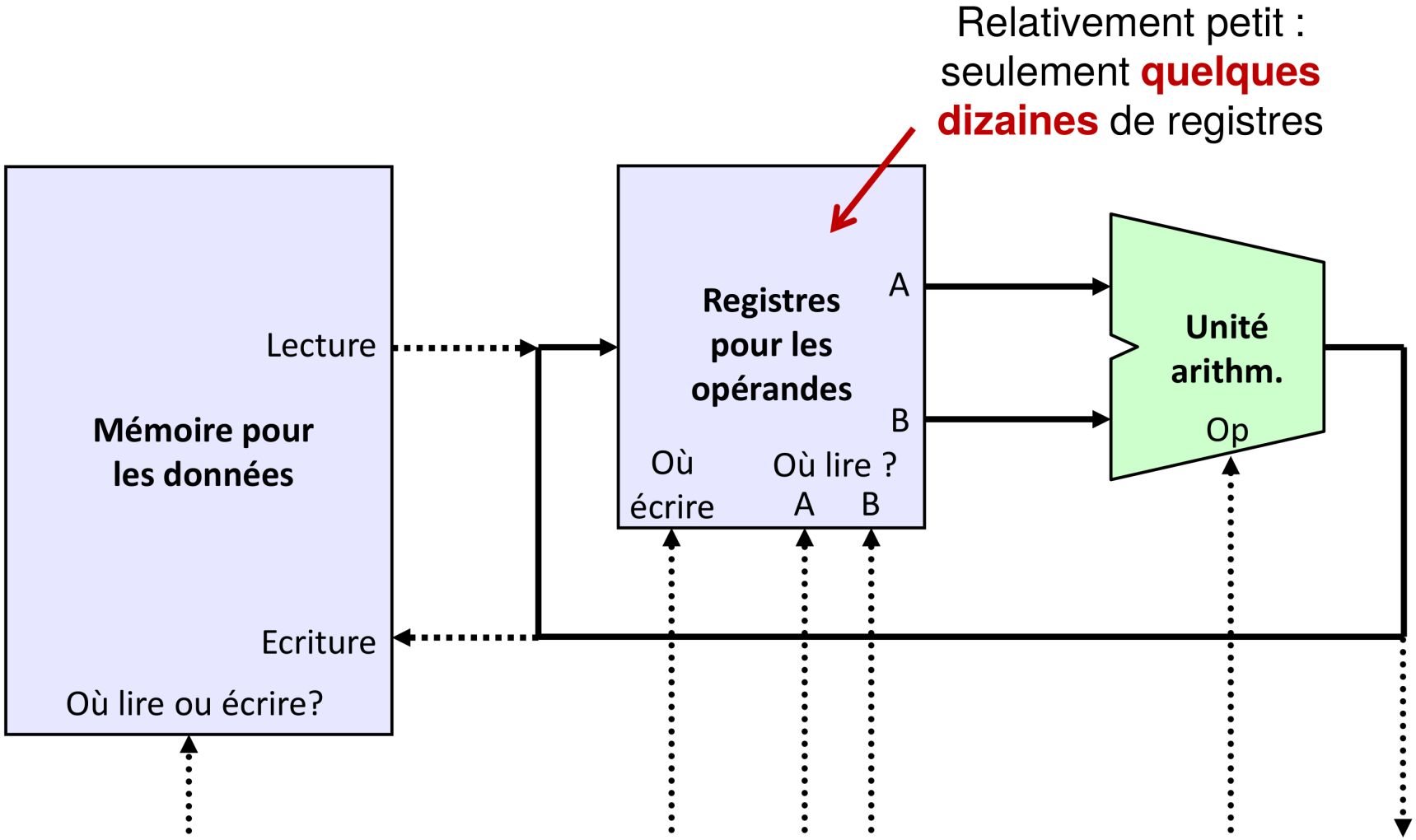


# Le circuit pour le contrôle – en cas de saut

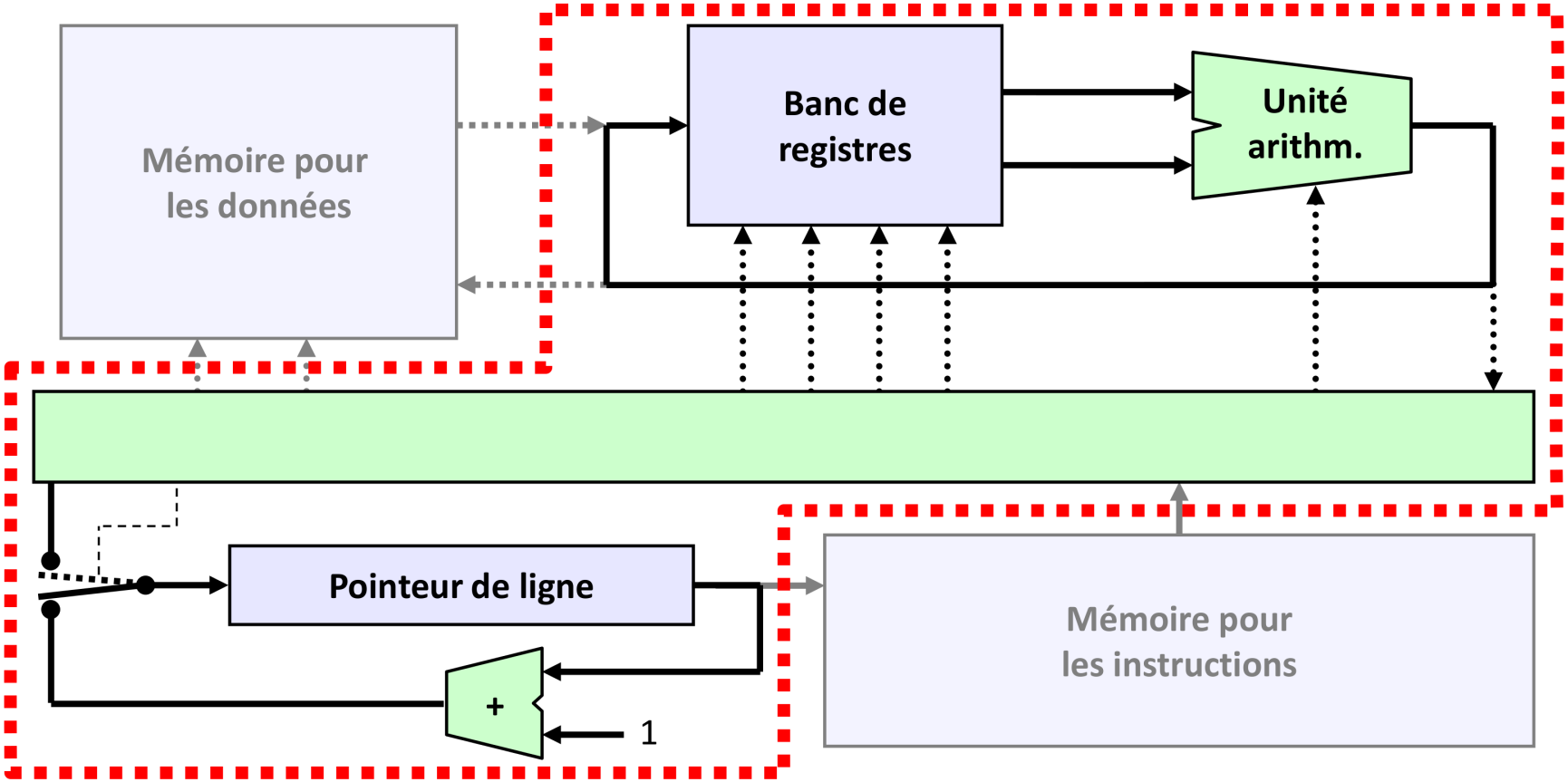


# De quoi a-t-on encore besoin ?

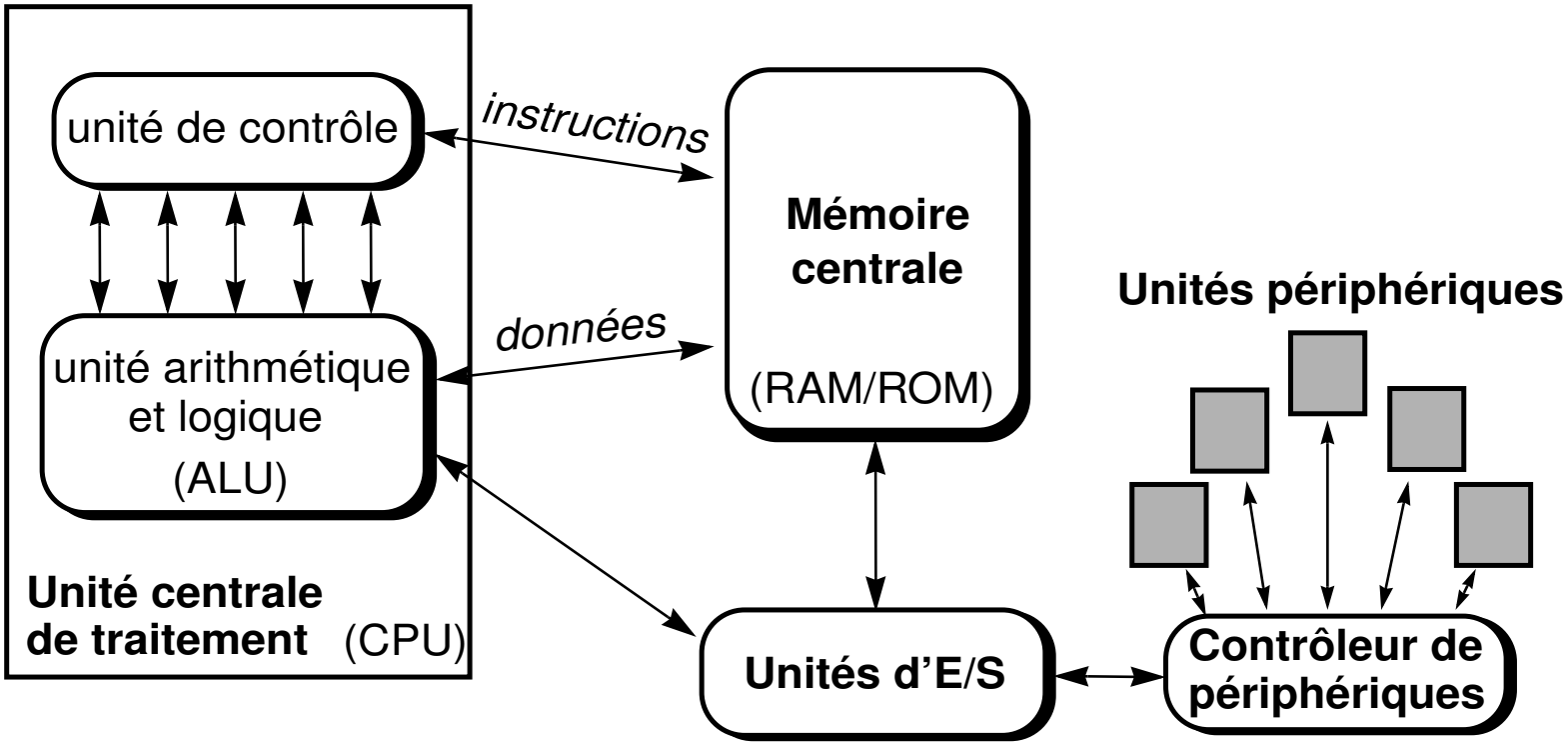
- d'une mémoire pour avoir plus de données



# Un processeur !

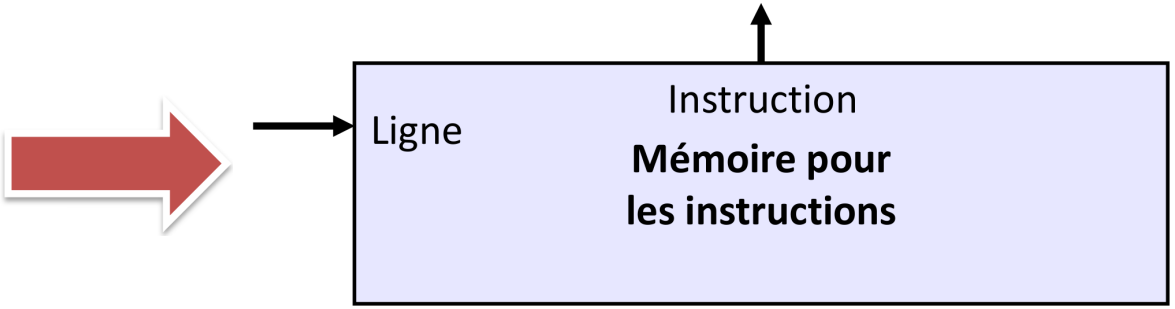


# Lien avec l'architecture de von Neumann



# Comment encoder les instructions ?

```
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
```




On peut inventer un encodage simple (voir leçon I.4) :

- quelques bits pour identifier l'*instruction*
  p.ex. 8 bits si on a moins de 256 instructions
- quelques bits pour identifier les *opérandes* :
  *registres* ou *constantes* ou *adresses* de lignes
  p.ex. 5 bits pour les registres ( $\Longrightarrow$ 32 registres max.)

# Comment encoder les instructions ?

Au final chaque ligne du programme en assembleur peut être codée sur typiquement 32 ou 64 bits (alignement avec les « mots mémoire »)

*Par exemple* « `somme r3, r3, r1` » pourrait être représenté sur 32 bits comme :

```
00010010000110001100001000000000
```

(compris comme :

```
00010010                00011   00011    00001    00000000
somme (avec 3 registres)   3       3        1    (inutilisé)
```

)

# Encoder les instructions

```
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
```

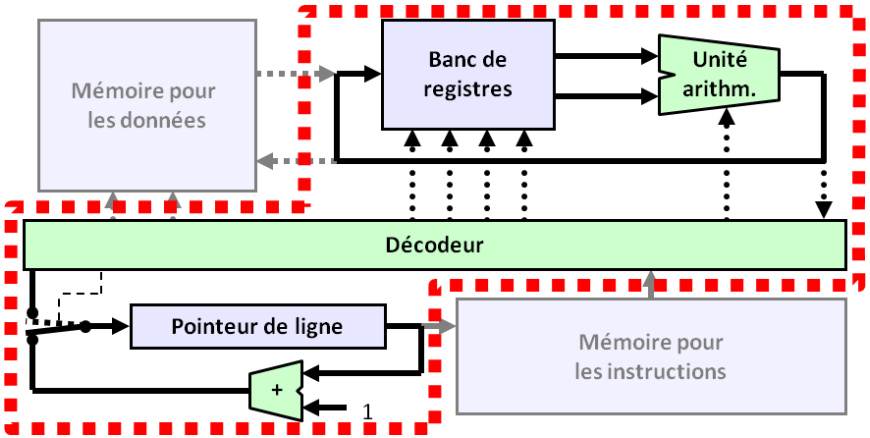
Langage **assembleur**

```
1: 00000001000110000000000000000000
2: 00001011000010000000000000000110
3: 00010010000110001100001000000000
4: 00010011000010000111111111111111
5: 00001111000000000000000000000010
6: 00000010000100001100000000000000
```

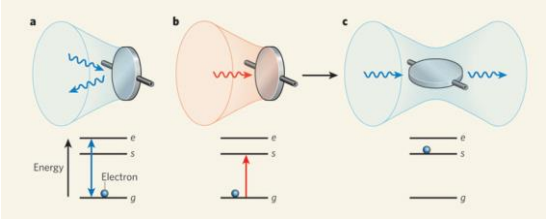
Langage **machine** (binaire)

# 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

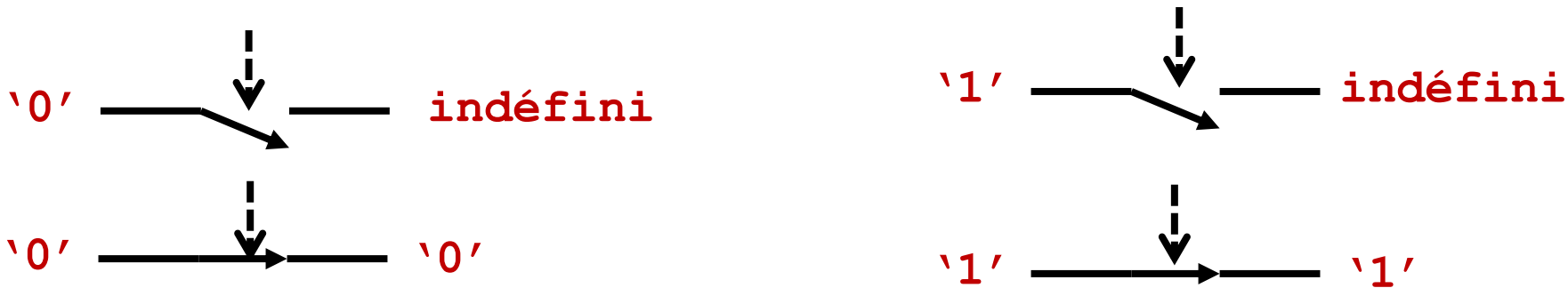



# Un interrupteur

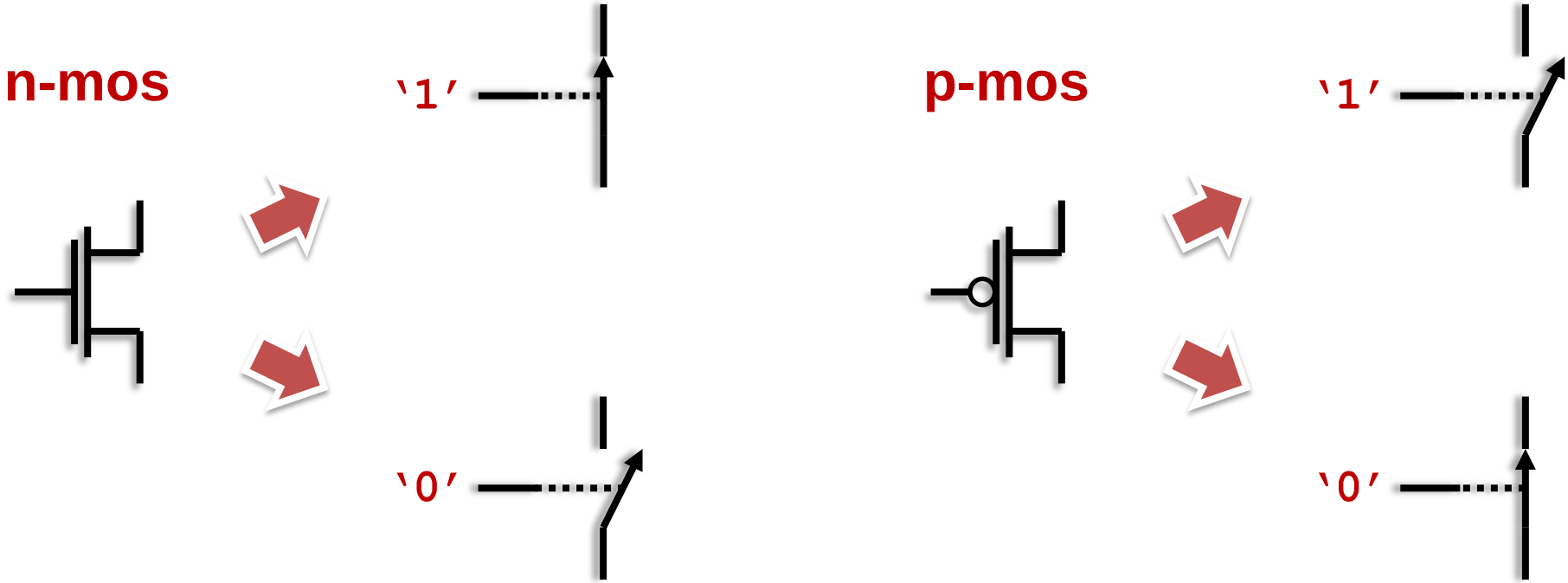
Ne propage rien s'il est ouvert


Propage son entrée s'il est fermé

# Un transistor = un interrupteur contrôlé



# Les transistors, des interrupteurs contrôlés

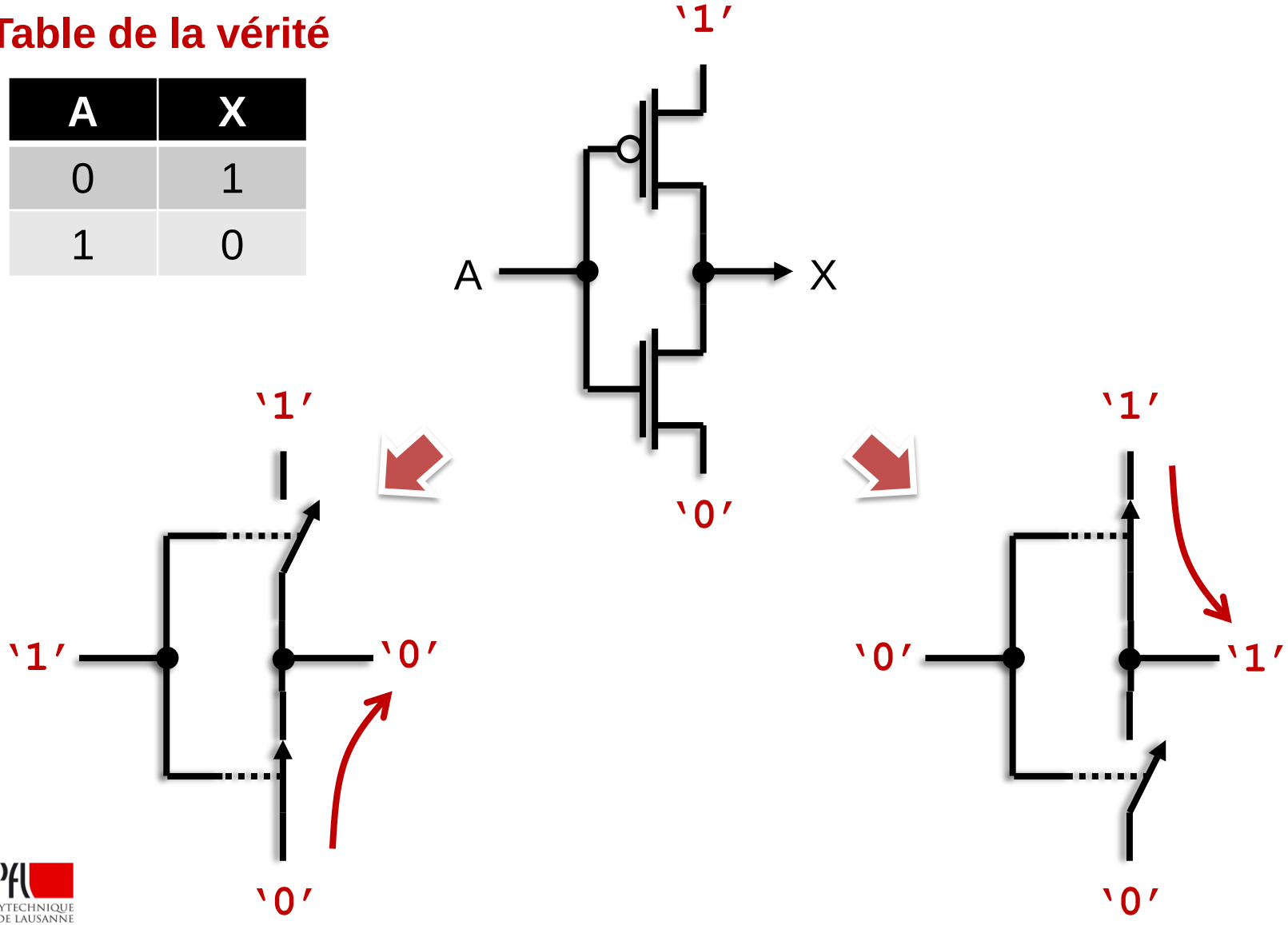



- Ils ne coûtent presque rien : un transistor pour faire un processeur moderne coûte entre $10^{-5}$ et $10^{-4}$ centimes (CHF, USD, EUR…)

# Un inverseur (en CMOS)

## Table de la vérité

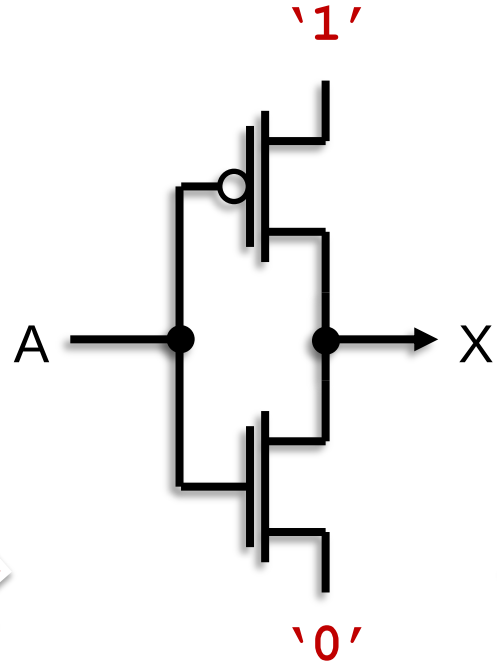
| A | X |
|---|---|
| 0 | 1 |
| 1 | 0 |

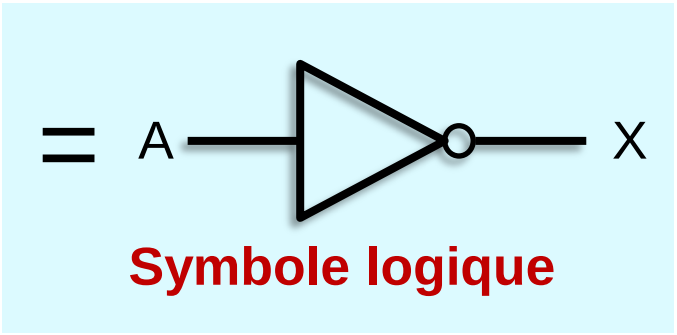


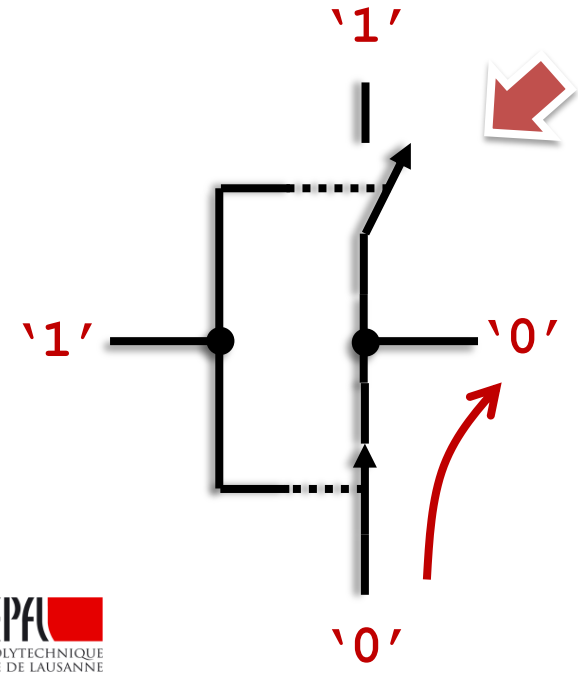
# Un inverseur (en CMOS)

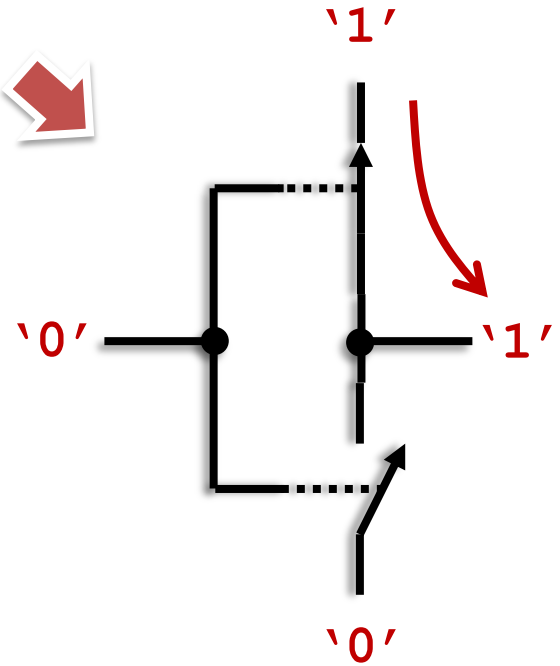
**Table de la vérité**

| A | X |
|---|---|
| 0 | 1 |
| 1 | 0 |

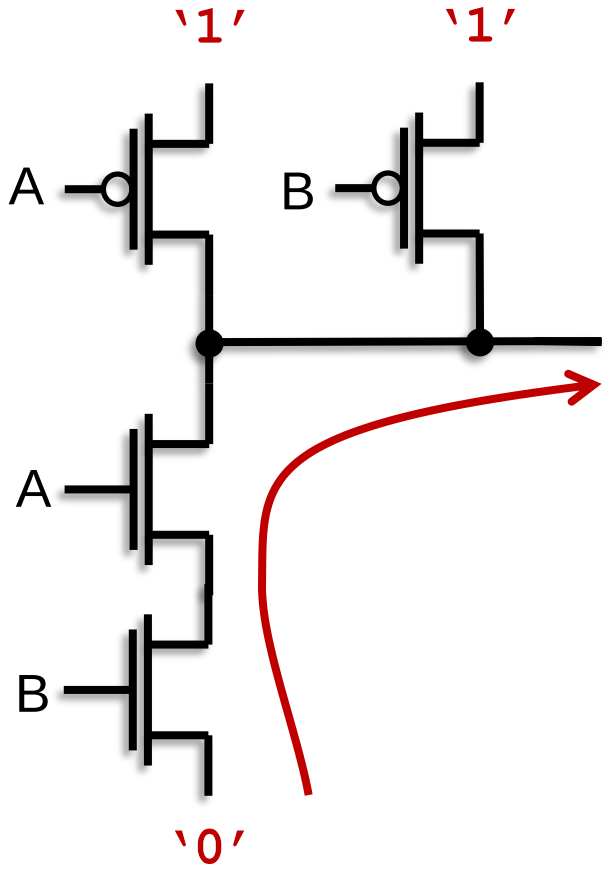


# Un circuit « et » avec sortie inversée




| A | B | A et B inversé |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

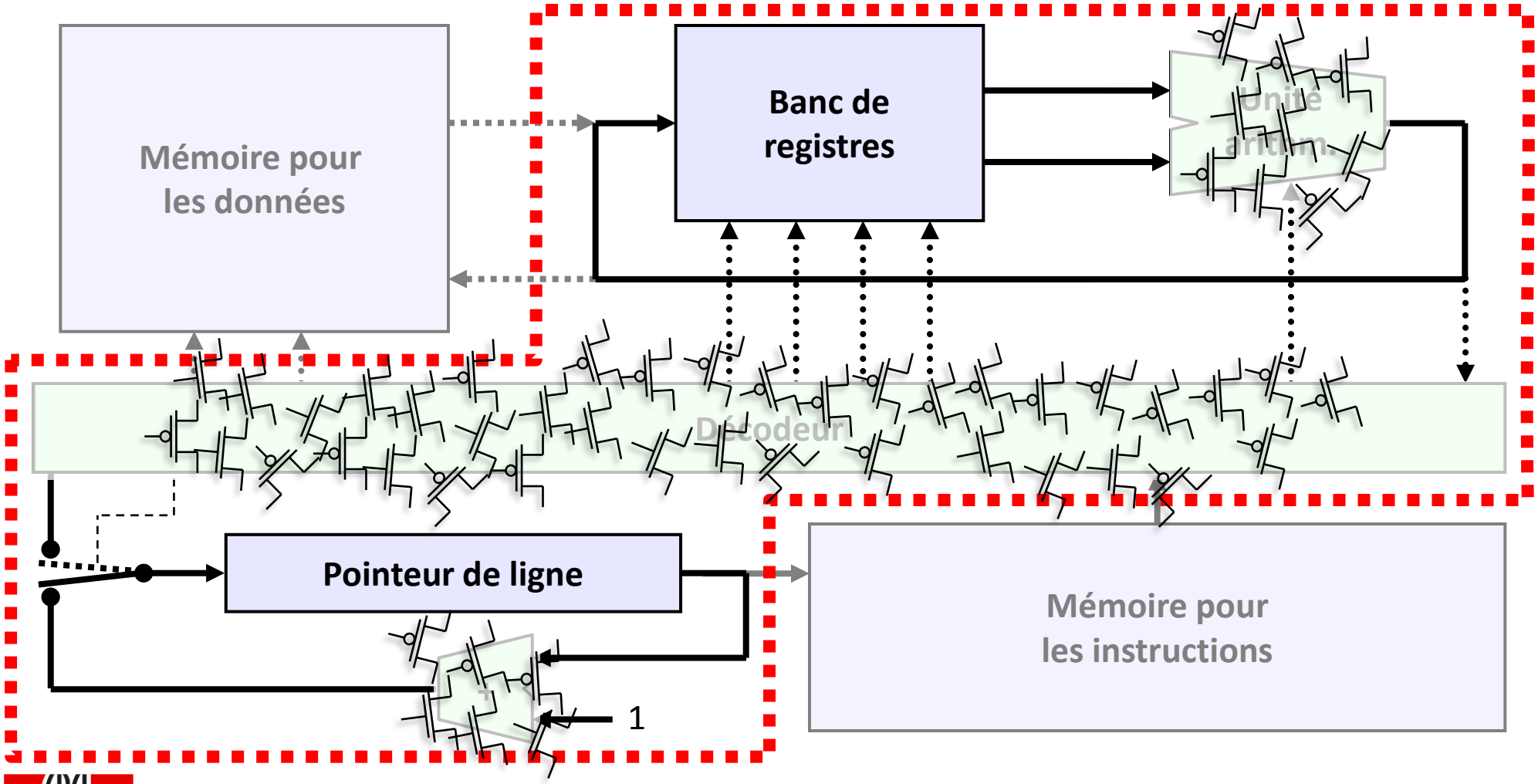
- La seule façon d'obtenir un `'0'` est de mettre deux `'1'` aux entrées A et B : la sortie est à `'0'` seulement si A **et** B sont à `'1'`

# La table de vérité d'un circuit

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

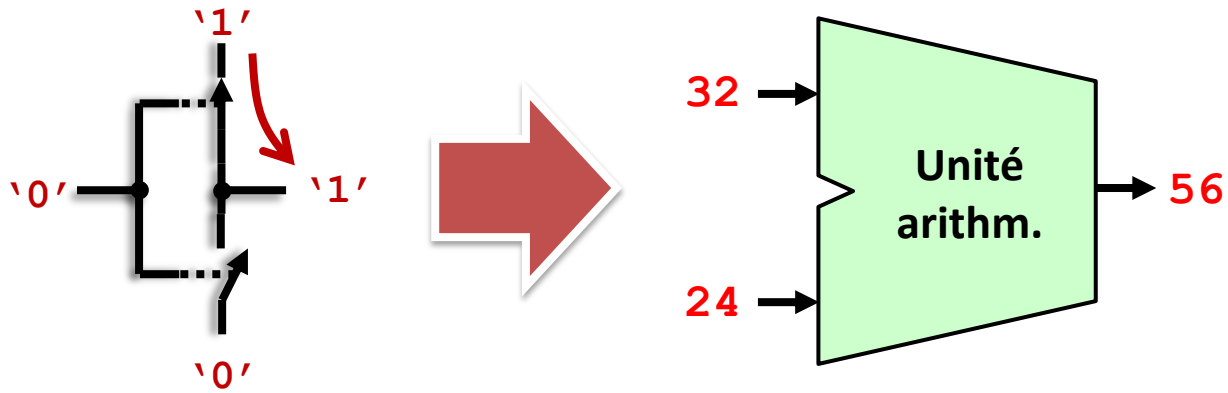
| entrée 1 | entrée 2 | sortie |
|---|---|---|
| 0 | 0 | ? |
| 0 | 1 | ? |
| 1 | 0 | ? |
| 1 | 1 | ? |

# Et notre processeur, donc ?

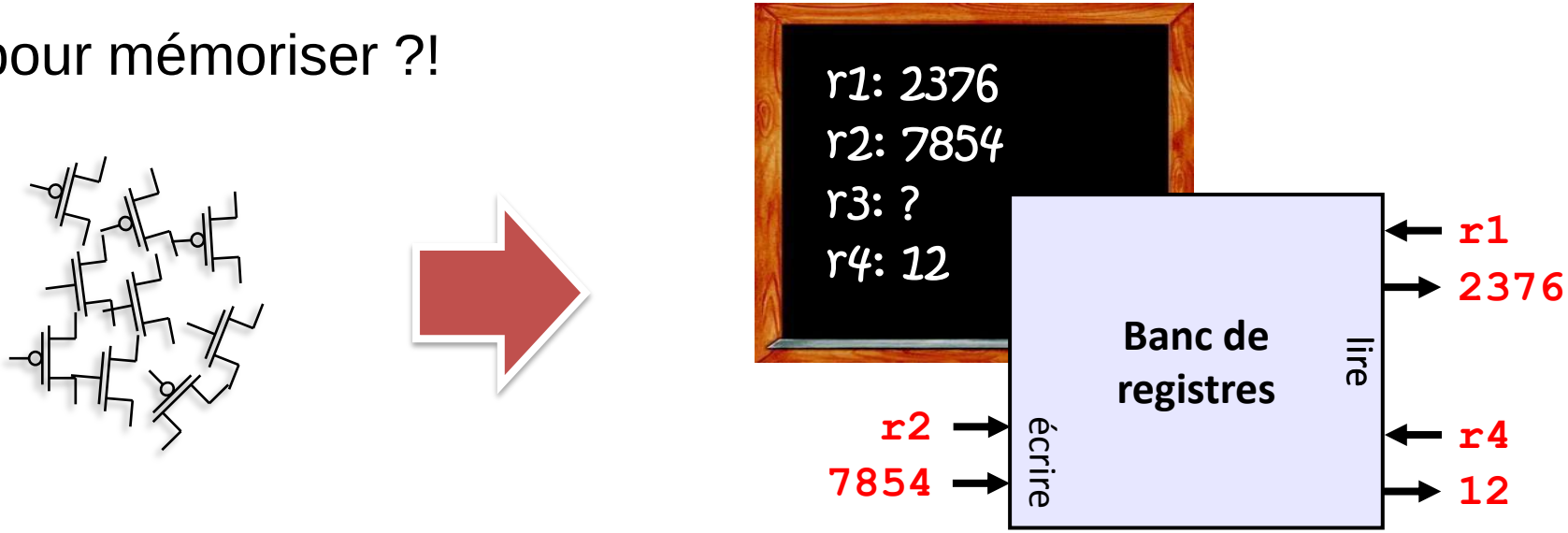


# Peut-on aussi mémoriser l'information ?

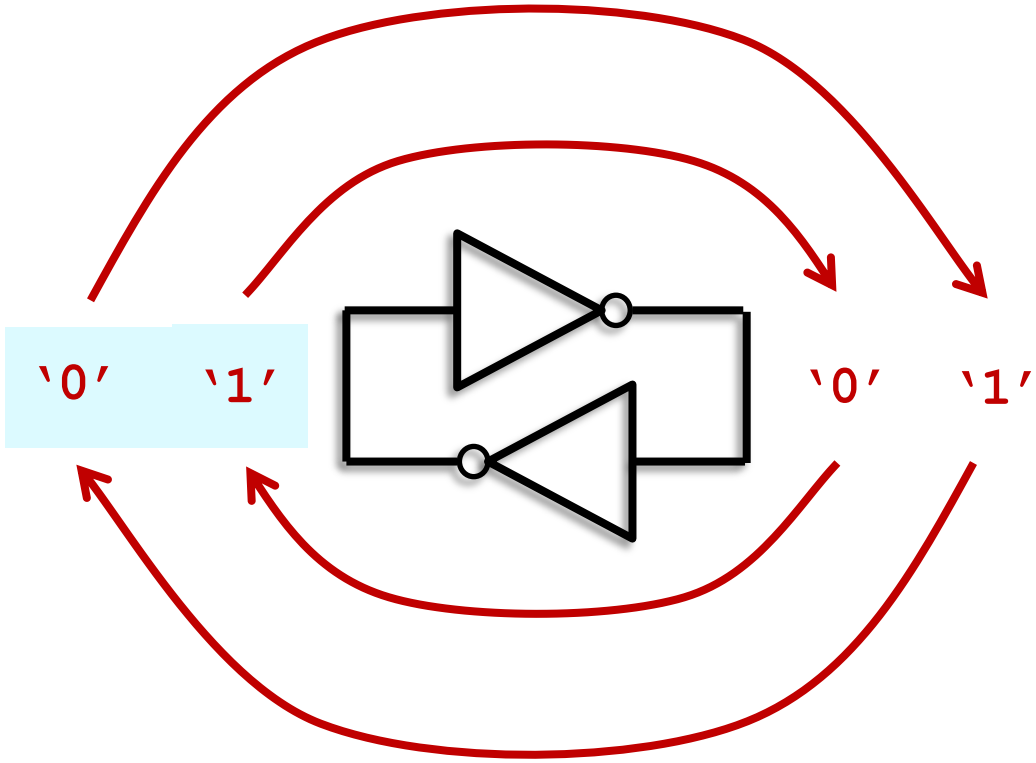
- Pour les calculs, tout va bien :



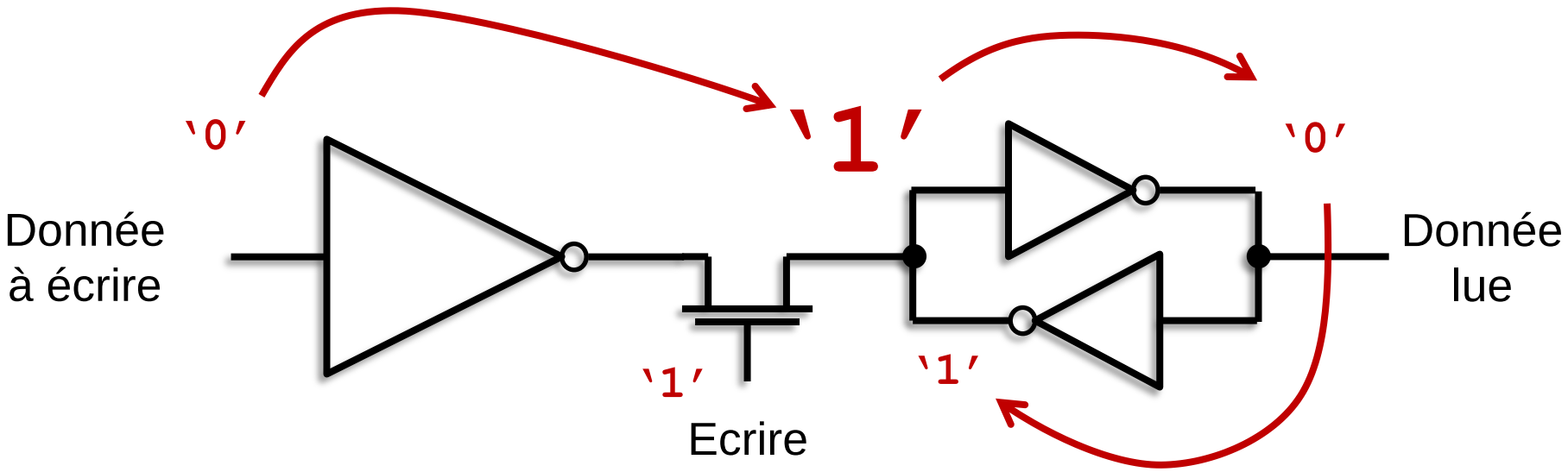

- Mais pour mémoriser ?!



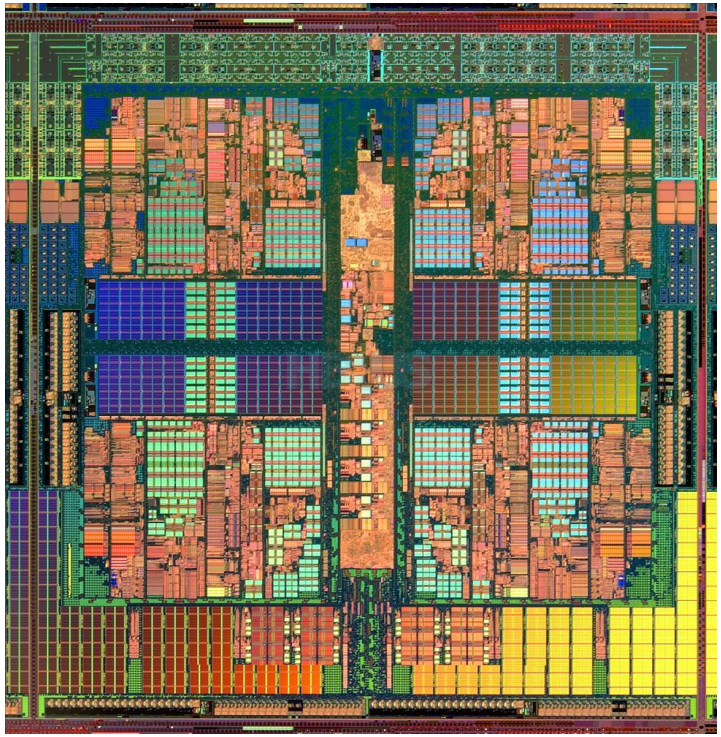
# Un circuit assez particulier



# Comment écrire cette mémoire ?

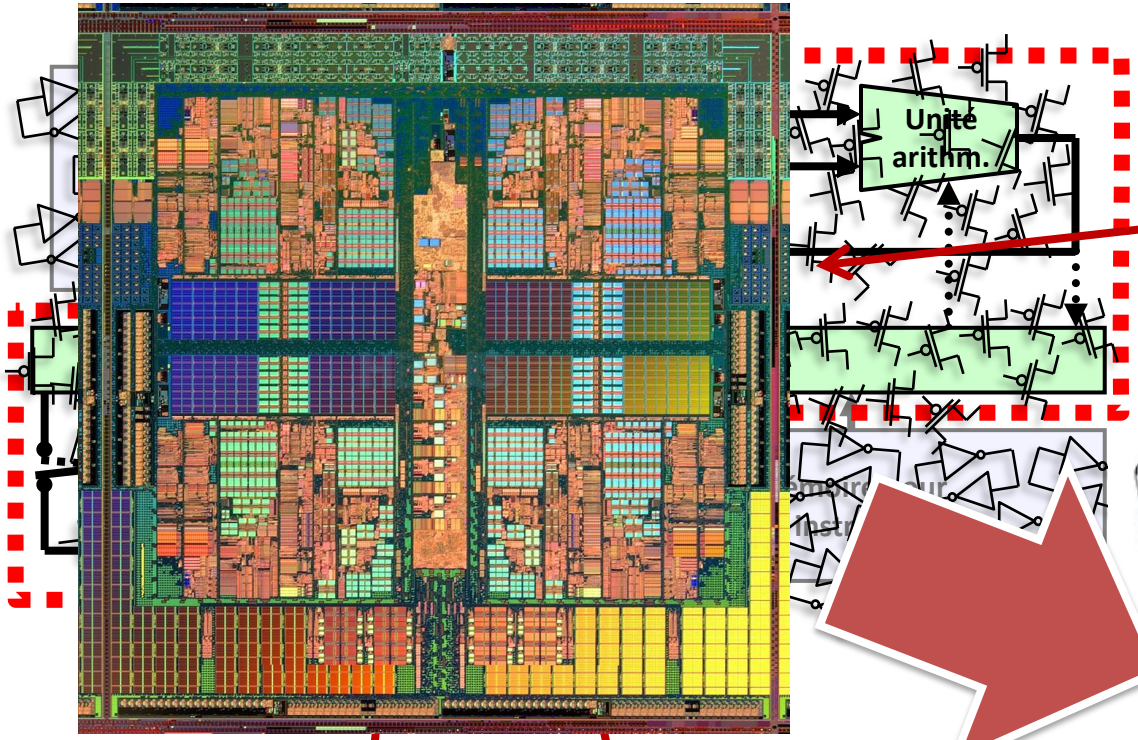


# On a atteint notre but !



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

# On a atteint notre but !

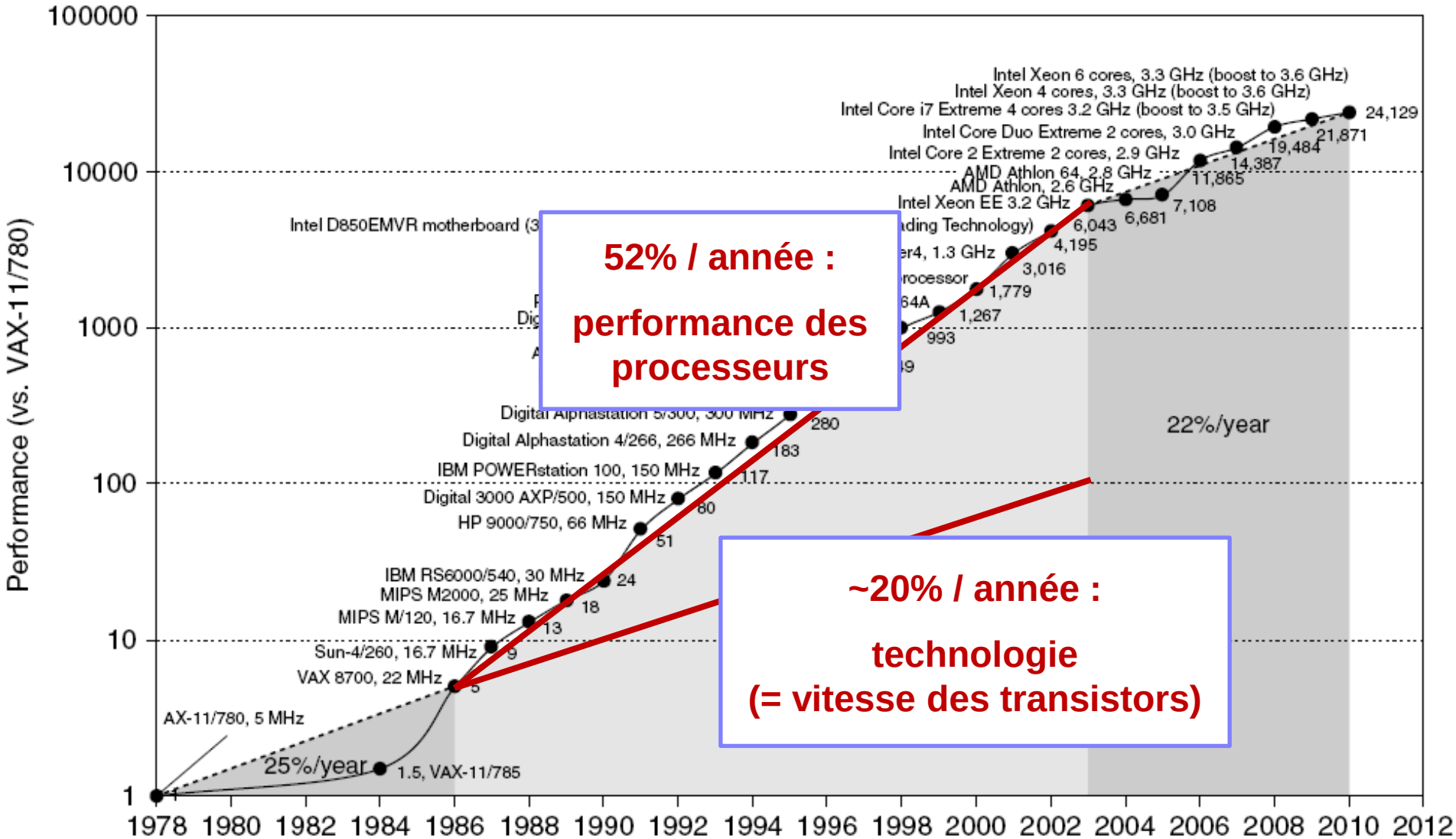



**Circuit intégré VLSI (aujourd'hui autour de $10^8$-$10^9$ transistors)**

**Architecture du processeur**

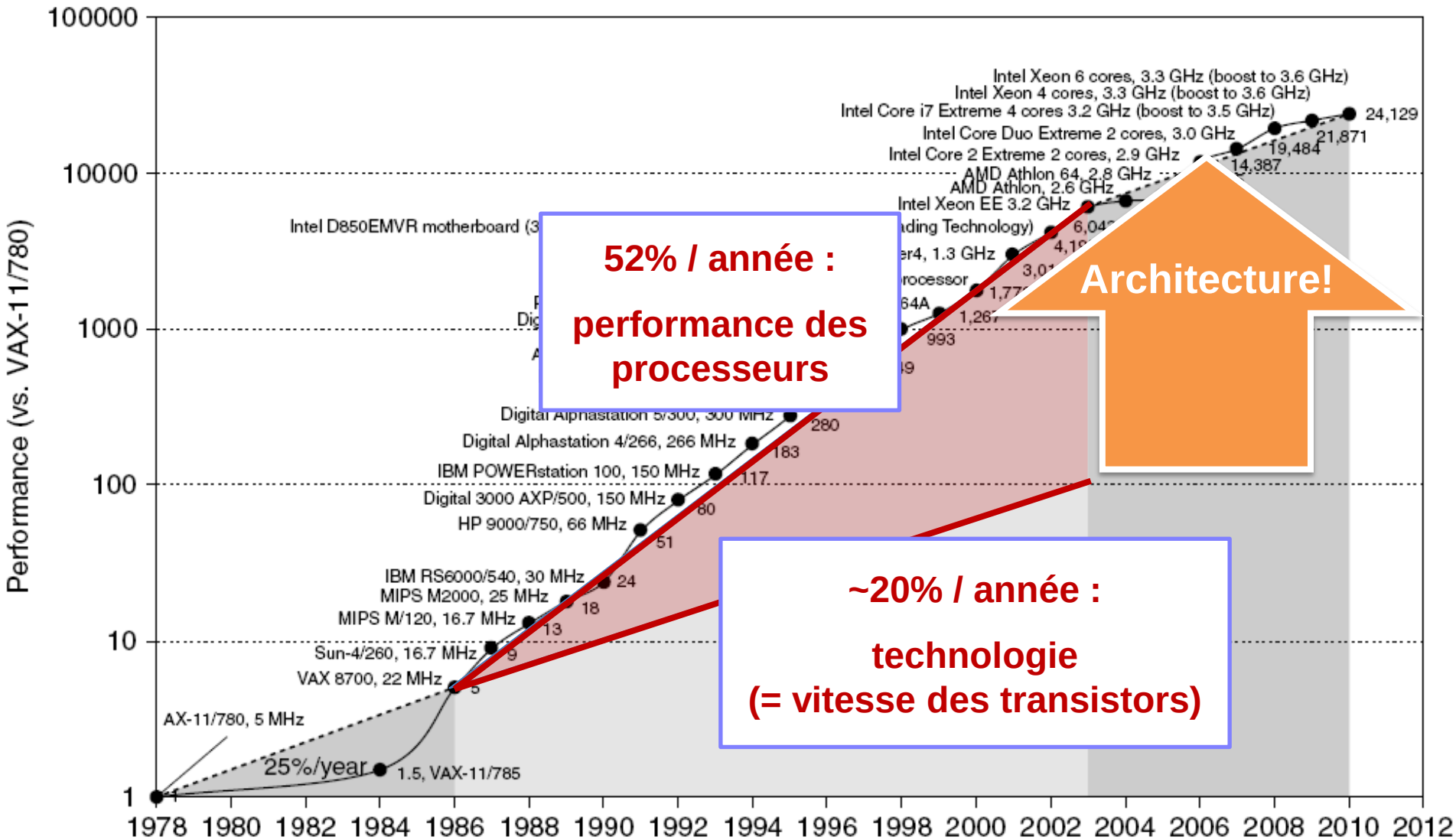
**Circuit électronique**

# La croissance de la performance




Source: Hennessy & Patterson, © MK 2011

# La croissance de la performance




Source: Hennessy & Patterson, © MK 2011

# Faire des sommes est facile…

| | | |
|---|---|---|
| A | 0 1 1 1 0 1 0 1 0 1 1 0 0 0 1 1 0 1 0 | + |
| B | 1 0 1 1 1 0 0 0 1 0 1 1 1 0 0 1 0 1 1 | = |
| | 1 1 | |
| | 0 1 0 1 | |

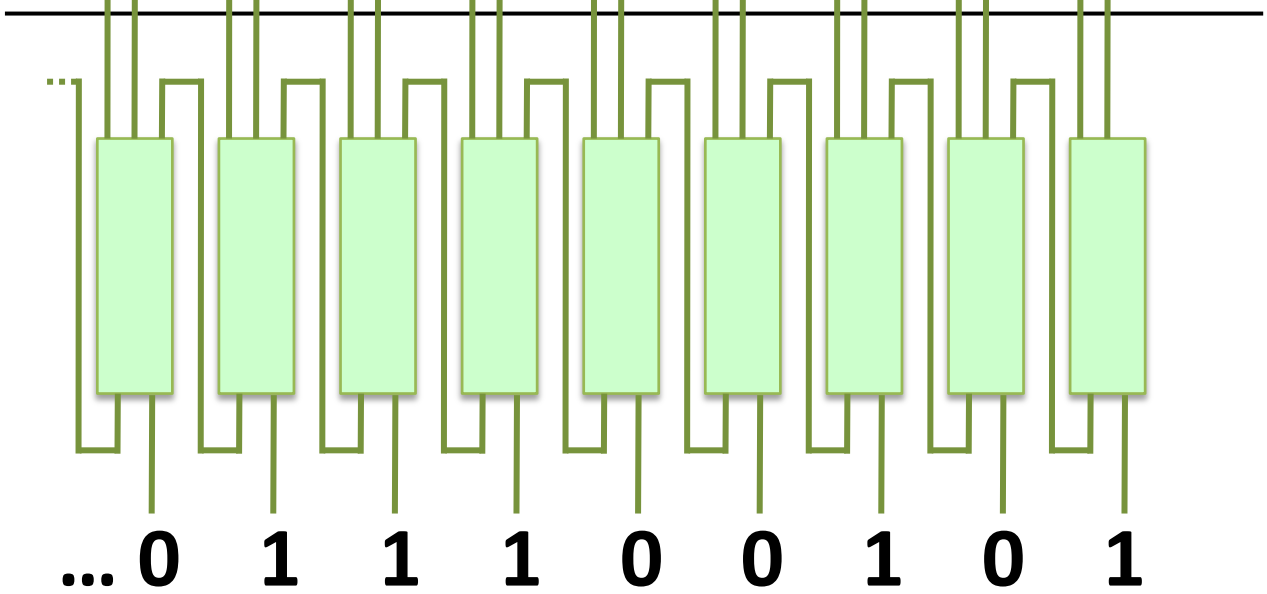
Retenue

**Sommes élémentaires**

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

# Faire un circuit est aussi (relativement) facile…

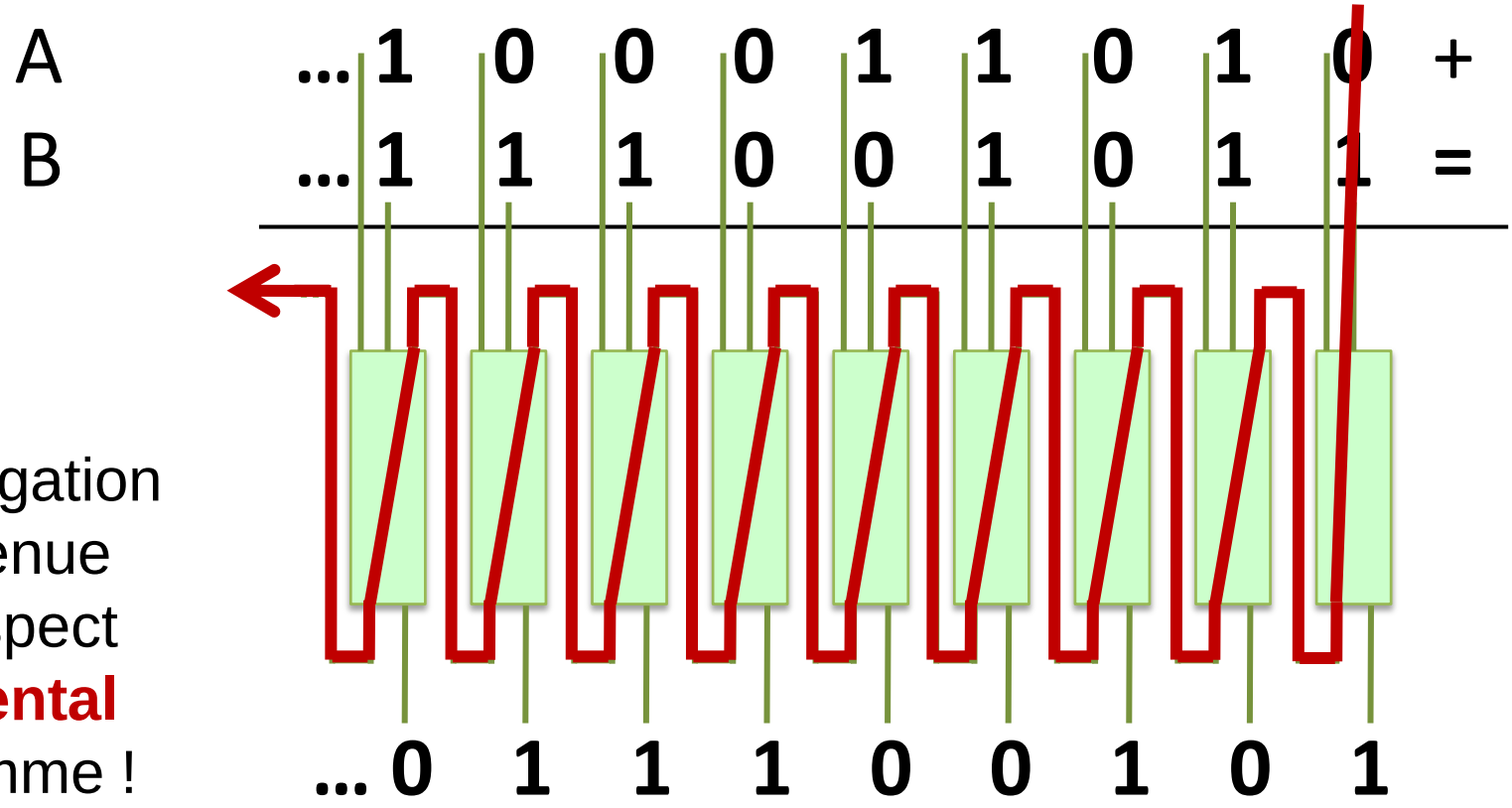
| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| A | … | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | + |
| B | … | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | = |




**Somme**

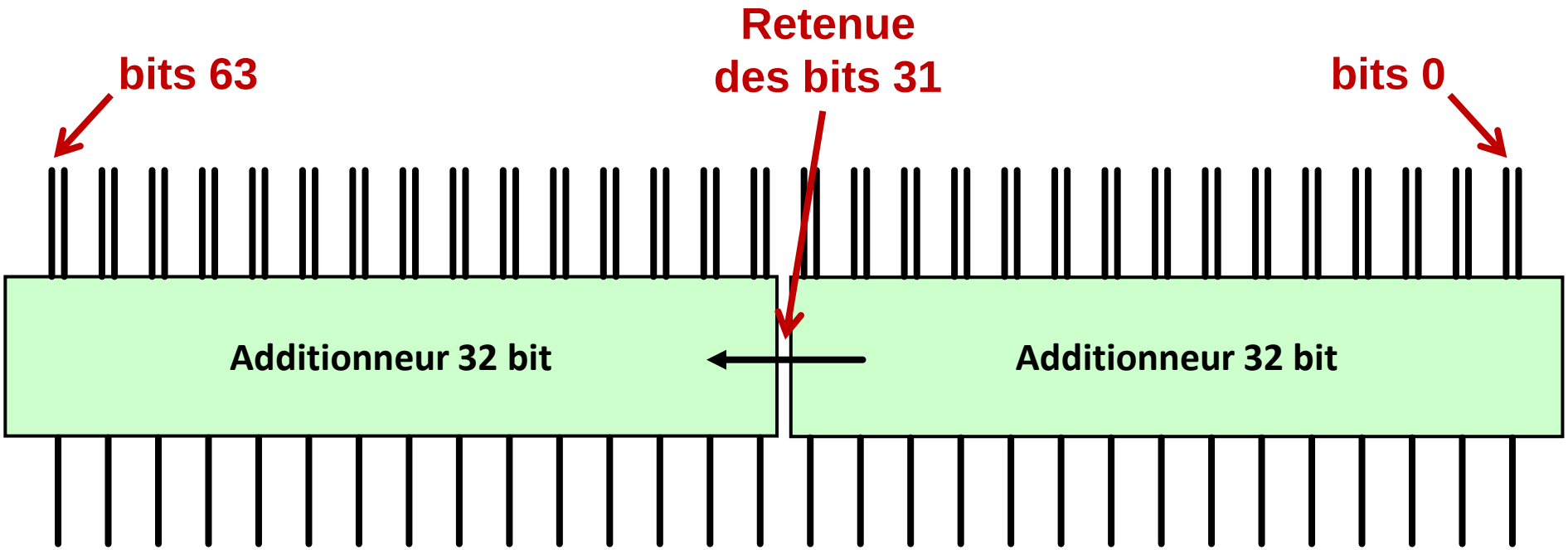
0 + 0 + 0 = 0
0 + 0 + 1 = 1
0 + 1 + 0 = 1
0 + 1 + 1 = 10
1 + 0 + 0 = 1
1 + 0 + 1 = 10
1 + 1 + 0 = 10
1 + 1 + 1 = 11

# Mais ce circuit est lent !

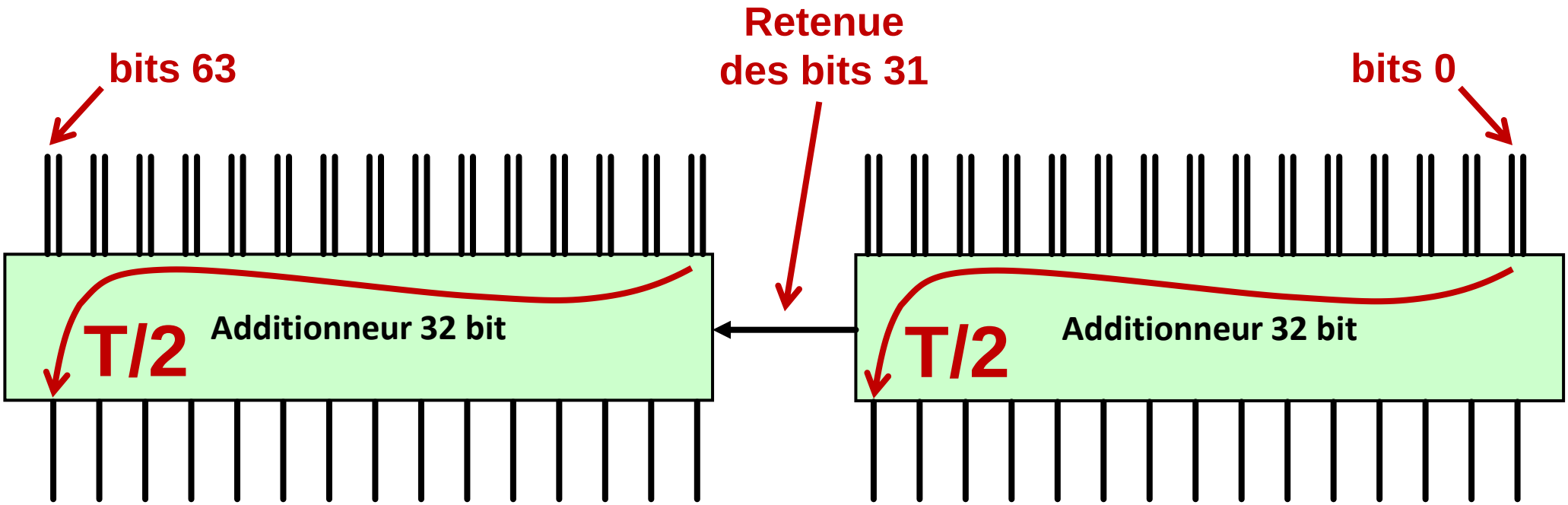



- La propagation de la retenue est un aspect **fondamental** de la somme !

# Peut-on faire mieux ?

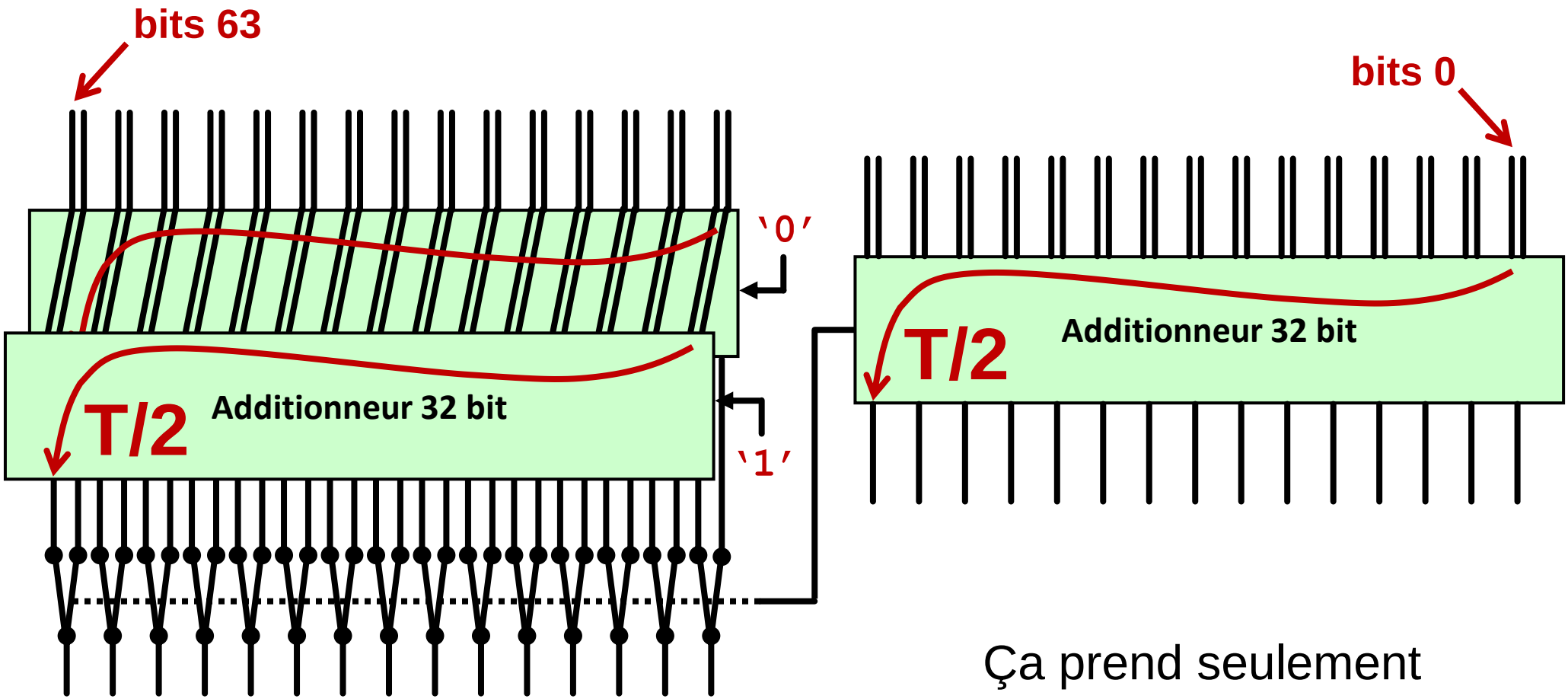


# Peut-on faire mieux ?




On n'a **rien** gagné…

# Peut-on faire mieux ?




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

# Le génie informatique (1)

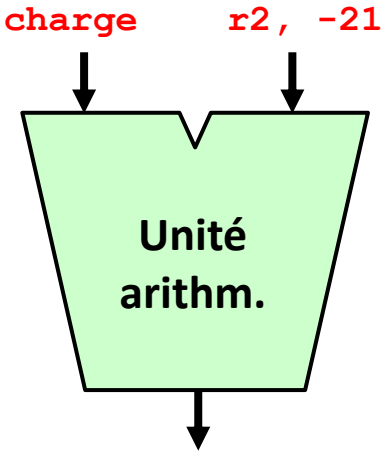
- On peut profondément changer la performance du circuit
  sans en changer la fonctionnalité
- On peut investir plus de transistors et plus d'énergie
  pour obtenir des circuits très rapides
- On peut ralentir les circuits
  pour épargner de l'énergie

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

# Notre processeur

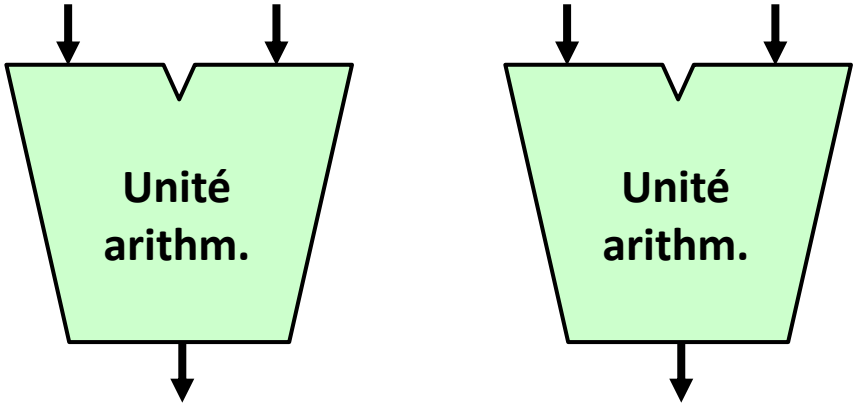
```
103: charge    r1, 0
104: charge    r2, -21
105: somme     r3, r7, r4
106: multiplie r2, r5, r9
107: soustrais r8, r7, r9
108: charge    r9, r4
109: somme     r3, r2, r1
110: soustrais r5, r3, r4
111: charge    r2, r3
112: somme     r1, r2, -1
113: somme     r8, r1, -1
114: divise    r4, r1, r7
115: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

On exécute approximativement **une instruction** par cycle



# 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
```

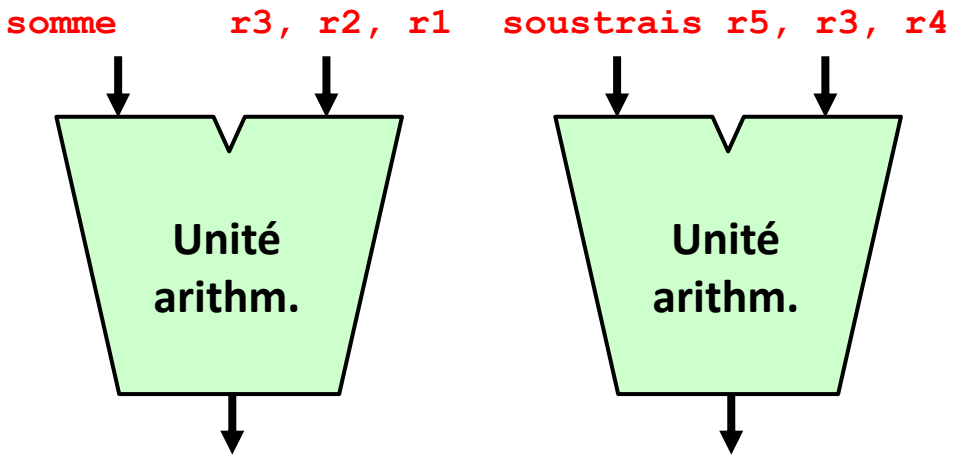


# Doubler le débit de notre processeur

```
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 **deux instructions** par cycle !

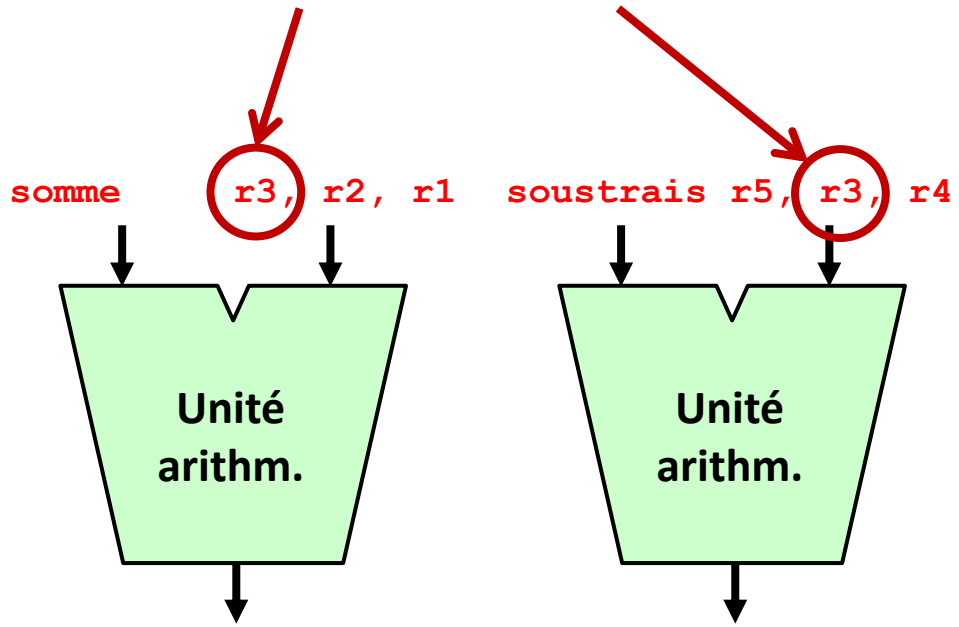
**Problèmes ?!**



# Doubler le débit de notre processeur

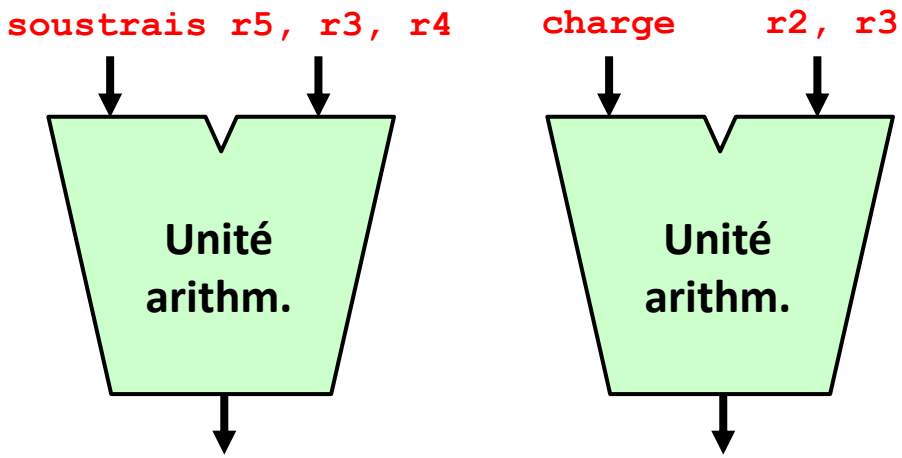
```
103: charge    r1, 0
104: charge    r2, -21
105: somme     r3, r7, r4
106: multiplie r2, r5, r9
107: soustrais r8, r7, r9
108: charge    r9, r4
109: somme     r3, r2, r1
110: soustrais r5, r3, r4
111: charge    r2, r3
112: somme     r1, r2, -1
113: somme     r8, r1, -1
114: divise    r4, r1, r7
115: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

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



# Doubler le débit de notre processeur

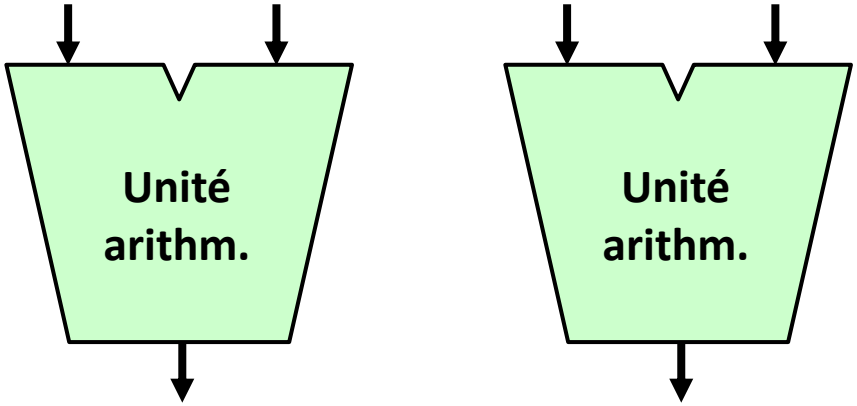
```
103: charge    r1, 0
104: charge    r2, -21
105: somme     r3, r7, r4
106: multiplie r2, r5, r9
107: soustrais r8, r7, r9
108: charge    r9, r4
109: somme     r3, r2, r1
110: soustrais r5, r3, r4
111: charge    r2, r3
112: somme     r1, r2, -1
113: somme     r8, r1, -1
114: divise    r4, r1, r7
115: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



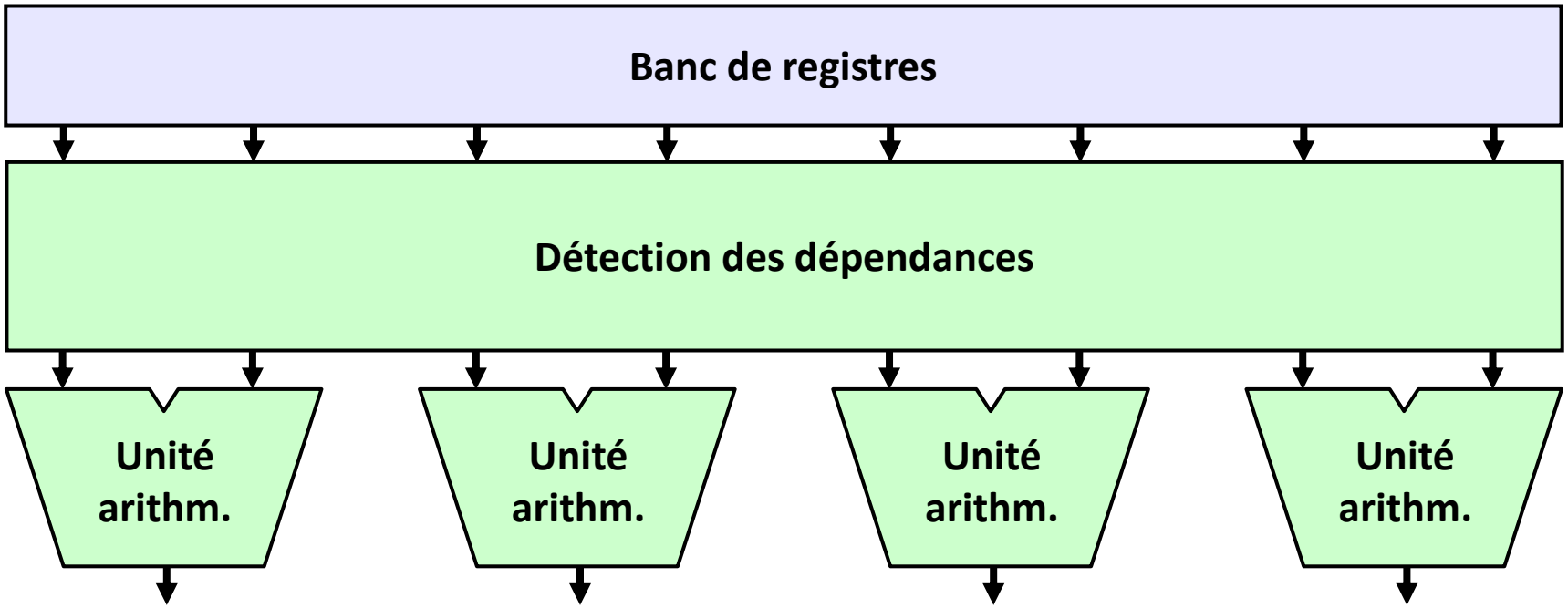
# ~~Doubler~~ le débit de notre processeur

```
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**)