



## Information, Calcul et Communication Module 3 : Systèmes

## Information, Calcul et Communication Ordinateur à programme enregistré

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

### La question de ce module

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

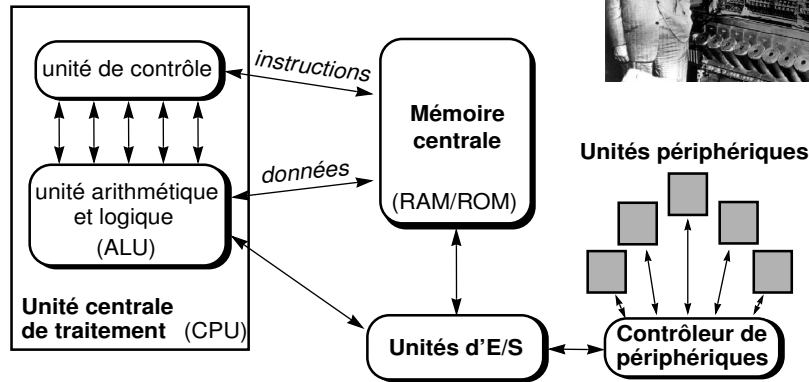
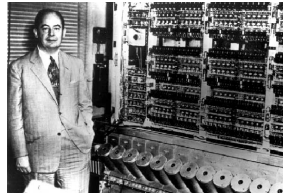
### Les réponses de ce module

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



# 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** ?

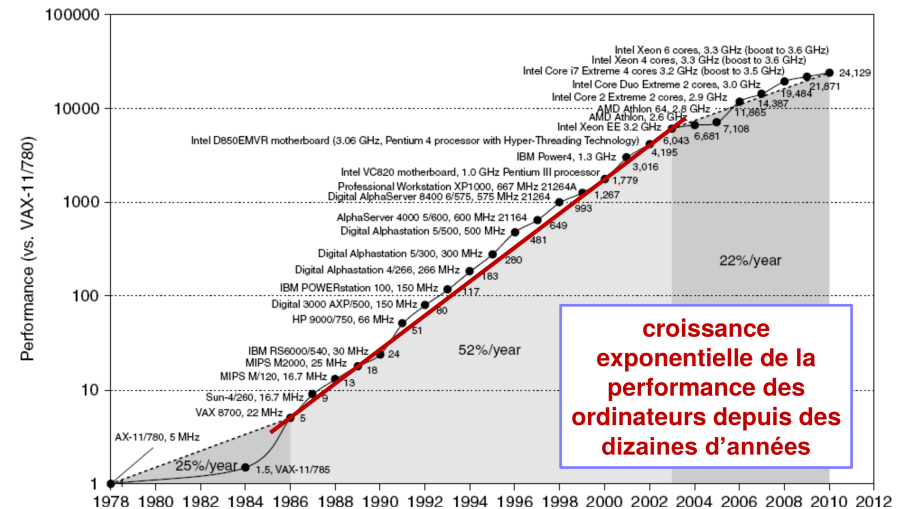
**Algorithme :**

<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 $x_2$

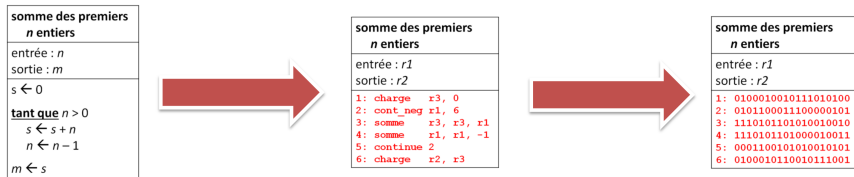


# La deuxième question de cette leçon

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

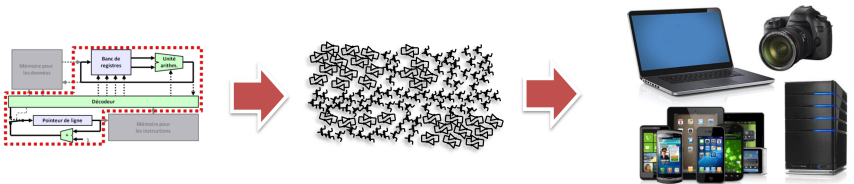


# Des algorithmes aux ordinateurs

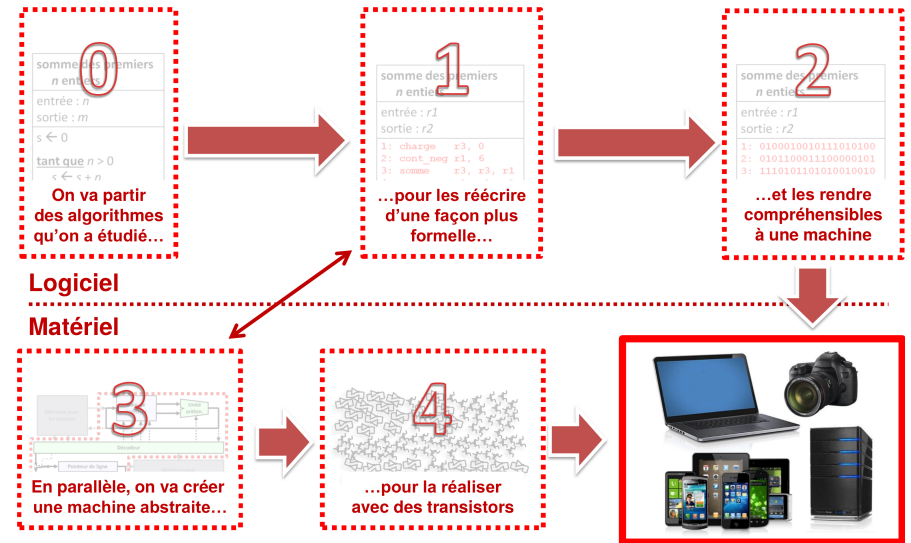


Logiciel

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 :  $n$ 
sortie :  $S(n)$ 

 $s \leftarrow 0$ 
Tant que  $n > 0$ 
|    $s \leftarrow s + n$ 
|    $n \leftarrow n - 1$ 
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 :  $r_1, \dots, r_6$ .

```

somme
entrée :  $r_1$ 
sortie :  $m = r_2 = S(r_1)$ 

 $r_3 \leftarrow 0$ 
Tant que  $r_1 > 0$ 
|    $r_3 \leftarrow r_3 + r_1$ 
|    $r_1 \leftarrow r_1 - 1$ 
 $r_2 \leftarrow r_3$ 
    
```

## Les registres

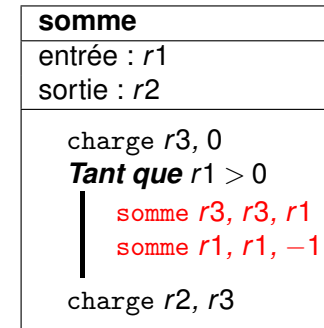
- ▶ On a besoin de mémoriser des valeurs.
- ▶ Ces valeurs seront stockées dans ce qui est appelé des « **registres** » : réalisation concrète dans notre machine de la notion de variable.
- ▶ On les représente simplement par  $r_1, \dots, r_6$ .
- ▶ Nous essayerons d'en avoir qu'un petit nombre limité (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*



## Les opérations ou « instructions »

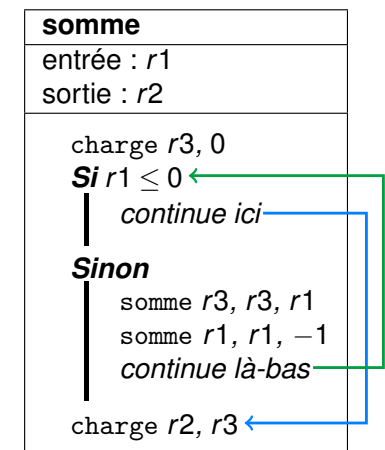
- ▶ On définit un nombre limité d'opérations ; p.ex.
  - ▶ *charge* pour l'assignation ;
  - ▶ *somme* pour l'addition ;
  - ▶ *soustrais* pour la soustraction.
- ▶ Toutes les opérations ont un **résultat** et opèrent sur **une ou deux** valeurs ou **opérandes**, jamais plus.
- ▶ Les opérandes sont soit des (contenus de) **registres**, soit des **constantes**.
- ▶ On écrit ces opérations ainsi :
  - somme destination, operande1, operande2*
  - ▶ Au lieu d'écrire  $s \leftarrow s + n$ , 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*

## 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





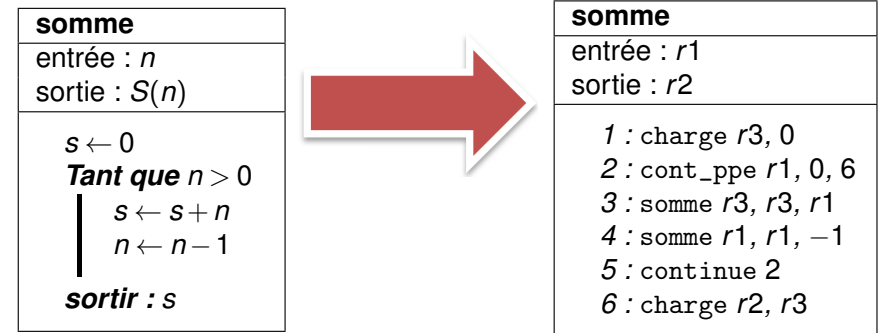
## Numérotation des lignes

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 »



## Exemple réel 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.gnu.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 où continuer dans la séquence (p.ex., `continue`, si cet endroit est toujours le même, ou `cont_TEST`, s'il ne faut y aller que dans certains cas) — on les appelle « **instructions de saut** »

# Essays de créer une telle machine...

Algorithme :

```

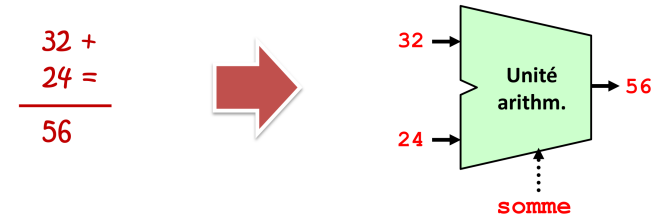
Second degré
entrée : b, c
sortie : {x ∈ ℝ : x² + b x + c = 0}

Δ ← b² - 4 c
Si Δ < 0
  afficher 0
Sinon
  Si Δ = 0
    x ← -b/2
    afficher x
  Sinon
    x₁ ← (-b-√Δ)/2
    x₂ ← (-b+√Δ)/2
    afficher x₁ et x₂
    
```

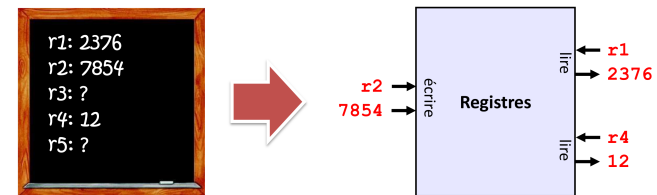


# De quoi a-t-on besoin ?

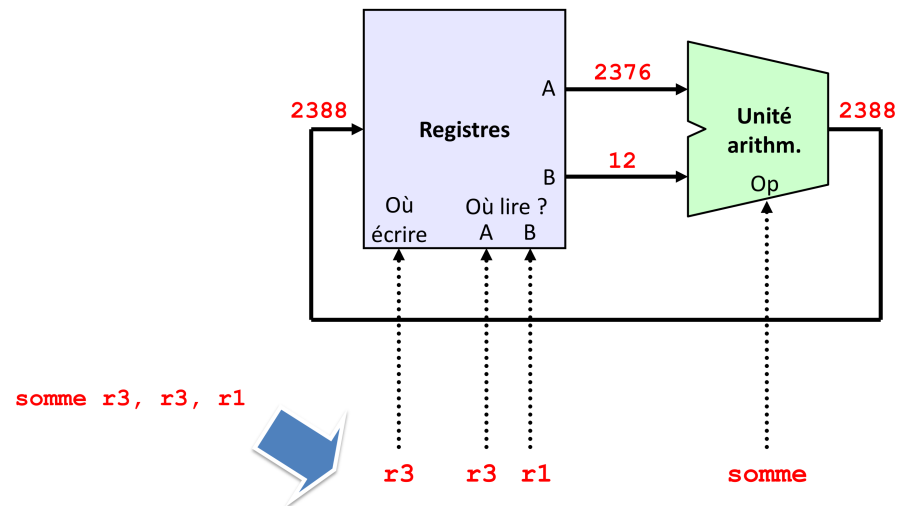
- ▶ L'**unité arithmétique et logique** (ALU) effectue les opérations arithmétiques



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

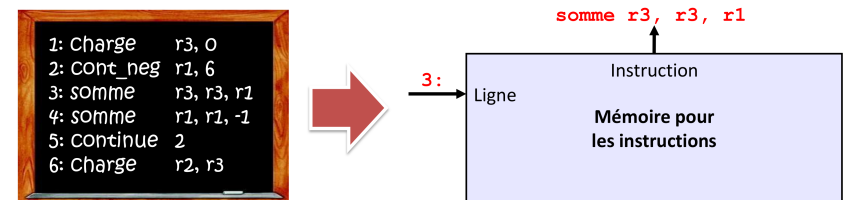


# Le circuit pour les calculs



# De quoi a-t-on encore besoin ?

- ▶ Le programme doit être enregistré quelque part

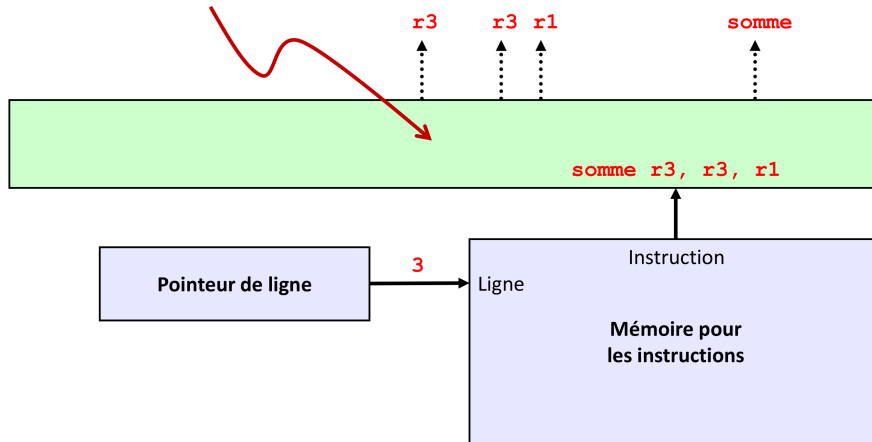


- ▶ Il faut pouvoir contrôler où on en est



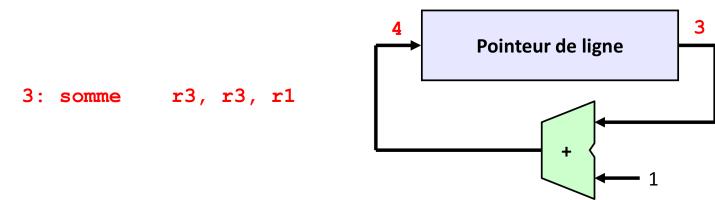
## Une première partie pour contrôler le tout...

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

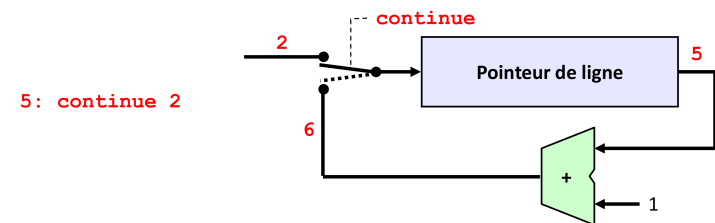


## En cas de saut ?

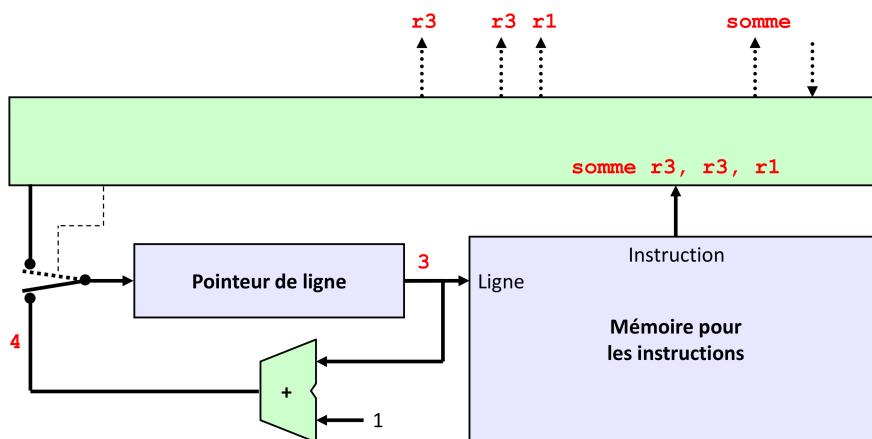
- ▶ La plupart du temps on passe simplement à la ligne suivante



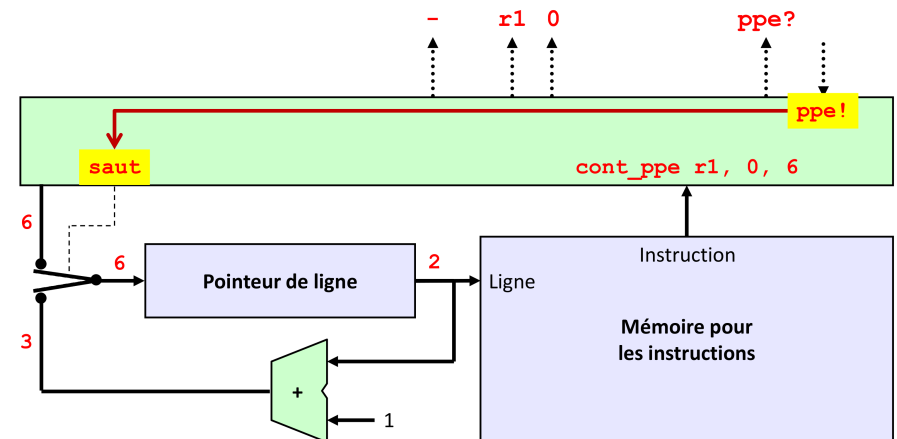
- ▶ Si on a une instruction de saut (p.ex. `continue`), on veut imposer une autre ligne



## Le circuit pour le contrôle

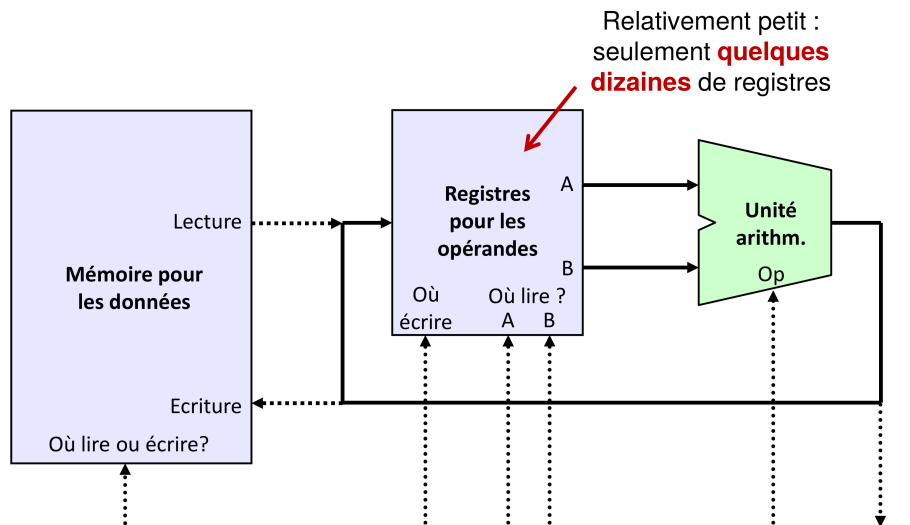


## Le circuit pour le contrôle – en cas de saut

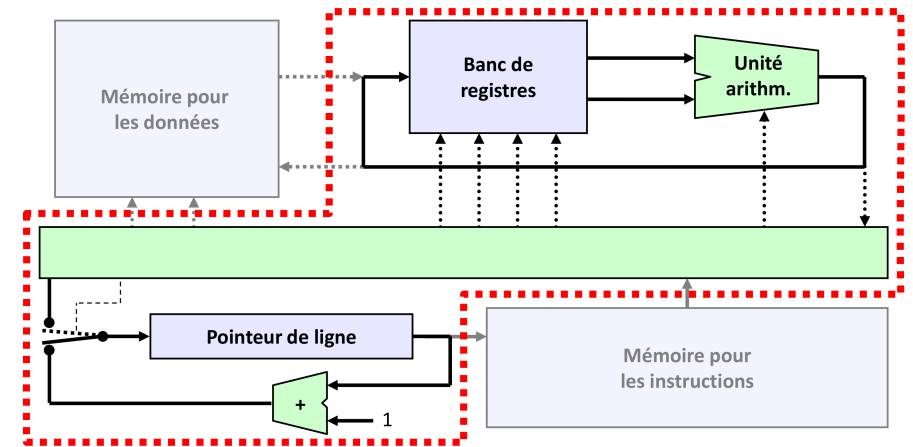


## De quoi a-t-on encore besoin ?

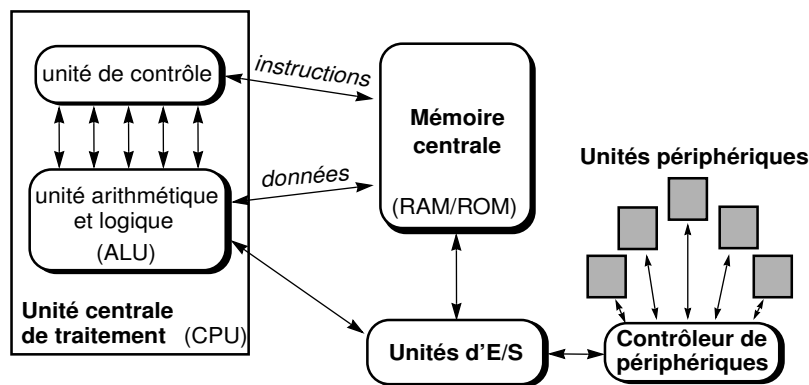
- D'une mémoire pour avoir plus de données :



## Un processeur ! (CPU)



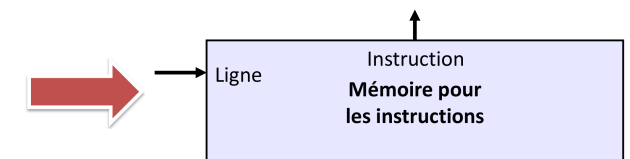
## Lien avec l'architecture de von Neumann



## Comment encoder les instructions ?

```

1: charge r3, 0
2: cont_ppc 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 (  $\implies$  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 représentée sur 32 bits comme :

0001001000011000110000100000000

(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



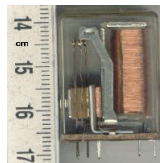
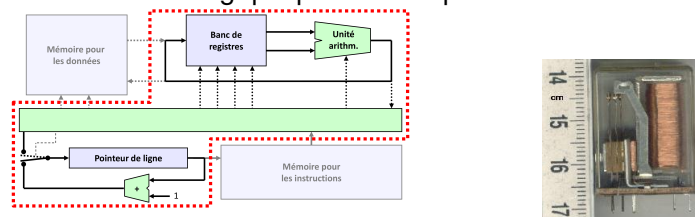
1: 00000010001100000000000000000000  
 2: 00001011000010000000000000000110  
 3: 00010010000110001100001000000000  
 4: 00010011000010000111111111111111  
 5: 00001111000000000000000000000010  
 6: 00000010000100001100000000000000

Langage assembleur

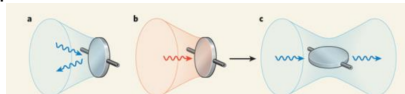
Langage machine (binaire)

## Technologie ?

- ▶ Notre machine est parfaitement abstraite et totalement indépendante du choix technologique pour son 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



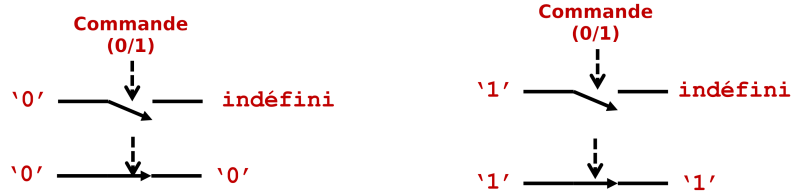
## Un interrupteur

Ne propage rien s'il est ouvert

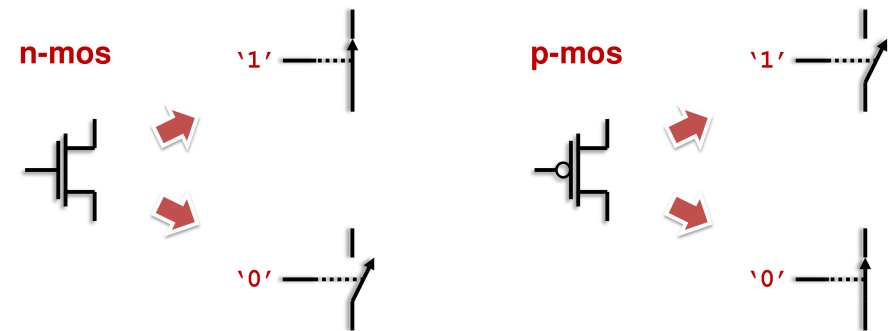


Propage son entrée s'il est fermé

## Transistors = interrupteur contrôlé (1/2)

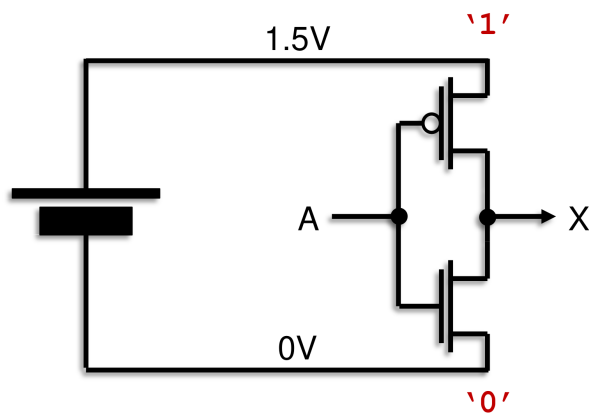


## Transistors = interrupteur contrôlé (2/2)



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

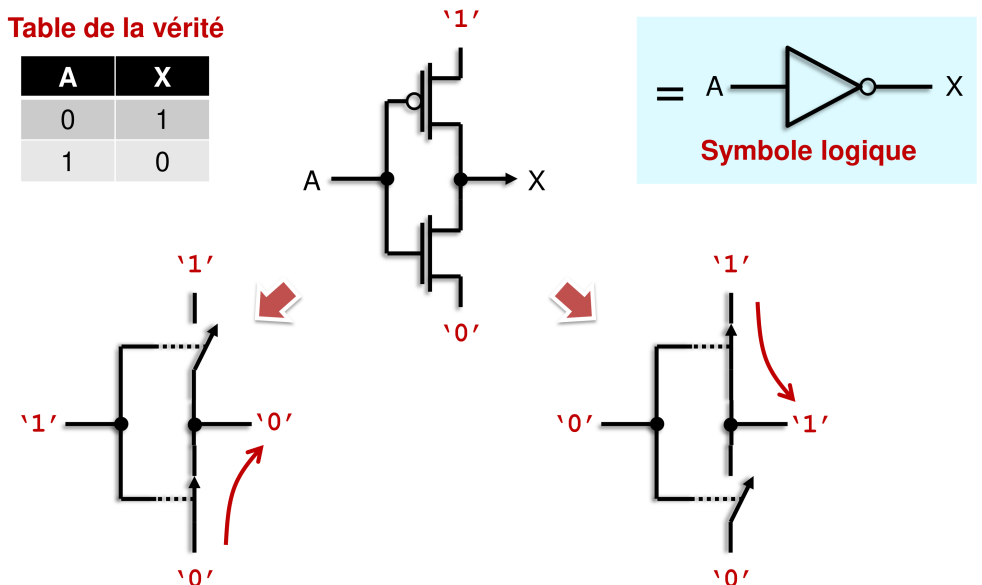
## Un inverseur (en CMOS)



## Un inverseur (en CMOS)

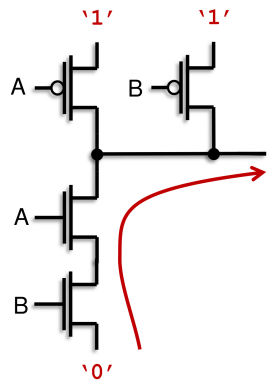
Table de la vérité

A	X
0	1
1	0





## Un circuit « et » avec sortie inversée

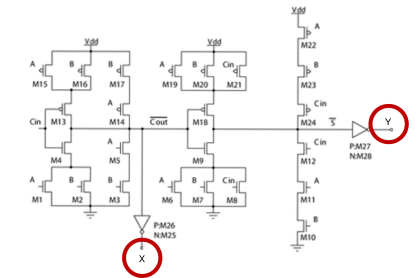


A	B	A et B inversé
0	0	1
0	1	1
1	0	1
1	1	0

- ▶ 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 beaucoup de fonctions !

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

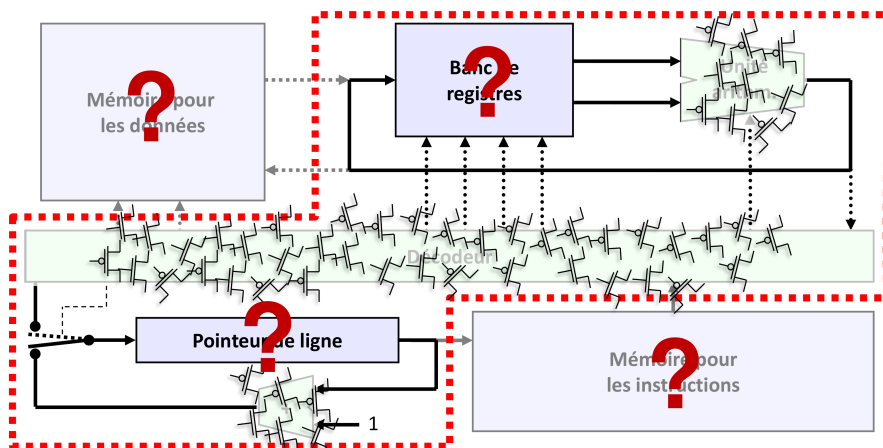


Les sorties XY sont la somme (en représentation binaire) des trois bits A, B et C à l'entrée



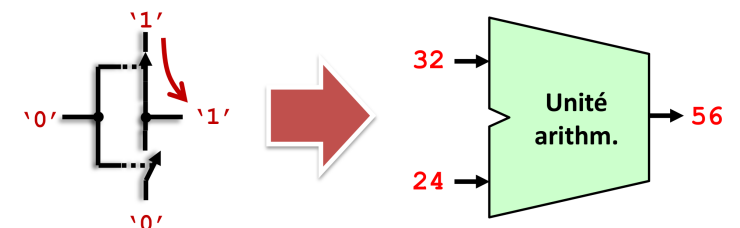
**Addition !**

## Et notre processeur, alors ?

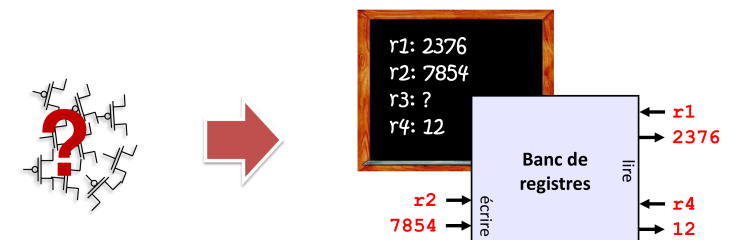


## Peut-on aussi mémoriser l'information ?

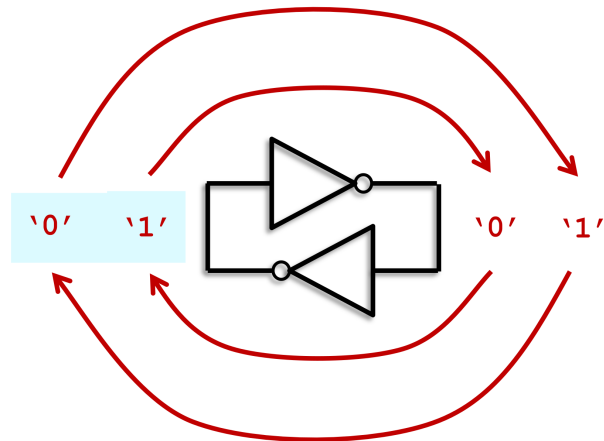
- ▶ Pour les calculs, tout va bien :



- ▶ Mais pour mémoriser ?...

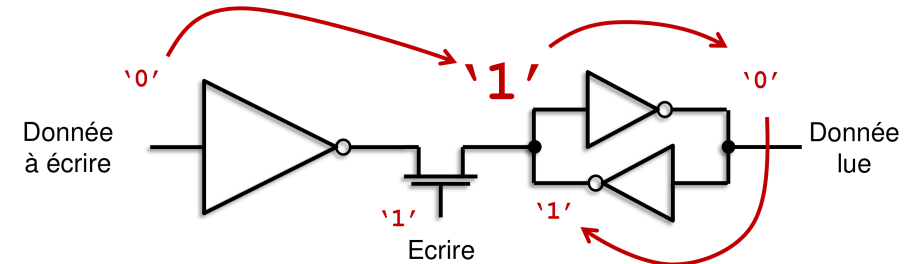


## Un circuit assez particulier

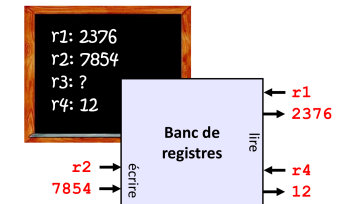


- ▶ Un circuit « bistable » c.-à-d. qui peut être dans un parmi deux états parfaitement stables  
■ **un élément mémoire !**

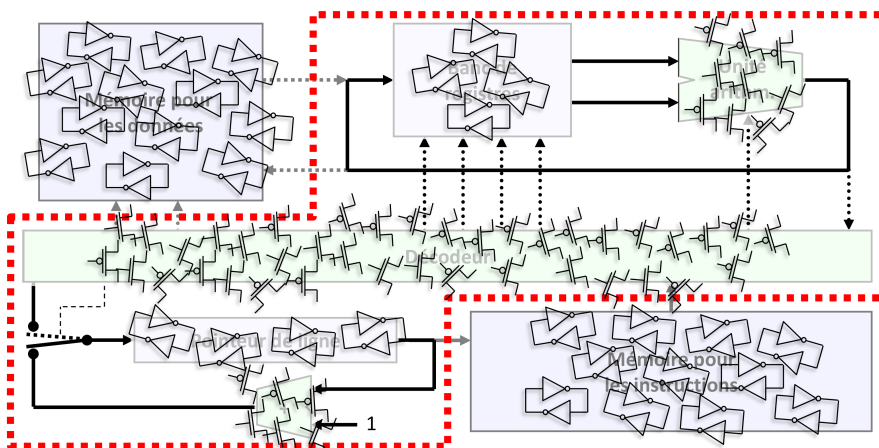
## Comment écrire dans cette mémoire ?



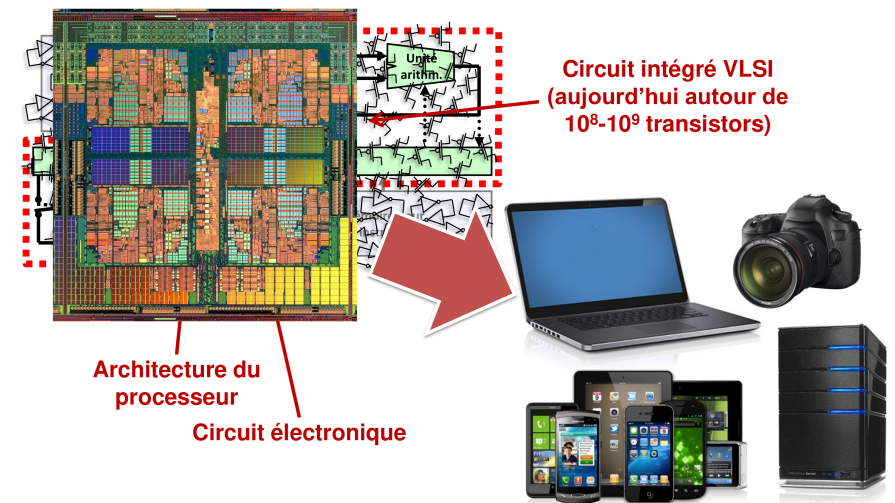
- ▶ Avec quelques transistors, on a un parfait circuit mémoire pour notre processeur



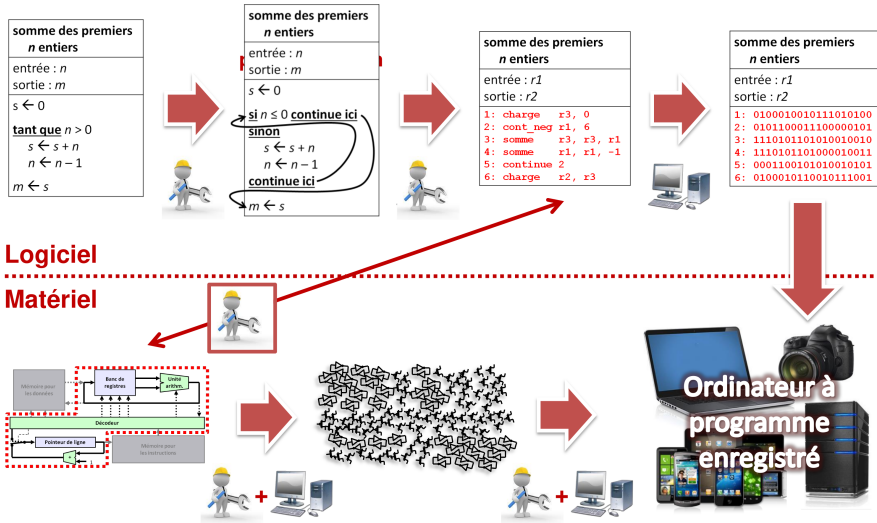
## Maintenant on sait tout faire !



## On a atteint notre but !



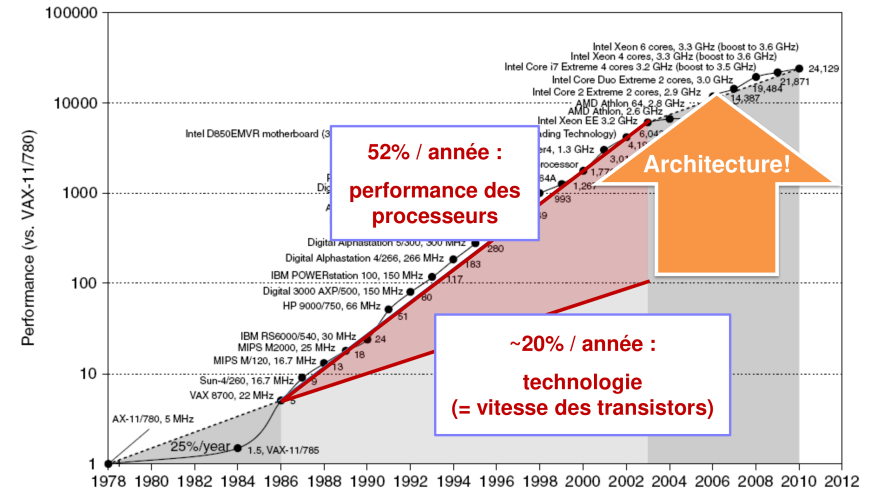
# Des algorithmes aux ordinateurs



Logiciel

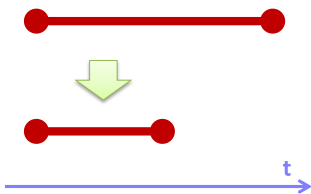
Matériel

# La croissance de la performance

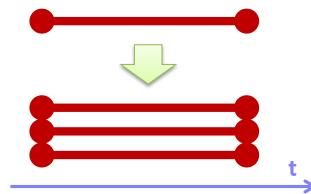


# Augmenter la performance ?

**= Réduire le délai**  
temps d'attente pour obtenir un résultat



**= Augmenter le débit**  
nombre de résultats dans l'unité de temps



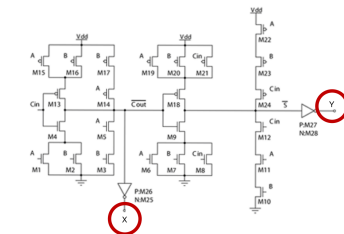
A suivre : deux exemples simples d'amélioration de la performance :

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... (1/2)

sur 1 bit (avec la retenue !):

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

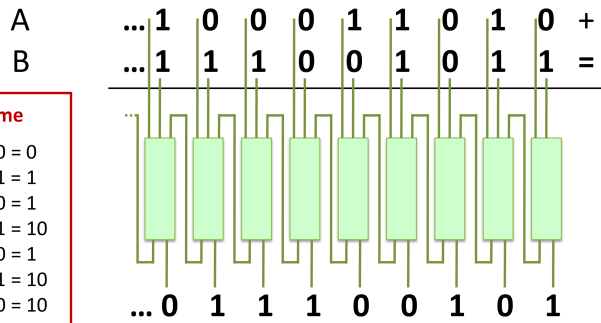


Les sorties XY sont la somme (en représentation binaire) des trois bits A, B et C à l'entrée

**Addition !**

## Faire des sommes est facile... (2/2)

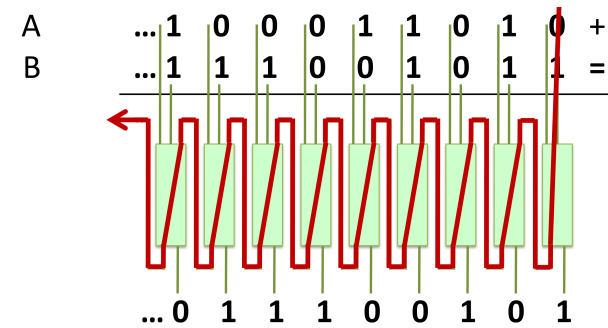
sur  $n$  bits :



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	

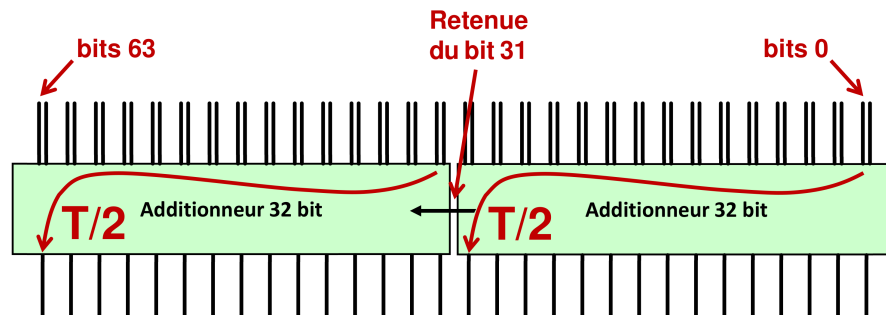
... 0 1 1 1 0 0 1 0 1

## Mais ce circuit est lent !



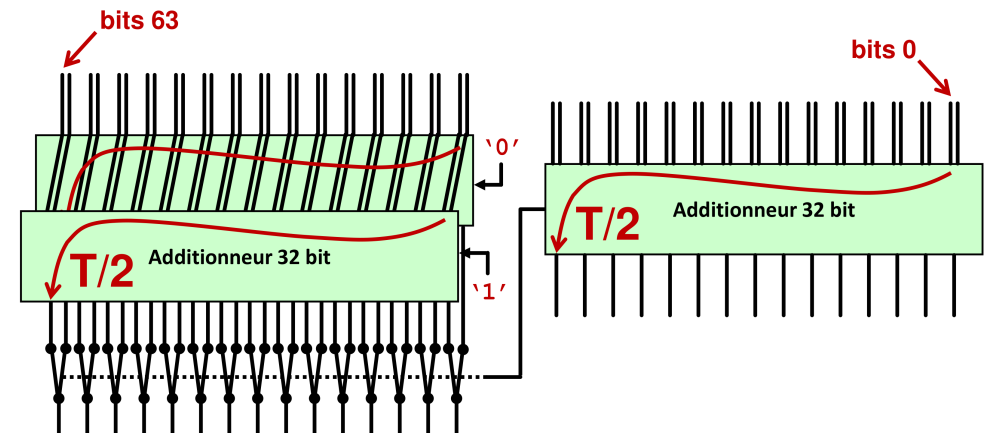
La propagation de la retenue est un aspect fondamental de la somme : ainsi implémenté, le **décal** d'un additionneur est donc *proportionnel au nombre de bits à additionner*.

## Peut-on faire mieux ?



On n'a rien gagné :  $T/2 + T/2 = T$

## Peut-on faire mieux ?



Là on a **divisé le temps par 2!**..  
 ...mais augmenté le travail/l'énergie par 1.5

## Ingénierie informatique (1/2)

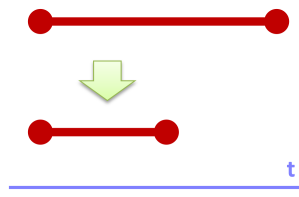
- ▶ On peut profondément changer la performance du circuit sans en changer la fonctionnalité.
- ▶ On peut investir plus de transistors et plus d'énergie pour obtenir des circuits très rapides.
- ▶ On peut ralentir les circuits pour épargner de l'énergie.

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



A suivre : le second exemple simple d'amélioration de la performance :

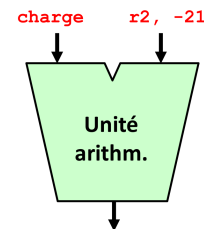
- ▶ Au niveau du circuit  
**Réduire le délai** d'un additionneur
- ▶ Au niveau de la structure du processeur  
**Augmenter le débit** d'instructions

## 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 ?

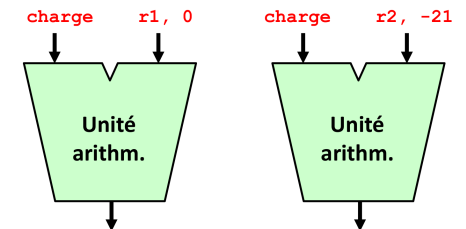


## Doubler le débit de notre processeur ?

```
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 peut maintenant **exécuter**  
**deux instructions** par cycle!

Problèmes ?



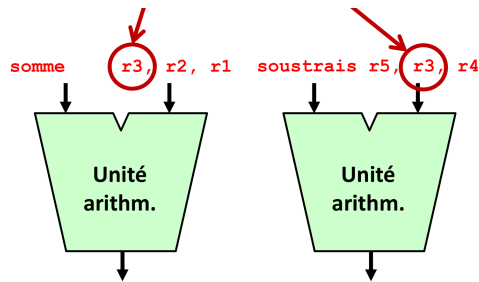
## Doubler le débit de notre processeur ?

```

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
    
```

On peut maintenant **exécuter deux instructions** par cycle !

Problèmes ?

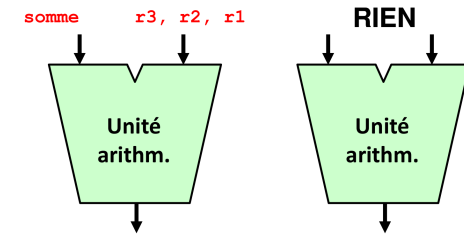


## Doubler le débit de notre processeur ?

```

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
    
```

Solution : retarder l'exécution de certaines instructions :

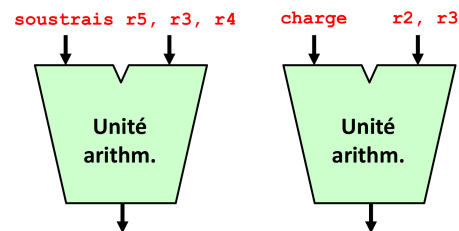


## Doubler le débit de notre processeur ?

```

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
    
```

Solution : et n'exécuter en parallèle que des instructions indépendantes :

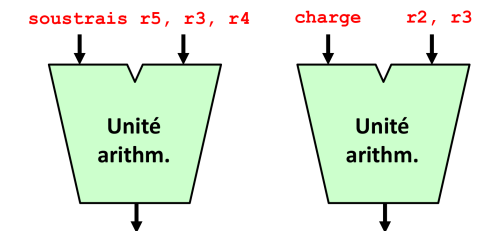


## ~~Doubler le débit de notre processeur~~

```

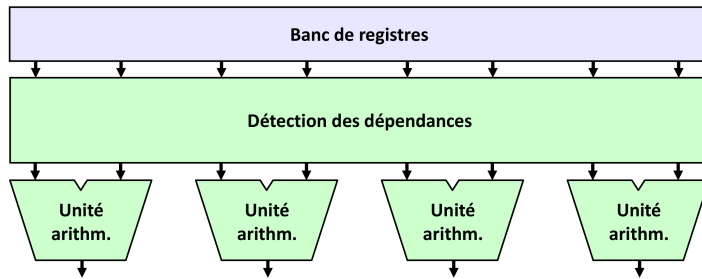
103: charge    r1, 0
104: charge    r2, -21
105: somme     r3, r7, r4
106: multiplie r2, r5, r9
107: soustrais r8, r7, r9
108: charge    r9, r4
109: somme     r3, r2, r1
110: soustrais r5, r3, r4
111: charge    r2, r3
112: somme     r1, r2, -1
113: somme     r8, r1, -1
114: divise   r4, r1, r7
115: charge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

On exécute maintenant **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 réordonnent les instructions et en exécutent avant que ce soit sûr qu'elles doivent être exécutées (p.ex. après une instruction comme `cont_neg`).

## Ingénierie informatique (2/2)

- ▶ 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.

Ceci est un exemple d'**architecture des ordinateurs** qui est une autre branche de l'**ingénierie informatique** (*Computer Engineering*)

## Ce que j'ai appris aujourd'hui

Dans ce cours, vous avez vu

- ▶ les principes de fonctionnement des ordinateurs actuels
- ▶ basés sur l'**architecture de von Neumann** :
  - ▶ processeur (CPU) ;
  - ▶ mémoire ;
  - ▶ périphériques ;
- ▶ comment un ordinateur traduit un langage de programmation dans ses instructions internes (compilation en langage machine) ;
- ▶ les compromis nécessaires entre performances et énergie consommée :
  - ▶ jouer sur le **délat** ;
  - ▶ et le **débit** (parallélisme).