université de BORDEAUX

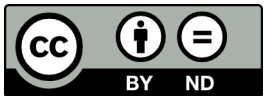
# Architecture des ordinateurs

(INF155)

F. Pellegrini

Université de Bordeaux





# 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

# Évolutions architecturales (1)

- 1946 : Ordinateur ENIAC
  - Architecture à base de lampes et tubes à vide : 30 tonnes, 170 m2 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

http://en.wikipedia.org/wiki/Sandy_Bridge

# 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 continuelle 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-coeurs, 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
  - 1,4 milliards de transistors pour le Core i7 8 cœurs
- 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)

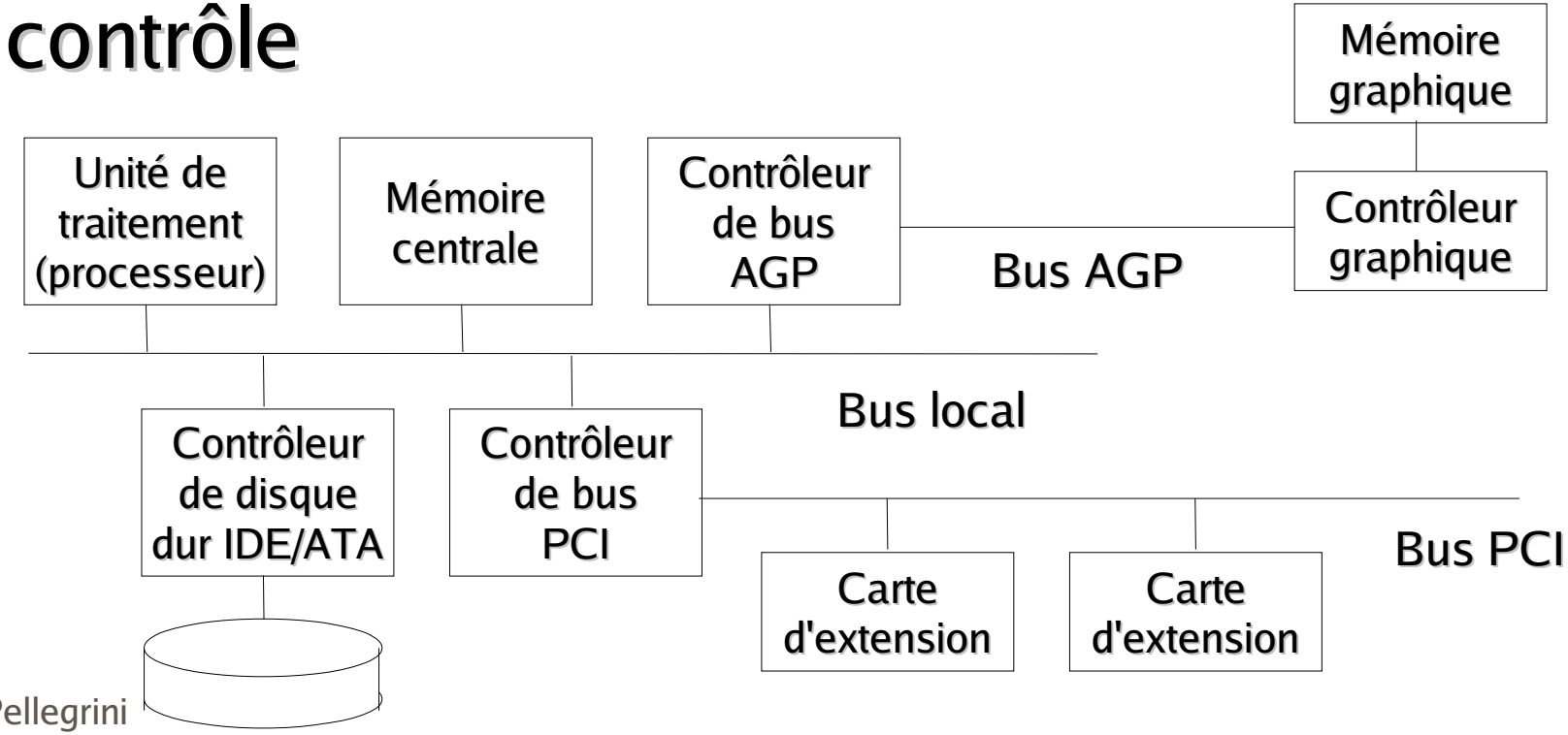
- Un ordinateur est une machine programmable de traitement de l'information
- Pour accomplir sa fonction, il doit pouvoir :
  - Acquérir de l'information de l'extérieur
  - Stocker en son sein ces informations
  - Combiner entre elles les informations à sa disposition
  - Restituer ces informations à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

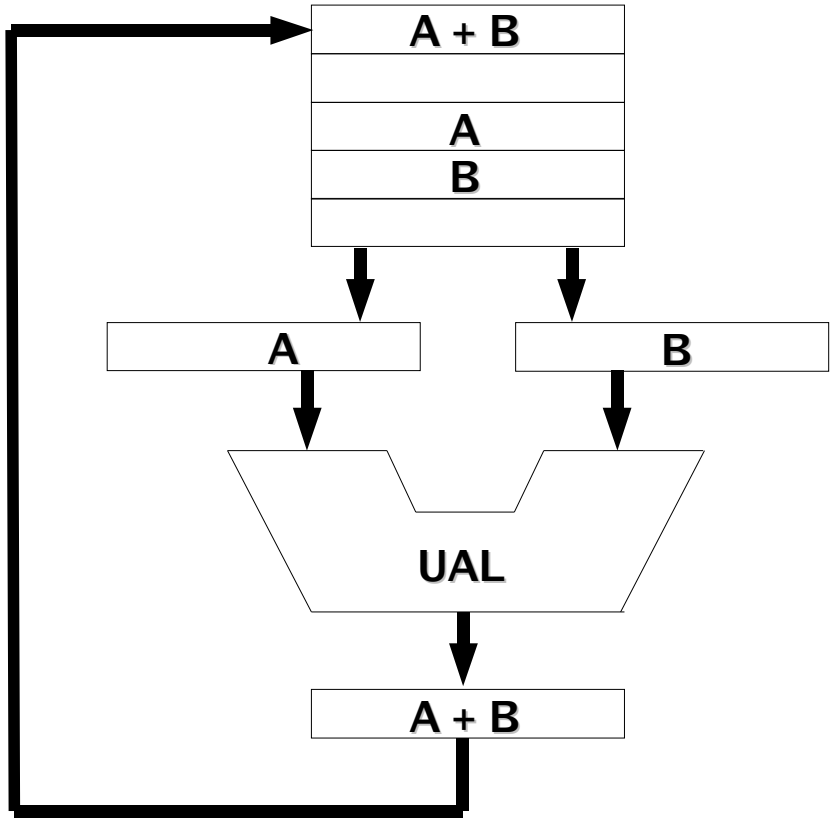
- Chaque registre peut stocker une valeur entière distincte, bornée par la taille des registres (nombre de bits)
- Certains registres sont spécialisés, comme :
  - le compteur ordinal (« *program counter* ») qui stocke l'adresse de la prochaine instruction à exécuter
  - le registre d'instruction («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




Registres généraux

Registres d'entrée de l'UAL

Registre de sortie de l'UAL

# 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

# Langages et exécution

- Les processeurs ne comprennent que le langage machine
  - Trop élémentaire, trop long à programmer
- Nécessité de construire des programmes à un niveau d'abstraction plus élevé
  - Meilleure expressivité et généricité
  - Moins de risques d'erreurs
- Besoin de passer d'un langage humainement compréhensible à une exécution machine

# Langages et machines virtuelles

- Exécuter un programme écrit dans un langage $L^1$ au moyen d'une machine comprenant le langage $L^0$ peut se faire de deux façons différentes
  - Par traduction
  - Par interprétation

# Traduction

- Consiste à remplacer chaque instruction de $L^1$ par une séquence équivalente d'instructions de $L^0$
- Le résultat de la traduction est un programme équivalent écrit en $L^0$
- L'ordinateur peut alors exécuter ce programme en $L^0$ à la place du programme écrit en $L^1$

# Interprétation

- Consiste à utiliser un programme écrit en $L^0$ qui prend comme entrée un programme écrit en $L^1$, parcourt ses instructions l'une après l'autre, et exécute directement pour chacune d'elles une séquence équivalente d'instructions en $L^0$
- Après que chaque instruction en $L^0$ a été examinée et décodée, on la met en oeuvre
- On ne génère pas explicitement de programme écrit en $L^0$

# Machines virtuelles

- On peut repenser le problème en postulant l'existence d'une machine $M^1$ exécutant directement le code $L^1$
- Si ces machines pouvaient être facilement réalisées, on n'aurait pas besoin de machines $M^0$ utilisant nativement le $L^0$
- Sinon, ces machines sont dites virtuelles
- Par nature, les langages $L^1$ et $L^0$ ne doivent pas être trop différents

# Machines multi-couches (1)

- En itérant ce raisonnement, on peut concevoir un ensemble de langages, de plus en plus complexes et expressifs, jusqu'à arriver à un niveau d'abstraction satisfaisant pour le programmeur
- Chaque langage s'appuyant sur le précédent, on peut représenter un ordinateur construit selon ce principe comme une machine multi-couches

# Machines multi-couches (2)

- Les premières machines ne disposaient que de la couche matérielle $M^0$
  - Réalisation de calculs élémentaires
  - Exécution directe des instructions du langage machine
- La mémoire étant rare et chère, il fallait avoir des instructions plus puissantes
  - Difficiles à câbler dans le matériel
  - Introduction d'une couche micro-codée réalisant l'interprétation des instructions complexes

# Machines multi-couches (3)

- Avec la nécessité de partager l'utilisation des machines, apparition des systèmes d'exploitation
  - Nouveau modèle de gestion de la mémoire et des ressources
  - Abstraction supplémentaire
- Après l'inflation des jeux d'instruction micro-programmés, retour à des jeux d'instructions plus restreints (RISC)
  - Disparition de la micro-programmation
  - Report de la charge de travail sur les compilateurs

# Machines multi-couches (4)

- La frontière entre ce qui est implémenté par matériel et ce qui est implémenté par logiciel est mouvante
  - Toute opération réalisée par logiciel peut être implémentée matériellement
  - Toute fonction matérielle peut être émulée par logiciel
- Le choix d'implémentation dépend de paramètres tels que le coût, la vitesse, la robustesse, et la fréquence supposée des modifications

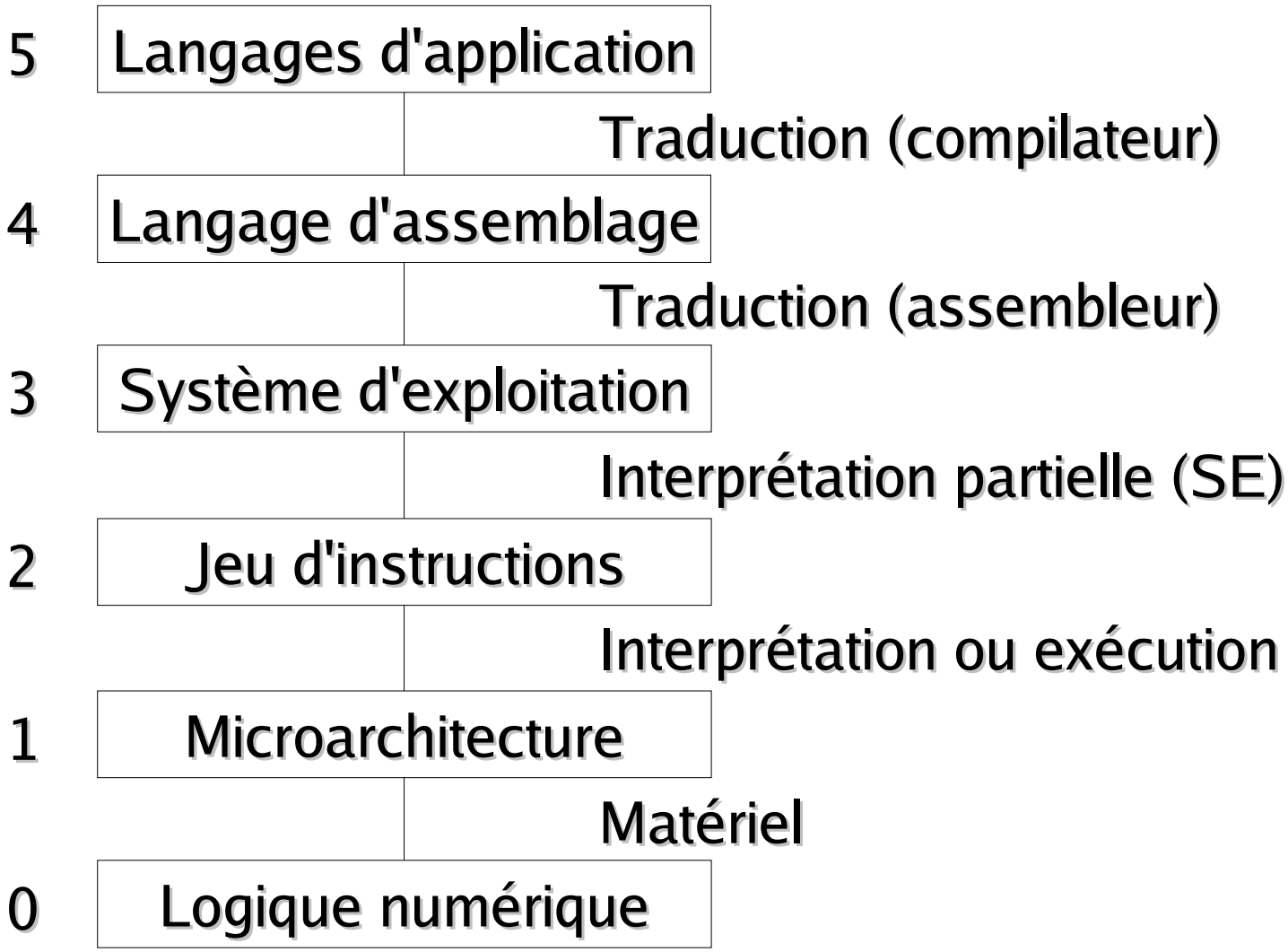
# Machines multi-couches (5)

- À un même jeu d'instructions peuvent correspondre plusieurs implémentations de coûts différents, utilisant des techniques différentes
  - Familles de processeurs
  - Utilisation d'un coprocesseur arithmétique câblé ou émulation des instructions par logiciel ?

# 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

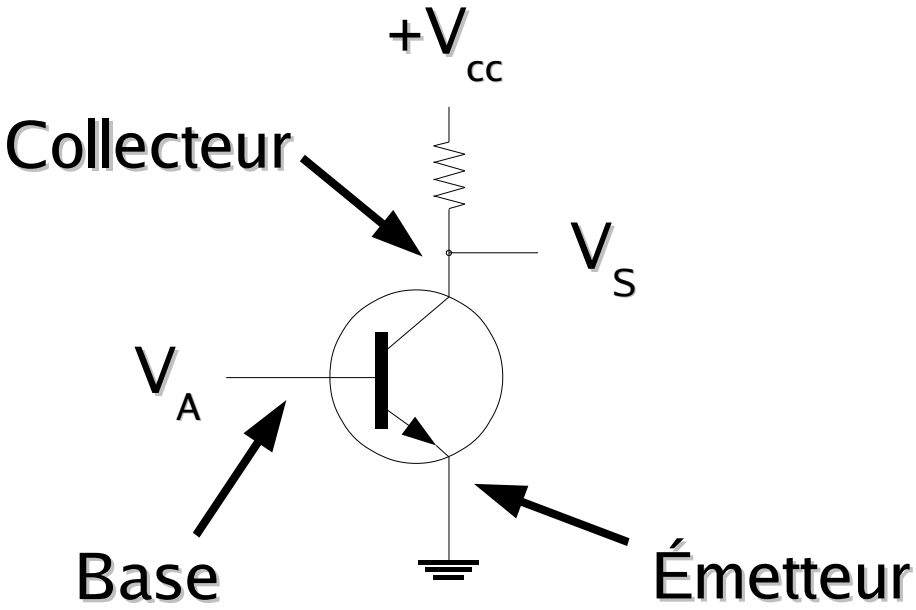
- Un circuit logique est un circuit qui ne manipule que deux valeurs logiques : 0 et 1
- À l'intérieur des circuits, on représente typiquement un état 0 par un signal de basse tension (proche de 0V) et un état 1 par un signal de haute tension (5V, 3,3V, 2,5V, 1,8V ou 0,9V selon les technologies)
- De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

# Transistors (1)

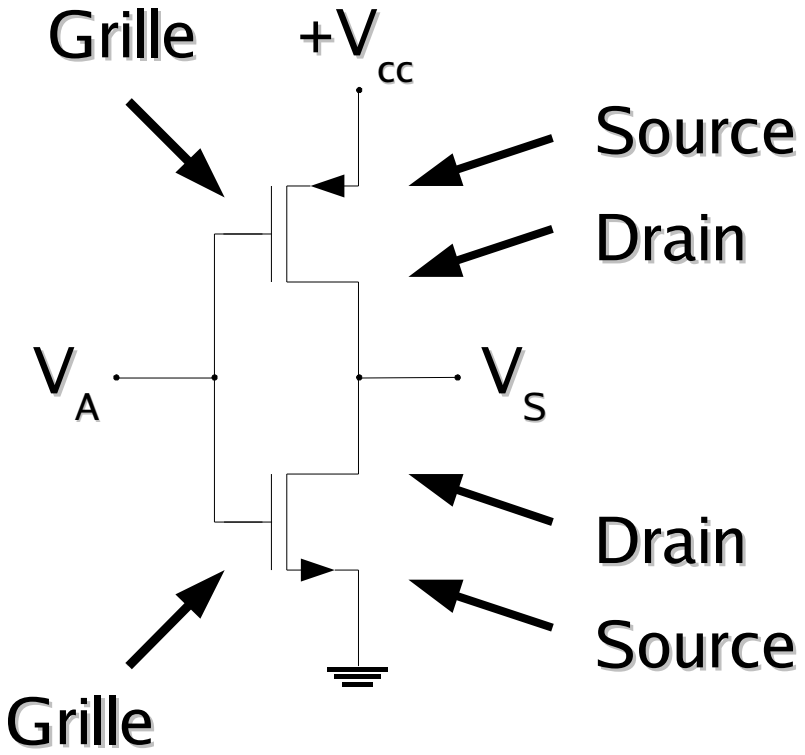
- L'électronique numérique repose sur le fait qu'un transistor peut servir de commutateur logique extrêmement rapide
- Deux technologies majeures :
  - Bipolaire : temps de commutation très rapide mais consommation élevée
    - Registres, SRAM, circuits spécialisés
  - CMOS : temps de commutation moins 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 :



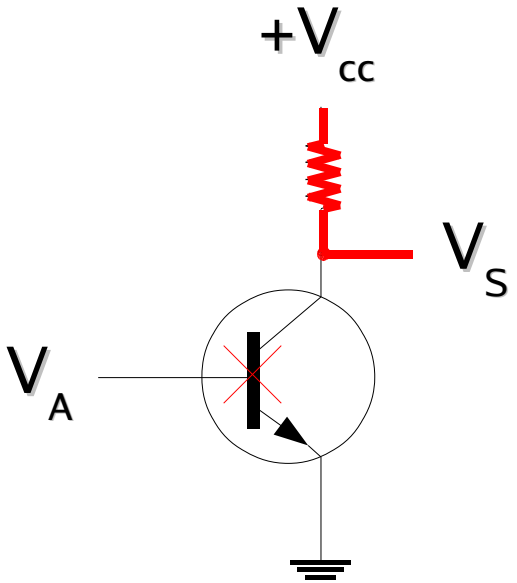

Bipolaire

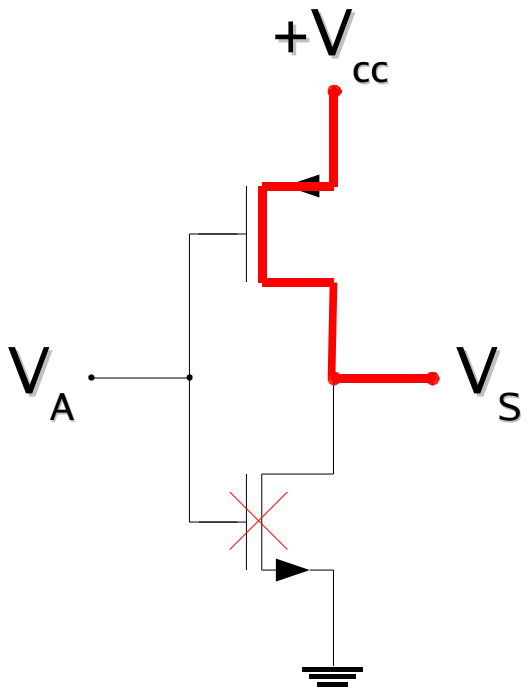



CMOS

# Transistors (3)

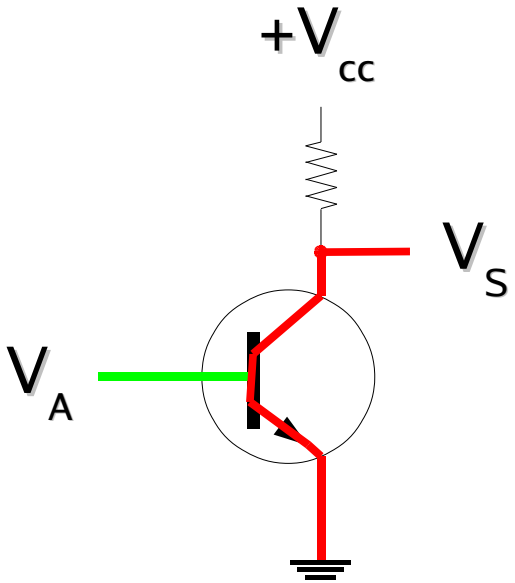
- Quand $V_A$ est bas, $V_S$ est haut

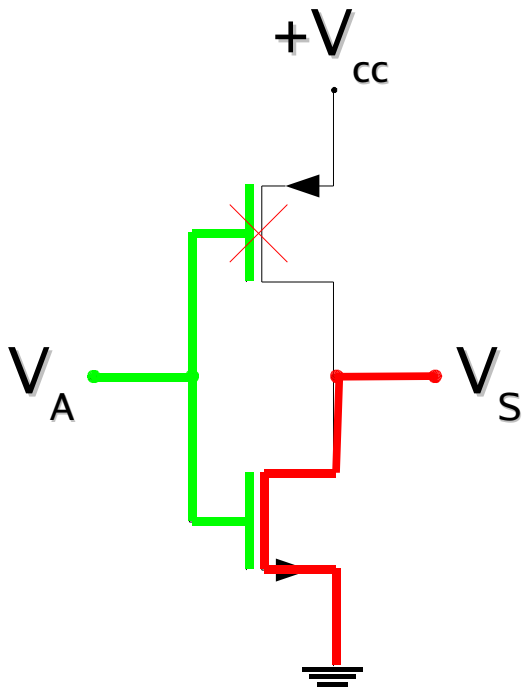


# Transistors (4)

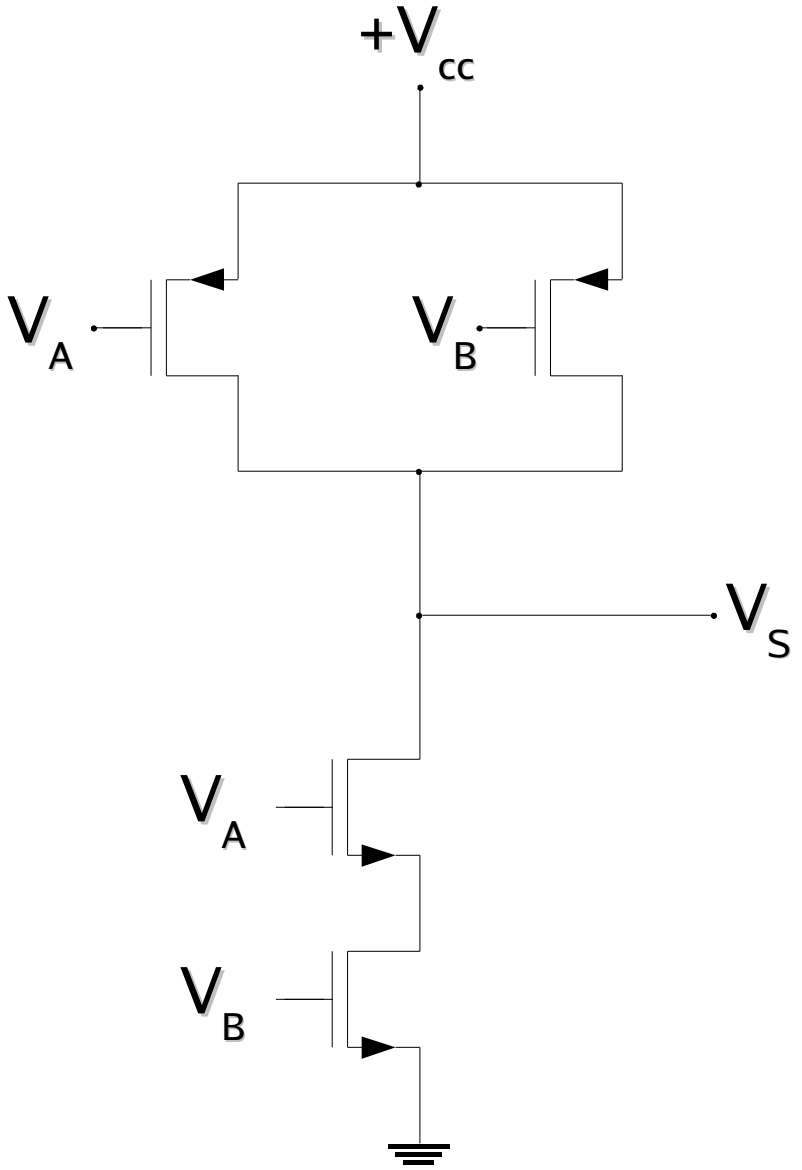
- Quand $V_A$ est bas, $V_S$ est haut
- Quand $V_A$ est haut, $V_S$ est bas
- Ce circuit est un inverseur



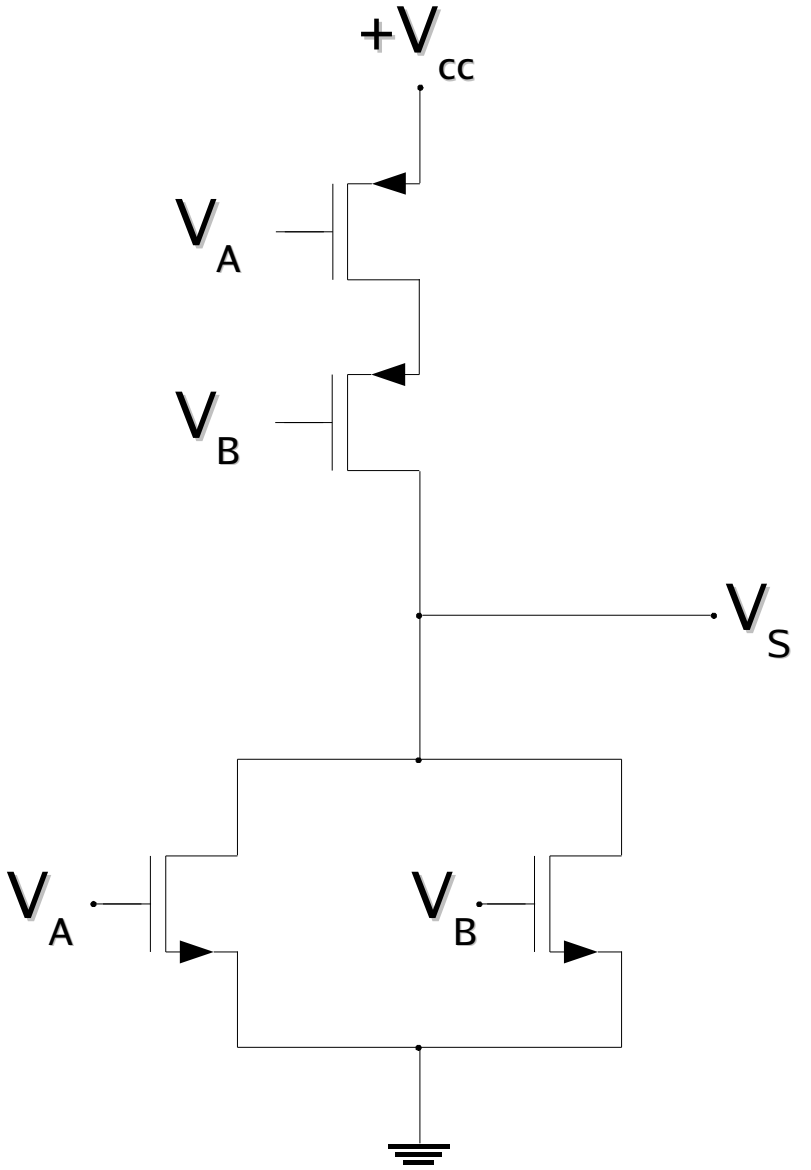
# Transistors (5)

- 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



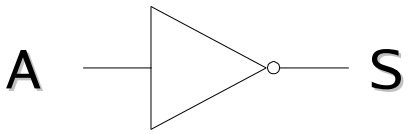
# Transistors (6)

- 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



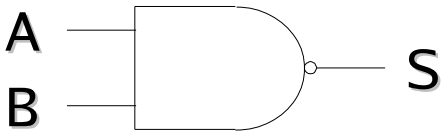
# Portes logiques (1)

- 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



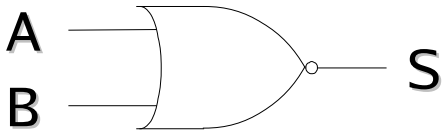

| A | S |
|---|---|
| 0 | 1 |
| 1 | 0 |

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)

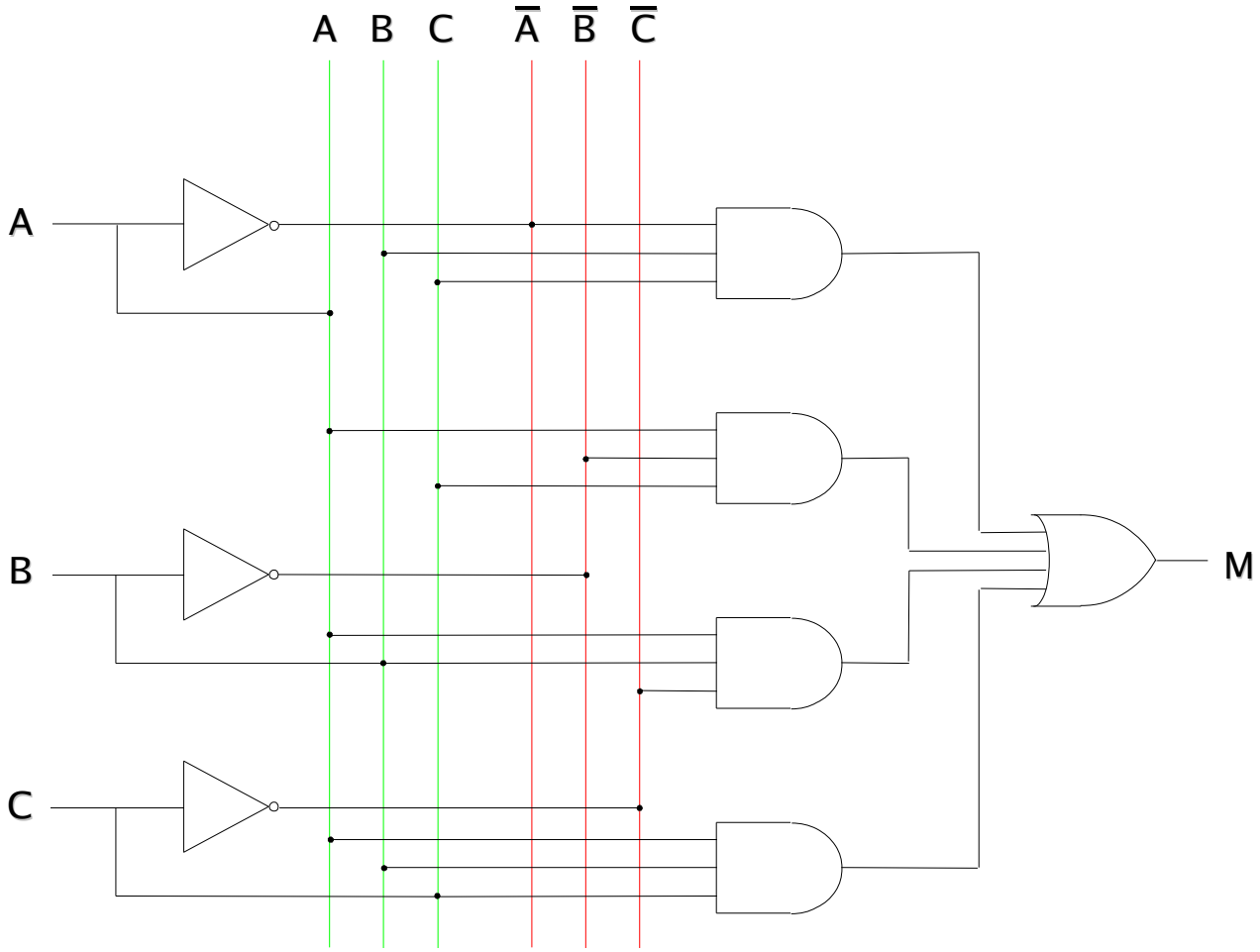
- En notant :
  - $\overline{A}$ le NOT de A
  - A + B le OR de A et B
  - A.B ou AB le AND de A et B

  on peut représenter une fonction comme somme logique de produits logiques
- Par exemple :
  - $A\overline{B}C$ vaut 1 seulement si A = 1 et B = 0 et C = 1
  - $A\overline{B}$ + $B\overline{C}$ 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 : $M = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$
- 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)

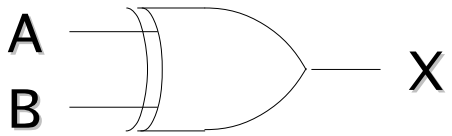
| Nom | Forme AND | Forme OR |
|---|---|---|
| Identité | 1A = A | 0 + A = A |
| Nul | 0A = 0 | 1 + A = 1 |
| Idempotence | AA = A | A + A = A |
| Inverse | AA = 0 | A + A = 1 |
| Commutativité | AB = BA | A + B = B + A |
| Associativité | (AB)C = A(BC) | (A + B) + C = A + (B + C) |
| Distributivité | 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
  - $\overline{ABC} = \overline{A} + \overline{B} + \overline{C}$
  - 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)

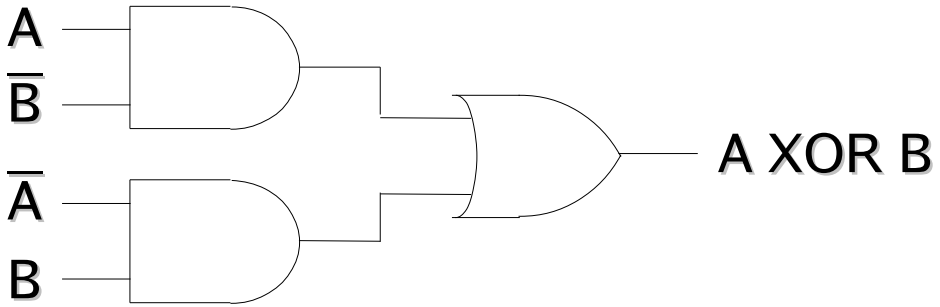
- Grâce aux identités, il est facile de convertir la représentation en somme de produits en une forme purement NAND ou NOR
- Exemple : la fonction « ou exclusif » ou XOR
  - XOR = $A\overline{B} + \overline{A}B$

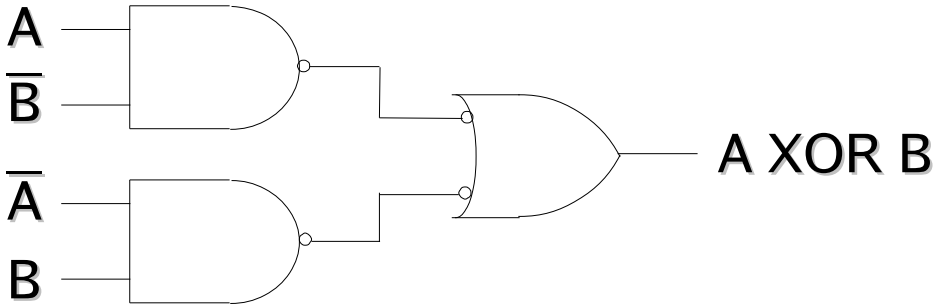


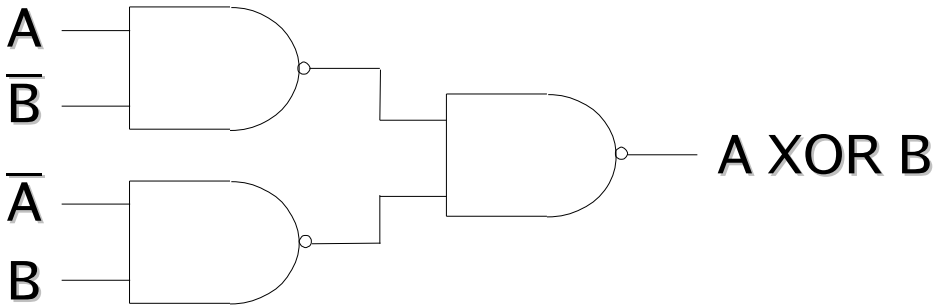

| A | B | X |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

XOR

# Porte XOR (2)



# Tables de Karnaugh (1)

- Les tables de Karnaugh permettent de simplifier une fonction logique en supprimant les variables redondantes
- Pour cela :
  - On écrit la table de vérité sous forme de tableau donc les entrées regroupent une ou deux variables, ordonnées selon un code de Gray
    - Un seul bit de différence entre deux mots consécutifs
- On cherche des groupes de « 1 » ou de « 0 » contigus, y compris à travers les bords

# Tables de Karnaugh (2)

- Exemple : fonction à 4 variables A, B, C, D
- On groupe les « 1 » (les plus nombreux) par paquets de 2x1, 4x1, 2x2, 4x2 (voire 4x4 !)
  - Chaque fois qu'on double la taille, on élimine une variable

CD

| F | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 1 | 1 | 0 |
| 01 | 0 | 0 | 1 | 0 |
| 11 | 1 | 1 | 1 | 1 |
| 10 | 1 | 1 | 1 | 1 |

AB

$F = \overline{B}D + CD + A$

$F = (\overline{B} + C)D + A$

# 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, ...

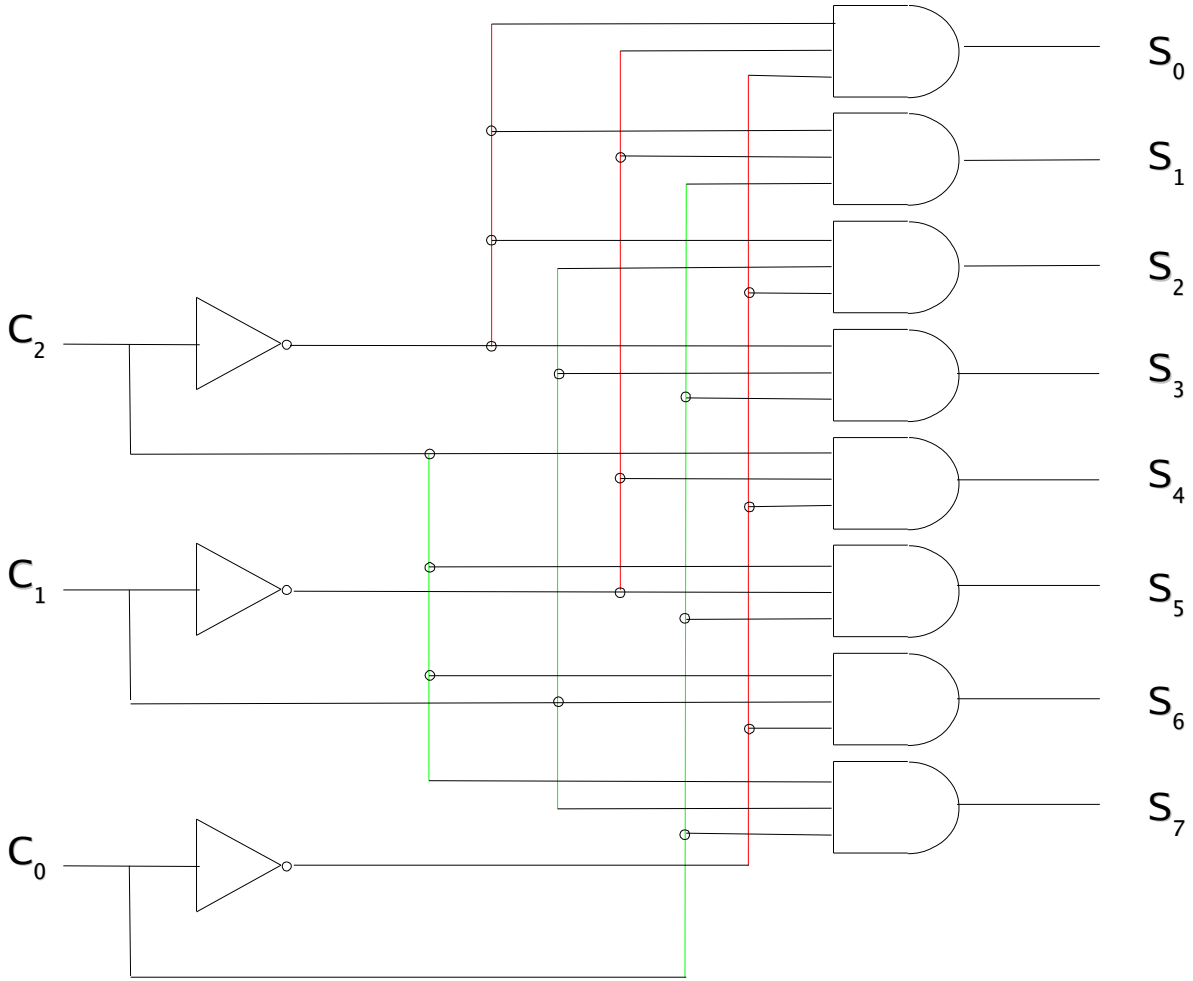
# Fonctions combinatoires

- 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
  - Comparateurs
  - ...

# 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 $S_i$

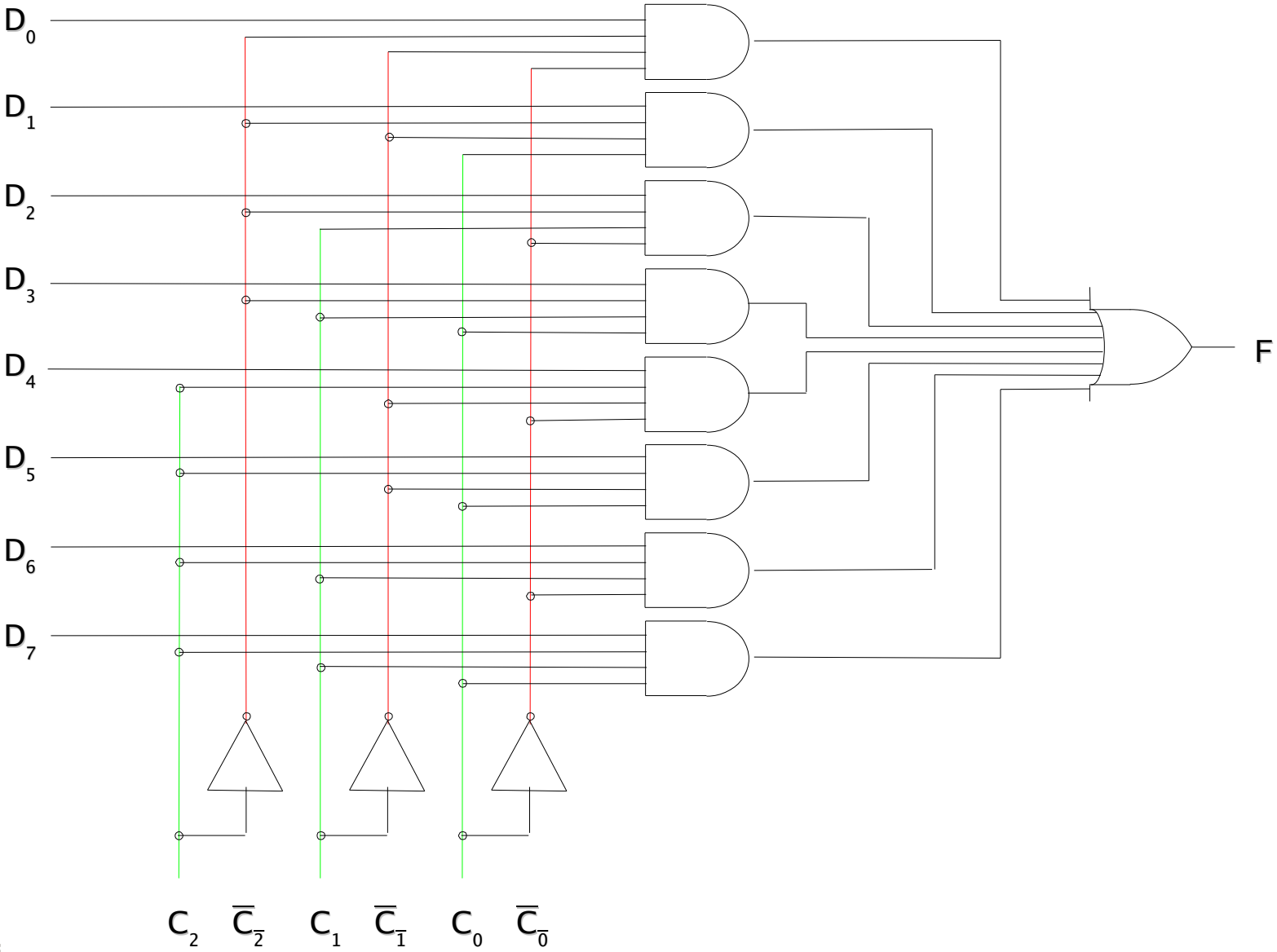
# Décodeur (2)



# Multiplexeur (1)

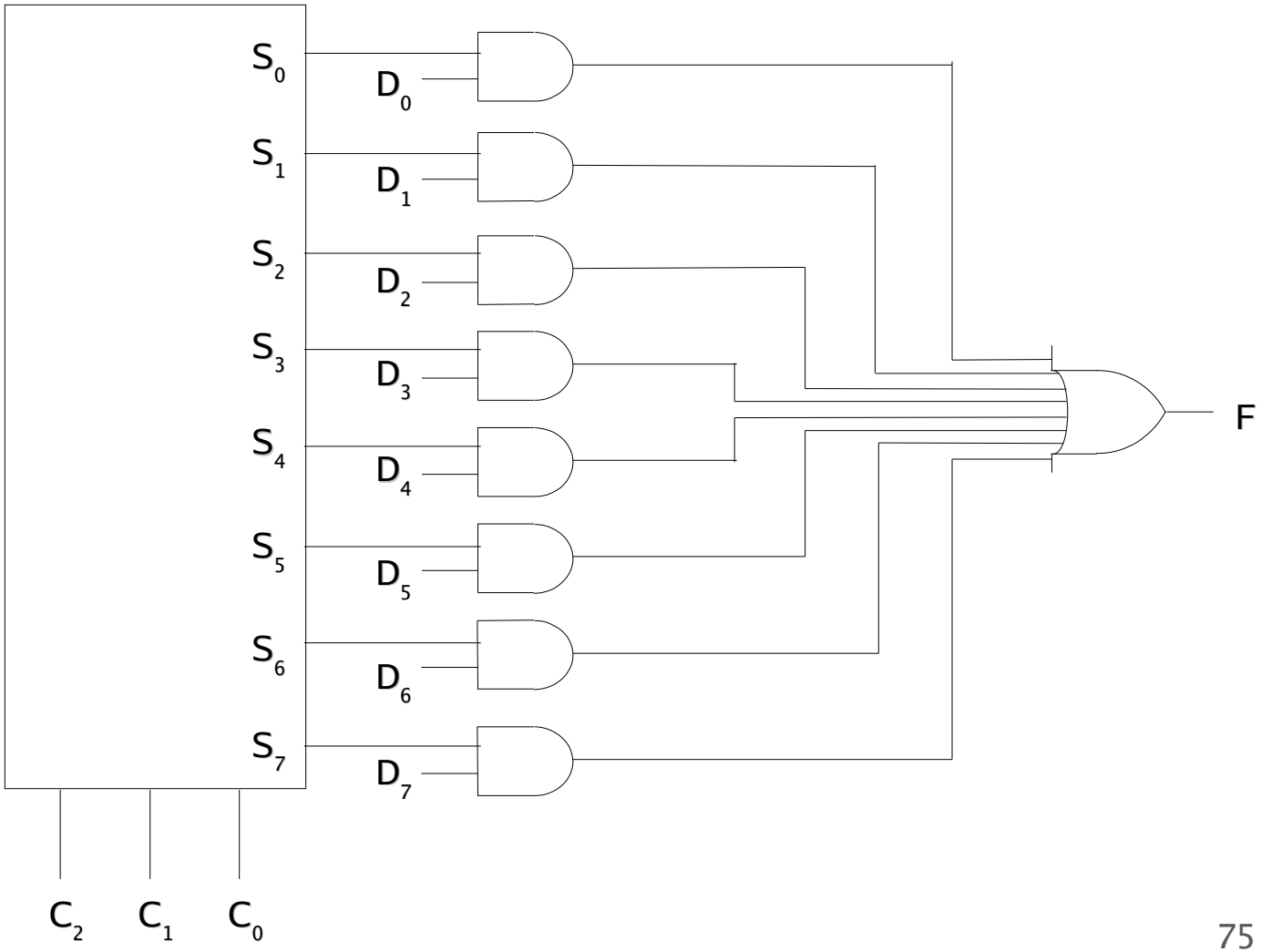
- Un multiplexeur est une fonction possédant $2^n$ entrées de données $D_i$, une unique sortie et n entrées de contrôle $C_j$ servant à sélectionner l'une des entrées
- La valeur de l'entrée sélectionnée est répercutée (routée) sur la sortie
- Les n entrées de contrôle codent un nombre binaire à n bits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

# Multiplexeur (2)



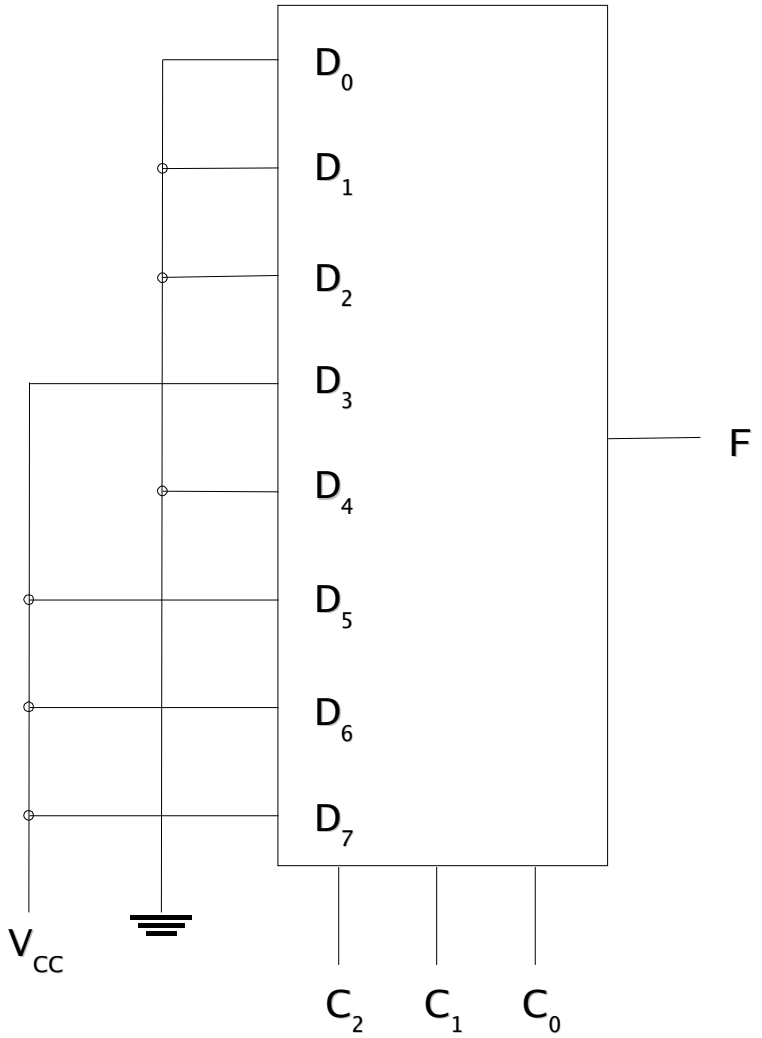
# Multiplexeur (3)

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



# Multiplexeur (4)

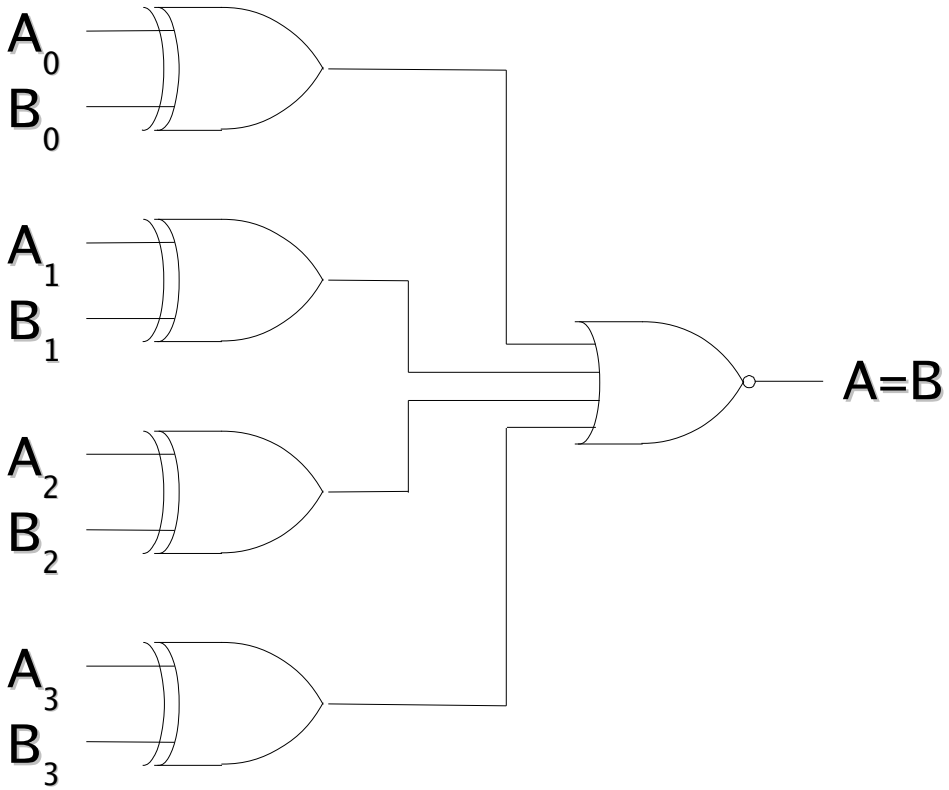
- Utilisation d'un multiplexeur pour implanter la fonction majorité



# 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)



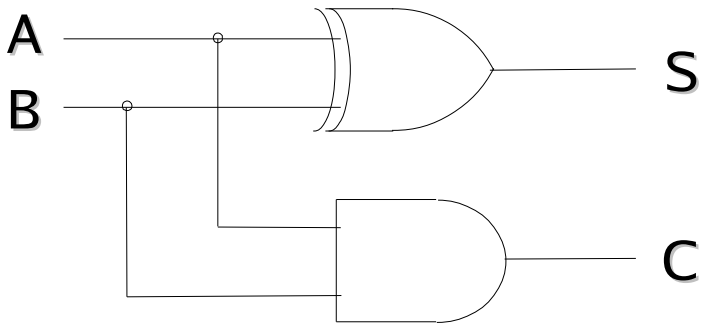
# Fonctions arithmétiques

- 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 »

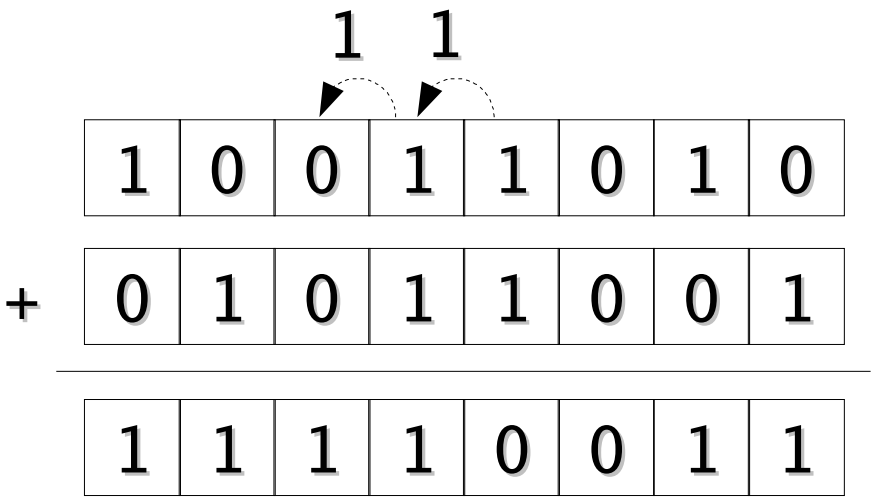
| A | B | C | S |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |




S : Somme
C : Retenue (« carry »)

# Additionneur (2)

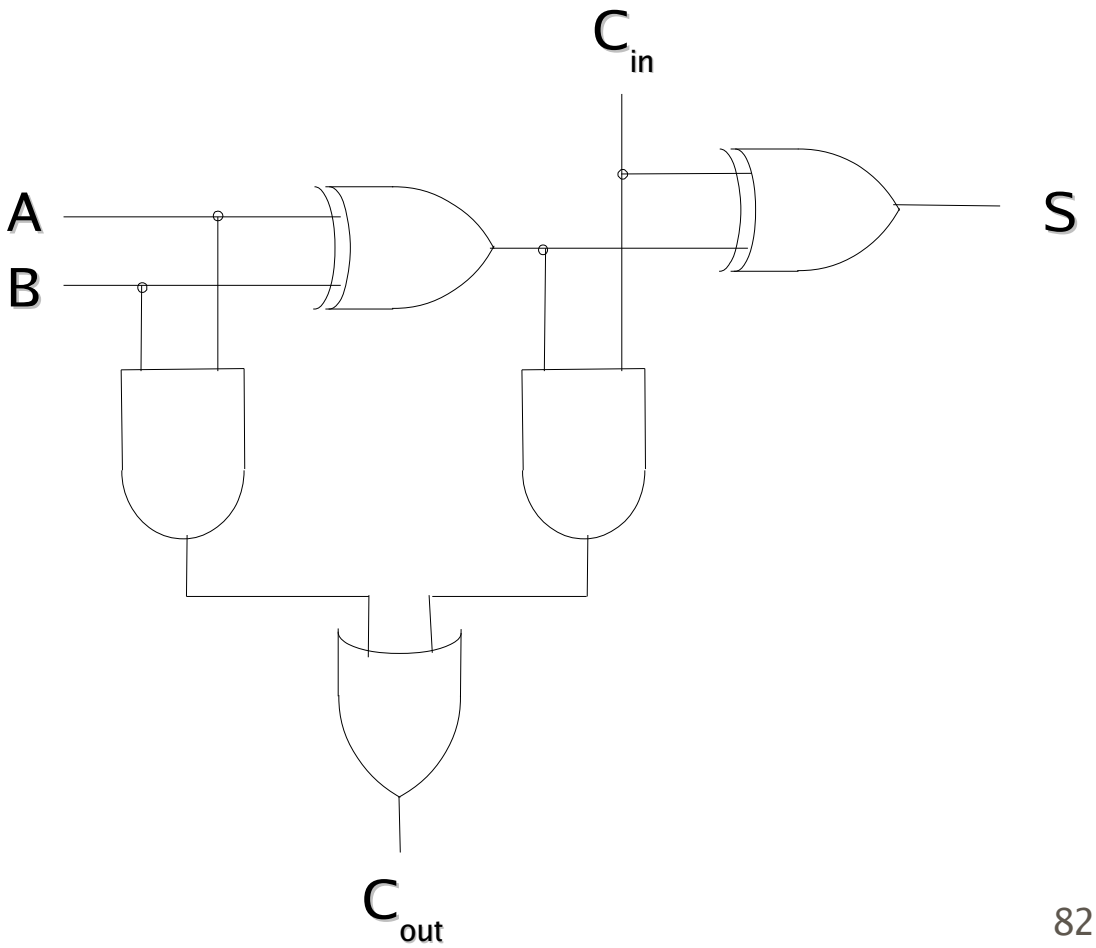
- En fait, pour additionner deux bits situés au milieu d'un mot, il faut aussi prendre en compte la retenue provenant de l'addition du bit précédent et propager sa retenue au bit suivant



# Additionneur (3)

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

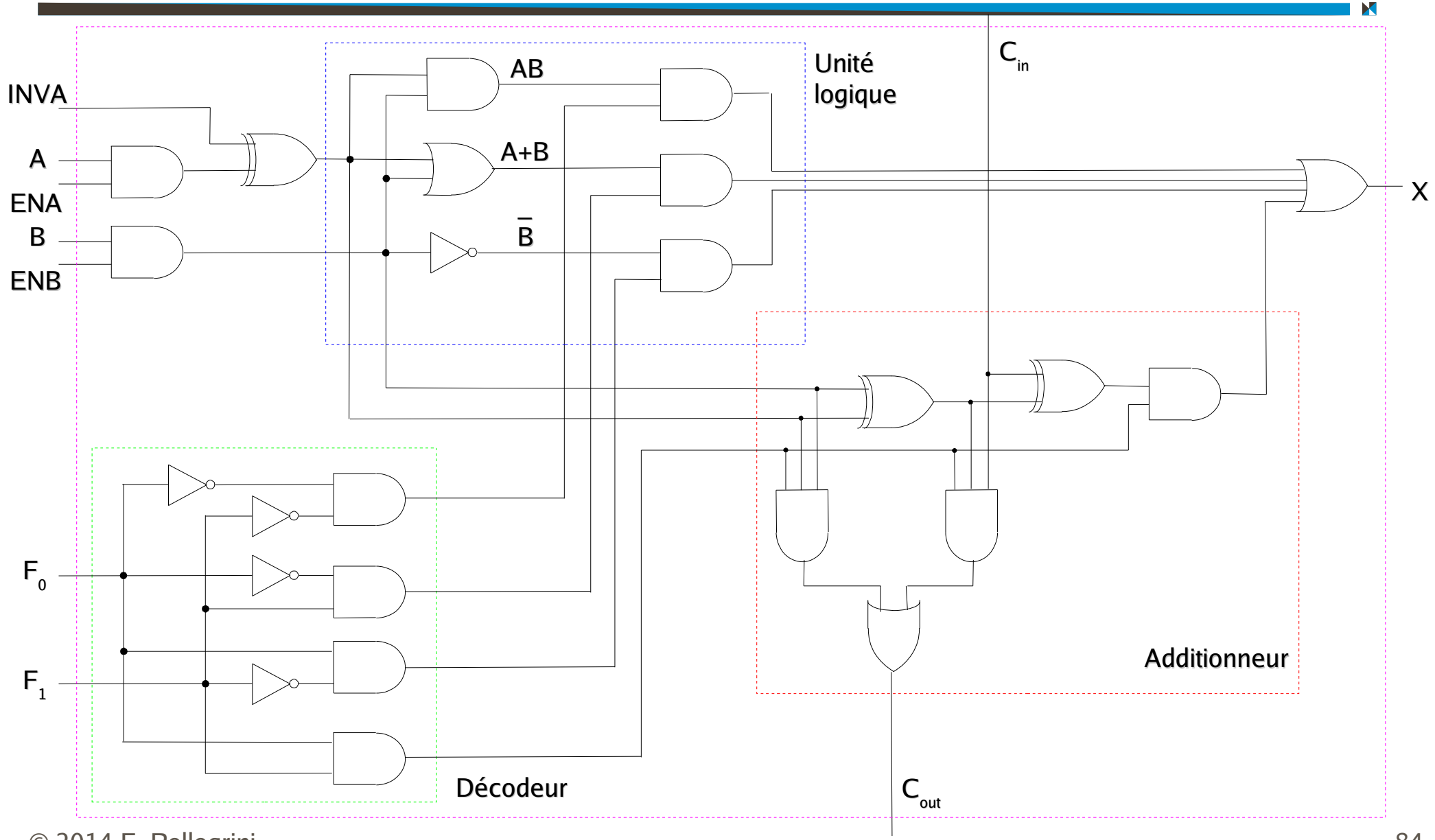
| A | B | Cin | Cout | S |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |



# Unité arithmétique et logique (1)

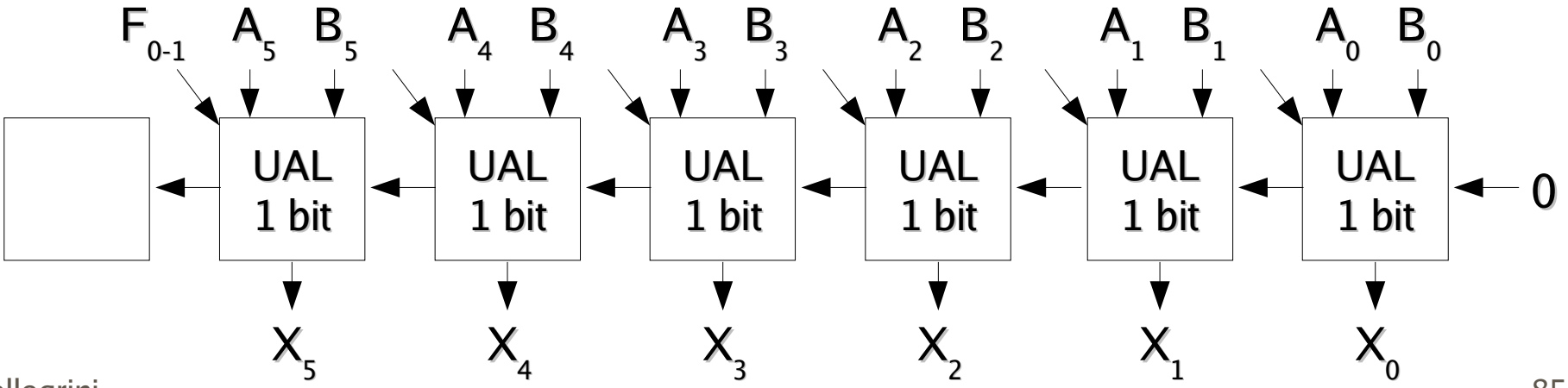
- La plupart des ordinateurs combinent au sein d'un même circuit les fonctions arithmétiques et logiques permettant de calculer l'addition, la soustraction, le AND ou le OU de deux mots machines : c'est l'Unité Arithmétique et Logique
- Le type de la fonction a calculer est déterminé par des entrées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

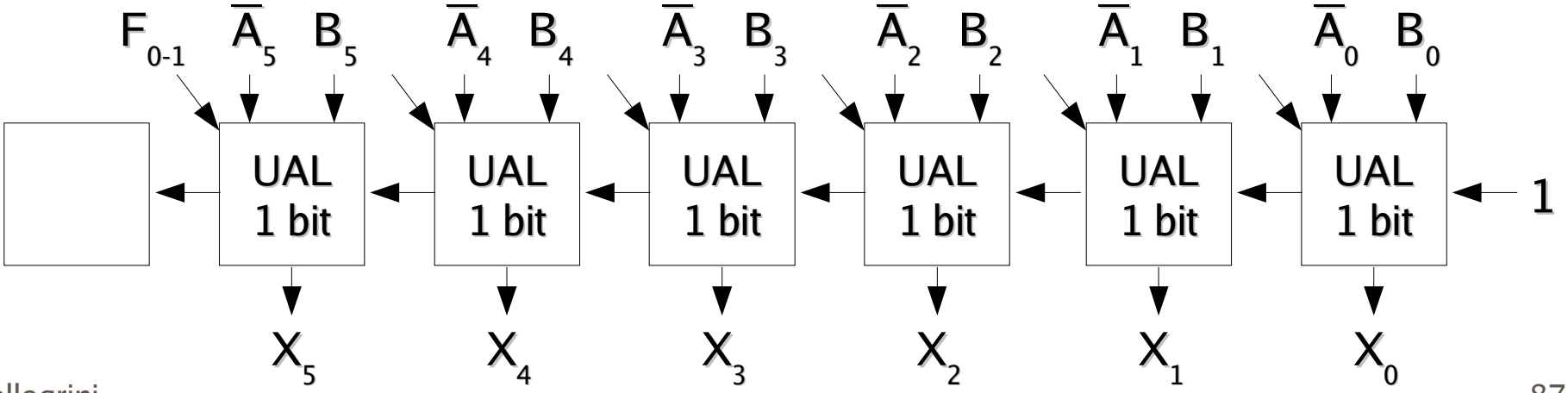


# Additionneur - Soustracteur (1)

- 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

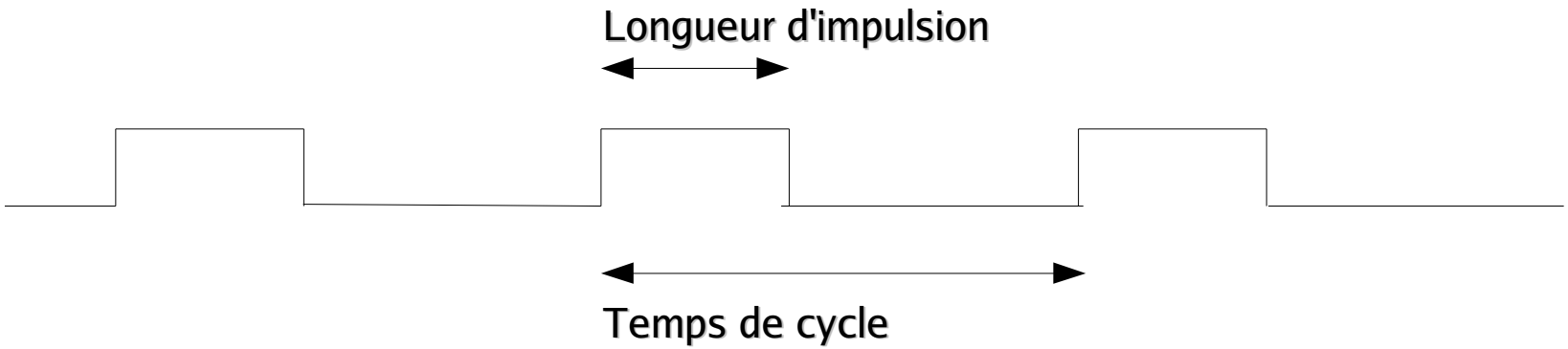


# Horloge (1)

- 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

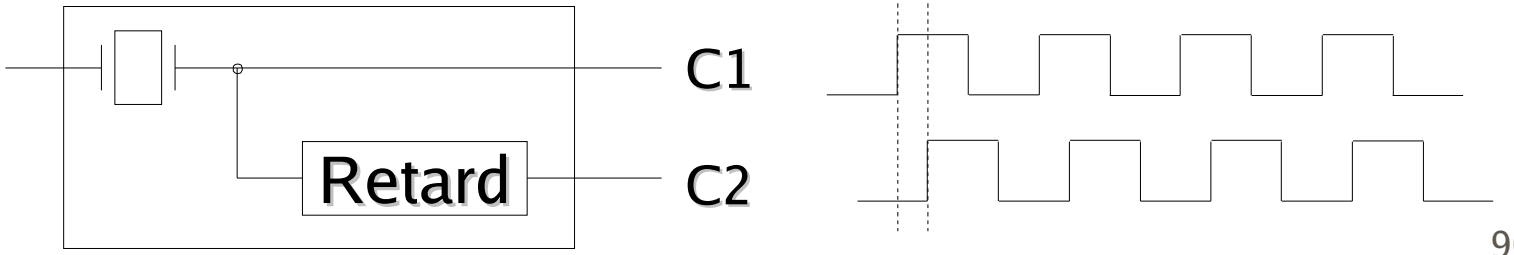
# Horloge (2)

- 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



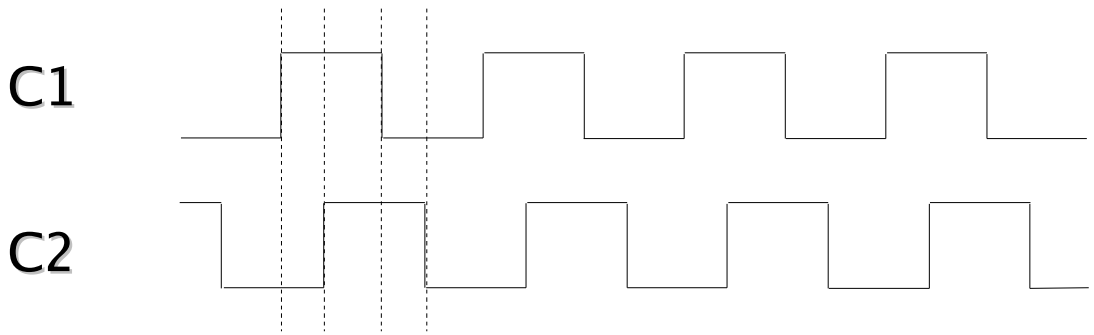
# Cycles et sous-cycles (1)

- 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



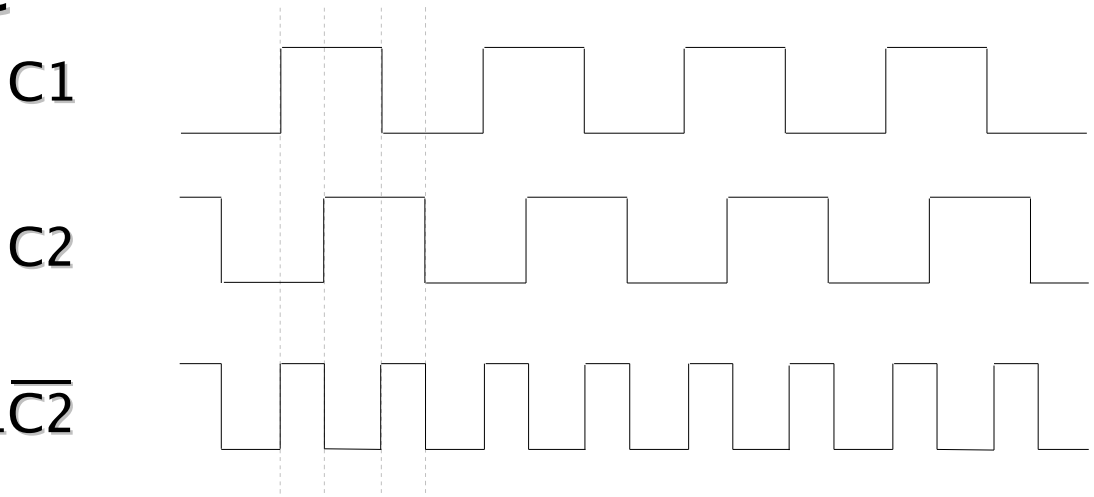
# Cycles et sous-cycles (2)

- 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)

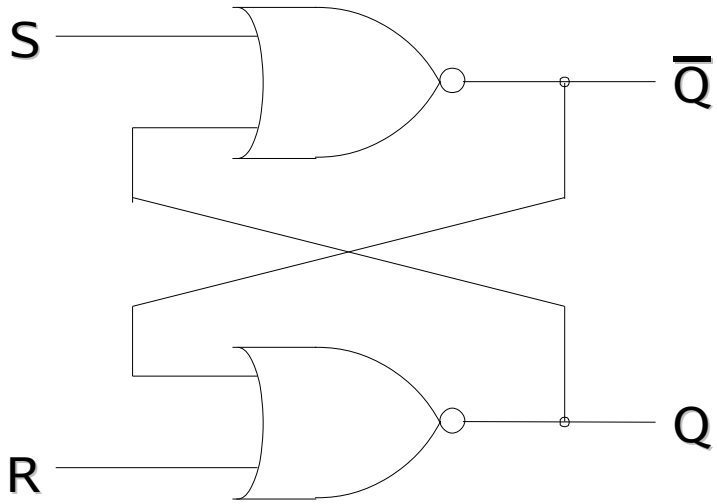
- La mémoire principale sert au stockage des programmes et de leurs données
- L'unité élémentaire de mémoire est le bit, pour « *binary digit* » (« chiffre binaire »), prenant deux valeurs, 0 ou 1
- Le stockage physique des bits dépend des technologies employées : différentiels de tension, moments magnétiques,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

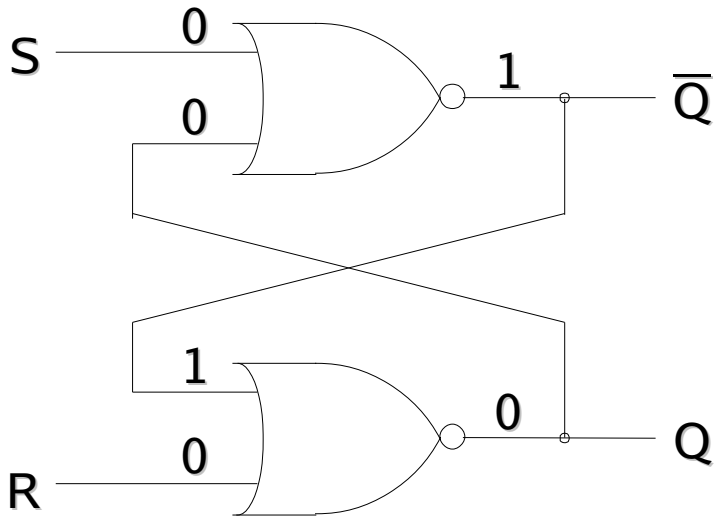
# Bascule SR

- 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 $\overline{Q}$ complémentaires l'une de l'autre



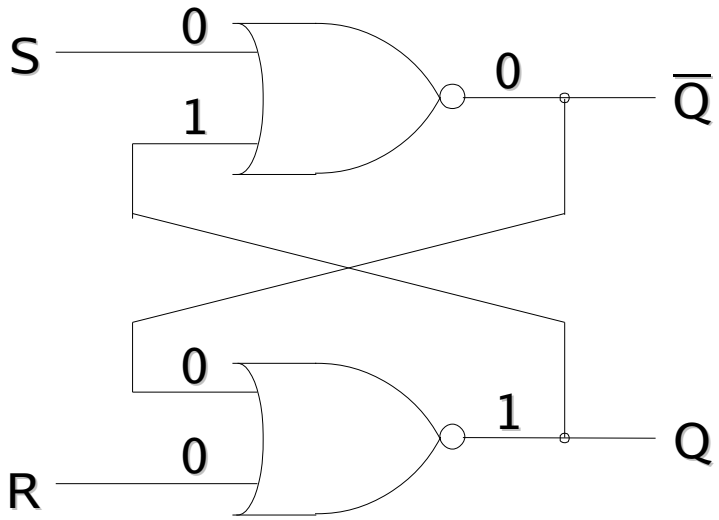
# Bascule SR – État 0

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



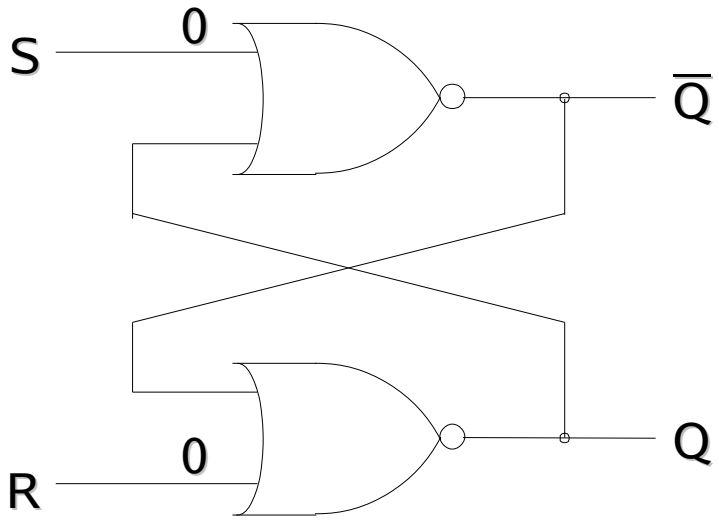
# Bascule SR – État 1

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



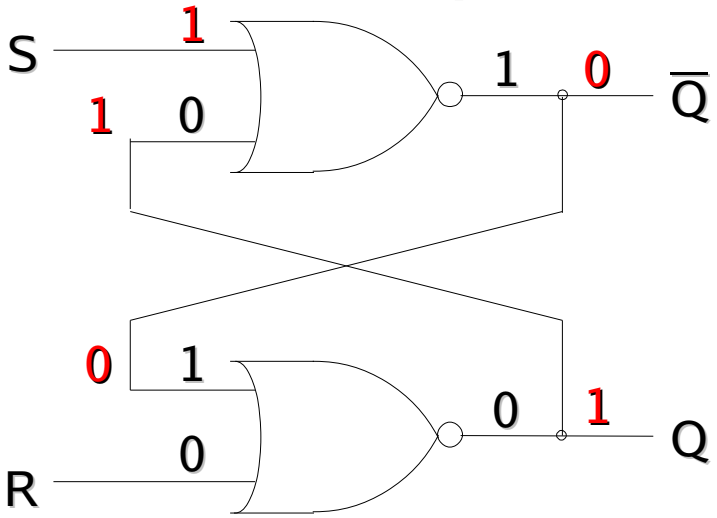
# Bascule SR – États stables

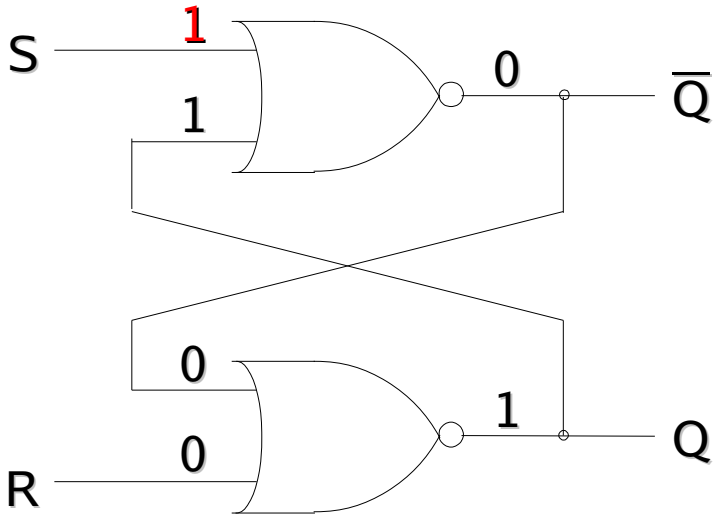
- Lorsque S et R valent 0 :
  - On ne peut avoir simultanément Q à 0 et $\overline{Q}$ à 0
  - On ne peut avoir simultanément Q à 1 et $\overline{Q}$ à 1
  - La bascule possède deux états stables, tels que Q = 0 ou Q = 1



# Bascule SR – Mise à 1

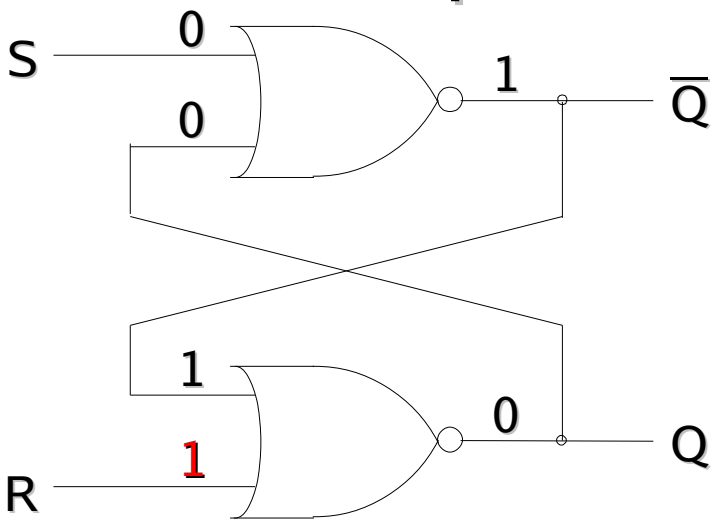
- Lorsque S vaut 1, que Q vaille 0 ou 1 :
  - La sortie de la porte du haut vaut 0, donc $\overline{Q}$ vaut 0
  - La sortie de la porte du bas vaut 1, donc Q vaut 1
  - Cet état est stable
- Même lorsque S repasse à 0, Q reste à 1

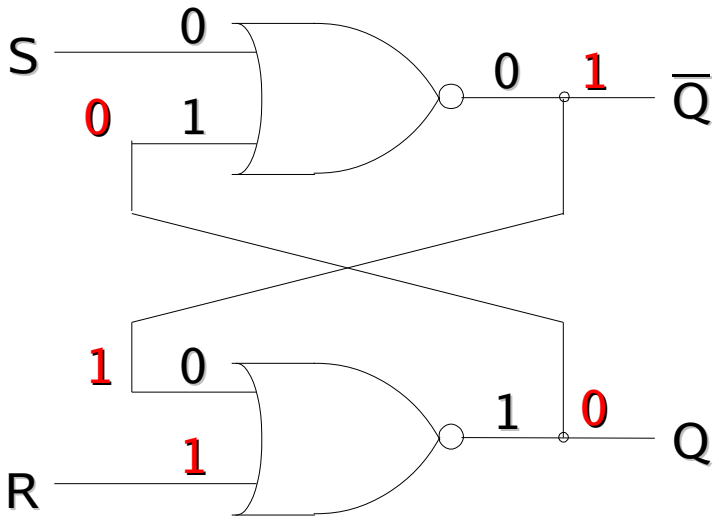


# Bascule SR – Mise à 0

- 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 $\overline{Q}$ vaut 1
  - Cet état est stable
- Même lorsque R repasse à 0, Q reste à 0



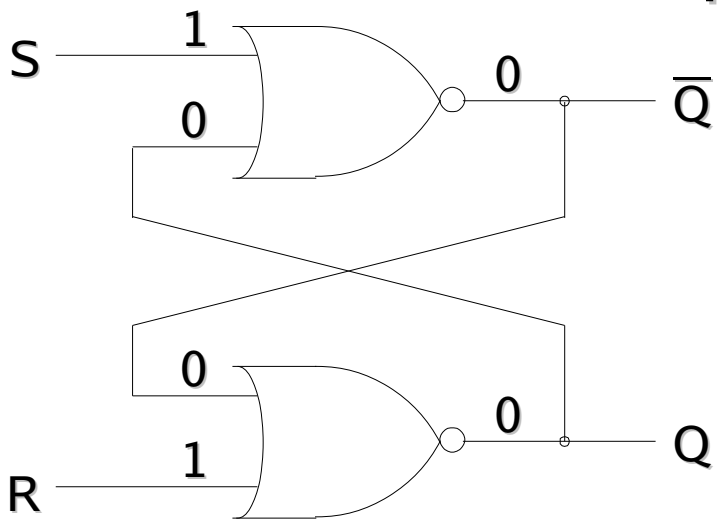



# Bascule SR – Résumé (1)

- 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

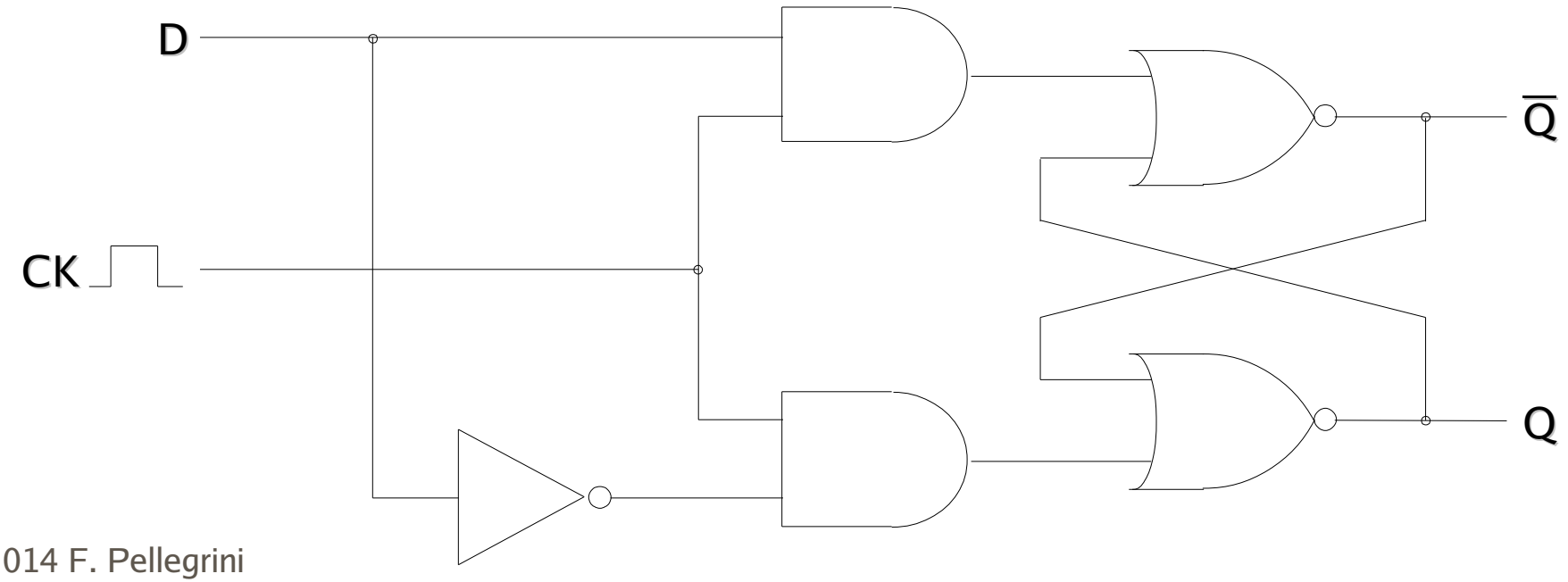
# Bascule SR – Résumé (2)

- 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 $\overline{Q}$ valent 0
  - Lorsque S et R repassent simultanément à 0, l'état final de la bascule est non prévisible



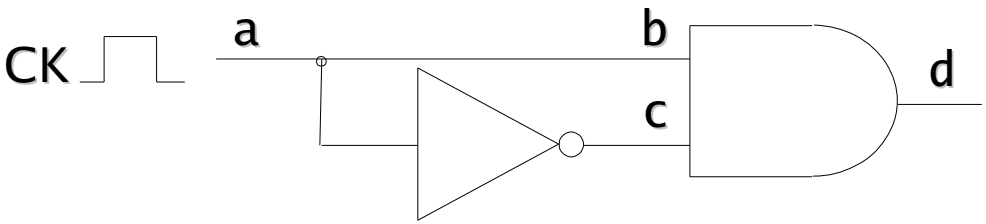
# Bascule D (1)

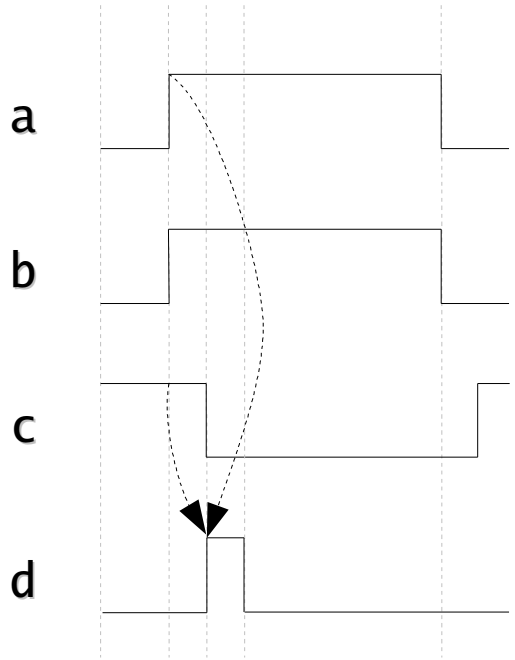
- Pour éviter cela, on n'a qu'un seul signal D
  - Destiné à l'entrée S, et que l'on inverse pour R
  - On commande la bascule par un signal d'activation
- On a une mémoire à 1 bit



# Bascule D (2)

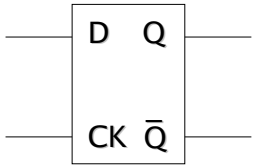
- Pour que l'on soit sûr que la valeur conservée en mémoire soit bien celle présente en début de cycle d'écriture, il faudrait n'autoriser l'écriture qu'au début du cycle, sur le front montant du signal d'écriture



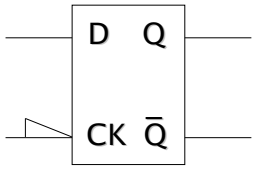



# Bascule D (3)

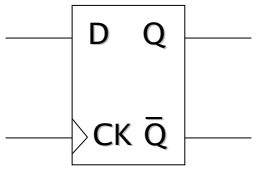
- Il existe ainsi plusieurs types de bascules D :
  - Activée par CK à l'état haut



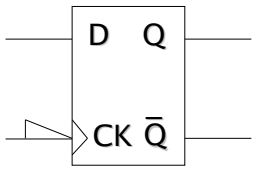

  - Activée par CK à l'état bas




  - 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

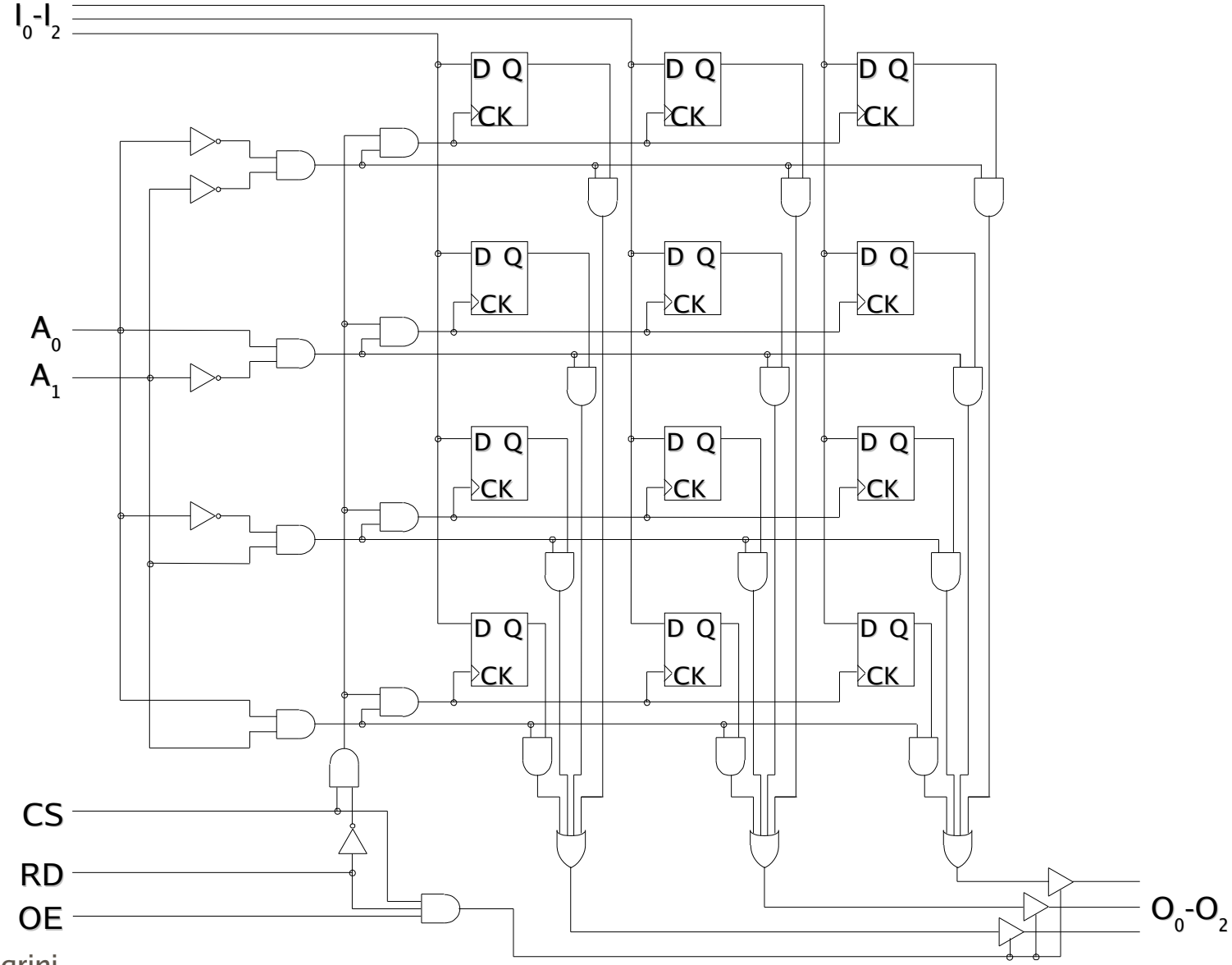
# Adressage mémoire (2)

- 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

# Correction d'erreurs mémoire (1)

- Les mémoires d'ordinateurs peuvent parfois stocker des valeurs erronées, du fait de sautes de tension ou autres
- Pour se protéger de ces phénomènes, les mémoires récentes utilisent des codes auto-correcteurs
  - La mémoire possède des capacités de stockage supplémentaires permettant l'enregistrement d'informations de contrôle
  - On vérifie la cohérence de ces informations chaque fois qu'un mot est lu à partir de la mémoire

# Correction d'erreurs mémoire (2)

- Si, pour chaque ensemble de m bits, on ajoute r bits de contrôle, on lit à chaque fois un ensemble de (m + r) bits constituant un mot de code, à traduire en le mot de m bits voulu
- Le nombre minimum de bits dont deux mots de code quelconques diffèrent est appelée distance de Hamming
  - Si la distance de Hamming d'un code est de 3, on peut corriger une erreur et en détecter deux

# 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 et consomme beaucoup
- On a donc une hiérarchie mémoire, avec au sommet des mémoire rapides et de petites tailles, et en bas des mémoires de grande capacité, très peu chères et peu rapides
  - Registres
  - Caches
  - Mémoire centrale
  - Disque dur ...

# Hiérarchie mémoire (2)

- La hiérarchie mémoire fonctionne grâce au principe 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 continuelle 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* »)

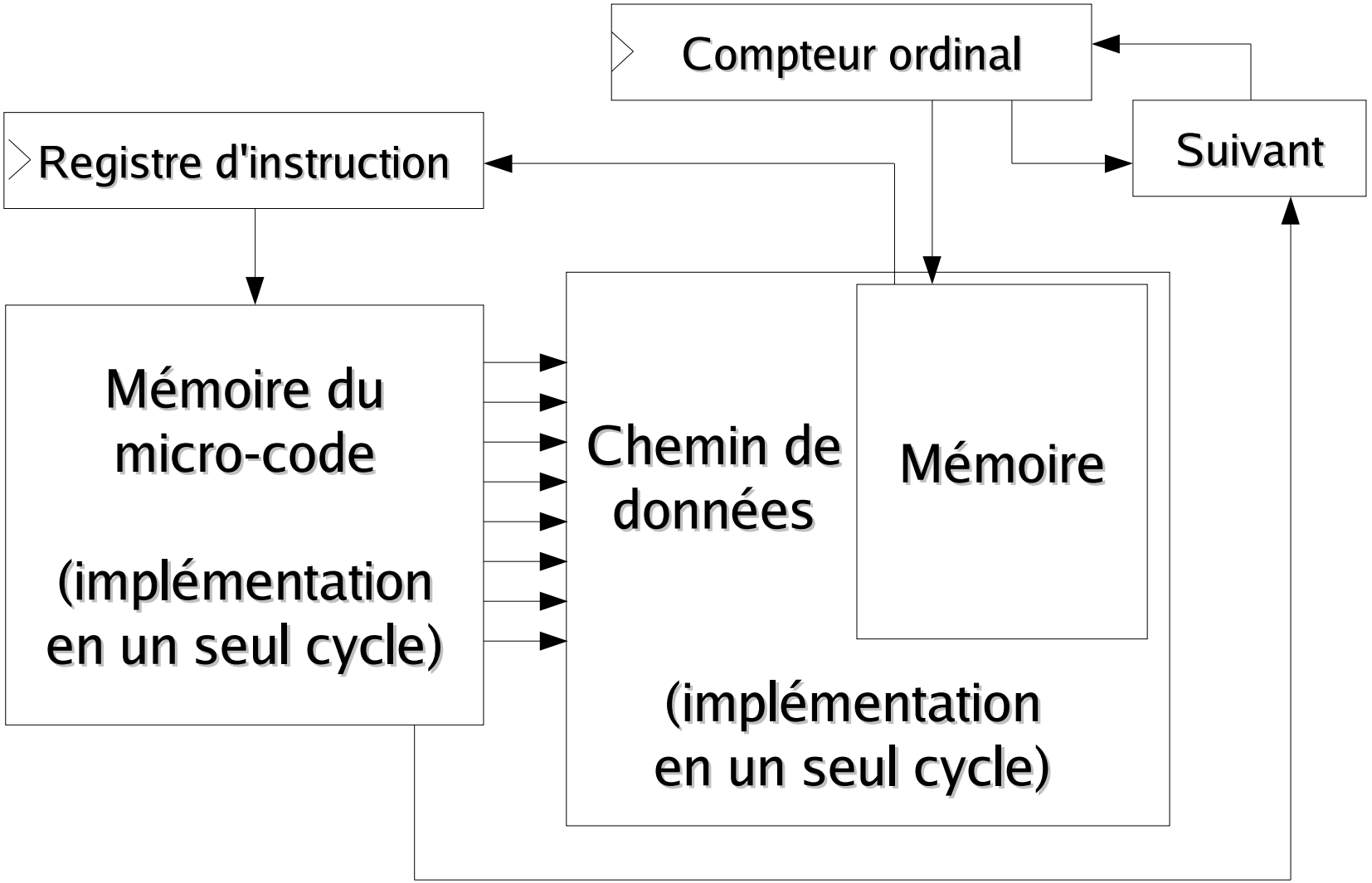
# Micro-architecture (2)

- 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

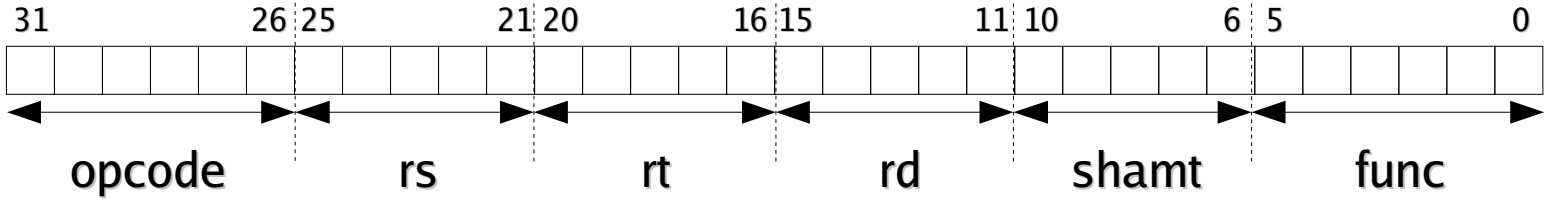


# Instructions (1)

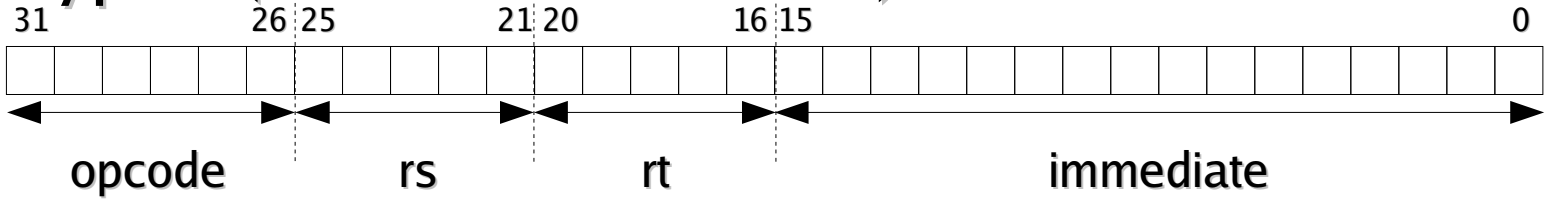
- Chaque instruction est composée d'un ou plusieurs champs
- Le premier, appelé « opcode », code le type d'opération réalisée par l'instruction
  - Opération arithmétique, branchement, etc.
- Les autres champs, optionnels, spécifient les opérandes de l'instruction
  - Registres source et destination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

# Instructions (2)

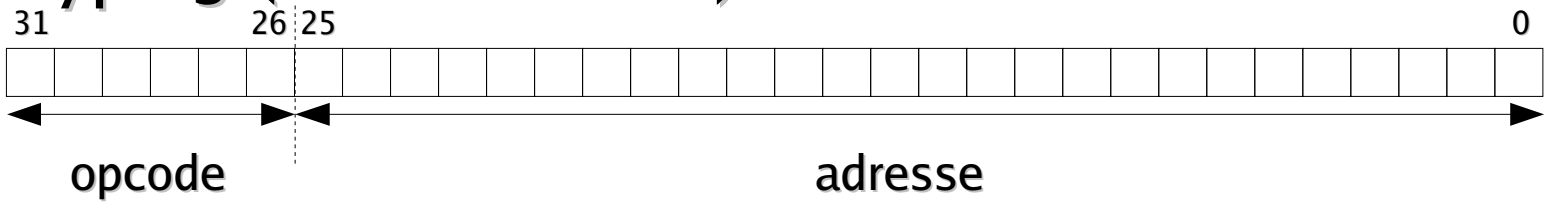
- Exemple : format des instructions RISC MIPS
  - Type R (registre)



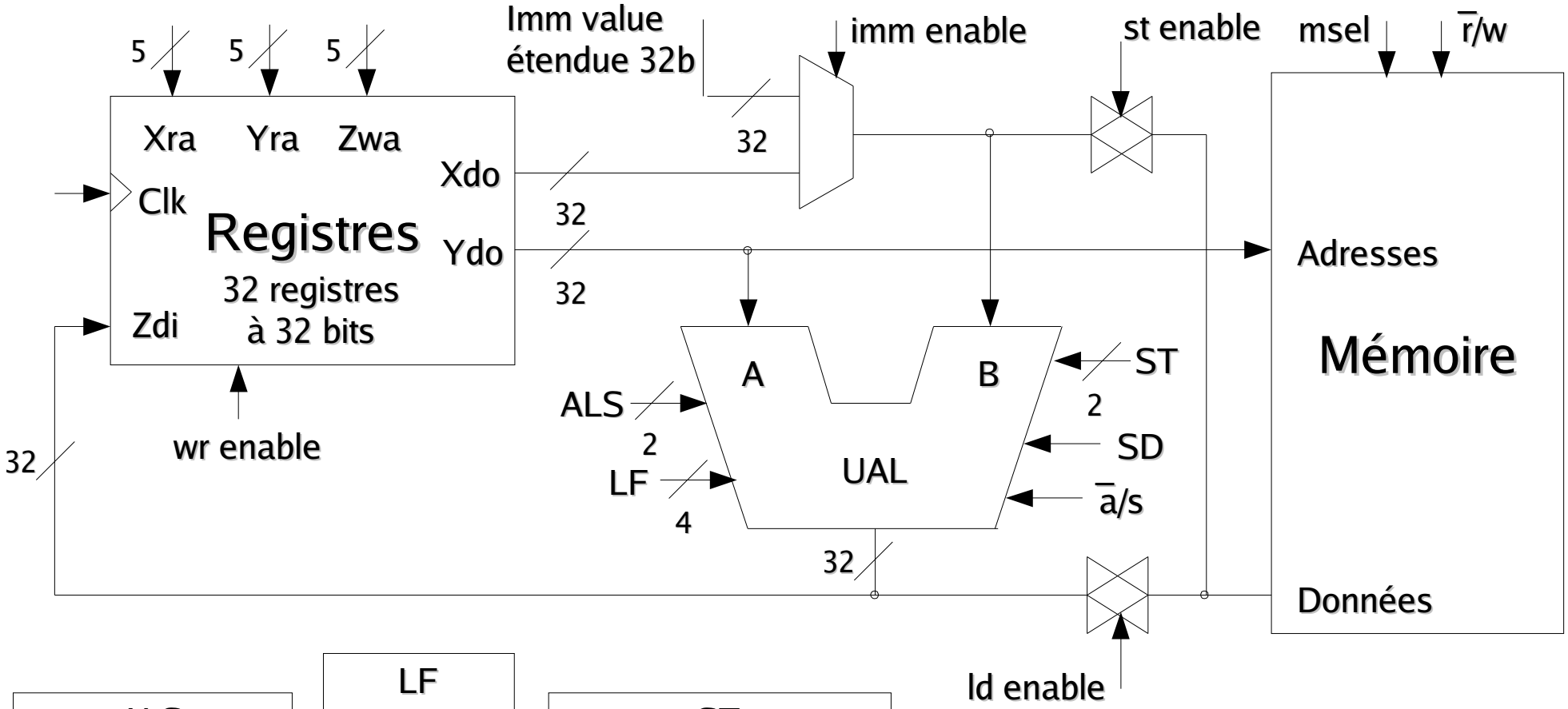

  - Type I (donnée immédiate)




  - Type J (branchement)



# Contrôle du chemin de données (1)




| ALS |
|---|
| 00 : Arithmétique |
| 01 : Logique |
| 10 : Décalage |
| 11 : Desactivée |

| 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 |

| 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

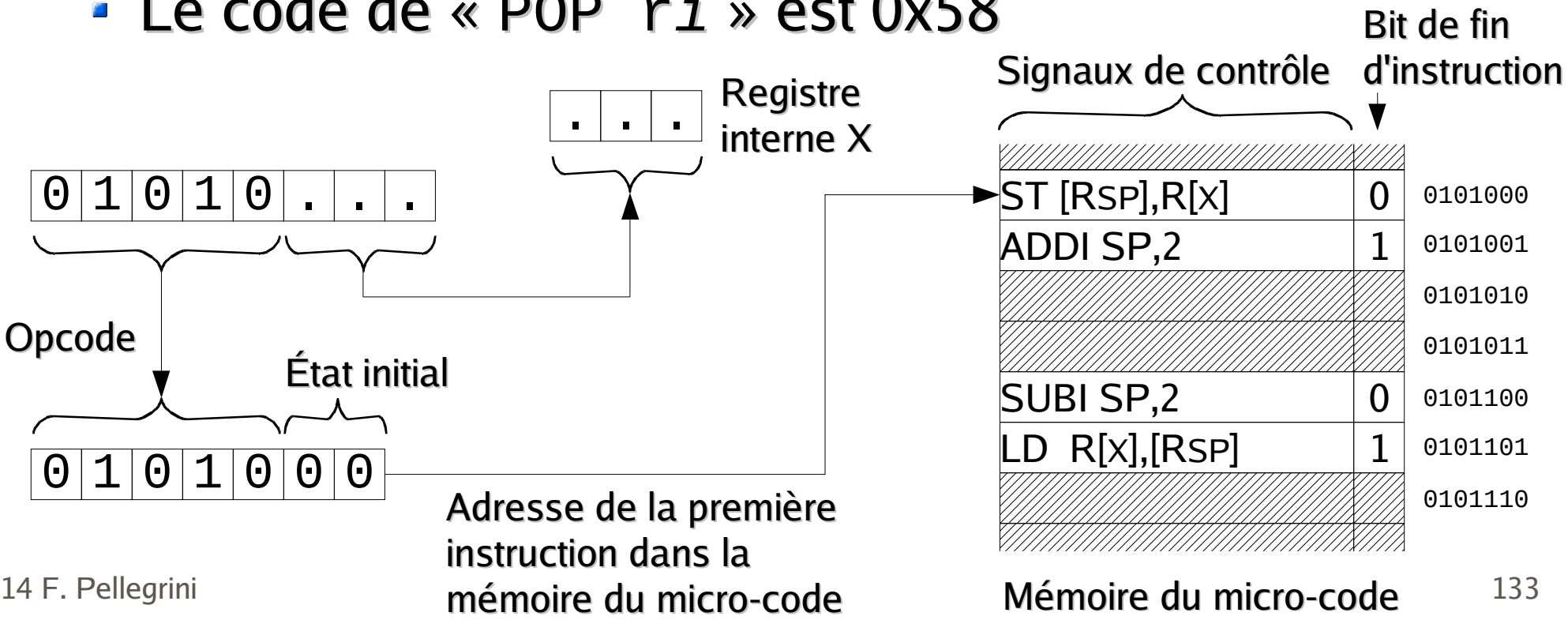
| Instruction | X (5) | Y (5) | Z (5) | we | imm_e | imm_val | ALS | a/s | LF | ST | SD | ld_e | st_e | r/w | msel |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| li r1,100 | x | x | 00001 | 1 | 1 | 0x0100 | 01 | x | 0101 | x | x | 0 | 0 | x | 0 |
| sw r0,(r1) | 00000 | 00001 |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

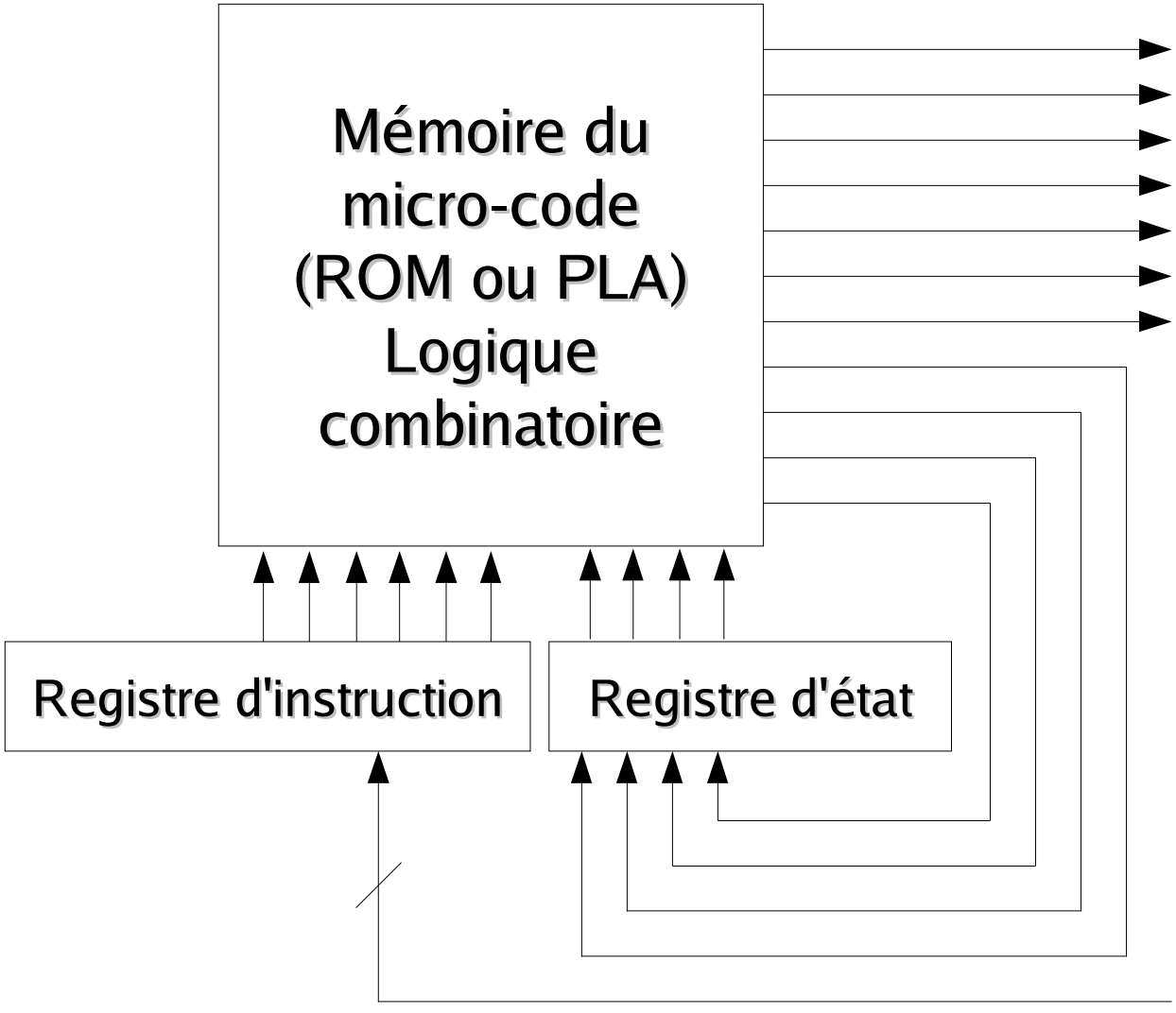
- Dans le cas d'un jeu d'instructions de type CISC, une instruction ISA doit être traduite en plusieurs micro-instructions
  - Cas des instructions « REP SCAS » des x86
- Chaque micro-instruction
  - S'exécute en un cycle élémentaire
  - Spécifie exactement les signaux de contrôle des différentes unités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




Lignes de contrôle
du chemin de données

Pas besoin de boucle
ni de registre d'état si
toutes les instructions
terminent en un
(micro-)cycle

Instruction suivante

# 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

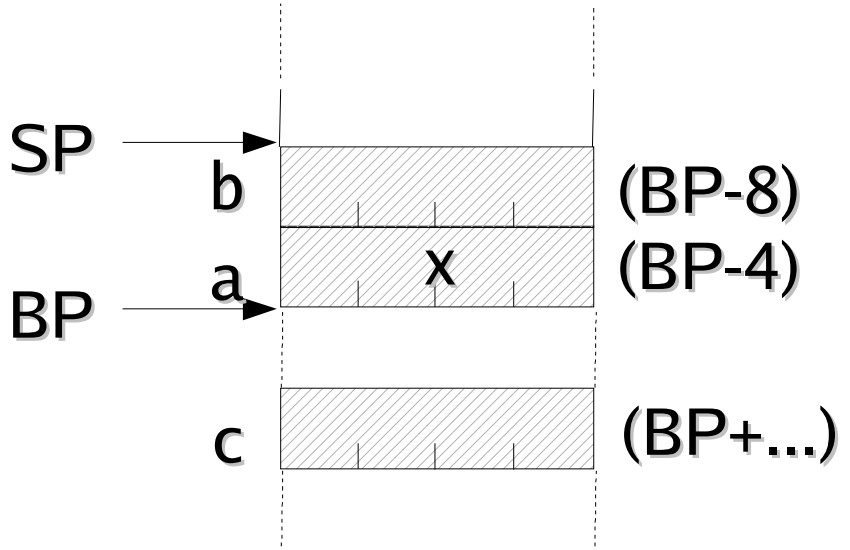
# Pile (2)

- Une pile est une zone de la mémoire que l'on n'accède jamais de façon absolue mais toujours relativement à un registre
- Gérée en fait au moyen de deux registres
  - Un registre de base (« *Base Pointer* », ou BP) pointe sur le début de la zone mémoire allouée pour les variables locales de la procédure courante
  - Un registre de sommet de pile (« *Stack Pointer* », ou SP) pointe sur le dernier mot mémoire alloué

# Pile (3)

- Les paramètres et les variables locales à la procédure courante sont référencées par rapport à la valeur courante de BP
- La zone de données référencée par BP et limitée par SP est appelée « contexte courant »

```
void
f (
int     c)
{
  int   a;
  int   b;
...
  f (a + 1);
...
}
```

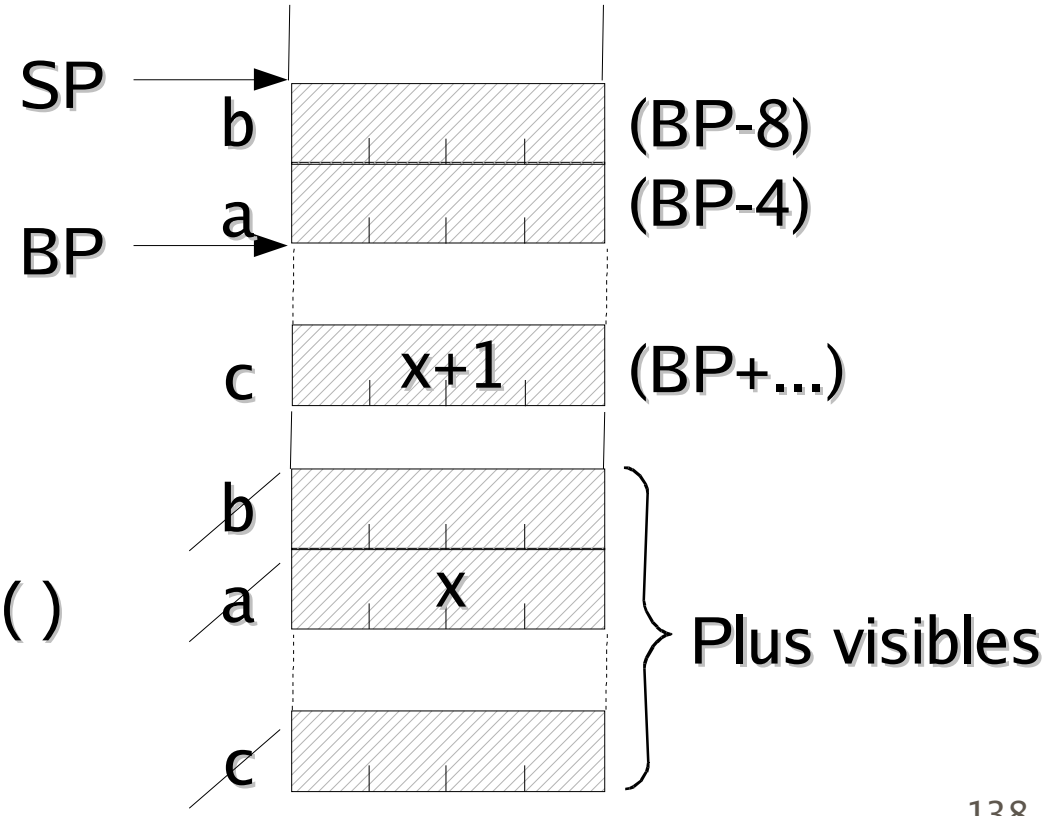


# Pile (4)

- Lors d'un appel de procédure, un nouveau contexte courant se crée au sommet de la pile
  - Comment gérer le retour au contexte appelant ?

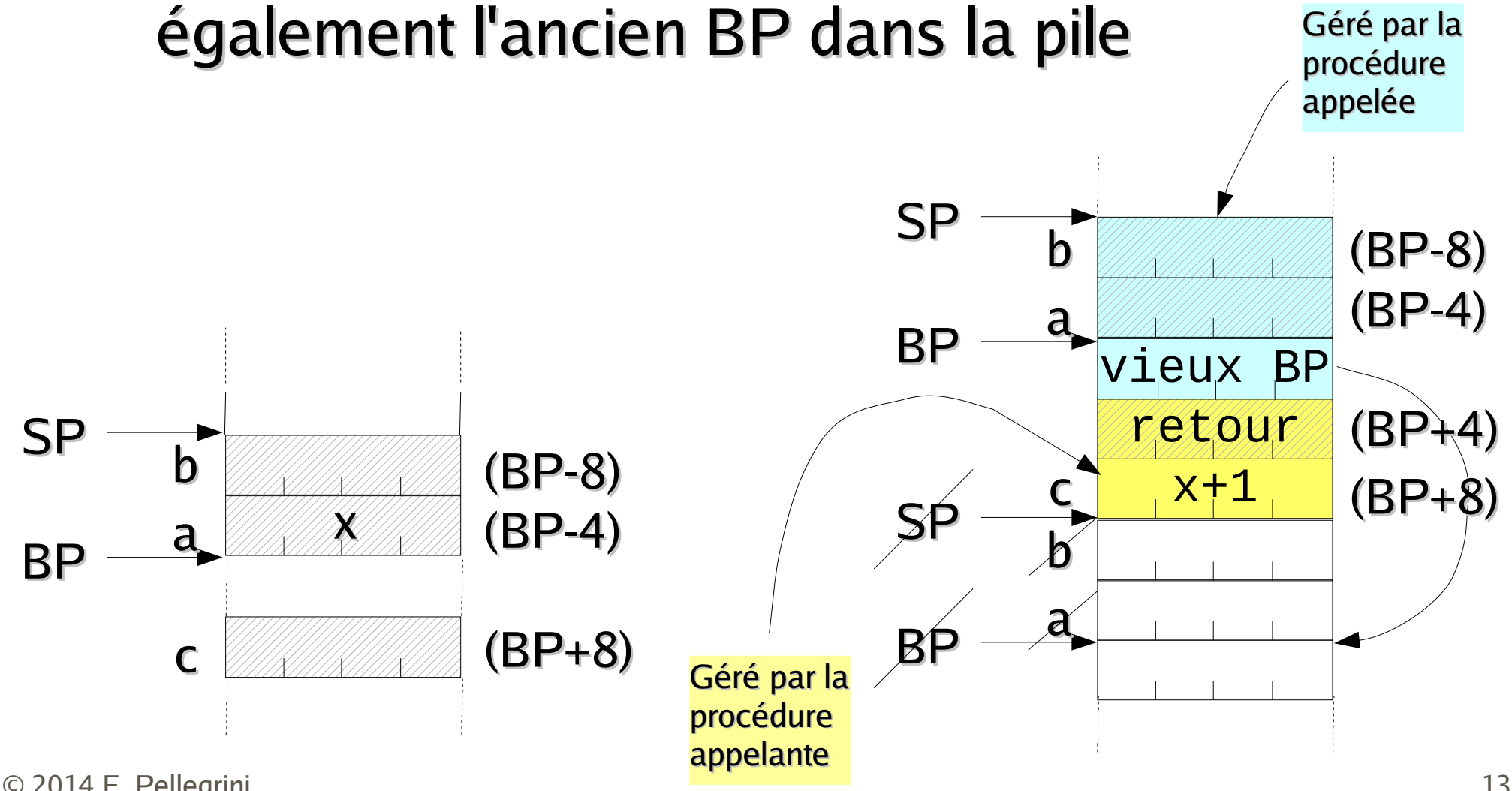
```
void
f (
int     c)
{
  int   a;
  int   b;
...
  f (a + 1);
...
}
```

Appel de `f()`



# Pile (5)

- Lors d'un appel de procédure, on sauve également l'ancien BP dans la pile



# Pile (6)

- Séquence d'appel d'une procédure, partie gérée par la procédure appelante :
  - Empilage des paramètres, dans l'ordre inverse de celui dans lequel ils sont listés dans la procédure
    - Autorise les fonctions à nombre d'arguments variables : le premier paramètre est le plus proche de BP, les autres sont dessous !
  - Appel de la procédure
    - Sauvegarde automatiquement l'adresse de retour dans la pile

# Pile (7)

- Séquence d'appel d'une procédure, partie gérée par la procédure appelée (début de procédure) :
  - Empilage de l'ancien BP dans la pile
  - Copie de la valeur de SP dans celle de BP
    - Le nouveau contexte est basé à la position courante de SP
    - Le premier paramètre est accessible à l'adresse de BP plus la taille de deux adresses entières (l'ancien BP et l'adresse de retour), donc (BP+8)
  - Soustraction à SP de la taille des variables locales
    - Réserve l'espace en cas d'appels ultérieurs

# Pile (8)

- Séquence de retour d'une procédure, partie gérée par la procédure appelée (fin de procédure) :
  - Remise dans SP de la valeur de BP
    - Libère la zone des variables locales à la procédure
  - Dépile BP
    - BP pointe de nouveau sur le contexte appelant
  - Appelle l'instruction de retour
    - Dépile la valeur de retour située dans la pile

# Pile (9)

- Séquence de retour d'une procédure, partie gérée par la procédure appelante :
  - Incrémentation de SP de la taille de tous les paramètres empilés avant l'appel à la procédure
  - Retour complet à l'état antérieur

# 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)

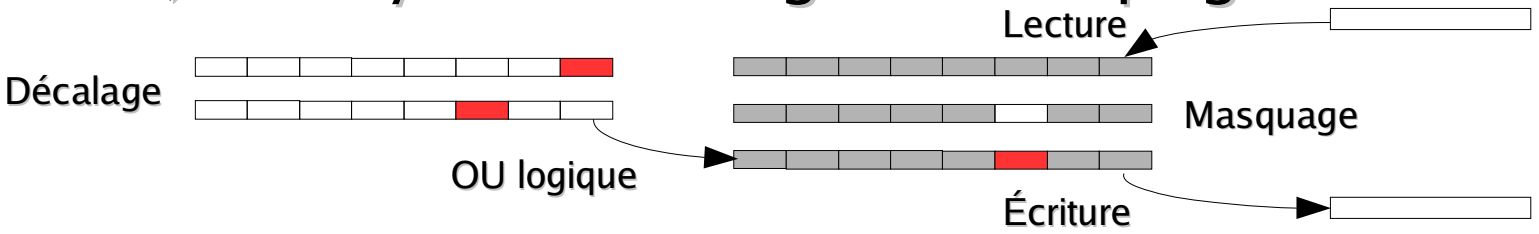
- Les types de données les plus couramment implémentés dans les jeux d'instructions sont :
  - Type entier
  - Type flottant
  - Type caractère

# Types de données entiers (1)

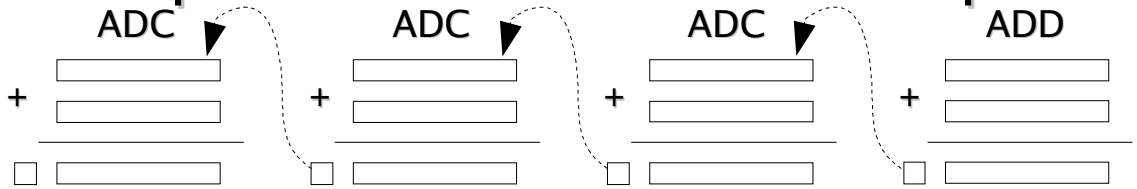
- Le type entier est toujours disponible
  - Sert au fonctionnement de la couche micro-architecture
- Toutes les architectures disposent de types entiers signés
  - Presque toujours codés en complément à deux
  - Il existe aussi souvent des types non signés
- Disponibles en plusieurs tailles
  - Quelques 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 émulés 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
  - Exemple : numéro d'IRQ (« *Interrupt ReQuest* »)

# 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)

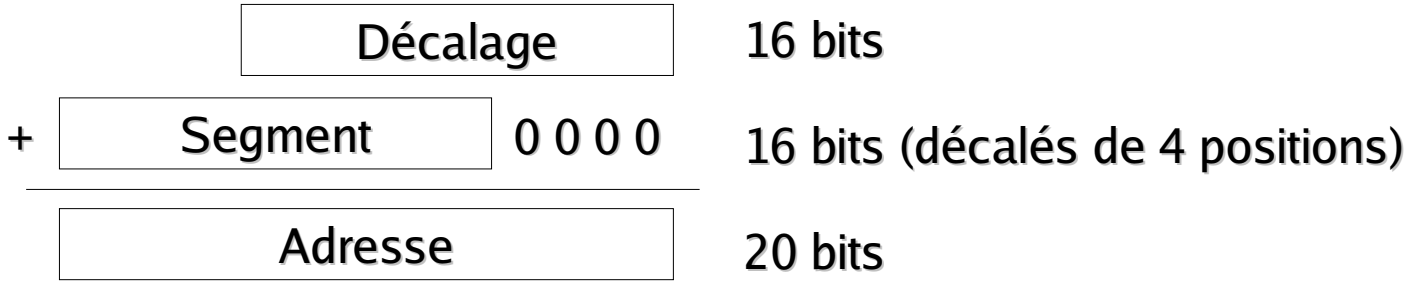
- Au niveau du jeu d'instructions, on trouve donc des instructions
  - Pour générer des interruptions logicielles
  - Pour autoriser ou non l'acceptation des interruptions
    - Ces instructions ne doivent pas être exécutables par les programmes d'application
    - Exécutables seulement en mode privilégié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

# Espace d'adressage (1)

- 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 fils d'adresses (adressage de 16 TéraMots)

# Espace d'adressage (2)

- Pour adresser plus de mots mémoire que ne peut en adresser un mot machine, on peut construire les adresses mémoires en combinant deux mots machine :
  - Un mot de poids fort définissant un segment mémoire
  - Un mot de poids faible définissant un déplacement (« *offset* ») dans le segment considéré
  - Cas du 8086 : mots de 16 bits et 20 fils d'adresses



# Architecture ISA du Pentium II (1)

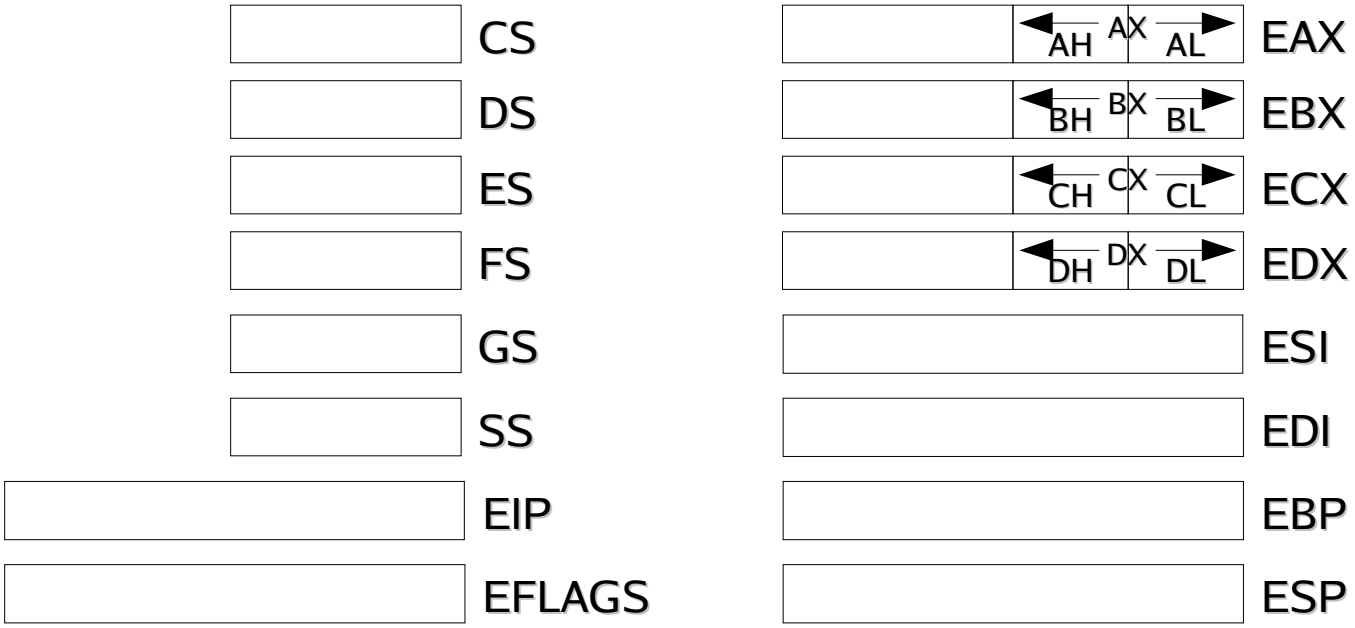
- Architecture appelée IA-32 (ou x86)
- Est le résultat d'une évolution continue depuis le processeur 8 bits 8080
- Maintien d'une compatibilité ascendente permettant encore l'exécution de programmes écrits pour le processeur 16 bits 8086 :
  - Mode réel : le processeur se comporte comme un 8086
  -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

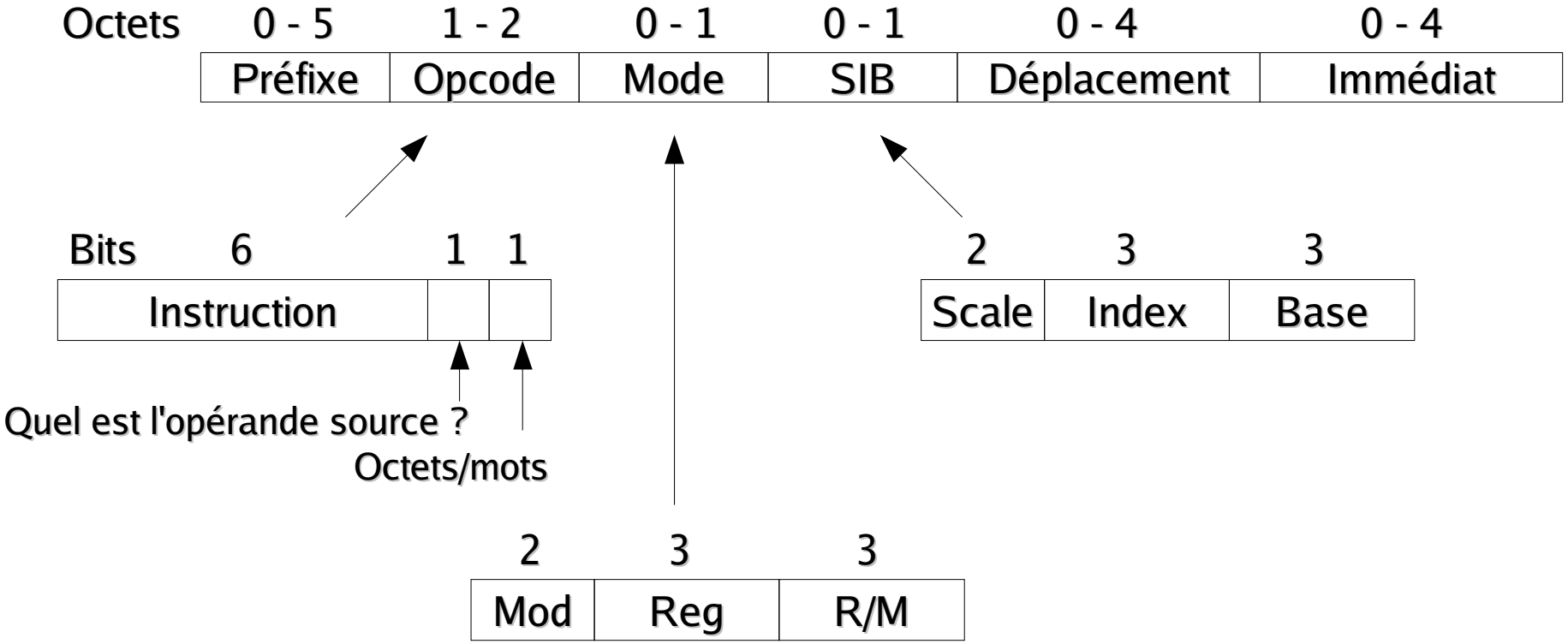
- Architecture à deux adresses, non orthogonale
  - Registres généraux spécialisés




- 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

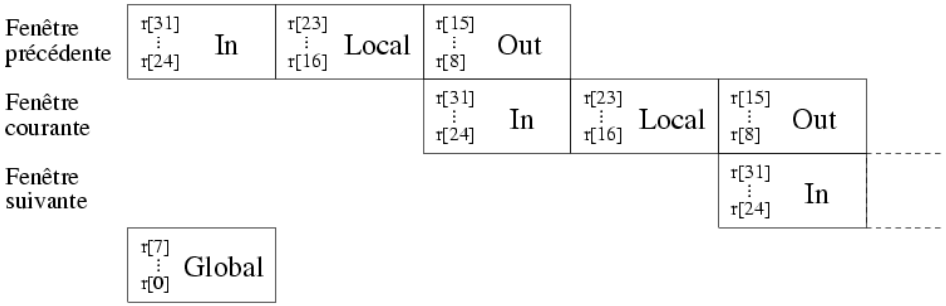


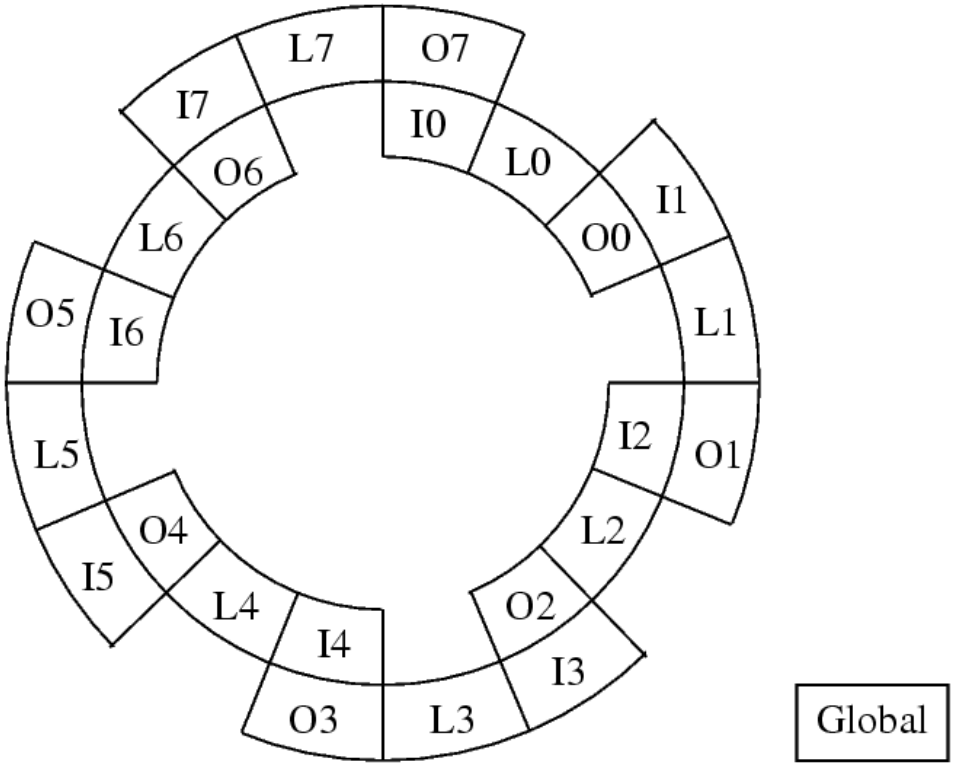
# Architecture ISA de l'UltraSparc (1)

- Architecture SPARC version 9
  - Processeurs développés par différentes entreprises selon les spécifications de l'ISA
- Architecture RISC 64 bits, load/store, orthogonale, à trois adresses
- Espace d'adressage linéaire de $2^{64}$ octets
  - Seulement 44 lignes d'adresse implémentées jusqu'ici

# Architecture ISA de l'UltraSparc (2)

- 32 registres visibles à un instant donné, parmi 136 réellement disponibles
  - Système de fenêtre glissante pour le passage des paramètres



# Architecture ISA de l'IA-64 (1)

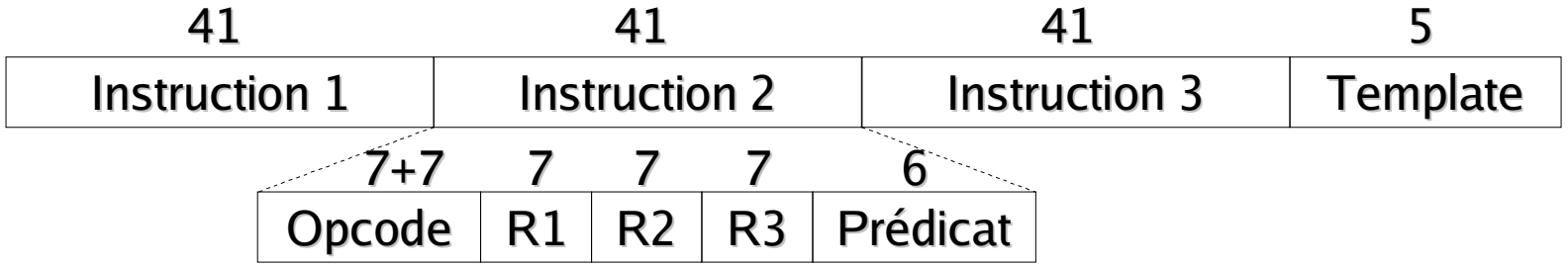
- L'architecture IA-32 avait atteint ses limites
  - Le surcoût de complexité dû au support de la compatibilité ascendente devenait trop pénalisant
- L'architecture IA-64 repart de zéro
  - Architecture RISC 64 bits, load/store, orthogonale
    - 128 registres généraux de 64 bits
    - 128 registres flottants de 82 bits
- Mais possibilité pour les processeurs Intel de fonctionner soit en mode IA-32, soit en mode IA-64

# Architecture ISA de l'IA-64 (2)

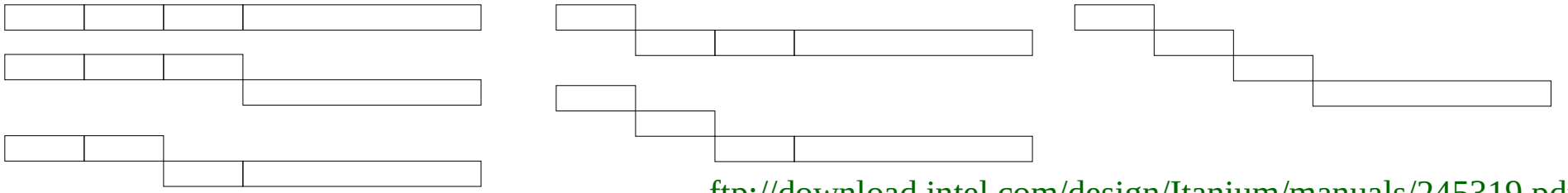
- Paradigme EPIC : « *Explicitely Parallel Instruction Computing* »
  - Le compilateur, lorsqu'il génère le code machine, identifie les instructions pouvant s'exécuter en parallèle sans risque de conflit
  - En fonction de la puissance du processeur, et plus précisément de son degré de superscalarité, plus ou moins de ces instructions indépendantes pourront être exécutées en parallèle
  - La complexité du séquencement des instructions est déportée vers le compilateur, permettant de simplifier, et donc d'accélérer, l'exécution

# Architecture ISA de l'IA-64 (3)

- Les instructions sont groupées par trois en liasses (« *bundles* ») de 128 bits




  - Un champ « *template* » spécifique à la liasse spécifie les dépendances temporelles entre instructions de la liasse ainsi que vis-à-vis de la liasse suivante



ftp://download.intel.com/design/Itanium/manuals/245319.pdf

# Architecture ISA de l'IA-64 (4)

- Grâce à des instructions à prédicat, on peut fortement réduire le nombre de branchements conditionnels, qui sont une cause très importante de perte de performance
  - Moins d'instructions
  - Absence de rupture de pipe-line lors de mauvaises prédictions de branchements

```
if (R1 == 0)
  R2 = R3;
```

```
      CMP   R1,0
      JNE   ET1
      MOV   R2,R3
ET1:  ...
```

```
      CMOVZ R2,R3,R1
      ...
```

# Architecture ISA de l'IA-64 (5)

- Grâce à des instructions conditionnelles, dont le prédicat est la valeur d'un des 64 registres de prédiction, on peut étendre ce principe à tous les types d'instructions
  - Des instructions permettent de positionner $P_{2i}$ et $P_{2i+1}$

```
if (R1 == R2)
  R3 = R4 + R5;
else
  R6 = R4 - R5;
```

```
      CMP   R1,R2
      BNE   ET1
      MOV   R3,R4
      ADD   R3,R5
      JMP   ET2
ET1:  MOV   R6,R4
      SUB   R6,R5
ET2:  ...
```

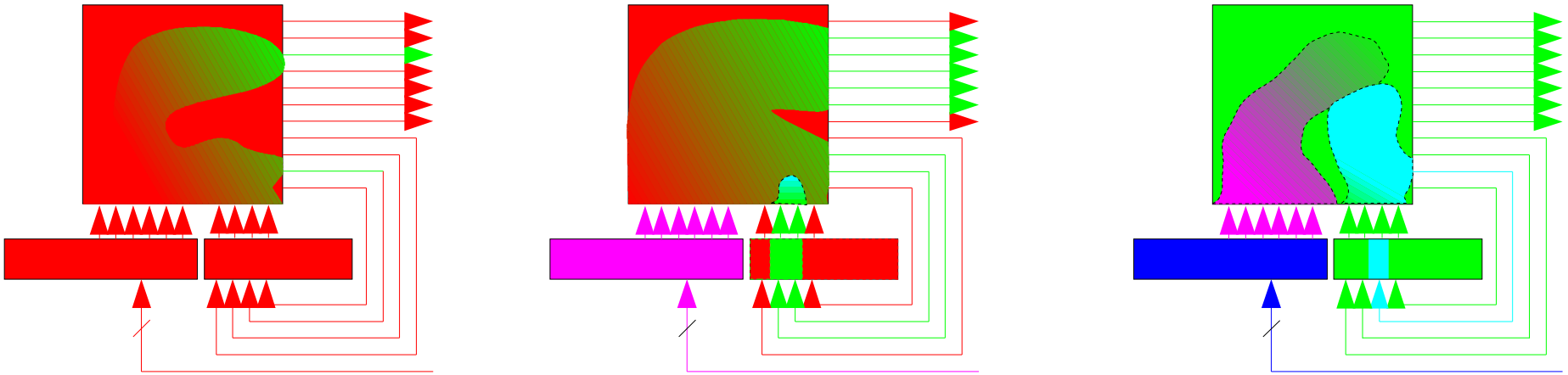
```
      CMPEQ R1,R2,P4
<P4>  ADD   R3,R4,R5
<P5>  SUB   R6,R4,R5
```

# Architecture ISA de l'IA-64 (6)

- Des instructions de pré-chargement (« *pre-fetching* ») permettent d'anticiper les accès à la mémoire et donc de recouvrir le temps d'accès aux données par des calculs utiles
  - Instruction LOAD spéculative pour démarrer une lecture par anticipation
  - Instruction CHECK pour vérifier si la donnée est bien présente dans le registre avant de l'utiliser
    - Si la donnée est présente, se comporte comme un NOP
    - Si l'accès est invalide, provoque une exception comme si la lecture venait d'avoir lieu

# Circuits synchrones (1)

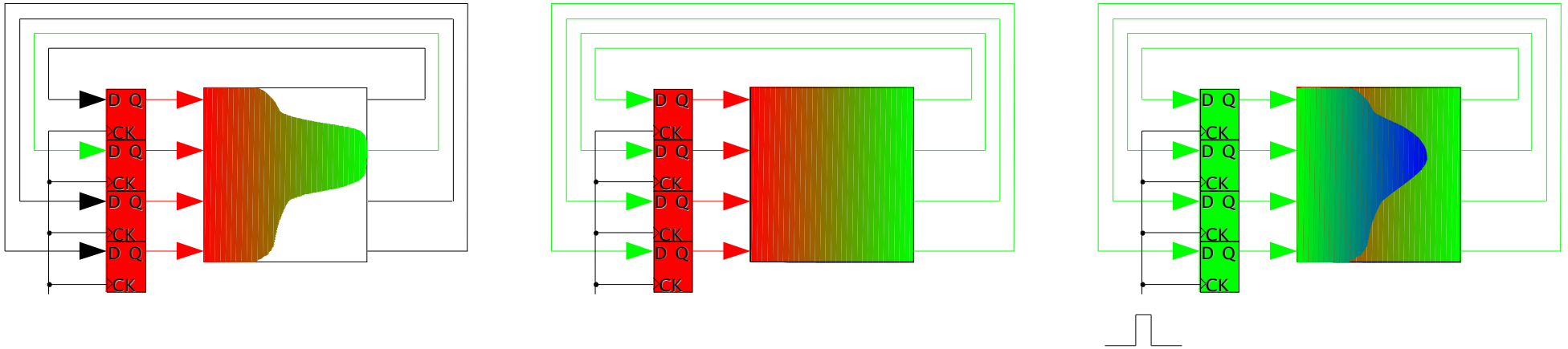
- En l'absence de synchronisation, les résultats des calculs des circuits avec boucle de rétroaction seraient inexploitables car faux



- 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

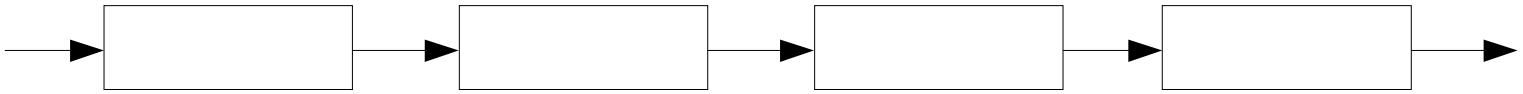
# Circuits synchrones (3)

- 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

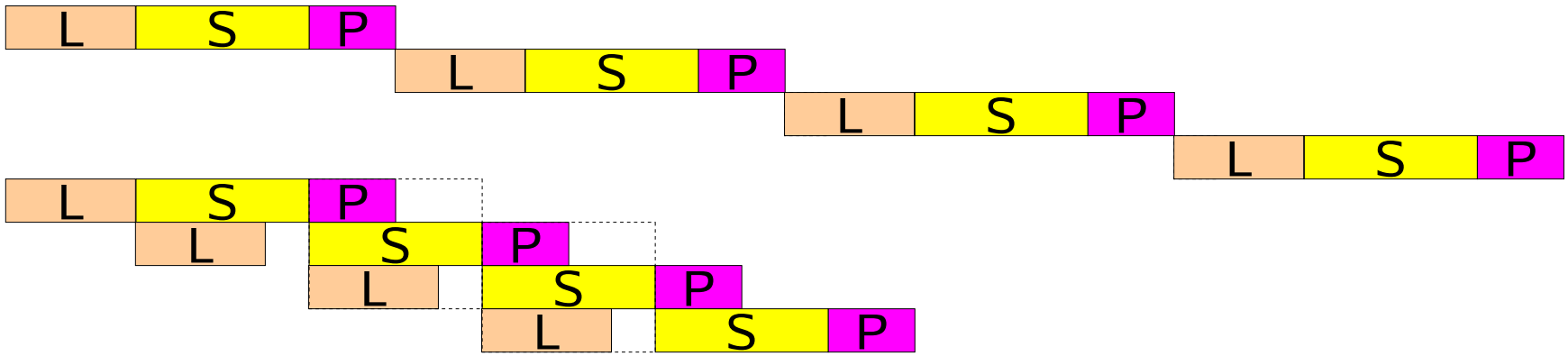
Unité fonctionnelle non pipelinée



Unité fonctionnelle pipelinée à 4 étages

# Pipe-line (2)

- Exemple : le lavomatique
  - Lavage : 30 minutes
  - Séchage : 40 minutes
  - Pliage : 20 minutes

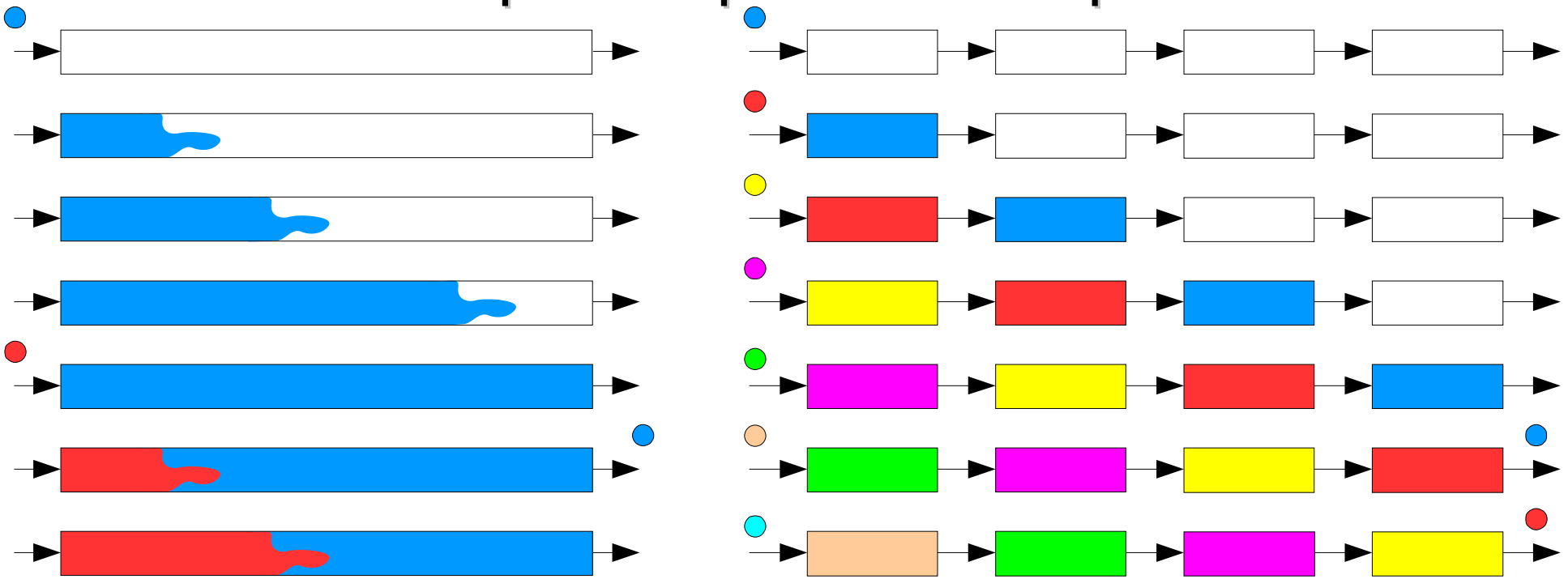


# Pipe-line (3)

- 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)
  - λ 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 + λ)
- 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
    - Nécessité d'introduire de plus en plus de « bulles »
    - Ruptures de pipe-line de plus en plus coûteuses
      - Problème des branchements conditionnels

# Dépendances et bulles (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 et bulles (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
```

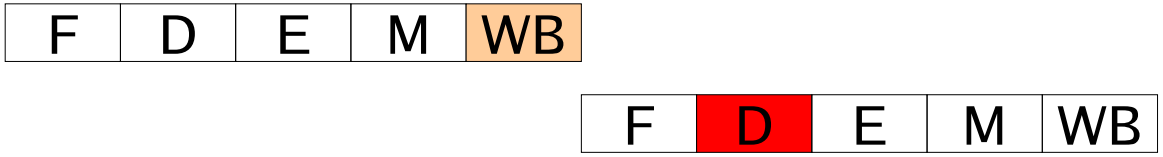
- Dépendance de contrôle

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

# Dépendances et bulles (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

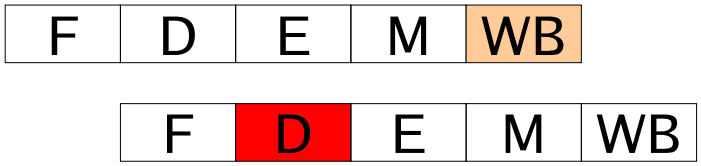
```
add  r2,r3,r1
add  r4,r1,r4
```



# Dépendances et bulles (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

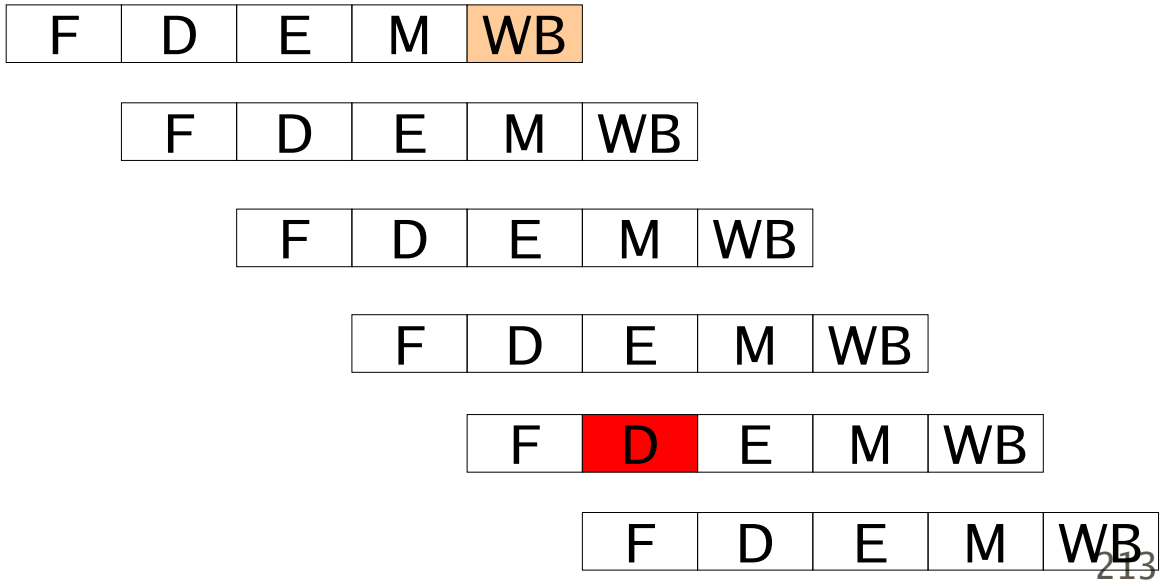
```
add  r2,r3,r1
add  r4,r1,r4
```



# Dépendances et bulles (5)

- Pour éviter les incohérences, on peut :
  - Intercaler des instructions indépendantes
    - Solution non portable selon la profondeur du pipe-line
  - Mettre des nop
    - Perte de performance du pipe-line

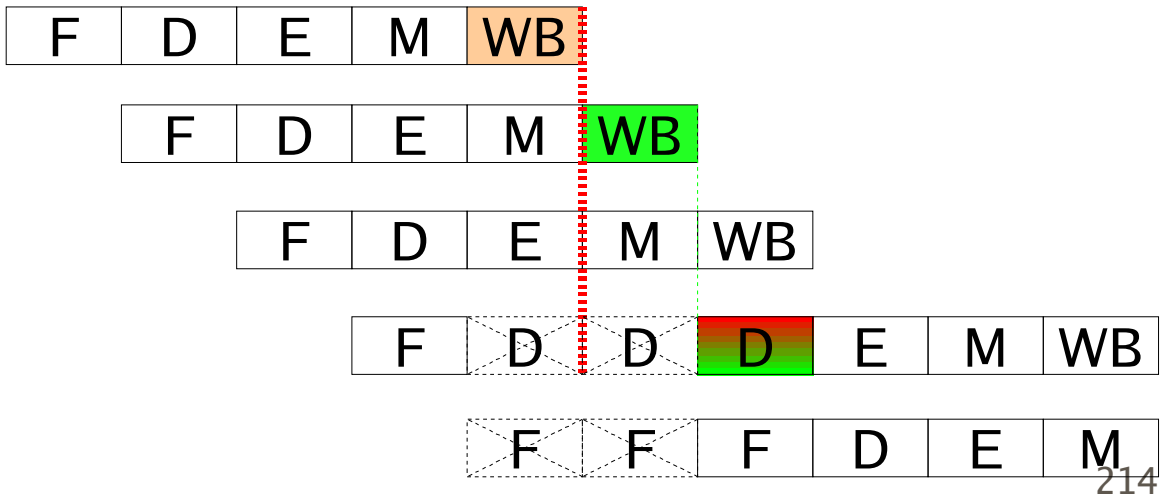
```
add  r2,r3,r1
mov 1,r4
add  r5,r6,r6
nop
add  r4,r1,r4
...
```



# Dépendances et bulles (6)

- Afin de ne pas augmenter inutilement la taille du code, les processeurs détectent automatiquement les conflits d'accès et insèrent automatiquement des « bulles » dans le pipe-line d'instructions
  - « *Register scoreboarding* » : marquage des usages

```
add  r2,r3,r1
mov 1,r4
add  r5,r6,r6
add  r4,r1,r4
...
```



# Dépendances et bulles (7)

- C'est au compilateur d'optimiser l'exécution du code en entrelaçant les instructions indépendantes
  - Nécessite plus de registres pour casser les anti-dépendances et les dépendances de résultat

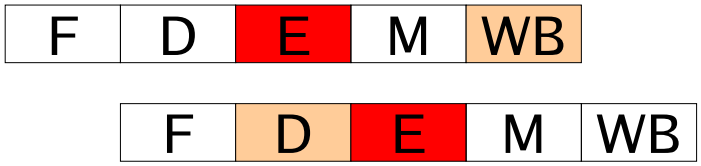
```
mul   [B],[C],r1
mul   [D],[E],r2
add   r1, r2,[A]
mul   [G],[H],r1
mul   [I],[J],r2
add   r1,r2,[F]
```

```
mul   [B],[C],r1
mul   [D],[E],r2
mul   [G],[H],r3
mul   [I],[J],r4
add   r1,r2,[A]
add   r3,r4,[F]
```

# *Data forwarding* (1)

- En fait, dans bien des cas, la valeur résultat d'une instruction existe avant le moment où elle doit être utilisée par l'instruction suivante
  - Cas des instructions arithmétiques et logiques, pour lesquelles les valeurs existent dès la fin de l'étage Execute

```
add  r2,r3,r1
add  r4,r1,r4
```




- Il est donc inutile d'attendre qu'elle transitent jusqu'à l'étage Write Back pour les utiliser

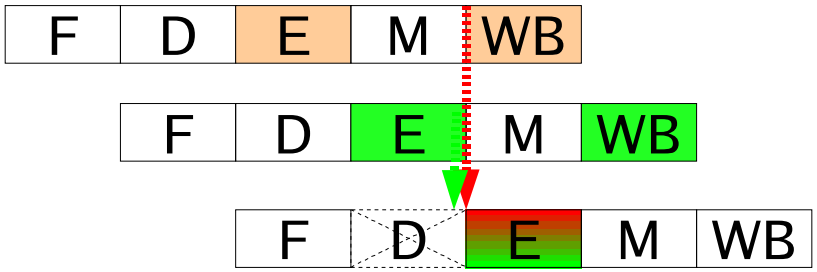
# *Data forwarding* (2)

- On peut donc modifier l'architecture du processeur pour réduire la pénalité associée aux dépendances
- Il suffit de propager la valeur calculée par une instruction vers l'étage qui la demande, sans attendre qu'elle ait été ré-écrite dans la banque de registres
  - Transmission anticipée de la valeur d'un registre : « *data forwarding* »

# *Data forwarding* (3)

- Dès qu'une valeur a été calculée (étage E) ou lue (étage M), on peut la renvoyer vers l'étage E sans passer par les phases WB et D

```
add   r2,r3,r1
mov   1,r4
add   r4,r1,r4
```




- Nécessite des multiplexeurs supplémentaires pour alimenter l'ALU à partir des étages suivants
  - Plus on a d'étages, plus le multiplexage est coûteux

# *Data forwarding* (4)

- Le *data forwarding* permet de résoudre les dépendances entre instructions arithmétiques
  - Dès qu'un résultat a été calculé dans l'étage d'exécution, il peut être réinjecté à cet étage
- Les seules dépendances entre instructions nécessitant un blocage proviennent de l'attente d'un résultat provenant des étages suivants
  - « *Load-use hazard* » :

```
mov  (r2),r1
add   r4,r1,r4
```

# Branchements conditionnels (1)

- Les branchements conditionnels représentent la plus grande source de bulles
- Tant que la condition n'est pas évaluée, on ne peut savoir avec certitude quelle branche exécuter
  - La solution la plus simple en termes d'architecture est de bloquer l'exécution des instructions suivantes jusqu'à disparition de l'incertitude
    - Mais c'est extrêmement coûteux !

# Branchements conditionnels (2)

- Pour réduire le coût des branchements conditionnels, on peut utiliser plusieurs techniques :
  - Réduire le nombre de branchements : déroulage de boucles (« *loop unrolling* »)
  - Exécuter de façon prospective l'une des deux branches
  - Prendre le plus souvent possible la « bonne » branche : prédiction de branchement

# Branchements conditionnels (3)

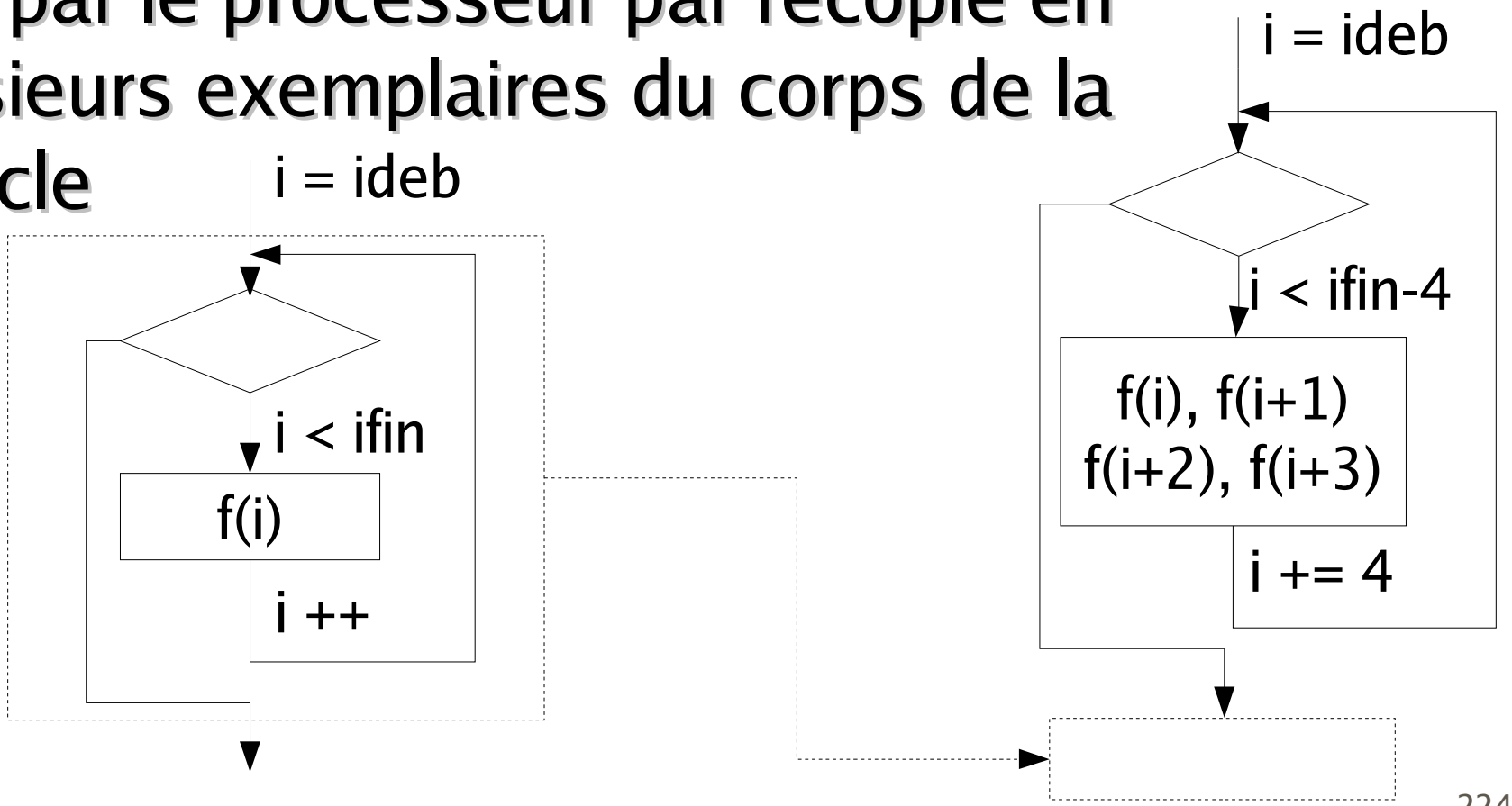
- Diminuer la pénalité de purge du pipe-line : branchement retardé
- Prendre les deux branches à la fois : exécution spéculative
  - Nécessite une architecture très complexe, mêlant super-scalarité et « *out-of-order execution* »

# Branchements conditionnels (4)

- Une source de dépendance spécifique aux processeurs CISC est le mot d'état programme
  - Entre une instruction modifiant les drapeaux et l'instruction de branchement conditionnel qui les utilise, il n'est pas possible d'intercaler d'autres instructions indépendantes mais qui modifient elles aussi les drapeaux
- Solution : disposer de drapeaux indépendants
  - Registres de prédicat de l'architecture IA-64
  - Permettent de pré-calculer plusieurs conditions

# Déroulage de boucle (1)

- Technique permettant de réduire le nombre de branchements conditionnels vus par le processeur par recopie en plusieurs exemplaires du corps de la boucle



# Déroulage de boucle (2)

- La duplication de code permet l'entrelacement d'instructions indépendantes
- L'exécution des dernières (ou premières) itérations peut se faire au prix d'un seul test par tour
  - On amortit le coût de détection des cas particuliers en les testant dans l'ordre croissant du nombre d'itérations restant à faire

# Déroulage de boucle (3)

```
int             t[1000], n, s, i;

n = f ();                       // f() renvoie 1000 mais le
for (i = 0, s = 0; i < n; i ++)     // compilateur ne le sait pas
  s += t[i];
```

```
    ...
    xorl %edx,%edx          # Somme à zéro
    movl %eax,%esi          # Nombre dans esi
    movl %edx,%eax          # Compteur à zéro
    cmpl %esi,%edx          # Valeur de fin atteinte ?
    jge .L45                # Si oui, rien à faire
    movl %esi,%ecx          # Copie compteur dans ecx
    leal -4000(%ebp),%ebx # Adresse tableau dans ebx
    andl $3,%ecx            # Si compteur multiple de 4
    je .L47                 # Va à la boucle déroulée
    cmpl $1,%ecx            # Si valeur modulo 4 est 1
    jle .L59                # Fait un tour et déroule
    cmpl $2,%ecx            # Si valeur modulo 4 est 2
    jle .L60                # Fait deux tours et déroule
    movl -4000(%ebp),%edx # Charge première valeur
    movl $1,%eax            # Un tour fait, reste deux
.L60:
    addl (%ebx,%eax,4),%edx     # Fait un tour de boucle
    incl %eax                   # Incrémente le compteur
.L59:
    addl (%ebx,%eax,4),%edx     # Fait un tour de boucle
    incl %eax                   # Incrémente le compteur
    cmpl %esi,%eax              # Valeur de fin atteinte ?
    jge .L45                    # Termine si c'est le cas
    .align 4                    # Alignement pour cache
.L47:                           # Corps de boucle déroulée
    addl (%ebx,%eax,4),%edx     # Déroulage d'ordre 4
    addl 4(%ebx,%eax,4),%edx
    addl 8(%ebx,%eax,4),%edx
    addl 12(%ebx,%eax,4),%edx
    addl $4,%eax                # Ajoute 4 au compteur
    cmpl %esi,%eax              # Valeur de fin atteinte ?
    jl .L47                     # Reboucle si non atteinte
.L45:   ...
```

# Exécution prospective

- On continue à exécuter de façon prospective l'une des deux branches
  - On ne purgera le pipe-line que s'il s'avère que l'on n'a pas suivi la bonne branche
  - Nécessite une circuiterie supplémentaire pour transformer en bulles les instructions en amont de l'étage d'exécution
- Pour prendre le plus possible la bonne branche, on peut mettre en place des méthodes de prédiction de branchement

# Prédiction de branchement

- Deux types de prédiction de branchement :
  - Statique : la prédiction effectuée sur un branchement donné est pré-calculée à la compilation
  - Dynamique : la décision prise sur un branchement à une adresse donnée dépend de décisions prises par le passé dans l'exécution du programme
    - Nécessaire quand le biais du branchement évolue avec le temps

```
max = a[0];
for (i = 1; i < N; i ++)
  if (a[i] > max)
    max = a[i];
```

# Prédiction de branchement (2)

- La prédiction statique peut être :
  - Identique pour tous les branchements :
    - « *Always taken* » : environ 60 % de réussite
    - « *Backward taken, forward not taken* » (BTFNT) : environ 65 % de réussite
      - Favorise les boucles : branchement de sortie descendant (non pris) ou remontant (pris)
  - Déterminée pour chaque branchement par le compilateur après analyse statique ou dynamique du code : environ 90 % de réussite
    - Drapeau ajouté à l'instruction de branchement conditionnel

# Prédiction de branchement (3)

- La prédiction dynamique peut être :
  - Locale : seul l'historique du branchement en question est considéré
    - Prédicteur à un ou deux bits conservant l'historique des dernières décisions prises pour le branchement
    - On ira dans le sens des dernières décisions
  - Globale : on considère les décisions prises pour les branchements précédents, quels qu'ils soient
    - Permet de traiter les branchements corrélés

# Branchement retardé (1)

- Lors d'une mauvaise prédiction, tous les étages en amont de l'étage d'exécution doivent être purgés
  - Facteur limitatif à l'augmentation de la longueur des pipe-lines d'instructions
- Si les instructions présentes dans le pipe-line étaient exécutables quel que soit la branche prise, il n'y aurait pas besoin de purger le pipe-line

# Branchement retardé (2)

- Un processeur mettant en œuvre un branchement retardé de d étapes ne purgera jamais les d instructions les plus proches de l'étage d'exécution
  - À charge pour le compilateur de trouver des instructions utiles à y placer ou d'y mettre des nop

```
load   r1, [A]
dec    r2
be     r2, pc+2
div    r1, r1, r2
mul    r1, r1, r3
```

Avec d = 0

```
dec    r2
bz     r2, pc+3
load   r1, [A]
div    r1, r1, r2
mul    r1, r1, r3
```

Avec d = 1

```
dec    r2
bz     r2, pc+4
load   r1, [A]
nop
div    r1, r1, r2
mul    r1, r1, r3
```

Avec d = 2

# 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* »)

# Mesure de la performance (1)

- La fréquence d'horloge est un critère important, mais pas déterminant
- D'autres critères entrent également en jeu :
  - RISC vs. CISC
  - Pipe-line
  - Superscalarité
  - Hiérarchie mémoire
  - …

# Mesure de la performance (2)

- La mesure la plus pertinente de l'efficacité de l'architecture est d'évaluer le nombre de cycles par instruction (CPI)
  - Idéalement égal à 1 pour les architectures scalaires
  - Idéalement égal à 1/d pour les architectures superscalaires de degré d
  - Le traitement des dépendances entre instructions fait augmenter cette valeur

# Mesure de la performance (3)

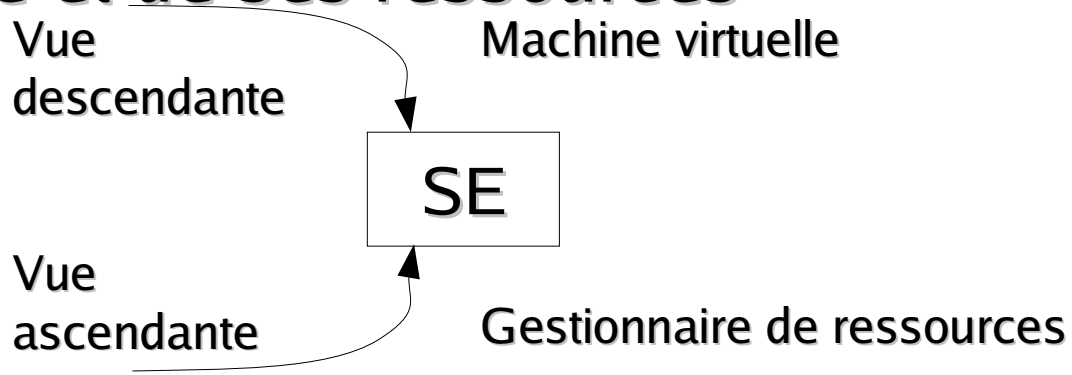
- Pour comparer les performances de deux machines différentes, il faut décomposer le temps d'exécution en ses constituants :
  - T : temps total
  - $\tau$ : temps de cycle
  - c : nombre de cycles $T = c \cdot \tau$
  - i : nombre d'instructions $T = i \cdot (c / i) \cdot \tau$
- Le nombre d'instructions dépend de l'ISA et du compilateur

# Système d'exploitation (1)

- Un système est un programme qui, du point de vue du programmeur, ajoute une variété d'instructions et de fonctionnalités en plus de celles déjà offertes par la couche ISA
- La couche système contient toutes les instructions de la couche ISA, moins les dangereuses, et plus celles ajoutées par le système, sous la forme d'appels système
- Les nouvelles instructions de la couche système sont toujours interprétées

# Système d'exploitation (2)

- Buts d'un système d'exploitation
  - Décharger le programmeur d'une tâche de programmation énorme et fastidieuse et lui permettre de se concentrer sur l'écriture de son application
  - Protéger le système et ses usagers de fausses manipulations
  - Offrir une vue simple, uniforme, et cohérente de la machine et de ses ressources



# Système d'exploitation (3)

- En tant que machine virtuelle, le système fournit :
  - Une vue uniforme des entrées/sorties
  - Une mémoire virtuelle et partageable
  - La gestion sécurisée des accès
  - La gestion des processus
  - La gestion des communications inter-processus

# Système d'exploitation (4)

- En tant que gestionnaire de ressources, le système doit permettre :
  - D'assurer le bon fonctionnement des ressources et le respect des délais
  - L'identification de l'usager d'une ressource
  - Le contrôle des accès aux ressources
  - L'interruption d'une utilisation de ressource
  - La gestion des erreurs
  - L'évitement des conflits

# Gestion de la mémoire

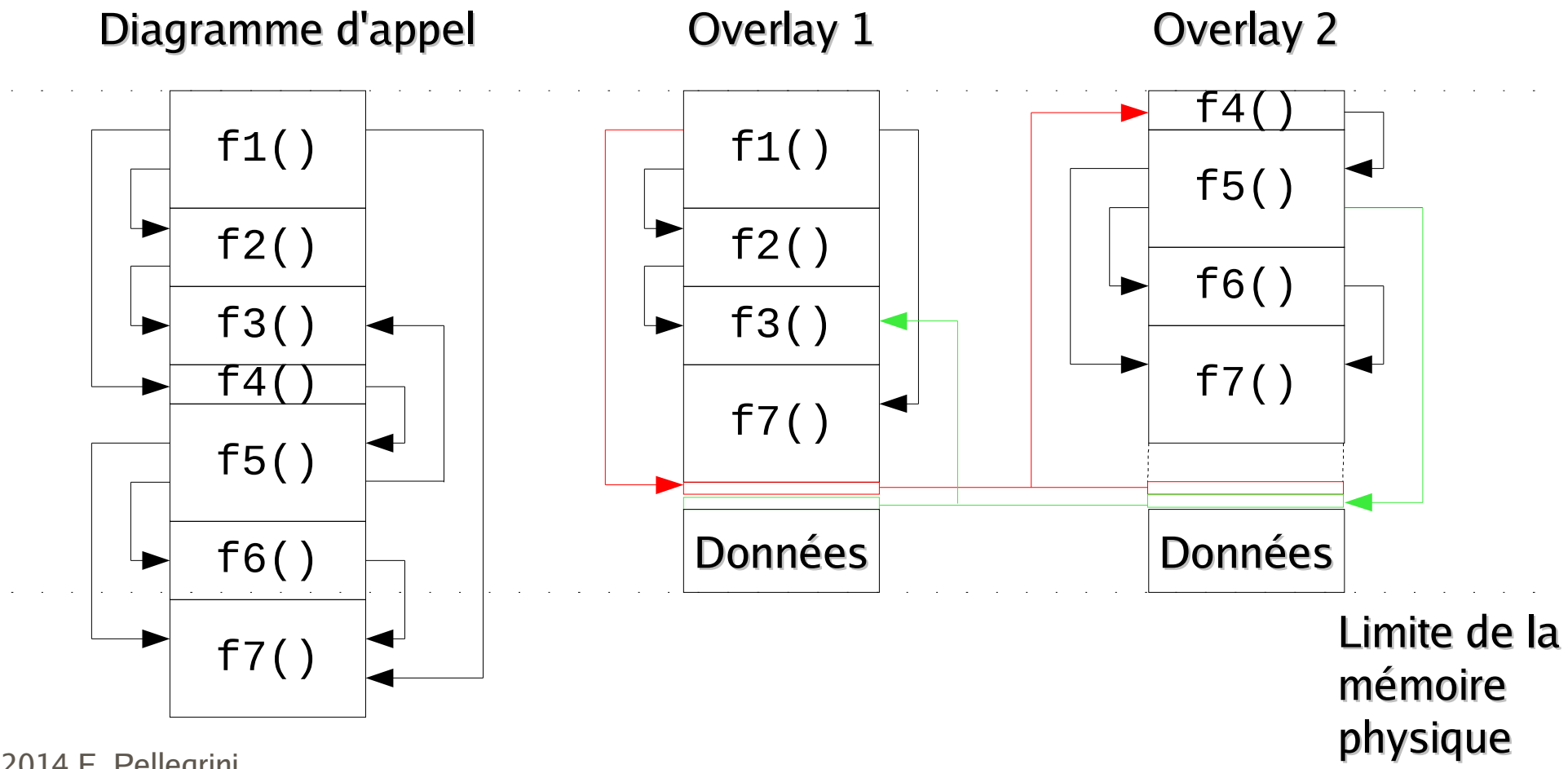
- Au début de l'informatique, les mémoires (vives et de masse) étaient de très faible capacité et très chères
  - Les plus gros programmes ne pouvaient tenir en mémoire, avec leurs données
- Deux solutions possibles :
  - Avoir plusieurs petits programmes
    - Mais surcoût de sauvegarde et de rechargement des données à chaque changement de programme
  - Remplacer à la volée le code en laissant les données en place en mémoire : système des overlays

# Overlays (1)

- Le programmeur, lors de l'édition de liens d'un programme, répartit les différents modules qui le composent en sous-ensembles exécutables les plus indépendants possibles : les overlays
  - Cas des passes d'un compilateur, par exemple
- Lorsque l'overlay en cours d'exécution a besoin d'appeler une fonction qu'il ne contient pas, une routine de service charge à sa place l'overlay qui la contient
  - Possibilité de dupliquer les fonctions les plus couramment utilisées

# Overlays (2)

- Exemple d'édition de liens d'un programme, conduisant à l'obtension de deux overlays



# Overlays (3)

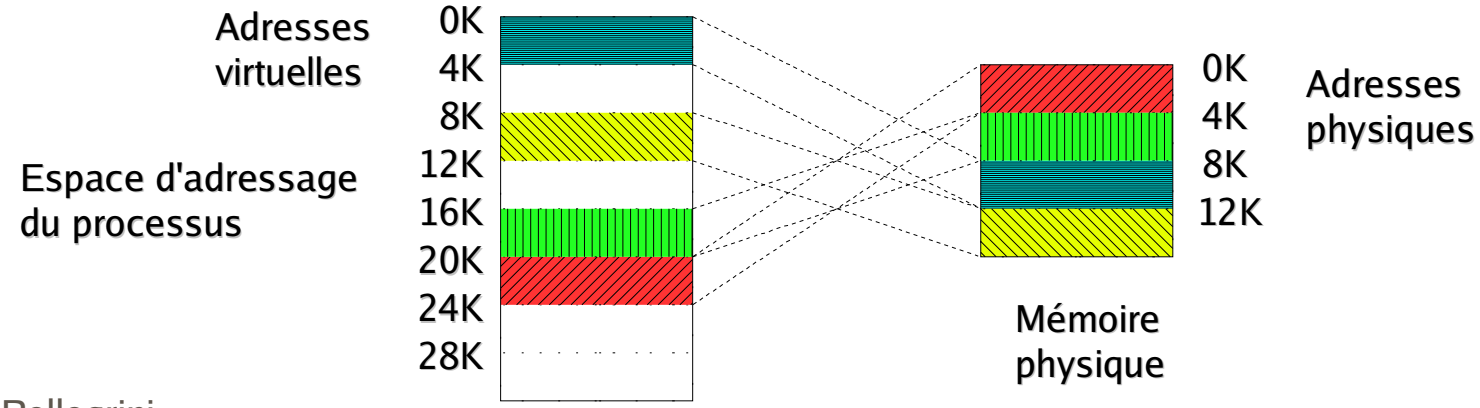
- La construction des overlays est fortement liée à la taille de la mémoire de la machine cible
  - Si moins de mémoire que la taille du plus grand overlay, le programme ne peut s'exécuter
  - Si taille plus grande, le programmeur aurait pu mieux utiliser la mémoire (moins de changements d'overlays)
- Problèmes :
  - Granularité des overlays trop élevée
  - Découpage statique et non pas dynamique

# Overlays (4)

- Pour rendre le système plus efficace, il faudrait mieux utiliser l'espace mémoire et réduire les coûts de chargement
  - Pas de groupage statique figé à l'édition de liens
    - Décharge l'utilisateur de la tâche de groupage
  - Manipulation de blocs de code de plus petite taille
    - Coûts de chargement et de déchargement limités
  - Chargement dynamique des portions de code
    - Position en mémoire pas connue à l'avance
    - Traduction à la volée entre adresses virtuelles et physiques
  - Gérer la zone de données et pas seulement de code

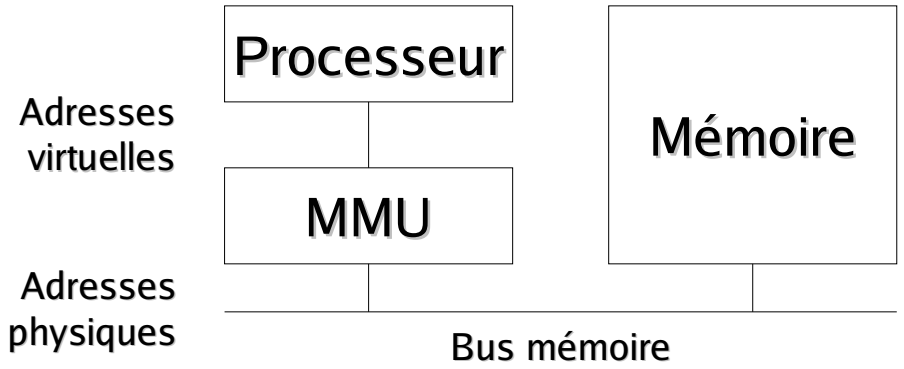
# Mémoire virtuelle (1)

- Système permettant de découpler l'espace d'adressage et la mémoire physique
- Découpage en pages de l'espace d'adressage virtuel du processus et de la mémoire physique
  - Taille variant de 512 octets à 64 Ko, en général 4 Ko
- Placement (« mapping ») des pages en mémoire



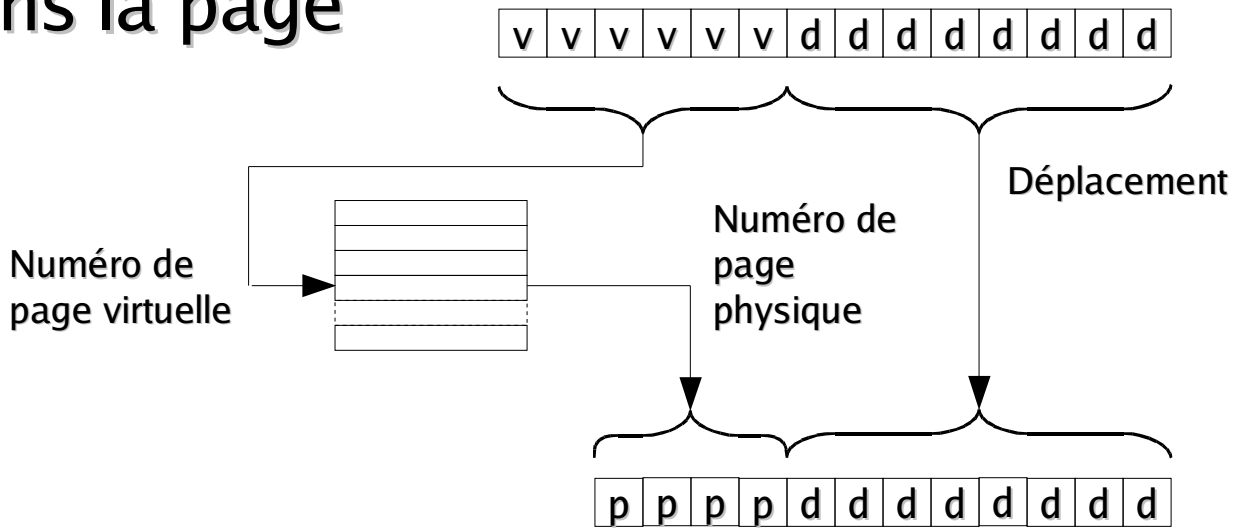
# Mémoire virtuelle (2)

- Un dispositif matériel appelé MMU (« *Memory Management Unit* ») fait la conversion à la volée entre adresses virtuelles et adresses physiques
  - Convertit chaque adresse virtuelle émise par le processeur en adresse physique



# Mémoire virtuelle (3)

- La conversion des adresses physiques en adresses virtuelles s'effectue au moyen de tables de pages
  - La partie haute de l'adresse virtuelle sert d'index
  - La partie basse sert de déplacement (« *offset* ») dans la page



# Mémoire virtuelle (4)

- Les pages du processus sont chargées en mémoire à la demande (« *demand paging* »)
  - Si on émet une adresse virtuelle correspondant à une page présente en mémoire physique, la MMU fait silencieusement le transcodage
  - Si la page virtuelle n'est pas présente en mémoire, la MMU génère une interruption de type « défaut de page » (« *page fault* ») à destination du processeur
  - Le processeur traite l'interruption en chargeant la page depuis le disque, avant de reprendre l'exécution
    - Politique de remplacement LRU (« *Least Recently Used* »)

# Mémoire virtuelle (5)

- La taille des pages influe sur la performance
- Avec des pages trop grandes :
  - Fragmentation interne plus importante
  - Les accès aléatoires génèrent plus de trafic mémoire
    - Matrice stockée par lignes et accédée par colonnes
- Avec des pages trop petites :
  - Les chargements à partir du disque sont moins efficaces (pénalité de latence à chaque chargement)
  - Taille de la table des pages trop importante

# Mémoire virtuelle (6)

- Certains problèmes persistent
  - Droits d'accès : gérer l'accès par pages est coûteux
    - Droits valables pour l'ensemble de zones mémoire fonctionnellement distinctes telles que : code, données initialisées constantes, données non constantes initialisées ou pas, pile, mémoire partagée, code du système, données du système, etc.
  - Granularité d'accès : les accès illégaux au delà des zones allouées d'une page ne sont pas détectés
    - Espace libre de la dernière page courante de la zone de données
    - Espace situé au delà du sommet de pile

# Mémoire virtuelle (7)

- Continuité de l'adressage : lors des chargements dynamiques de bibliothèques, on doit placer le code et les données dans des zones de nature différente
  - Réserver à l'avance des plages libres dans l'espace d'adressage pose des problèmes :
    - Fragmentation externe si ces zones ne sont pas remplies
    - Collisions entre zones si une zone atteint sa taille maximale pré-allouée
- Nécessité d'un mécanisme de gestion de zones disjoint de la pagination

# Segmentation (1)

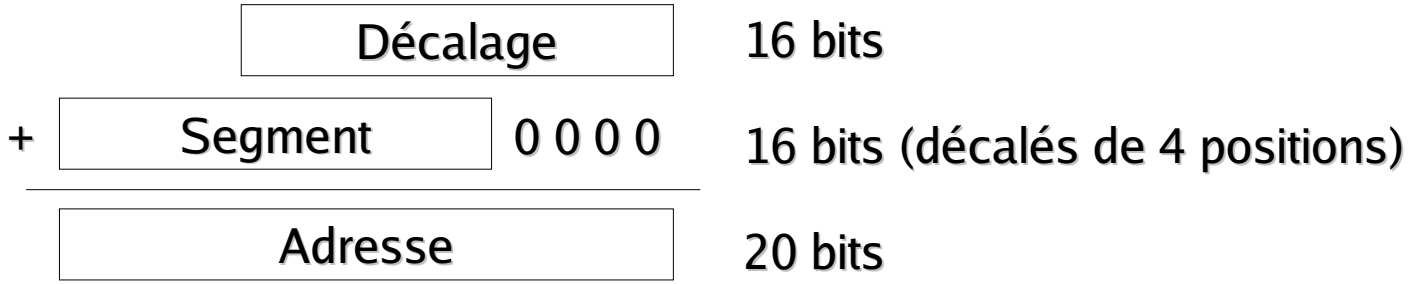
- La segmentation consiste à avoir autant d'espaces d'adressage que de zones mémoire d'usages différents
  - Adresses partent à partir de 0 pour chaque segment

# Segmentation (2)

- Des descripteurs de segments sont associés à chaque segment
  - Taille à l'octet près (pas de problème de granularité)
  - Droits d'accès
  - Référence à la racine d'une table de pages privée ou bien adresse de début du segment dans la mémoire virtuelle (permet facilement les déplacements logiques de segments dans l'espace d'adressage)

# Gestion mémoire du Pentium II (1)

- Sur le 8086, une adresse était obtenue par ajout d'une valeur de segment, sur 16 bits, multipliée par 16, avec une valeur de décalage sur 16 bits, pour former une adresse sur 20 bits

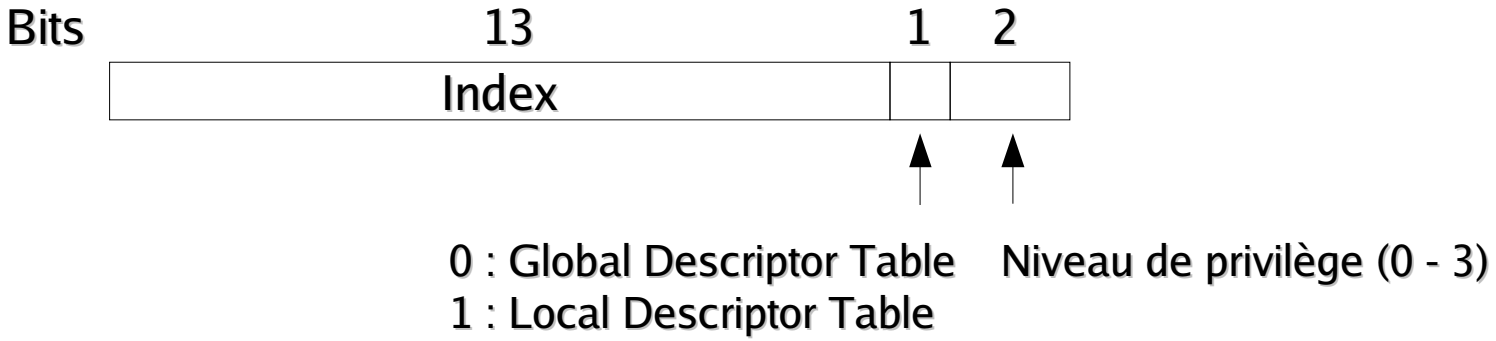


# Gestion mémoire du Pentium II (2)

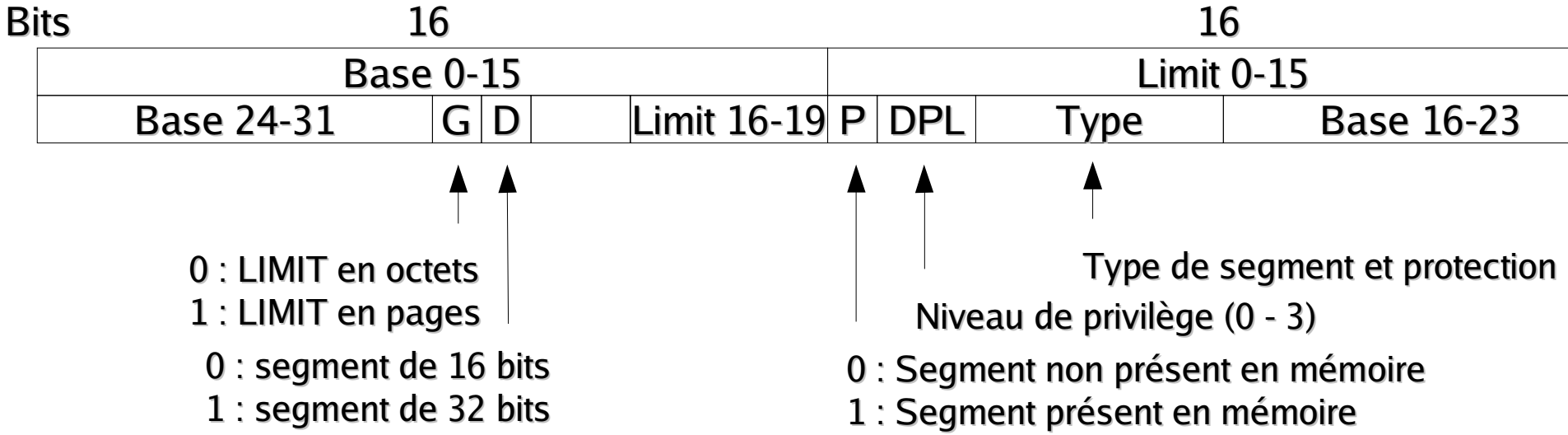
- À partir du 80286, on a un vrai adressage segmenté
- Les registres de segments deviennent des sélecteurs de segments indexant des descripteurs de segments dans des tables
- Deux tables sont maintenues par la MMU
  - GDT : table globale contenant les descripteurs communs à tous les processus (segments du système)
  - LDT : table locale à chaque processus

# Gestion mémoire du Pentium II (3)

- Sélecteur de segment en mode protégé

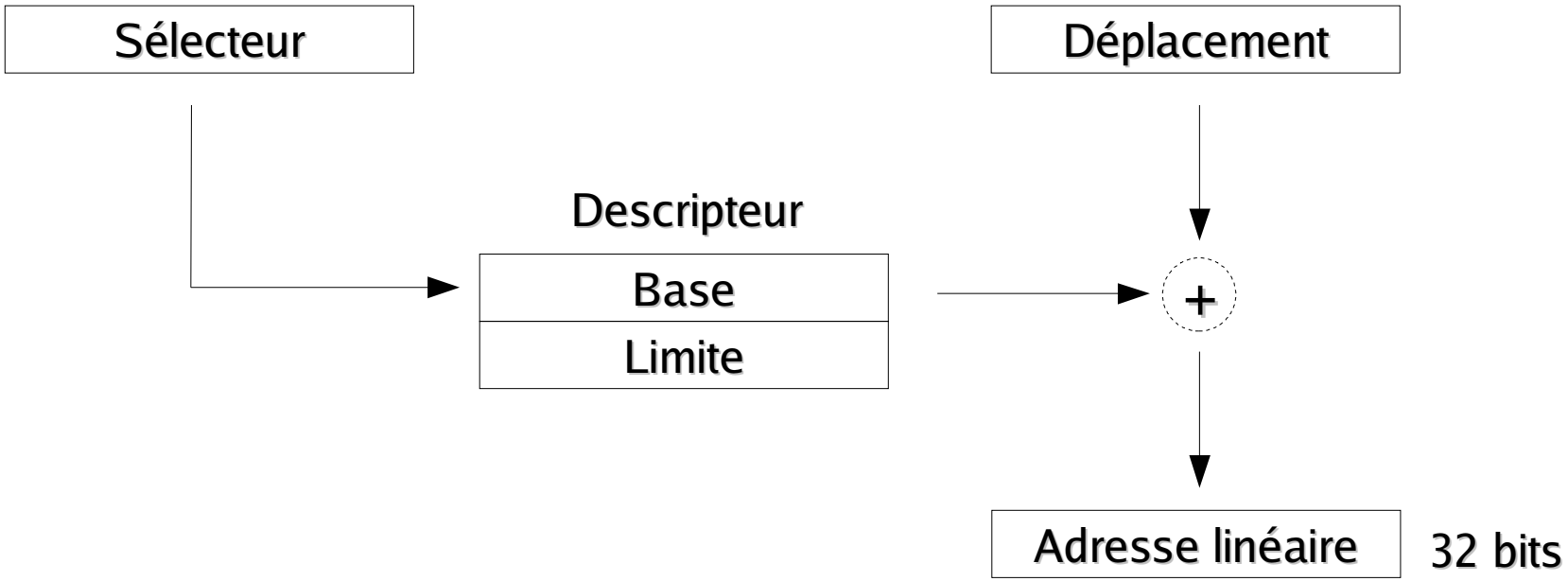



- Descripteur de segment en mode protégé



# Gestion mémoire du Pentium II (4)

- L'adresse virtuelle linéaire est calculée en ajoutant la valeur du déplacement à l'adresse de base contenue dans le descripteur



# Fichiers (1)

- La gestion des entrées-sorties est un autre apport majeur de la couche système
- Mise à disposition d'une abstraction de fichier sous forme de suite ordonnée d'octets
- Possibilités d'accès en lecture, en lecture/écriture, ou seulement en écriture suivant le type de périphérique modélisé
  - Sécurité
  - Confidentialité
  - Intégrité

# Fichiers (2)

- Les fichiers existent en dehors de l'espace d'adressage des programmes
  - Nécessité de leur nommage
- Appel système d'ouverture de fichier permettant d'ouvrir un « canal » de communication entre le fichier et l'espace d'adressage du programme
  - Récupération d'un descripteur du fichier ouvert
- Utilisation de ce descripteur pour échanger des blocs de données entre le fichier et la mémoire

# Fichiers (3)

- Les fichiers sont administrés au sein de systèmes de fichiers
  - Systèmes logiques ou physiques
    - Possibilité de partitionner un disque unique en plusieurs systèmes de fichiers
    - Possibilité de regrouper plusieurs disques au sein d'un système de fichiers logique unique
  - Fournissent des modèles d'organisation interne tels que répertoires, structures arborescentes de répertoires, clés, etc.

# Langage d'assemblage (1)

- La couche langage d'assemblage diffère des précédentes en ce qu'elle est implémentée par traduction et non par interprétation
  - L'existence de moyens de stockage non volatils fournis par la couche système d'exploitation permet de rentabiliser le coût de traduction
- Deux sortes de traducteurs :
  - Assembleur : d'un langage d'assemblage vers un langage machine
  - Compilateur : d'un langage de haut niveau vers un langage d'assemblage ou un langage machine

# Langage d'assemblage (2)

- Un langage d'assemblage offre une représentation symbolique d'un langage machine
  - Utilisation de noms symboliques pour représenter les instructions et leurs adresses
- L'assembleur convertit cette représentation symbolique en langage machine proprement dit
  - Possibilité d'accès aux fonctionnalités offertes par la couche système d'exploitation

# Langage d'assemblage (3)

- Avantages du langage d'assemblage par rapport au langage machine
  - Facilité de lecture et de codage
    - Les noms symboliques des instructions (« codes mnémoniques ») sont plus simples à utiliser que les valeurs binaires dans lesquelles elles seront converties
  - Facilité de modification
    - Pas besoin de recalculer à la main les adresses des destinations de branchements ou des données lorsqu'on modifie le programme
- Aussi offerts par les langages de haut niveau

# Langage d'assemblage (4)

- Avantages du langage d'assemblage par rapport aux langages de haut niveau
  - Accès à la machine
    - Le langage d'assemblage permet d'accéder à l'ensemble des fonctionnalités et des instructions disponibles sur la machine cible au niveau de la couche système d'exploitation
  - Performance
    - Un programme en langage d'assemblage est souvent plus compact et plus efficace qu'un programme en langage de haut niveau

# Langage d'assemblage (5)

- Inconvénients du langage d'assemblage par rapport aux langages de haut niveau
  - Coût de développement
    - Longueur du code source
    - Difficulté de programmation
      - Nombre de mnémoniques
      - Complexité des modes d'adressage
  - Difficulté de maintenabilité
    - Variation des performances relatives des instructions entre plusieurs processeurs de la même famille
  - Non portabilité entre familles de processeurs

# Langage d'assemblage (6)

- Les utilisations principales des langages d'assemblage sont donc les applications critiques en termes de taille et/ou de performance et/ou d'accès au matériel
  - Applications embarquées et enfouies
  - Pilotes de périphériques
  - Modules de changement de contexte entre processus

# Instructions des langages d'assemblage

- Les instructions des langages d'assemblage sont habituellement constituées de trois champs :
  - Un champ d'étiquette
  - Un champ d'instruction
  - Un champ de commentaires (commentaires ligne)

# Champ étiquette

- Champ optionnel
- Associe un nom symbolique à l'adresse à laquelle il se trouve
  - Destination de branchement ou d'appel de sous-routine pour les étiquettes situées en zone de code
  - Adresses de variables ou constantes mémoire pour les étiquettes situées en zone de données

# Champ instruction (1)

- Contient soit une instruction soit une directive destinée à l'assembleur lui-même
- Une instruction est constituée :
  - Du code mnémonique du type de l'opération
  - De représentations symboliques des opérandes
    - Noms symboliques de registres
    - Représentations de constantes numériques sous différents formats
      - Décimal, binaire, octal, hexadécimal
    - Expressions parenthésées ou à base de crochets permettant d'exprimer les différents modes d'adressage

# Champ instruction (2)

- Les types et longueurs des opérandes manipulés peuvent être spécifiés
  - Par les noms des registres utilisés
    - Cas de l'architecture x86 : `AL` (8 bits), `AX` (16 bits), `EAX` (32 bits)
  - Par un code mnémonique différent
    - Cas de l'architecture SPARC : `LDSB`, `LDSH`, `LDSW` pour charger respectivement des octets, des demi-mots ou des mots signés
  - Par un suffixe accolé au code mnémonique
    - Cas de l'architecture M68030 : suffixes `.B`, `.W` ou `.L` pour manipuler des octets, des mots, ou des mots longs

# Champ commentaire

- Ne sert qu'au programmeur et pas à l'assembleur
- Permet au programmeur de laisser des explications utiles sur ce que fait son programme, ligne par ligne
- Un programme en assembleur sans commentaires est affreusement difficile à lire
  - Difficulté de la rétro-ingéniérie à partir du code objet d'un programme

# Directives

- Influent sur le comportement de l'assembleur
- Ont de multiples fonctions
  - Définition de constantes symboliques
    - Analogue au `#define` du pré-processeur C
    - Cas de l'architecture x86 : `etiquette EQU valeur`
  - Définition de segments pour le code binaire produits par l'assembleur (code ou données)
  - Alignement par mots du code et des données
  - Réservation et initialisation d'espace mémoire
  - Macro-instructions

# Macro-instructions

- Une macro-définition permet d'associer un nom à un fragment de code en langage d'assemblage
  - Analogue à la directive `#define` du pré-processeur C
  - Possibilité de définir des paramètres formels
  - Cas de l'architecture x86 : directive `MACRO` ... `ENDM`
- Après qu'une macro a été définie, on peut l'utiliser à la place du fragment de texte
  - Remplacement textuel avant l'assemblage

# Assemblage (1)

- Problème des références descendantes
  - Comment savoir à quelle adresse se fera un branchement si on ne connait pas encore le nombre et la taille des instructions intermédiaires
  - Nécessité de connaître l'ensemble du code source avant de générer le code en langage machine
- L'assemblage s'effectue toujours en deux passes
  - Première passe : analyse du code source
  - Deuxième passe : génération du code en langage machine

# Assemblage (2)

- Lors de la première passe, l'assembleur :
  - Vérifie la syntaxe du code assembleur
  - Collecte les définitions des macros et stocke le texte du code assembleur associé
  - Effectue l'expansion des noms de macros trouvés
  - Détermine les tailles mémoire nécessaires au stockage de chacune des instructions et directives rencontrées
  - Collecte dans une table des symboles les définitions des étiquettes et de leur valeur, déterminée à partir de la somme des tailles des instructions précédentes

# Assemblage (3)

- Lors de la deuxième passe, l'assembleur :
  - Selon l'implémentation, relit le code assembleur ou le flot d'instructions résultant de l'expansion des macros réalisée lors de la première passe
  - Vérifie si les étiquettes éventuellement référencées par les instructions ont toutes été trouvées lors de la première passe
  - Génère le code en langage machine pour les instructions, en y intégrant les adresses mémoire des étiquettes qu'elles utilisent
  - Ajoute au fichier objet la table des symboles connus

# Édition de liens

- Produit un exécutable à partir de fichiers objets
  - Ajout d'un en-tête de démarrage au début du fichier
- La plupart des programmes sont constitués de plus d'un fichier source
  - Problème des références croisées entre fichiers
    - Références définies dans un fichier et utilisées dans d'autres
  - Problème de relocation de code
    - Les adresses définitives des données et des fonctions dans le programme exécutable dépendent de la façon dont les différents modules sont agencés en mémoire

# Références croisées

- Les symboles externes doivent être déclarés dans chacun des modules dans lesquels ils sont utilisés
  - Insérés dans la table des symboles en tant que références déclarées mais pas définies
  - L'assembleur ajoute les adresses des instructions utilisant ces références à la table des instructions non résolues
- Ne doivent être définis que dans un seul module
  - Message d'erreur de l'éditeur de liens si définitions multiples ou pas de définition

# Relocation de code

- Lors de l'assemblage, les adresses des instructions de branchements et d'appels absolus sont stockées dans une table spécifique
  - Table de relocation incorporée au fichier objet
  - Les branchements relatifs locaux ne sont pas concernés
- Lors de l'édition de liens, les adresses absolues de toutes les références sont calculées et les déplacements absolus sont intégrés aux codes binaires des instructions

# Fichier objet

- Un fichier objet contient :
  - Le code en langage machine résultant de l'assemblage ou de la compilation du fichier en code source correspondant
  - La table des symboles
    - Symboles définis localement et déclarés comme visibles de l'extérieur du module
    - Symboles non locaux déclarés et utilisés dans le module
  - La table de relocation
    - Adresses des instructions utilisant des adresses absolues (locales et non-locales)

# Bibliothèques (1)

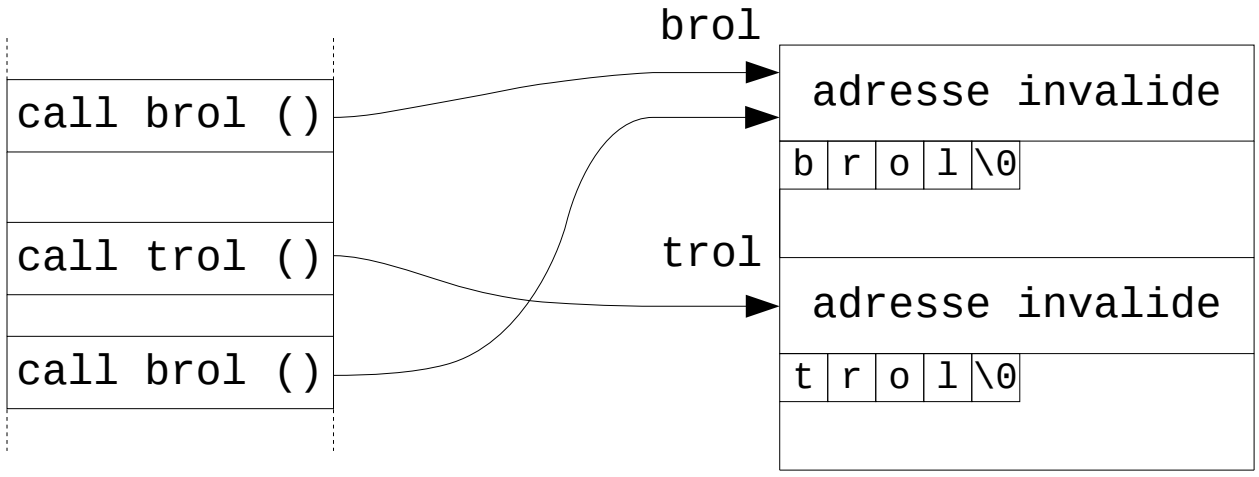
- Lorsqu'on fournit un ensemble de fonctions rendant un ensemble cohérent de services, il serait préférable de grouper les fichiers objets associés sous la forme d'un unique fichier
  - Facilite la manipulation et la mise à jour
- Ce service est rendu par les fichiers de bibliothèque
  - Fichiers servant à archiver des fichiers objet
  - Utilisables par l'éditeur de liens

# Bibliothèques (2)

- Deux types de bibliothèques
  - Bibliothèques statiques
    - Format en « `lib*.a` » (Unix) ou « `*.lib` » (DOS)
    - Liées à l'exécutable lors de la compilation
      - Augmentent (parfois grandement) la taille des exécutables
      - On n'a plus besoin que de l'exécutable proprement dit
  - Bibliothèques dynamiques
    - Format en « `lib*.so` » (Unix, « *shared object* ») ou « `*.dll` » (Windows, « *dynamic loadable library* »)
    - Liées à l'exécutable lors de l'exécution
      - Permettent la mise à jour indépendante des bibliothèques
      - Problème si pas présentes (variable « `LD_LIBRARY_PATH` »)

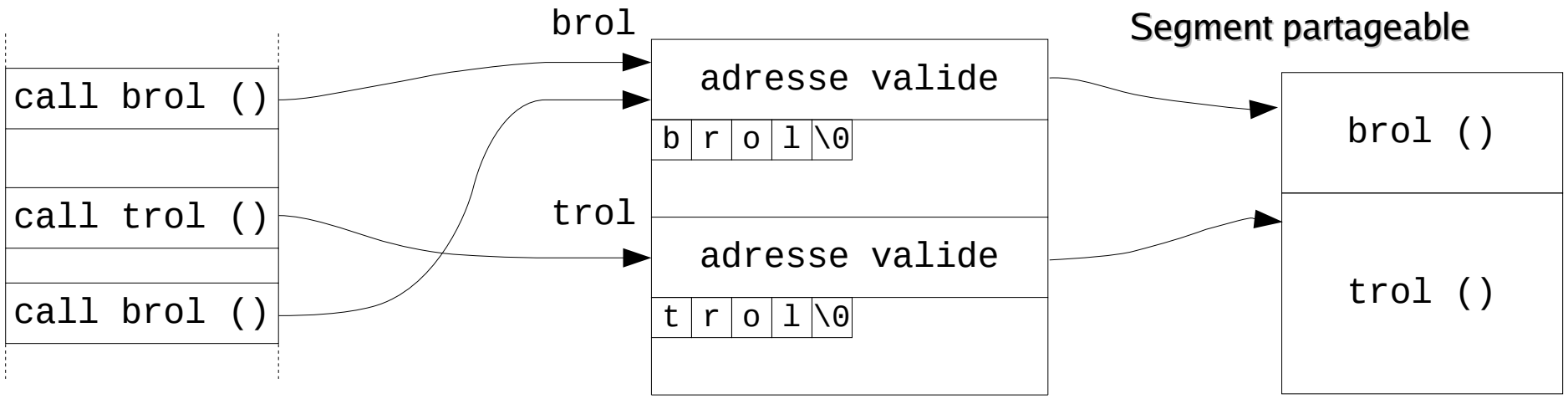
# Édition dynamique de liens (1)

- Lorsque l'éditeur de liens trouve la définition d'un symbole dans une bibliothèque dynamique
  - Il ajoute au programme un fichier objet spécial, issu de la bibliothèque, appelé « module d'importation »
  - Le module contient les adresses (invalides) des branchements ainsi que le nom de la fonction



# Édition dynamique de liens (2)

- Lors d'un appel à une fonction dynamique, il y a erreur de segmentation et traitement
  - Le contenu de la bibliothèque dynamique est chargé dans un nouveau segment partageable si elle n'était pas déjà présente dans le système
  - Les adresses modifiées pointent dans ce segment



# Édition dynamique de liens (3)

- Dans d'autres systèmes, il n'y a pas d'indirections sur les adresses des routines
  - Les adresses des instructions de branchements sont stockées dans la table de relocation et associées au nom de la bibliothèque dynamique à charger
  - Lors du chargement du programme, toutes les bibliothèques dynamiques nécessaires sont chargées en mémoire dans des segments partagés si elles ne l'étaient pas déjà, et les adresses des branchements sont modifiées en conséquence

# Édition dynamique de liens (4)

- Certains systèmes permettent aussi le chargement dynamique de bibliothèques à la demande lors de l'exécution du programme
  - Analogue au chargement dynamique des modules dans le noyau
  - Demande explicite par le programmeur
    - Cas des systèmes de type Unix : `dlopen()`, `dlsym()`, `dlclose()`
  - Respecte le paradigme de l'éditeur de liens
    - Pas de symbole non résolu à l'édition de liens
    - Manipulation des symboles dynamiques par pointeurs

# Bus (1)

- Un bus est un chemin électrique commun reliant plusieurs équipements
- Défini par un protocole, qui spécifie ses caractéristiques mécaniques et électriques
- Exemples :
  - PC bus : bus interne des PC/XT conçu par IBM
  - AGP : bus pour carte graphique
  - SCSI : dialogue avec des périphériques
  - FireWire : connexion entre équipements grand public ...

# Bus (2)

- Caractéristiques de quelques bus de type PC

| Date | Type de bus | Largeur | Fréquence | Débit max. |
|---|---|---|---|---|
| 1981 | ISA (PC/XT) | 8 | 4,77 | 4,77 |
| 1985 | ISA (PC/AT) | 16 | 8,33 | 16,33 |
| 1987 | MCA (PS/2) | 32 | 10 | 40 |
| 1988 | EISA | 32 | 8,33 | 33 |
| 1991 | VESA | 32 | 33 | 133 |
| 1994 | | 32 | 66 | 264 |
| | | 64 | 66 | 528 |
| 1993 | PCI | 32 | 33 | 132 |
| 1994 | | 64 | 33 | 264 |
| 1995 | | 64 | 66 | 528 |
| 1999 | | 64 | 133 | 1024 |

# Bus (3)

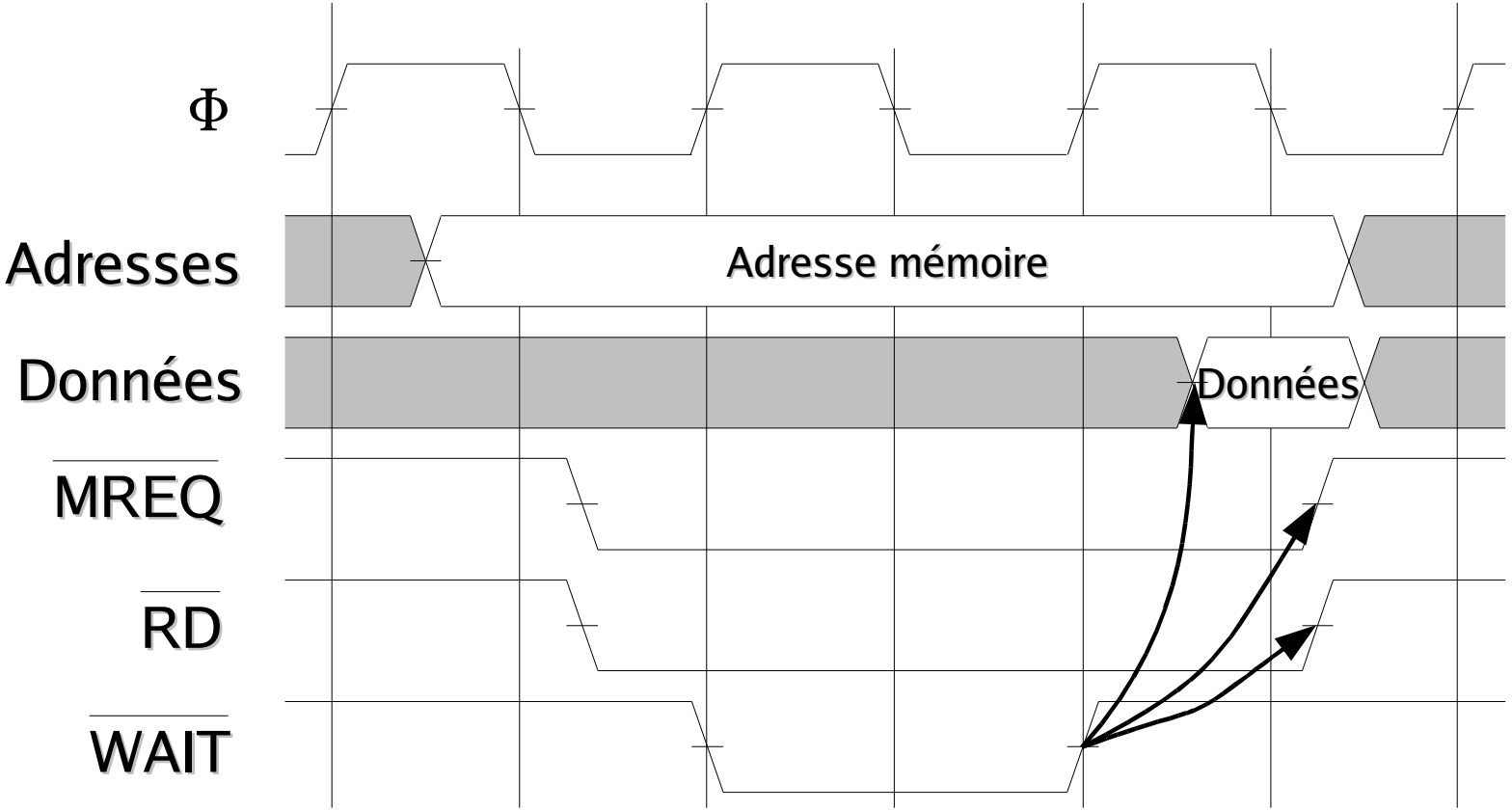
- Les équipements attachés au bus peuvent être :
  - Actifs, s'ils peuvent initier des transferts de données par eux-mêmes (maîtres)
  - Passifs, s'ils ne font qu'attendre des requêtes et y répondre de façon synchrone (esclaves)
- Certains équipements peuvent avoir les deux rôles à la fois
  - Un contrôleur de disque reçoit passivement les requêtes en provenance du processeur, mais pilote activement le transfert des blocs de données vers la mémoire une fois qu'ils ont été lus sur le disque

# Bus (4)

- Les lignes des bus sont divisées fonctionnellement en trois catégories
  - Lignes de contrôle
  - Lignes d'adresses
  - Lignes de données (éventuellement multiplexées avec les lignes d'adresses)
- Les bus peuvent être :
  - Synchrones : pilotés par une horloge maître
  - Asynchrones : nécessité de négociation entre le maître et l'esclave pour se synchroniser

# Bus (5)

- Chronogramme d'une lecture mémoire sur un bus synchrone avec un cycle d'attente




Cycle d'attente

# Bus USB (1)

- Bus destiné aux périphériques externes
- Caractéristiques souhaitées à sa conception :
  - Pas de configuration matérielle
  - Connexion sans avoir besoin d'ouvrir l'ordinateur
  - Un seul type de câble (plus vrai)
  - Alimentation fournie par l'unité centrale
  - 127 périphériques raccordables
  - Support des équipements temps-réels
  - Branchement et débranchement à chaud

# Bus USB (2)

- Topologie en arbre
  - Un contrôleur principal (« *root hub* ») connecté au bus maître de l'ordinateur (bus PCI)
  - Des contrôleurs esclaves pour chaque équipement
  - Aucune communication directe entre équipements
- Possibilité de définir des sous-canaux de communication sur un équipement donné
- Caractéristiques du bus USB

| Date | Type de bus | Débit max. |
|---|---|---|
| 1996 | USB 1.0 | 1,5 |
| 2001 | USB 2.0 | 60 |

# Contrôleur de périphérique (1)

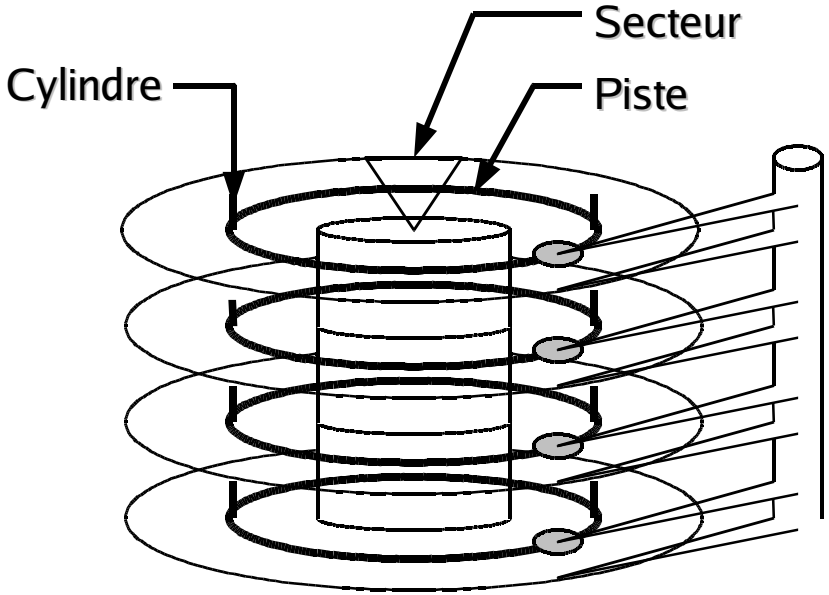
- Tout périphérique d'entrée/sortie est constitué de deux parties
  - Un sous-système contenant la plupart de l'électronique de commande, appelé le contrôleur de périphérique
  - Le matériel du périphérique proprement dit

# Contrôleur de périphérique (2)

- Lors des échanges entre l'unité centrale et les périphériques, le processeur doit intervenir pour initialiser l'opération, mais il n'est pas nécessaire qu'il s'occupe des transferts mémoire
- Certains contrôleurs sont capables d'accéder la mémoire centrale sans que l'unité centrale soit mobilisée, faisant ainsi des DMA (« Direct Memory Access »)

# Disque dur (1)

- Constitués d'un empilement de disques rigides tournant sur le même axe et revêtus d'une couche magnétique
  - Diamètre d'une dizaine de centimètres à moins de trois centimètres
  - Surfaces survolées par des têtes de lecture/écriture magnétique montées sur un bras mobile



# Disque dur (2)

- Une piste est la zone couverte par une tête de lecture en un tour de disque lorsque le bras reste dans une position donnée
- Un cylindre est la zone couverte sur tous les disques par l'ensemble des têtes de lecture en un tour de disque lorsque le bras reste dans une position donnée
- Un secteur est une portion de piste représentant une fraction de la surface angulaire totale

# Disque dur (3)

- Les secteurs sont l'unité de stockage élémentaire sur le disque
  - On lit et écrit par secteurs
- Un secteur est constitué :
  - D'un préambule, servant à synchroniser la tête avant d'entamer l'opération de lecture/écriture
  - De la zone d'informations, en général de 512 octets
  - D'une zone de contrôle servant au contrôle et à la correction des données enregistrées
- Il existe un espace libre entre chaque secteur

# Disque dur (4)

- Les plateaux sont en alliage d'aluminium ou en céramique pour les plus récents
- Ils sont recouverts de particules magnétiques dont le champ peut prendre deux orientations
  - Entre 500 et 1000 particules utilisées pour chaque bit
- Deux orientations possibles des particules :
  - Longitudinale : plus faible densité (20 Gbit/cm$^2$), et sensibilité plus importante au super-paramagnétisme
  - Perpendiculaire : plus grande densité (40 Gbit/cm$^2$), mais plus difficile à fabriquer

# Disque dur (5)

- Trois types de têtes
  - Inductive : la même tête sert à magnétiser la piste et à détecter les variations d'orientation magnétique, qui génèrent un courant induit
  - Magnéto-résistive : la tête d'écriture est inductive, mais la tête de lecture utilise l'effet magnéto-résistif : les variations du champ magnétique causent des variations de résistivité de la tête. Permet un stockage bien plus dense de l'information.
  - Magnéto-résistive géante : l'utilisation de matériaux en couches minces multiplie leurs propriétés magnéto-résistives (d'un facteur supérieur à 100)

# Disque dur (6)

- La distance radiale entre la ou les têtes de lecture et le centre du disque sert à identifier la piste courante
- La largeur d'une piste dépend de la taille des têtes et de la précision de positionnement
  - Largeur d'une piste de 2 à 10 microns
  - Entre 1000 et 4000 pistes par centimètre, en tenant compte des espaces nécessaires entre pistes
  - Densité d'écriture sur la piste de 100 000 à 200 000 bits par centimètre linéaire

# Disque dur (7)

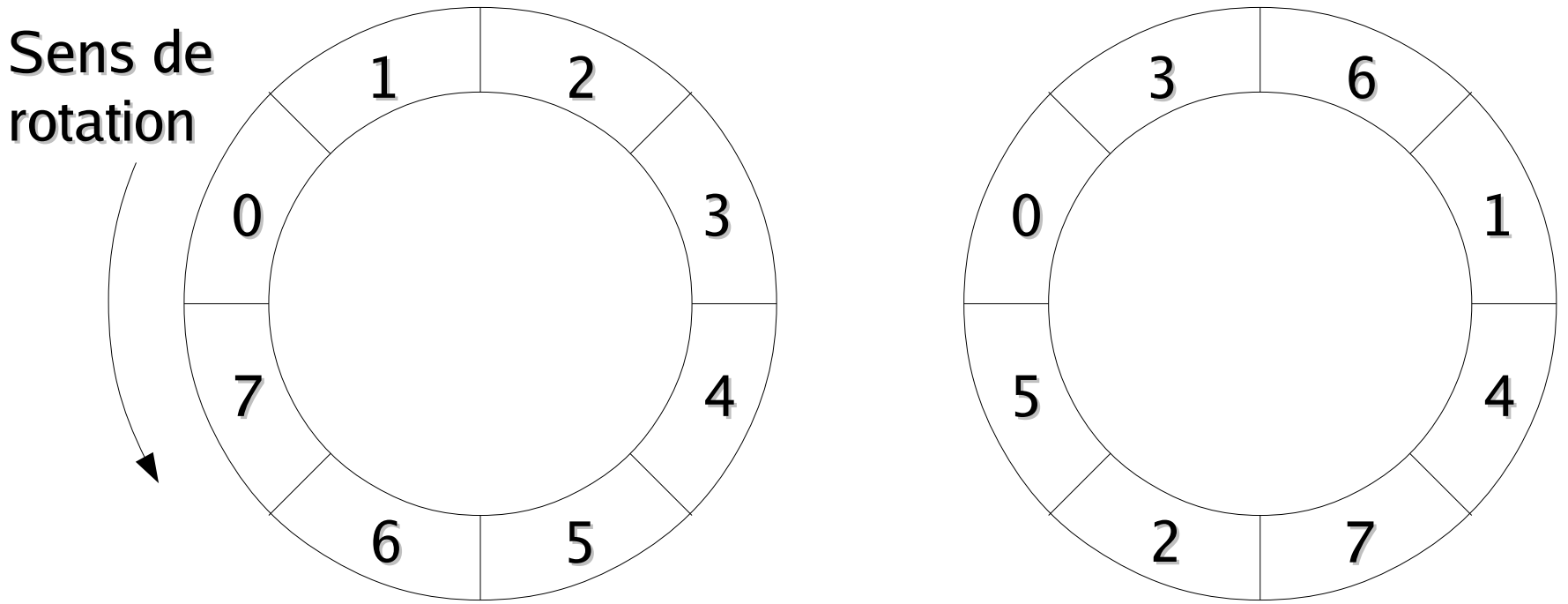
- La longueur des pistes n'est pas constante
  - Les pistes sont plus longues à la périphérie qu'au centre du disque ($c = 2 \pi r$)
- Deux méthodes possibles pour gérer cela :
  - Ne rien faire, et densité de stockage d'informations plus importante près de l'axe de rotation
  - Avoir un nombre de secteurs par piste variable
    - Surface du disque découpée en groupes de cylindres
    - Moins de secteurs par piste dans les groupes situés le plus près du centre, plus dans ceux de la périphérie
    - Augmente la capacité de stockage du disque

# Disque dur (8)

- Les performances d'un disque dépendent :
  - Du temps de positionnement du bras entre deux cylindres quelconques
    - Entre 5 et 15 millisecondes en moyenne
    - Environ 1 milliseconde entre deux cylindres consécutifs
  - Du temps d'attente du passage du bon secteur sous la tête de lecture
    - Dépend de la vitesse de rotation du disque
      - 3600, 5400, 7200 ou 10800 tours par minute
      - Temps moyen de latence compris entre 3 et 8 millisecondes
  - Du débit de transfert de l'information
    - De 5 à 40 Mo/s

# Disque dur (9)

- Afin de diminuer le temps de latence rotationnelle lors de la lecture de gros fichiers, les secteurs sont entrelacés sur les pistes (« *interleaving* »)



# CD-ROM (1)

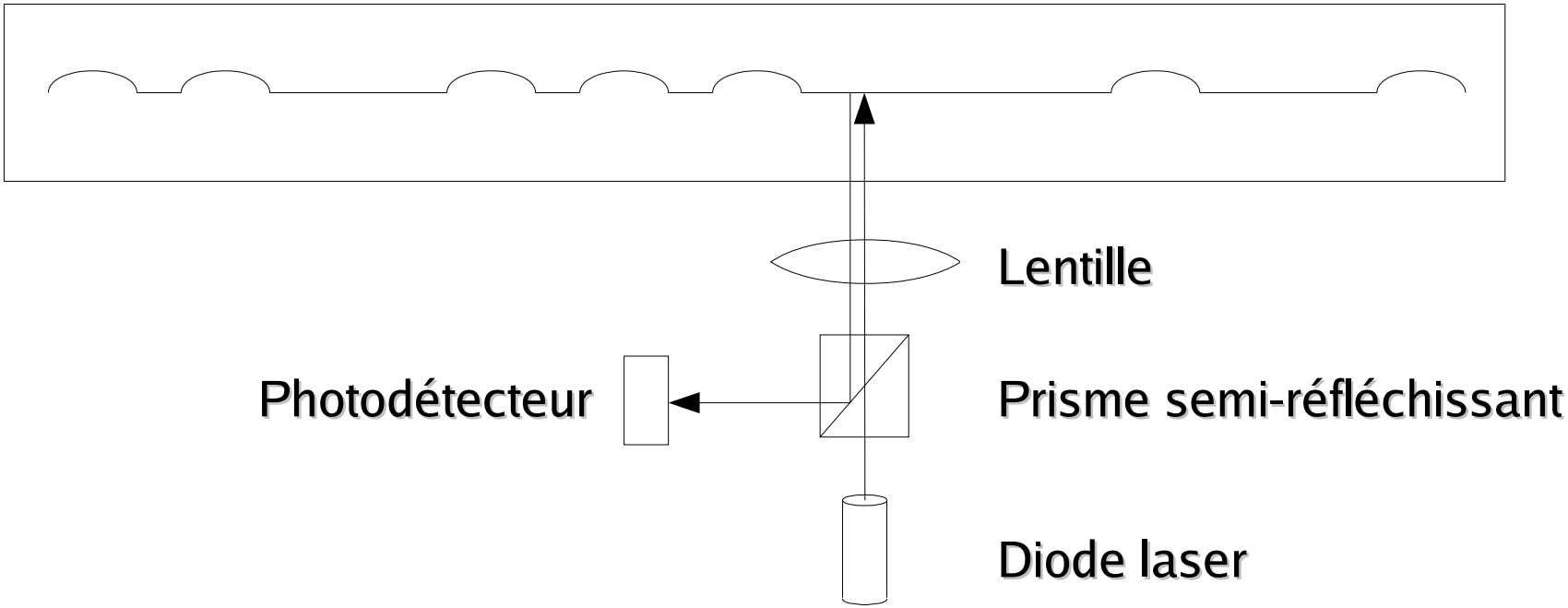
- Création du CD audio par Philips et Sony au début des années 1980
  - Héritier du disque vidéo grand format (30 cm) développé par Philips à la fin des années 1970
  - Spécifications techniques normalisées par l'ISO
    - Diamètre de 12 cm
    - Épaisseur de 1,2 mm
    - Diamètre du trou central de 1,5 cm
    - ...

# CD-ROM (2)

- Un CD est constitué d'une mince pellicule d'aluminium prise entre deux couches de matériau plastique transparent (polycarbonate)
- La couche d'aluminium est constituée d'un ensemble de zone plates (« *lands* ») et de micro-cuvettes (« *tips* ») servant à coder l'information
  - Taille des trous de l'ordre de quelques microns

# CD-ROM (3)

- La lecture se fait au moyen d'un rayon laser
  - Les zones plates réflechissent le rayon
  - Les cuvettes dispersent le rayon



# CD-ROM (4)

- Les microcuvettes et les zones plates sont inscrites sur une spirale continue
  - Pas des cercles concentriques comme pour les disques magnétiques
- La spirale part du centre du disque vers le bord
  - 22188 révolutions
  - Environ 600 spires par mm
  - 5,6 km de long en déroulé

# CD-ROM (5)

- Afin d'avoir un flot de données continu et constant en écoute audio, la vitesse de rotation du disque s'adapte en fonction de la distance radiale du bras portant la lentille
  - Vitesse linéaire de 120 cm/s
  - Vitesse angulaire de :
    - 530 tours/mn au centre du disque
    - 200 tours/mn au bord du disque
- Pour les CD-ROM, on travaille à des vitesses multiples de la vitesse standard

# CD-ROM (6)

- Les spécifications techniques du formatage des données informatiques sur la spirale du CD audio sont contenues dans le « Livre jaune »
  - Les octets sont encodés sous la forme de symboles de 14 bits permettant la correction d'erreur
  - Une trame de 588 bits est constituée de 42 symboles servant à encoder 24 octets avec correction d'erreur
  - Un secteur est constitué d'un groupe de 98 trames servant à encoder 2048 octets, avec un préambule de 16 octets de synchronisation et de numérotation du secteur, et d'une zone de contrôle de 288 octets

# CD-ROM (7)

- Du fait de la très grande redondance, la charge utile d'un CD-ROM n'est que de 28 %
  - Un CD-ROM peut encoder 650 Mo de données
- Le débit reste faible par rapport à un disque magnétique
  - En vitesse normale (x1), un CD-ROM lit 75 secteurs par seconde, soit 150 ko/s
  - En vitesse x32, on n'atteint que 4,8 Mo/s
  - On ne peut augmenter la vitesse de rotation sans briser les disques, dont le plastique est peu solide

# CD-ROM (8)

- Afin de rendre  les CD-ROM utilisables sur le plus d'architectures possible, le format du système de fichiers a été normalisé
  - Norme ISO 9660
- Trois niveaux
  - Niveau 1 : Nommage « à la MS-DOS » : 8+3 caractères, majuscules uniquement, huit sous-catalogues au plus, fichiers contigus
  - Niveau 2 : Noms de fichiers sur 32 caractères
  - Niveau 3 : fichiers non contigus

# CD-R (1)

- Les CD-R sont des CD initialement vierges sur lesquels on peut inscrire, une seule fois, des informations
  - Spécifications dans le « Livre orange »
- Ils contiennent un film réfléchissant sur lequel est plaquée une couche de matière colorée photosensible
  - Habituellement de la cyanine (bleue-verte) ou de la phtalocyanine (jaune orangée)

# CD-R (2)

- L'écriture s'effectue avec un laser de forte puissance (8 à 16 mW) qui opacifie la couche photosensible par chauffage au point de contact avec la surface réfléchissante
- La lecture s'effectue de façon classique avec un laser de faible puissance (0,75 mW)

# CD-RW

- Permettent la lecture et l'écriture
- La couche d'enregistrement est constituée d'un alliage métallique possédant un état cristallin et un état amorphe de réflectivités différentes
- Leur laser fonctionne avec trois puissances
  - Faible puissance : lecture
  - Moyenne puissance : fonte de l'alliage et refroidissement léger permettant le passage à l'état cristallin fortement réflechissant : réinitialisation
  - Haute puissance : fonte et passage à l'état amorphe

# DVD

- Même principe que les CD-ROM, mais densification de l'information
  - Laser de longueur d'onde plus petite (0,65 microns contre 0,78 microns)
  - Microcuvettes plus petites (0,4 microns, contre 0,8 microns)
  - Spires plus serrées (0,74 microns contre 1,6 microns)
- La capacité passe à 4,7 Go

# Écran (1)

- Principal dispositif de sortie des ordinateurs actuels
- Différentes technologies disponibles en fonction des usages et des contraintes
  - Tube cathodique (« *Cathod Ray Tube* », ou CRT)
  - Cristaux liquides (« *Liquid Crystal Display* », ou LCD)
  - Diodes émittrices organiques (« *Organic Light Emitting Diode* », ou OLED)
  - Plasma

# Écran (2)

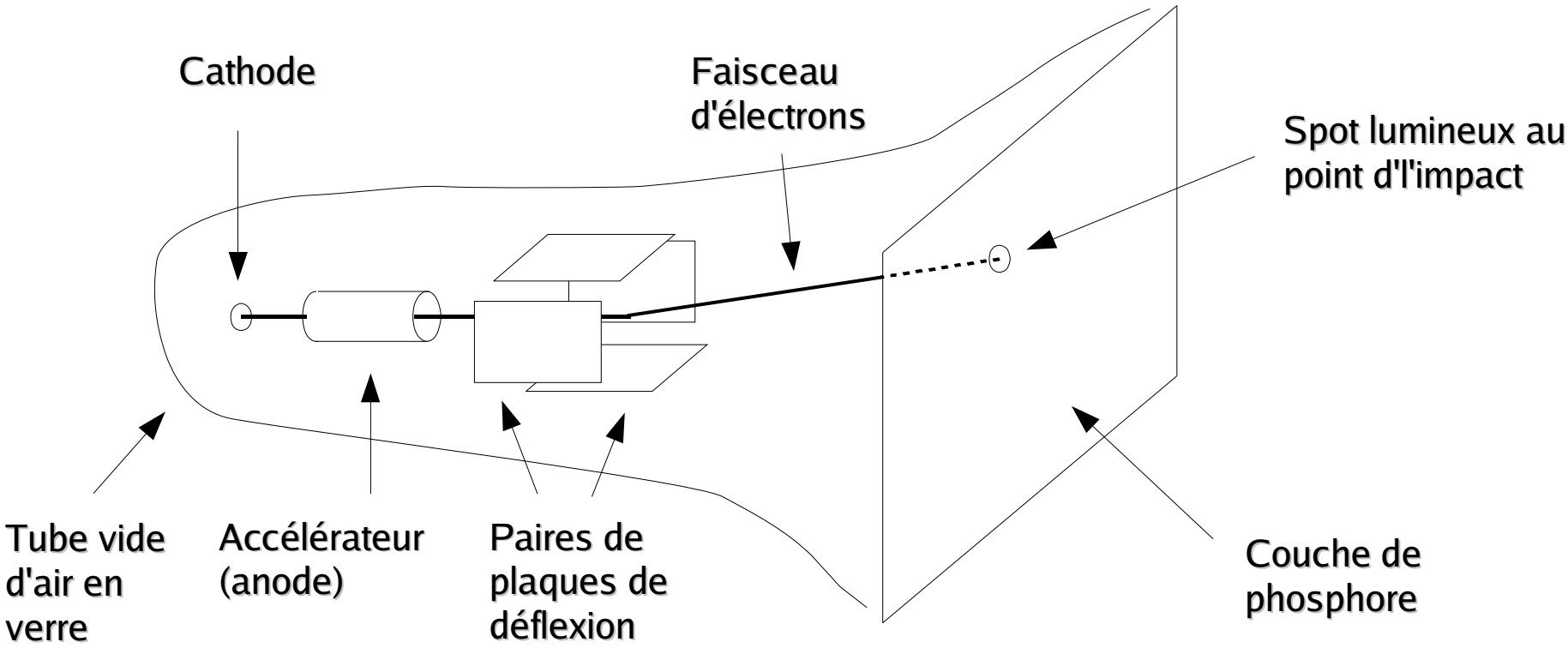
- Paramètres caractéristiques d'un écran
  - Résolution affichable : nombre maximal de pixels que l'écran peut afficher ou que l'oeil de l'observateur peut discriminer
    - Qualité visuelle des images statiques
  - Fréquence de rafraîchissement : nombre d'images distinctes que l'écran peut afficher par seconde
    - Fluidité des animations et confort de vision
  - Rémanence : temps que persiste l'impression visuelle d'une image après qu'elle a disparu
    - Confort de vision des animations

# Écran à tube cathodique (1)

- Un faisceau d'électrons balaye la surface interne de l'écran et excite une matière électroluminescente (phosphore) visible à travers la paroi de verre du fond du tube
- Les électrons sont produits par un ou plusieurs canons à électrons situés à la base du tube, accélérés, puis déviés par des plaques de déflexion portant une très haute tension

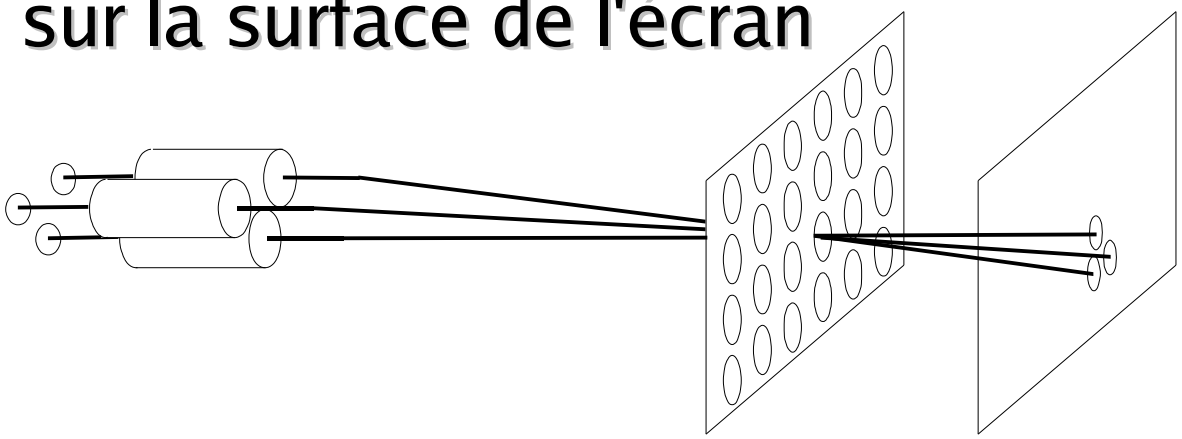
# Écran à tube cathodique (2)

- Écran monochrome : une seule sorte de phosphore tapisse uniformément la surface du tube (blanc, vert ou orange)



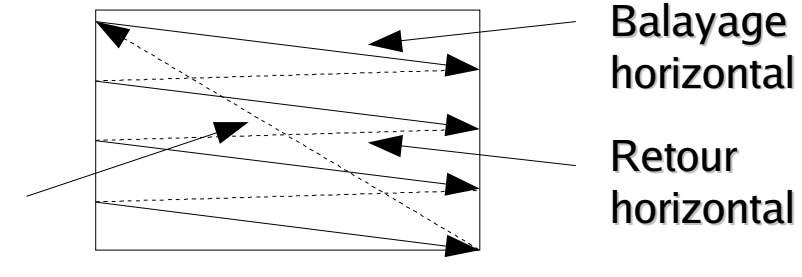
# Écran à tube cathodique (3)

- Écran couleur : on a trois canons indépendants qui balayent la surface ensemble, à travers un masque (grille) empêchant les pixels de « baver » les uns sur les autres
  - Le « pitch » est le pas de masque de la grille ainsi que la finesse du dépôt des points de phosphore sur la surface de l'écran



# Écran à tube cathodique (4)

- Deux types de contrôle du faisceau
  - Écrans vectoriels : le faisceau d'électrons suit exactement le tracé de l'objet à dessiner
    - Images de type « fil de fer » autrefois utilisées en CAO ou pour des applications spécialisées (écrans radar)
    - Pas de notion de pixels pour les écrans monochromes
    - Pixels de la taille du masque pour les écrans couleur
  - Écrans à balayage (« raster scan ») : le faisceau effectue un balayage régulier de toute la surface de l'écran
    - Résolution plus limitée



# Écran LCD (1)

- Les écrans LCD sont basés sur deux propriétés des cristaux liquides
  - Leur pouvoir polarisant sur la lumière qui les traverse
  - Leur capacité à s'orienter selon les lignes d'un champ électrique

# Écran LCD (2)

- Les écrans LCD sont constitués :
  - D'un certain nombre de lampes fluorescentes servant au rétro-éclairage de l'écran
  - D'un film polarisant ne conservant que la lumière polarisée dans une direction donnée
  - D'une couche de cristaux liquides, comprise entre un ensemble d'électrodes, permettant ou non d'orienter la lumière dans une direction orthogonale à celle de la polarisation initiale
  - D'un deuxième film polarisant filtrant la lumière ayant traversé les cristaux liquides

# Écran LCD (3)

- Si les cristaux liquides sont orientés dans le sens des deux filtres polarisants, la lumière passe et le pixel est allumé
- Si les cristaux liquides sont orientés dans le sens orthogonal à celui des deux filtres polarisants, le pixel est éteint
- On a en fait trois sous-pixels par pixel, codant les trois composantes lumineuses, et suivis de filtres colorés

# Écran LCD (4)

- Deux technologies
  - Matrice passive : chaque pixel est activé successivement par balayage de l'ensemble des pixels selon une technologie matricielle ligne-colonne
  - Matrice active : chacun des pixels conserve individuellement son niveau d'activité grâce à un jeu de transistors localisés au niveau de chaque pixel
    - TFT (« *Thin-Film Transistors* »)

# Écran OLED

- Constitué d'une couche de matière organique électro-luminescente prise en sandwich entre une cathode métallique et une matrice d'anodes transparentes analogue à celle des écrans à cristaux liquides
- Pas besoin de rétro-éclairage
- Écran extrêmement fin et consommant peu
- Excellente vitesse de rafraîchissement
- Problèmes de résistance à la chaleur, en voie de résolution

# Carte graphique (1)

- Dispositif de visualisation servant à la production de signaux vidéo à destination de dispositifs d'affichage (écrans, etc)
- Les plus anciennes cartes graphiques sont de simples mémoires d'image (« *frame buffer* ») construites autour d'une mémoire double-port
  - Accès en lecture et écriture par le processeur, qui la voit comme de la RAM classique
  - Accès en lecture seule par un registre à décalage qui alimente en intensité la sortie d'un signal vidéo

# Carte graphique (2)

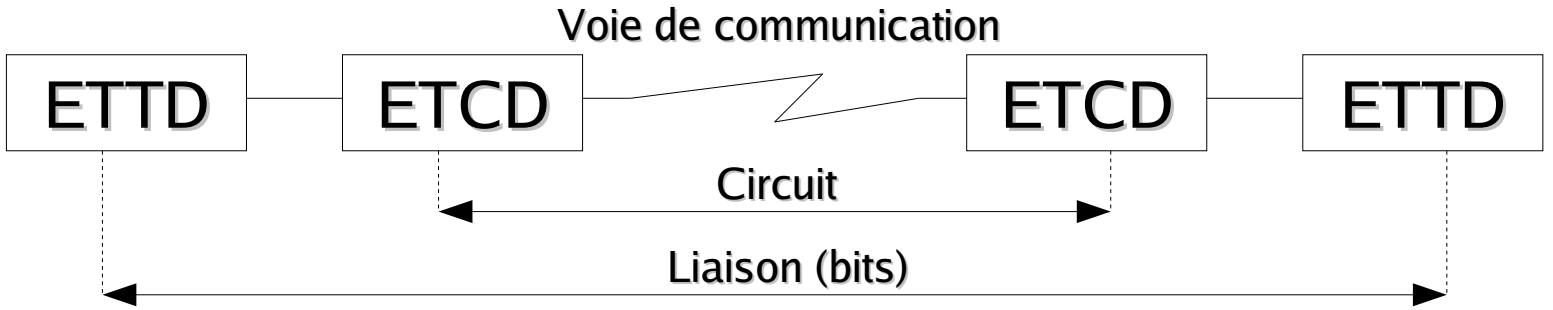
- A mesure que la puissance des processeurs a augmenté, de nouvelles fonctionnalités ont été mises en œuvre pour soulager le CPU
  - Manipulation de « sprites » (opérations BitBLT)
  - Tracé accéléré de primitives 2D (lignes, cercles)
  - Interpolation de textures
  - Calculs de projection 3D de polygones
  - Calculs d'illumination programmables
- La carte 3D est même parfois utilisée comme co-processeur arithmétique

# Modem (1)

- Jusqu'à récemment, la seule connectivité dont pouvait disposer la majorité des utilisateurs était la liaison téléphonique
  - Utilisation des lignes téléphoniques pour les transmissions entre ordinateurs
- Caractéristiques d'une liaison téléphonique
  - Transmission analogique et non pas numérique
  - Bande de fréquence de 300 à 3400 Hz, échantillonnée à 8000 Hz sur 256 niveaux : 64 kb/s
  - Pas adaptée au transport de signaux numériques

# Modem (2)

- La modulation consiste à encoder une information au sein d'un signal porteur moins sujet aux atténuations, distorsions et bruits
  - Nécessité d'un MOdulateur pour encoder le signal et d'un DÉModulateur pour le décoder : MODEM
- Modélisation d'une liaison



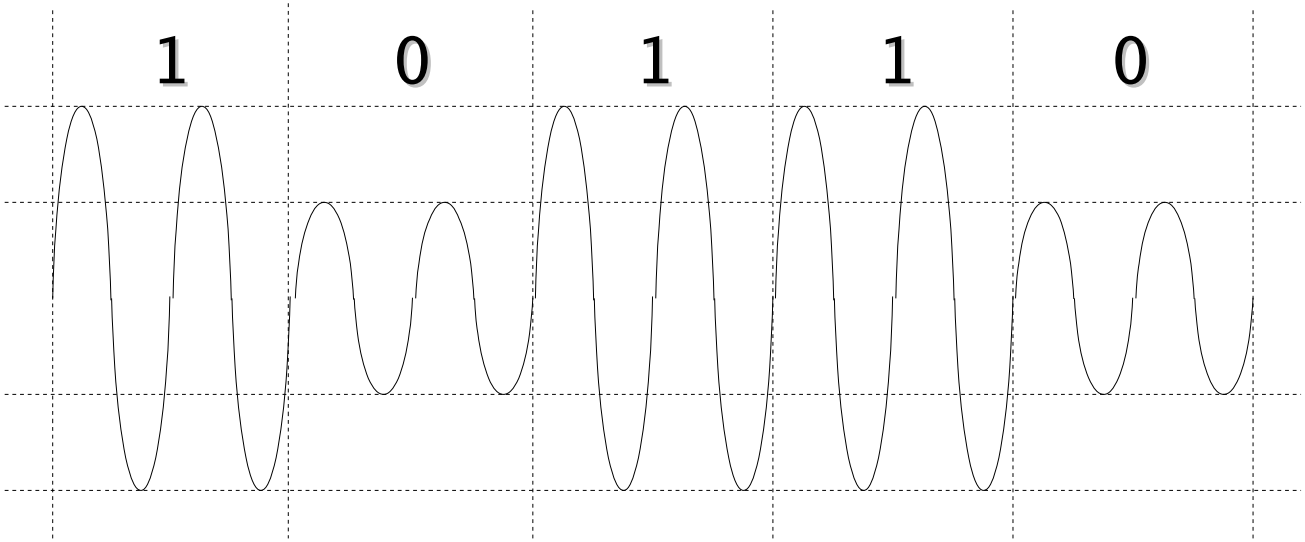

ETCD : Équipement Terminal de Circuit de Données (modem)
ETTD : Équipement Terminal de Traitement de Données (terminal, ordinateur)

# Modulation (1)

- Il existe trois types de modulation
  - Modulation d'amplitude
  - Modulation de fréquence
  - Modulation de phase

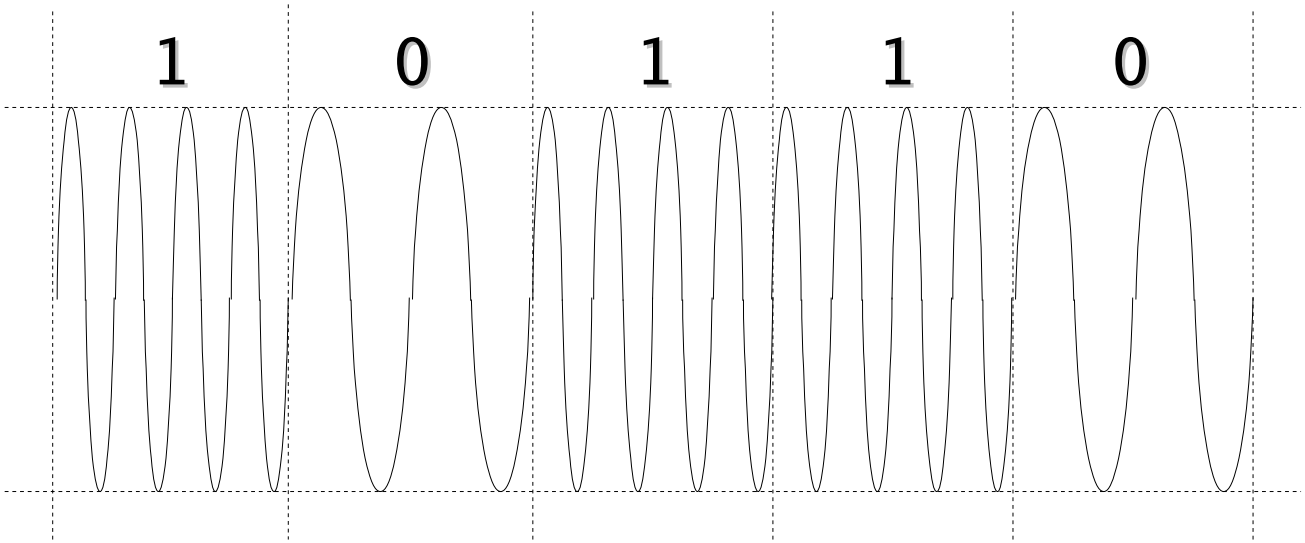
# Modulation (2)

- Modulation d'amplitude
  - Le signal module l'amplitude de la porteuse



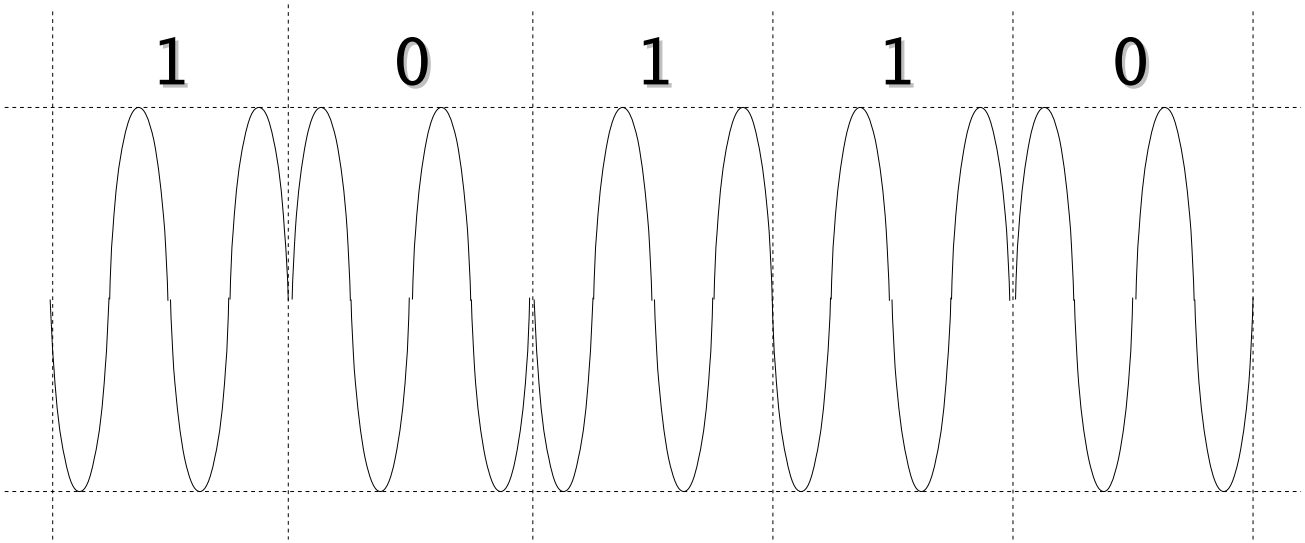
# Modulation (3)

- Modulation de fréquence
  - Le signal module la fréquence de la porteuse



# Modulation (4)

- Modulation de phase
  - Les changements d'état du signal modulent la phase de la porteuse



# Modulation (5)

- Pour des débits faibles (< 1200 bits/s), on utilise la modulation de fréquence
- Pour des débits plus importants, on utilise conjointement la modulation de phase et d'amplitude
- Lorsque le signal peut prendre plus de deux états différents, on peut ne plus parler en bits/s mais en bauds/s
  - Nombre d'états par seconde
  - Le débit en bits/s est un multiple du débit en bauds/s

# Multiplexage

- Assemblage de plusieurs signaux issus de sources distinctes en un seul signal composite destiné à être transmis sur une voie de communication commune
- Deux types de multiplexage
  - En temps : une voie à haut débit est décomposée en plusieurs voies à bas débit par allocation à chacune d'un quantum de temps d'émission
  - En fréquence : la bande passante du support est décomposée en bandes de fréquences (canaux) utilisables par des voies à bas débit

# ADSL (1)

- L'évolution des technologies Ethernet a montré que l'on pouvait atteindre des débits très importants sur de simples paires de fils
- Les technologies DSL (« *Digital Subscriber Line* ») permettent d'utiliser la « boucle locale » (connexion filaire point-à-point entre le central téléphonique et l'abonné) pour transmettre des informations supplémentaires
  - Peu coûteuses
  - Pas besoin d'implanter d'autres réseaux, de type câble coaxial ou fibre optique

# ADSL (2)

- Multiplexage en fréquence des informations sur la paire téléphonique
  - Bande de 0 à 4 kHz utilisée par la téléphonie « classique »
  - Bande de 4 kHz à 1.1 MHz, découpée en plusieurs canaux montants et descendants, utilisée pour la transmission d'informations numériques
- Utilisation de transmissions analogiques sur la boucle locale (même si DSL veut dire « boucle locale numérique »...)
  - Les lignes RNIS ne peuvent supporter l'ADSL

# ADSL (3)

- Au niveau du central, des DSLAMs (« DSL *Access Multiplexer* ») multiplexent en fréquence la voie téléphonique et les canaux de données
- Au niveau de l'abonné, un filtre passe-bas permet d'extraire le signal téléphonique, et un modem DSL gère la transmission sur les canaux montants et descendants

# ADSL (4)

- Les débits obtensibles dépendent :
  - De la longueur de la paire torsadée entre le central téléphonique et l'abonné
  - Des sources d'interférences et de bruits électro-magnétiques rencontrées
  - De la technologie DSL utilisée

# ADSL (5)

- Plusieurs technologies mises en œuvre :
  - HDSL : utilisation de deux paires de fils pour atteindre 1.1 Mbit/s sur 5 km
  - SDSL : utilisation d'une seule paire sur 3 km pour obtenir les débits de l'HDSL
  - ADSL : « *Asymmetric DSL* » : on a moins de canaux montants (256 kbits/s) que de canaux descendants
    - 4,9 km 2,048 Mbits/s (E1)
    - 3,7 km 6,312 Mbits/s (DS2)
    - 2,7 km 8,448 Mbits/s

# ADSL (6)

- VDSL : « *Very high speed DSL* » ou « *Video DSL* »
  - 1,5 km 12,96 Mbits/s (1/4 STS-1)
  - 1 km 25,82 Mbits/s (1/2 STS-1)
  - 300 m 51,84 Mbits/s (STS-1)

# Jeux de caractères (1)

- Conventions de codage des caractères représentables sur la machine
  - Associe à chaque symbole une valeur binaire
- Les symboles peuvent être :
  - Des caractères
  - Des marques diacritiques combinables avec les caractères qui les suivent
    - Accents, esprits, clés, etc.

# Jeux de caractères (2)

- Cette convention de codage est nécessaire :
  - Au sein de l'ordinateur, pour que les caractères saisis au clavier soient bien ceux restitués à l'écran et sur l'imprimante
  - Entre ordinateurs, pour permettre l'échange d'informations
- Nécessité d'un langage commun pour que deux entités puissent communiquer

# Jeu de caractères ASCII (1)

- « *American Standard Code for Information Interchange* »
- Codé sur 7 bits, pour la transmission modem
  - Codes de 0x00 à 0x1F : caractères de commande utilisés dans le dialogue avec les périphériques
    - NUL, CR, LF, BS, BEL, EOT, ...
  - Codes de 0x20 à 0x2F : caractères de ponctuation
  - Codes de 0x30 à 0x39 : chiffres de '0' à '9'
  - Codes de 0x41 à 0x5A : lettres majuscules
  - Codes de 0x61 à 0x7A : lettres minuscules

# Jeu de caractères ASCII (2)

- Comme les ordinateurs manipulent des octets, les valeurs de 0x80 à 0xFF restent disponibles
- Utilisées par chaque fabricant de machine à sa guise, sans normalisation
  - Exemple des pages de code étendues des BIOS PC
    - Caractères semi-graphiques
    - Symboles mathématiques
    - Caractères accentués, ...
  - Exemple des pages de code de Windows
- Tentative de normalisation ISO 646 « Latin-1 »

# Jeux de caractères ISO 8859

- Normalisation de pages de codes sur un octet
  - Les 128 premières valeurs équivalent au code ASCII
  - Les 128 derniers caractères dépendent du numéro de page de code
- Numéros de page de code :
  - 8859-1 : langues romanes (« Latin-1 »)
  - 8859-2 : langues slaves d'origine latine (hongrois, polonais, tchèque, ...)
  - 8859-3 : autres langues écrites avec l'alphabet latin (turc, maltais, espéranto)

# Jeu de caractères Unicode

- Code l'ensemble des alphabets internationaux (arabe, chinois, hébreu, hindi, japonais, …)
  - Codé sur 16 bits
  - 65536 codes « points » différents, pour environ 200 000 signes utilisés par toutes les langues écrites
  - Les 256 premiers codes équivalent au « Latin-1 »
- Le consortium Unicode décide de l'attribution des codes encore inutilisés
  - Maximise l'utilisation des codes diacritiques pour économiser les codes

# Imprimante (1)

- En dépit de l'explosion de la transmission par canaux numériques, l'impression papier reste très utilisée, surtout pour les documents officiels
  - Diminution de la quantité de papier « poussé »
  - Explosion de l'impression de documents pour usage individuel (usage unique)
    - Facilité d'utilisation de ce médium (prix, robustesse, facilité d'usage, etc)

# Imprimante (2)

- Il existe de nombreuses technologies d'imprimante, caractérisées par leur coût et leurs performances
  - Imprimantes à marteaux
  - Imprimantes à aiguilles
  - Imprimantes thermiques
  - Imprimantes à jet d'encre
  - Imprimantes laser, ...

# Imprimante à marteaux

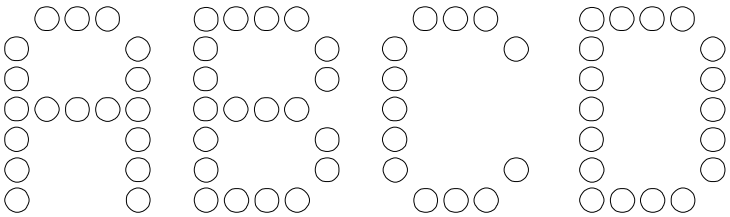
- Les toutes premières imprimantes étaient des machines à écrire améliorées : un ensemble de marteaux correspondant à chaque caractère venaient frapper la feuille de papier posée sur le chariot mobile en y appuyant un ruban encreur
- Les imprimantes se sont ensuite adaptées aux gros volumes de papier à imprimer : le papier en accordéon reste fixe et un chariot portant les marteaux se déplace
  - Imprimantes « à marguerites »
  - Changement de la marguerite pour changer de fonte

# Imprimante à rouleaux

- Pour les gros volumes d'impression, le déplacement du chariot est trop lent
- Les imprimantes à rouleaux sont constituées d'un ensemble de rouleaux contenant chacun l'ensemble des caractères imprimables, pivotables individuellement, couvrant toute une ligne, et qui sont appliqués simultanément sur la feuille de papier
  - On imprime ligne par ligne
  - Utilisées jusqu'à récemment pour imprimer les déclarations d'impôt...

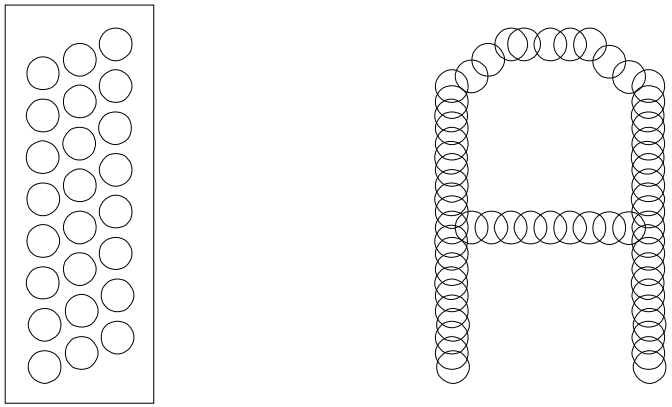
# Imprimante à aiguilles (1)

- Les imprimantes à aiguilles portent sur leur chariot une ligne d'aiguilles commandées individuellement par des électro-aimants et frappant le ruban pour dessiner les caractères
  - Servent encore à l'impression de factures sur papiers carbone
  - Possibilité de faire du graphisme (sommaire)



# Imprimante à aiguilles (2)

- Sur les modèles haut de gamme, les aiguilles se chevauchent pour améliorer le rendu des figures tracées



- Certaines imprimantes à aiguilles disposaient d'un ruban quadrichrome permettant de réaliser des impressions en couleur

# Imprimante thermique

- L'impression ne se fait pas par dépôt de matière sur le papier mais par noircissement par brûlure de la surface du papier thermosensible
  - Impression monochrome
  - Le papier thermosensible vieillit très mal et devient très rapidement illisible
- Pour les applications actuelles (terminaux de paiement), on n'a pas de chariot mobile mais toute une rangée de pointes chauffantes

# Imprimante à jet d'encre

- Le principe est le même que pour l'imprimante à aiguilles : un chariot mobile porte une tête capable d'imprimer une ligne verticale de points sur la feuille
- L'encre est transférée par micro-jets à partir des buses alignées sur la tête
- Chaque injecteur est constitué d'une chambre dans laquelle l'encre est chauffée jusqu'au point d'ébullition, et s'échappe par la buse de sortie pour aller frapper le papier

# Imprimante laser (1)

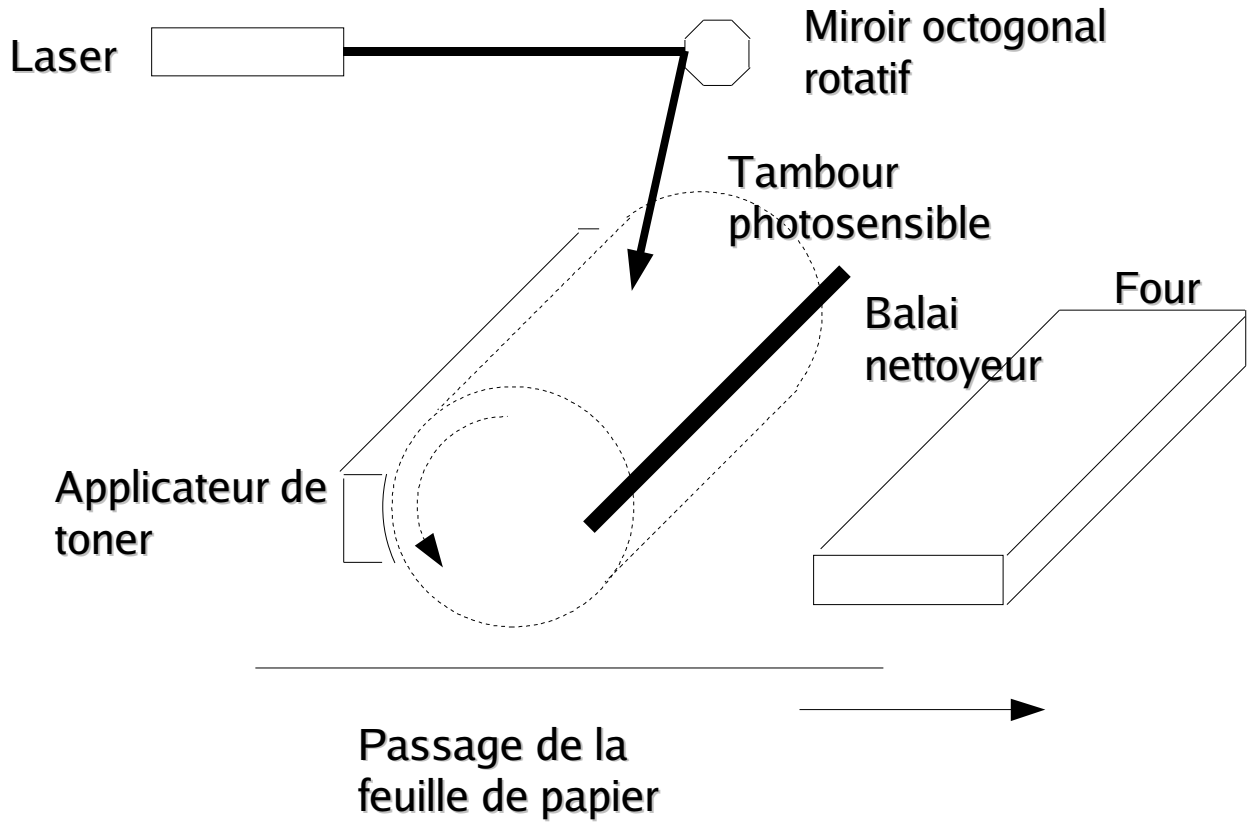
- Le coeur de l'imprimante est un tambour rotatif photosensible dont la surface passe devant plusieurs équipements
  - Au début du cycle d'impression, la surface du tambour est électrisée à un potentiel élevé, qui la rend photosensible
  - La surface est ensuite balayée par le pinceau d'un rayon laser, au moyen d'un miroir rotatif tournant à grande vitesse
    - Le laser décharge le tambour à son point d'impact
    - En modulant l'énergie du laser, on crée une image sous forme de potentiels sur la surface du tambour

# Imprimante laser (2)

- Le toner, poudre noire électrostatique, est attirée par la surface encore électrisée du tambour, et se dépose ensuite sur la surface du papier
- Le reste de toner présent sur le rouleau est ensuite nettoyé, et un nouveau cycle recommence
- La feuille de papier passe elle dans un four qui cuit le toner et permet qu'il ne parte plus de la feuille

# Imprimante laser (3)

- Schéma de fonctionnement



# À faire... (1)

- Détailler PCI
- Micro-architecture avec état (coup du champ JMP à destination variable dans le micro-code)

# Trucs en vrac

- Le salon des refusés (pour le moment...)