INTRODUCTION

INF4170 – Architecture des ordinateurs

UQÀM

Dans l'introduction, la partie de la matière du cours des Profs. Peña & Perez-Uribe & Mosqueron est utilisée

Introduction

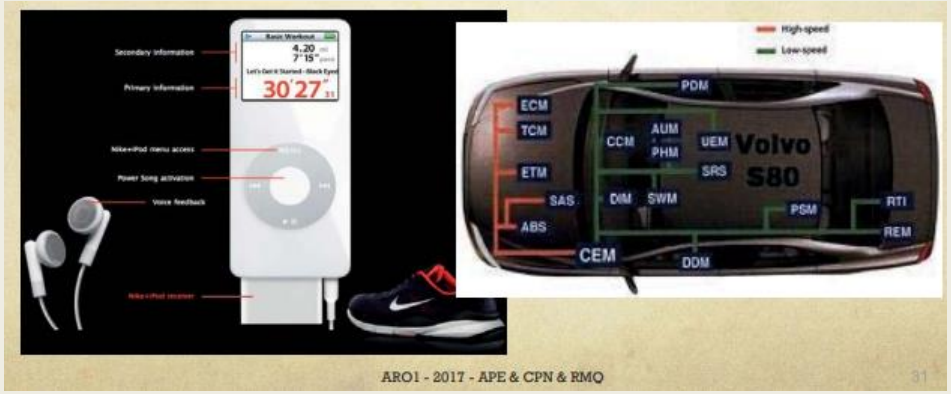
- Les systèmes informatiques jouent un rôle de plus en plus important dans notre société

Système informatique

- Un système informatique est un ensemble de composants de type logiciel (software) et matériel (hardware), mis ensemble pour collaborer dans l'exécution d'une application
- Le principal composant matériel est l'ordinateur
- Un informaticien doit comprendre le fonctionnement de tous les composants d'un système, sans se limiter au logiciel

Le monde informatique

- La grande majorité des microprocesseurs se trouvent non pas dans les ordinateurs, sur les bureaux, mais "embarqués" dans la plupart des outils qui nous entourent (voitures, chaussures, machines à café, postes de télé et de radio, téléphones mobiles, etc)
- Une Mercedes classe S : 65 processeurs

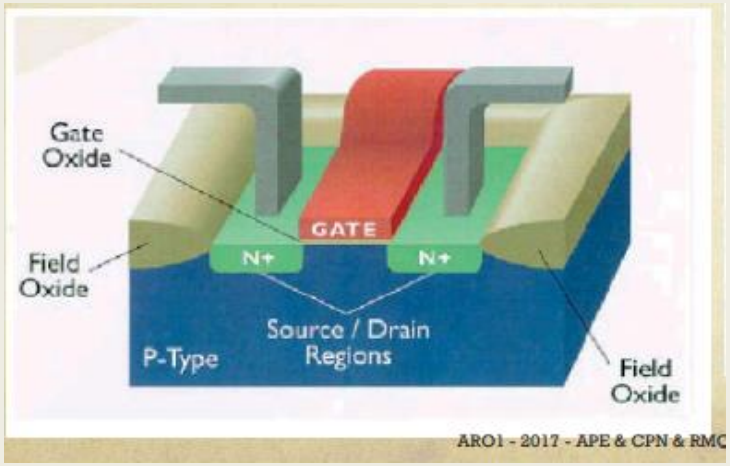


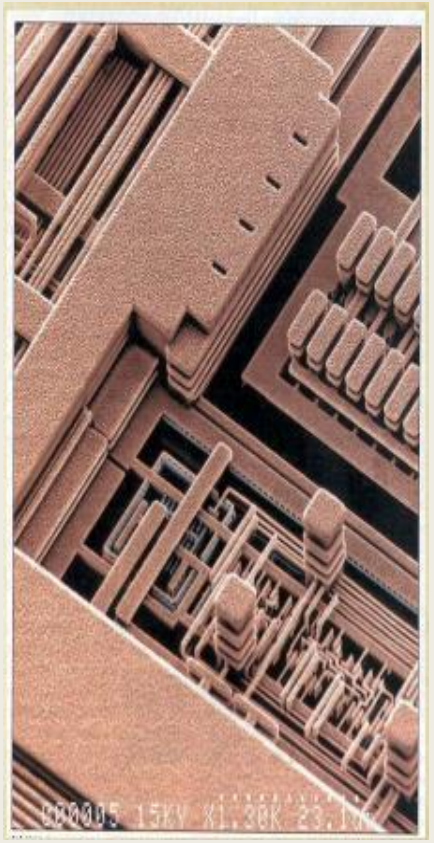
Que trouve-t-on dans le coffre d'un véhicule autonome ?

Le modèle Roborace (prototype autonome) conçu par Nvidia, embarque un ordinateur de bord qui adapte automatiquement sa conduite en fonction des retours de ses capteurs

Technologie

- Cette évolution a été permise par l'évolution de la technologie, qui a permis une miniaturisation croissante des dispositifs électroniques (le transistor)



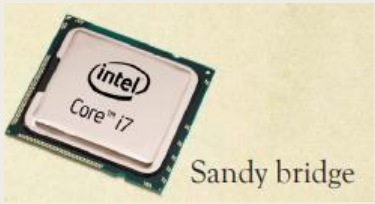


Loi de Moore

- En avril 1965, six ans après l'invention du circuit intégré, Gordon Moore, co-fondateur d'Intel réalisa une prédiction connue comme la loi de Moore: le nombre de transistors dans un circuit intégré sera multiplié par deux chaque année
- La loi est appliquée à tous les paramètres de la technologie, notamment vitesse et performance

10th Gen Intel Core i9-10980HK

- Ce qui est vrai, c'est qu'Intel introduit un nouveau processus de fabrication tous les 2 ans:
- 2000: 0.13 m
- 2002: 90 nm
- 2006: 65 nm
- 2008: 45 nm
- 2010: 32 nm
- 2012: 22 nm
- 2014: 14nm
- 2016: 10nm
- 2018: 7nm

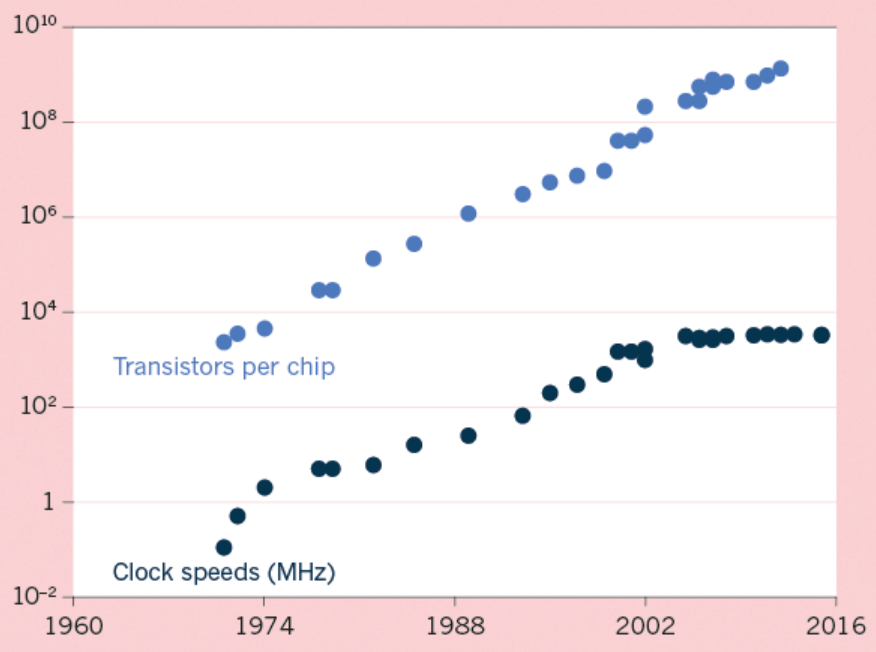


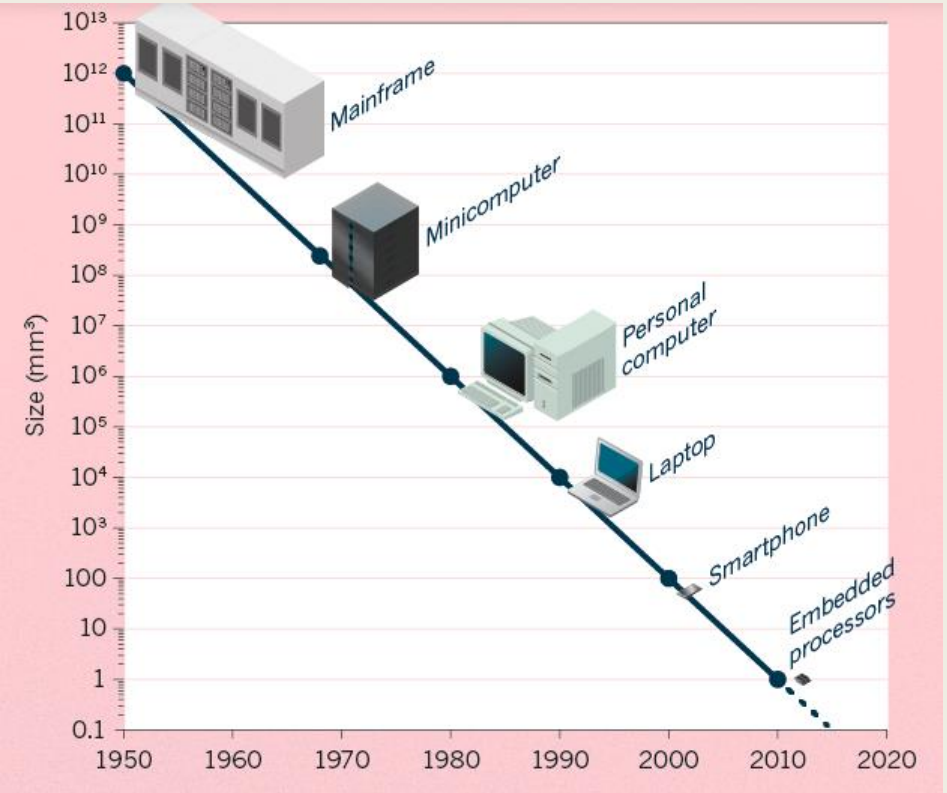
Ivy bridge de Intel (3D transistors)

Processeurs Core M de Intel (Q4 2014)

"Intel's next-gen 7nm chips are delayed until at least 2022"

Loi de Moore





le premier microprocesseur commercialisé par Intel en 1971 intégrait 2 300 transistors d'une finesse de gravure de 10 µm (micromètres), la génération de microprocesseurs actuels en intègre plus de 4 300 000 000, soit 1 869 565 fois plus.

La Loi de Moore, toujours d’actualité?

- « La cadence a ralenti ces dernières années. Elle est aujourd'hui sur le point de s'arrêter. À force de graver des composants électroniques toujours plus fins, années après années, passant du micro au nano, de l'échelle du cheveu à celle des **bactéries**, l'industrie de la microélectronique a fini par atteindre l'**atome**. **La limite est là**. La course à la **miniaturisation s'achève**. En **2018**, seuls **trois** industriels au monde étaient encore en lice pour graver des composants électroniques de **7** nm (nanomètres) : **Intel**, **Samsung** et **TSMC**, le fournisseur taïwanais d'**Apple**. Seuls les **deux derniers** sont aujourd'hui capables de franchir l'obstacle suivant des **5** nm. IBM, Toshiba, Sony… tous les autres géants de l'électronique ont déclaré forfait. Et si la prochaine étape, fixée à **3** nm, est peut-être atteignable, personne, sans doute, n'ira au-delà. **En 2021, 2022 au plus tard, il en sera fini de la loi de Moore**. » **Hugo Leroux. Science&Vie**

La Loi de Moore, toujours d'actualité?

- La limite est physique
  - *on s'approche de l'échelle de l'atome, où les lois quantiques de l'infiniment petit prévalent*
- Avec 5 nm, le plus petit des transistors actuels équivaut à seulement 10 atomes de silicium mis bout à bout ⟹
  - *Échappement des électrons*
    - de ralentissement des performances
    - de dysfonctionnements
- La démarche lancée l'an dernier et baptisée "More than Moore" essaie à ce titre de réfléchir au futur et à l'après miniaturisation
  - *Fabrication de puces en 3D, changement de matériaux ou encore combinaison de plusieurs fonctions sont notamment à l'ordre du jour*

1997 2011 2015

DEEP BLUE (IBM)
> 10 milliards opérations/sec
200 million mouvements/sec
1.5 Tonnes

iPhone 4S
> 10 milliards op/sec
140g

Smartphones
> 100 milliards op/sec

Objectif du cours

- Étudier les méthodes et techniques utilisées dans les architectures modernes pour améliorer les performances
- Comprendre les principes de base du fonctionnement interne des ordinateurs et comprendre comment cette organisation interne affecte les performances.
- Comprendre les interrelations entre logiciel et matériel, particulièrement dans les machines modernes
- Avoir un aperçu des directions futures vers lesquelles les architectures vont se développer

Architecture des ordinateurs

- Architecture des ordinateurs : domaine de l'informatique centre sur les machines
  - *point de vue à la fois matériel et logiciel*
- Comprendre comment les principes de base du fonctionnement interne des ordinateurs affecte les performances
- Décoder la relation entre l'architecture et les applications :
  - *Adapter les méthodes numériques, les algorithmes et la programmation*
  - *Comprendre le comportement d'un programme*
  - *Optimiser les codes de calcul en fonction de l'architecture*

Comprendre la performance d'un programme

- Performance d'un programme dépend de la combinaison des facteurs suivants:
  - *Algorithmes utilisés*
  - *Ensemble des logiciels utilisés lors de création et de traduction du programme en code machine*
  - *Efficacité d'exécution des instructions générées par un ordinateur*
- **Performance est affectée et par le matériel et par les logiciels**

Concept clé de conception

Abstraction

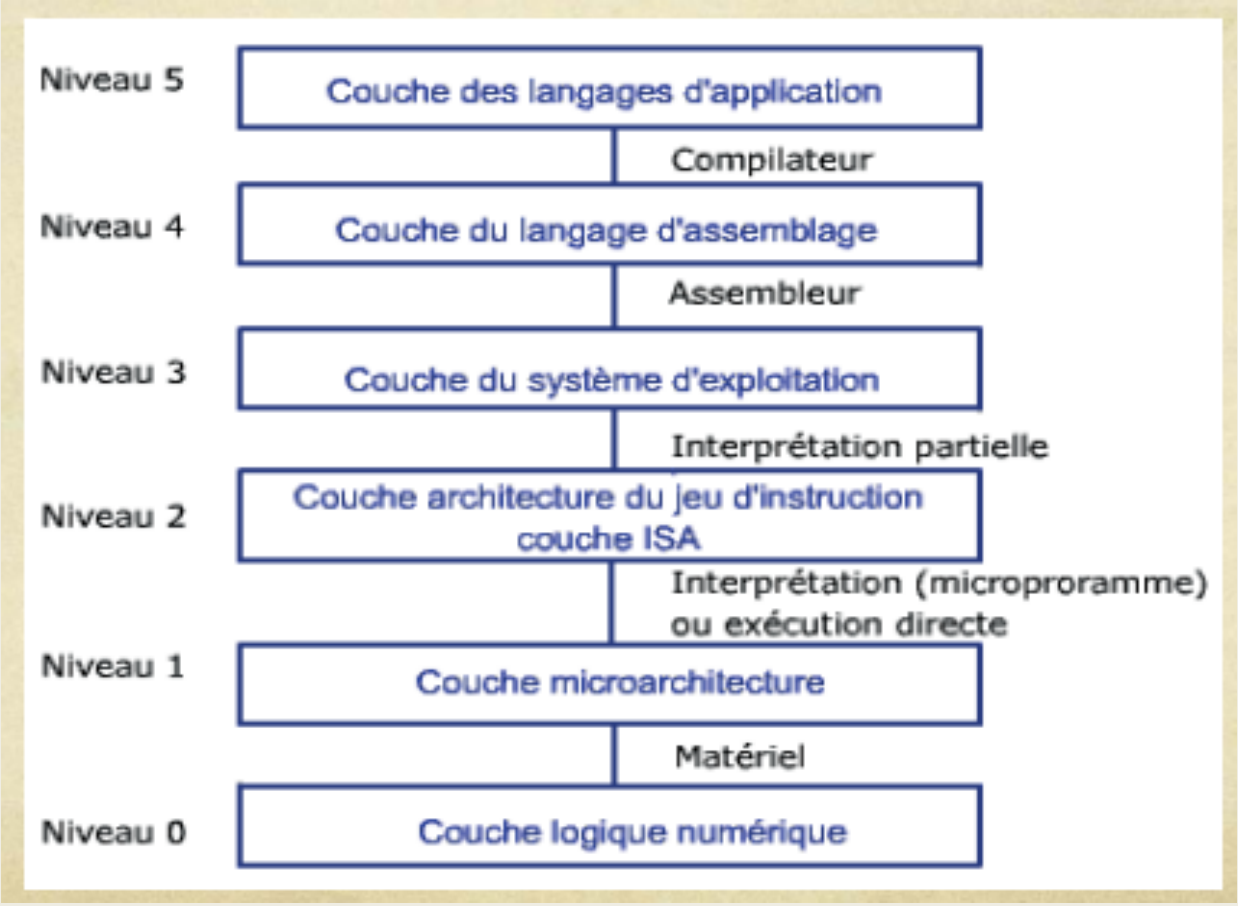
Abstraction

- La solution d'un problème en informatique va du niveau le plus abstrait vers le niveau le plus détaillé
- Un grand système informatique est organisé de façon hiérarchique: une partie prend ses ordres de la partie hiérarchique supérieure et, à son tour, peut transmettre ses ordres à des parties inférieures hiérarchiquement
- Il n'est pas nécessaire de connaître complètement chaque niveau d'un système informatique pour l'utiliser correctement

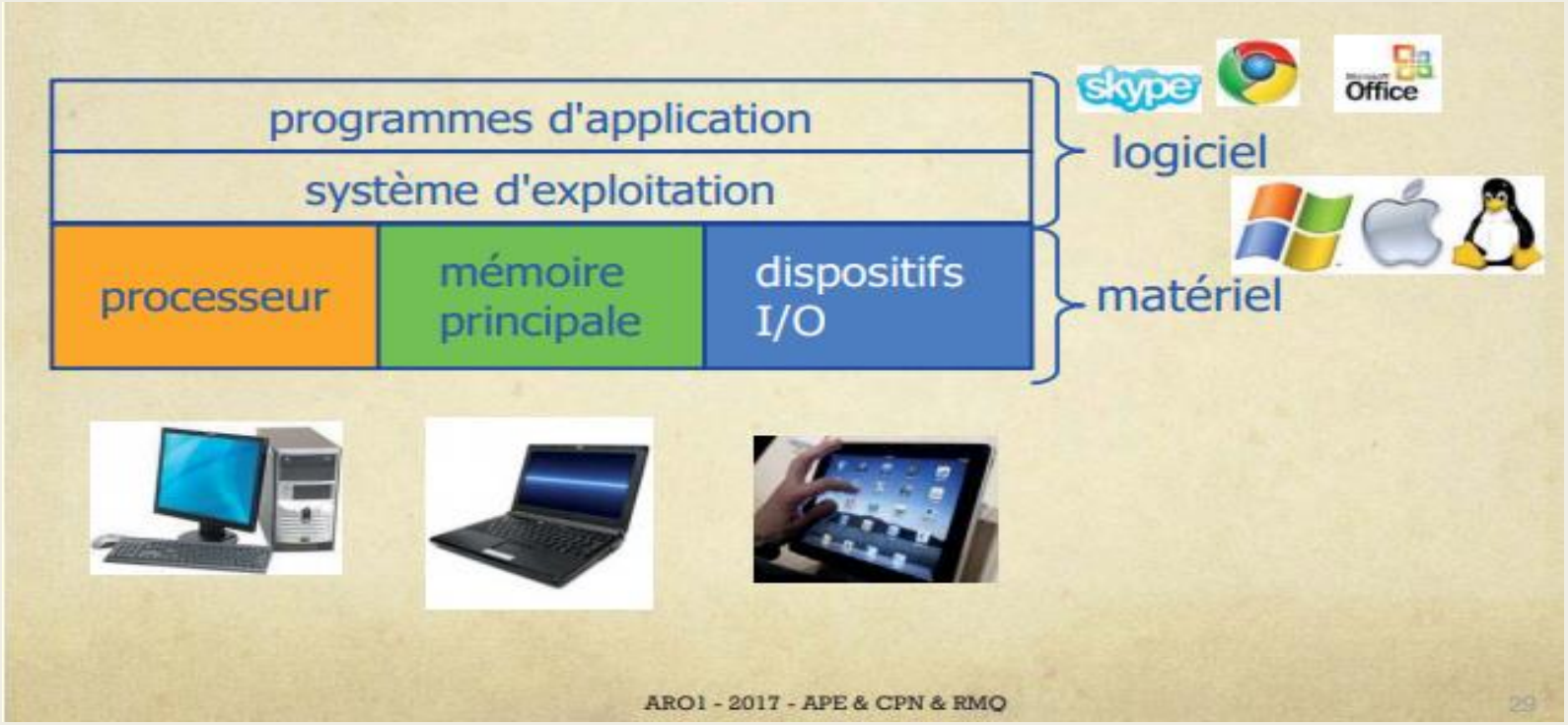
Niveaux d'abstraction en informatique

- Les niveaux
  - *Application (algorithme)*
  - *Langage de haut niveau*
  - *Système d'exploitation*
  - *Architecture de la machine*
  - *Microarchitecture*
  - *Circuits logiques*
  - *Dispositifs électroniques*
- A chaque niveau on peut utiliser un langage différent

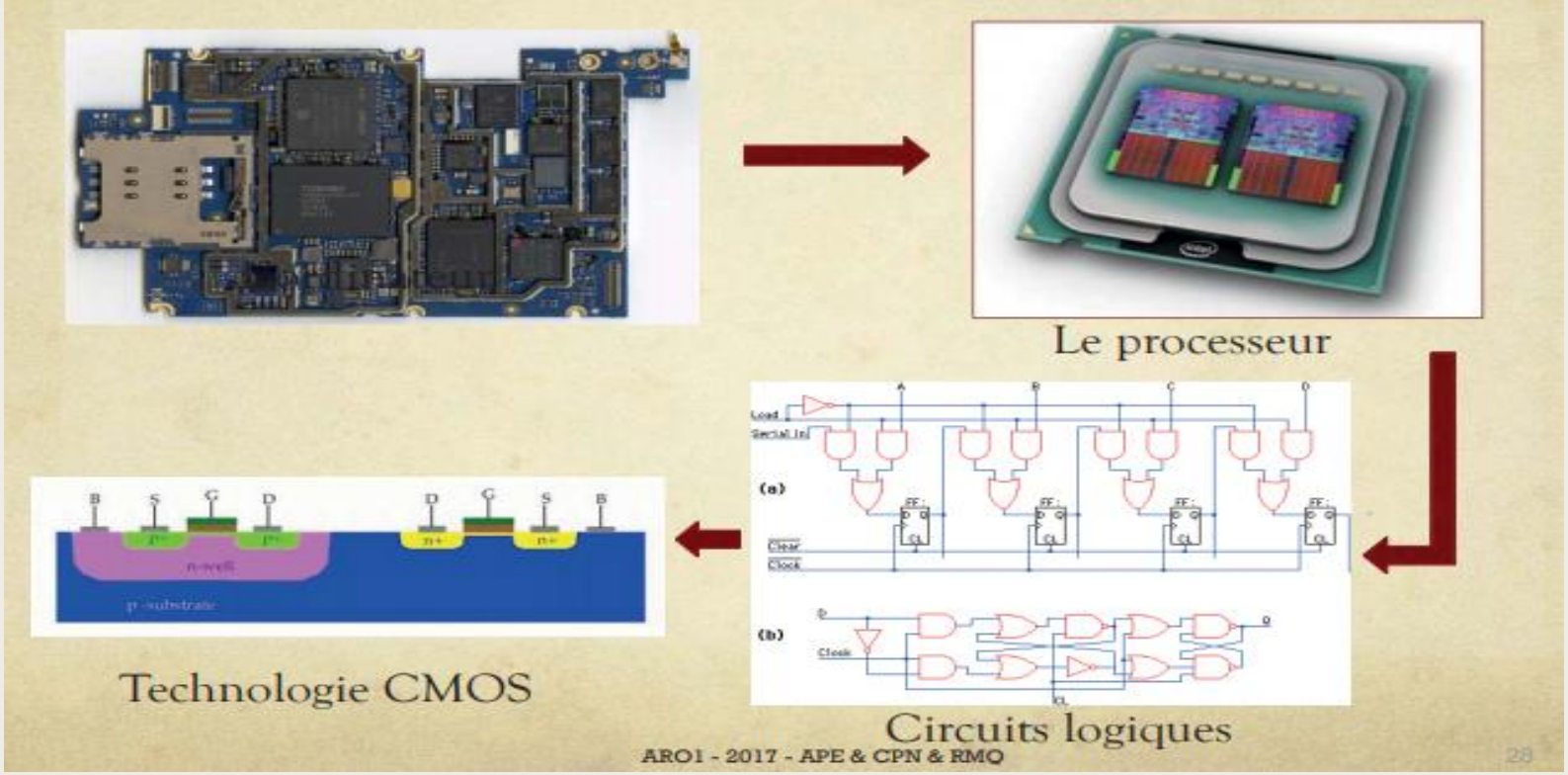
Niveaux d'abstraction



Niveaux d'abstraction

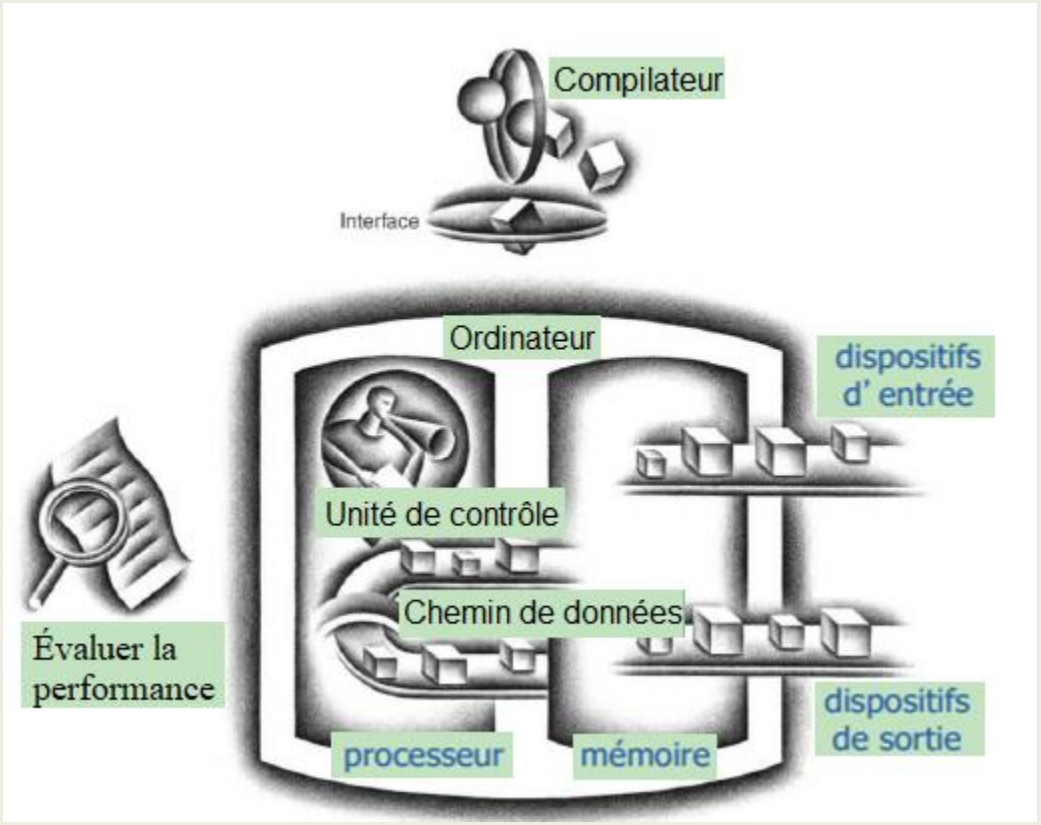


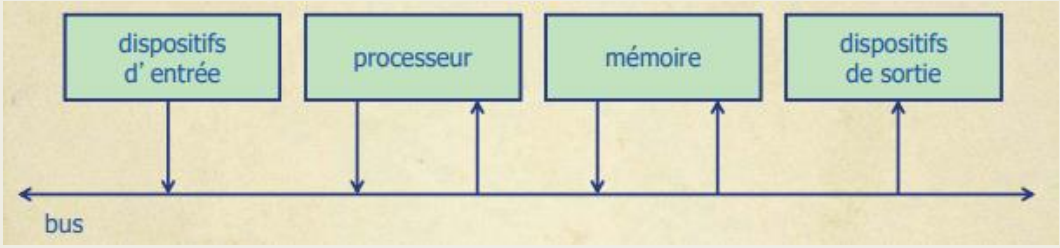
Niveaux d'abstraction



Principaux composants d'un ordinateur

- l'ordinateur est un dispositif de calcul universel
- L'universalité de l'ordinateur est possible grâce à la programmation: l'utilisateur doit indiquer par un programme les pas à suivre pour exécuter une tâche particulière
- On peut voir 4 grandes parties dans un ordinateur:

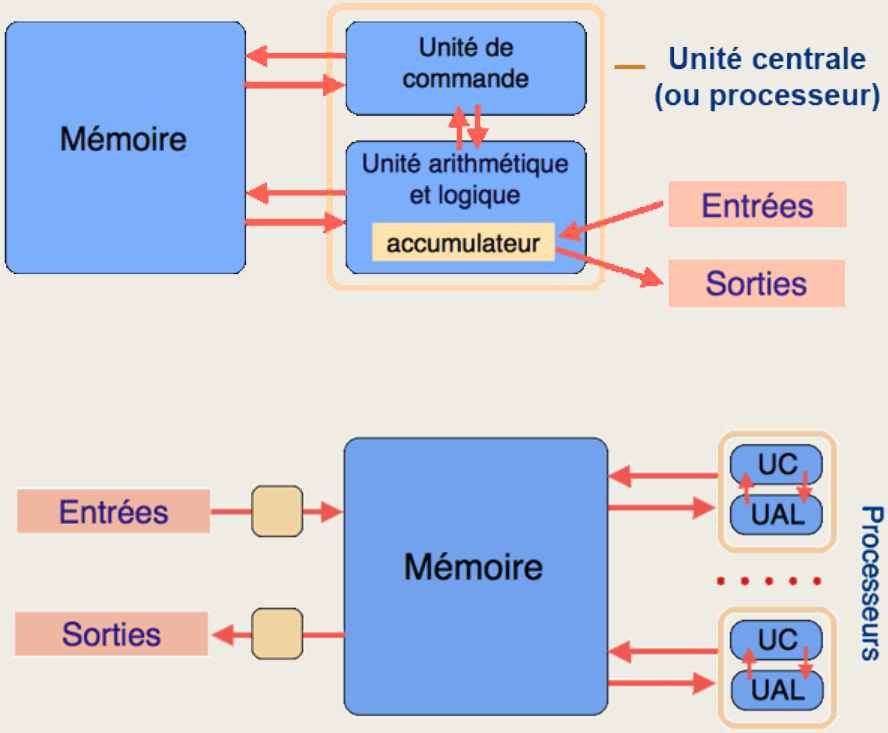




Ordinateur

- Un ordinateur est une machine électronique composée de plusieurs parties interconnectées par des fils (bus)
- Le bus est un ensemble de fils électriques interconnectant les différents composants
- A tout moment, tout fil dans l'ordinateur se trouve à un voltage haut ou bas interprété comme un 1 ou un 0

Modèle de Von Neumann



Concepts Clés

1. Séparation entre l'unité de commande et l'unité arithmétique
2. l'idée du programme enregistré : les instructions, au lieu d'être codées sur un support externe sont enregistrées dans la mémoire selon un codage conventionnel

https://interstices.info/le-modele-darchitecture-de-von-neumann/

Puissance de deux

- 2^{10} = 1 kilo $\approx 10^3$ (1024)
- 2^{20} = 1 mega $\approx 10^6$(1,048,576)
- 2^{30} = 1 giga $\approx 10^9$(1,073,741,824)
- 2^{40} = 1 tera $\approx 10^{12}$

Architecture des ordinateurs

- Un peu d'histoire ...
- https://www.tvibrant.com/watch/Comment-%C3%A7a-marche--La-fabuleuse-aventure-du-micro-ordinateur,-1994-,-Documentaire__W20d8Pj2fy8/

PERFORMANCE

INF4170 – Architecture des ordinateurs

UQÀM

Introduction

- Lorsqu'on choisit un ordinateur, on veut que la machine choisie soit performante

- De quelle façon peut-on évaluer la performance d'un ordinateur ?

Introduction

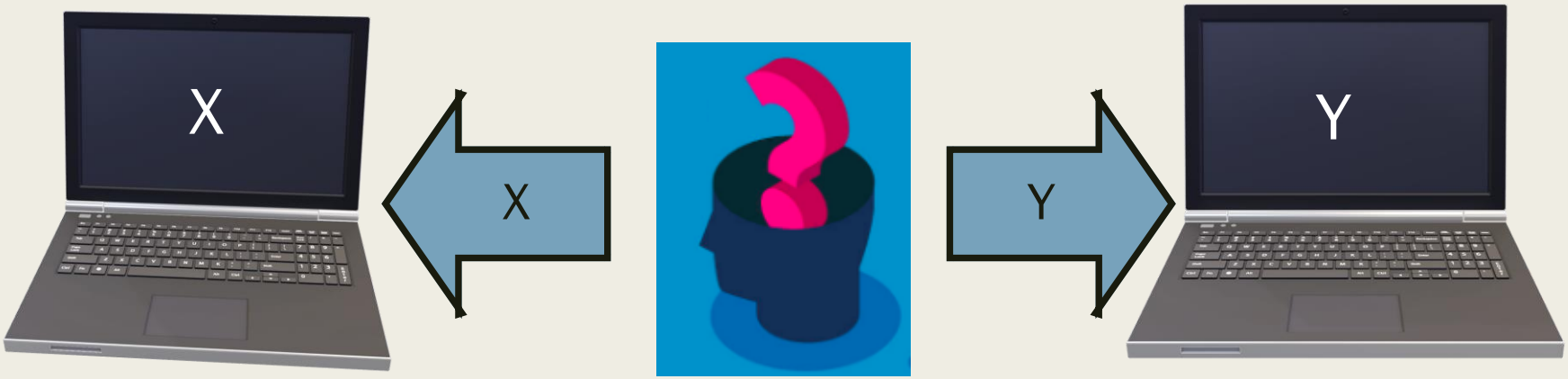
- Comment la performance est mesurée ?
- Quelles sont les limitations de ces mesures et leur importance dans le choix d’un ordinateur ?

Introduction

- On va définir:
  - *Différents moyens de déterminer la performance*
  - *Métriques de la performance d'un point de vue utilisateur et concepteur*

Introduction

Quand on dit qu'un ordinateur est plus rapide qu'un autre, qu'est-ce que cela signifie ?



https://www.gettyimages.fr/photos/homme-interrogatif?phrase=homme%20interrogatif&sort=mostpopular

Définir la performance

- Utilisateur d'une machine de bureau
  - La réduction du temps de réponse ou du temps écoulé (temps écoulé entre le début et la fin d'un événement)
- Responsable de centre informatique avec un gros système serveur
  - L'augmentation du débit (quantité totale de travail effectué en un temps donné)

Mesurer la performance

- **Temps de réponse**
- **Débit**

On a besoin de différentes métriques ainsi que les différents ensembles d'applications pour estimer la performance

Métrique: Temps d'exécution

- Temps d'exécution peut être défini (définition simple) comme: *Temps écoulé*

- Performance et temps d'exécution sont réciproques, l'augmentation de performance diminue le temps d'exécution

$$Performance_X = \frac{1}{Temps\ exécution_X}$$

Métrique: Temps d'exécution

- Comparer des conceptions différentes
  - *Comparer les performances relatives de deux machines différentes, X et Y*

$$Performance_X > Performance_Y$$

$$\frac{1}{Temps\ exécution_X} > \frac{1}{Temps\ exécution_Y}$$

$$Temps\ exécution_Y > Temps\ exécution_X$$

Métrique: Temps d'exécution

« X est n fois plus rapide que Y »

$$\frac{Performance_X}{Performance_Y} = \frac{Temps\ exécution_Y}{Temps\ exécution_X} = n$$

Mesure clé: Temps

- La mesure clé:

Temps

- L’ordinateur qui réalise la même quantité de travail en un temps moindre est le plus rapide.

Mesurer la performance

- **Temps écoulé**: temps nécessaire pour terminer une tâche, qui comprend tout
  - *accès disque*
  - *accès mémoire*
  - *activités d'entrée-sortie*
  - *temps système du système d'exploitation*

Temps CPU

- Multiprogrammation:
  - *CPU peut travailler sur un autre programme lors d'une attente d'E/S sans pouvoir réduire le temps d'exécution d'un programme*
- **Temps CPU** correspond au temps de travail du CPU sans considérer le temps d'attente des E/S ou pour exécuter d'autres programmes
  - *Temps de réponse vu par l'utilisateur est le temps écoulé du programme, pas le temps CPU*

Temps CPU

- Temps CPU
  - *Temps CPU utilisateur*
  - *Temps CPU système*
  - *Différencier ces deux temps*

Difficile

- Utilisons la somme du temps CPU utilisateur et du temps CPU système

Mesurer la performance

- Performance fondée sur
  - *Le **temps écoulé***
  - *Le **temps CPU***
- **Performance système** est utilisé pour le temps écoulé sur un système sans charge
- **Performance CPU** renvoie au temps CPU utilisateur sur un système sans charge

Comprendre les performances des programmes

- Différentes applications - différents aspects de performance
- Par ex.
  - *Applications sur les serveurs, dépendent de la performance des E/S*
  - *E/S : le matériel et le logiciel*
- Conclusion: considérer les différentes métriques de performance

Comprendre les performances des programmes

- Pour améliorer la performance d'un programme
  - *Comprendre la signification des métriques*
  - *Utiliser les métriques dans la recherche des parties problématiques de programme*

Comprendre les performances des programmes

- Utilisateur d'ordinateur, Concepteurs d'ordinateurs
  - Vitesse d'exécution des fonctions de base
    - Fréquence d'horloge, cycle d'horloge sont utilisés dans les contextes différents
- Lien entre un cycle d'horloge et le temps d'exécution

Performance CPU

- Les utilisateurs et les concepteurs utilisent les différentes métriques pour évaluer la performance

- Si on arrive à relier ces différentes métriques, on pourrait estimer l'impact du changement de conception d'un point de vue de l'utilisateur

Performance CPU

- Temps CPU d'un programme :

1. $$\textbf{Temps CPU} = \textbf{Cycles horloge CPU pour un programme} \times \textbf{Temps cycle horloge}$$
2. $$\textbf{Temps CPU} = \frac{\textbf{Cycles horloge CPU pour un programme}}{\textbf{Fréquence horloge}}$$

- Concepteur matériel peut améliorer le temps CPU en réduisant
  - *Nombre de cycles nécessaires pour exécuter un programme*
  - *Temps du cycle d'horloge*

Performance CPU

- Équations de la performance n'incluent pas aucune référence au nombre d'instructions du programme
- Le programme exécute les instructions générées par le compilateur
  - *performance globale du programme dépend du nombre de ces instructions*

Performance CPU

$$\textbf{\textit{Cycles horloge CPU}}\ pour\ un\ programme$$
$$= \ Nombre\ d'instructions$$
$$\times\ Le\ nombre\ moyen\ de\ cycles\ d'horloge\ par\ instruction$$

$$Le\ nombre\ moyen\ de\ cycles\ d'horloge\ par\ instruction$$
$$= \textbf{\textit{CPI}}$$

- CPI fournie le moyen de comparer les différents types de jeu d'instructions et les différentes implémentations

L’équation de performance CPU

- L’équation de performance CPU:
- $$\boldsymbol{Temps\ CPU = Nombre\ instructions \times CPI \times Temps\ cycle\ horloge}$$

- Comme la fréquence est la valeur inverse du cycle d’horloge, on peut réécrire l’équation de la manière suivante :
- $$\boldsymbol{Temps\ CPU = \frac{Nombre\ instructions \times CPI}{Fréquence\ horloge}}$$

L'équation de performance CPU

- $\boldsymbol{Temps\ CPU = Nombre\ instructions \times CPI \times Temps\ cycle\ horloge}$
- $\boldsymbol{Temps\ CPU = \frac{Nombre\ instructions \times CPI}{Fréquence\ horloge}}$
- Performance du CPU dépend de trois caractéristiques :
  - *le temps de cycle (ou fréquence)*
  - *le nombre de cycles par instruction*
  - *le nombre d'instructions*
- Le temps CPU dépend de la même manière de ces trois composantes :
  - *Amélioration de 10% de n'importe laquelle des trois conduit à une amélioration de 10% du temps CPU*

L'équation de performance CPU

- Temps CPU dépend:
  - *le temps de cycle (ou fréquence)*
  - *le nombre de cycles par instruction*
  - *le nombre d'instructions*
- Aucune de ces caractéristiques ne peut être modifiée indépendamment des autres
- Les technologies de base impliquées dans les modifications sont aussi interdépendantes :
  - *Temps de cycle – Technologie matérielle et organisation*
  - *CPI – Structure et architecture du jeu d'instructions*
  - *Nombre d'instructions – Architecture du jeu d'instructions et technologie des compilateurs.*

Exemple

- Un concepteur de compilateur explore les deux séquences de code. Le concepteur du matériel fournit l'information suivante :

CPI pour chaque classe d'instructions			
	A	B	C
CPI	1	2	3

Exemple

- Le développeur de compilateur considère deux séquences d'instructions à générer pour un énoncé en langage de haut niveau :

Séquence de code	Nombre d'instructions pour chaque classe d'instructions		
	A	B	C
Séquence de code #1	2	1	2
Séquence de code #2	4	1	1

Exemple

- Questions à répondre :

1. Quelle séquence exécute plus de code ?
2. Quelle séquence est plus rapide ?
3. Quel est le CPI de chaque séquence ?

Exemple, réponses

Séquence de code	Nombre d'instructions pour chaque classe d'instructions		
	A	B	C
Séquence de code #1	2	1	2
Séquence de code #2	4	1	1

1. Quelle séquence exécute plus de code ?

Séquence #1 exécute 2 + 1 + 2 = 5 instructions

Séquence #2 exécute 4 + 1 + 1 = 6 instructions

Exemple, réponses

2. Quelle séquence est plus rapide ?

Calculons le nombre total de cycles CPU pour savoir quelle séquence est plus rapide en utilisant la formule suivante : $Cycles\ CPU = \sum_{i=1}^{n} NI_i \times CPI_i$

- NI_i représente le nombre de fois où l'instruction *i* est exécutée dans un programme et CPI_i le nombre moyen de cycles pour l'instruction *i*.

$$Cycles\ CPU_1 = (2 \times 1) + (1 \times 2) + (2 \times 3)$$
$$= 2 + 2 + 6 = 10\ cycles$$
$$Cycles\ CPU_2 = (4 \times 1) + (1 \times 2) + (1 \times 3)$$
$$= 4 + 2 + 3 = 9\ cycles$$

Exemple, réponses

3. Quel est le CPI de chaque séquence ?

$$CPI = \frac{Cycles\ CPU\ pour\ un\ programme}{Nombre\ instructions}$$

$$CPI_1 = \frac{Cycles\ CPU\ pour\ un\ programme_1}{Nombre\ instructions_1} = \frac{10}{5} = 2$$

$$CPI_2 = \frac{Cycles\ CPU\ pour\ un\ programme_2}{Nombre\ instructions_2} = \frac{9}{6} = 1.5$$

Mesures de base des différentes composantes de performance aux différents niveaux

- $$Temps = \frac{Secondes}{Programme} = \frac{Instructions}{Programme} \times \frac{Cycles\ horloge}{Instruction} \times Secondes/Cycle$$

Composantes de performance	Unités de mesure
Temps CPU pour un programme	Secondes d'exécution pour le programme
Compteur des instructions d'un programme	Le nombre d'instructions d'un programme
CPI	Le nombre moyen de cycles d'horloge par instruction
Cycle d'horloge	Secondes par cycle

Mesurer trois caractéristiques

- $$Temps = \frac{Secondes}{Programme} = \frac{Instructions}{Programme} \times \frac{Cycles\ horloge}{Instruction} \times Secondes/Cycle$$
  - Temps de cycle (ou la fréquence), le nombre de cycles par instruction et le nombre d'instructions ?
  - Temps d'exécution CPU – exécuter le programme en mesurant le temps
  - Cycle d'horloge est publié dans la documentation de l'ordinateur
  - Nombre d'instructions ?
  - CPI ?

Mesurer trois caractéristiques

- $$Temps = \frac{Secondes}{Programme} = \frac{Instructions}{Programme} \times \frac{Cycles\ horloge}{Instruction} \times Secondes/Cycle$$
  - Nombre d'instructions
    - *Logiciels de profilage*
    - *Simulateur de l'architecture*
    - *Compteurs du matériel inclus dans la plupart des processeurs qui enregistrent plusieurs mesures incluant le nombre des instructions exécutées, le CPI moyen, et les sources de perte de performance*
  - **Pour faire correctement la comparaison de deux machines, il faut prendre en considération tout les 3 facteurs qui forment le temps d'exécution**

Comprendre la performance des programmes

- $$Temps = \frac{Secondes}{Programme} = \frac{Instructions}{Programme} \times \frac{Cycles\ horloge}{Instruction} \times Secondes/Cycle$$

  - Performance d'un programme dépend de la combinaison des facteurs suivants:
    - *Algorithmes utilisés*
    - *Ensemble de logiciels utilisés lors de création et de traduction du programme en code machine*
    - *Efficacité d'exécution des instructions générées par un ordinateur*

Comprendre la performance des programmes

Composante (Matériel ou Logiciel)	Qu'est-ce qui est affecté ?	De quelle façon ?
Algorithme	Nombre d'instructions exécutées, CPI	L'algorithme détermine le nombre d'instructions exécutées dans un programme. L'algorithme pourrait affecter le CPI en favorisant les différents types d'instructions (lentes ou rapides).
Langage de programmation	Nombre d'instructions exécutées, CPI	Le code est traduit en un ensemble d'instructions influençant de cette manière le nombre d'instructions exécutées par le CPU. Le langage de programmation peut affecter le CPI. Plus abstrait est le langage (« data abstraction », OO), plus il effectue des appels indirects, plus hautes seront les valeurs de CPI.
Compilateur	Nombre d'instructions exécutées, CPI	L'efficacité du compilateur affecte les deux choses : le nombre d'instructions exécutées et le nombre moyen de cycles par instruction car il traduit le programme en langage source vers le langage machine. Le rôle de compilateur pourrait être très complexe et le compilateur pourrait influencer le CPI de différente manière.
Jeu d'instructions	Nombre d'instructions exécutées, Fréquence d'horloge, CPI	Le jeu d'instructions affecte les 3 aspects de la performance CPU car il définit un ensemble des instructions disponibles, le coût en cycles pour chaque instruction et une fréquence globale d'horloge du processeur.

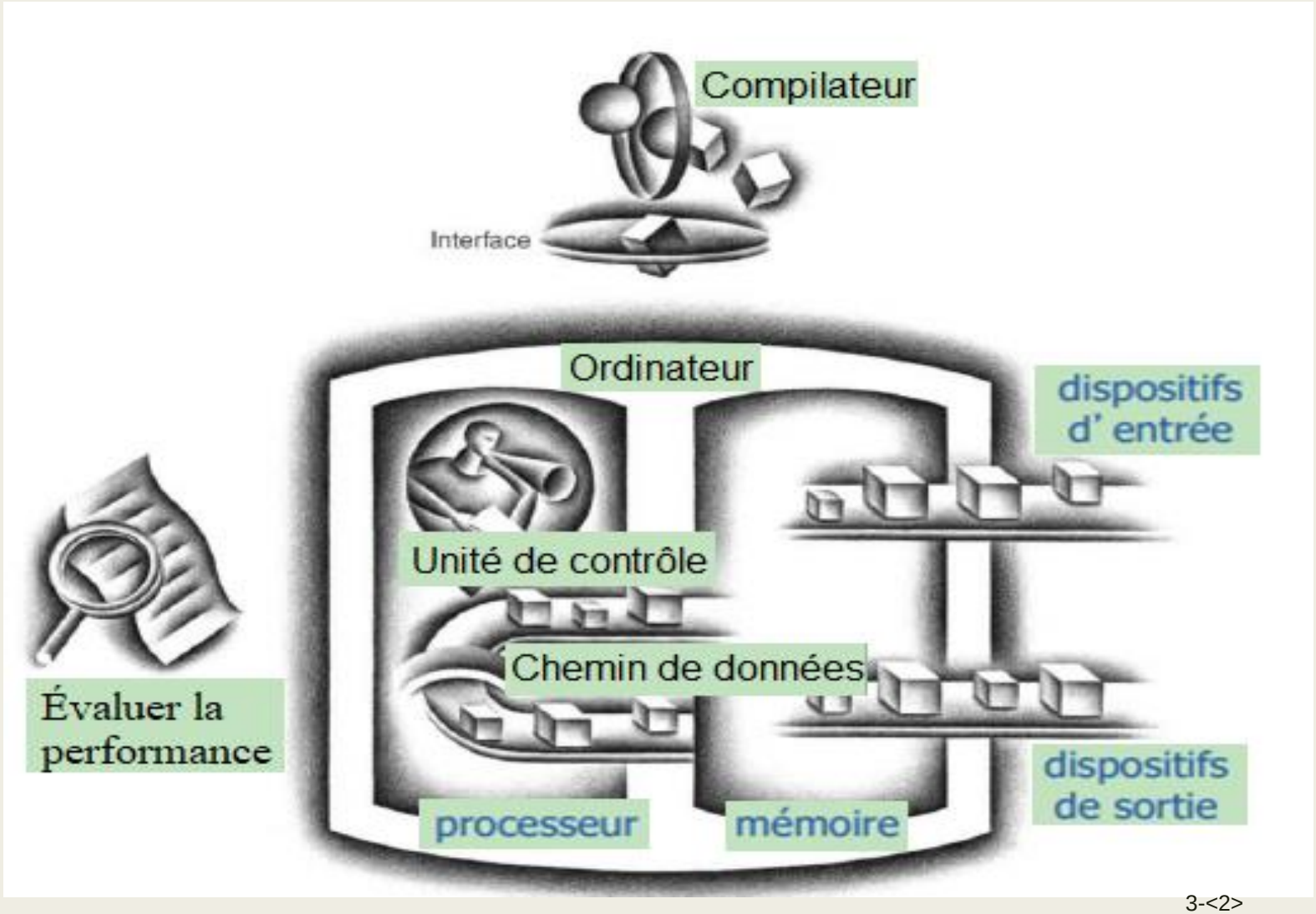
JEU D'INSTRUCTIONS

ISA
(INSTRUCTION SET ARCHITECTURE)

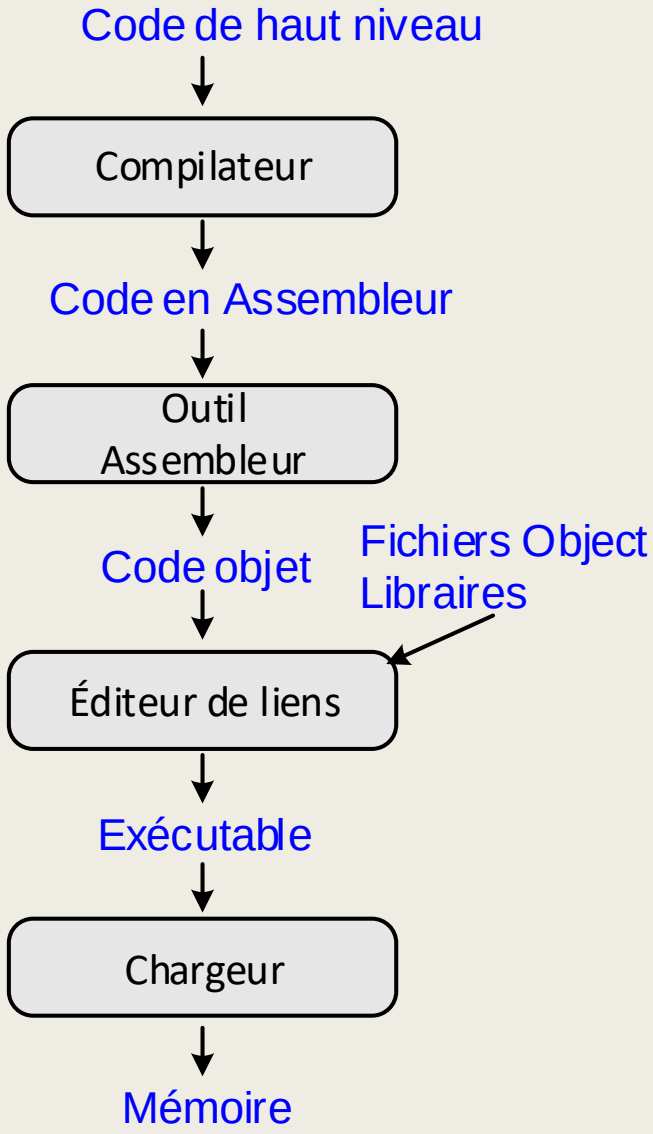
Introduction

MIPS

Introduction



Comment on compile et exécute une application ?



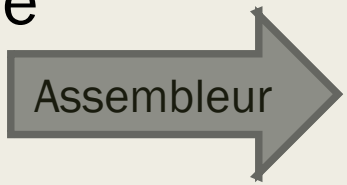
Jeu d’instructions

- Jeu d’instructions est la partie de l’ordinateur vue par le programmeur
- Le **jeu d'instructions** définit
  - *quelles sont les instructions supportées par le processeur*
  - *quels sont les registres du processeur manipulables par le programmeur (les registres architecturaux)*
  - *la façon dont ces instructions et les opérandes sont représentés en mémoire*

Langages machine

- Une instruction machine est une chaîne binaire composée essentiellement de deux parties
  - *Le code opération*
    - Désigne le type d'opération à effectuer
  - *Le reste de l'instruction*
    - Sert à designer les opérandes
      - *Les données sur lesquelles l'opération définie par le code opération doit être réalisée*

Langages d'assemblage

- Le langage d'assemblage est une variante symbolique du langage machine où certains champs binaires représentant des codes opérations ou des adresses mémoires sont remplacés par des chaînes de caractères plus aisément manipulables par l'être humain: mnémoniques
- Le langage d'assemblage représente le même jeu d'instructions que le langage machine et est spécifique de la machine
- Langage d'assemblage 

langage machine

RISC vs. CISC

- Deux grandes catégories de processeurs CISC et RISC
  - *se distinguent par la conception de leurs jeux d'instructions*
- **CISC (Complex Instruction Set Computer)**
  - *Jeu étendu d'instructions complexes*
  - *Une instruction peut exécuter plusieurs opérations élémentaires (ex : charger une valeur en mémoire, faire une opération arithmétique et ranger le résultat en mémoire)*
  - *instructions proches des constructions typiques des langages de haut niveau*
  - *Exemples : Motorola 68000, x86 Intel, AMD…*

RISC vs. CISC

- RISC (Reduced Instruction Set Computer)
  - *jeu d'instructions réduit*
  - *Une instruction exécute une seule opération élémentaire (micro-instruction)*
  - *plus uniforme (même taille, s'exécute en un cycle d'horloge)*
  - *Exemples : MIPS, Motorola 6800, PowerPC, Ultra SPARC (Sun), …*

Classes des opérations

Opérations	Exemples
Arithmétique et logique	Opérations arithmétiques sur les entiers et opérations logiques : addition, soustraction, et (ou) multiplication, division, opérations logiques sur bits
Transfert de données	Chargements et rangements
Contrôle	Branchements, sauts, appel et retour de procédure
Système	Appels système (SE), instructions de gestion de la mémoire virtuelle
Sur des flottants	Opérations sue les flottants: addition, multiplications, divisions, comparaison
Sur des décimaux	Addition décimale, multiplication décimale, conversion décimale vers caractère
Graphique	Opérations sur pixels et vertex, compression/décompression
Sur des chaînes	Transfert, comparaison, recherche

Classification des jeux d'instructions

- Selon le type de stockage interne on distingue :
  - *Architecture à pile*
    - Opérandes sont stockés implicitement sur le somme de la pile
  - *Architecture à accumulateur*
    - Un opérande est implicitement l'accumulateur
  - *Architectures à registres généraux*
    - Tous les opérandes sont explicites – soit des registres ou des cases mémoire
    - 2 types d'ordinateurs à registres
      - *Architecture registre-mémoire*
      - *Architecture chargement-rangement ou registre-registre*

MIPS (RISC)
Architecture à registres généraux de type chargement-rangement

Instructions: Addition

Code de haut niveau

```
a = b + c;
```

Code en assembleur MIPS

```
add a, b, c
```

- add: mnémonique indique quelle opération doit être effectuée
- b, c: opérandes sources
- a: opérande de destination

Instructions: Soustraction

- Soustraction est similaire à l'addition. On change seulement la mnémonique.

High-level code	**MIPS assembly code**
`a = b - c;`	`sub a, b, c`

- sub: code opération
- b, c: opérandes sources
- a: opérande de destination

Principe de conception

- Simplicité favorise régularité

- Format d'instructions est consistant
  - *Le même nombre d'opérandes*
    - 2 opérandes sources et 1 destination
- Plus facile encoder et gérer en matériel

Instructions: le code plus complexe

- Le code plus complexe est exprimé par plusieurs instructions MIPS simples

Code de haut niveau

```
a = b + c - d;
```

Code en assembleur MIPS

```
add t, b, c
sub a, t, d
```

Principe de conception

Rendre rapide le cas courant

- MIPS inclue les instructions simples et les plus fréquemment utilisées
  - *Le matériel décodant et exécutant des instructions pourrait être simple, de petite taille et rapide*
  - *Les instructions plus complexes (les moins fréquentes) pourrait être implémentées en utilisant plusieurs instructions simples*

Opérandes

```
add t, b, c
```

- Ordinateur a besoin un emplacement physique des opérandes binaires
- Opérandes binaires peuvent être stockés dans
  - Registres
  - Mémoire
  - Constantes (aussi appelées *immédiates*)

Opérandes: Registres

- Mémoire est lente
- Plusieurs architectures possèdent un petit ensemble de registres – mémoire interne aux processeurs
- MIPS-32 a 32 registres de 32-bits
- MIPS est appelé une architecture de 32-bits car MIPS manipule les données de 32 bits : mot (terminologie de RISC-V)
- Il existe une version de l'architecture MIPS de 64 bits – double mot (RISC-V)

Principe de conception 3

Tous ce qui est petit fonctionne rapidement

- MIPS possède un petit ensemble de registres
- De même que le processus de recherche parmi quelques livres sur une table nécessite moins de temps que la recherche dans une librairie, la recherche de l'information parmi les 32 registres est plus rapide que parmi 1000 registres ou dans une grande mémoire

Registres MIPS-32

Nom	Numéro du registre	Utilisation
`$0`	**0**	Une constante 0
`$at`	**1**	Temporaire de l'assembleur
`$v0-$v1`	**2-3**	Valeurs de retour de procédures
`$a0-$a3`	**4-7**	Arguments de procédures
`$t0-$t7`	**8-15**	Temporaires
`$s0-$s7`	**16-23**	Variables sauvegardées
`$t8-$t9`	**24-25**	Encore des temporaires
`$k0-$k1`	**26-27**	Temporaires du SE
`$gp`	**28**	Pointeur global
`$sp`	**29**	Pointeur de pile
`$fp`	**30**	Pointeur de cadre
`$ra`	**31**	Adresse de retour de procédure

Instructions utilisant les opérandes dans les registres

- Instruction **`add`**

Code de haut niveau

```
a = b + c;
```

MIPS

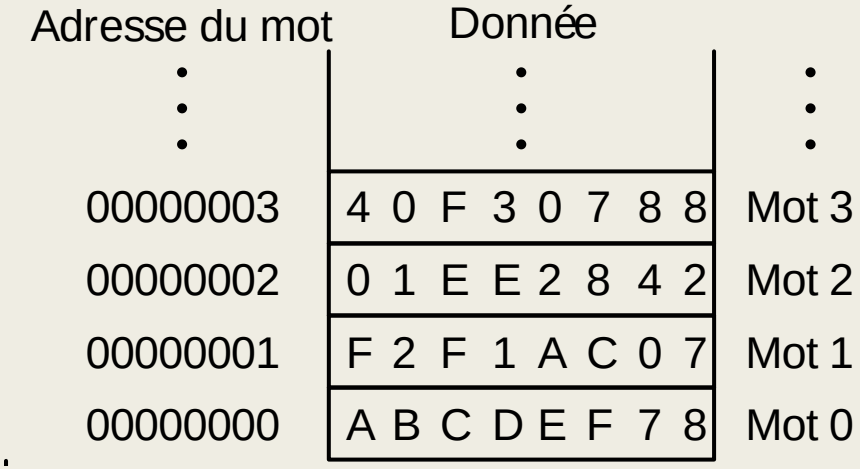
```
# $s0 = a, $s1 = b
# $s2 = c
add $s0, $s1, $s2
```

Opérande : Contenu d'une adresse mémoire

- 32 registres ne peuvent pas stocker toute l'information
- Solution: stocker les données en mémoire
- Mémoire est grande et elle peut contenir beaucoup de données, mais la mémoire est lente
  - Solution: utiliser la combinaison de 2 dispositifs de stockage
    - Registres
    - Mémoire

Mémoire adressable par mot

- Chaque mot de 32 bits possède une adresse unique

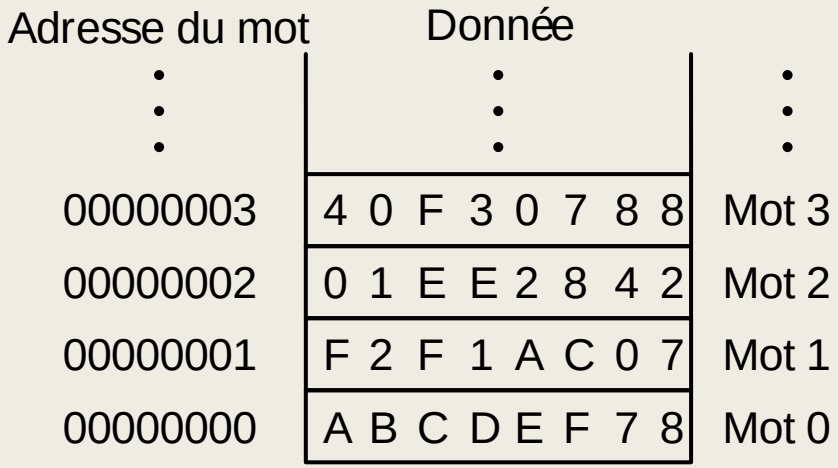


Lire la mémoire adressable par mot

- Les lectures de mémoire s'appellent chargement ou *loads*
- Mnémonique: **`lw`** – chargement mot
- **Exemple:** Charger un mot de l'adresse 1 dans le registre `$s3`
- Calcul de l'adresse mémoire:
  - Ajouter l'adresse de base (le contenu de `$0`) à un déplacement, *offset*, (1)
  - adresse = ($0 + 1) = 1
- N'importe quel registre pourrait être utilisé pour stocker l'adresse de base
- `$s3` contient la valeur 0xF2F1AC07 après l'exécution de l'instruction

Code en langage d'assemblage MIPS

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

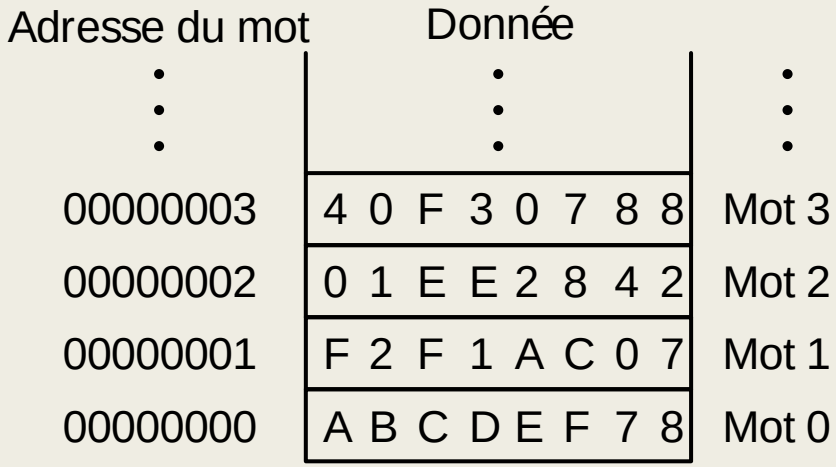


Écrire une mémoire adressable par mot

- Les écritures de mémoire s'appellent *stores*
- Mnémonique: *store word* (`sw`)
- **Exemple:** Écrire (store) le contenu du `$t4` dans la mémoire à l'adresse 8
- Déplacement peut être écrit en décimal (défaut) ou en hexadécimal (0x préfix)
- Calcul de l'adresse mémoire :
  - Ajouter l'adresse de base (le contenu de `$0`) à un déplacement, *offset*, (0x8)
  - adresse: ($0 + 0x8) = 8
- N'importe quel registre pourrait être utilisé pour stocker l'adresse de base

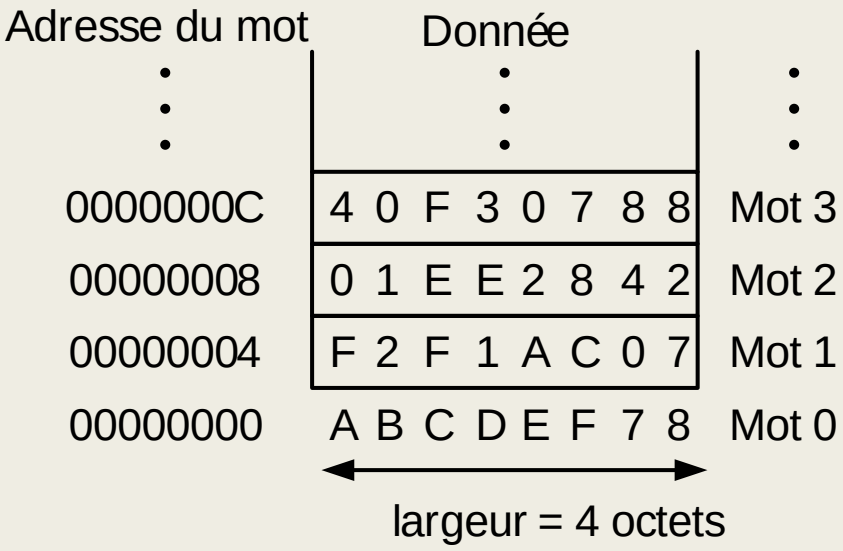
Code en langage d'assemblage MIPS

```
sw $t4, 0x7($0)  #write the value in $t4 to memory word 7
```



Mémoire adressable par octet

- Chaque octet possède une adresse unique
- On peut ranger/charger les mots ou les octets individuelles: “load byte” (`lb`) et “store byte” (`sb`)
- Chaque mot de 32-bits a 4 octets, ainsi l’adresse de mot mémoire est un multiple de 4

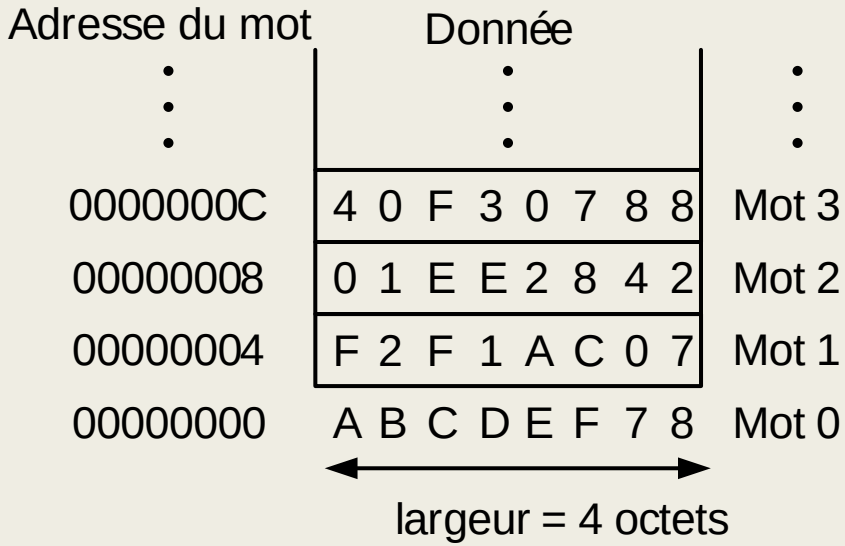


Lire une mémoire adressable par octet

- L'adresse de mot doit être multiple de 4. Par exemple,
  - L'adresse du mot mémoire 2 est 2 × 4 = 8
  - L'adresse du mot mémoire 10 est 10 × 4 = 40 (0x28)
- Charger un mot de donnée stocké dans la mémoire à l'adresse 4 dans `$s3`.
- `$s3` contient la valeur 0xF2F1AC07 après avoir terminé l'exécution.
- **MIPS est adressable par octet et non par mot**

Code en langage d'assemblage MIPS

```
lw $s3, 4($0)  # read word at address 4 into $s3
```

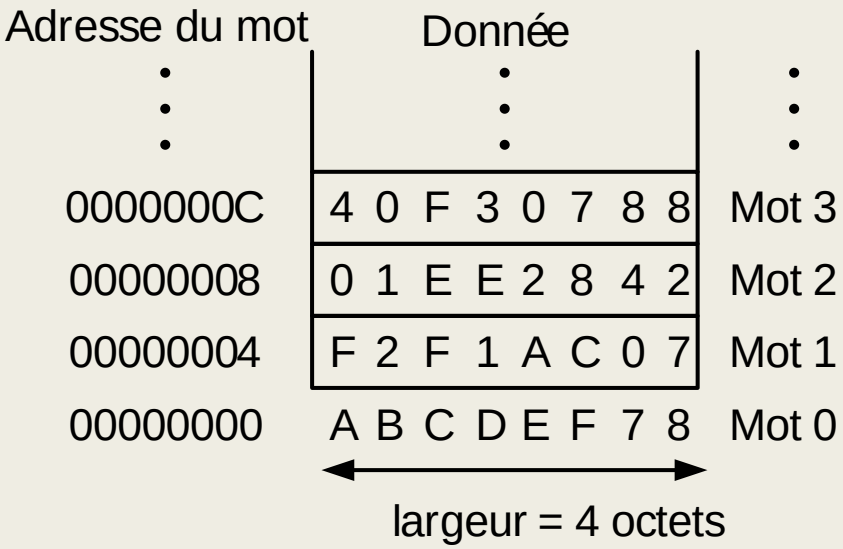


Écrire une mémoire adressable par octet

- **Exemple:** stocker le contenu du `$t7` dans la mémoire à l'adresse 0x2C (44)

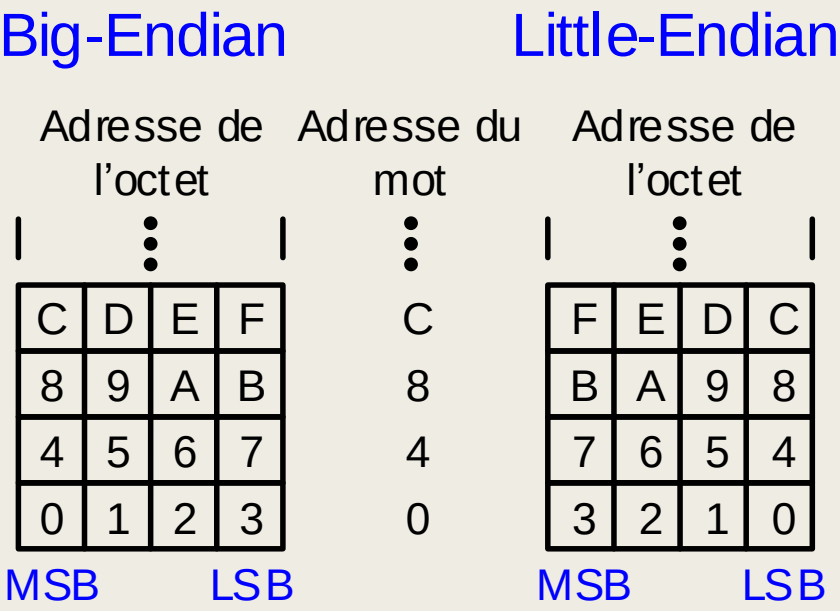
Code en langage d'assemblage MIPS

```
sw $t7, 44($0)  # write $t7 into address 44
```



Mémoire « Big-Endian » et « Little-Endian »

- Comment numéroter les octets dans un mot contenant plusieurs octets ?
- L'adresse d'un mot reste la même avec ces deux conventions
- Little-endian: Les octets sont numérotés à partir de « petit bout » ou « little end » - la partie la moins significative (LSB)
- Big-endian: Les octets sont numérotés à partir de « grand bout » ou « big end » - la partie la plus significative (MSB)



Exemple de Big- et Little-Endian

- Supposons que `$t0` contient 0x23456789. Après l'exécution du code plus bas sur un système avec la convention « big-endian », quelle valeur sera placée dans `$s0`?
- Sur un système avec la convention « little-endian » ?

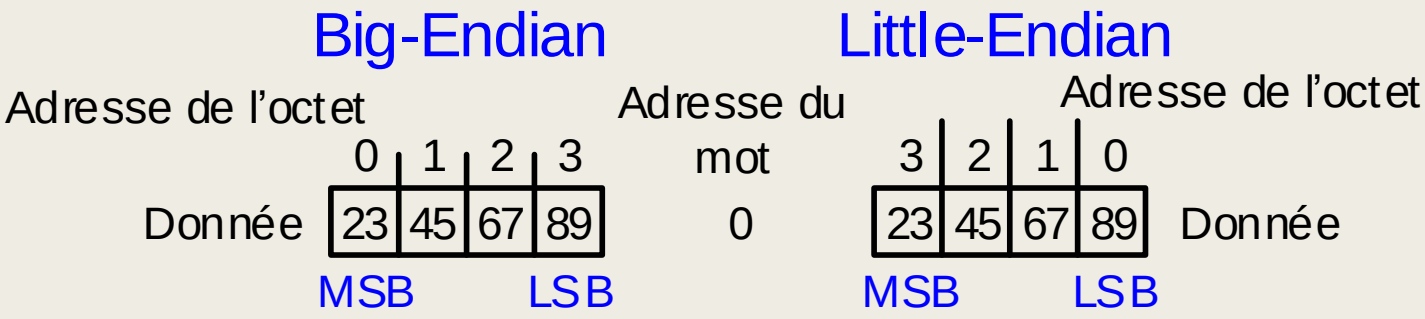
```
sw $t0, 0($0)
lb $s0, 1($0)
```

Exemple de Big- et Little-Endian

- Supposons que `$t0` contient 0x23456789. Après l'exécution du code plus bas sur un système avec la convention « big-endian », quelle valeur sera placée dans `$s0`?
- Sur un système avec la convention « little-endian » ?

```
sw $t0, 0($0)
lb $s0, 1($0)
```

- Big-endian: 0x00000045
- Little-endian: 0x00000067



Accès alignés

- L'accès à un objet de taille ***s*** octets à l'adresse ***A*** est aligné si ***A mod s = 0***

Valeur des 3 bits de poids faible de l'adresse octet								
Largeur d'objet	0	1	2	3	4	5	6	7
2 octets	Aligné		Aligné					
2 octets		Non aligné		Non aligné		Non aligné		Non aligné

Accès alignés

- Pourquoi concevoir un ordinateur avec des restrictions d'alignement ?
  - *Les désalignements provoquent des complications matérielles*
    - La mémoire est généralement alignée sur un multiple d'une frontière de mots ou de doubles mots
  - *Un accès mémoire non aligné peut donc nécessiter plusieurs accès mémoire alignés*
- Même avec des ordinateurs autorisant les accès alignés le programmes avec des accès alignés s'exécutent plus vite
- RISC-V, Intel x86 – accès sans restrictions d’alignement
- MIPS - oui

Principe de conception

Bonne conception nécessite les bons compromis

- Plusieurs formats d'instructions permettent la flexibilité
  - `add`, `sub` : *utilisent 3 opérandes registres*
  - `lw`, `sw` : *utilisent 2 opérandes registres et une constante*
- Nombre de formats reste petit

Opérandes: Constantes/Immédiates

- Les opérandes – valeurs immédiates apparaissant dans les instructions machine
- L'instruction additionner une valeur immédiate (`addi`) ajoute cette valeur au contenu du registre qui stocke une variable
- La valeur immédiate est représentée comme un nombre sur 16 bits en complément à deux

Code de haut niveau

```
a = a + 4;
b = a - 12;
```

Code en langage d'assemblage MIPS

```
# $s0 = a, $s1 = b
addi $s0, $s0, 4
addi $s1, $s0, -12
```

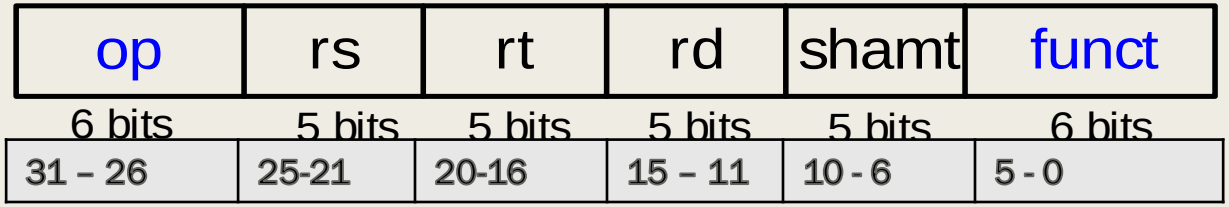
Langage machine

- Ordinateurs comprennent seulement les 1's et 0's
- Langage machine: représentation binaire des instructions
- Instructions MIPS sont encodées sur 32-bits
  - *Encore, simplicité favorise régularité: les données et les instructions sont de 32-bits*
- Trois types d'instructions:
  - ***Type R:***
  - ***Type I:***
  - ***Type J:***

Instruction de Type R

- *Type Registre*
- 3 opérandes registres:
  - `rs`, `rt`: registres sources
  - `rd`: registre destination
- Autres champs:
  - `op`: code opération ou *opcode* (0 pour les instructions de type R)
  - `funct`: le champ *fonction* détermine une opération à effectuer
  - `shamt`: ce champ est utilisé seulement pour les instructions de décalage, pour les autres - ce champ est toujours 0

Type R



Exemples, les instructions de type R

Code en assembleur

```
add $s0, $s1, $s2

sub $t0, $t3, $t5
```

Valeurs des champs

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Code Machine

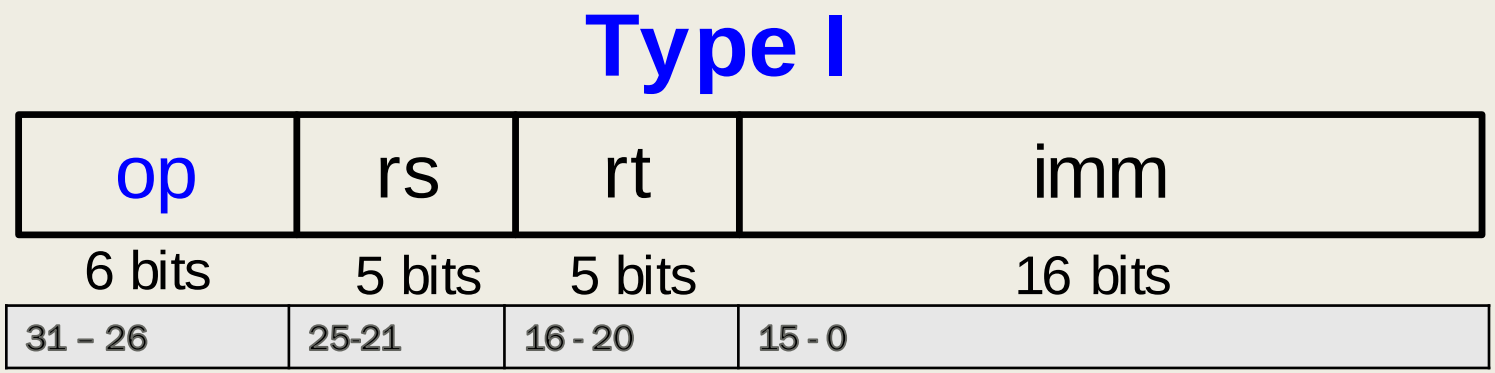
op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Remarquez l'ordre des registres dans le code assembleur et dans le code machine !

```
add rd, rs, rt
```

Type I

- *Type Immédiat*
- 3 opérandes:
  - `rs`, `rt`: opérandes registres
  - `imm`: valeur immédiate encodée sur 16-bits en complément à 2
- Autres champs:
  - `op`: code opération
  - Opération est complétement définie par opcode



I-Type, Exemples

Code assembleur

```
addi $s0, $s1, 5
addi $t0, $s3, -12
lw   $t2, 32($0)
sw   $s1,  4($t1)
```

Valeurs des champs

op	rs	rt	imm
8	17	16	5
8	19	8	-12
35	0	10	32
43	9	17	4
6 bits	5 bits	5 bits	16 bits

Notez: l'ordre différent des registres en assembleur et en code machine:

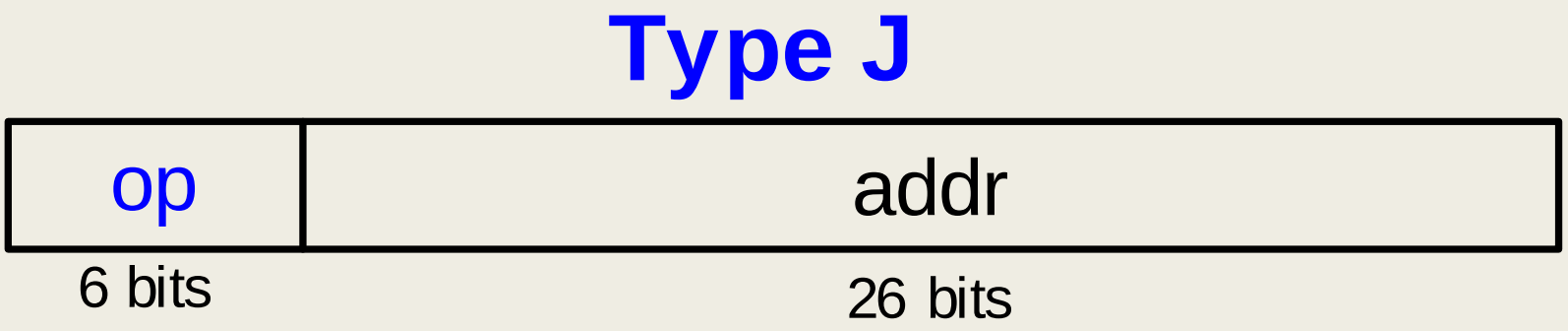
```
addi rt, rs, imm
lw   rt, imm(rs)
sw   rt, imm(rs)
```

Code machine

op	rs	rt	imm	
001000	10001	10000	0000 0000 0000 0101	(0x22300005)
001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
101011	01001	10001	0000 0000 0000 0100	(0xAD310004)
6 bits	5 bits	5 bits	16 bits	

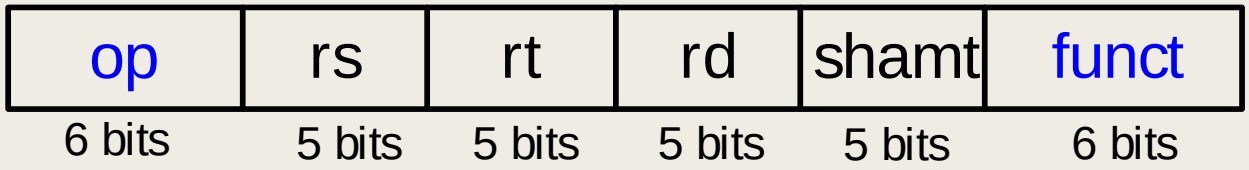
Type J

- *Type « Jump »*
- L'adresse de l'opérande de 26-bits (`addr`)
- Utilisé pour les instructions de saut et saut avec lien

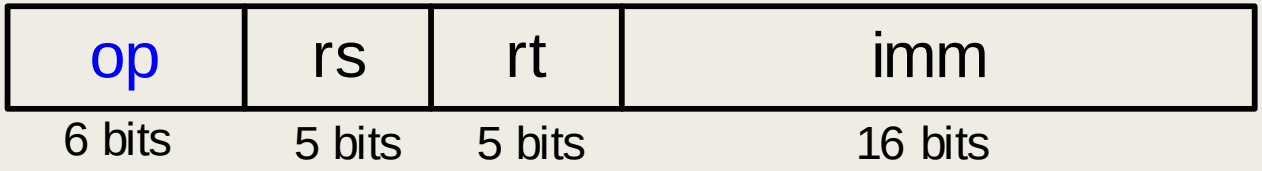


Résumé: Formats d'instructions MIPS

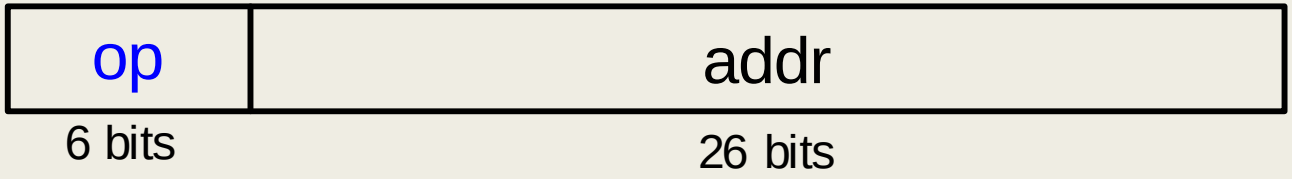
Type R



Type I

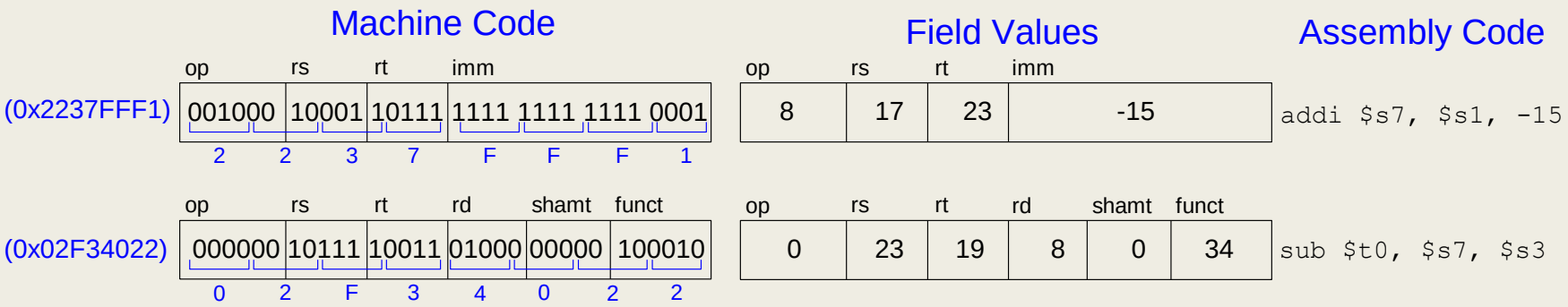


Type J



Interpréter le langage machine

- Commencer avec code opération
- Code opération indique de quelle façon découper les bits restants
- Si opcode contient tous les 0's
  - Instruction de type R
  - Bits du champs funct définissent l'opération
- Si non
  - opcode indique l'opération à effectuer



Instructions Logiques

- `and`, `or`, `xor`, `nor`
  - `and`: utiles pour *masquer* les bits
  - Masquer tous les bits sauf le dernier octet:

    0xF234012F AND 0x000000FF = 0x0000002F
  - `or`: utiles pour *combiner* les champs de bits
    - Combiner 0xF2340000 avec 0x000012BC:

      0xF2340000 OR 0x000012BC = 0xF23412BC
  - `nor`: utile pour inverser les bits:
    - A NOR $0 = NOT A
- `andi`, `ori`, `xori`
  - La valeur immédiate de 16-bits est étendues de manière non signée

Instructions Logiques, Exemples

Registres sources

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

Code Assembleur

```
and $s3, $s1, $s2
or  $s4, $s1, $s2
xor $s5, $s1, $s2
nor $s6, $s1, $s2
```

Résultat

$s3								
$s4								
$s5								
$s6								

Instructions Logiques, Exemples

Registres sources

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

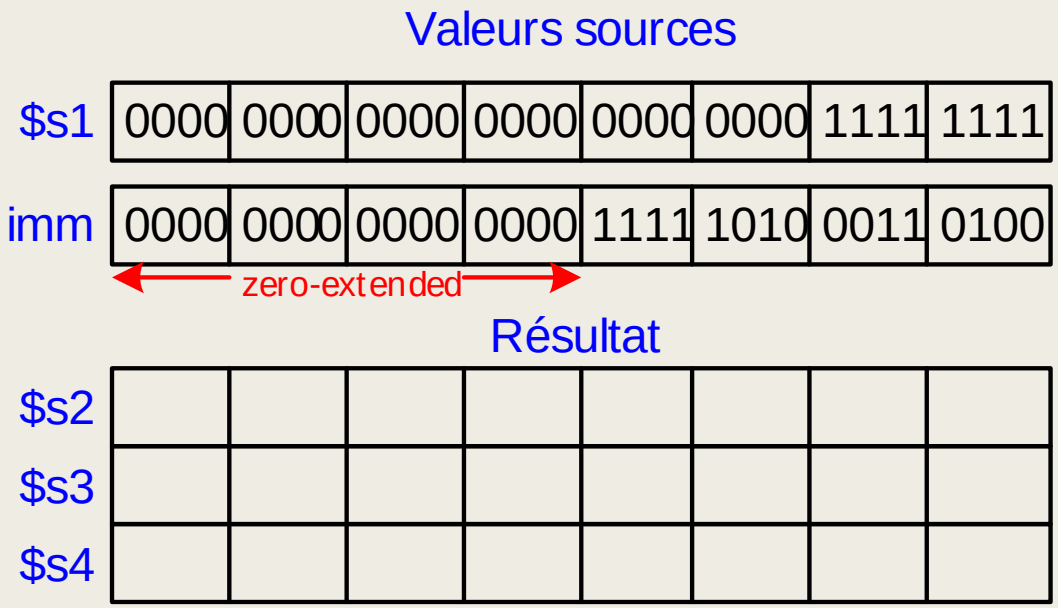
Code Assembleur

```
and $s3, $s1, $s2
or  $s4, $s1, $s2
xor $s5, $s1, $s2
nor $s6, $s1, $s2
```

Résultat

$s3	0100	0110	1010	0001	0000	0000	0000	0000
$s4	1111	1111	1111	1111	1111	0000	1011	0111
$s5	1011	1001	0101	1110	1111	0000	1011	0111
$s6	0000	0000	0000	0000	0000	1111	0100	1000

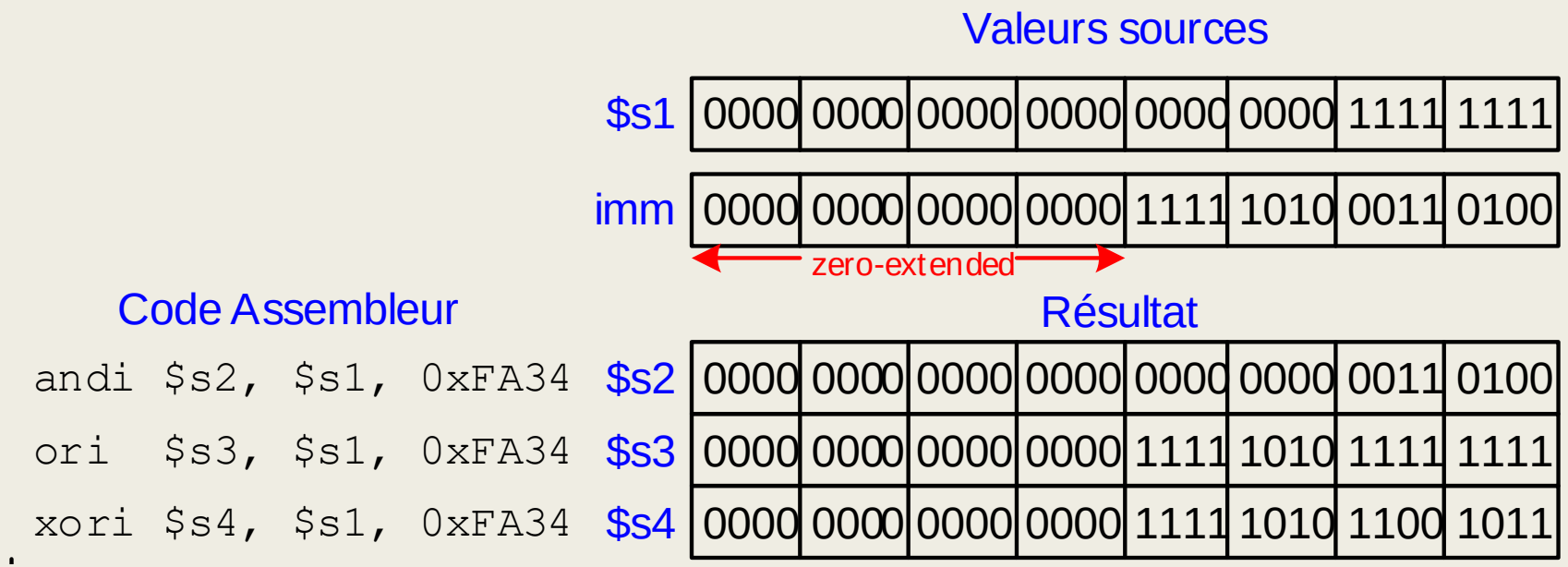
Instructions Logiques, Exemples



Code Assembleur

```
andi $s2, $s1, 0xFA34
ori  $s3, $s1, 0xFA34
xori $s4, $s1, 0xFA34
```

Instructions Logiques, Exemples



Instructions de décalage

- `sll:` décalage à gauche
  - **Exemple:** `sll $t0, $t1, 5  # $t0 <= $t1 << 5`
- `srl:` décalage logique à droite (non signé)
  - **Exemple:** `srl $t0, $t1, 5  # $t0 <= $t1 >> 5`
- `sra:` décalage arithmétique à droite (signé)
  - **Exemple:** `sra $t0, $t1, 5  # $t0 <= $t1 >>> 5`

Instructions de décalage avec variables:

- `sllv:` décalage logique à gauche avec variable

  Exemple: `sllv $t0, $t1, $t2 # $t0 <= $t1 << $t2`
- `srlv:` décalage logique à droite avec variable

  Exemple: `srlv $t0, $t1, $t2 # $t0 <= $t1 >> $t2`
- `srav:` décalage arithmétique à droite avec variable

  Exemple: `srav $t0, $t1, $t2 # $t0 <= $t1 >>> $t2`

Instructions de décalage

Code en Assembleur	op	rs	rt	rd	shamt	funct
`sll $t0, $s1, 2`	0	0	17	8	2	0
`srl $s2, $s1, 2`	0	0	17	18	2	2
`sra $s3, $s1, 2`	0	0	17	19	2	3
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Code Machine

op	rs	rt	rd	shamt	funct	
000000	00000	10001	01000	00010	000000	(0x00114080)
000000	00000	10001	10010	00010	000010	(0x00119082)
000000	00000	10001	10011	00010	000011	(0x00119883)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Génération des constantes

- Constantes de 16-bits en utilisant `addi`:

Code de haut niveau

```
// int is a 32-bit signed word
int a = 0x4f3c;
```

Code en langage d'assemblage MIPS

```
# $s0 = a
addi $s0, $0, 0x4f3c
```

- Constantes de 32-bits en utilisant load upper immediate (`lui`) et `ori`:

  (`lui` charge une constante de 16 bits dans la partie la plus significative du registre et positionne tous le reste à 0.)

Code de haut niveau

```
int a = 0xFEDC8765;
```

Code en langage d'assemblage MIPS

```
# $s0 = a
lui $s0, 0xFEDC
ori $s0, $s0, 0x8765
```

Multiplication, Division

- Registres spéciaux: `lo`, `hi`
- Multiplication 32 × 32, résultat sur 64 bits
- `mult $s0, $s1`
  - Résultat dans {`hi`, `lo`}
- Division de 32-bits, quotient de 32-bits, reste de division de 32-bits
  - `div $s0, $s1`
  - Quotient dans `lo`
  - Rest de la division dans `hi`
- Transfère des registres spéciales lo/hi
  - `mflo $s2`
  - `mfhi $s3`

Instructions de contrôle

- Permet au programme de changer le flot d'exécution d'instructions
  - *Branchement*
    - branch if equal (`beq`)
    - branch if not equal (`bne`)
  - *Instructions de saut*
    - jump (`j`)
    - jump register (`jr`)
    - jump and link (`jal`)

Branchement conditionnel(beq)

```
# Assembleur MIPS
 addi     $s0, $0, 4          # $s0 = 0 + 4 = 4
 addi $s1, $0, 1              # $s1 = 0 + 1 = 1
 sll  $s1, $s1, 2             # $s1 = 1 << 2 = 4
 beq  $s0, $s1, target        # branch is taken
 addi $s1, $s1, 1             # not executed
 sub  $s1, $s1, $s0           # not executed

target:                       # label
   add  $s1, $s1, $s0         # $s1 = 4 + 4 = 8
```

Étiquettes indiquent les emplacements d'instructions dans un programme. Elles ne pourront être les mots réservés et doivent être utilisées avec « : ».

Branchement non pris(bne)

```
# Assembleur MIPS
  addi    $s0, $0, 4          # $s0 = 0 + 4 = 4
  addi    $s1, $0, 1          # $s1 = 0 + 1 = 1
  sll     $s1, $s1, 2         # $s1 = 1 << 2 = 4
  bne     $s0, $s1, target    # branch not taken
  addi    $s1, $s1, 1         # $s1 = 4 + 1 = 5
  sub     $s1, $s1, $s0       # $s1 = 5 - 4 = 1

target:
  add     $s1, $s1, $s0       # $s1 = 1 + 4 = 5
```

Instruction de saut (j)

```
# Assembleur MIPS
   addi $s0, $0, 4              # $s0 = 4
   addi $s1, $0, 1              # $s1 = 1
   j    target                  # jump to target
   sra  $s1, $s1, 2             # not executed
   addi $s1, $s1, 1             # not executed
   sub  $s1, $s1, $s0           # not executed

 target:
   add  $s1, $s1, $s0           # $s1 = 1 + 4 = 5
```

Instruction de saut (`jr`)

```
# Assembleur MIPS
0x00002000        addi $s0, $0, 0x2010
0x00002004        jr   $s0
0x00002008        addi $s1, $0, 1
0x0000200C        sra  $s1, $s1, 2
0x00002010        lw   $s3, 44($s1)
```

Comparaison: « positionner si plus petit »

Code de haut niveau

```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for (i=1; i < 101; i = i*2) {
  sum = sum + i;
}
```

Code en assembleur MIPS

```
# $s0 = i, $s1 = sum
        addi $s1, $0, 0
        addi $s0, $0, 1
        addi $t0, $0, 101
loop:   slt  $t1, $s0, $t0
        beq  $t1, $0, done
        add  $s1, $s1, $s0
        sll  $s0, $s0, 1
        j    loop
done:
```

INF 4170 – Architecture des ordinateurs

Exercices, Jeu d'instructions MIPS

1. Traduire le code assembleur MIPS suivant en langage machine.

```
add     $t0, $s0, $s1
lw      $t0, 0x20($t7)
addi    $s0, $0, -10
addi    $t1, $0,0xff
xor     $t1, $s0, $t1
xori    $t1, $s0, 0xff
lui     $s1, 0xabcd
ori     $s1, $s1,0xef89
```

Solution

Assembleur	Code machine
`add $t0, $s0, $s1`	**02114020**
`lw $t0, 0x20($t7)`	**8de80020**
`addi $s0, $0, -10`	**2010fff6**
`addi $t1, $0,0xff`	**200900ff**
`xor $t1, $s0, $t1`	**02094826**
`xori $t1, $s0, 0xff`	**3a0900ff**
`lui $s1, 0xabcd`	**3c11abcd**
`ori $s1, $s1,0xef89`	**3631ef89**

2. Simuler l'exécution du code MIPS suivant :

```
addi    $s0, $0, -10
xori    $t1, $s0, 0xff
```

$S0	1111 1111 1111 1111 1111 1111 1111 0110	fffffff6
0xff	0000 0000 0000 0000 0000 0000 1111 1111	
$t1	1111 1111 1111 1111 1111 1111 0000 1001	ffffff09

3. Traduire le code machine en code en assembleur MIPS

Code machine	Assembleur
0x23bdfff8	**addi $sp, $sp, -8**
0xafbf0004	**sw $ra, 4($sp)**
0x0088482a	**slt $t1, $a0, $t0**
0x03e00008	**jr $ra**
0x8fbf0004	**lw $ra, 4($sp)**
0x00420018	**mult $v0,$v0**
0x014b4822	**sub $t1, $t2, $t3**

JEU D'INSTRUCTIONS

ISA

(INSTRUCTION SET ARCHITECTURE)

Appels de procédure

Définitions

- Appelant (Caller): procédure appelante (`main`)
- Appelée (Callee): procédure appelée (`sum`)

Code de haut niveau

```
void main()
{
  int y;
  y = sum(42, 7);
  ...
}

int sum(int a, int b)
{
  return (a + b);
}
```

Appels des Procédures

Convention d'appels des procédures

- Appelant:
  - *Passe les **arguments** à l'appelé.*
  - *saute à la fonction appelée*
- Appelé:
  - ***exécute une procédure***
  - ***retourne le résultat** à l'appelant*
  - ***retour au point de l'appel (instruction suivante)***
  - ***ne doit modifier les registres utilisés par l'appelant***

Conventions MIPS:

- Appel d'une procédure: jump and link (`jal`) – saut avec registre lien
- Retour de la procédure: jump register (`jr`) – registre saut
- Arguments: `$a0 – $a3`
- Valeur de retour: `$v0, $v1`

Appels des Procédures

Code de haut niveau

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

Code assembleur MIPS

```
0x00400200 main: jal  simple
0x00400204       add  $s0, $s1, $s2
...

0x00401020 simple: jr $ra
```

`jal:` saute à `simple` et sauvegarde `PC`+4 dans le registre spécial – registre d'adresse de retour (`$ra`).

Dans ce cas, `$ra` = 0x00400204 après l'exécution `jal`.

`jr $ra:` saute à l'adresse dans `$ra`, dans ce cas 0x00400204.

Arguments et Valeurs de retour

Code de haut niveau

```
int main()
{
  int y;
  ...
  y = diffofsums(2, 3, 4, 5);  // 4 arguments
  ...
}

int diffofsums(int f, int g, int h, int i)
{
  int result;
  result = (f + g) - (h + i);
  return result;                    // return value
}
```

Conventions MIPS:

- Arguments : `$a0` - `$a3`
- Valeurs de retour : `$v0`

Arguments et Valeurs de retour

Code assembleur MIPS

```
# $s0 = y

main:
   ...
   addi $a0, $0, 2     # argument 0 = 2
   addi $a1, $0, 3     # argument 1 = 3
   addi $a2, $0, 4     # argument 2 = 4
   addi $a3, $0, 5     # argument 3 = 5
   jal  diffofsums     # call procedure
   add  $s0, $v0, $0   # y = returned value
   ...

# $s0 = result
diffofsums:
   add $t0, $a0, $a1   # $t0 = f + g
   add $t1, $a2, $a3   # $t1 = h + i
   sub $s0, $t0, $t1   # result = (f + g) - (h + i)
   add $v0, $s0, $0    # put return value in $v0
   jr  $ra             # return to caller
```

Code de haut niveau

```
int main()
{
  int y;
  ...
  y = diffofsums(2, 3, 4, 5);
// 4 arguments
  ...
}

int diffofsums(int f, int g,
    int h, int i)
{
  int result;
  result = (f + g) - (h + i);
  return result;// return value
}
```

Arguments et Valeurs de retour

Assembleur MIPS

```
# $s0 = result
diffofsums:
   add $t0, $a0, $a1   # $t0 = f + g
   add $t1, $a2, $a3   # $t1 = h + i
   sub $s0, $t0, $t1   # result = (f + g) - (h + i)
   add $v0, $s0, $0    # put return value in $v0
   jr  $ra             # return to caller
```

- `diffofsums` réécrit 3 registres: `$t0`, `$t1`, et `$s0`
- `diffofsums` peut utiliser *la pile* pour stocker les contenus des registres utilisés temporairement

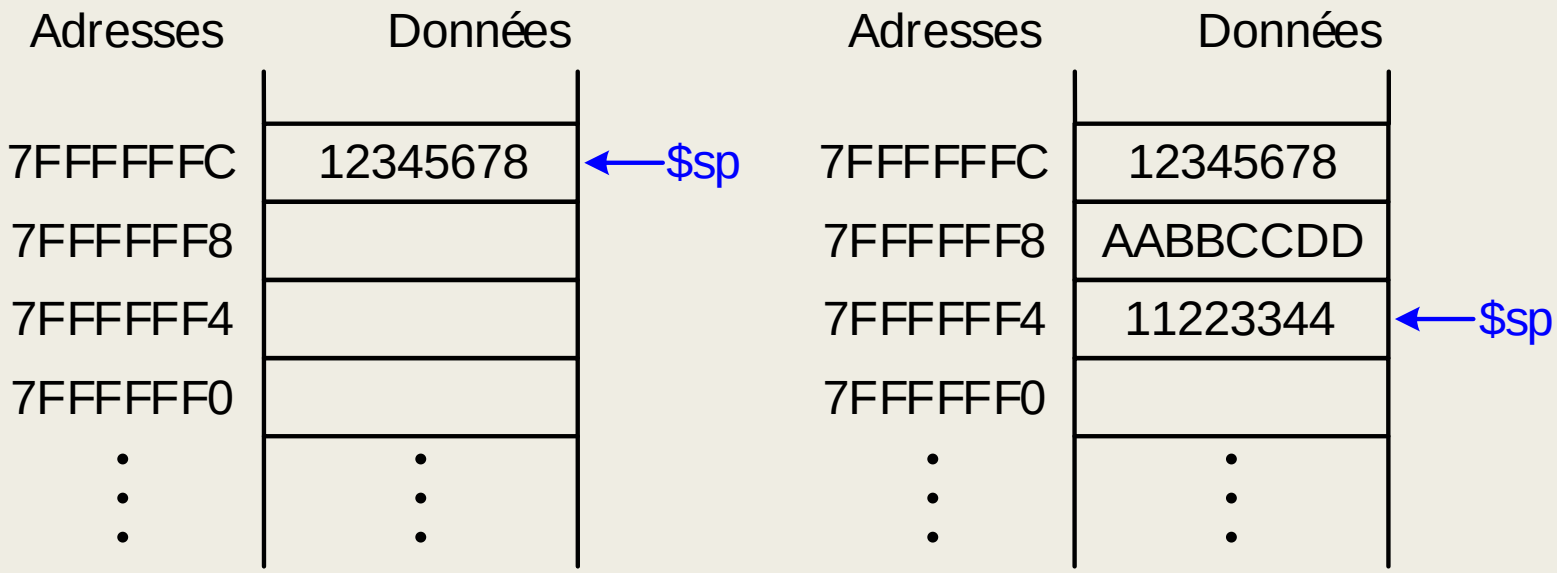
La Pile

- **Pile d'exécution** est une région mémoire qui sert à enregistrer des informations au sujet des fonctions actives dans un programme informatique

- Les données enregistrées sont organisées en structure de données de type PILE: « last-in-first-out (LIFO) », dernier arrivé, premier sorti

La Pile

- Grandit vers le bas
  - d'adresses hautes vers les adresses basses
- Pointeur de pile (Stack pointer: `$sp`)
  - pointe vers le sommet de la pile



Utilisation de la pile par les fonctions

- Fonctions appelées ne doivent pas avoir les effets de bord
- Mais, `diffofsums` réécrit 3 registres: `$t0`, `$t1`, `$s0`

```
# Code en assembleur MIPS
# $s0 = result
diffofsums:
  add $t0, $a0, $a1  # $t0 = f + g
  add $t1, $a2, $a3  # $t1 = h + i
  sub $s0, $t0, $t1  # result = (f + g) - (h + i)
  add $v0, $s0, $0   # put return value in $v0
  jr  $ra            # return to caller
```

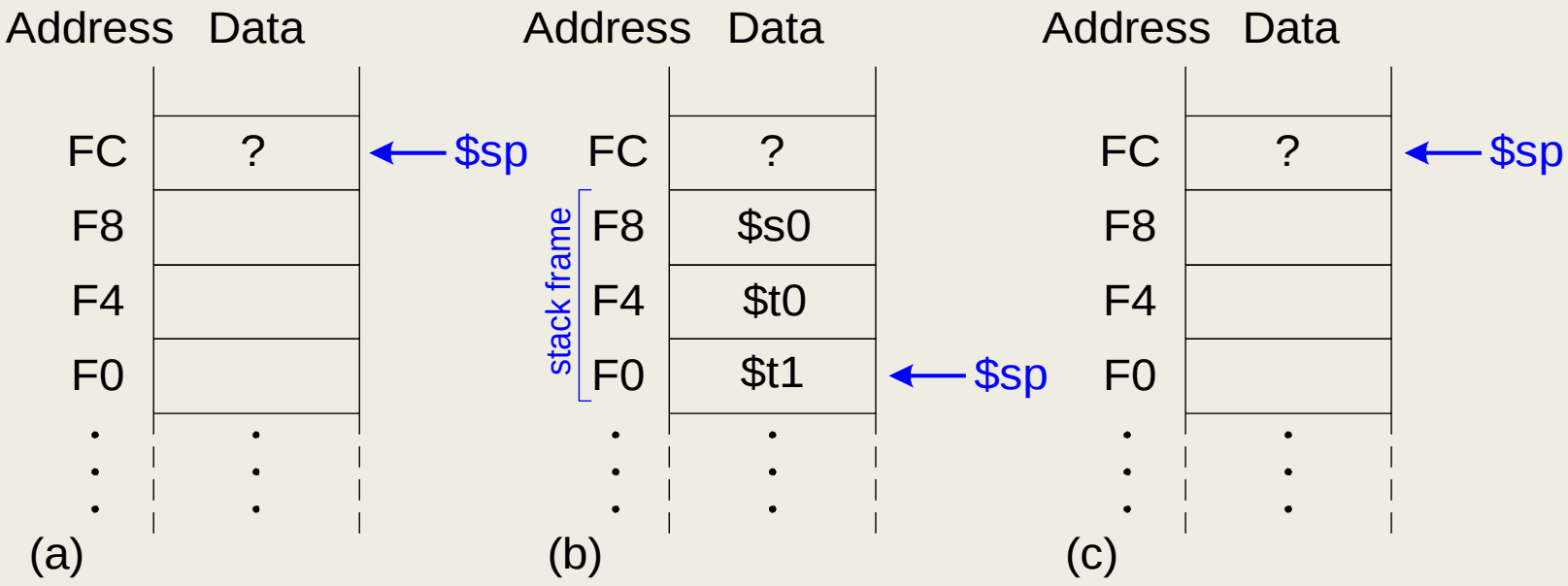
Stocker les contenus de registres dans la pile

```
diffofsums: # $s0 = result
  addi $sp, $sp, -12  #make space on stack to store 3 registers
  sw   $s0, 8($sp)    # save $s0 on stack
  sw   $t0, 4($sp)    # save $t0 on stack
  sw   $t1, 0($sp)    # save $t1 on stack
  add  $t0, $a0, $a1  # $t0 = f + g
  add  $t1, $a2, $a3  # $t1 = h + i
  sub  $s0, $t0, $t1  # result = (f + g) - (h + i)
  add  $v0, $s0, $0   # put return value in $v0
  lw   $t1, 0($sp)    # restore $t1 from stack
  lw   $t0, 4($sp)    # restore $t0 from stack
  lw   $s0, 8($sp)    # restore $s0 from stack
  addi $sp, $sp, 12   # deallocate stack space
  jr   $ra            # return to caller
```

La pile pendant l'appel de **`diffofsums`**

```
addi $sp, $sp, -12
sw   $s0, 8($sp)
sw   $t0, 4($sp)
sw   $t1, 0($sp)
```

```
lw   $t1, 0($sp)
lw   $t0, 4($sp)
lw   $s0, 8($sp)
addi $sp, $sp, 12
```



Registres

Préservés *Sauvegardés par Appelé*	Non préservés *Sauvegardés par Appelant*
`$s0 - $s7`	`$t0 - $t9`
`$ra`	`$a0 - $a3`
`$sp`	`$v0 - $v1`
Pile au dessus de `$sp`	Pile au dessous de `$sp`

Appels imbriqués

```
proc1:
  addi $sp, $sp, -4    # faire une espace sur la pile
  sw   $ra, 0($sp)     # stocker $ra sur la pile
  jal  proc2
  ...
  lw   $ra, 0($sp)     # restaurer $s0 de la pile
  addi $sp, $sp, 4     # desallouer l'espace
                       #                dans la pile
  jr  $ra              # retourner à l'appelant
```

Stocker les registres $sX sur la pile

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -4  # make space on stack to
                     # store one register
  sw  $s0, 0($sp)    # save $s0 on stack
                     # no need to save $t0 or $t1
  add $t0, $a0, $a1  # $t0 = f + g
  add $t1, $a2, $a3  # $t1 = h + i
  sub $s0, $t0, $t1  # result = (f + g) - (h + i)
  add $v0, $s0, $0   # put return value in $v0
  lw  $s0, 0($sp)    # restore $s0 from stack
  addi $sp, $sp, 4   # deallocate stack space
  jr  $ra            # return to caller
```

Procédures récursives

Code de haut niveau

```
int factorial(int n) {
  if (n <= 1)
    return 1;
  else
    return (n * factorial(n-1));
}
```

Procédures récursives

```
int factorial(int n) {
  if (n <= 1)
    return 1;
  else
    return (n * factorial(n-1));
}
```

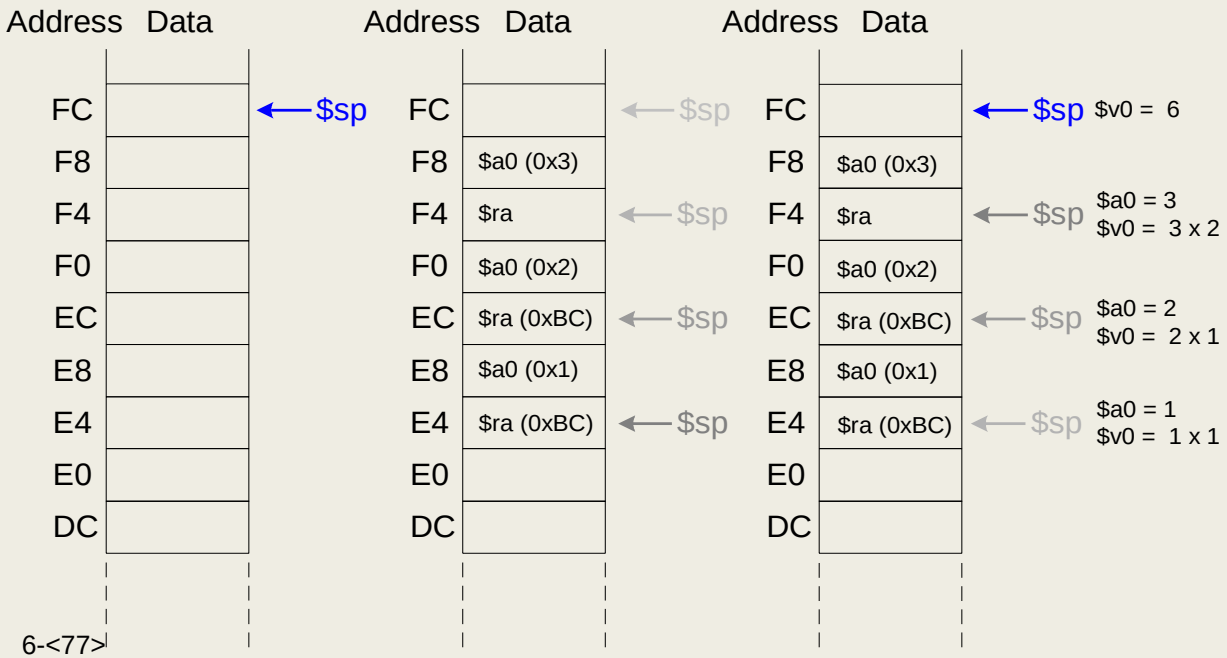
Code en assembleur MIPS

```
0x90 factorial: addi $sp, $sp, -8   # make room
0x94            sw   $a0, 4($sp)    # store $a0
0x98            sw   $ra, 0($sp)    # store $ra
0x9C            addi $t0, $0, 2
0xA0            slt  $t0, $a0, $t0  # a <= 1 ?
0xA4            beq  $t0, $0, else  # no: go to else
0xA8            addi $v0, $0, 1     # yes: return 1
0xAC            addi $sp, $sp, 8    # restore $sp
0xB0            jr   $ra            # return
0xB4       else: addi $a0, $a0, -1  # n = n - 1
0xB8            jal  factorial      # recursive call
0xBC            lw   $ra, 0($sp)    # restore $ra
0xC0            lw   $a0, 4($sp)    # restore $a0
0xC4            addi $sp, $sp, 8    # restore $sp
0xC8            mul  $v0, $a0, $v0  # n * factorial(n-1)
0xCC            jr   $ra            # return
```

Procédures récursives, Pile

```
0x90 factorial: addi $sp, $sp, -8   # make room
0x94            sw   $a0, 4($sp)    # store $a0
0x98            sw   $ra, 0($sp)    # store $ra
0x9C            addi $t0, $0, 2
0xA0            slt  $t0, $a0, $t0 # a <= 1 ?
0xA4            beq  $t0, $0, else # no: go to else
0xA8            addi $v0, $0, 1    # yes: return 1
0xAC            addi $sp, $sp, 8   # restore $sp
0xB0            jr   $ra           # return
0xB4      else: addi $a0, $a0, -1  # n = n - 1
0xB8            jal  factorial     # recursive call
0xBC            lw   $ra, 0($sp)   # restore $ra
0xC0            lw   $a0, 4($sp)   # restore $a0
0xC4            addi $sp, $sp, 8   # restore $sp
0xC8            mul  $v0, $a0, $v0 # n * factorial(n-1)
0xCC            jr   $ra           # return
```

```
int factorial(int n) {
  if (n <= 1)
    return 1;
  else
    return (n * factorial(n-1));
}
```



Résumé, Appels des Procédures

- Appelant (p.ex. main())
  - *Placer les arguments dans* `$a0-$a3`
  - *Sauvegarder les registres dont il veut utiliser le contenu après l'appel qui pourraient être modifiés par les procédures appelées (*`$ra`*,* `$t0-t9`*)*
  - `jal procédure appelée`
  - *Restaurer les registres sauvegardés*
  - *Chercher le résultat dans* `$v0`
- Appelé (diffOfSum)
  - *Sauvegarder les registres qui pourraient être utilisés par les procédures appelantes (*`$s0-$s7`*)*
  - *Exécuter le corps d'une procédure*
  - *Placer le résultat dans* `$v0`
  - *Restaurer les registres sauvegardés*
  - `jr $ra`

Modes d'adressage

De quelle façon les opérandes pourraient être adressés dans les instructions ?

1. Par les numéros des registres où ils sont stockés – Mode d'adressage ***Registres***
2. Par l'opérande lui-même – Valeur Immédiate – Mode d'adressage ***Immédiat***
3. Par l'adresse mémoire utilisant l'adresse de base dans le calcul de l'adresse effective de l'opérande – Mode d'adressage – ***Adresse de Base***
4. Opérande est le registre PC (opérations de branchement conditionnel) – Mode d'adressage ***Relatif au PC***
5. Opérande est le registre PC (opérations de saut) – Mode d'adressage ***Pseudo Direct***

Modes d'adressage

Registres

- Opérandes se trouvent dans les registres
  - *Exemple:* `add $s0, $t2, $t3`
  - *Exemple:* `sub $t8, $s1, $0`

Immédiat ou littéral

- Valeur directement encodée dans l'instruction sur 16-bits est utilisée comme opérande
  - *Exemple:* `addi $s4, $t5, -73`
  - *Exemple:* `ori  $t3, $t7, 0xFF`

Modes d'adressage

Adresse de Base

L'adresse de l'opérande cellule mémoire est calculée :

Le contenu de l'adresse de base + la valeur immédiate sur 16 bits élargie de manière signée sur 32 bits

- *Exemple:* `lw  $s4, 72($0)`
  - Adresse = R[`$0] + 72`
- *Exemple:* `sw  $t2, -24($t1)`
  - Adresse = R[`$t1] - 24`

Modes d'adressage

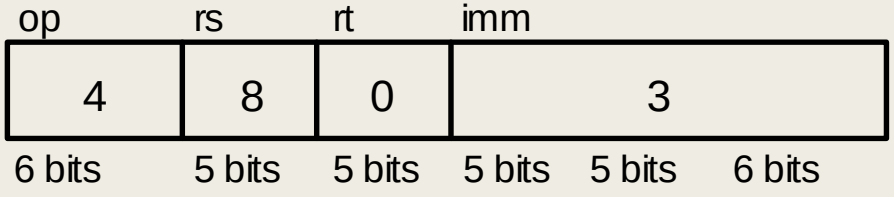
Relatif au PC

```
0x10          beq     $t0, $0, else
0x14          addi    $v0, $0, 1
0x18          addi    $sp, $sp, i
0x1C          jr      $ra
0x20  else:   addi    $a0, $a0, -1
0x24          jal     factorial
```

Code en Assembleur

```
beq $t0, $0, else
(beq $t0, $0, 3)
```

Valeurs des champs



Modes d'adressage

j, jal, jalr, jr

Pseudo Direct : j et jal *Registres : jalr et jr*

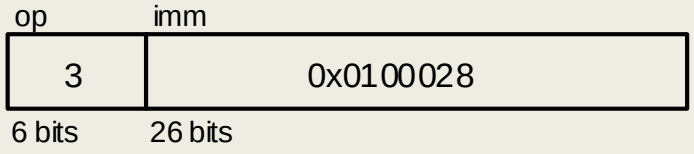
```
0x0040005C          jal   sum
...
0x004000A0  sum:    add   $v0, $a0, $a1
```

JTA 0000 0000 0100 0000 0000 0000 1010 0000 (0x004000A0)

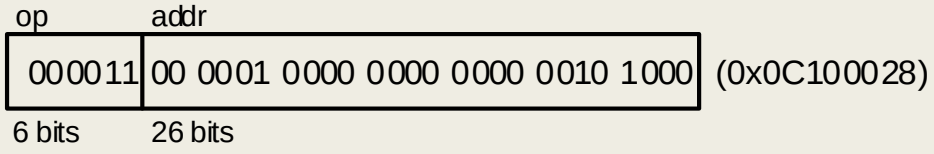
26-bit addr 0000 0000 0100 0000 0000 0000 1010 0000 (0x0100028)

0 1 0 0 0 2 8

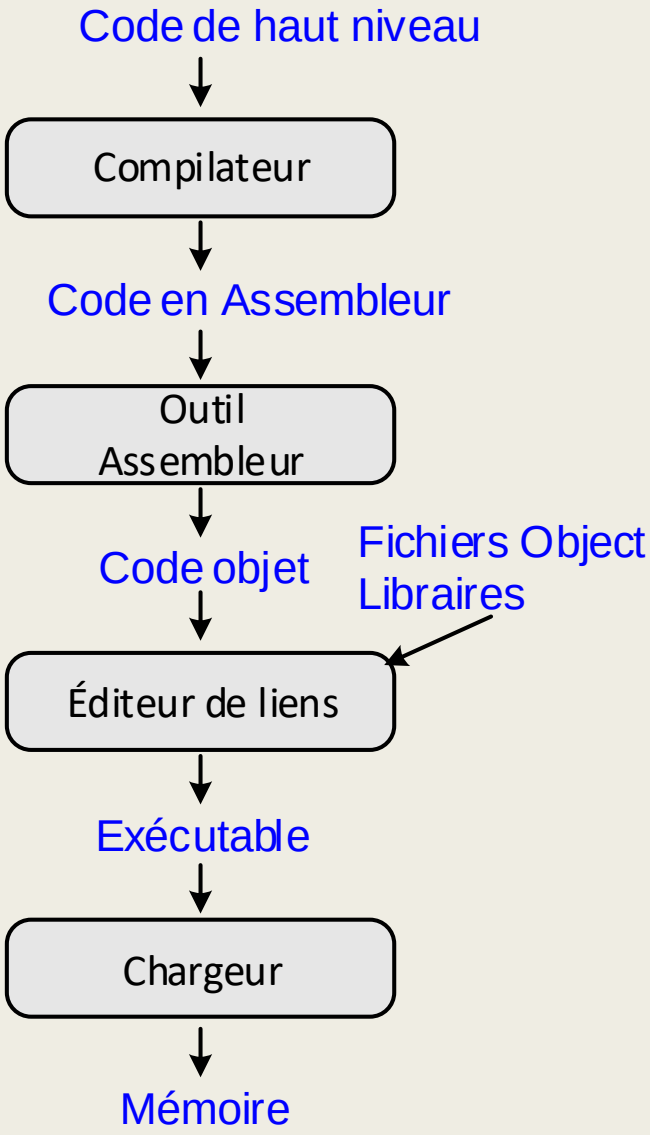
Valeurs des champs



Code Machine



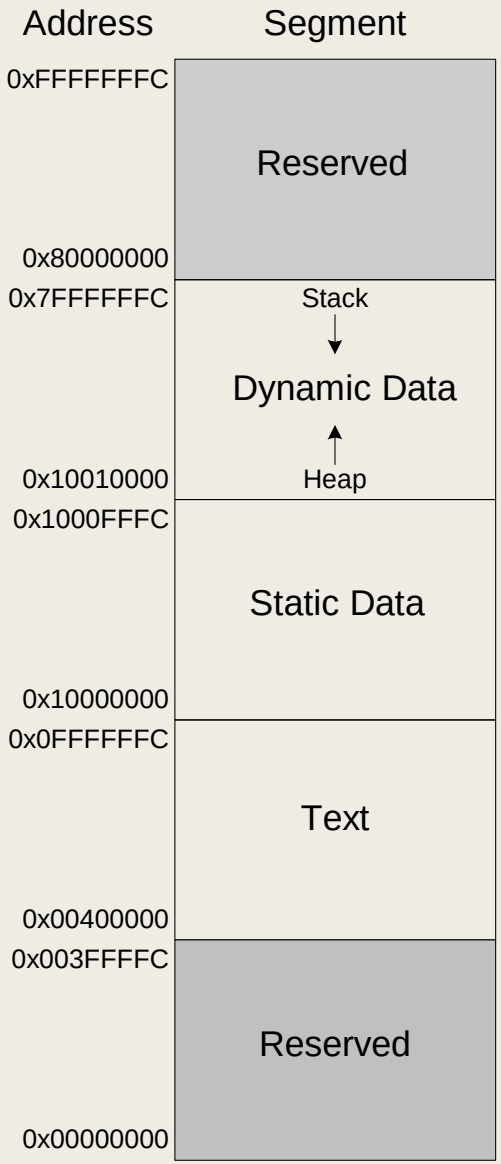
Comment on compile et exécute une application ? MIPS ?



Quelle information doit être stockée dans la mémoire ?

- Instructions (aussi appelées *text*)
- Données (aussi appelées *data*)
  - Globales/statiques: allouées avant l'exécution du programme
  - Dynamiques: allouées par le program en cours d'exécution
- Quelle est la taille de la mémoire ?
  - Au plus 2^{32} = 4 giga octets (4 GB)
  - De 0x00000000 à 0xFFFFFFFF

Partition RAM, architecture MIPS



Exemple, Programme: code en C

```
int f, g, y;  // global variables
int main(void) {
  f = 2;
  g = 3;
  y = sum(f, g);
  return y;
}
int sum(int a, int b) {
  return (a + b);
}
```

Exemple, Programme: code en MIPS

```
int f, g, y;  // global

int main(void)

{

  f = 2;

  g = 3;

  y = sum(f, g);

  return y;

}

int sum(int a, int b) {

  return (a + b);

}
```

```
.data
f:
g:
y:
.text
main:
  addi $sp, $sp, -4   # stack frame
  sw   $ra, 0($sp)    # store $ra
  addi $a0, $0, 2     # $a0 = 2
  sw   $a0, f         # f = 2
  addi $a1, $0, 3     # $a1 = 3
  sw   $a1, g         # g = 3
  jal  sum            # call sum
  sw   $v0, y         # y = sum()
  lw   $ra, 0($sp)    # restore $ra
  addi $sp, $sp, 4    # restore $sp
  jr   $ra            # return to OS
sum:
  add  $v0, $a0, $a1  # $v0 = a + b
  jr   $ra            # return
```

Exemple, Programme: Table des symboles

Symbole	Adresse

Exemple, Programme: Table des symboles

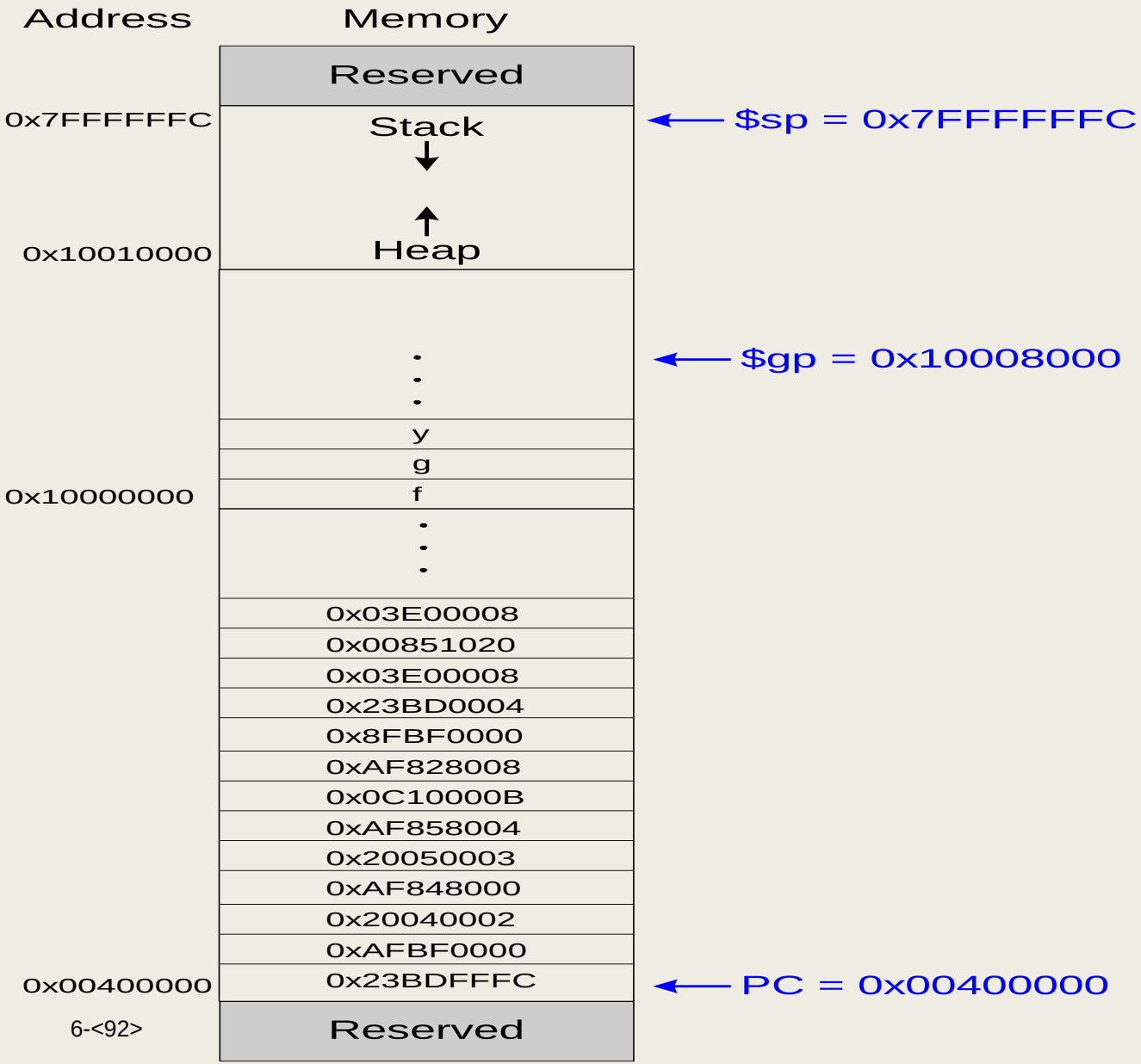
Symbole	Adresse
`f`	0x10000000
`g`	0x10000004
`y`	0x10000008
`main`	0x00400000
`sum`	0x0040002C

Exemple, Programme: Exécutable

Executable file header	Text Size	Data Size
	0x34 (52 bytes)	0xC (12 bytes)
Text segment	**Address**	**Instruction**
	0x00400000	0x23BDFFFC
	0x00400004	0xAFBF0000
	0x00400008	0x20040002
	0x0040000C	0xAF848000
	0x00400010	0x20050003
	0x00400014	0xAF858004
	0x00400018	0x0C10000B
	0x0040001C	0xAF828008
	0x00400020	0x8FBF0000
	0x00400024	0x23BD0004
	0x00400028	0x03E00008
	0x0040002C	0x00851020
	0x00400030	0x03E0008
Data segment	**Address**	**Data**
	0x10000000	f
	0x10000004	g
	0x10000008	y

```
addi $sp, $sp, -4
sw   $ra, 0 ($sp)
addi $a0, $0, 2
sw   $a0, 0x8000 ($gp)
addi $a1, $0, 3
sw   $a1, 0x8004 ($gp)
jal  0x0040002C
sw   $v0, 0x8008 ($gp)
lw   $ra, 0 ($sp)
addi $sp, $sp, -4
jr   $ra
add  $v0, $a0, $a1
jr   $ra
```

Exemple, Programme: Mémoire



Autres opérations MIPS

- Pseudo instructions
- Instructions signées et non signées
- Instructions flottantes

Pseudo instructions, Exemples

Pseudo instruction	Instructions MIPS
`li $s0, 0x1234AA77`	`lui $s0, 0x1234` `ori $s0, 0xAA77`
`mul $s0, $s1, $s2`	`mult $s1, $s2` `mflo $s0`
`clear $t0`	`add $t0, $0, $0`
`move $s1, $s2`	`add $s2, $s1, $0`
`nop`	`sll $0, $0, 0`

Instruction signées et non signées

- Addition et soustraction
- Multiplication et division
- Plus petit que

Addition et Soustraction

- Signées: `add`, `addi`, `sub`
  - La même implémentation matériel pour les deux versions: signée et non signée
  - Différence: pour la version signée, le processeur signale une exception sur le dépassement de capacité arithmétique
- Non signées: `addu`, `addiu`, `subu`

Multiplication et Division

- Signée: `mult`, `div`
- Non signée: `multu`, `divu`
- 0xFFFFFFFF
  - Un nombre positif très grand
  - 0xFFFFFFFF = -1 – selon la convention complément à 2 pour les représentations des entiers signés

Plus petit que (Set Less Than)

- Signées: slt, slti
- Non signées: sltu, sltiu

Opérations flottantes

- Coprocesseur des instructions en virgule flottante (Coprocesseur 1)
- 32 registres d'instructions en virgule flottante de 32-bit bits (`$f0` - `$f31`)
- Valeurs en double précision sont stockées dans les deux registres d'instructions en virgule flottante p.e., `$f0` et `$f1`, `$f2` et `$f3`, etc.
  - Donc, les registres d'instructions en virgule flottante de double précision : `$f0`, `$f2`, `$f4`, etc.

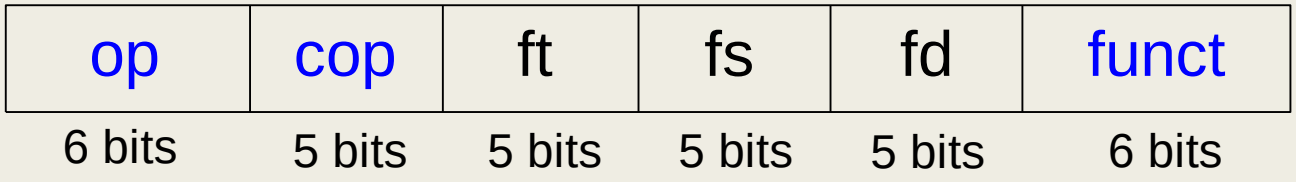
Instructions en virgule flottante

Nom	Numéro de registre	Utilisation
`$fv0 - $fv1`	0, 2	Valeurs de retour
`$ft0 - $ft3`	4, 6, 8, 10	Variables temporaires
`$fa0 - $fa1`	12, 14	Arguments de procédures
`$ft4 - $ft5`	16, 18	Variables temporaires
`$fs0 - $fs5`	20, 22, 24, 26, 28, 30	Variables sauvegardées

Format F-Type

- `Opcode` = 17 (010001_2)
- Simple précision:
  - `cop` = 16 (010000_2)
  - `add.s`, `sub.s`, `div.s`, `neg.s`, `abs.s`, etc.
- Double précision:
  - `cop` = 17 (010001_2)
  - `add.d`, `sub.d`, `div.d`, `neg.d`, `abs.d`, etc.
- 3 opérandes registres:
  - `fs`, `ft`: opérandes sources
  - `fd`: opérandes destinations

F-Type



Instructions en virgule flottante de branchement

- Positionner/Effacer le flag de condition : `fpcond`
  - Égalité: `c.eq.s`, `c.eq.d`
  - Plus petit: `c.lt.s`, `c.lt.d`
  - Plus petit ou égale: `c.le.s`, `c.le.d`

Ex:
```
c.eq.s $f0,$f1
bc1f   fin
add.s  $f2,$f1,$f0
fin:   nop
```

- Branchement Conditionnel
  - `bc1f`: branchement si `fpcond` est FALSE
  - `bc1t`: branchement si `fpcond` est TRUE
- Loads et stores
  - `lwc1`: `lwc1 $ft1, 42($s1)`
  - `swc1`: `swc1 $fs2, 17($sp)`

INF 4170 – Architecture des ordinateurs

Exercices, Jeu d'instructions MIPS, partie 2 : instructions de contrôle, mode d'adressage

1.

a) Traduire en code machine les instructions sélectionnées.
b) Indiquer le mode d'adressage pour toutes les instructions sauf celles d'appel système (`syscall`)

Adresse	Code	Mode d'adressage
0x00400000	`main: addi  $a0, $0, 5`	Imm
0x00400004	`jal  f`	Pseudo direct
0x00400008	`add  $a0, $v0, $0  # $a0=integer to print`	Reg.
0x0040000C	`addi  $v0, $0, 1  # $v0=1 service to print int`	Imm
0x00400010	`syscall`	X
0x00400014	`addi  $v0,$0,10  # $v0=10 service to exit`	Imm.
0x00400018	`syscall`	X

	`# int f(int n){`
	`#  int res = 0;`
	`#  while( n > = 0){`
	`#  res = res + (1 << n);`
	`#  n--;`
	`#  }`
	`# }`	

0x0040001C	`f:  addi  $v0,$0,0`	Imm
0x00400020	`addi  $t0,$0,1`	Imm

0x00400024	while:	beq $a0,$0,done	Rel. au PC
0x00400028		sllv $t0,$t0,$a0	Reg.
0x0040002C		addu $v0,$t0,$v0	Reg.
0x00400030		addi $a0,$a0,-1	Imm.
0x00400034		j while	Pseudo direct
0x00400038	done:	jr $ra	Reg.

jal f

beq $a0,$0,done

j while

jr $ra

Adresse eff. Des étiquettes	Code Assembleur	Code machine base 2	Code machine base 16
f: 0x0040001C	jal f	000011 0000 0100 0000 0000 0000 0001 11	0x0C100007
done est encodée par déplacement +4	beq $a0,$0,done	000100 00100 00000 0000 0000 0000 0100	0x10800004
while: 0x00400024	j while	000010 00 0001 0000 0000 0000 0000 1001	0x08100009
	jr $ra	000000 11111 00000 00000 00000 001000	0x03e00008

2. Traduire le code machine suivant en code en langage d'assemblage MIPS :

```
                0x1084fffe
0x00400004      0x08100000
```

Code machine en base 16	Code machine en base 2	
0x1084fffe	**0001 0000 1000 0100 1111 1111 1111 1110** **-2**	**beq $a0, $a0, loop (étiquette loop est encodée par le déplacement -2)**
0x08100000	**0000 1000 0001 0000 0000 0000 0000 0000** **26 bits de l'adresse:** **00000100000000000000000000** **4 bits du PC de j : 0000 26 bits 00** **0000 0000 0100 0000 0000 0000 0000 00 00** Donc, adresse de branchement est **0x00400000**	**j étiquette qui encode l'adresse 0x00400000**

CIRCUITS LOGIQUES COMBINATOIRES

Circuits logiques

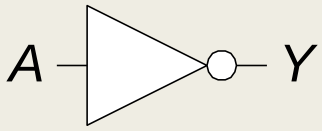
- Un circuit logique est un circuit dans lequel seules 2 valeurs logiques sont possibles
  - *0 ou 1*
- En pratique : circuit électrique (transistors) dans lequel une faible tension représente le signal 0 alors qu'une tension élevée correspond au signal 1
- Composants de base : les portes logiques qui permettent de combiner ces signaux binaires

Portes Logiques

- Une porte logique est un composant qui reçoit en entrée une ou plusieurs valeurs binaires et renvoie en sortie une unique valeur binaire en implémentant une fonction logique:
  - *inversion (NOT), AND, OR, NAND, NOR, etc.*
- Une valeur en entrée:
  - *Porte NOT, tampon*
- 2-entrées:
  - *AND, OR, XOR, NAND, NOR, XNOR*
- Plusieurs entrées

Portes logiques d'une entrée

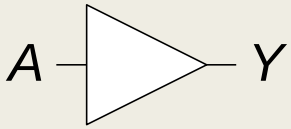
NOT



$Y = \overline{A}$

A	Y
0	1
1	0

BUF Tampon

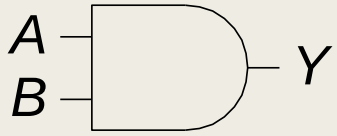


$Y = A$

A	Y
0	0
1	1

Portes logiques de deux entrées

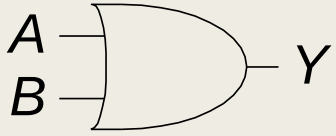
AND



$Y = AB$

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

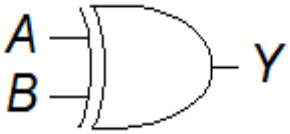
OR



$Y = A + B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

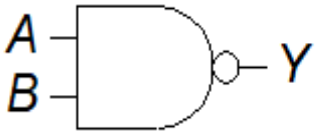
XOR



$Y = A \oplus B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

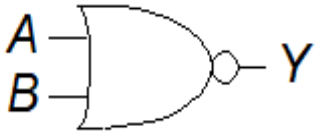
NAND



$Y = \overline{AB}$

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

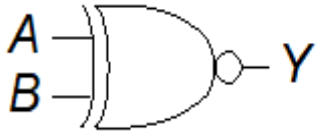
NOR



$Y = \overline{A + B}$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

XNOR



$Y = \overline{A \oplus B}$

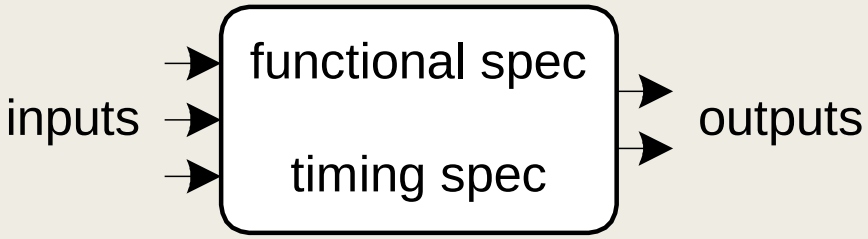
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

PORTES LOGIQUES DE DEUX ENTRÉES

Circuit logique

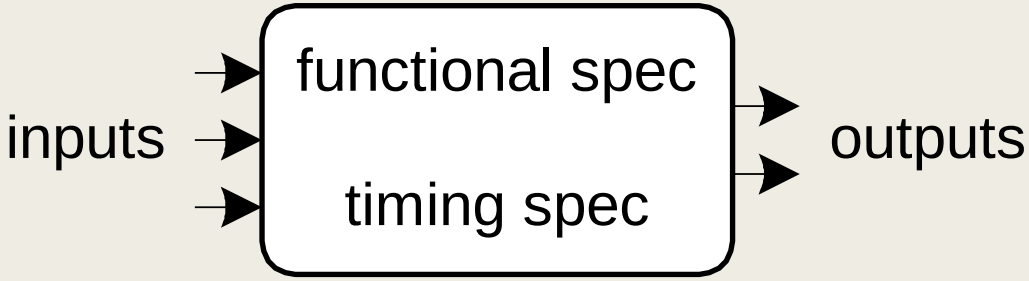
Un circuit logique est composé:

- Entrées
- Sorties
- Spécification fonctionnelle
- Spécification temporelle



Types des circuits logiques

- Circuits logiques combinatoires
  - *Sans mémoire*
  - *Sorties sont définies par les valeurs des entées actuelles*
- Circuits logiques séquentiels
  - *Possèdent une mémoire*
  - *Sorties sont définies par les entrées antérieures et actuelles*

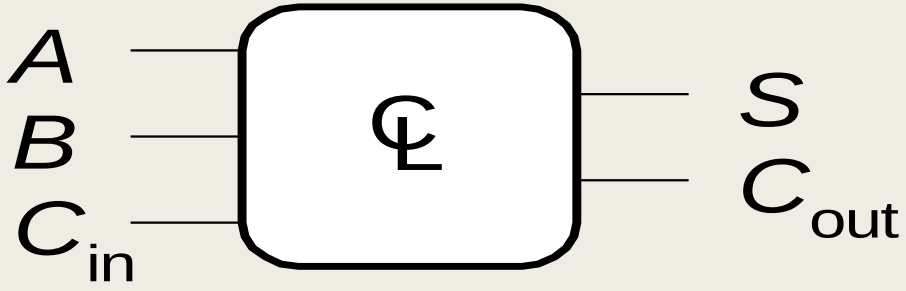


Équations Booléennes

- Pour décrire les circuits réalisables en combinant des portes logiques, on a besoin d'une algèbre opérant sur les variables 0 et 1
- Une fonction booléenne (logique) à une ou plusieurs variables est une fonction qui renvoie une valeur ne dépendant que de ces variables.
- Exemple:

$S = F(A, B, C_{in})$

$C_{out} = F(A, B, C_{in})$



$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + AC_{in} + BC_{in}$$

Table de vérité

- Une fonction logique à n variables possède seulement 2^n combinaisons d'entrées possibles
- On peut définir une fonction logique par une table à 2^n lignes donnant la valeur de la fonction pour chaque combinaison d'entrée
  - *Table de vérité de la fonction*

Notation

- $\bar{A}$ le NOT de A
- A + B le OR de A et B
- AB le AND de A et B

Somme-de-Produits (SOP)

- Toute fonction peut être décrite en spécifiant lesquelles des combinaisons d'entrée donnent 1
- Avec chaque ligne on peut associer un AND logique unique égale à 1
- On peut, donc, représenter une fonction logique comme le OU logique d'un ensemble de conditions « et » (AND) sur les combinaisons d'entrée donnant 1

A	*B*	*Y*	minterm
0	0	0	$\overline{A}\ \overline{B}$
0	1	1	$\overline{A}\ B$
1	0	0	$A\ \overline{B}$
1	1	1	$A\ B$

$$Y = \mathrm{F}(A, B) = \bar{A}B + AB$$

Produits-de-sommes (POS)

- Toutes les équations booléennes peuvent être décrites en forme POS
- Un maxterm est une somme (OU) de littéraux
- Avec chaque ligne on peut associer un maxterme unique égale à 0
- On peut, donc, représenter une fonction logique comme le ET logique de maxtermes sur les combinaisons d'entrée donnant 0

A	*B*	*Y*	maxterm
0	0	0	$A + B$
0	1	1	$A + \overline{B}$
1	0	0	$\overline{A} + B$
1	1	1	$\overline{A} + \overline{B}$

$$Y = F(A, B) = (A + B)(\overline{A} + B)$$

Simplification de l'implantation

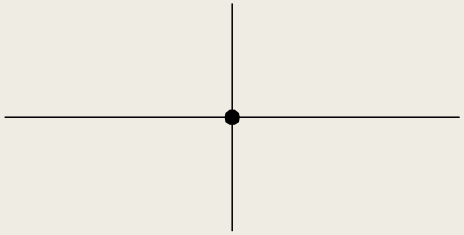
- *Deux fonctions sont équivalentes si et seulement si leurs tables de vérité sont identiques*
- *Il est intéressant d'implanter une fonction avec le moins de portes possibles*
  - Économie de place sur le processor
  - Réduction de la consommation électrique
  - Réduction du temps de parcours du signal
- *Il est possible de simplifier les formes canoniques SOP (ou POS)*

Convention dans les schémas

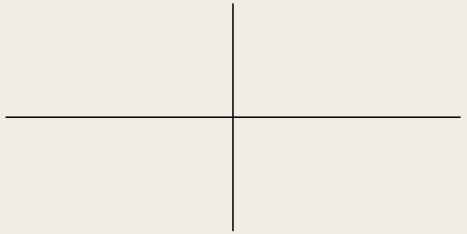
- Jonction T – toujours connexion
- Si deux fils se croisent, il n'y a pas connexion des deux fils sauf si l'intersection est matérialisée par un gros point.

wires connect at a T junction

wires connect at a dot



wires crossing without a dot do not connect



Fonctions combinatoires

- Pour réaliser des circuits logiques complexes on ne part pas des portes logiques elles-mêmes mais de sous–ensembles fonctionnels tells que:
  - *Fonctions combinatoires et arithmétiques*
  - *Horloges*

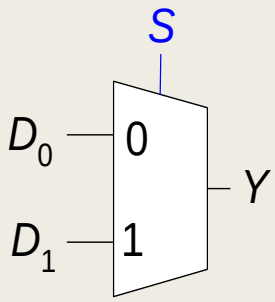
Fonctions combinatoires

- Une fonction combinatoire dans le sens général est une fonction possédant des entrées et des sorties multiples, tells que les valeurs des sorties ne dépendent que des valeurs d'entrée
- Cette classe comprend :
  - *Multiplexeurs*
  - *Décodeurs*
  - *…*

Multiplexeur (Mux)

- Entrées:
  - *2^n lignes d'entrée (données): D_0, D_1,..., D_{2^n}*
  - *N lignes de sélection: a; b; c; ...*
  - *Sortie: une seule sortie S*
- Rôle: Aiguiller la valeur de l'une des 2^n lignes d'entrée vers la sortie S
  - *La ligne d'entrée choisie est désignée grâce aux bits de sélection*

2:1 Mux

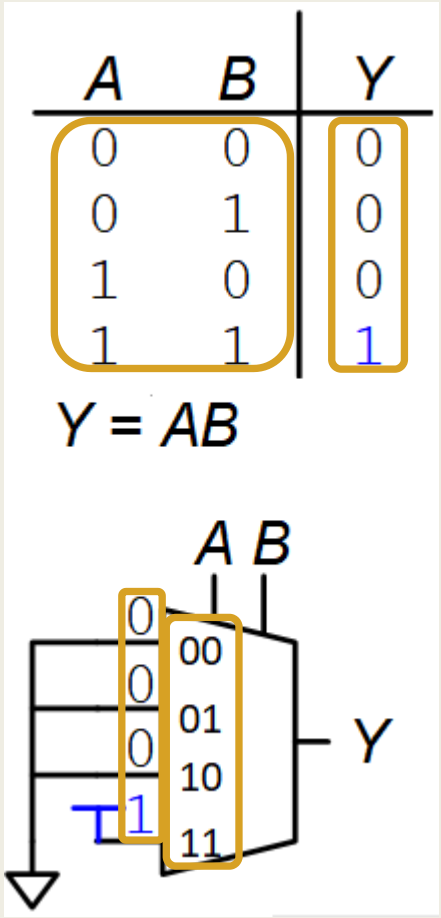


S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

S	Y
0	D_0
1	D_1

Implémentation des fonctions logiques en utilisant des MUXs

- Une **table de correspondance** (*Look-Up Table* (*LUT*)) est un terme informatique et électronique désignant une liste d'association de valeurs
- Méthode de calcul par consultation
- Utilisation de MUX comme LUT



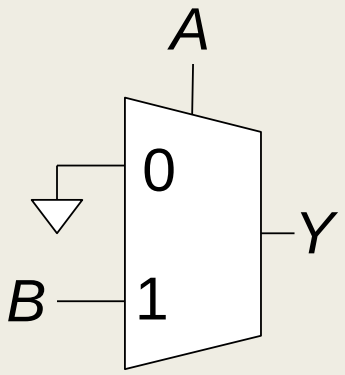
Implémentation des fonctions logiques en utilisant des MUXs

- Réduisons la taille de MUX

$Y = AB$

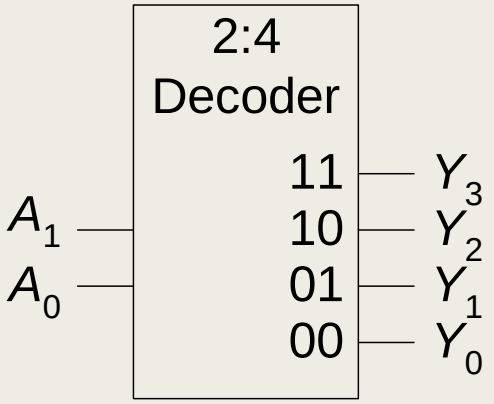
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

A	Y
0	0
1	B



Décodeurs

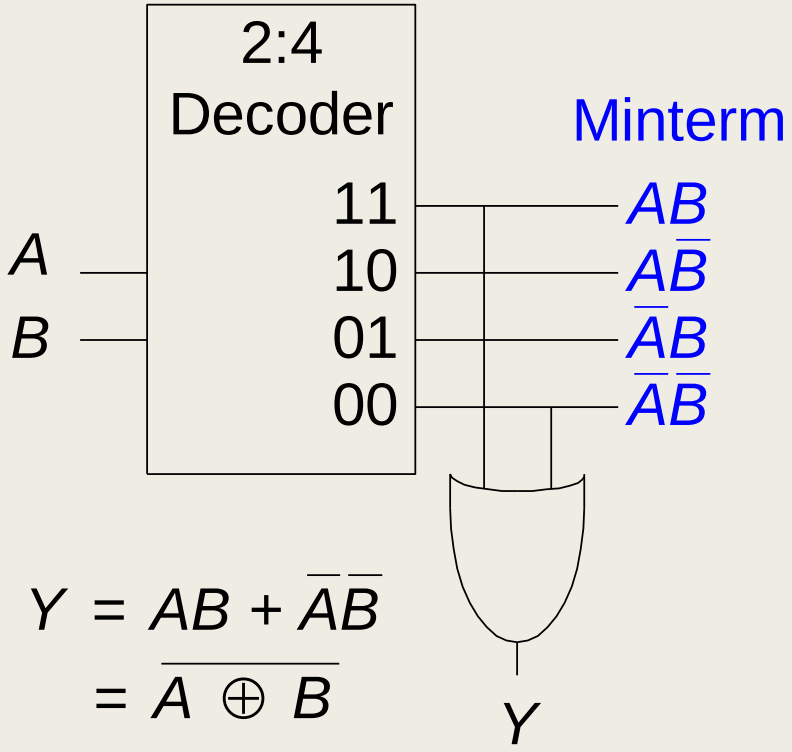
- N entrées, 2^N sorties
- Un décodeur est une fonction qui prend un nombre binaire à n bits en entrée et se sert de celui-ci pour sélectionner l'une de ses 2^N sorties



A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Implémentation des fonctions logiques en utilisant des des décodeurs + une porte OU

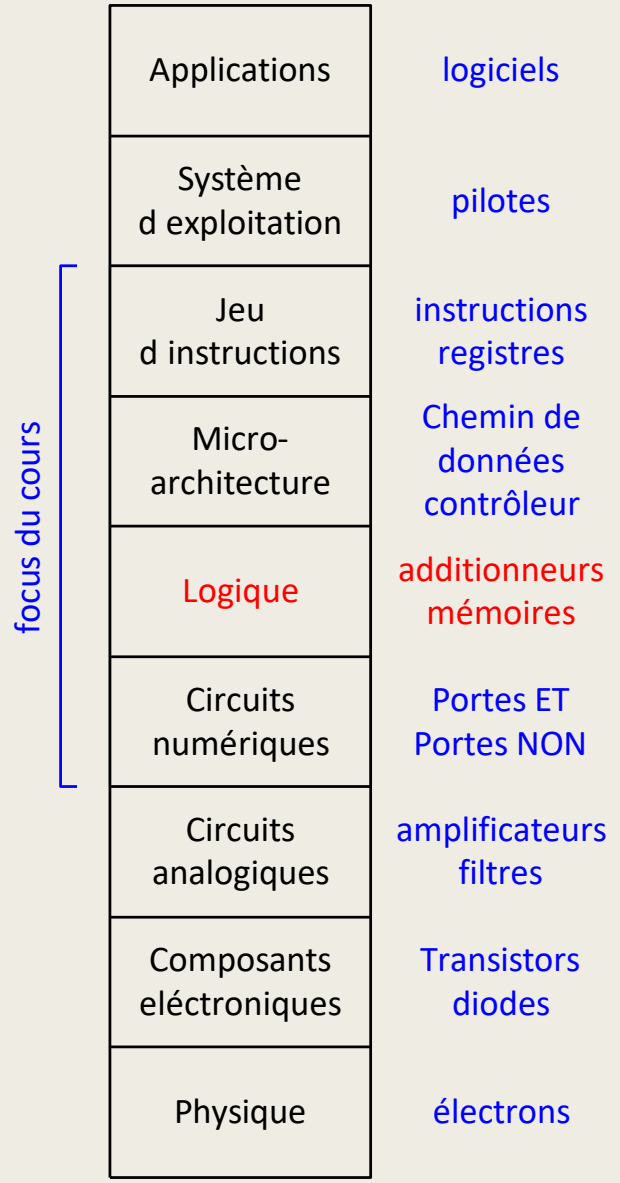
- OU de minterms



PRINCIPAUX COMPOSANTS NUMÉRIQUES

Composants numériques
Chapitre 5

- Introduction
- Circuits arithmétiques
- Blocs de base séquentiels
- Tableaux mémoires
- Tableaux logiques programmables

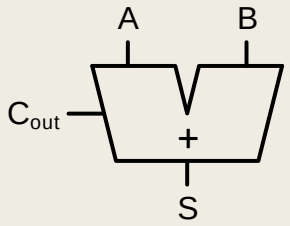


Introduction

- Composants numériques principaux:
  - *Portes, multiplexeurs, décodeurs, registres, circuits arithmétiques, conteurs, tableaux mémoire, tableaux logiques programmables*
- Composants numériques montrent l'application des principes d'hiérarchie, de modularité et de régularité:
  - *Hiérarchie de composants*
  - *Plusieurs modules avec les interfaces et fonctions bien définies*
  - *Structure régulière*
    - Permet la conception de composants de tailles variables
- Utiliserons ces composants pour créer un processeur

Additionneur 1-Bit

Semi Additionneur

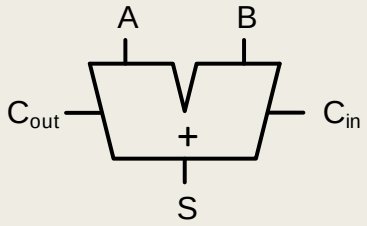


A	B	C_{out}	S
0	0		
0	1		
1	0		
1	1		

S =

C_{out} =

Additionneur complet



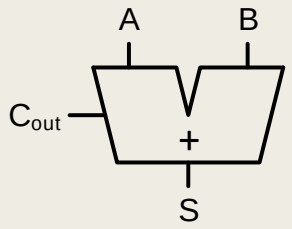
C_{in}	A	B	C_{out}	S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

S =

C_{out} =

Additionneur 1-Bit

Semi Additionneur

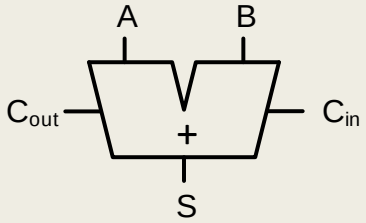


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

S =

C_{out} =

Additionneur Complet



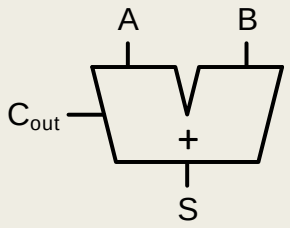
C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

S =

C_{out} =

Additionneur 1-Bit

Semi Additionneur

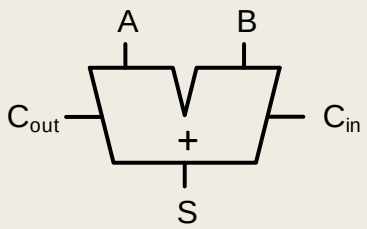


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = A \oplus B$

$C_{out} = AB$

Additionneur complet



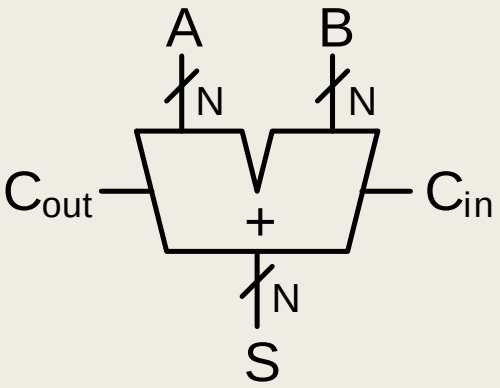
C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$S = A \oplus B \oplus C_{in}$

$C_{out} = AB + AC_{in} + BC_{in}$

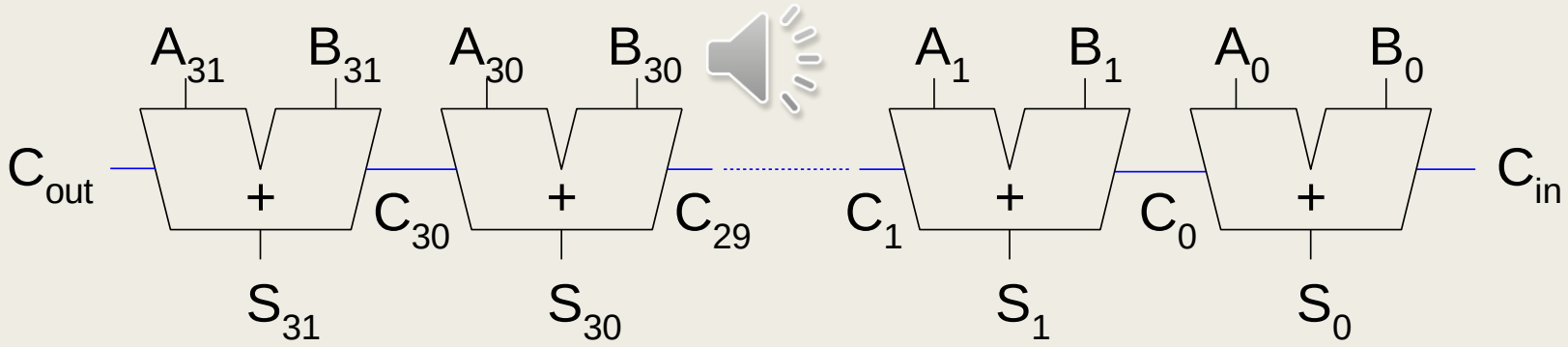
Additionneurs sur plusieurs bits, additionneurs à **propagation de retenue**

- Plusieurs types de “carry propagate adders” (CPAs) :
  - Additionneur à propagation de retenue simple (lent)
  - Additionneur à anticipation de retenue (rapide)
  - Additionneur avec le calcul de préfixes (le plus rapide des 3)
- Les deux derniers additionneurs sont plus rapides pour additionner longs nombres mais nécessitent plus de matériel.



Additionneur à propagation de retenue

- Enchâine les additionneurs d'un bit ensemble
- La retenue est propagée à travers de toute la chaine
- Désavantage: lent



Additionneur à propagation de retenue, Délai

- Le délai d'un additionneur de N bits à propagation de retenue:

$$t_{\text{ripple}} = N * t_{FA}$$

t_{FA} est le délai associé à un additionneur complet

Additionneur à anticipation de retenue

- Idée: calculer la retenue sortante (carry out, C_{out}) pour des blocs de k bits en utilisant les signaux *générer* et *propager*
- **Définitions:**
  - Une colonne (bit i soit génère une retenue soit propage une retenue
  - Une **génération de retenue** (G_i) et une **propagation de retenue** (P_i) pour chaque colonne:
    - Une retenue ne peut avoir lieu que lorsque les deux bits d'entrée sont des 1: $G_i = A_i B_i$
    - La propagation d'une retenue d'entrée peut avoir lieu lorsqu'au moins un bit d'entrée vaut 1: $P_i = A_i + B_i$
    - Une retenue de colonne (C_i) est:

$$C_i = A_i B_i + (A_i + B_i) C_{i-1} = G_i + P_i C_{i-1}$$

Additionneur à anticipation de retenue

- Étape 1: calculer les signaux *générer* (G) et *propager* (P) pour chaque colonne (un seul bit)
- Étape 2: calculer G et P pour chaque bloc de k bits
- Étape 3: Propager C_{in} à travers de chaque propager/générer bloc de k bits

Additionneur à anticipation de retenue

- Par exemple, nous pouvons calculer les signaux *générer* et *propager* pour chaque blocs de 4 bits: $G_{3:0}$ et $P_{3:0}$:
  - Un bloc de 4 bits va générer une retenue sortante si la colonne 3 génère une retenue (G_3) ou si la colonne 3 propage une retenue (P_3) qui dans son tour avait été générée ou propagée par la colonne précédente:

$$G_{3:0} = G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0))$$

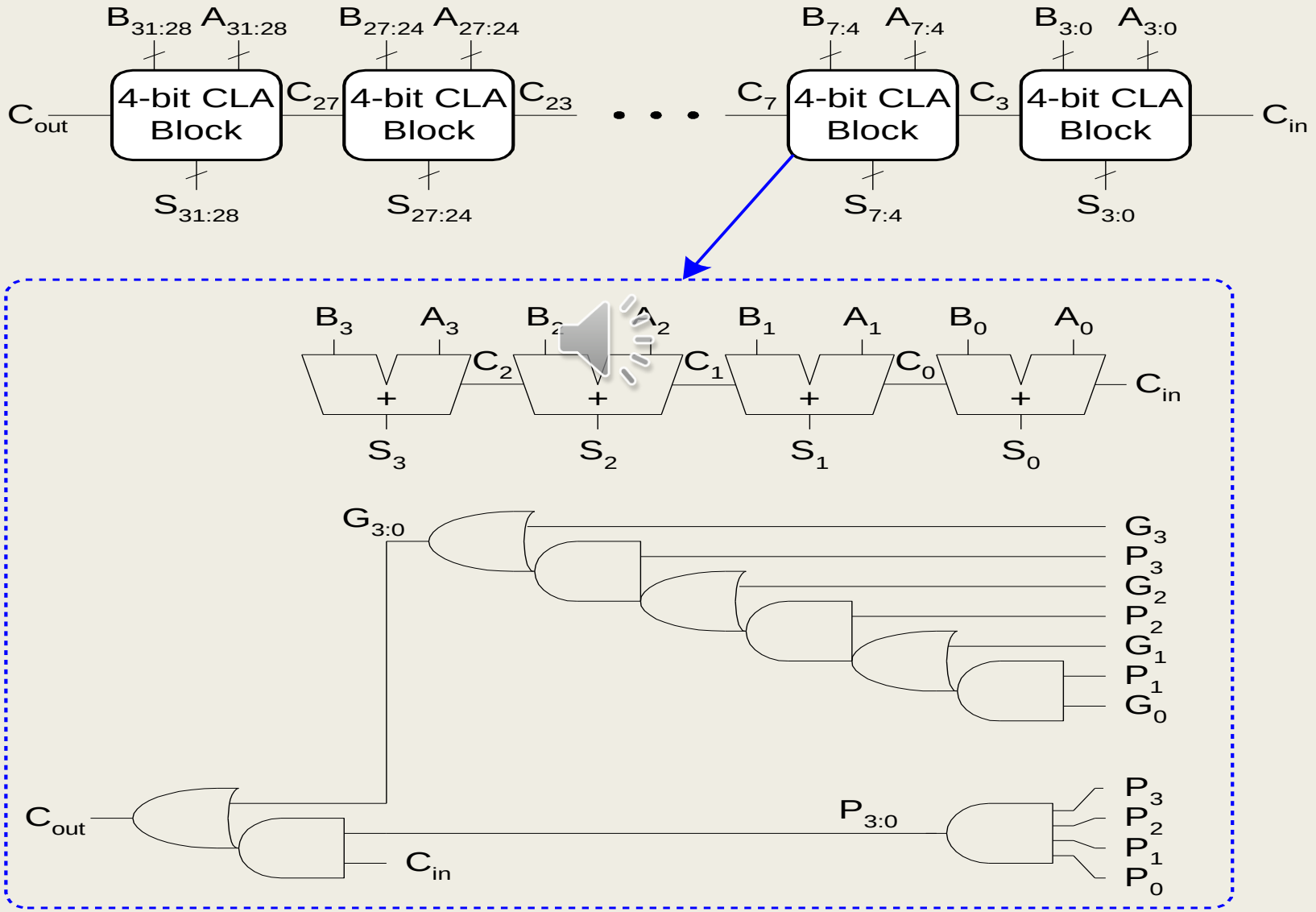
  - Un bloc va propager une retenue lorsque toutes les colonnes de ce bloc propagent la retenue:

$$P_{3:0} = P_3 P_2 P_1 P_0$$

  - La retenue du bloc de 4 bits (C_i):

$$C_i = G_{i:j} + P_{i:j} C_{i-1}$$

Additionneur à anticipation de retenue de 32 bits avec les blocs de 4 bits



Additionneur avec le calcul des préfixes

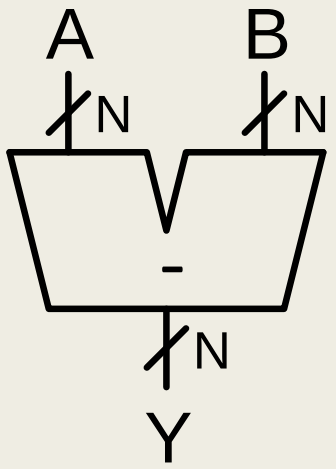
- Calcule la retenue entrante (C_{i-1}) pour chaque colonne le plus rapidement possible et ensuite calcule la somme:

$$S_i = (A_i \oplus B_i) \oplus C_i$$

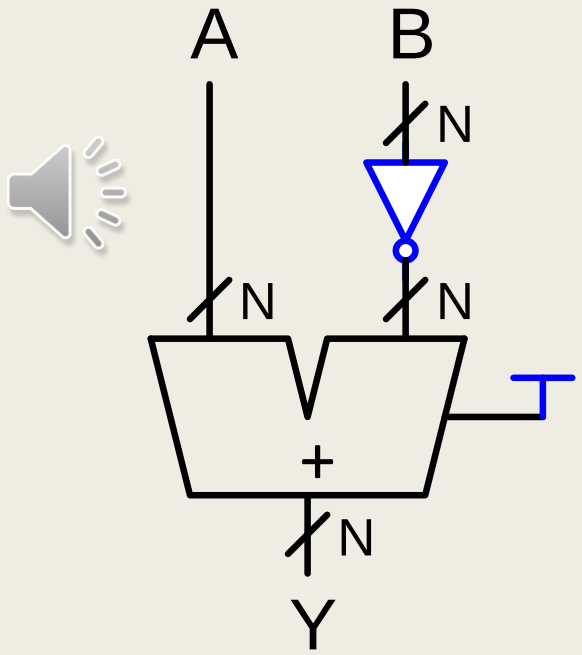
- Calcule G et P pour 1 bit, ensuite pour des blocs de 2 bits, ensuite pour les blocs de 4 bits, de 8 bits, etc. jusqu'au point où un signal de la retenue entrante (signal *générer*) est connu pour chaque colonne
- Possède $\log_2 N$ étages

Soustracteur

Symbol

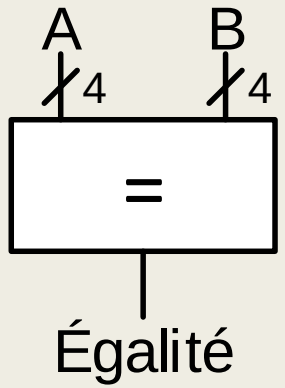


Implémentation

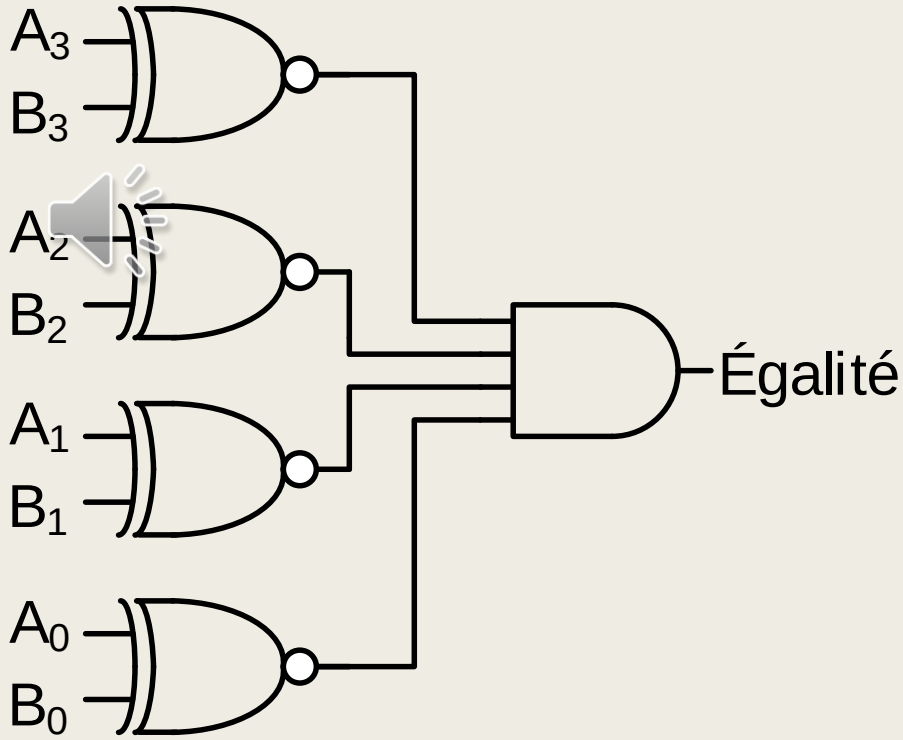


Comparateur: Égalité

Symbol

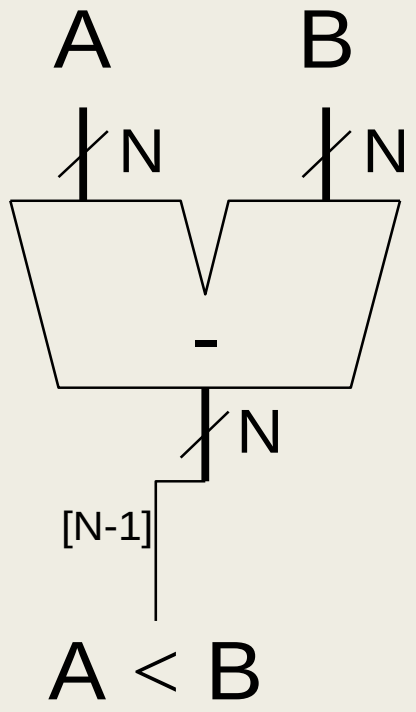


Implémentation

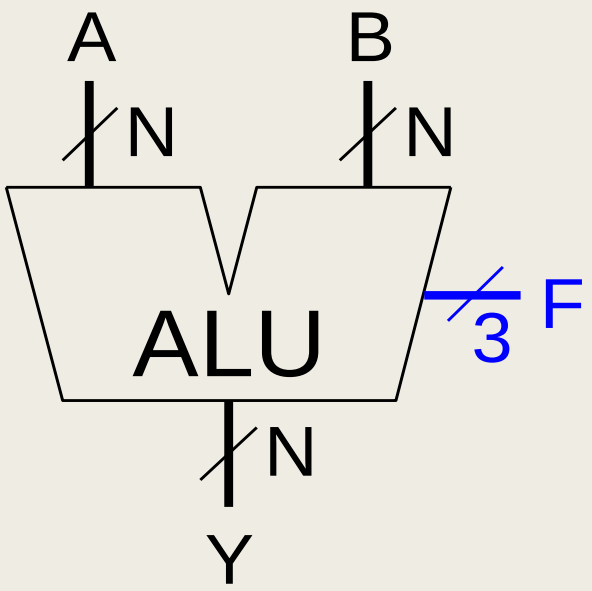


Comparateur: Plus petit que

- Pour les nombres non signés

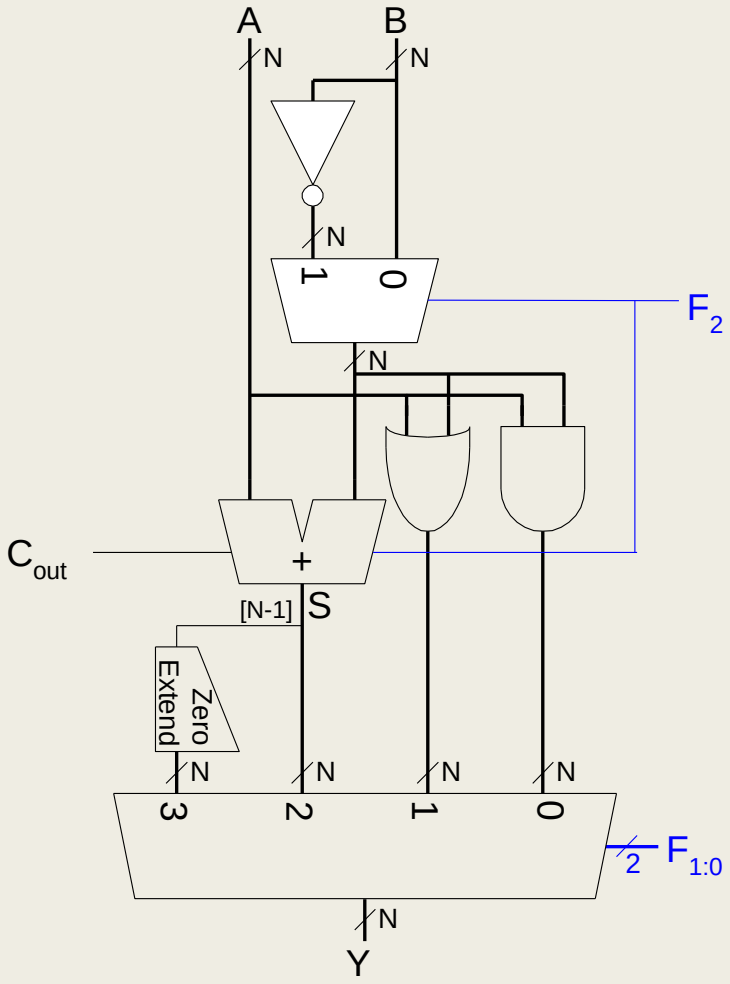


Unité Arithmétique et Logique (UAL)
Arithmetic Logic Unit (ALU)



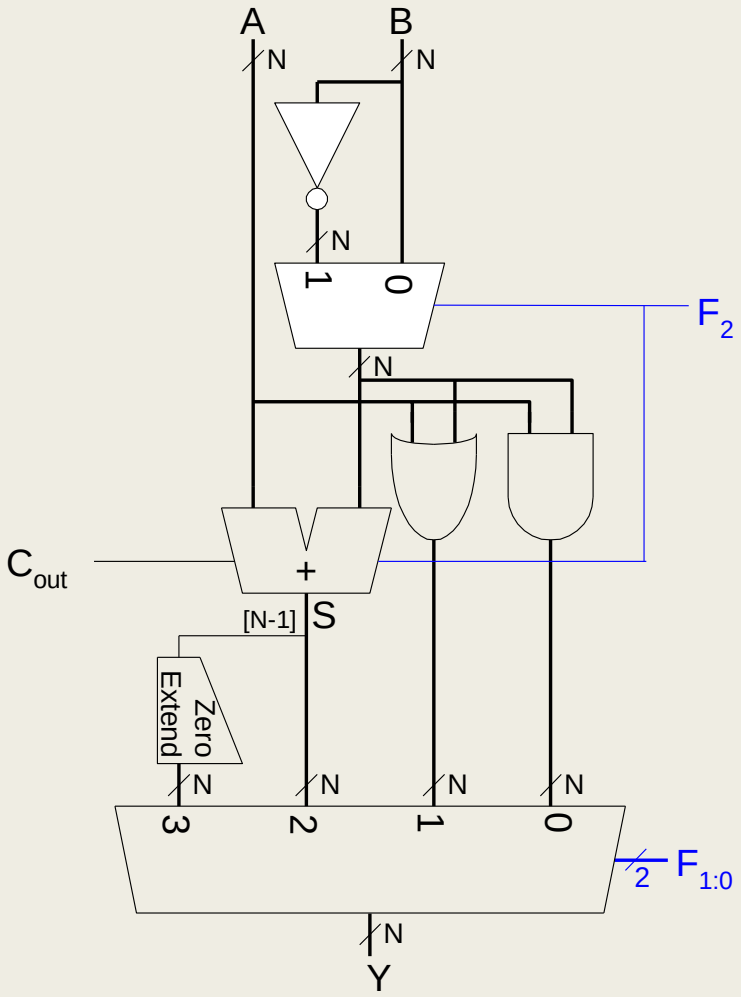
$F_{2:0}$	Fonction
000	A and B
001	A or B
010	A + B
011	Non utilisé
100	A and $\bar{B}$
101	A or $\bar{B}$
110	A - B
111	SLT (Comparateur Plus Petit que)

Implémentation de l'UAL



$F_{2:0}$	Fonction
000	A and B
001	A or B
010	A + B
011	Non utilisé
100	A and $\bar{B}$
101	A or $\bar{B}$
110	A - B
111	SLT (Comparateur Plus Petit que)

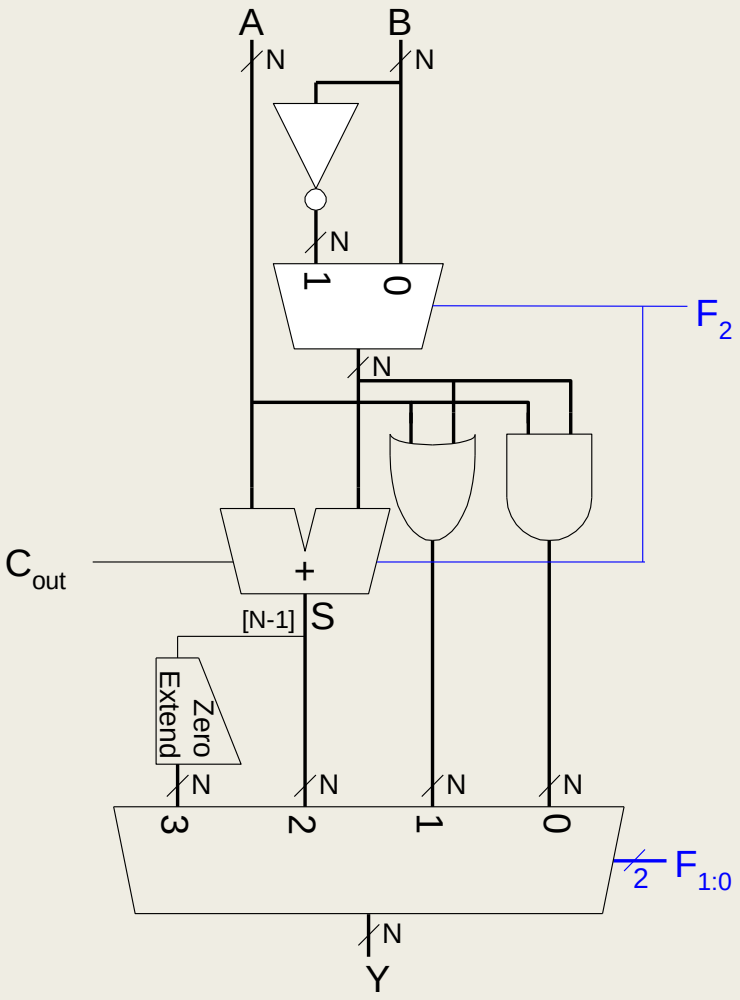
Set Less Than (SLT), Plus Petit que, Exemple



- Configurer l'UAL de 32 bits pour exécuter l'opération SLT (Comparaison si un opérande est plus petit que l'autre; si le résultat de comparaison est VRAI, le résultat sur 32 bits retourné doit être 0…01 et 0…0 dans le cas contraire). Supposer $A = 25$ et $B = 32$.

$F_{2:0}$	Fonction
000	A and B
001	A or B
010	A + B
011	Non utilisé
100	A and $\bar{B}$
101	A or $\bar{B}$
110	A - B
111	SLT (Comparateur Plus Petit que)

Set Less Than (SLT), Plus Petit que Exemple

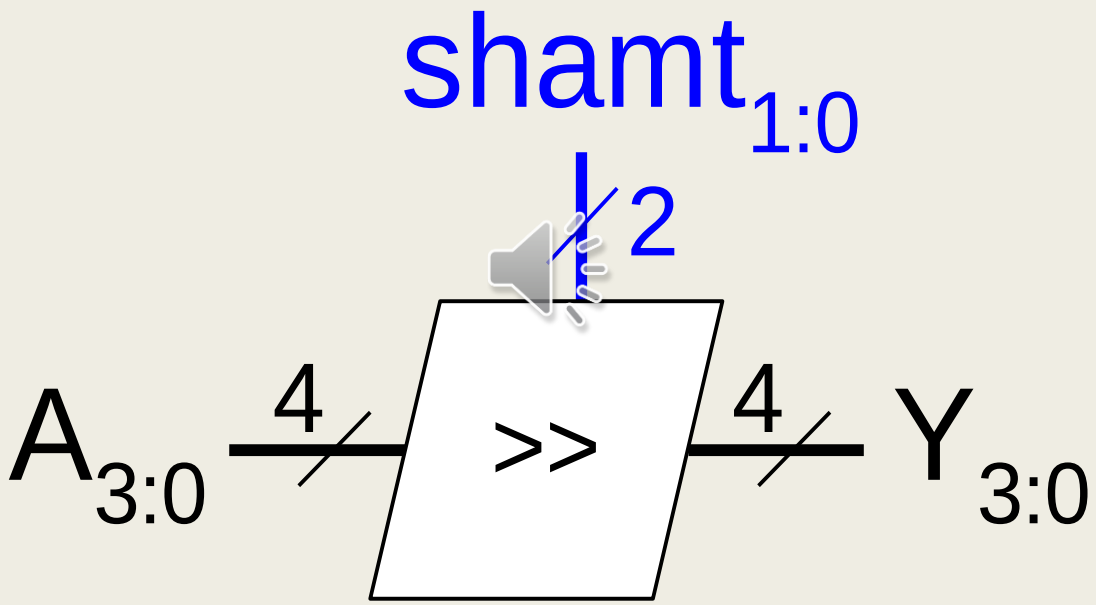


- A=25 plus petit que B=32, donc Y devrait être la représentation de 1 sur 32 bits (0x00000001).
- Pour SLT, $F_{2:0}$ = 111.
- F_2 = 1 configure l'additionneur en soustracteur; 25 - 32 = -7
- La représentation de -7 en complément à 2 possède 1 dans la position la plus significative, S_{31} = 1
- Avec $F_{1:0}$ = 11, le multiplexeur final sélectionne $Y = S_{31}$ étendu sur 32 bits = 0x00000001.

Circuits combinatoires de décalage

- **Décalage logique :** décale les bits de gauche ou droite et remplisse les positions vides par les zéros
  - *Ex: 11001 >> 2 = 00110*
  - *Ex: 11001 << 2 = 00100*
- **Décalage arithmétique :** se diffère de décalage logique seulement au décalage à droite: remplisse les espaces vides par le bit le plus significatif (msb) de la valeur initiale
  - *Ex: 11001 >>> 2 = 11110*
  - *Ex: 11001 <<< 2 = 00100*
- **Rotation:** fait la rotation des bits de manière circulaire: les bits sortants d'un bout reviennent aux positions libérées de l'autre bout
  - *Ex: 11001 ROR 2 = 01110*
  - *Ex: 11001 ROL 2 = 00111*

Circuit de décalage logique à droite



Circuits de décalage comme multiplicateurs et diviseurs

- Décalage à gauche de N bits multiplie un nombre par 2^N
  - *Ex: 00001 << 2 = 00100 ($1 \times 2^2 = 4$)*
  - *Ex: 11101 << 2 = 10100 ($-3 \times 2^2 = -12$)*

- Décalage arithmétique à droite de *N* bits divise un nombre par 2^N
  - *Ex: 01000 >>> 2 = 00010 ($8 \div 2^2 = 2$)*
  - *Ex: 10000 >>> 2 = 11100 ($-16 \div 2^2 = -4$)*

Multiplicateurs

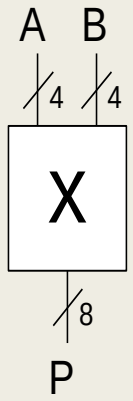
- Étapes de multiplication des nombres en décimal et en binaire:
  - *Produits partiels sont formés par la multiplication de chaque chiffre du multiplicateur par le multiplicande*
  - *Des produits partiels décalés sont additionnés pour former le résultat*

Decimal		**Binary**
230	multiplicand	0101
x 42	multiplier	x 0111
460	partial products	0101
+ 920		0101
		0101
		+ 0000
9660	result	0100011

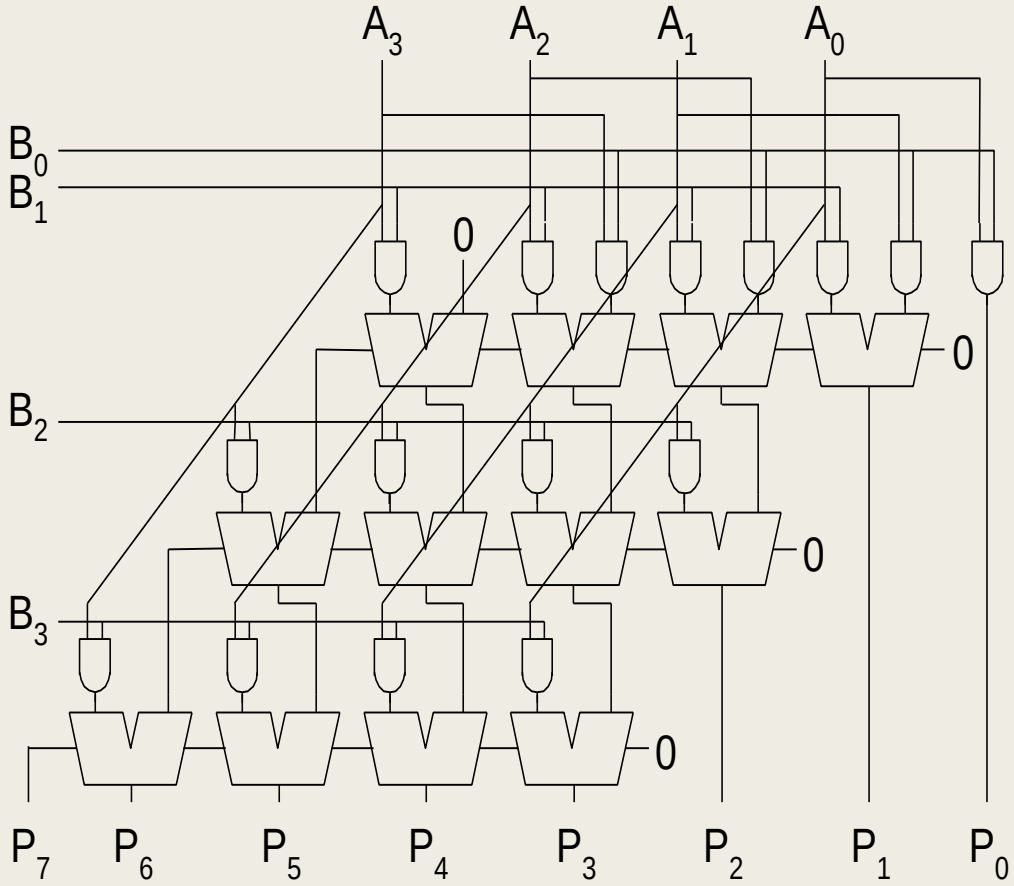
230 x 42 = 9660

5 x 7 = 35

Multiplieur 4 x 4



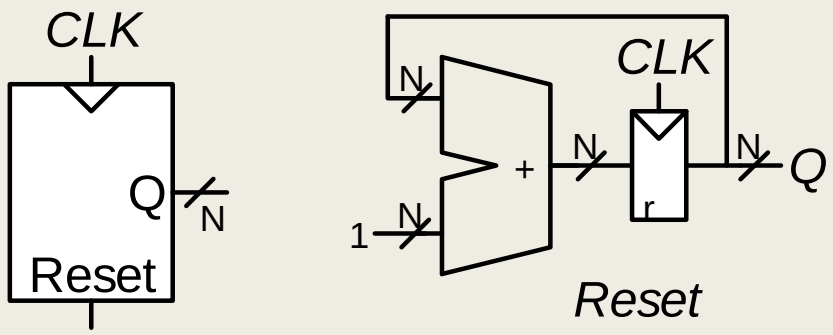
$$
\begin{array}{rcccccccc}
 & & & A_3 & A_2 & A_1 & A_0 & & \\
\times & & & B_3 & B_2 & B_1 & B_0 & & \\
\hline
 & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 & & \\
 & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & & & \\
 & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & & & \\
+ & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & & \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$



Compteurs

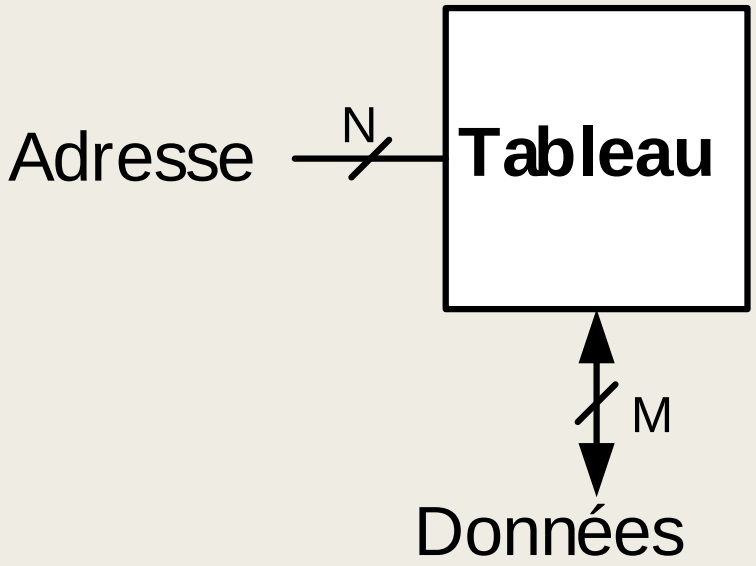
- Incrémentent une valeur stockée chaque front d'horloge
- Utilisés pour parcourir cycliquement des nombres. Par exemple,
  - 000, 001, 010, 011, 100, 101, 110, 111, 000, 001…
- Applications des compteurs:
  - Affichage d'horloge numérique
  - Compteur ordinal de programme (PC): garde la trace d'instruction en cours d'exécution

Symbol **Implémentation**



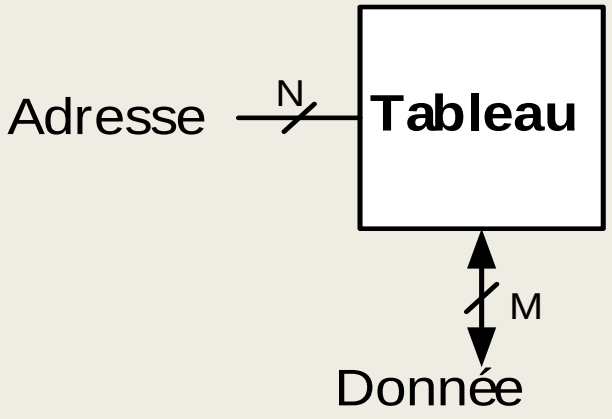
Tableaux Mémoires

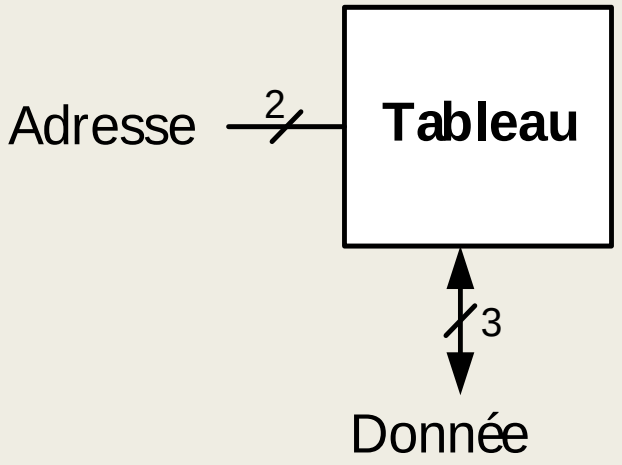
- Stockent efficacement les grandes quantités de données
- Trois types communs:
  - Dynamic random access memory (DRAM), RAM dynamique
  - Static random access memory (SRAM), RAM statique
  - Read only memory (ROM), mémoire morte, qui est un circuit accessible uniquement en lecture (plus maintenant)
- Une valeur de M bits peut être lue ou écrite à chaque adresse unique de N-bits

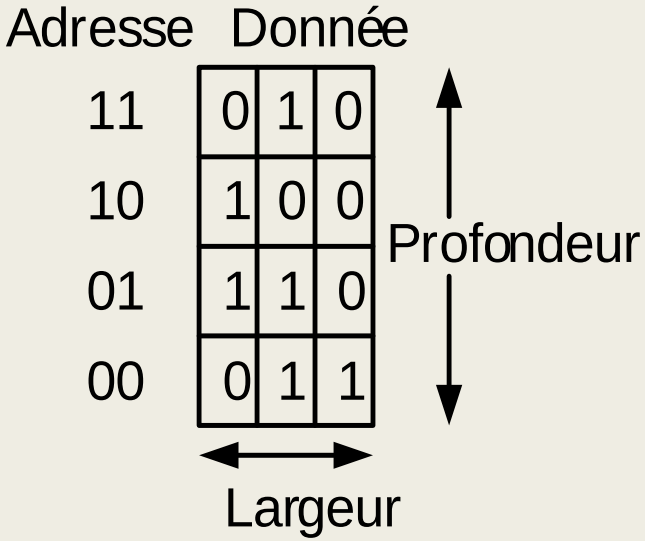


Tableaux Mémoires

- Tableau 2D des cellules qui stockent 1 bit
- Un tableau avec N bits d'adresse et *M* bits de données:
  - 2^N lignes et *M* colonnes
  - **Profondeur:** nombre des lignes (nombre des unités adressables)
  - **Largeur:** nombre des colonnes (taille d'unité adressable)
  - **Taille de tableau:** profondeur × largeur = $2^N \times M$



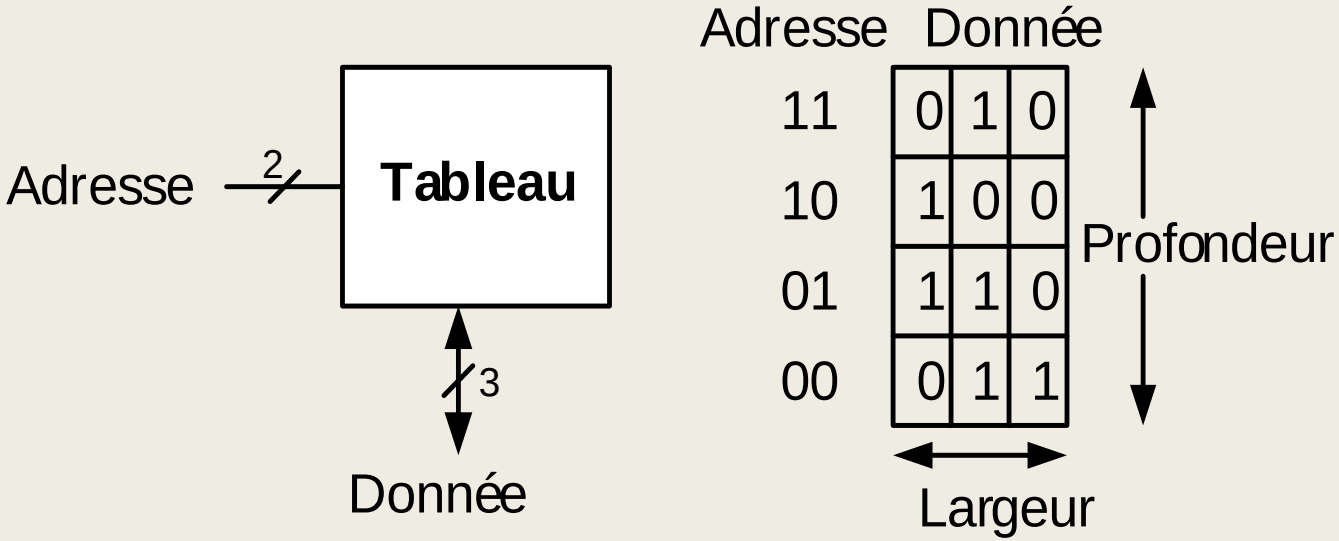




Tableaux Mémoires, Exemple

- Tableau de $2^2 \times 3$-bits
- Nombre des unités adressables: 4
- Taille d'unité adressable: 3-bits
- Par exemple, le contenu de l'adresse 2 (10) est la donnée 100

Exemple:



Mémoires

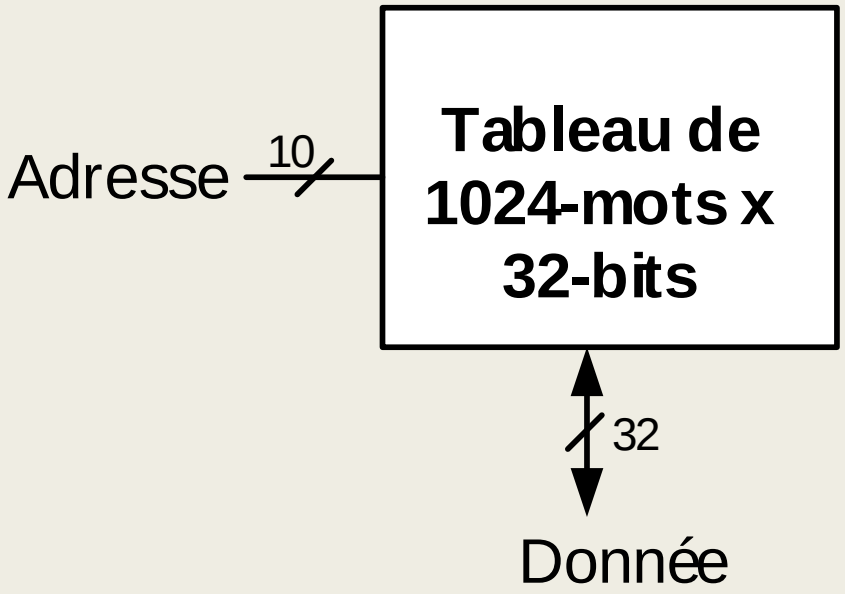
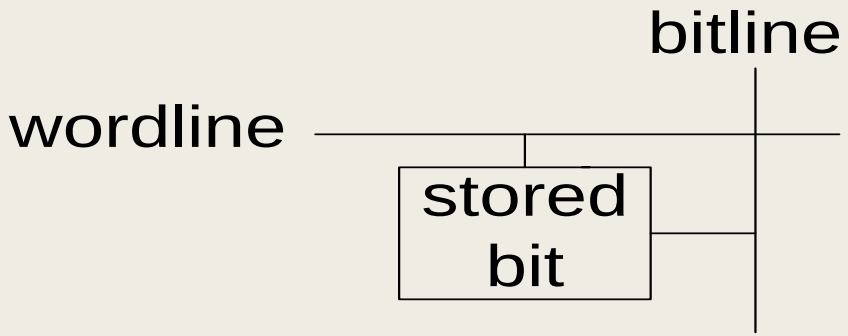


Tableau Mémoire, Cellule d'un bit

Exemple:



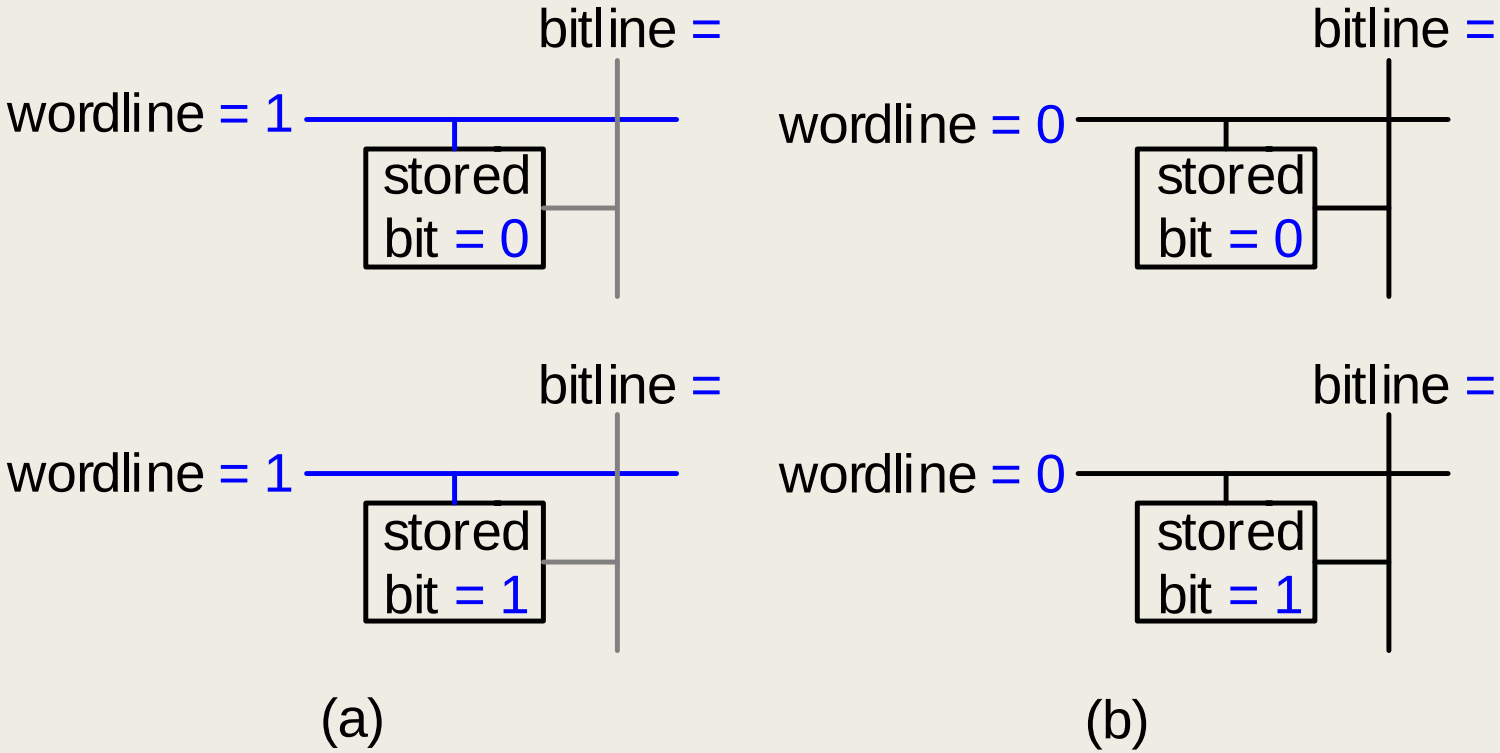
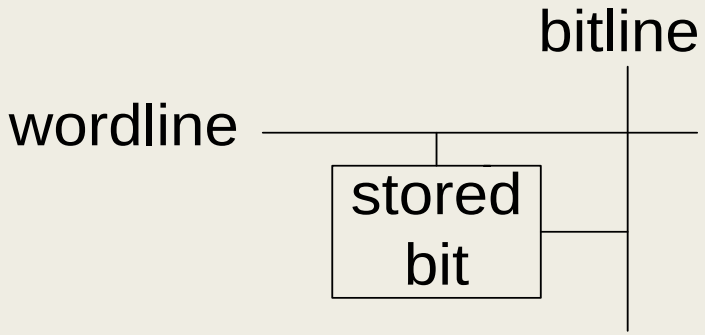


Tableau Mémoire, Cellule d'un bit



Exemple:

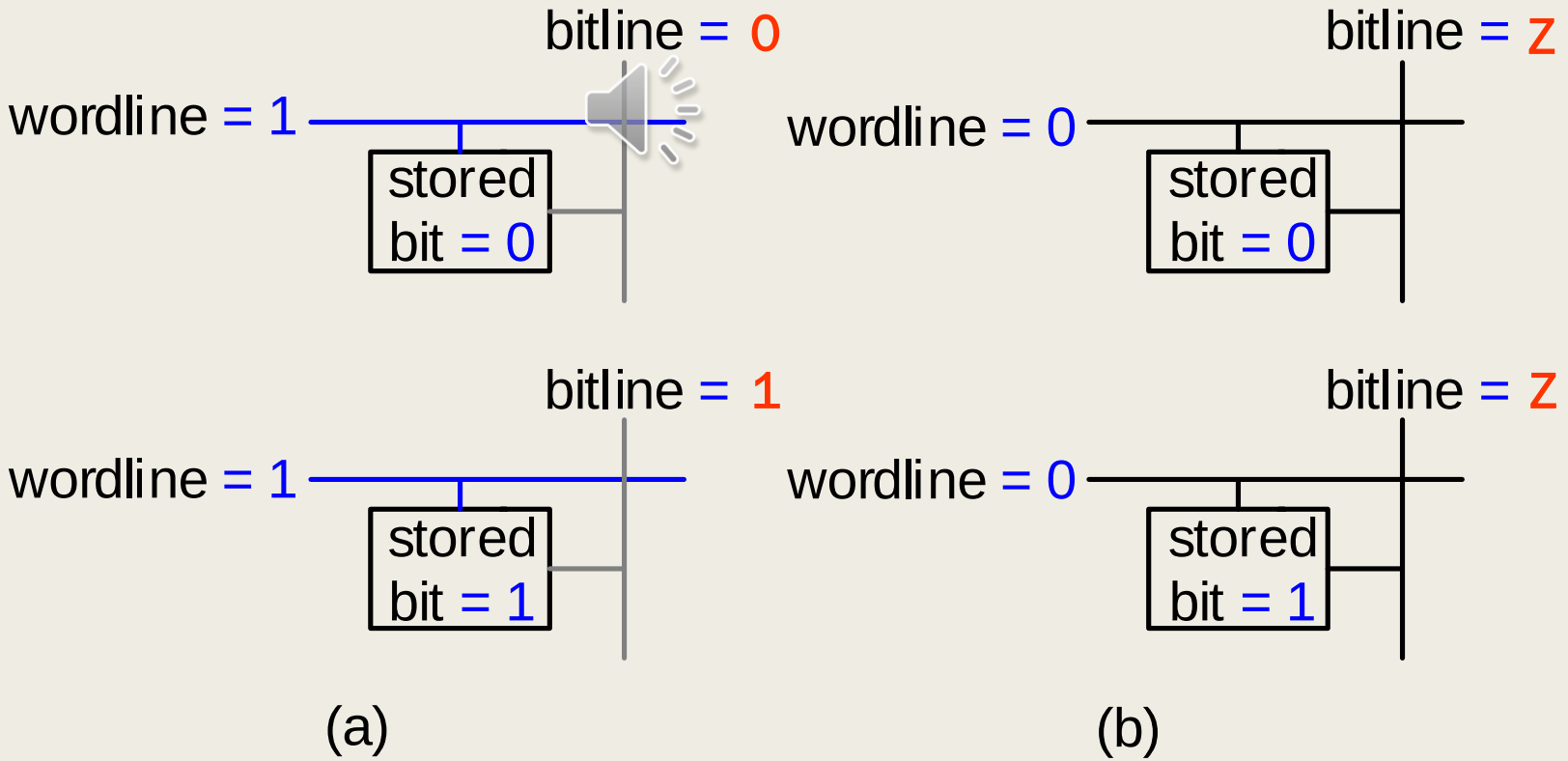
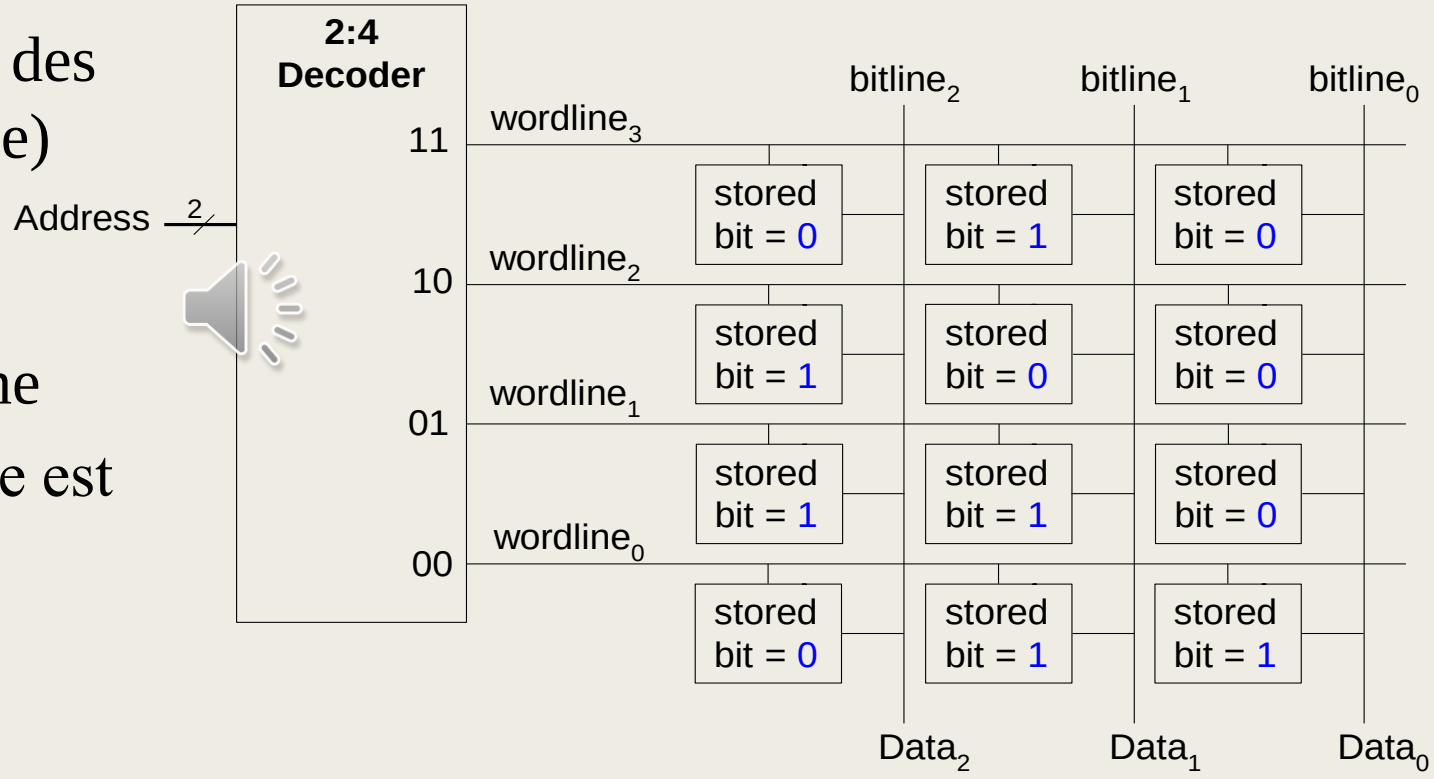


Tableau Mémoire

- **Ligne d'adresse:**
  - Permet à une seule ligne des cellules (unité adressable) d'être écrite ou lue
  - Une adresse unique correspond à chaque ligne
  - Une seule ligne d'adresse est active (HIGH) à chaque moment



Types de mémoires

- Random access memory (RAM) : volatile
- Read only memory (ROM) : nonvolatile

RAM: Random Access Memory

Volatiles: des mémoires *volatiles* sont des mémoires qui ne conservent leur contenu que lorsqu'elles sont sous tension

- Écrite et lue rapidement
- Mémoire principale (vive) dans un ordinateur est RAM (DRAM)

ROM: Read Only Memory – Mémoire morte

- **Non volatile:** retienne des données hors tension électrique
- Lecture rapide, mais l'écriture est impossible ou lente
- Mémoire Flash dans les caméscopes, dans les caméras numériques, clés USB sont toutes ROMs
- Historiquement appelée mémoire en lecture seule ou mémoire morte
  - était écrite au moment de la fabrication
  - Une fois configurée, cette mémoire ne pouvait plus modifiée
  - Ce n'est plus le cas avec la mémoire FLASH ainsi que d'autres types de ROMs aujourd'hui

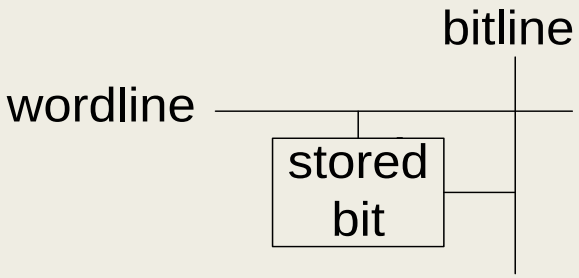
Types de RAM

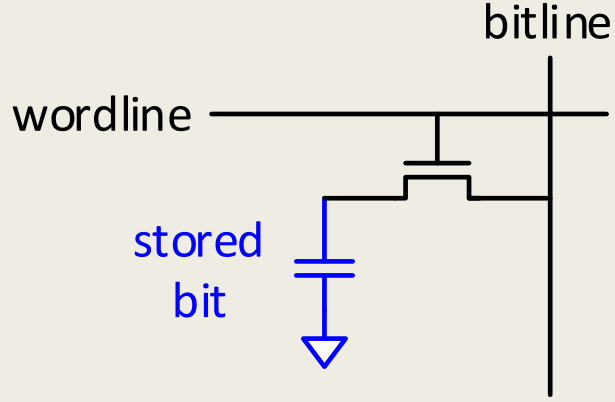
- Deux grandes catégories de mémoires vives
  - Les **mémoires dynamiques** (**DRAM**, *Dynamic Random Access Module*)
    - peu coûteuses
    - utilisées pour la mémoire centrale de l'ordinateur
  - Les **mémoires statiques** (**SRAM**, *Static Random Access Module*)
    - rapides et onéreuses

    utilisées pour les mémoires cache du processeur
- Se différencient par la moyenne de stocker l'information
  - DRAM utilise un condensateur
  - SRAM utilise les inverseurs couplés (bascules)

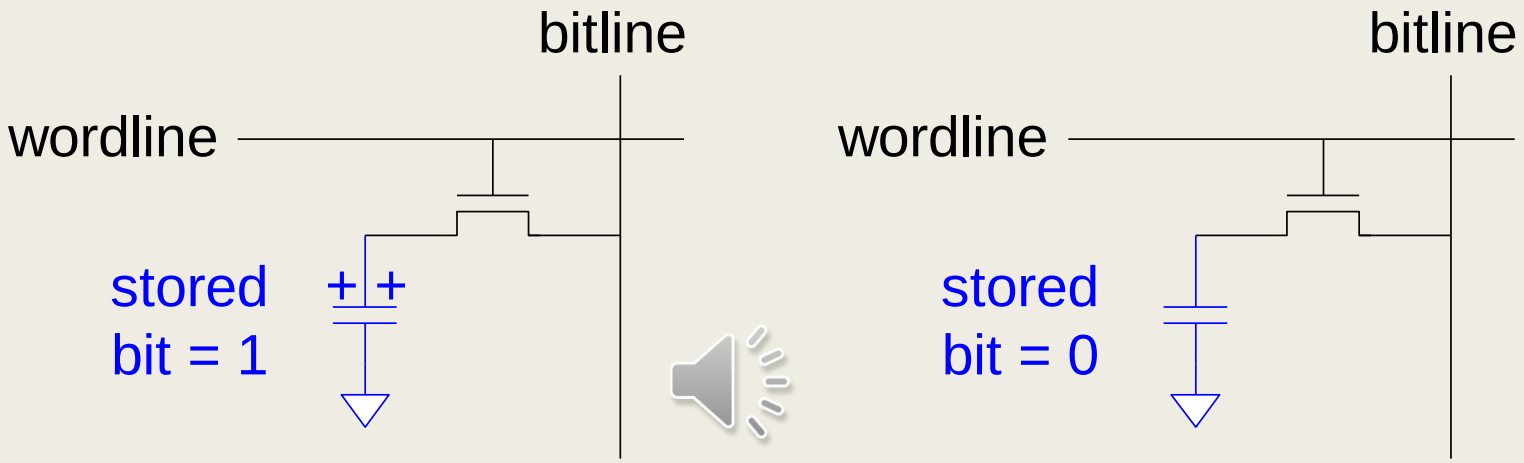
DRAM

- La mémoire vive est constituée de centaines de milliers de petits condensateurs emmagasinant des charges
  - Lorsque un condensateur est chargé, l'état logique du condensateur est égal à 1, dans le cas contraire il est à 0, ce qui signifie que chaque condensateur représente un bit de la mémoire
- Chaque condensateur est couplé à un transistor permettant de « récupérer » ou de modifier l'état du condensateur
- Appelée *dynamique* car , étant donné que les condensateurs se déchargent ainsi que la lecture détruit la charge, il faut constamment les recharger (rafraîchir)

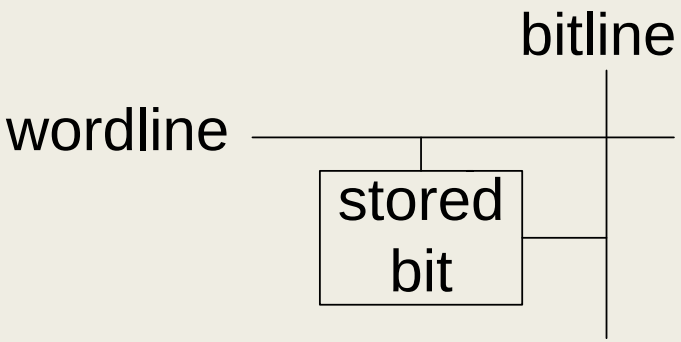


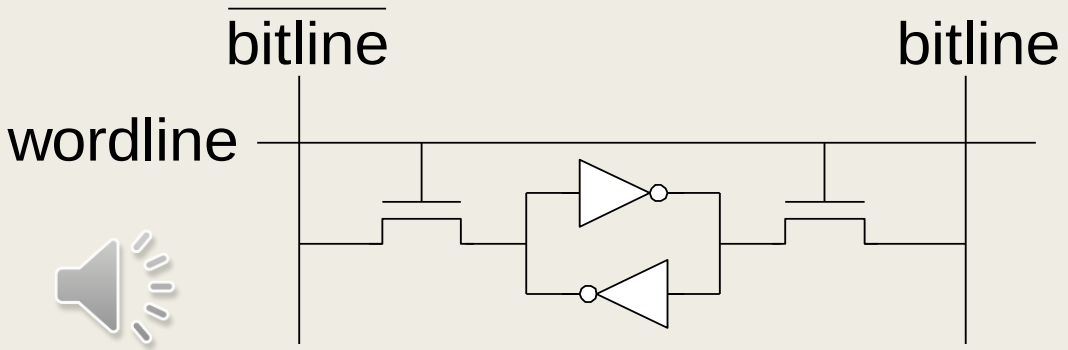


DRAM


bitline
wordline
stored
bit = 1
+ +
bitline
wordline
stored
bit = 0

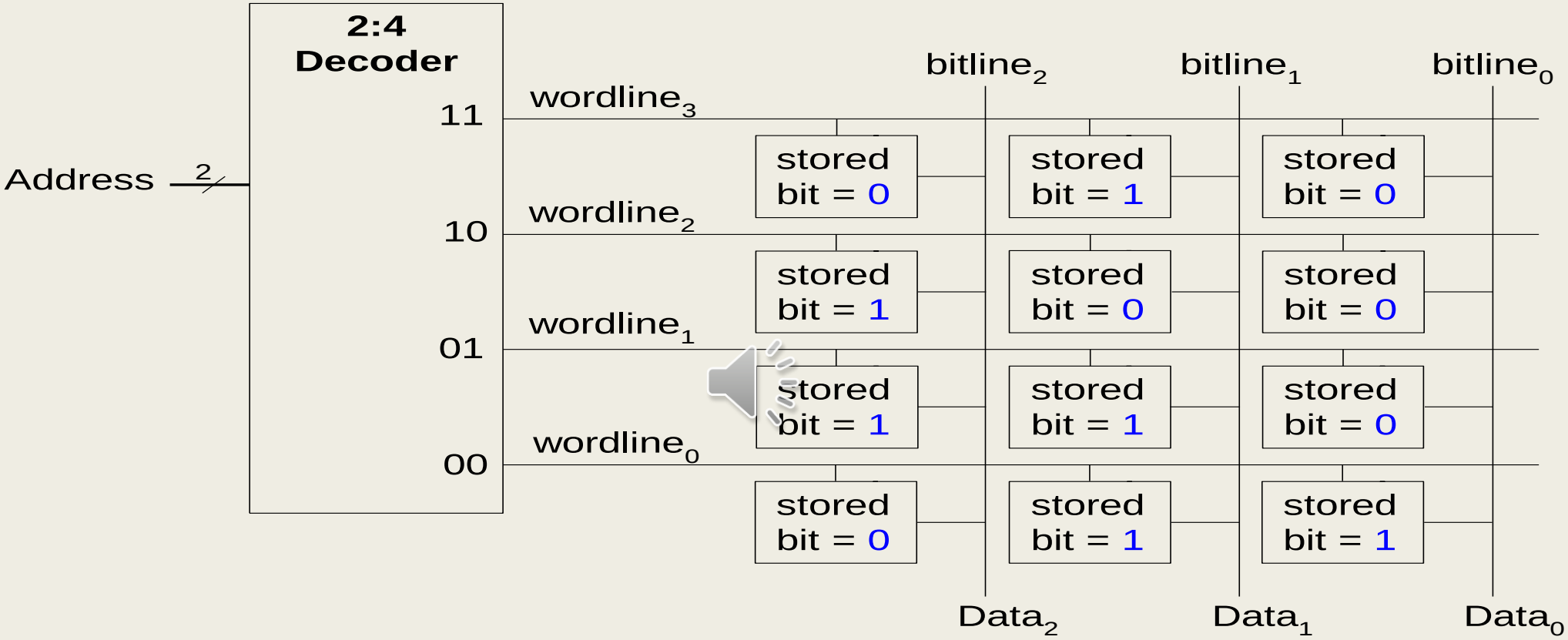
SRAM



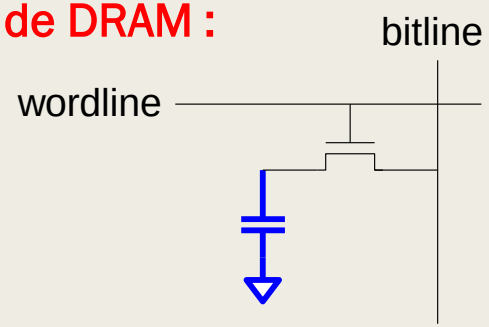


- SRAM est plus onéreuse et moins dense mais beaucoup moins énergivore et plus rapide que la mémoire dynamique
  - 6 à 25 ns de temps d'accès contre 60 à 120

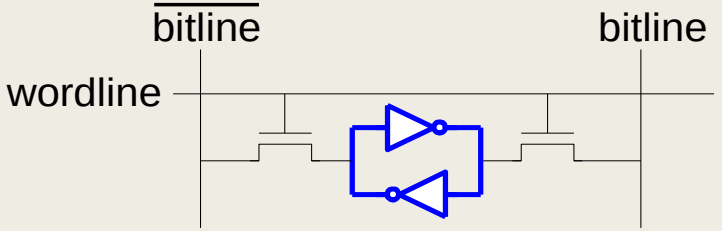
Tableau Mémoire



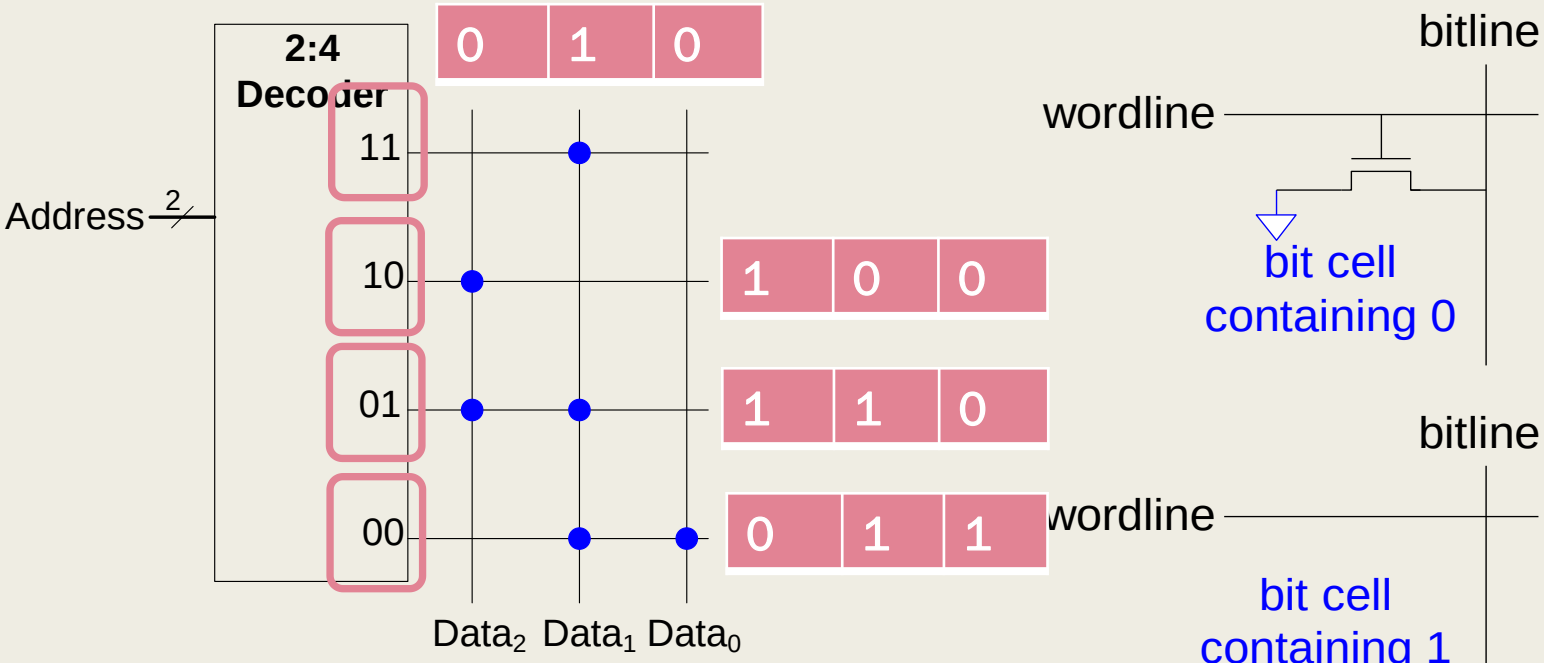
Cellule d'un bit de DRAM :



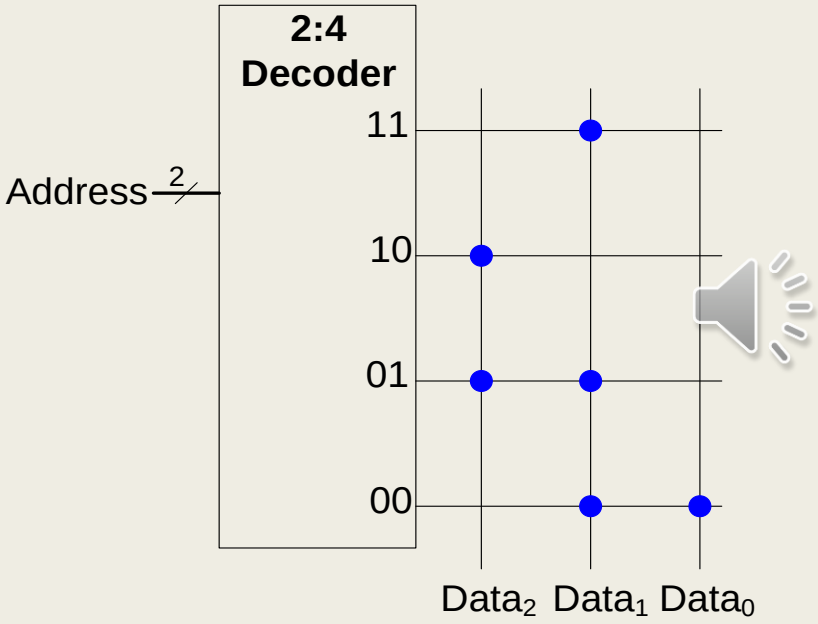
Cellule d'un bit de SRAM:



ROMs: Notation des points



Contenu de ROM

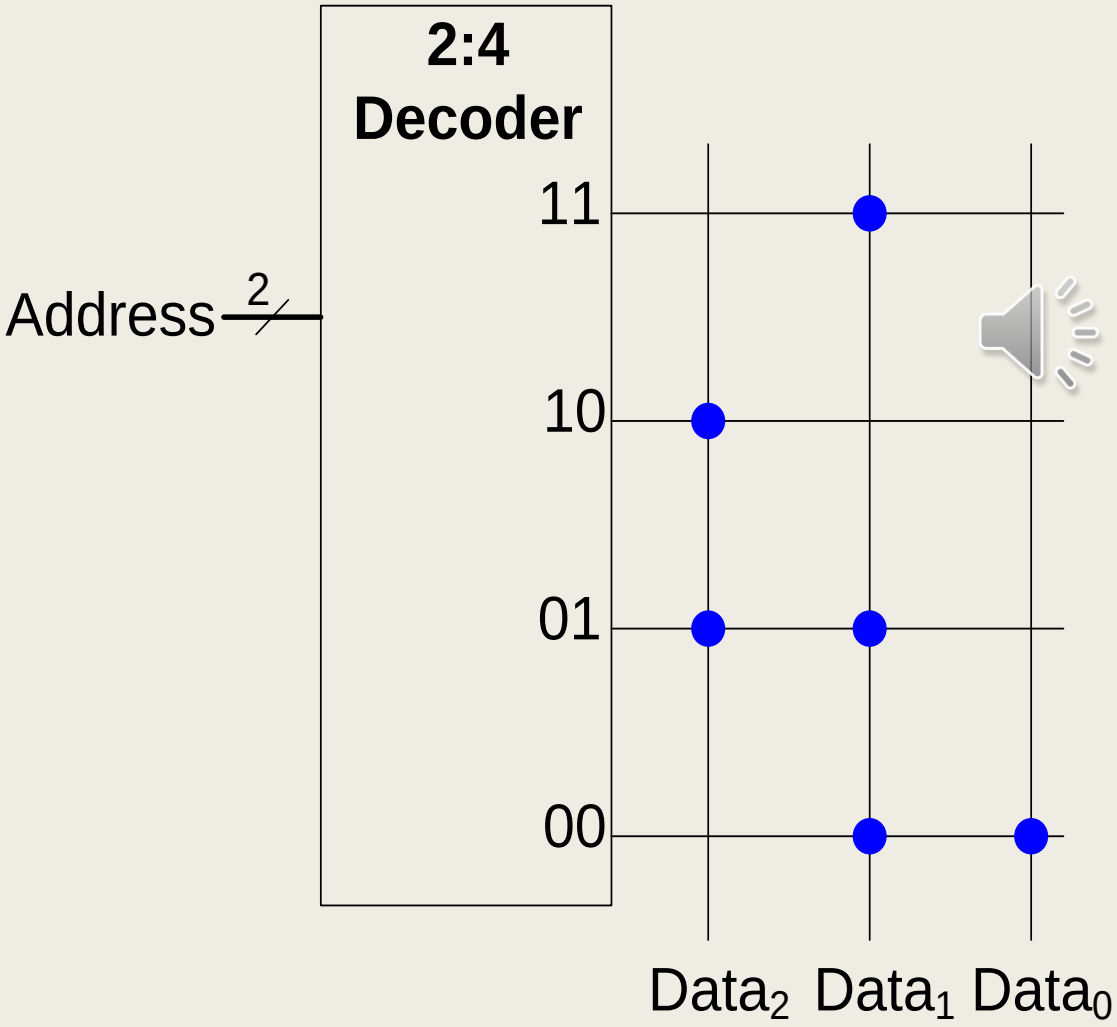


Address	Data		
11	0	1	0
10	1	0	0
01	1	1	0
00	0	1	1

depth

width

Mémoires pour la réalisation de fonctions logiques : ROM



$$Data_2 = A_1 \oplus A_0$$

$$Data_1 = \overline{A_1} + A_0$$

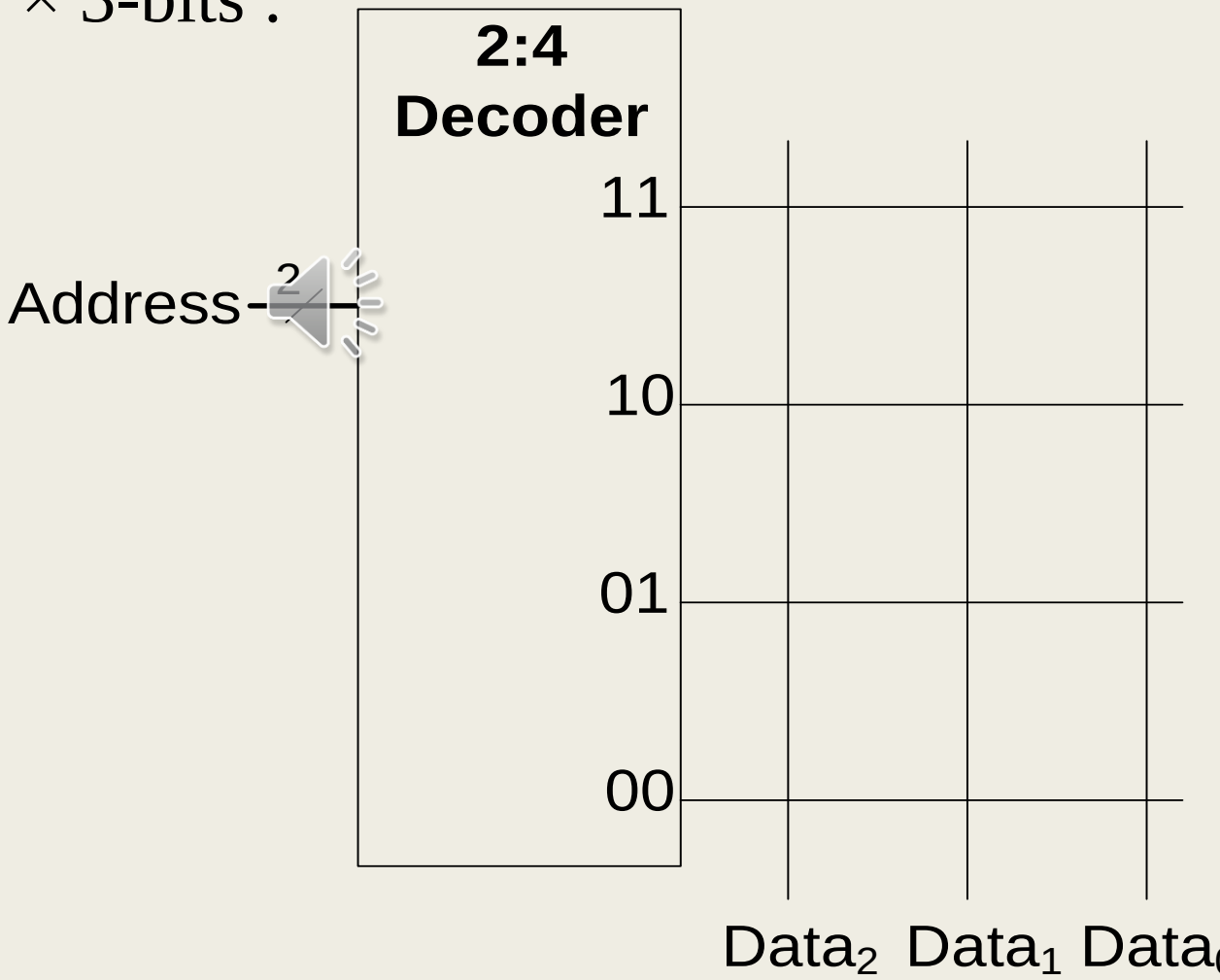
$$Data_0 = \overline{A_1}\,\overline{A_0}$$

$Data_2 = A_1 \oplus A_0$

A1	A0	Data2
0	0	0
0	1	1
1	0	1
1	1	0

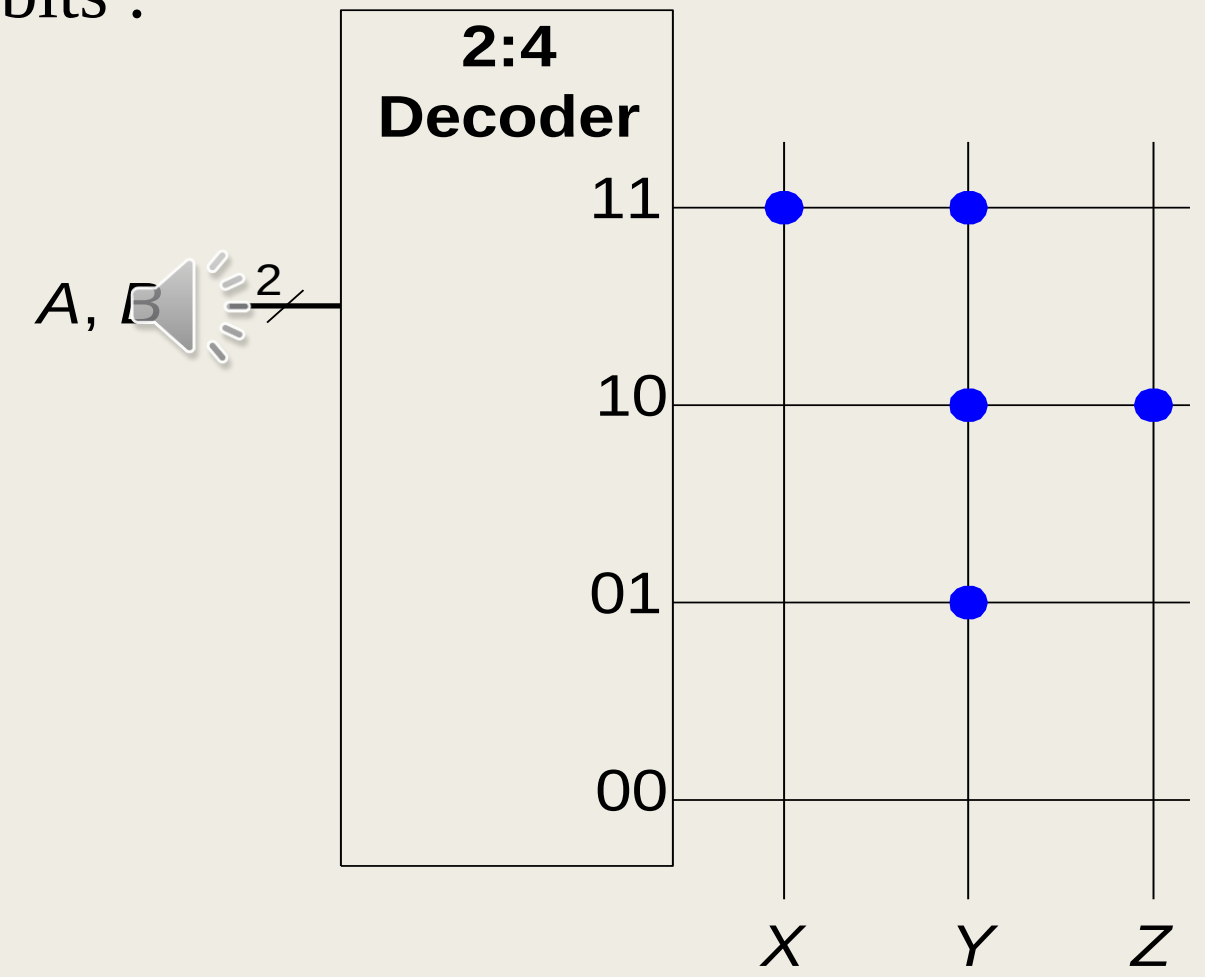
Exemple: LUT avec ROMs

- Implémentez les fonctions logiques suivantes en utilisant une mémoire ROM de $2^2 \times 3$-bits :
  - $X = AB$
  - $Y = A + B$
  - $Z = A\overline{B}$



Exemple: LUT avec ROMs

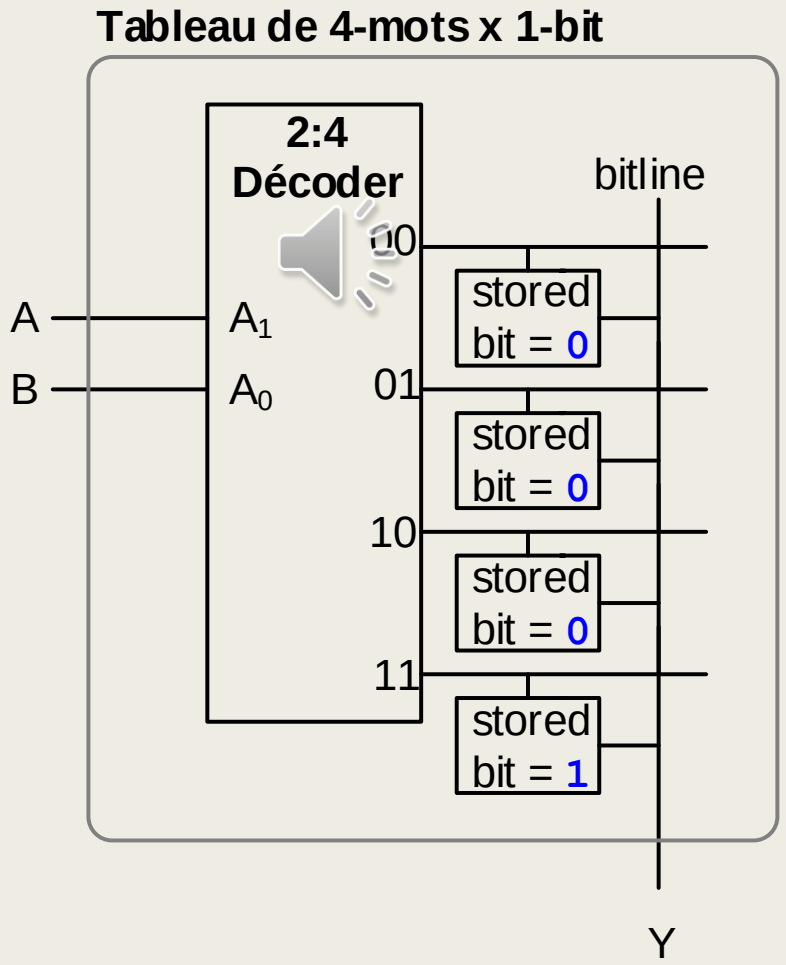
- Implémentez les fonctions logiques suivantes en utilisant une mémoire ROM de $2^2 \times 3$-bits :
  - $X = AB$
  - $Y = A + B$
  - $Z = A\overline{B}$



LUT avec n'importe quel Tableau Mémoire

Table de vérité

A	B	Y
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

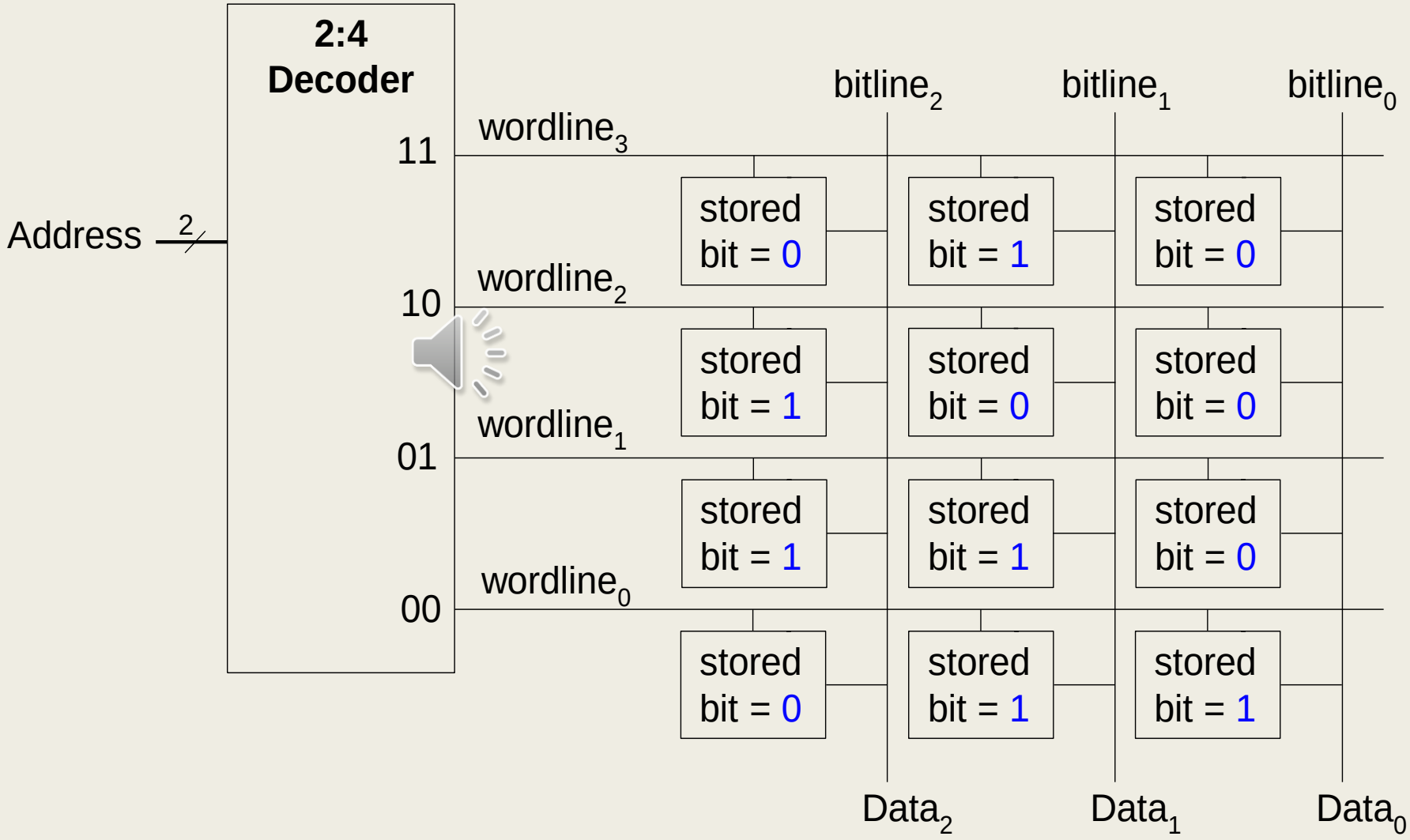


LUT avec Tableau Mémoire

$$Data_2 = A_1 \oplus A_0$$

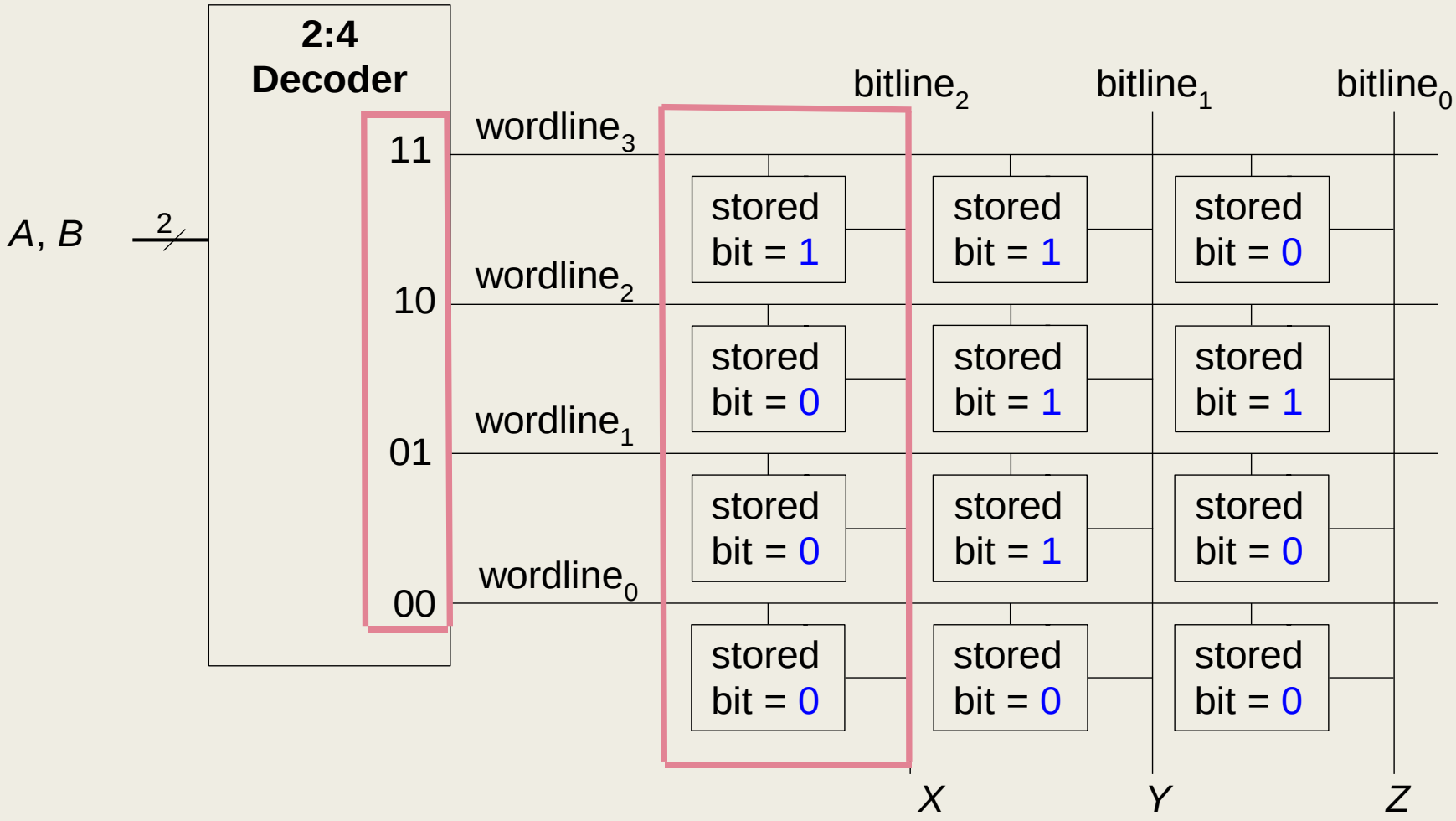
$$Data_1 = \overline{A_1} + A_0$$

$$Data_0 = \overline{A_1}\,\overline{A_0}$$



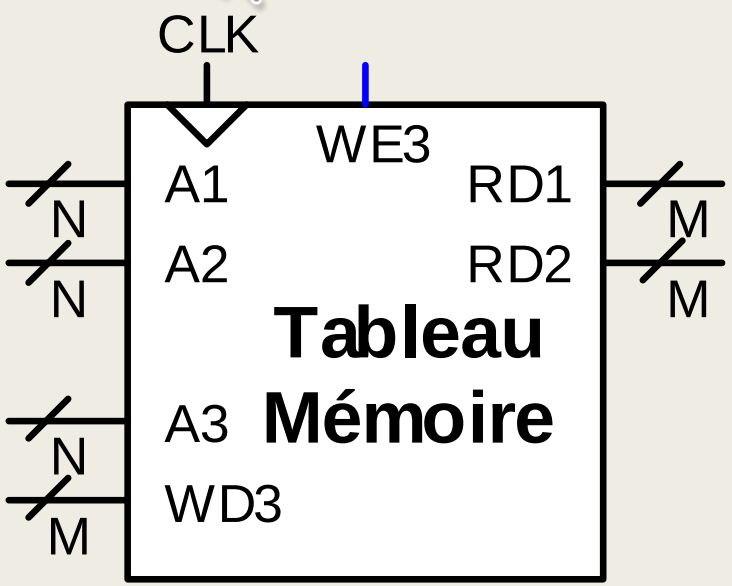
LUT avec Tableau Mémoire

- Implement the following logic functions using a $2^2 \times 3$-bit memory array:
  - $X = AB$
  - $Y = A + B$
  - $Z = A\overline{B}$



Mémoires Multiports

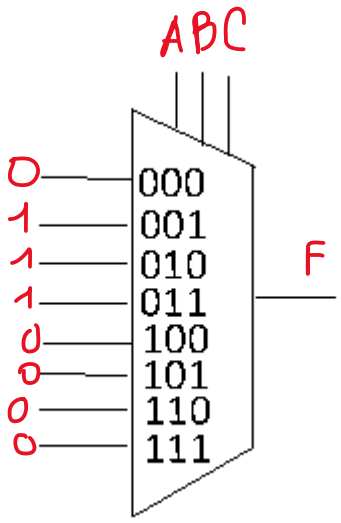
- **Port:** Paire adresse/donnée
- Exemple, Mémoire de 3-ports
  - 2 ports (A1/RD1, A2/RD2) en lecture
  - 1 port (A3/WD3, WE3 permet l'écriture) en écriture
- Une petite mémoire multiports est appelée un *fichier de registres*



INF 4170 – Architecture des ordinateurs
Exercices

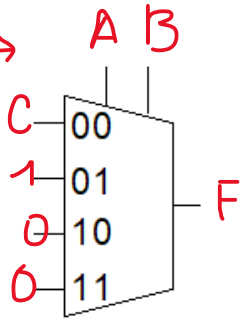
- Multiplexeurs
- Décodeurs
- ROM
- Implémentations des fonctions logiques avec les multiplexeurs, décodeur et OU logique, LUT avec ROM

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

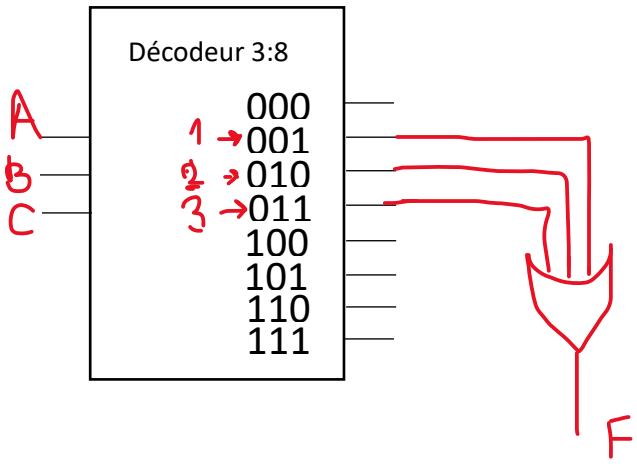


A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

A	B	F
0	0	C
0	1	1
1	0	0
1	1	0



A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0



A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0