

# Architecture des ordinateurs

## (INF155)

F. Pellegrini  
Université de Bordeaux

Ce document est copiable et distribuable librement et gratuitement à la condition expresse que son contenu ne soit modifié en aucune façon, et en particulier que le nom de son auteur et de son institution d'origine continuent à y figurer, de même que le présent texte.

# Ordinateur et logiciel

---

- Les technologies numériques sont maintenant omniprésentes
  - Elles sont le moteur et l'objet de ce qu'on appelle la « révolution numérique »
- Elles sont basées sur l'interaction entre :
  - Des programmes, aussi appelés logiciels, décrivant des processus de traitement de l'information : biens immatériels
  - Des ordinateurs, capables d'exécuter ces programmes : biens matériels

# Représentation de l'information

---

- L'information est représentée au sein des composants de l'ordinateur sous forme de différents états de la matière :
  - « Trou » ou « pas trou » sur la surface d'un cédérom ou DVD
  - Orientation nord ou sud d'un matériau magnétique
  - Lumière ou absence de lumière émise par un laser
  - Courant électrique ou non
- Ce sont souvent des représentations à deux états, c'est-à-dire « binaires »

# Constituants élémentaires (1)

- Presque tous les ordinateurs sont construits à base de circuits électroniques
- Les circuits électroniques sont réalisés au moyen de transistors
  - Composant élémentaire, dont le courant de sortie dépend de deux valeurs d'entrée
    - Un transistor a donc trois « pattes »
      - Appelées : base, émetteur et collecteur
    - Analogue à un « robinet à électricité » : plus il arrive de courant sur la base, plus le courant circule de l'émetteur vers le collecteur



# Constituants élémentaires (2)

---

- Dans les ordinateurs, on utilise les transistors en mode saturé, c'est-à-dire « tout ou rien »
  - Fonctionnement analogue à celui d'un interrupteur
    - Robinet fermé ou ouvert en grand
    - Soit le courant passe, soit il ne passe pas du tout
  - Représentation des valeurs binaires « 0 » et « 1 »
- En combinant plusieurs transistors, on peut effectuer des calculs complexes
  - Sur la base de montages en série ou en parallèle
- Regroupement au sein de « circuits intégrés »

# Performance (1)

---

- Les calculs des ordinateurs sont cadencés par une horloge
  - Plus la fréquence de l'horloge est élevée, et plus l'ordinateur pourra effectuer d'opérations par seconde (s'il n'est pas ralenti par autre chose...)
  - On mesure la fréquence d'une horloge en Hertz (Hz)
    - Nombre de battements par seconde
      - 1 kHz (kilo-Hertz) =  $10^3$  Hz
      - 1 MHz (méga-Hertz) =  $10^6$  Hz
      - 1 GHz (giga-Hertz) =  $10^9$  Hz
      - 1 THz (téra-Hertz) =  $10^{12}$  Hz

# Performance (2)

---

- En fait, ce qui importe aux usagers, c'est le nombre d'opérations (plus généralement, « d'instructions ») qu'un ordinateur est capable d'effectuer par seconde
  - On la mesure en MIPS, pour « millions d'instructions par seconde »
- On pense souvent que la puissance d'un ordinateur dépend de sa fréquence de fonctionnement
  - C'est loin d'être toujours vrai !

[http://en.wikipedia.org/wiki/Instructions\\_per\\_second](http://en.wikipedia.org/wiki/Instructions_per_second)

# Évolutions architecturales (1)

---

- 1946 : Ordinateur ENIAC
  - Architecture à base de lampes et tubes à vide : 30 tonnes, 170 m<sup>2</sup> au sol, 5000 additions par seconde
  - 0,005 MIPS, dirons-nous...
- 1947 : Invention du transistor
- 1958 : Invention du circuit intégré sur silicium
  - Multiples transistors agencés sur le même substrat



# Évolutions architecturales (2)

---

- 1971 : Processeur Intel 4004
  - 2300 transistors dans un unique circuit intégré
  - Fréquence de 740 kHz, 0,092 MIPS
- ...40 ans d'une histoire très riche...
- 2011 : Processeur Intel Core i7 2600K
  - Plus de 1,4 milliards de transistors
  - Fréquence de 3,4 GHz
  - 4 cœurs, 8 threads
  - 128300 MIPS

# Évolutions architecturales (3)

---

- Entre le 4004 et le Core i7 2600K :
  - La fréquence a été multipliée par 4600
  - La puissance en MIPS a été multipliée par 1,4 million
- La puissance d'un ordinateur ne dépend clairement pas que de sa fréquence !
- Intérêt d'étudier l'architecture des ordinateurs pour comprendre :
  - Où les gains se sont opérés
  - Ce qu'on peut attendre dans le futur proche

# Barrière de la chaleur (1)

---

- Plus on a de transistors par unité de surface, plus on a d'énergie à évacuer
- La dissipation thermique évolue de façon proportionnelle à  $V^2 * F$ 
  - La tension de fonctionnement des circuits a été abaissée
    - De 5V pour les premières générations à 0,9V maintenant
  - Il n'est plus vraiment possible de la diminuer avec les technologies actuelles
    - Le bruit thermique causerait trop d'erreurs

# Barrière de la chaleur (2)

---

- La fréquence ne peut raisonnablement augmenter au delà des 5 GHz
  - « Barrière de la chaleur »
- La tendance est plutôt à la réduction
  - « *Green computing* »
  - On s'intéresse maintenant à maximiser le nombre d'opérations par Watt
  - Mais on veut toujours plus de puissance de calcul !

# Barrière de la complexité (1)

---

- À surface constante, le nombre de transistors double tous les 2 ans
  - « Loi de Moore », du nom de Gordon Moore, co-fondateur d'Intel, énoncée en 1965
  - Diminution continue de la taille de gravage des transistors et circuits sur les puces de silicium
    - On grave actuellement avec un pas de 14 nm
- Limites atomiques bientôt atteintes...
  - Donc plus possible d'intégrer plus
  - Mais on veut toujours plus de puissance de calcul !

# Barrière de la complexité (2)

---

- Que faire de tous ces transistors ?
  - On ne voit plus trop comment utiliser ces transistors pour améliorer individuellement les processeurs
  - Des processeurs trop complexes consomment trop d'énergie sans aller beaucoup plus vite
- Seule solution actuellement : faire plus de processeurs sur la même puce !
  - Processeurs bi-cœurs, quadri-cœurs, octo-cœurs, ... déjà jusqu'à 128 cœurs !
  - Mais comment les programmer efficacement ?!

# Barrière de la complexité (3)

---

- L'architecture des ordinateurs a été l'un des secteurs de l'informatique qui a fait le plus de progrès
- Les ordinateurs d'aujourd'hui sont très complexes
  - Plus d'un milliard de transistors dans un processeur
- Nécessité d'étudier leur fonctionnement à différents niveaux d'abstraction
  - Du composant au module, du module au système
  - Multiples niveaux de hiérarchie

# Structure d'un ordinateur (1)

---

- Un ordinateur est une machine programmable de traitement de l'information
- Pour accomplir sa fonction, il doit pouvoir :
  - Acquérir de l'information de l'extérieur
  - Stocker en son sein ces informations
  - Combiner entre elles les informations à sa disposition
  - Restituer ces informations à l'extérieur



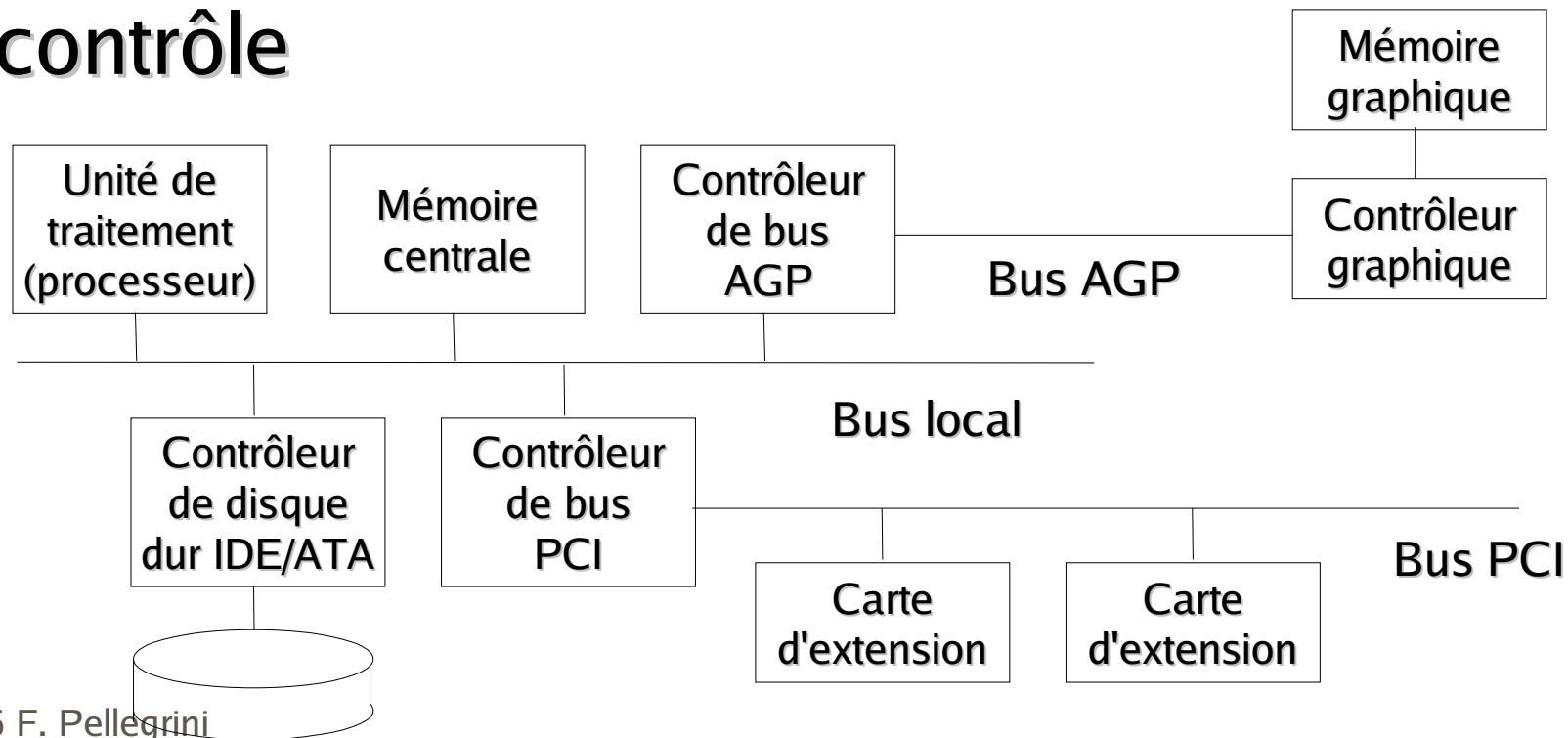
# Structure d'un ordinateur (2)

---

- L'ordinateur doit donc posséder :
  - Une ou plusieurs unités de stockage, pour mémoriser le programme en cours d'exécution ainsi que les données qu'il manipule
  - Une unité de traitement permettant l'exécution des instructions du programme et des calculs sur les données qu'elles spécifient
  - Différents dispositifs « périphériques » servant à interagir avec l'extérieur : clavier, écran, souris, carte graphique, carte réseau, etc.

# Structure d'un ordinateur (3)

- Les constituants de l'ordinateur sont reliés par un ou plusieurs bus, ensembles de fils parallèles servant à la transmission des adresses, des données, et des signaux de contrôle



# Unité de traitement (1)

---

- L'unité de traitement (ou CPU, pour « *Central Processing Unit* »), aussi appelée « processeur », est le cœur de l'ordinateur
- Elle exécute les programmes chargés en mémoire centrale en extrayant l'une après l'autre leurs instructions, en les analysant, et en les exécutant

# Unité de traitement (2)

---

- L'unité de traitement est composé de plusieurs sous-ensembles distincts
  - L'unité de contrôle, qui est responsable de la recherche des instructions à partir de la mémoire centrale et du décodage de leur type
  - L'unité arithmétique et logique (UAL), qui effectue les opérations spécifiées par les instructions
  - Un ensemble de registres, zones mémoires rapides servant au stockage temporaire des données en cours de traitement par l'unité centrale

# Registres

---

- Chaque registre peut stocker une valeur entière distincte, bornée par la taille des registres (nombre de bits)
- Certains registres sont spécialisés, comme :
  - le compteur ordinal (« *program counter* ») qui stocke l'adresse de la prochaine instruction à exécuter
  - le registre d'instruction (« *instruction register* »), qui stocke l'instruction en cours d'exécution
  - l'accumulateur, registre résultat de l'UAL, etc.

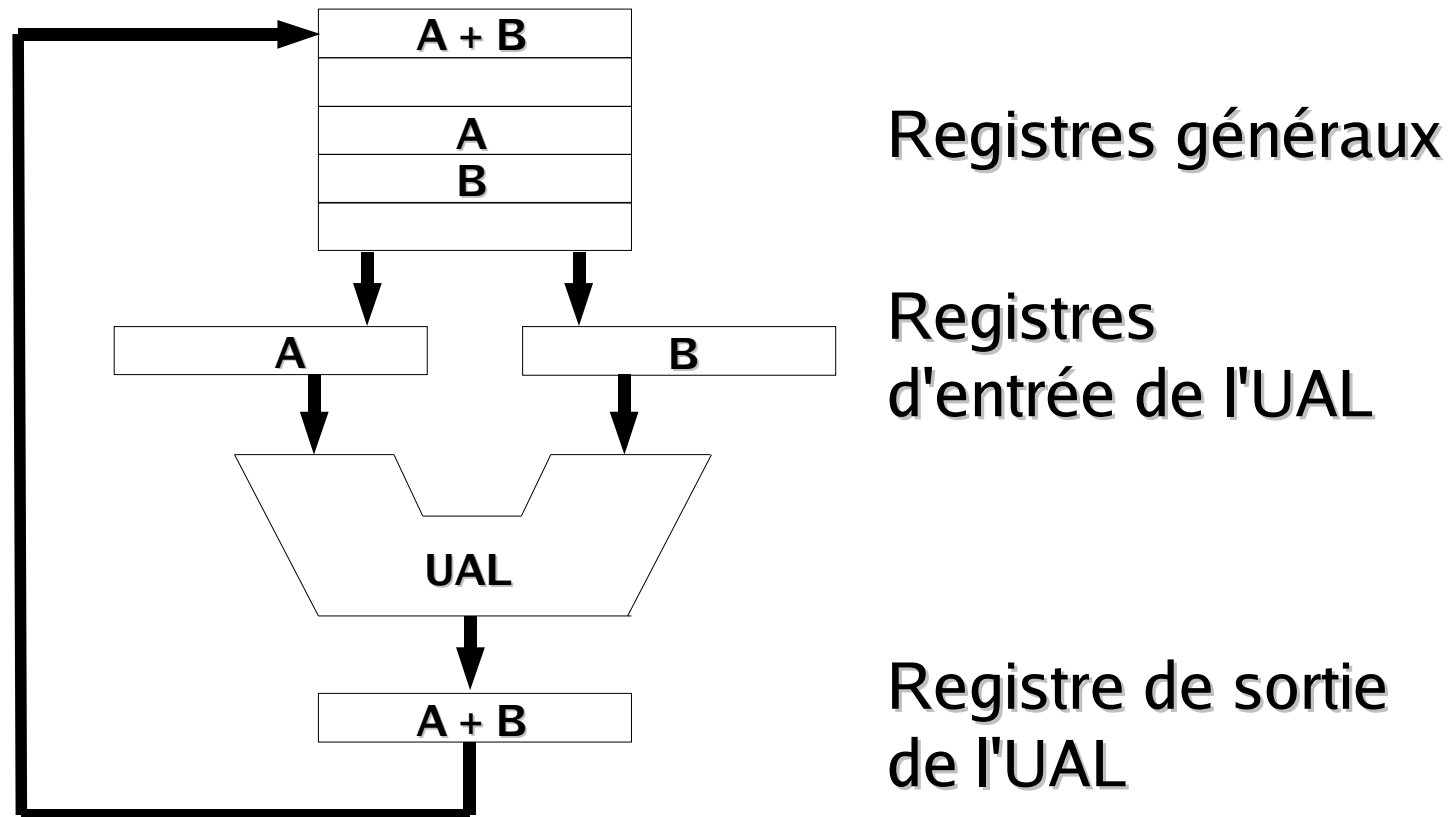
# Chemin de données (1)

---

- Le chemin de données représente la structure interne de l'unité de traitement
  - Comprend les registres, l'UAL, et un ensemble de bus internes dédiés
  - L'UAL peut posséder ses propres registres destinés à mémoriser les données d'entrées afin de stabiliser leurs signaux pendant que l'UAL calcule
- Le chemin des données conditionne fortement la puissance des machines
  - Pipe-line, superscalarité, ...

# Chemin de données (2)

- Chemin de données d'une machine de type « Von Neumann »



# Exécution d'une instruction (1)

---

- L'exécution d'une instruction par l'unité centrale s'effectue selon les étapes suivantes :
  - 1 Charger la prochaine instruction à exécuter depuis la mémoire vers le registre d'instruction
  - 2 Décoder (analyser) l'instruction venant d'être lue
  - 3 Faire pointer le compteur ordinal vers l'instruction suivante (y compris dans le cas de branchements)
  - 4 Localiser en mémoire les données nécessaires
  - 5 Charger si nécessaire les données dans l'UAL
  - 6 Exécuter l'instruction, puis recommencer

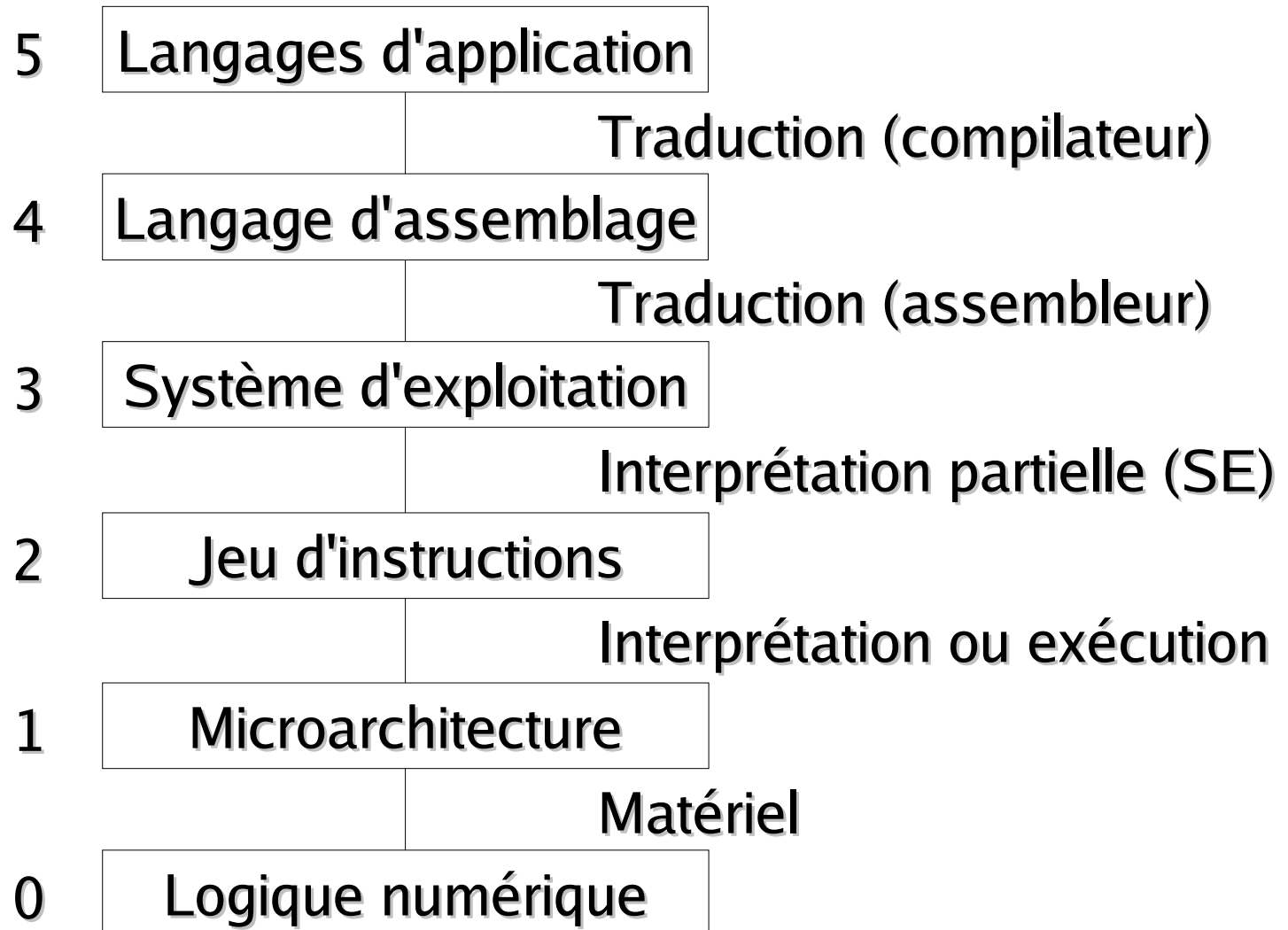


# Architecture des ordinateurs

---

- Les ordinateurs modernes sont conçus comme un ensemble de couches
- Chaque couche représente une abstraction différente, capable d'effectuer des opérations et de manipuler des objets spécifiques
- L'ensemble des types de données, des opérations, et des fonctionnalités de chaque couche est appelée son architecture
- L'étude de la conception de ces parties est appelée « architecture des ordinateurs »

# Machines multi-couches actuelles



# Couche logique numérique

---

- Les objets considérés à ce niveau sont les portes logiques, chacune construite à partir de quelques transistors
- Chaque porte prend en entrée des signaux numériques (0 ou 1) et calcule en sortie une fonction logique simple (ET, OU, NON)
- De petits assemblages de portes peuvent servir à réaliser des fonctions logiques telles que mémoire, additionneur, ainsi que la logique de contrôle de l'ordinateur

# Couche microarchitecture

---

- On dispose à ce niveau de plusieurs registres mémoire et d'un circuit appelé UAL (Unité Arithmétique et Logique, ALU) capable de réaliser des opérations arithmétiques élémentaires
- Les registres sont reliés à l'UAL par un chemin de données permettant d'effectuer des opérations arithmétiques entre registres
- Le contrôle du chemin de données est soit microprogrammé, soit matériel

# Couche jeu d'instruction

---

- La couche de l'architecture du jeu d'instructions (Instruction Set Architecture, ISA) est définie par le jeu des instructions disponibles sur la machine
- Ces instructions peuvent être exécutées par microprogramme ou bien directement

# Couche système d'exploitation

---

- Cette couche permet de bénéficier des services offerts par le système d'exploitation
  - Organisation mémoire, exécution concurrente
- La plupart des instructions disponibles à ce niveau sont directement traitées par les couches inférieures
- Les instructions spécifiques au système font l'objet d'une interprétation partielle (appels système)

# Couche langage d'assemblage

---

- Offre une forme symbolique aux langages des couches inférieures
- Permet à des humains d'interagir avec les couches inférieures

# Couche langages d'application

---

- Met à la disposition des programmeurs d'applications un ensemble de langages adaptés à leurs besoins
- Langages dits « de haut niveau »



# Comment aborder tout cela ?

---

- Approches courante : de bas en haut pour les besoins, puis de haut en bas pour les solutions
  - Travaux pratiques difficiles au début...
- Par deux fronts à la fois :
  - À partir de la couche ISA
    - Programmation en langage machine : y86
  - À partir des transistors et portes logiques
    - Construction de circuits « sur papier »

# Circuits logiques

---

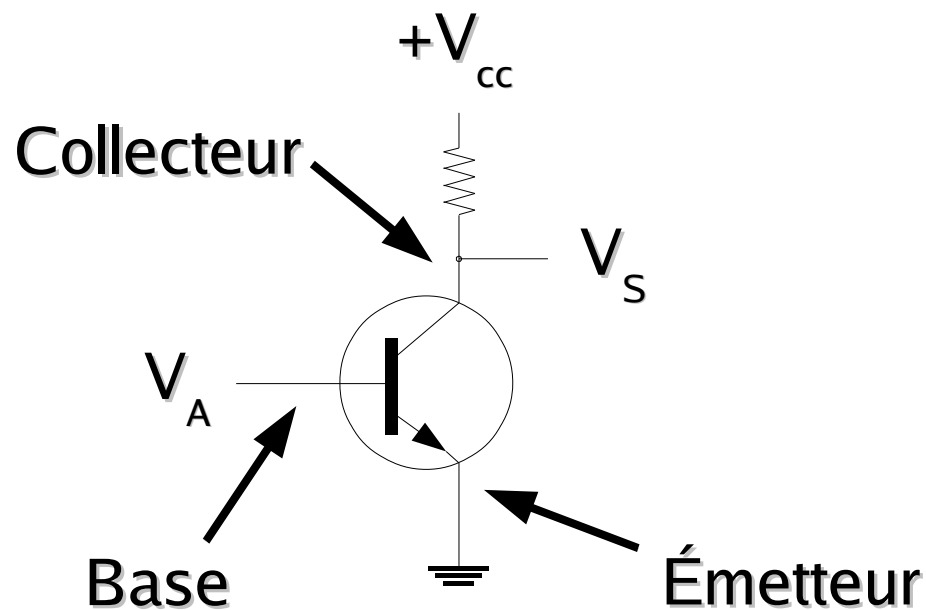
- Un circuit logique est un circuit qui ne manipule que deux valeurs logiques : 0 et 1
- À l'intérieur des circuits, on représente typiquement un état 0 par un signal de basse tension (proche de 0V) et un état 1 par un signal de haute tension (5V, 3,3V, 2,5V, 1,8V ou 0,9V selon les technologies)
- De minuscules dispositifs électroniques, appelées « portes », peuvent calculer différentes fonctions à partir de ces signaux

# Transistors (1)

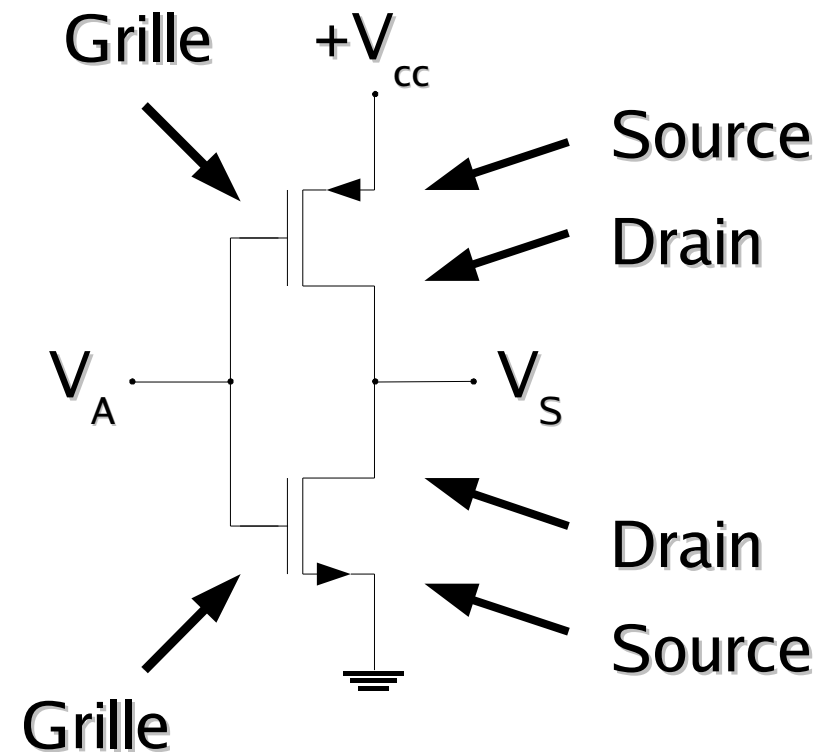
- L'électronique numérique repose sur le fait qu'un transistor peut servir de commutateur logique extrêmement rapide
- Deux technologies majeures :
  - Bipolaire : temps de commutation très rapide mais consommation élevée
    - Registres, SRAM, circuits spécialisés
  - CMOS : temps de commutation moins rapide mais consommation beaucoup moins élevée
    - 90 % des circuits sont réalisés en CMOS
- Possibilité de mixage bipolaire-CMOS : BiCMOS

# Transistors (2)

- Avec un transistor bipolaire ou deux transistors CMOS, on peut créer un premier circuit combinatoire :



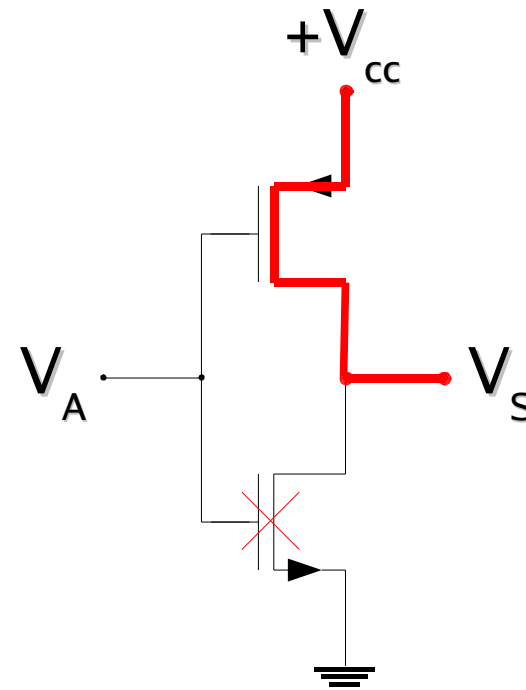
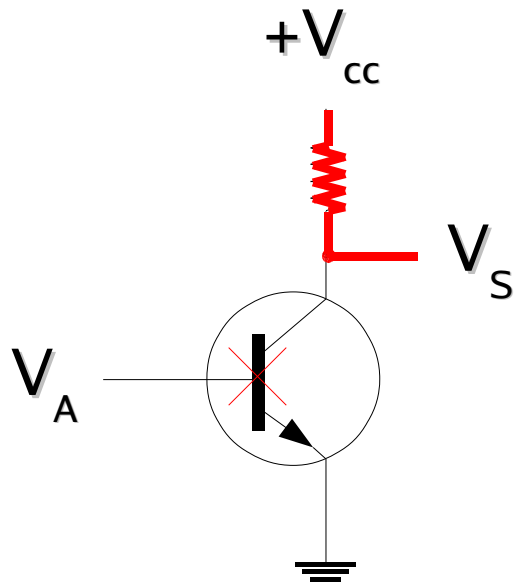
**Bipolaire**



**CMOS**

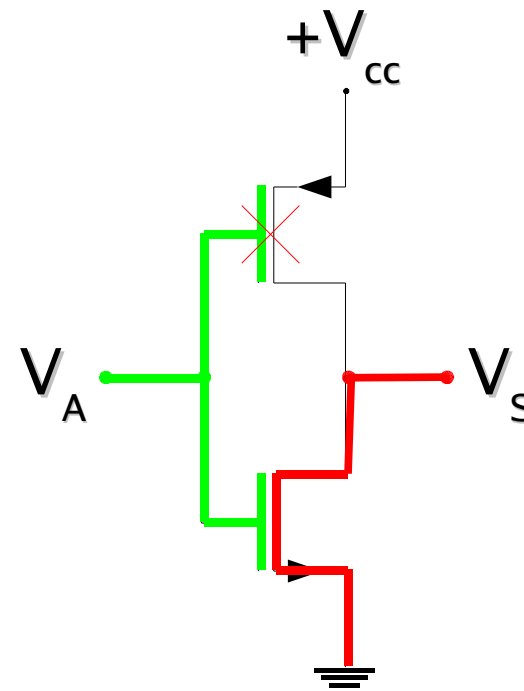
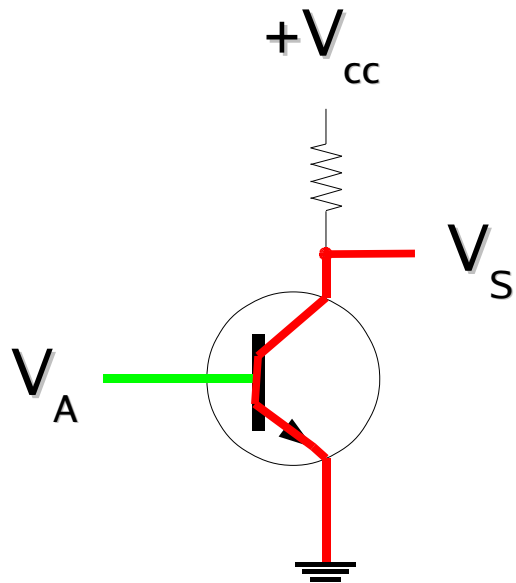
# Transistors (3)

- Quand  $V_A$  est bas,  $V_S$  est haut



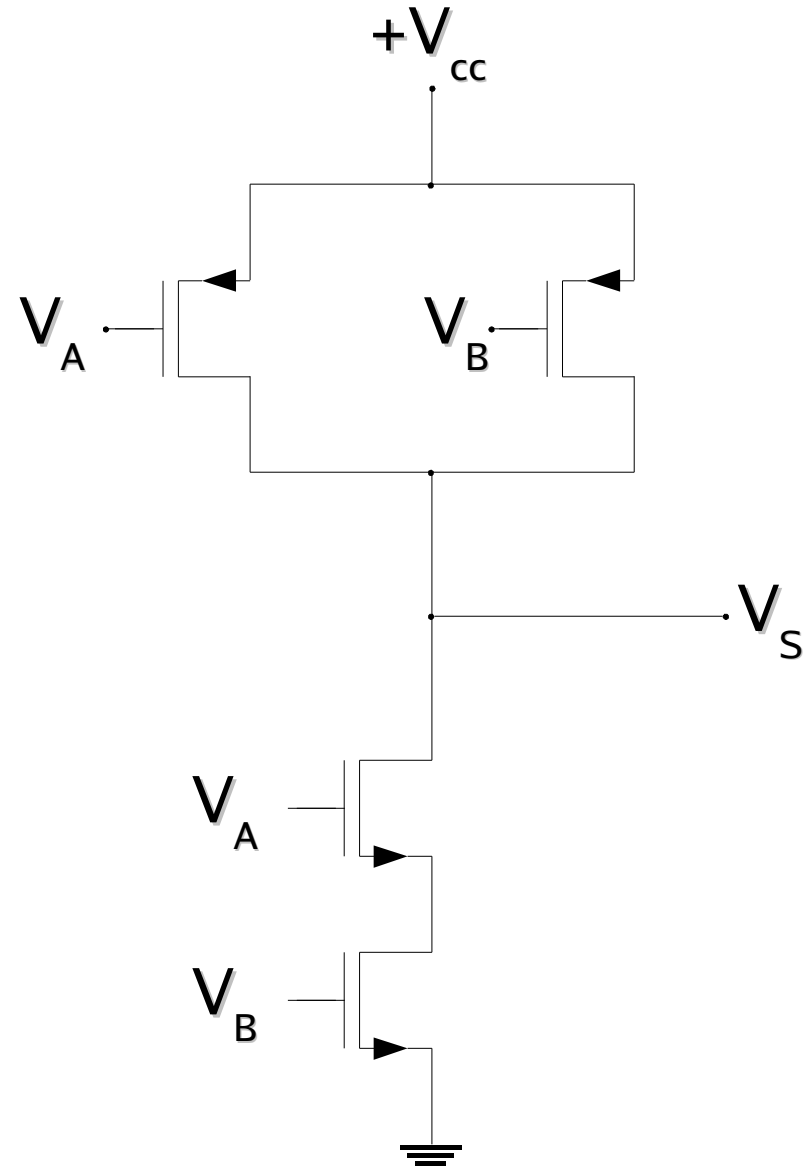
# Transistors (4)

- Quand  $V_A$  est bas,  $V_S$  est haut
- Quand  $V_A$  est haut,  $V_S$  est bas
- Ce circuit est un inverseur



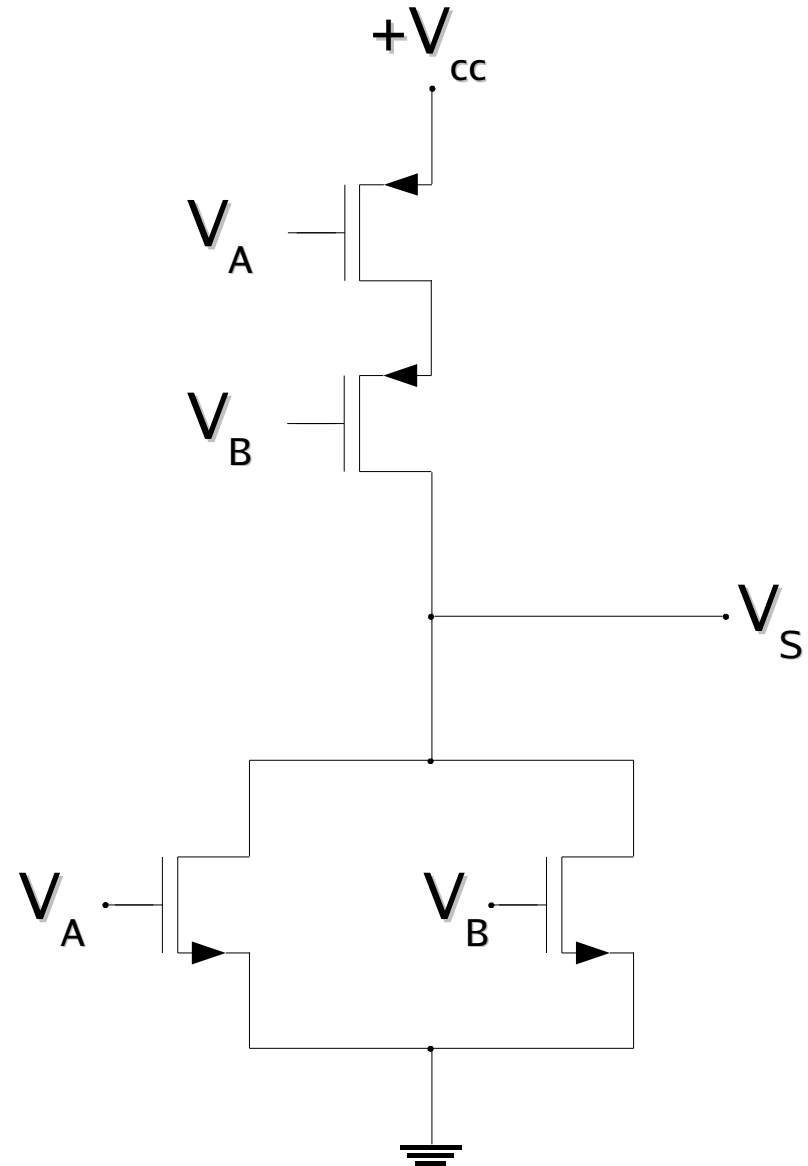
# Transistors (5)

- En combinant quatre transistors CMOS, on peut obtenir un circuit tel que  $V_S$  n'est dans l'état bas que quand  $V_A$  et  $V_B$  sont tous les deux dans l'état haut



# Transistors (6)

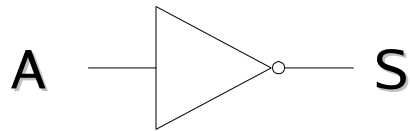
- En combinant quatre transistors CMOS, on peut obtenir un circuit tel que  $V_S$  est dans l'état bas si  $V_A$  ou  $V_B$ , ou bien les deux, sont dans l'état haut





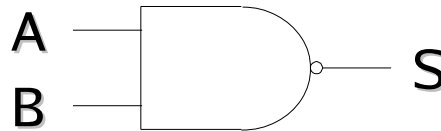
# Portes logiques (1)

- En identifiant l'état haut à la valeur 1 et l'état bas à la valeur 0, on peut exprimer la valeur de sortie de ces trois circuits à partir des valeurs de leurs entrées



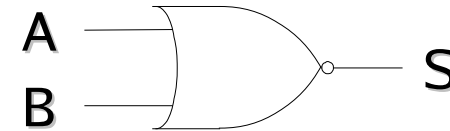
A	S
0	1
1	0

NON



A	B	S
0	0	1
0	1	1
1	0	1
1	1	0

NAND



A	B	S
0	0	1
0	1	0
1	0	0
1	1	0

NOR

# Portes logiques (2)

---

- Quelle que soit la technologie, les portes NAND et NOR nécessitent moins de transistors que les portes AND et OR, qui nécessitent un inverseur en plus
- Les circuits des ordinateurs sont donc plutôt construits avec des portes NAND et NOR
  - Ces portes ont parfois plus de deux entrées, mais en ont rarement plus de 8 (problème de « fan-in/out »)
  - Les portes NAND et NOR sont dites « complètes », car tout circuit peut être implanté uniquement au moyen de l'un de ces types de portes

# Algèbre booléenne

---

- Pour décrire les circuits réalisables en combinant des portes logiques, on a besoin d'une algèbre opérant sur les variables 0 et 1
- Algèbre booléenne
  - G. Boole : 1815 – 1864
  - Algèbre binaire étudiée par Leibniz dès 1703

# Fonctions booléennes (1)

---

- Une fonction booléenne à une ou plusieurs variables est une fonction qui renvoie une valeur ne dépendant que de ces variables
- La fonction NON est ainsi définie comme :
  - $f(A) = 1$  si  $A = 0$
  - $f(A) = 0$  si  $A = 1$

# Fonctions booléennes (2)

- Une fonction booléenne à  $n$  variables a seulement  $2^n$  combinaisons d'entrées possibles
- Elle peut être complètement décrite par une table à  $2^n$  lignes donnant la valeur de la fonction pour chaque combinaison d'entrées
  - Table de vérité de la fonction
- Elle peut aussi être décrite par le nombre à  $2^n$  bits correspondant à la lecture verticale de la colonne de sortie de la table
  - NAND : 1110, NOR : 1000, AND : 0001, etc.

# Fonctions booléennes (3)

---

- Toute fonction peut être décrite en spécifiant lesquelles des combinaisons d'entrée donnent 1
- On peut donc représenter une fonction logique comme le « ou » logique (OR) d'un ensemble de conditions « et » (AND) sur les combinaisons d'entrée

# Fonctions booléennes (4)

- En notant :
  - $\bar{A}$  le NOT de A
  - $A + B$  le OR de A et B
  - $A.B$  ou  $AB$  le AND de A et B

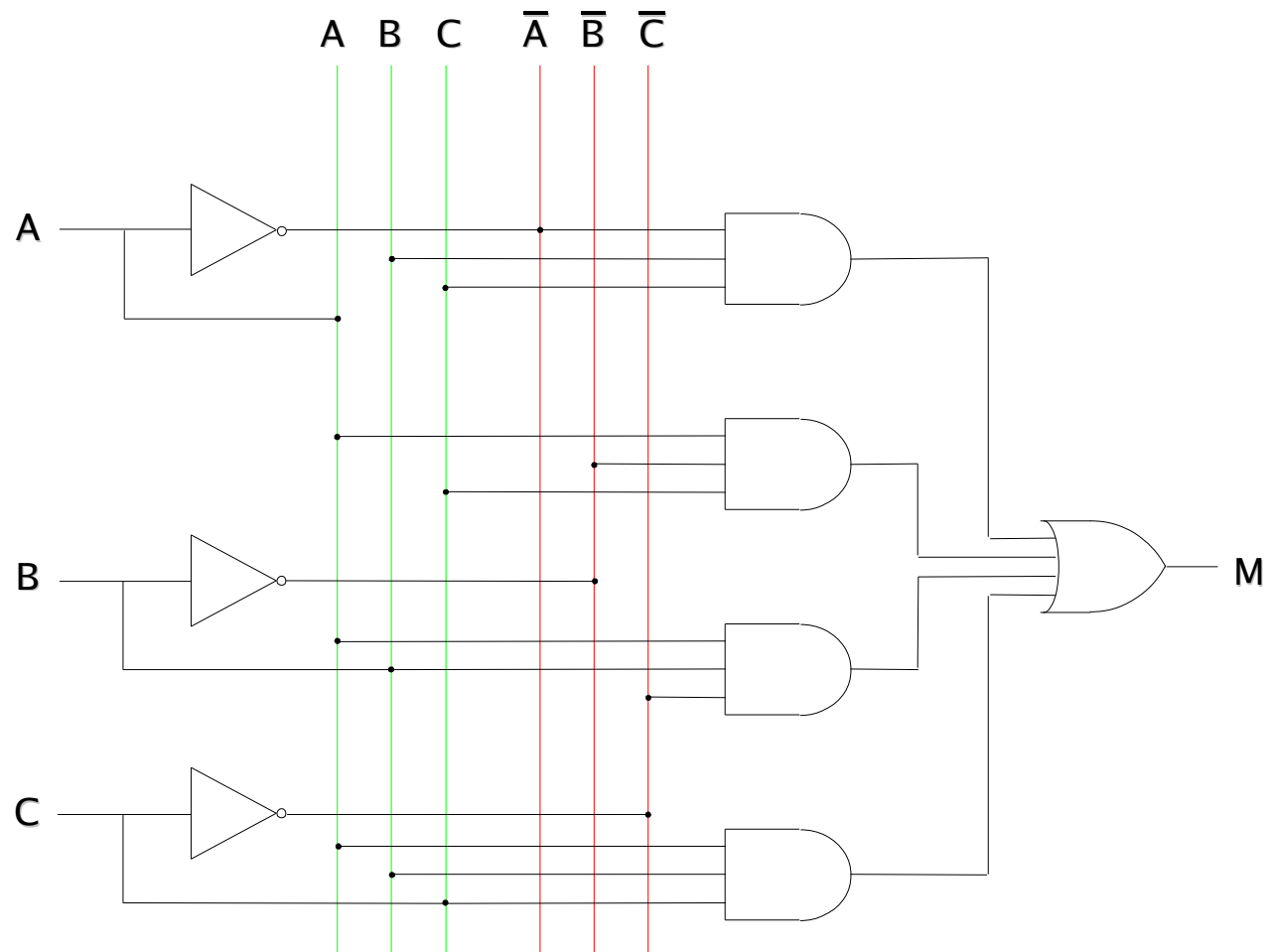
on peut représenter une fonction comme  
somme logique de produits logiques

- Par exemple :
  - $A\bar{B}C$  vaut 1 seulement si  $A = 1$  et  $B = 0$  et  $C = 1$
  - $A\bar{B} + B\bar{C}$  vaut 1 si et seulement si ( $A = 1$  et  $B = 0$ )  
ou bien ( $B = 1$  et  $C = 0$ )

# Fonctions booléennes (5)

- Exemple : la fonction majorité M

A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1





# Fonctions booléennes (6)

- Toute fonction logique de  $n$  variables peut donc être décrite sous la forme d'une somme logique d'au plus  $2^n$  produits de termes
  - Par exemple :  $M = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$
- Cette formulation fournit une méthode directe pour implanter n'importe quelle fonction booléenne

# Simplification de l'implantation (1)

---

- Deux fonctions sont équivalentes si et seulement si leurs tables de vérité sont identiques
- Il est intéressant d'implanter une fonction avec le moins de portes possible
  - Économie de place sur le processeur
  - Réduction de la consommation électrique
  - Réduction du temps de parcours du signal
- L'algèbre booléenne peut être un outil efficace pour simplifier les fonctions

# Simplification de l'implantation (2)

---

- La plupart des règles de l'algèbre ordinaire restent valides pour l'algèbre booléenne
- Exemple :  $AB + AC = A(B + C)$ 
  - On passe de trois portes à deux
  - Le nombre de niveaux de portes reste le même

# Simplification de l'implantation (3)

---

- Pour réduire la complexité des fonctions booléennes, on essaye d'appliquer des identités simplificatrices à la fonction initiale
- Besoin d'identités remarquables pour l'algèbre booléenne

# Identités booléennes (1)

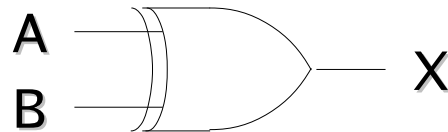
Nom	Forme AND	Forme OR
Identité	$1A = A$	$0 + A = A$
Nul	$0A = 0$	$1 + A = 1$
Idempotence	$AA = A$	$A + A = A$
Inverse	$AA = 0$	$A + A = 1$
Commutativité	$AB = BA$	$A + B = B + A$
Associativité	$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
Distributivité	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$
Absorbtion	$A(A + B) = A$	$A + AB = A$
De Morgan	$AB = A + B$	$A + B = A B$

# Identités booléennes (2)

- Chaque loi a deux formes, qui sont duales si on échange les rôles respectifs de AND et OR et de 0 et 1
- La loi de De Morgan peut être étendue à plus de deux termes
  - $\overline{ABC} = \bar{A} + \bar{B} + \bar{C}$
  - Notation alternative des portes logiques :
    - Une porte OR avec ses deux entrées inversées est équivalente à une porte NAND
    - Une porte NOR peut être dessinée comme une porte AND avec ses deux entrées inversées

# Porte XOR (1)

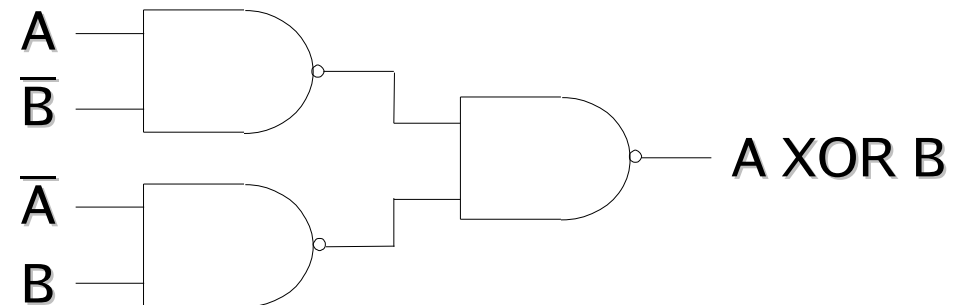
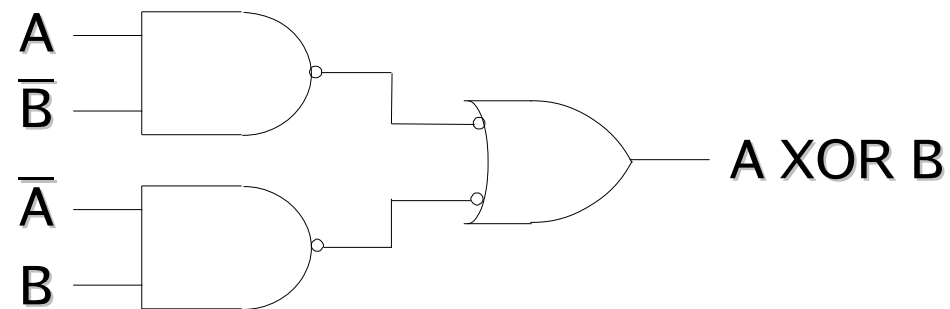
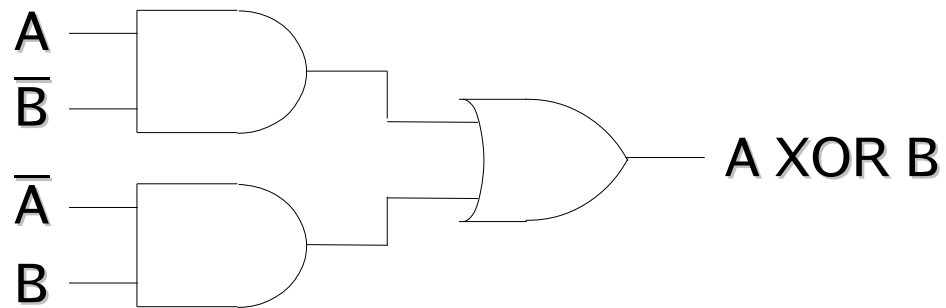
- Grâce aux identités, il est facile de convertir la représentation en somme de produits en une forme purement NAND ou NOR
- Exemple : la fonction « ou exclusif » ou XOR
  - $XOR = A\bar{B} + \bar{A}B$



A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

XOR

# Porte XOR (2)





# Fonctions logiques élémentaires

---

- Pour réaliser des circuits logiques complexes, on ne part pas des portes logiques elles-mêmes mais de sous-ensembles fonctionnels tels que :
  - Fonctions combinatoires et arithmétiques
  - Horloges
- L'implantation elle-même peut se faire à différents niveaux d'intégration
  - PLD, SRAM, ASICs, FPGAs, ...

# Fonctions combinatoires

---

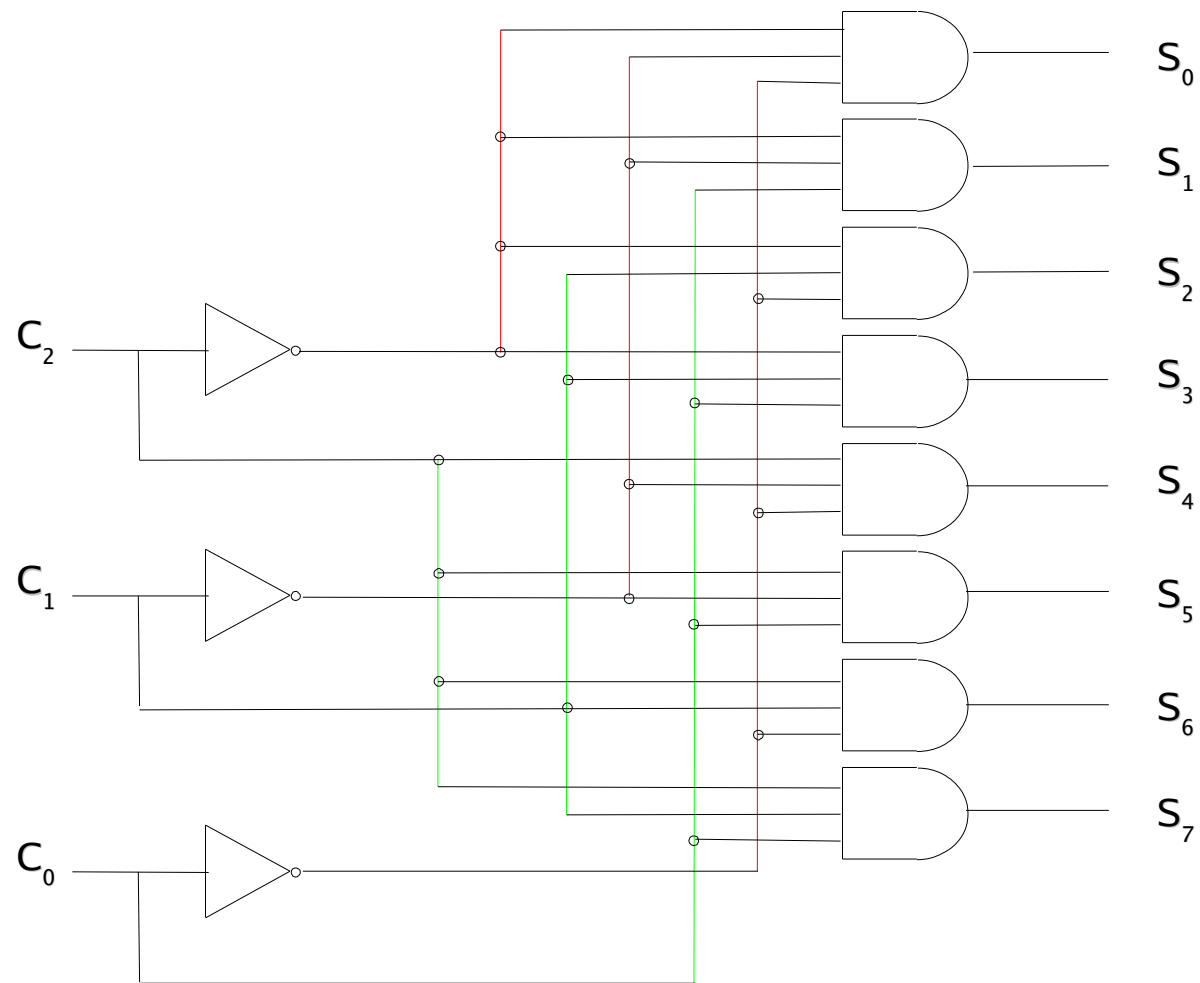
- Une fonction combinatoire est une fonction possédant des entrées et des sorties multiples, telles que les valeurs des sorties ne dépendent que des valeurs d'entrée
- Cette classe comprend les fonctions telles que :
  - Décodeurs
  - Multiplexeurs
  - Compareurs
  - ...

# Décodeur (1)

---

- Un décodeur est une fonction qui prend un nombre binaire  $C$  à  $n$  bits en entrée et se sert de celui-ci pour sélectionner l'une de ses  $2^n$  sorties  $S_i$

# Décodeur (2)

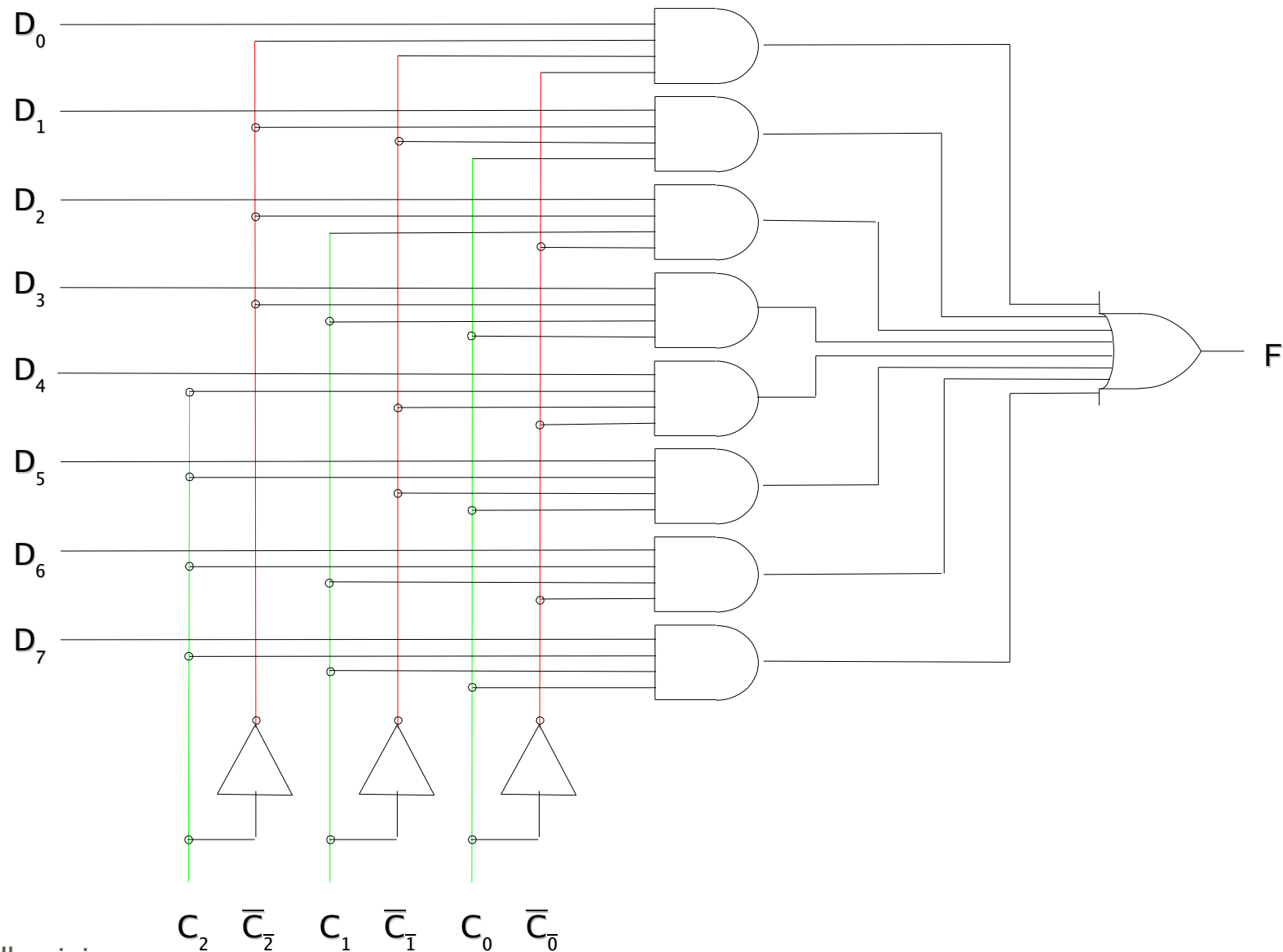


# Multiplexeur (1)

---

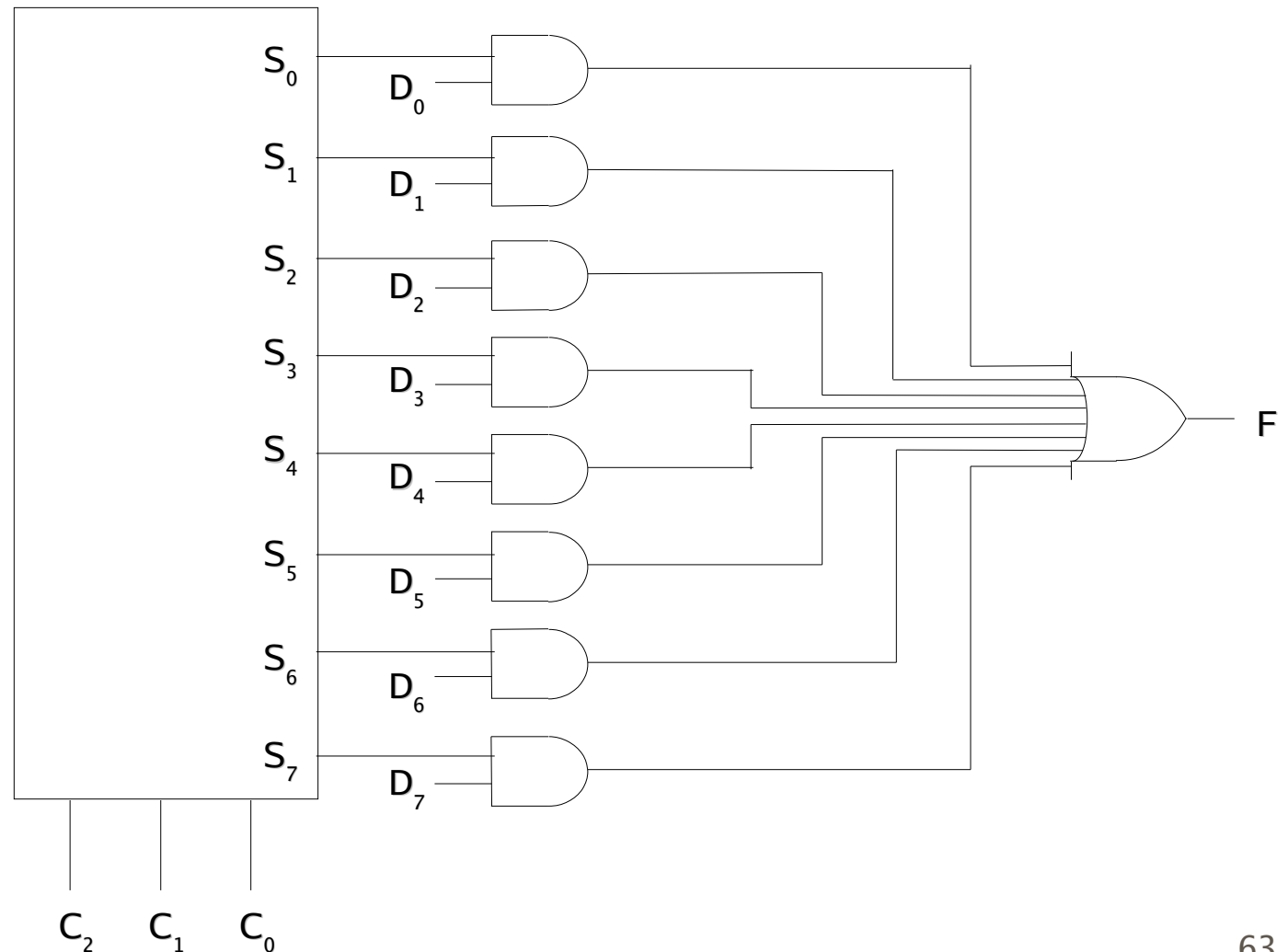
- Un multiplexeur est une fonction possédant  $2^n$  entrées de données  $D_i$ , une unique sortie et  $n$  entrées de contrôle  $C_j$  servant à sélectionner l'une des entrées
- La valeur de l'entrée sélectionnée est répercutée (routée) sur la sortie
- Les  $n$  entrées de contrôle codent un nombre binaire à  $n$  bits  $C$  spécifiant le numéro de l'entrée sélectionnée

# Multiplexeur (2)



# Multiplexeur (3)

- Implantation d'un multiplexeur à partir d'un décodeur





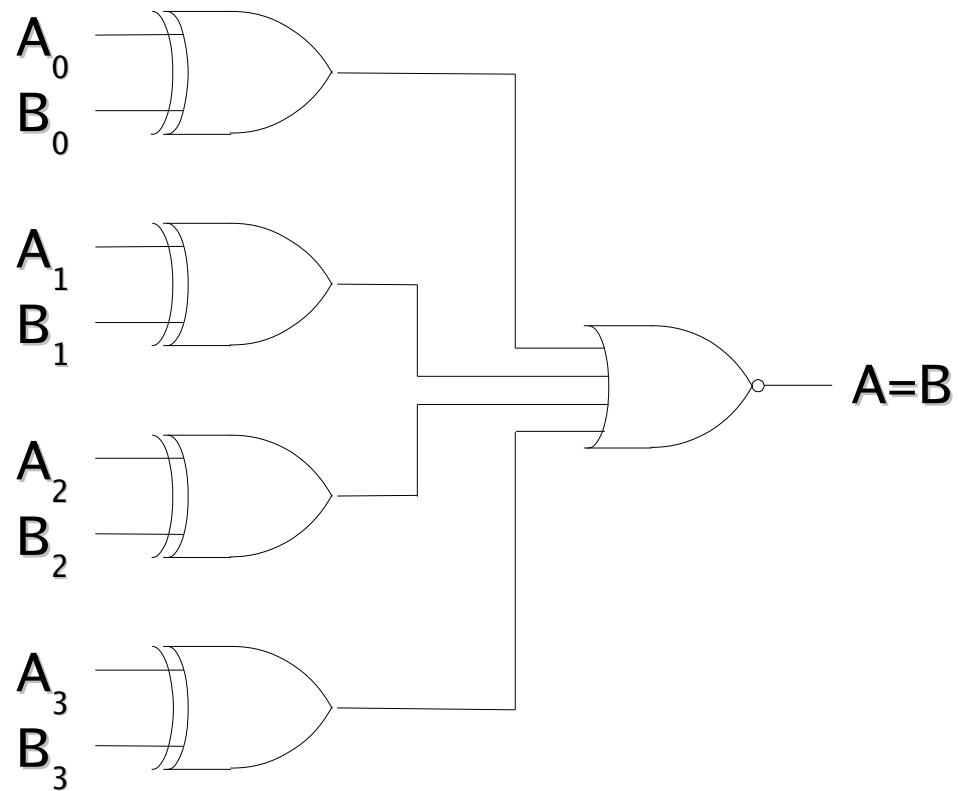


# Comparateur (1)

---

- Un comparateur est une fonction qui compare deux mots et qui produit 1 s'ils sont égaux bit à bit ou 0 sinon
- On le construit à partir de portes XOR, qui produisent 1 si deux bits en regard sont différents

# Comparateur (2)



# Fonctions arithmétiques

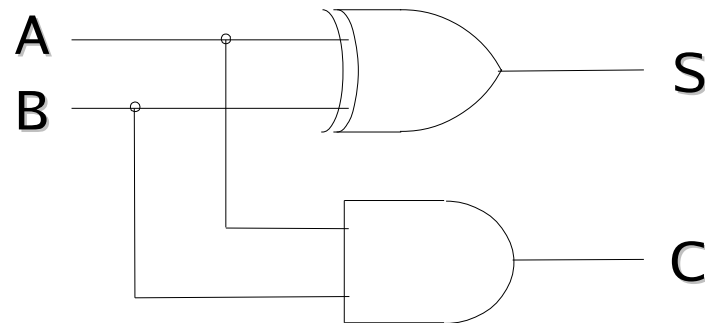
---

- Les fonctions arithmétiques sont des fonctions logiques représentant des opérations arithmétiques simples telles que :
  - Additionneur
  - Décaleur
  - Unité arithmétique et logique

# Additionneur (1)

- Tous les processeurs disposent d'un ou plusieurs circuits additionneurs
- Ces additionneurs sont implantés à partir de fonctions appelées « demi-additionneurs »

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



S : Somme

C : Retenue (« carry »)

# Additionneur (2)

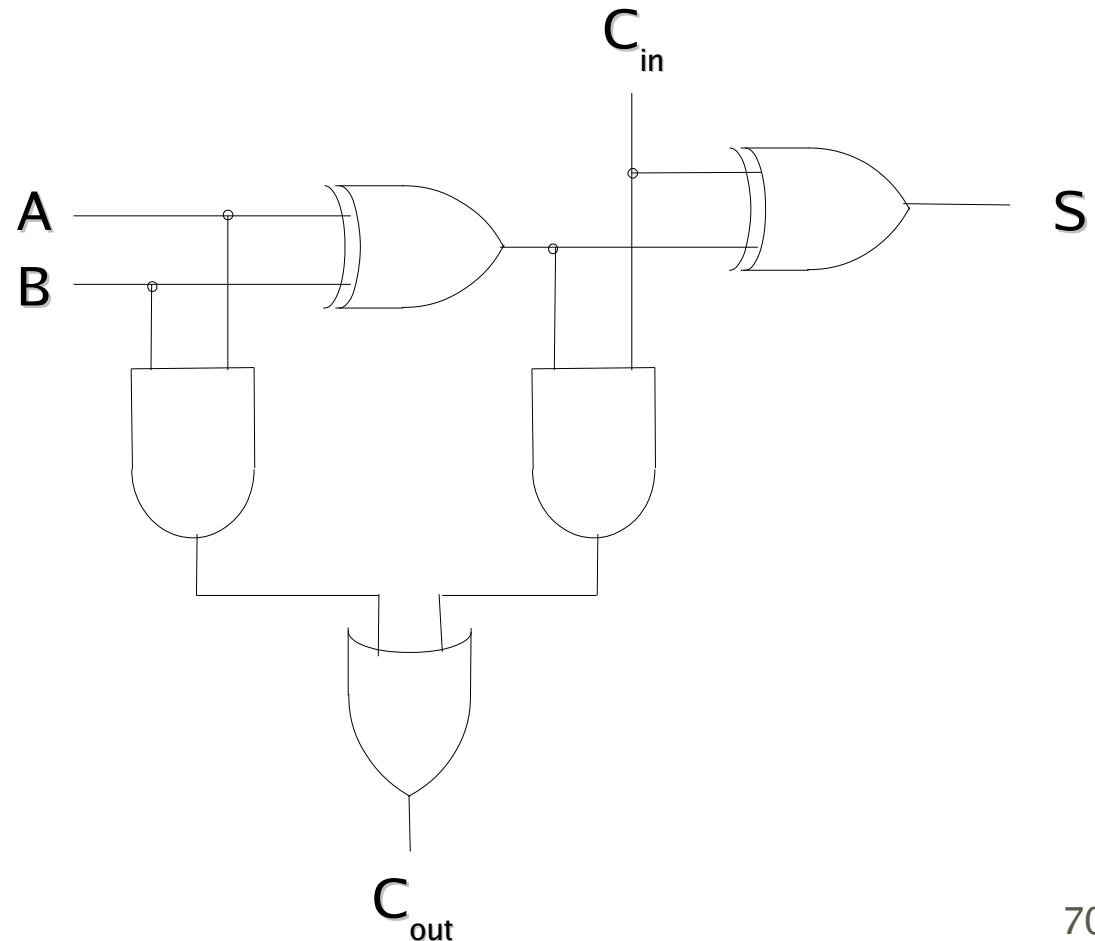
- En fait, pour additionner deux bits situés au milieu d'un mot, il faut aussi prendre en compte la retenue provenant de l'addition du bit précédent et propager sa retenue au bit suivant

$$\begin{array}{cccccccc}
 & & 1 & 1 & & & & \\
 & & \swarrow & \searrow & & & & \\
 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
 + & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\
 \hline
 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1
 \end{array}$$

# Additionneur (3)

- On utilise donc deux demi-additionneurs pour réaliser une tranche d'additionneur complet

A	B	C <sub>in</sub>	C <sub>out</sub>	S
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

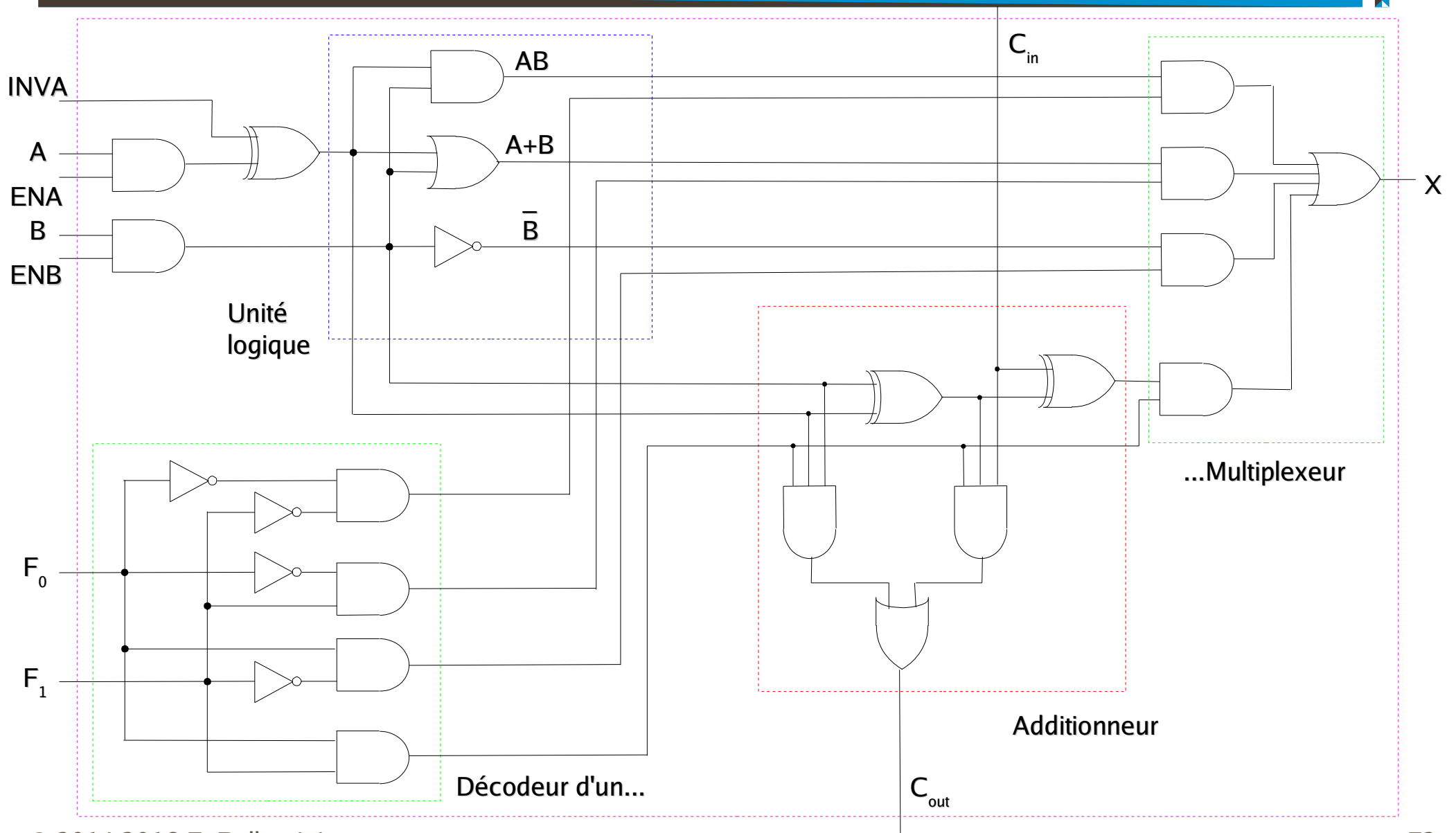


# Unité arithmétique et logique (1)

---

- La plupart des ordinateurs combinent au sein d'un même circuit les fonctions arithmétiques et logiques permettant de calculer l'addition, la soustraction, le AND ou le OU de deux mots machines : c'est l'Unité Arithmétique et Logique
- Le type de la fonction à calculer est déterminé par des entrées de contrôle

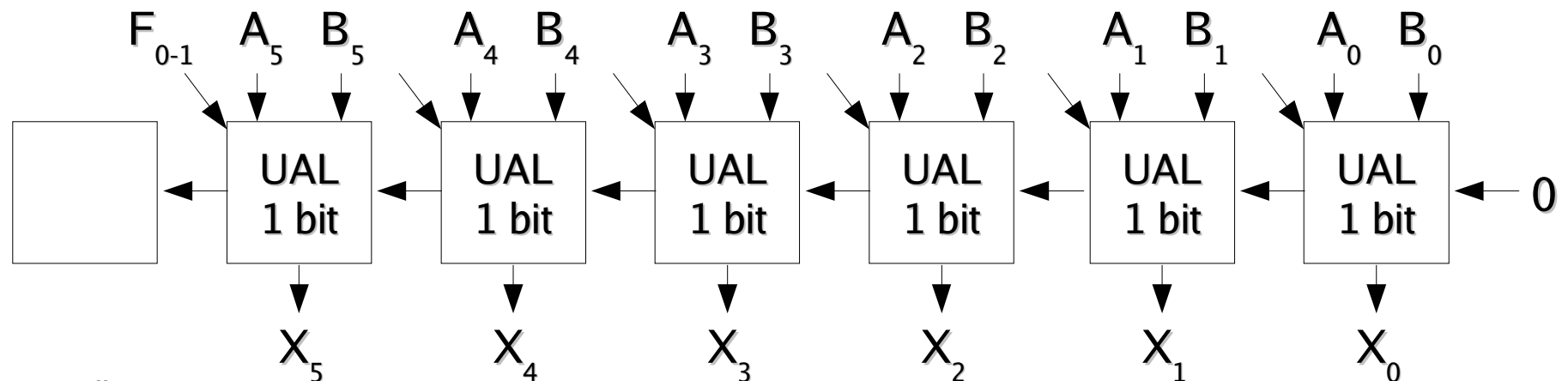
# Unité arithmétique et logique (2)





# Unité arithmétique et logique (2)

- Pour opérer sur des mots de  $n$  bits, l'UAL est constituée de la mise en série de  $n$  tranches d'UAL de 1 bit
- Pour l'addition, on chaîne les retenues et on injecte un 0 dans l'additionneur de poids faible
  - « *Ripple Carry Adder* », en fait peu efficace



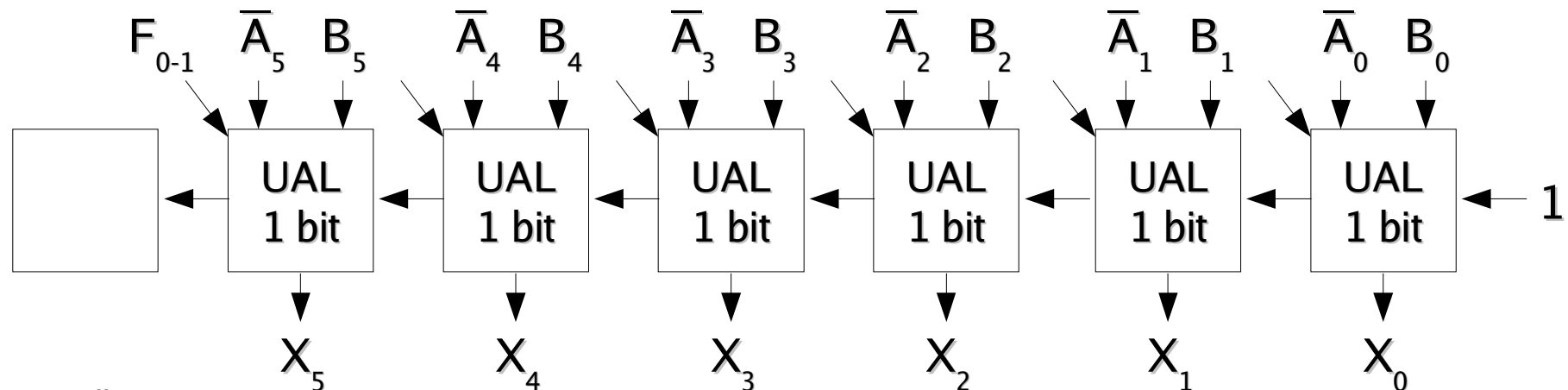
# Additionneur - Soustracteur (1)

---

- En notation « complément à deux », l'opposé d'un nombre est obtenu :
  - En complémentant tous les bits de ce nombre
  - En ajoutant 1 au résultat
- Soustraire A à B revient à calculer  $B + (-A)$ , ce qui peut se faire :
  - En ajoutant B au complément de A
  - En ajoutant 1 au résultat

# Additionneur - Soustracteur (2)

- L'unité arithmétique et logique dispose déjà de toute la circuiterie nécessaire !
  - La broche INVA permet d'effectuer la complémentation des bits de A avant l'addition
  - Il suffit d'injecter un 1 au lieu d'un 0 dans la retenue de l'additionneur de poids faible



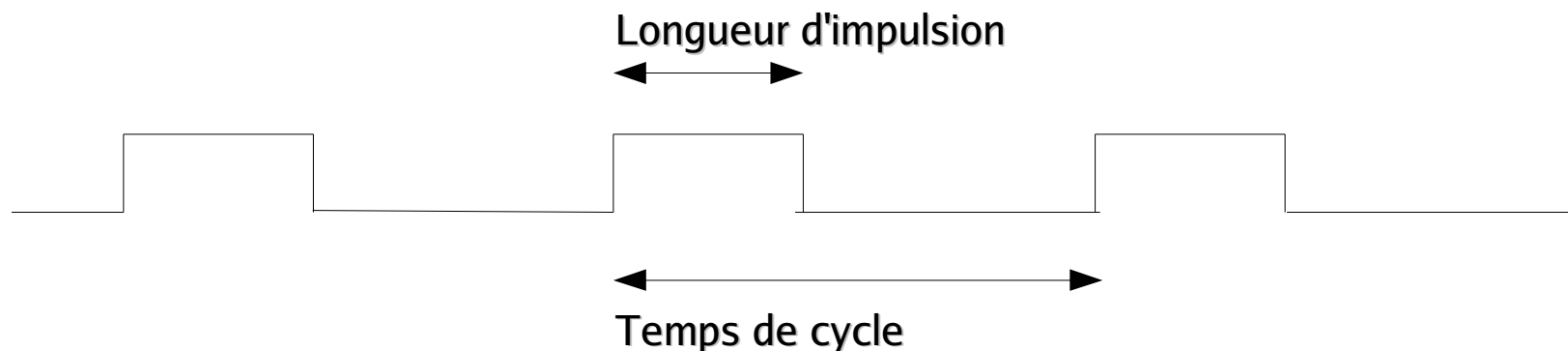
# Horloge (1)

---

- Dans de nombreux circuits numériques, il est essentiel de pouvoir garantir l'ordre dans lequel certains événements se produisent
  - Deux événements doivent absolument avoir lieu en même temps
  - Deux événements doivent absolument se produire l'un après l'autre
  - 98 % des circuits numériques sont synchrones
- Nécessité de disposer d'une horloge pour synchroniser les événements entre eux

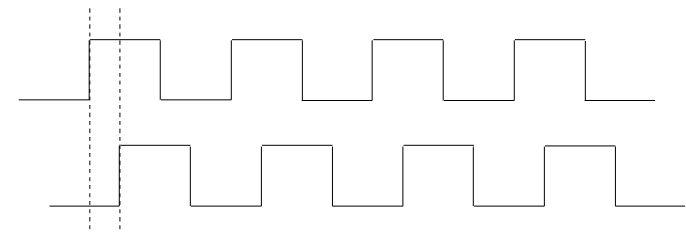
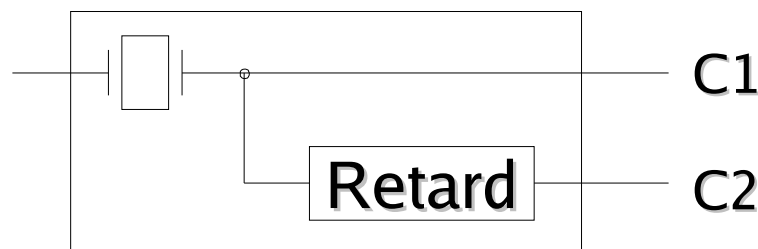
# Horloge (2)

- Une horloge est un circuit qui émet de façon continue une série d'impulsions caractérisées par :
  - La longueur de l'impulsion
  - L'intervalle entre deux pulsations successives, appelé temps de cycle de l'horloge



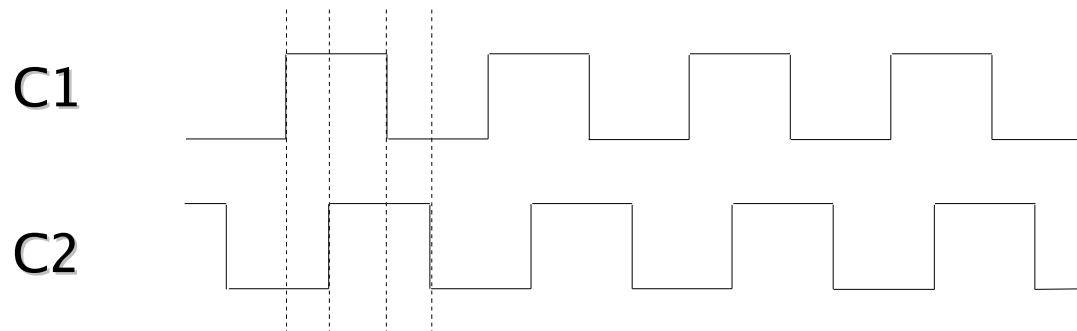
# Cycles et sous-cycles (1)

- Dans un ordinateur, de nombreux événements ont à se produire au cours d'un cycle d'horloge
- Si ces événements doivent être séquencés dans un ordre précis, le cycle d'horloge doit être décomposé en sous-cycles
- Un moyen classique pour cela consiste à retarder la copie d'un signal d'horloge primaire afin d'obtenir un signal secondaire décalé en phase



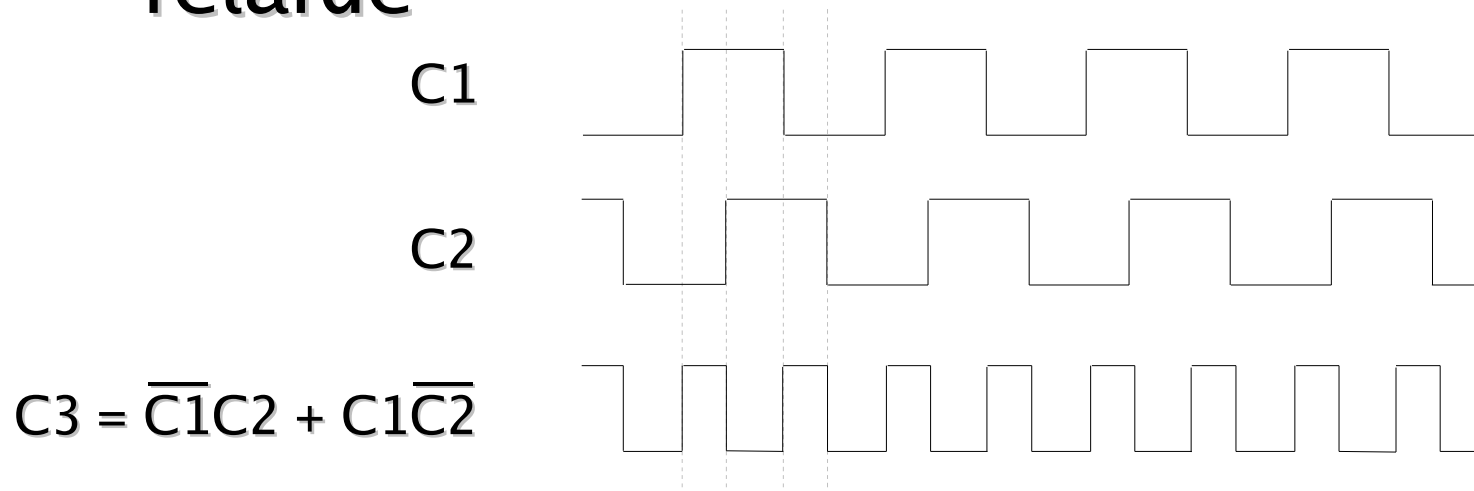
# Cycles et sous-cycles (2)

- On dispose alors de quatre bases de temps au lieu de deux
  - Fronts montant et descendant de C1
  - Fronts montant et descendant de C2



# Cycles et sous-cycles (3)

- Pour certaines fonctions, on s'intéressera plutôt aux intervalles qu'à des instants précis
  - Action possible seulement lorsque C1 est haut
- On peut alors construire des sous-intervalles en s'appuyant sur les signaux original et retardé





# Mémoire (1)

---

- La mémoire principale sert au stockage des programmes et de leurs données
- L'unité élémentaire de mémoire est le bit, pour « *binary digit* » (« chiffre binaire »), prenant deux valeurs, 0 ou 1
- Le stockage physique des bits dépend des technologies employées : différentiels de tension, moments magnétiques, cuvettes ou surfaces planes, émission de photons ou non, etc.

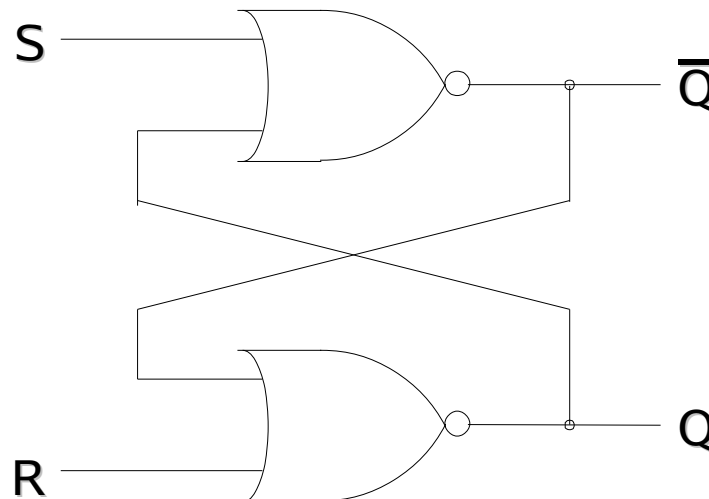
# Mémoire (2)

---

- Pour stocker les informations, il faut un circuit capable de « se souvenir » de la dernière valeur d'entrée qui lui a été fournie
- À la différence d'une fonction combinatoire, sa valeur ne dépend donc pas que de ses valeurs d'entrée courantes
  - Présence de boucles de rétroaction pour préserver l'état courant
- On peut construire un tel circuit à partir de deux portes NAND ou deux portes NOR rebouclées

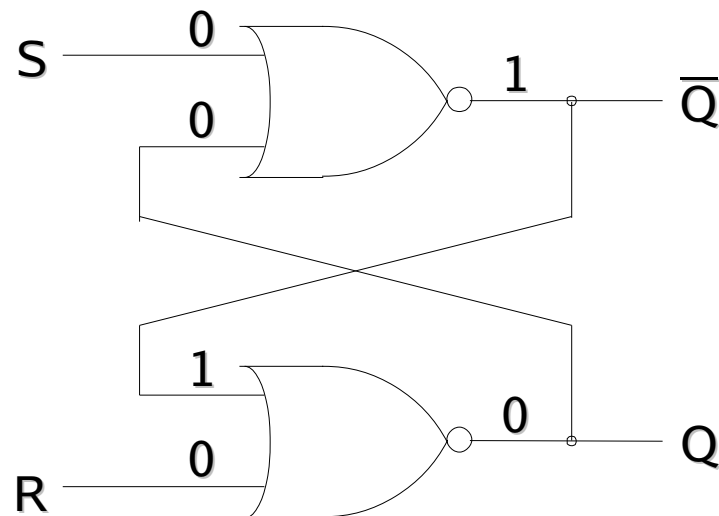
# Bascule SR

- Une bascule SR est une fonction qui a deux entrées et deux sorties
  - Une entrée S pour positionner la bascule
  - Une entrée R pour réinitialiser la bascule
  - Deux sorties Q et  $\bar{Q}$  complémentaires l'une de l'autre



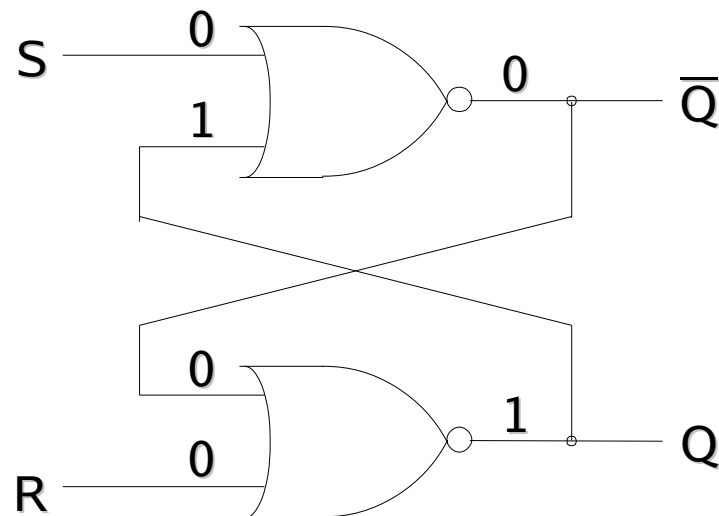
# Bascule SR – État 0

- Si S et R valent 0, et que Q vaut 0, alors :
  - $\bar{Q}$  vaut 1
  - Les deux entrées de la porte du bas sont 0 et 1, donc Q vaut 0
- Cette configuration est cohérente et stable



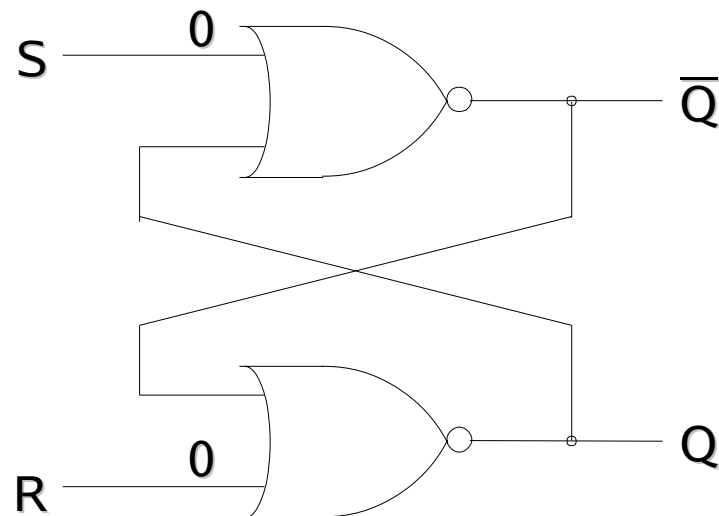
# Bascule SR – État 1

- Si S et R valent 0, et que Q vaut 1, alors :
  - $\bar{Q}$  vaut 0
  - Les deux entrées de la porte du bas sont 0 et 0, donc Q vaut 1
- Cette configuration est cohérente et stable



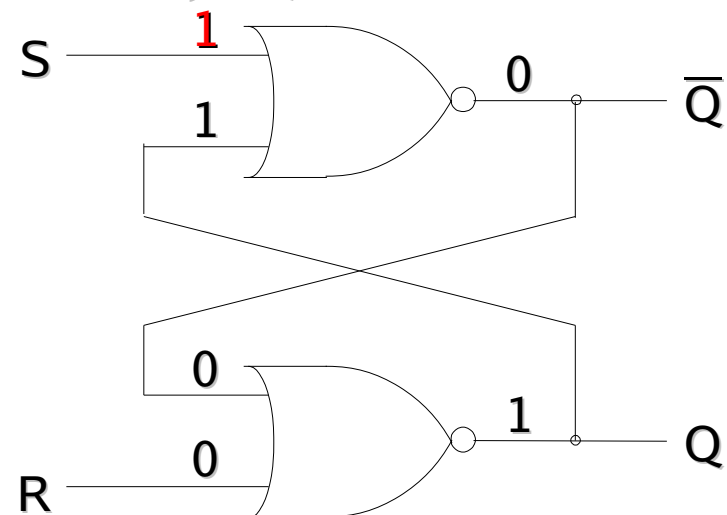
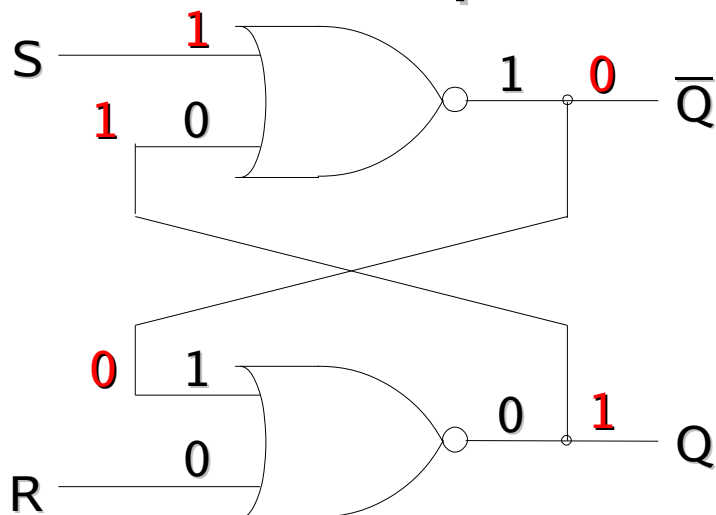
# Bascule SR – États stables

- Lorsque S et R valent 0 :
  - On ne peut avoir simultanément Q à 0 et  $\bar{Q}$  à 0
  - On ne peut avoir simultanément Q à 1 et  $\bar{Q}$  à 1
  - La bascule possède deux états stables, tels que  $Q = 0$  ou  $Q = 1$



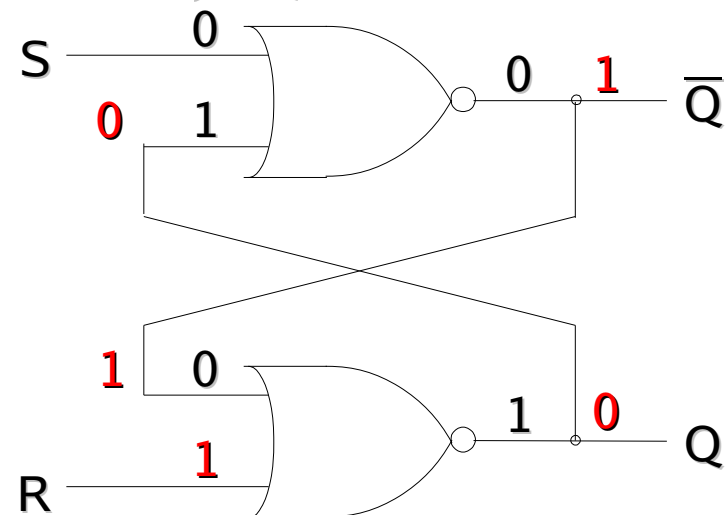
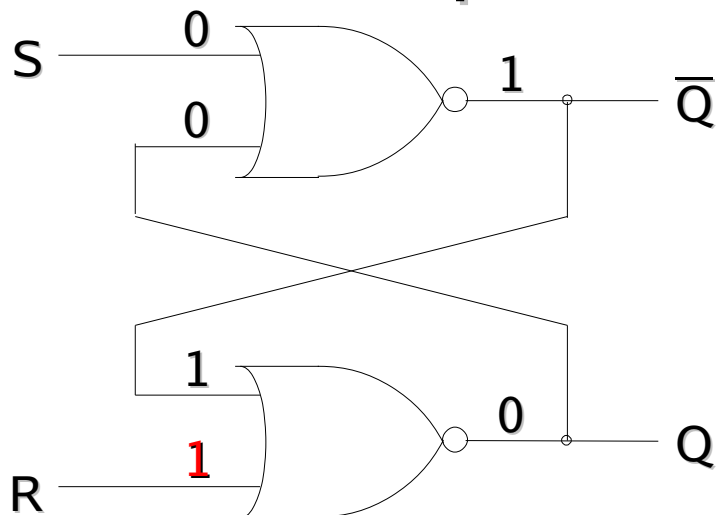
# Basculer SR – Mise à 1

- Lorsque S vaut 1, que Q vaille 0 ou 1 :
  - La sortie de la porte du haut vaut 0, donc  $\bar{Q}$  vaut 0
  - La sortie de la porte du bas vaut 1, donc Q vaut 1
  - Cet état est stable
- Même lorsque S repasse à 0, Q reste à 1



# Bascule SR – Mise à 0

- Lorsque R vaut 1, que Q vaille 0 ou 1 :
  - La sortie de la porte du bas vaut 0, donc Q vaut 0
  - La sortie de la porte du haut vaut 1, donc  $\bar{Q}$  vaut 1
  - Cet état est stable
- Même lorsque R repasse à 0, Q reste à 0





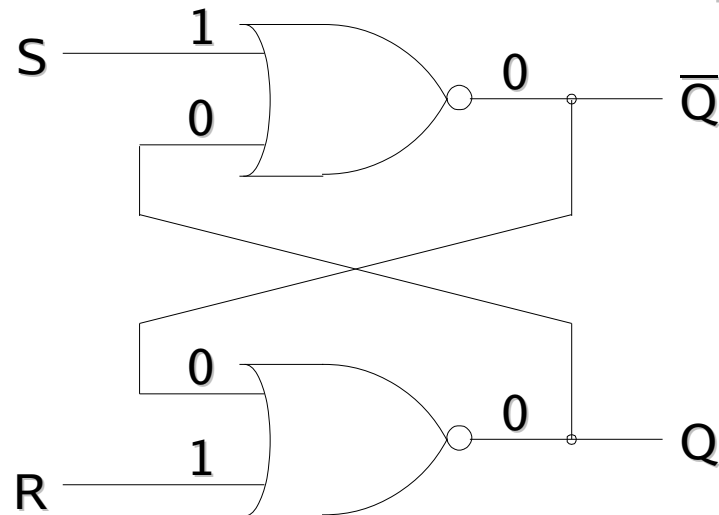
# Bascule SR – Résumé (1)

---

- Lorsque S vaut temporairement 1, la bascule se stabilise dans l'état  $Q = 1$ , quel que soit son état antérieur
- Lorsque R vaut temporairement 1, la bascule se stabilise dans l'état  $Q = 0$ , quel que soit son état antérieur
- La fonction mémorise laquelle des entrées S ou R a été activée en dernier
- Cette fonction peut servir de base à la création de mémoires

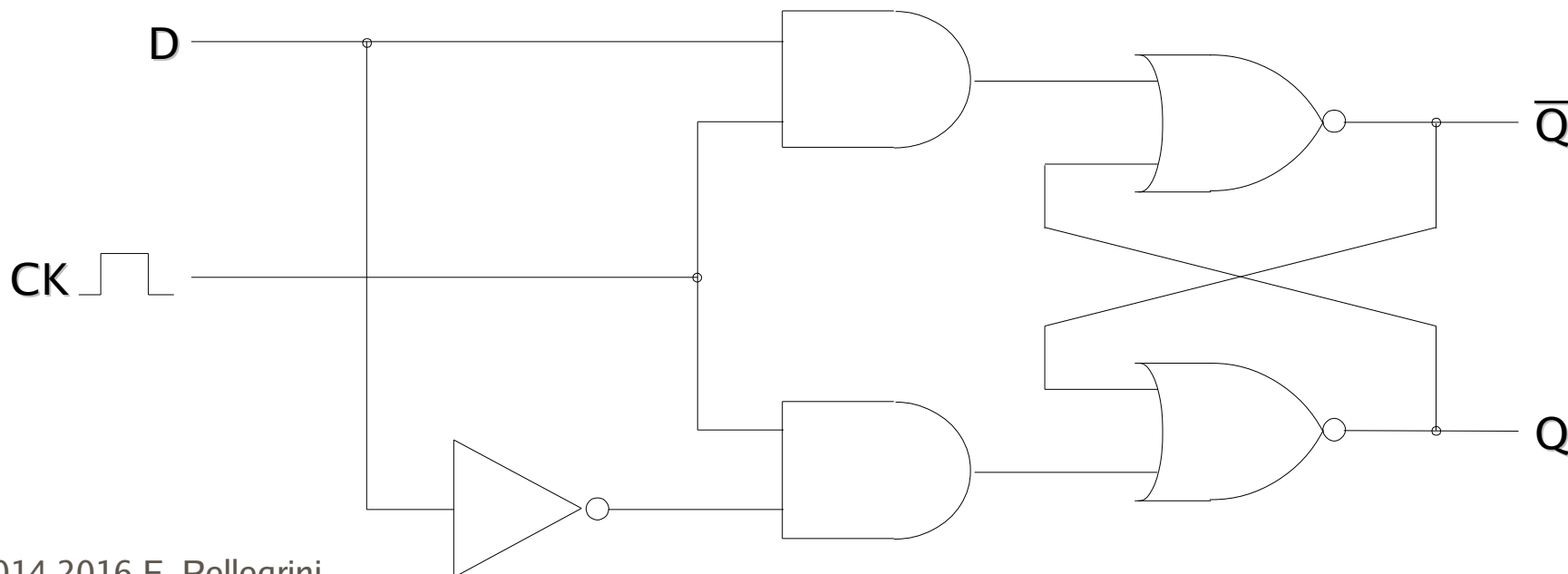
# Bascule SR – Résumé (2)

- Cependant, l'état de la bascule peut être indéterminé
  - Lorsque S et R sont simultanément à 1, on est dans un état stable dans lequel Q et  $\bar{Q}$  valent 0
  - Lorsque S et R repassent simultanément à 0, l'état final de la bascule est non prévisible



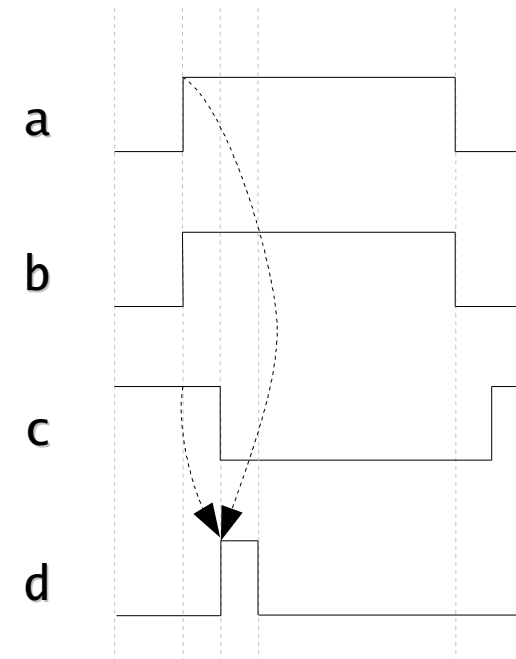
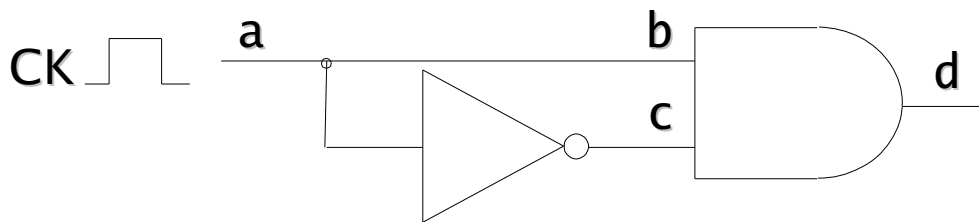
# Bascule D (1)

- Pour éviter cela, on n'a qu'un seul signal D
  - Destiné à l'entrée S, et que l'on inverse pour R
  - On commande la bascule par un signal d'activation
- On a une mémoire à 1 bit



# Bascule D (2)

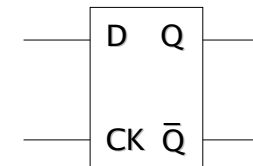
- Pour que l'on soit sûr que la valeur conservée en mémoire soit bien celle présente en début de cycle d'écriture, il faudrait n'autoriser l'écriture qu'au début du cycle, sur le front montant du signal d'écriture



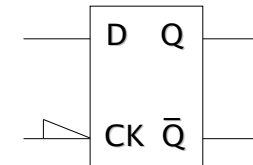
# Bascule D (3)

- Il existe ainsi plusieurs types de bascules D :

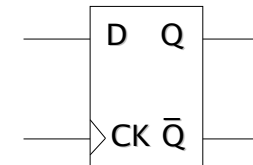
- Activée par CK à l'état haut



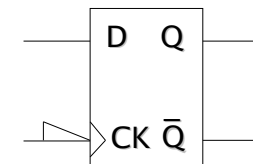
- Activée par CK à l'état bas



- Activée sur front montant



- Activée sur front descendant



# Adressage mémoire (1)

---

- Les mémoires informatiques sont organisées comme un ensemble de cellules pouvant chacune stocker une valeur numérique
- Chaque cellule possède un numéro unique, appelé adresse, auquel les programmes peuvent se référer
- Toutes les cellules d'une mémoire contiennent le même nombre de bits
- Une cellule de  $n$  bits peut stocker  $2^n$  valeurs numériques différentes

# Adressage mémoire (2)

---

- Deux cellules mémoire adjacentes ont des adresses mémoires consécutives
- Les ordinateurs, basés sur le système binaire, représentent également les adresses sous forme binaire
- Une adresse sur  $m$  bits peut adresser  $2^m$  cellules distinctes, indépendamment du nombre de bits contenus dans chaque cellule

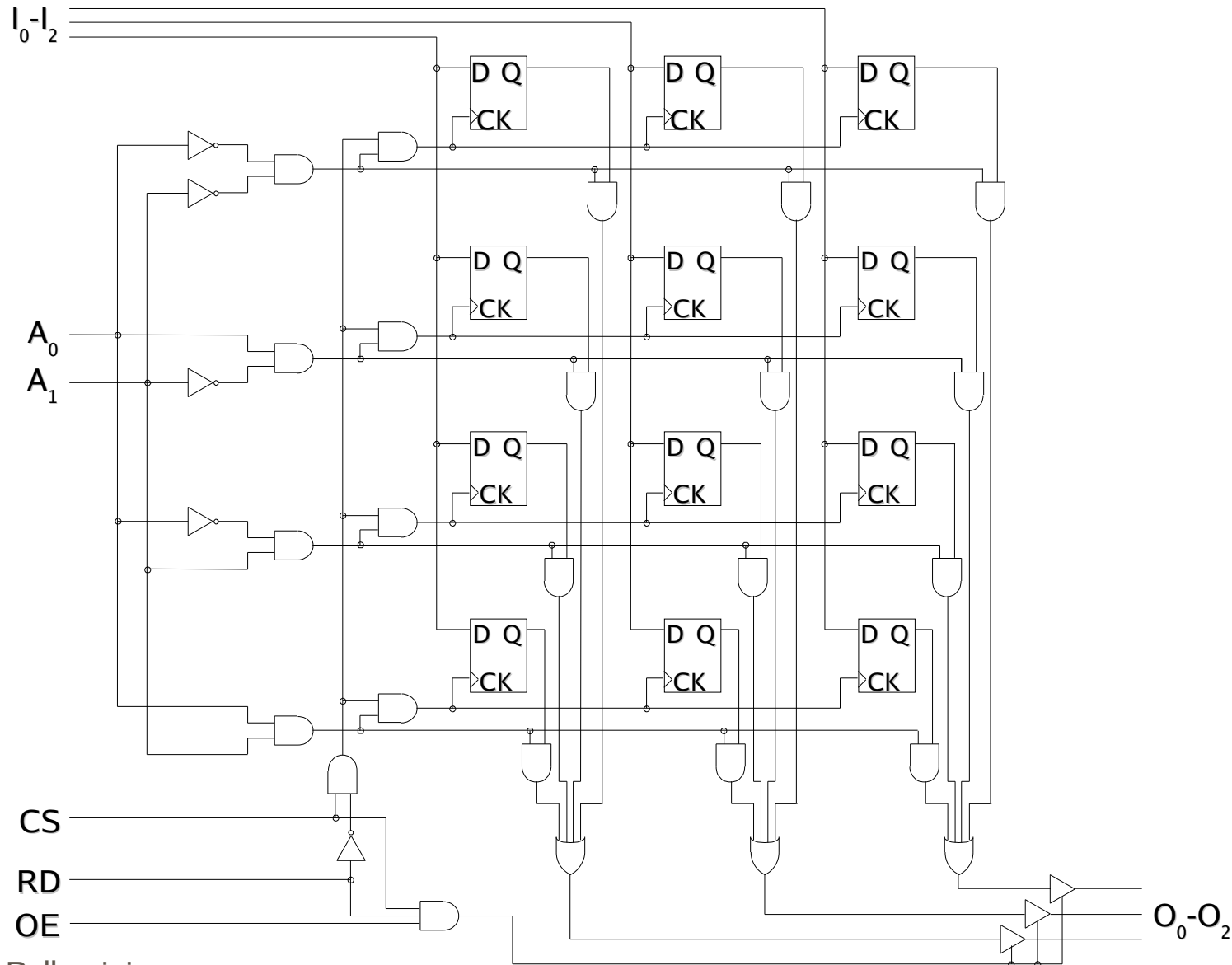
# Adressage mémoire (3)

---

- La cellule est la plus petite unité mémoire pouvant être adressée
  - Il y a maintenant consensus autour d'une cellule à 8 bits, appelée « octet » (« byte » en anglais)
- Afin d'être plus efficaces, les unités de traitement ne manipulent plus des octets individuels mais des mots de plusieurs octets
  - 4 octets par mot pour une machine 32 bits
- La plupart des mémoires travaillent aussi par mots



# Principe d'un circuit mémoire (1)



# Principe d'un circuit mémoire (2)

- Mémoire à 4 mots de 3 bits
- Ce circuit possède trois broches de commande
  - CS (« *chip select* ») : actif pour sélectionner ce circuit mémoire
  - RD (« *read* ») : positionné à 1 si l'on souhaite réaliser une lecture, et à 0 si l'on souhaite une écriture
  - OE (« *output enable* ») : positionné à 1 pour activer les lignes de sortie
    - Utilise des interrupteurs à trois états (0, 1, déconnecté)
    - Permet de connecter  $I_0-I_2$  et  $O_0-O_2$  sur les mêmes lignes

de données

# Types de mémoire

---

- Plusieurs critères caractérisent les mémoires
  - Type d'accès
    - Accès aléatoire : RAM R/W, (((E)E)P)ROM, Flash
    - FIFO : registres à décalage
  - Possibilité d'écriture
    - Pas : ROM
    - Unique : PROM
    - Multiple : RAM R/W, (E)EPROM, Flash
  - Volatilité
    - Les données stockées ne sont conservées que tant que la mémoire est alimentée électriquement

# Mémoire RAM (1)

---

- « *Random Access Memory* »
  - Les mots de la mémoire peuvent être accédés sur demande dans n'importe quel ordre
- Cette catégorie comprend en théorie toutes les mémoires à accès aléatoire telles que mémoires volatiles, ((E)E)P)ROM, Flash, etc.
- Dans le langage courant ce terme est utilisé pour désigner uniquement la mémoire volatile

# Mémoire RAM (2)

- Variétés principales de RAM R/W volatiles
  - RAM statique
    - Circuits actifs à base de portes logiques rebouclées
    - Conservent leurs valeurs sans intervention particulière
  - RAM dynamique
    - Basée sur des petits condensateurs, moins gourmands en place et en consommation électrique
    - Nécessite un rafraîchissement régulier des charges
- Variétés principales de RAM R/W non volatiles
  - EEPROM, Flash : Stockage par charges électriques
  - M-RAM : Stockage magnétique

# Mémoire ROM

---

- « *Read Only Memory* » (« Mémoire morte »)
  - Les données stockées perdurent même quand la mémoire n'est pas alimentée
  - Le contenu, figé à la fabrication, ne peut plus être modifié d'aucune façon
  - Analogue à l'implantation d'une fonction booléenne dépendant des valeurs d'adresses fournies
- Coûteuse du fait de la fabrication en petite série

# Mémoire PROM

---

- « Programmable ROM »
- Les ROMs sont trop longues à faire fabriquer par rapport aux cycles de développement des équipements
- La PROM, livrée vierge (tous bits à 1), peut être programmée avec un équipement adapté
  - Destruction de mini-fusibles par surtension
  - Une seule écriture possible

# Mémoire EPROM

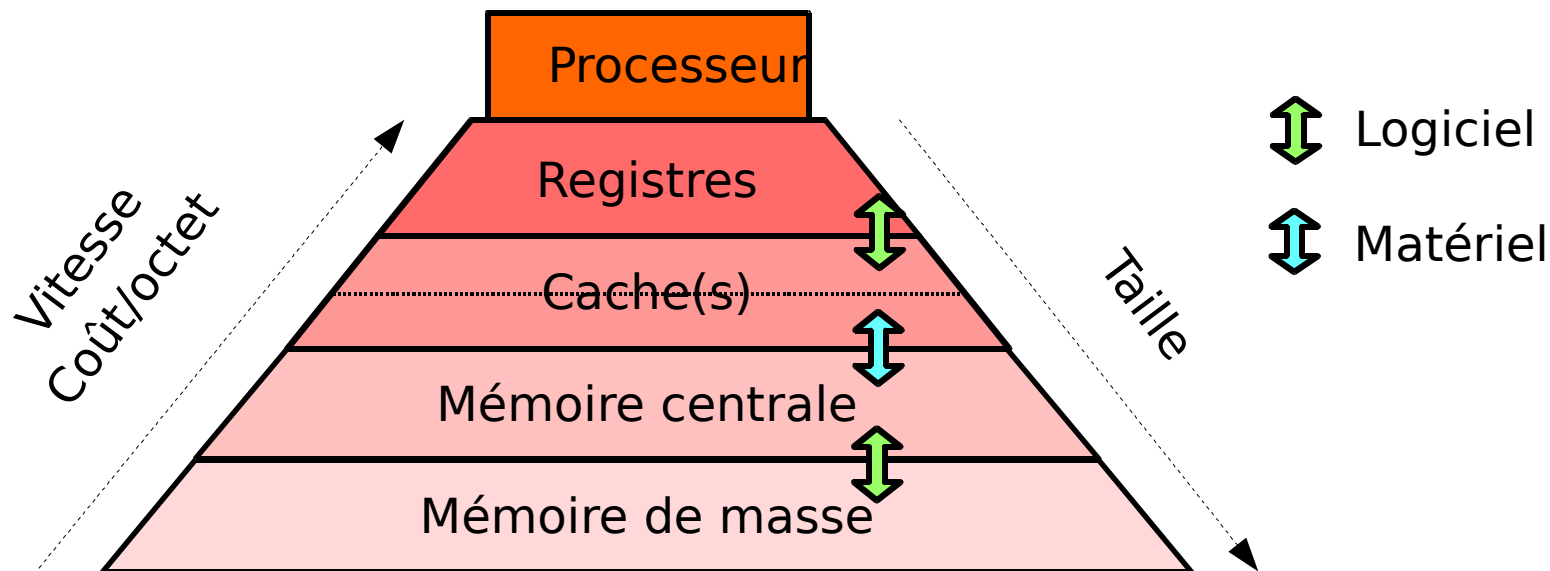
---

- Les mémoires PROM sont encore trop chères
  - Grande consommation lors des phases de développement
- Les mémoires EPROM peuvent être réutilisées en les réinitialisant par exposition aux rayons ultra-violets
  - Petite fenêtre en mica sur le boîtier (mais pastille adhésive pour éviter les UV des tubes fluorescents)
- Les mémoires EEPROM et Flash sont effaçables électriquement



# Hiérarchie mémoire (1)

- La mémoire rapide est très chère, consomme beaucoup et est donc de taille limitée
- On résout ce problème au moyen d'une hiérarchie mémoire



# Hiérarchie mémoire (2)

---

- La hiérarchie mémoire fonctionne grâce aux principes de localité :
  - Localité temporelle : plus un mot mémoire a été accédé récemment, plus il est probable qu'il soit ré-accédé à nouveau
  - Localité spatiale : plus un mot mémoire est proche du dernier mot mémoire accédé, plus il est probable qu'il soit accédé
- Les caches tirent parti de ce principe
  - Sauvegardent les informations les plus récemment accédées, en cas de ré-accès

# Paradigmes architecturaux

---

- L'augmentation continue de la vitesse de traitement du cycle du chemin de données provient de la mise en oeuvre d'un ensemble de principes généraux de conception efficaces
  - Simplification des jeux d'instructions
  - Utilisation du parallélisme au niveau des instructions (« *Instruction-Level Parallelism* », ou ILP)
  - Apparition des architectures multi-cœurs

# RISC et CISC (1)

---

- Plus les instructions sont simples à décoder, plus elles pourront être exécutées rapidement
- Après une tendance à la complexification des jeux d'instructions (« *Complex Instruction Set Computer* », ou CISC), pour économiser la mémoire, on a conçu à nouveau des processeurs au jeu d'instructions moins expressif mais pouvant s'exécuter beaucoup plus rapidement (« *Reduced Instruction Set Computer* », ou RISC)

# RISC et CISC (2)

- Les architectures RISC se distinguent par un certain nombre de choix de conception
  - Toute instruction est traitée directement par des composants matériels (pas de micro-code)
  - Le format des instructions est simple (même taille, peu de types différents)
  - Seules les instructions de chargement et de sauvegarde peuvent accéder à la mémoire
  - Présence d'un grand nombre de registres
  - Architecture orthogonale : toute instruction peut utiliser tout registre : que des registres généralistes

# Micro-architecture (1)

---

- La couche micro-architecture implémente le jeu d'instructions spécifié par la couche d'architecture du jeu d'instructions (ISA) en s'appuyant sur la couche la logique numérique
- La conception de la micro-architecture dépend du jeu d'instruction à implémenter, mais aussi du coût et des performances souhaités
  - Jeux d'instructions plus ou moins complexes (RISC/CISC)
  - Utilisation de l'ILP (« *Instruction-Level Parallelism* »)

# Micro-architecture (2)

- L'exécution d'une instruction peut se décomposer en plusieurs sous-étapes
  - Recherche (« *Fetch* »)
    - Étant donné l'adresse de la prochaine instruction à exécuter, récupération de l'instruction
  - Decodage (« *Decode* »)
    - Détermination du type et de la nature des opérandes
  - Exécution (« *Execute* »)
    - Mise en oeuvre des unités fonctionnelles
  - Terminaison (« *Complete* »)
    - Modification en retour des registres ou de la mémoire

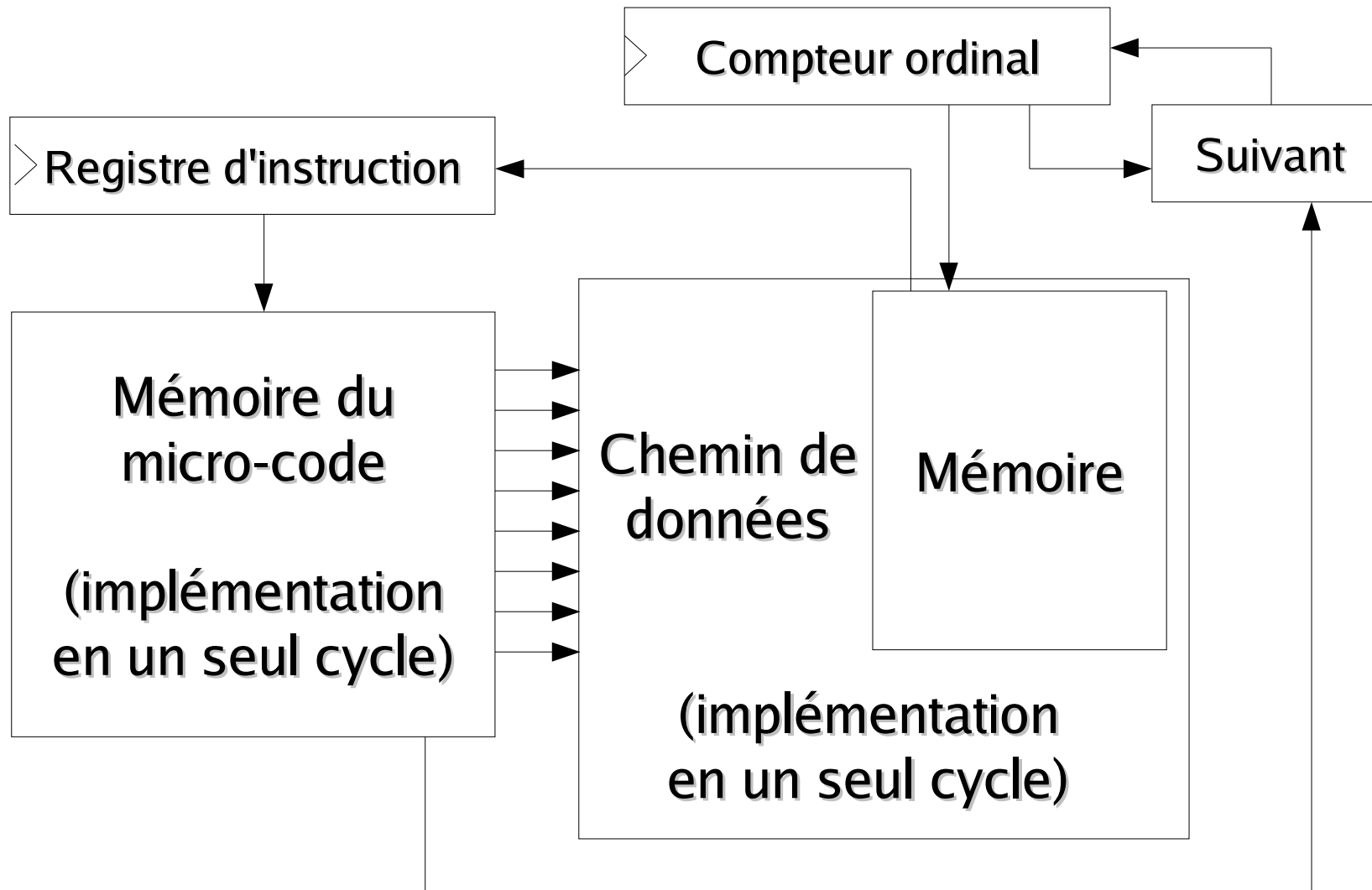
# Micro-architecture (3)

---

- On peut imaginer la conception du niveau micro-architecture comme un problème de programmation
  - Chaque instruction du niveau ISA est une fonction
  - Le programme maître (micro-programme) est une boucle infinie qui détermine à chaque tour la bonne fonction à appeler et l'exécute
  - Le micro-programme dispose de variables d'état accessibles par chacune des fonctions, et modifiées spécifiquement selon la nature de la fonction
    - Compteur ordinal, registres généraux, etc.



# Schéma d'un processeur élémentaire



# Instructions (1)

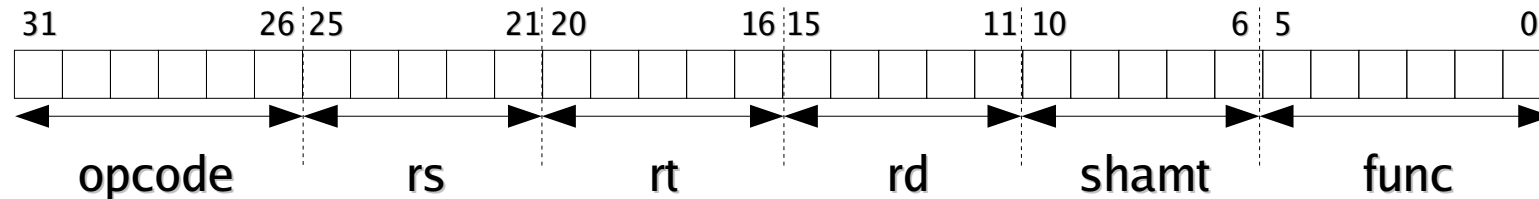
---

- Chaque instruction est composée d'un ou plusieurs champs
- Le premier, appelé « opcode », code le type d'opération réalisée par l'instruction
  - Opération arithmétique, branchement, etc.
- Les autres champs, optionnels, spécifient les opérandes de l'instruction
  - Registres source et destination des données à traiter
  - Adresse mémoire des données à lire ou écrire, etc.

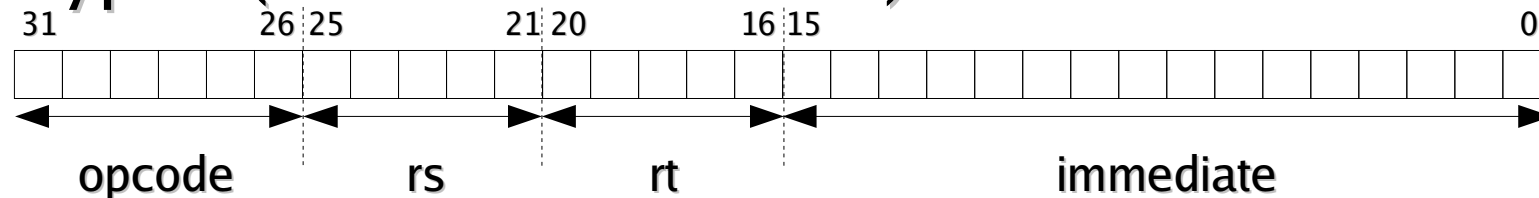
# Instructions (2)

- Exemple : format des instructions RISC MIPS

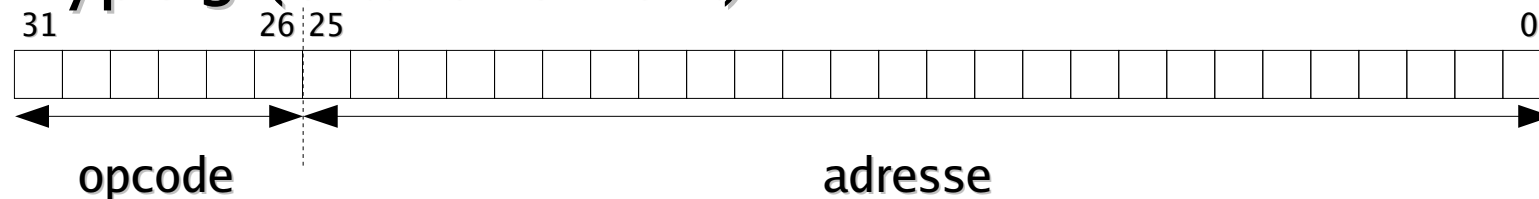
- Type R (registre)



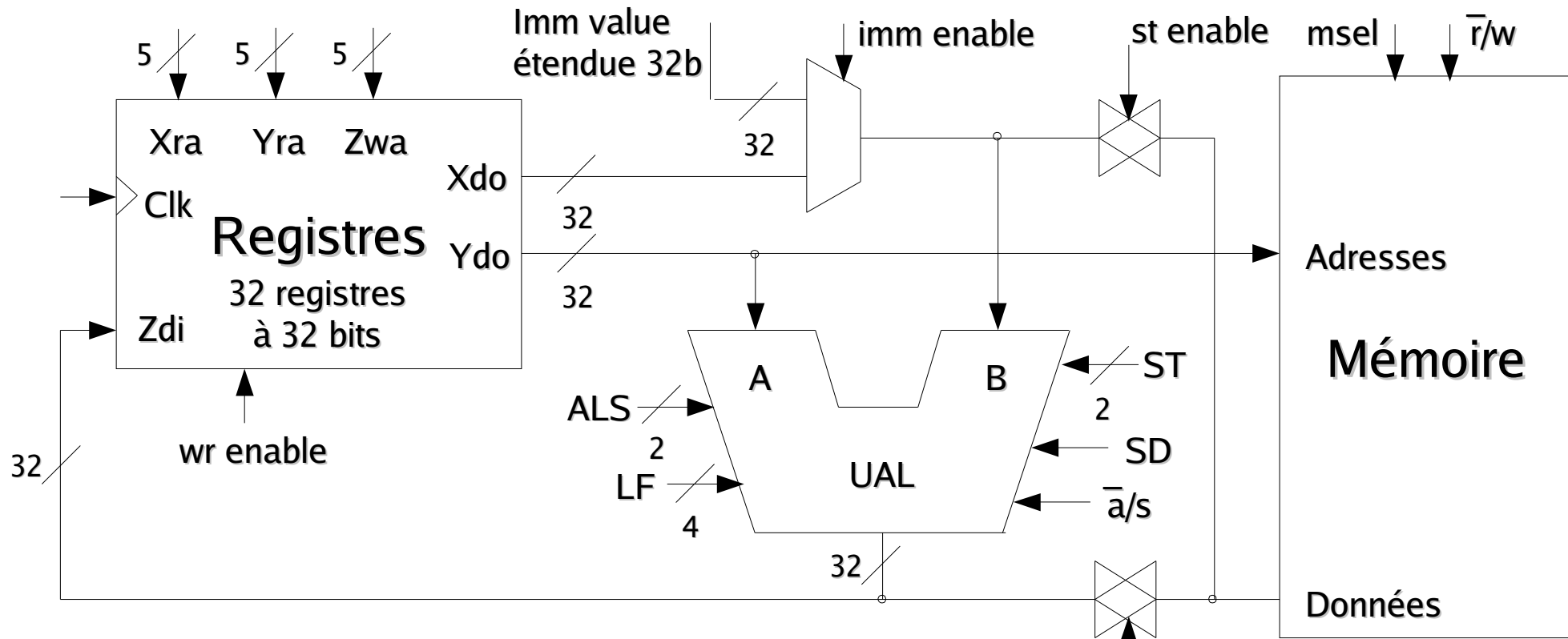
- Type I (donnée immédiate)



- Type J (branchement)



# Contrôle du chemin de données (1)



Id enable

ALS	
00	: Arithmétique
01	: Logique
10	: Décalage
11	: Désactivée

LF
0001 : AND
0011 : A
0101 : B
0110 : XOR
0111 : OR

ST
00 : Pas de décalage
01 : Arithmétique
10 : Logique
11 : Rotation

SD	
0	Décalage à gauche
1	Décalage à droite

$\bar{a}/s$   
 0 : Ajoute  
 1 : Soustrait

# Contrôle du chemin de données (2)

- La logique de contrôle du micro-code associe à chaque (micro-)instruction un mot binaire commandant le chemin de données
  - Exemple : mise à zéro des mots mémoire situés aux adresses 0x0100 et 0x0104

## Signaux de contrôle du chemin de données

Instruction	X (5)	Y (5)	Z (5)	we	imm_e	imm_val	ALS	a/s	LF	ST	SD	ld_e	st_e	r/w	msel
li r1,100	x	x	00001	1	1	0x0100	01	x	0101	x	x	0	0	x	0
sw r0,(r1)	00000	00001	x	0	0	x	11	x	x	x	x	0	1	1	1
add r1,r1,4	x	00001	00001	1	1	0x0004	00	0	x	x	x	0	0	x	0
sw r0,(r1)	00001	00000	x	0	0	x	11	x	x	x	x	0	1	1	1

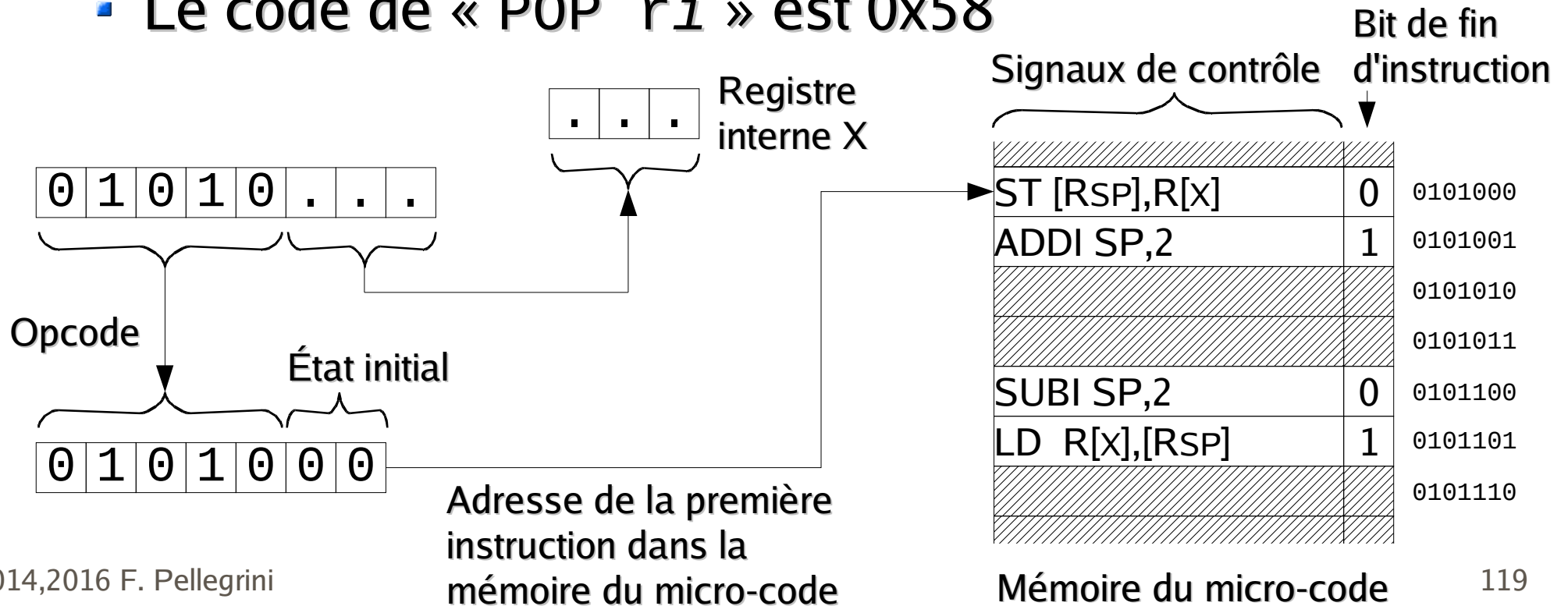
# Interprétation du micro-code (1)

---

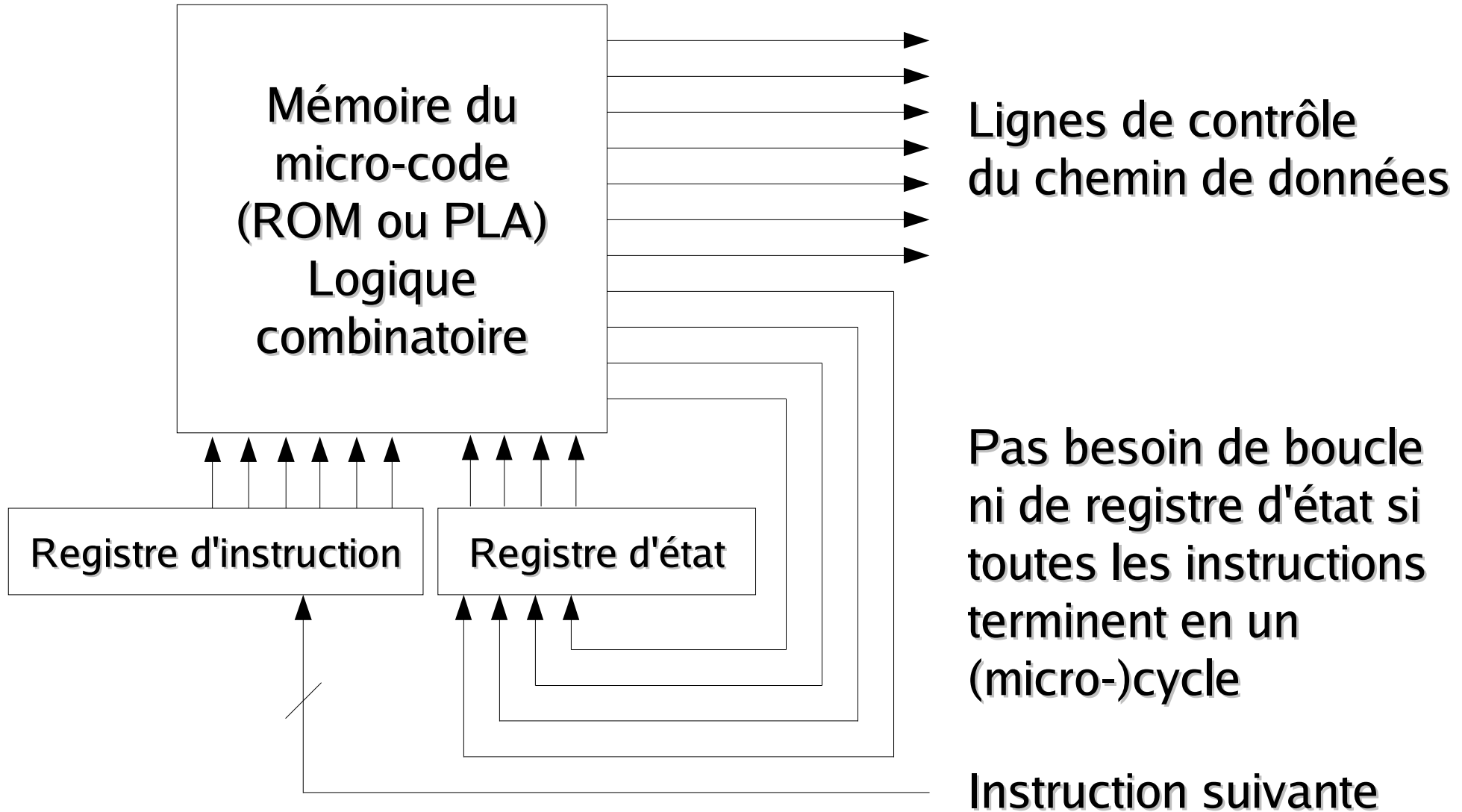
- Dans le cas d'un jeu d'instructions de type CISC, une instruction ISA doit être traduite en plusieurs micro-instructions
  - Cas des instructions « REP SCAS » des x86
- Chaque micro-instruction
  - S'exécute en un cycle élémentaire
  - Spécifie exactement les signaux de contrôle des différentes unités fonctionnelles
- Nécessité d'un séquenceur de micro-instructions
  - Machine d'états finis implémentable en ROM

# Interprétation du micro-code (2)

- Exemple figuratif de micro-codage pour l'instruction « PUSH  $ri$  » du processeur 8086
  - Code machine 0x50 + n° du registre sur 3 bits
  - Le code de « POP  $ri$  » est 0x58



# Interprétation du micro-code (3)





# Pile (1)

---

- Presque tous les langages de programmation incluent le concept de procédure disposant de paramètres d'appel et de variables locales
  - Ces variables peuvent être accédées pendant l'exécution de la procédure mais pas depuis la procédure appelante
  - Elles ne peuvent résider à une adresse absolue en mémoire, car cela empêcherait la réentrance
- Nécessité de créer dynamiquement des instances de ces variables lors des appels de procédures et de les supprimer à la fin

# Pile (2)

---

- Une pile est une zone de la mémoire que l'on n'accède jamais de façon absolue mais toujours relativement à un registre
- Gérée au moyen d'un registre dédié, le pointeur de pile (« *Stack Pointer* », ou SP)
  - Pointe sur le dernier mot mémoire alloué
  - Instructions dédiées à l'empilage et au dépilage :
    - De données : push/pop
    - D'adresses de retour : call/ret

# Architecture du jeu d'instructions (1)

---

- La couche ISA (« *Instruction Set Architecture* ») définit l'architecture fonctionnelle de l'ordinateur
- Sert d'interface entre les couches logicielles et le matériel sous-jacent
- Définit le jeu d'instructions utilisable pour coder les programmes, qui peut être :
  - Directement implémenté de façon matérielle
    - Pas de registre d'état interne servant de compteur ordinal pour l'exécution des micro-instructions
  - Implémenté sous forme micro-programmée

# Architecture du jeu d'instructions (2)

---

- Le jeu d'instructions est indépendant de considérations d'implémentation telles que superscalarité, pipe-lining, etc.
- Liberté d'implémentation en fonction des coûts de conception et de fabrication, de la complexité de réalisation, et donc du coût souhaité
  - Définition de familles de processeurs en fonction des applications visées (du téléphone portable au super-calculateur)
- Nécessité pour le compilateur de connaître l'implémentation de la machine cible pour générer du code efficace

# Types de données (1)

---

- Le niveau ISA définit les types de données gérés nativement par le jeu d'instructions
  - Autorise l'implémentation matérielle des types considérés
  - Définit la nature (entier, flottant, caractère) et la précision des types supportés
- Le programmeur n'est pas libre de choisir le format de ses données s'il veut bénéficier du support matériel offert par la couche ISA

# Types de données (2)

---

- Les types de données les plus couramment implémentés dans les jeux d'instructions sont :
  - Type entier
  - Type flottant
  - Type caractère

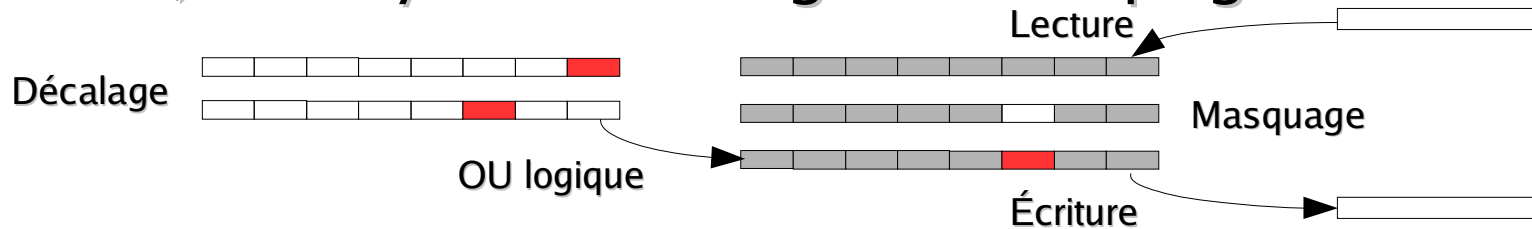
# Types de données entiers (1)

---

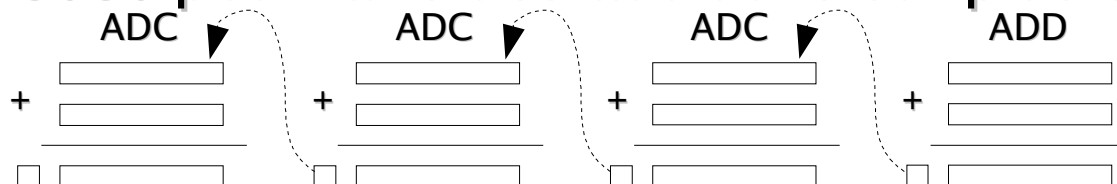
- Le type entier est toujours disponible
  - Sert au fonctionnement de la couche micro-architecture
- Toutes les architectures disposent de types entiers signés
  - Presque toujours codés en complément à deux
  - Il existe aussi souvent des types non signés
- Disponibles en plusieurs tailles
  - Quelques unes choisies parmi les tailles classiques de 8, 16, 32, 64 bits (jamais de type booléen)

# Types de données entiers (2)

- Les types entiers non supportés :
  - Soit doivent être émulés de façon logicielle
    - Cas du type caractère (8 bits) sur le CRAY-1 (mots de 64 bits) au moyen de décalages et masquages de bits



- Soit font l'objet d'un support partiel par le matériel
  - Cas des instructions ADD/ADC (« *add with carry* ») sur le 8080 pour faire des additions sur plus d'un octet





# Types de données flottants

---

- Les types flottants sont très souvent disponibles
  - Sauf sur les processeurs bas de gamme, où les nombres flottants sont émulés logiciellement
- Disponibles en plusieurs tailles
  - 32, 64, 80, ou 128 bits
- Souvent gérés par des registres séparés
  - Cas des 8 registres flottants de l'architecture x86, organisés sous forme de pile

# Types de données caractères

---

- La plupart des ordinateurs sont utilisés pour des tâches de bureautique ou de gestion de bases de données manipulant des données textuelles
- Quelques jeux d'instructions proposent des instructions de manipulation de suites de caractères
  - Caractères émulsés par des octets (ASCII), des mots de 16 bits (Unicode), voire de 32 bits
  - Cas de l'architecture x86 avec les instructions micro-codées CMPS, SCAS, STOS, etc. utilisables avec les préfixes REP, REPZ, REPNZ

# Type de données booléen

---

- Il n'existe pas de type booléen natif sur les processeurs
  - Pas de possibilité d'adressage en mémoire
- Le type booléen est généralement émulé par un type entier (octet ou mot)
  - Valeur fausse si la valeur entière est zéro
  - Valeur vraie sinon
  - Cas de l'instruction `beq r1, r0, addr` et `bne` du jeu d'instructions MIPS, compatibles avec cette convention de codage

# Type de données référence

---

- Une référence est un pointeur sur une adresse
- Elle est émulée par un type de données entier
  - Soit registres entiers généralistes
  - Soit registres entiers spécifiques d'adresses
    - Cas du CRAY-1 : 8 registres d'adresses sur 24 bits et 8 registres entiers sur 64 bits
- Utilisation de ces registres pour accéder aux données en mémoire, en fonction des modes d'adressage disponibles
  - Cas des registres SP et BP de gestion de la pile

# Format des instructions (1)

---

- Chaque instruction est composée d'un ou plusieurs champs
- Le premier, appelé « opcode », code le type d'opération réalisée par l'instruction
  - Opération arithmétique, branchement, etc.
- Les autres champs, optionnels, qui spécifient où rechercher les opérandes de l'instruction, sont appelés « adresses »
  - Les instructions ont toujours de zéro à trois adresses

# Format des instructions (2)

- Différentes façons de concevoir l'adressage
  - Architecture à trois adresses : on a deux adresses source et une adresse destination, qui peut être équivalente à l'une des adresses source
    - Cas de l'architecture MIPS : instruction `add s1, s2, dst` pouvant être utilisée en `add r1, r2, r1`
  - Architecture à deux adresses : on a toujours une adresse source, non modifiée, et une adresse destination, modifiée ou mise à jour selon que l'opération fait ou non intervenir son ancienne valeur
    - Cas de l'architecture x86 : instructions `MOV dst, src` ou `ADD dst, src`

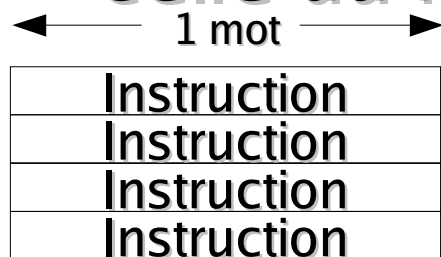
# Format des instructions (3)

---

- Différentes façons de concevoir l'adressage
  - Architecture à une adresse : toutes les instructions de calcul opèrent entre une adresse et un registre unique, appelé « accumulateur »
    - Anciennes architectures de type 8008
    - Trop de transferts entre l'accumulateur et la mémoire
  - Architecture à zéro adresses : les adresses des opérandes sont implicites, situées au sommet d'une pile d'opérandes, où seront placés les résultats
    - Cas de l'architecture JVM

# Format des instructions (4)

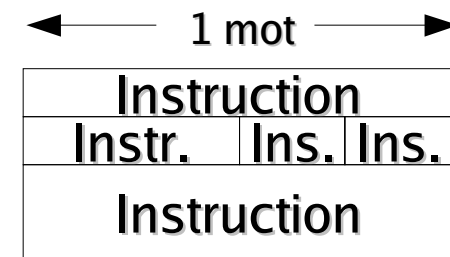
- Les instructions peuvent soit toutes être de la même taille, soit être de tailles différentes
  - Avoir toutes les instructions de même taille facilite le décodage mais consomme plus de mémoire
- La taille des instructions peut être plus petite, plus grande, ou de longueur équivalente à celle du mot mémoire



MIPS (32 bits)



IA-64 (64 bits)



X86 (32 bits : IA-32)



# Format des instructions (5)

---

- Un jeu d'instructions est dit « orthogonal » si, quand une instruction opère sur un registre, elle peut opérer sur l'ensemble des registres de même type (registres entiers, registres flottants)
  - Facilite le décodage des instructions
  - Implémenté naturellement au sein des architectures de type RISC

# Modes d'adressage

---

- Les modes d'adressage sont les différentes manières dont on peut accéder aux opérandes des instructions
  - Adressage immédiat
  - Adressage direct
  - Adressage registre
  - Adressage indirect par registre
  - Adressage indexé
  - Adressage basé indexé

# Adressage immédiat

- Le plus simple pour une instruction est que sa partie d'adresse contienne directement la valeur de l'opérande
  - Réservé aux constantes
  - Aucun accès mémoire supplémentaire nécessaire
- Exemples
  - Branchements : l'adresse (déplacement relatif ou absolu) est spécifiée dans le corps de l'instruction :  
b 0C2F4
  - Chargement de registres : li r1, 100

# Adressage direct

---

- Une méthode pour accéder à une valeur en mémoire consiste à donner son adresse pour qu'on puisse y accéder directement
  - On accédera toujours à la même zone mémoire
  - Réservé aux variables globales dont les adresses sont connues à la compilation

# Adressage registre

---

- Conceptuellement équivalent à l'adressage direct, mais on spécifie un numéro de registre plutôt qu'un numéro de mot mémoire
  - Mode le plus couramment utilisé
    - Les accès aux registres sont très rapides
    - Les numéros de registres se codent sur peu de bits (compacité des instructions à plusieurs adresses)
  - Une grande partie du travail des compilateurs consiste à déterminer quelles variables seront placées dans quels registres à chaque instant, afin de diminuer les temps d'accès et donc d'exécution

# Adressage indirect par registre

---

- L'opérande spécifié provient de la mémoire ou y sera stockée, mais son adresse est contenue dans un registre de numéro donné plutôt que codée explicitement dans le corps de l'instruction
  - Le registre est un pointeur sur l'opérande
  - On peut référencer une zone mémoire sans avoir à coder son adresse dans l'instruction
  - On peut modifier dynamiquement l'adresse de la zone mémoire référencée en modifiant la valeur du registre

# Adressage indexé

- Ce mode combine les caractéristiques de l'adressage direct et de l'adressage registre
- L'opérande considéré est localisé à une distance fixe de l'adresse fournie par un registre
  - Les champs de l'instruction sont le numéro du registre ainsi que le déplacement relatif (« *offset* ») à ajouter à son contenu
- Exemple : accès aux variables locales et paramètres placés dans la pile, par rapport au registre BP : `MOV AX, (BP+4)`

# Adressage basé indexé

---

- L'adresse mémoire de l'opérande est calculée à partir de la somme des valeurs de deux registres (un registre de base et un registre d'index) ainsi que d'une valeur de déplacement optionnelle
- Exemple : accès aux champs des structures contenues dans un tableau
  - Le registre de base est l'adresse de début du tableau
  - Le registre d'index référence l'adresse de début de la bonne structure par rapport à l'adresse du tableau
  - Le déplacement référence la position du début du champ par rapport au début de la structure



# Types d'instructions

---

- Les instructions de la couche ISA peuvent être groupées en une demi-douzaine de classes, que l'on retrouve sur toutes les architectures
  - Copie de données
  - Calcul
  - Branchements, branchements conditionnels et comparaisons
  - Entrées/sorties et interruptions
  - Gestion de la mémoire

# Instructions de copie de données

---

- Les instructions de copie de données ont deux usages principaux
  - Réaliser l'affectation de valeurs à des variables
    - Recopie de valeurs dans des variables temporaires devant servir à des calculs ultérieurs
  - Placer une copie de valeurs utiles là où elles pourront être accédées le plus efficacement
    - Utilisation des registres plutôt que de la mémoire
- On a toujours des instruction de copie entre registres, ou entre registre et mémoire, mais moins souvent de mémoire à mémoire

# Instructions de calcul (1)

---

- Ces instructions représentent les opérations réalisables par l'unité arithmétique et logique, mais sur des opérandes qui ne sont pas nécessairement tous des registres
  - Calculs entre mémoire et registres (cas du x86)
- Les instructions de calcul les plus couramment utilisées peuvent faire l'objet d'un format abrégé
  - Instruction INC R1 remplaçant la séquence MOV R2, 1 et ADD R1, R2, par exemple

# Instructions de calcul (2)

- Dans une architecture de type « *load/store* », les seules instructions pouvant accéder à la mémoire sont les instructions `load` et `store` de copie entre mémoire et registre
- Les instructions de calcul ne prennent dans ce cas que des opérandes registres
  - Simplifie le format et le décodage des instructions
  - Permet d'optimiser l'utilisation de l'unité arithmétique et logique (pas de cycles d'attente des opérandes mémoire)

# Instructions de branchement (1)

---

- L'instruction de branchement inconditionnel déroute le flot d'exécution du programme vers une adresse donnée
- L'instruction d'appel de sous-programme déroute aussi le flot d'exécution mais en plus sauvegarde l'adresse située après l'instruction afin de permettre le retour à la fonction appelante
  - Sauvegarde dans un registre ou dans la pile

# Instructions de branchement (2)

---

- Les instructions de comparaison et de branchement conditionnel servent à orienter le flot d'exécution en fonction du résultat de l'évaluation d'expressions booléennes
  - Implémentation des tests
  - Implémentation des boucles

# Instructions de branchement (3)

---

- Deux implémentations possibles :
  - Instructions de comparaison et de branchement distinctes utilisant un registre d'état du processeur
    - Cas de l'architecture x86 : instruction CMP mettant à jour les bits Z, S, O du mot d'état programme PSW, et instructions de branchement JEQ, JNE, JGE, etc. les utilisant comme conditions de branchement
  - Instructions de branchement conditionnel prenant en paramètres les noms de deux registres comparés à la volée pour décider du branchement
    - Cas des architecture MIPS et Power : avoir une seule instruction facilite la réorganisation dynamique de code

# Instructions d'entrée/sortie

- Diffèrent considérablement selon l'architecture
  - Mettent en œuvre un ou plusieurs parmi trois schémas d'E/S différents
    - E/S programmées avec attente de disponibilité
      - Très coûteux car le processeur ne fait rien en attendant
      - Cas des instructions IN et OUT de l'architecture x86
    - E/S par interruptions
      - Le périphérique avertit le processeur, au moyen d'une interruption, chaque fois que son état change (coûteux)
    - E/S par DMA (« *Direct Memory Access* »)
- Un circuit spécialisé se charge des échanges de données



# Instructions de gestion de priorité (1)

---

- Les micro-architectures modernes implémentent nativement des mécanismes matériels permettant de distinguer entre deux modes d'exécution
  - Mode non privilégié : accès restreint à la mémoire, interdiction d'exécuter les instructions d'entrées-sorties
  - Mode privilégié : accès à tout l'espace d'adressage et à toutes les instructions
- Instructions spécifiques de passage entre les deux modes

# Instructions de gestion de priorité (2)

---

- Servent à isoler le système d'exploitation des programmes d'application
  - Les appels système s'exécutent en mode privilégié, pour pouvoir accéder à l'ensemble des ressources de la machine
  - Les programmes d'application s'exécutent en mode non privilégié, et ne peuvent donc accéder directement au matériel sans passer par les routines de contrôle d'accès du système
  - Le passage du mode non privilégié au mode privilégié ne peut se faire que de façon strictement contrôlée (traps et interruptions)

# Instructions d'interruption (1)

---

- Les interruptions sont des événements qui, une fois reçus par le processeur, conduisent à l'exécution d'une routine de traitement adaptée
  - L'exécution du programme en cours est suspendue pour exécuter la routine de traitement
  - Analogue à un appel de sous-programme, mais de façon asynchrone
- Il existe plusieurs types d'interruptions, identifiées par leur numéro

# Instructions d'interruption (2)

---

- Les interruptions peuvent être :
  - Asynchrones : interruptions « matérielles » reçues par le processeur par activation de certaines de ses lignes de contrôle
    - Gestion des périphériques
  - Synchrones : interruptions générées par le processeur lui-même :
    - Par exécution d'une instruction spécifique (« trap »)
      - Exemple : l'instruction INT de l'architecture x86
      - Sert à mettre en œuvre les appels système
    - Sur erreur logicielle (erreur d'accès mémoire, de calcul ...)
    - Sert à mettre en œuvre les exceptions

# Instructions d'interruption (3)

---

- Lorsque le processeur accepte d'exécuter une interruption :
  - Il sauvegarde dans la pile l'adresse de la prochaine instruction à exécuter dans le cadre du déroulement normal
  - Il se sert du numéro de l'interruption pour indexer une table contenant les adresses des différentes routines de traitement (« vecteur d'interruptions »)
  - Il se déroute à cette adresse
    - Passage en mode privilégié si le processeur en dispose

# Instructions d'interruption (4)

---

- Au niveau du jeu d'instructions, on trouve donc des instructions
  - Pour générer des interruptions logicielles
  - Pour autoriser ou non l'acceptation des interruptions
    - Ces instructions ne doivent pas être exécutables par les programmes d'application
    - Exécutables seulement en mode privilégié
- La modification du vecteur d'interruptions ne peut se faire qu'en mode privilégié
- Protection par segmentation de la mémoire

# Espace d'adressage

---

- La plupart des couches ISA considèrent la mémoire comme un espace linéaire et continu commençant de l'adresse 0 à l'adresse  $2^{32}-1$  ou  $2^{64}-1$ 
  - En pratique, on n'utilise pas plus de 44 bits d'adresses (adressage de 16 TéraMots)

# Architecture ISA du Pentium II (1)

---

- Architecture appelée IA-32 (ou x86)
- Est le résultat d'une évolution continue depuis le processeur 8 bits 8080
- Maintien d'une compatibilité ascendante permettant encore l'exécution de programmes écrits pour le processeur 16 bits 8086 :
  - Mode réel : le processeur se comporte comme un 8086
  - Mode virtuel : le processeur simule un 8086
  - Mode protégé : utilise l'ensemble du processeur



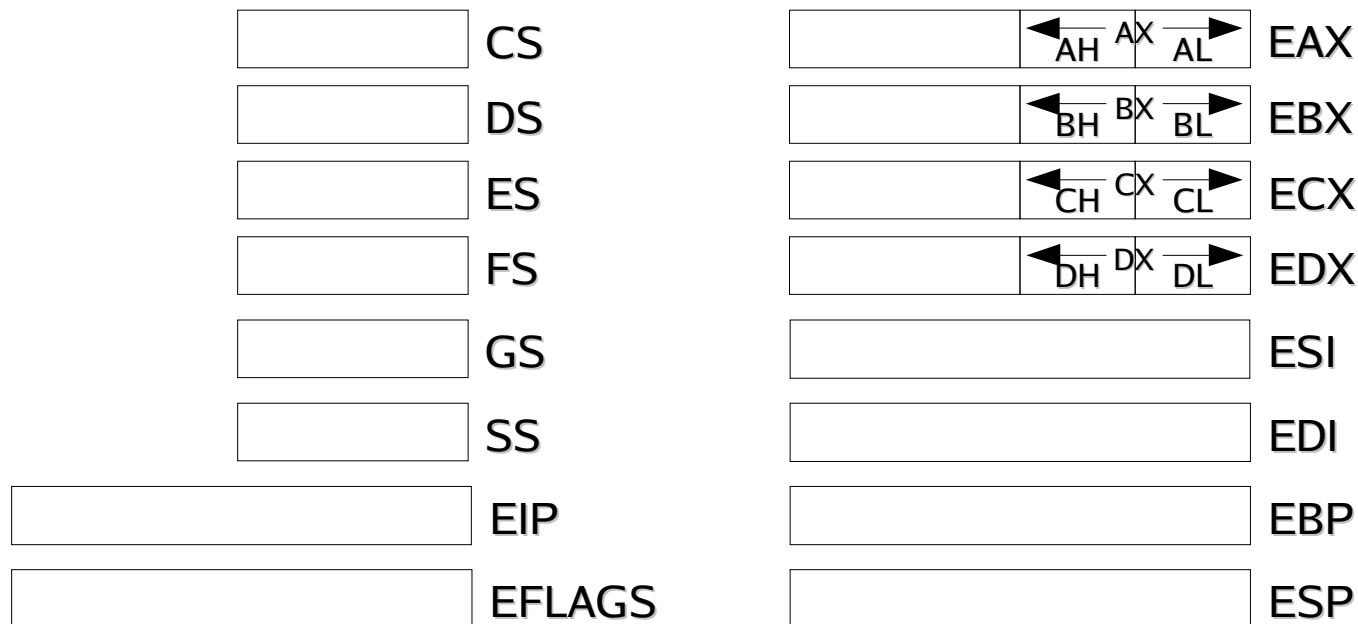
# Architecture ISA du Pentium II (2)

---

- Enrichissement continu du jeu d'instructions :
  - Passage à une architecture 32 bits avec le 80386
  - Ajout des instructions MMX (« *MultiMedia eXtension* ») par Intel
  - Ajout des instructions « 3D Now! » (par AMD) et SSE (« *Streaming SIMD Extension* », par Intel)
  - Passage à une architecture 64 bits avec l'Opteron d'AMD (architecture appelée x86-64 par AMD ou EM64T par Intel)

# Architecture ISA du Pentium II (3)

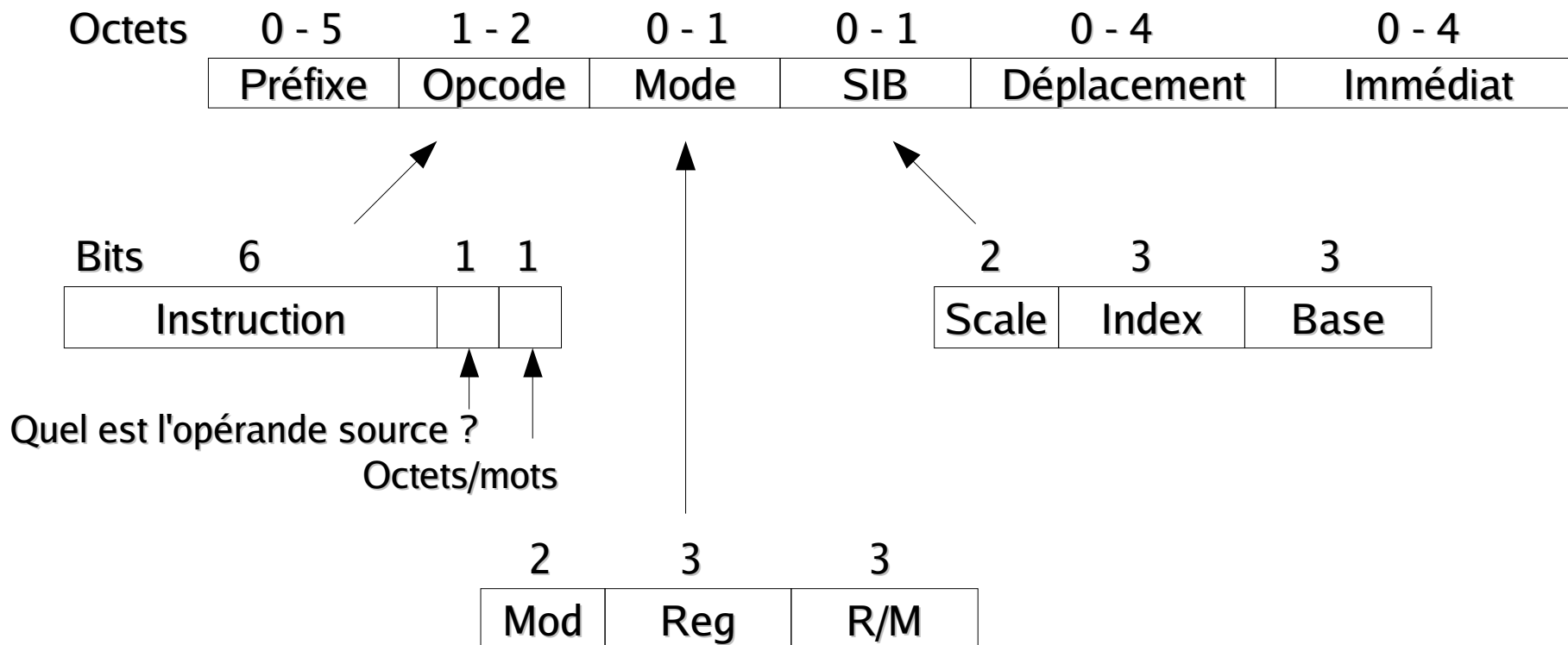
- Architecture à deux adresses, non orthogonale
  - Registres généraux spécialisés



- Mémoire organisée en 16384 segments de  $2^{32}$  octets

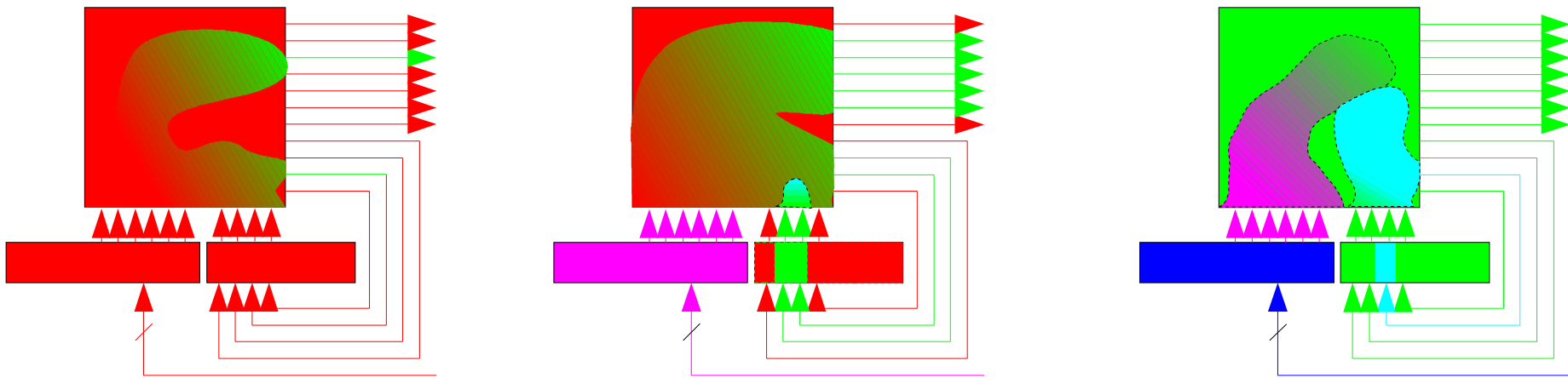
# Architecture ISA du Pentium II (4)

- La structure des instructions est complexe et irrégulière
  - Code opération expansif



# Circuits synchrones (1)

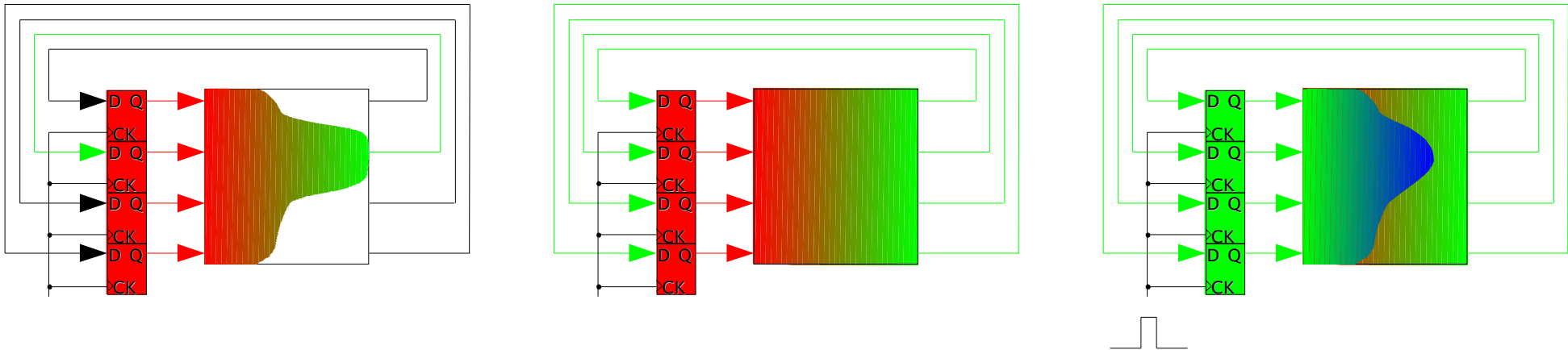
- En l'absence de synchronisation, les résultats des calculs des circuits avec boucle de rétroaction seraient inexploitable car faux



- Il faut mettre en place des « barrières » pour empêcher les résultats du tour courant de « déborder » sur le tour suivant

# Circuits synchrones (2)

- On peut réaliser ces barrières au moyen de bascules D faisant « verrou » (« *latch* »)



- Les bascules doivent être pilotées par une horloge

# Circuits synchrones (3)

---

- La fréquence de l'horloge doit être choisie de telle sorte que :
  - Le temps de cycle permette au circuit de se stabiliser
    - Dépend de la longueur du chemin critique du circuit
  - Le temps d'impulsion soit suffisamment court pour éviter toute interférence entre phases de calcul
    - Dépend de la longueur du plus court chemin
    - Impulsion la plus courte possible pour éviter tout problème

# Circuits synchrones (3)

---

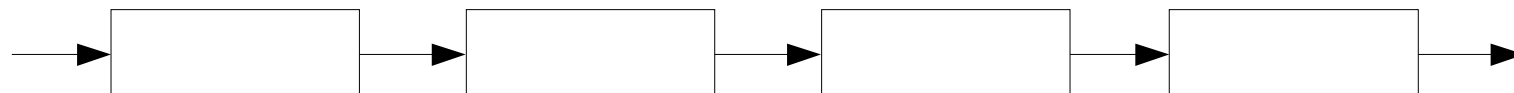
- La fréquence de l'horloge doit être choisie de telle sorte que :
  - Le temps de cycle permette au circuit de se stabiliser
    - Dépend de la longueur du chemin critique du circuit
  - Le temps d'impulsion soit suffisamment court pour éviter toute interférence entre phases de calcul
    - Dépend de la longueur du plus court chemin
    - Impulsion la plus courte possible pour éviter tout problème

# Pipe-line (1)

- Lorsqu'un même traitement se répète dans le temps, et peut être découpé en sous-tâches élémentaires, on peut mettre en place une chaîne de traitement appelée pipe-line
  - Le nombre de sous-unités fonctionnelles est appelé nombre d'étages du pipe-line



Unité fonctionnelle non pipelinée



Unité fonctionnelle pipelinée à 4 étages



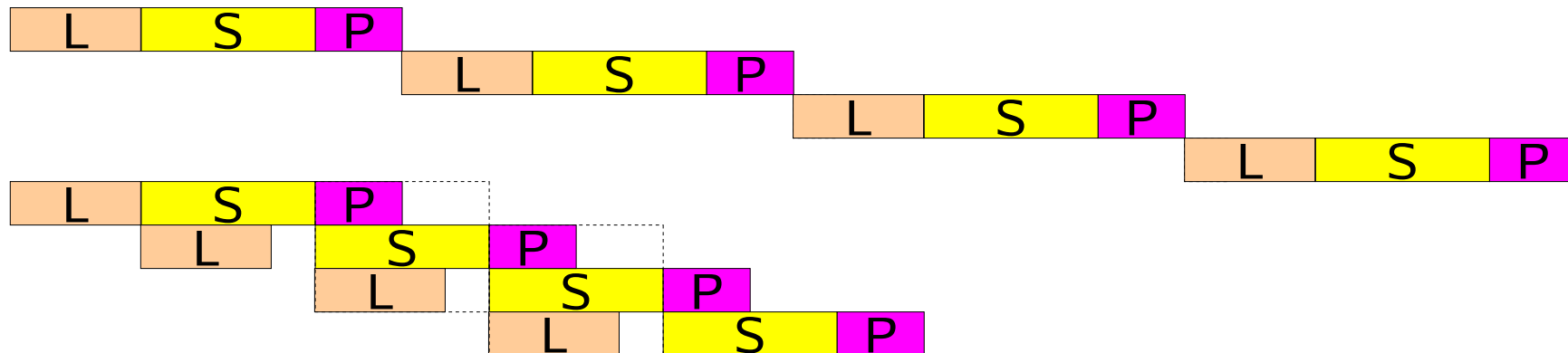
# Pipe-line (2)

- Exemple : le lavomatique

- Lavage : 30 minutes L

- Séchage : 40 minutes S

- Pliage : 20 minutes P



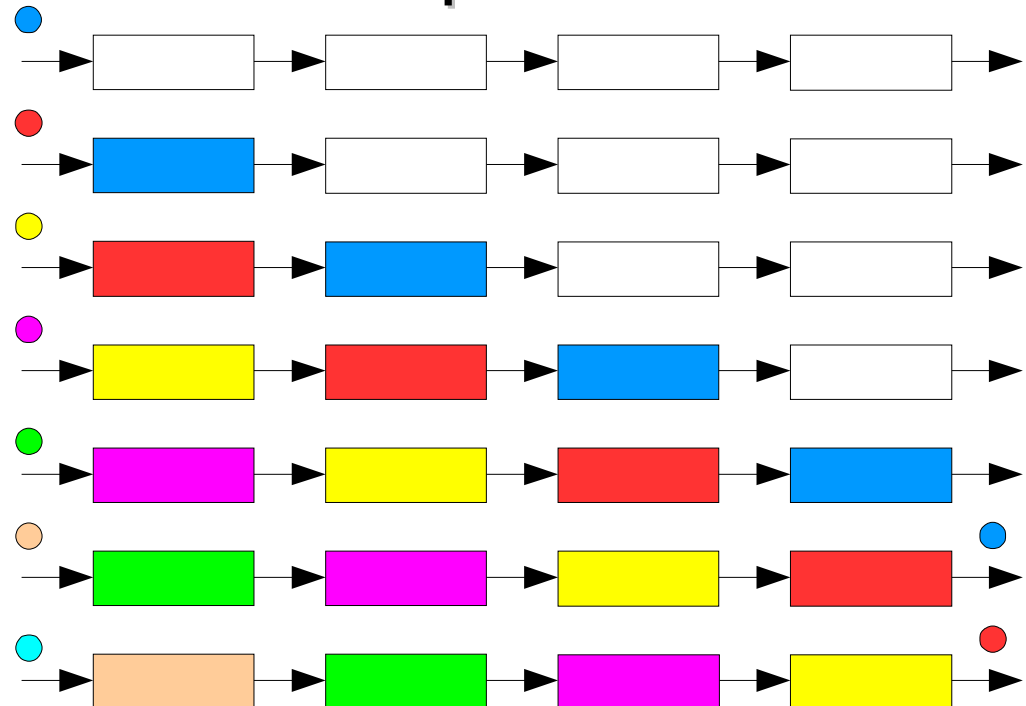
# Pipe-line (3)

---

- Trois conditions sont nécessaires à la bonne mise en œuvre d'un pile-line :
  - Une même opération doit être répétée dans le temps
  - Cette opération doit pouvoir être décomposée en étapes (« *stages* », improprement traduit en « étages ») indépendantes
  - La durée de ces étages doit être à peu près la même

# Pipe-line (4)

- Un pipe-line à  $p$  étages sort son premier résultat après  $p$  cycles élémentaires, puis un résultat par cycle élémentaire
- N'est utile que si l'opération se répète !



# Pipe-line (5)

---

- Pour isoler les différents étages du pipe-line, on utilise des latches
- La fréquence de cadencement est limitée par la durée de l'étage le plus long
  - Il faut y ajouter à cette durée le temps de traversée du latch associé

# Pipe-line (6)

- Soient :
  - $T$  le temps de traversée du circuit non pipe-liné
  - $p$  la profondeur du pipe-line (nombre d'étages)
  - $\lambda$  Le temps de traversée d'un latch
- Si le pipe-line est idéalement équilibré, le circuit pipe-liné exécute  $n$  instructions en  $(p + n - 1)$  étapes de temps unitaire  $(T / p + \lambda)$
- Le circuit non pipe-liné exécute  $n$  instructions en  $n.T$  étapes

# Pipe-line (7)

- L'efficacité du pipe-line est donc égale à :

$$n.T / ((n + p - 1)(T / p + \lambda))$$

- L'efficacité maximale théorique d'un pipe-line équilibré de profondeur  $p$  :
  - Est strictement inférieure à  $p$ 
    - Intérêt d'augmenter  $p$  pour augmenter l'efficacité du pipe-line
    - Revient à augmenter le degré de parallélisme du circuit
  - Tend vers  $p$  quand  $n$  tend vers  $+\infty$ 
    - En supposant  $\lambda$  petit devant  $T / p$

# Pipe-line d'instruction (1)

---

- La tâche la plus répétitive qu'un processeur ait à effectuer est la boucle de traitement des instructions
- Il faut pouvoir décomposer le traitement d'une instruction en sous-étapes de durée à peu près équivalente

# Pipe-line d'instruction (2)

---

- Étapes classiques du traitement des instructions :
  - « *Fetch* » : Récupération de la prochaine instruction à exécuter
  - « *Decode* » : Décodage de l'instruction
  - « *Read* » : Lecture des opérandes (registre ou mémoire)
  - « *Execute* » : Calcul, branchement, etc...
  - « *Write* » : Écriture du résultat (registre ou mémoire)
- Elles-mêmes découpables en sous-étapes



# Pipe-line d'instruction (3)

---

- La création des pipe-lines d'instructions et l'augmentation de leur profondeur a été un facteur déterminant de l'amélioration de la performance des processeurs :
  - 5 étages pour le Pentium
  - 12 étages pour les Pentium II et III
  - 20 étages pour le Pentium IV

# Pipe-line d'instruction (4)

---

- Pourquoi ne pas continuer à augmenter la profondeur des pipe-lines d'instructions ?
  - Problème de taille des niveaux
    - Le surcoût des *latches* augmente en proportion
  - Problème d'équilibrage des niveaux
    - Plus la granularité souhaitée est fine, plus il est difficile de séparer les fonctions logiques en blocs équilibrés
  - Problème de dépendances entre instructions

# Dépendances d'instructions (1)

---

- Les instructions exécutées en séquence sont rarement indépendantes
- On identifie classiquement quatre types de dépendances
  - Certaines sont réelles, et reflètent le schéma d'exécution
  - D'autres sont de fausses dépendances :
    - Accidents dans la génération du code
    - Manque d'informations sur le schéma d'exécution

# Dépendances d'instructions (2)

- Dépendance réelle

```
mov [A],r1
...
add r1,r2,r3
```

- Anti-dépendance

```
add r1,r2,r4
...
mov [A],r1
```

- Dépendance de résultat

```
add r2,r3,r1
...
mov [A],r1
```

- Dépendance de contrôle

```
bz r4,etiq
div r1,r4,r1
etiq: ...
```

# Dépendances d'instructions (3)

- Lorsque le processeur n'est pas pipe-liné, des instructions dépendantes peuvent être exécutées l'une après l'autre sans problème
  - Le résultat de l'instruction précédente est connu au moment où on en a besoin pour la suivante

add r2,r3,r1

F	D	E	M	WB
---	---	---	---	----

add r4,r1,r4

F	D	E	M	WB
---	---	---	---	----

# Dépendances d'instructions (4)

- Lorsque le processeur est pipe-liné, séquencer deux instructions dépendantes peut conduire à des incohérences
  - La valeur d'un registre est lue dans la banque de registres avant que l'instruction précédente l'y ait placée

add r2,r3,**r1**

F	D	E	M	WB
---	---	---	---	----

add r4,**r1**,r4

F	<b>D</b>	E	M	WB
---	----------	---	---	----

- De multiples techniques existent pour amoindrir les effets de ce problème

# Superscalarité

---

- Afin d'augmenter le nombre d'instructions traitées par unité de temps, on fait en sorte que le processeur puisse lire et exécuter plusieurs instructions en même temps
  - Problèmes de dépendances entre instructions
  - Entrelacement de code effectué par le compilateur
  - Réordonnancement dynamique des instructions par le processeur (exécution « *out of order* »)