Architecture des ordinateurs - Cours

Introduction

Nous allons voir ici comment fonctionne une "machine" informatique qui peut être un ordinateur personnel ou un smartphone, un ordinateur de bord de voiture, un robot ou encore un objet connecté.

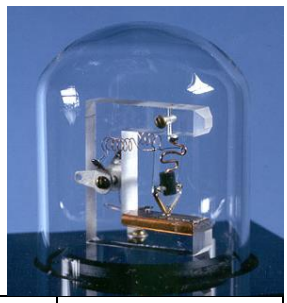
Nous allons nous pencher sur le premier modèle qui a été fait de ces machines informatiques : le modèle d'architecture séquentielle qui a été finalisé par Von Neumann en 1945. Ce modèle est simpliste au regard de la complexité et de la diversité des machines actuelles, néanmoins il permet de mieux appréhender comment fonctionne une machine informatique.

I Du transistor au modèle d'architecture séquentielle

I.1) Le composant électronique de base : le transistor (Hors Programme : pour culture générale)

Le transistor est le composant électronique de base d'un processeur. C'est, essentiellement, un interrupteur électronique. Composé de trois parties : source, grille et drain il fonctionne comme suit (description simplifiée) : selon qu'une tension est appliquée ou pas à la grille, le courant peut passer ou pas de la source vers le drain (c'est un peu comme un robinet pour le courant).

Autrement dit un transistor permet de contrôler le passage du courant entre la source et le drain. Il existe de nombreux procédés de fabrication et de nombreux types de transistors. On peut noter :



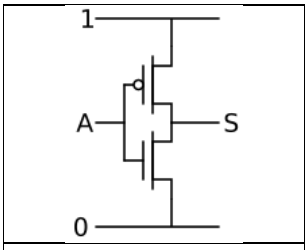
- Que le premier transistor (ci-contre), fabriqué en 1947, faisait plusieurs centimètres.
- Qu'ils sont utilisés en grand nombre dans les micro-processeurs :

Année	Nom	Nombre de transistors	Fréquence
1972	Intel 4004	2 300	740 kHz
1982	Intel 80286	134 000	6 MHz
1993	Intel Pentium	3 100 000	66 MHz
2003	AMD Athlon 64	105 900 000	3,2 GHz
2019	AMD Threadripper 2990WX	19 200 000 000	3 GHz
2022	Apple M1 Ultra	114 000 000 000	3,2 GHz

- Qu'ils sont également utilisés en grand nombre dans les RAM (137 milliards de transistors dans la SDRAM de 128 Gb de Samsung en 2018, 2048 milliards de transistors dans la mémoire Flash de 8Tb de Samsung en 2019)
- Que le seuil de réduction des dimensions – toujours nécessaire pour continuer d'augmenter les performances – sera atteint d'ici quelques années (seuil de gravure de 5 nm atteint en 2019, prévu à 3nm en 2023) et que d'autres approches technologiques que la réduction des dimensions devront être envisagées pour continuer à doubler la puissance des processeurs tous les deux ans (loi empirique de Moore). Ainsi – en 2019 – certains spécialistes doutent que l'on puisse dépasser le seuil de 3 nm.
- Que cette réduction des dimensions des transistors conduit les industriels leaders du secteur (Intel, TSMC, Samsung…) à faire des investissements colossaux (plusieurs dizaines de milliards d'euros) pour poursuivre leur développement.

I.2) Portes logiques (Hors Programme : pour culture générale)

Voilà ci-contre comment on peut construire électroniquement l'opérateur NOT (on parle de *porte logique* NOT lorsqu'on s'intéresse au montage électronique) à partir de deux transistors. Plus généralement tous les opérateurs booléens usuels (NOT, AND, OR, XOR, NAND …) peuvent se réaliser électroniquement sous forme de portes logiques construites à partir de transistors.



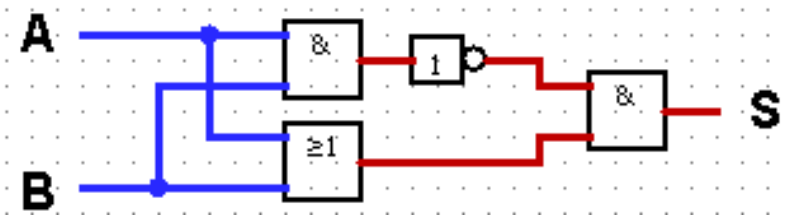
Porte logique NOT

I.3) Circuits combinatoires et fonctions booléennes

Lorsqu'on assemble des *portes logiques* entre elles on peut obtenir des *circuits combinatoires* qui comportent plusieurs entrées booléennes (généralement notées A, B, C …) et une seule sortie booléenne (généralement notée S). La table de vérité obtenue correspond au tableau de valeurs d'une *fonction booléenne*.

Exemple :

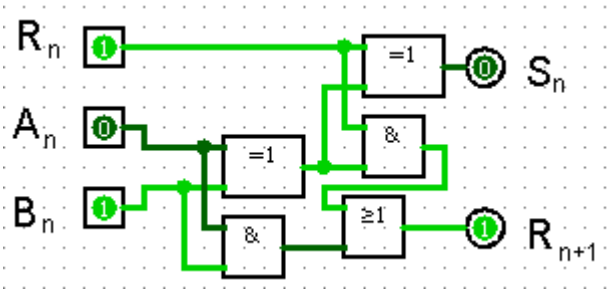
a) Compléter la table de vérité du circuit combinatoire ci-dessous :



A	B	S

b) À quel opérateur booléen cela correspond-il ?

En assemblant plusieurs circuits combinatoires, on obtient facilement plusieurs sorties. On peut par exemple assez aisément obtenir un additionneur binaire. La table de vérité de l'assemblage de portes logiques ci-contre (qui comporte des portes XOR) correspond bien à la table de vérité de l'additionneur binaire à 1 bit (pour rappel aller voir le cours sur les booléens).



r_n	a_n	b_n	s_n	r_{n+1}
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Et en assemblant plusieurs circuits ainsi en cascade, on obtient un additionneur binaire sur 4, 8, 16 bits au choix. Vous pourrez consulter le fichier Logisim 'additionneur_4_bits.circ' disponible sur progalgo pour jouer un peu avec.

En résumé de ce I.3), les portes logiques permettent d'obtenir des circuits combinatoires réalisant des fonctions booléennes qui permettent d'effectuer des additions binaires et, plus généralement, toute une gamme d'opération arithmétiques et logiques (multiplications, divisions, comparaisons, test d'égalité etc.)

I.4) Autres circuits usuels (Hors Programme : pour culture générale)

Vous trouverez sur progalgo deux circuits fondamentaux dans l'architecture d'un ordinateur.

Premier circuit : circuit combinatoire pouvant aiguiller une donnée

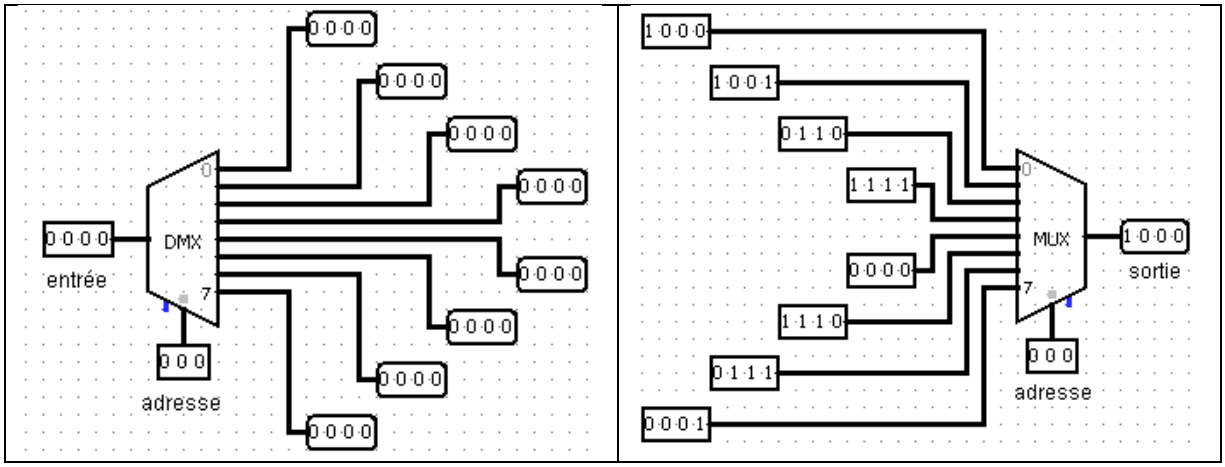
Le démultiplexeur permet, grâce à une adresse binaire entre 0 et 7, de choisir sur quelle sortie entre 0 et 7 on va envoyer la donnée située sur l'unique entrée. Il existe aussi le multiplexeur : on a cette fois-ci une seule sortie et des entrées de 0 à 7. Grâce à une adresse entre 0 et 7, on va sélectionner quelle entrée va être envoyée sur la sortie.

On comprend bien leur utilité : dans un circuit électronique les données circulent d'un endroit à l'autre et partagent des "routes" en commun (on parle de *bus de données*). Il y a donc des "carrefours" auxquels on doit aiguiller les données. C'est le rôle des démultiplexeurs et des multiplexeurs.

Exemples : .

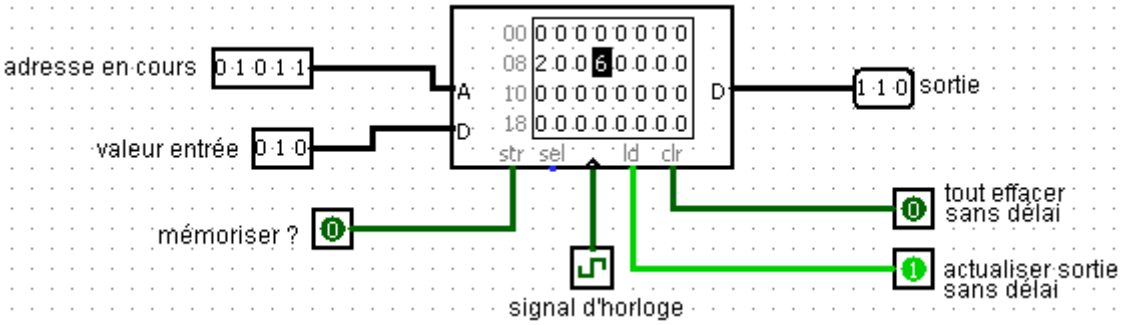
a) Sur le démultiplexeur ci-dessous à gauche, on met l'entrée à 0110 et l'adresse à 110. Avec un crayon rouge, modifier la valeur de la sortie concernée.

b)Sur le multiplexeur ci-dessous à droite, on met l'adresse à 100. Avec un crayon rouge, modifier la valeur de sortie.



Second circuit : circuit séquentiel pouvant mémoriser une donnée

On parle ici de circuit *"séquentiel"* car l'état de la sortie (0 ou 1) va dépendre de l'état actuel des entrées mais aussi de l'état des entrées précédentes (autrement dit de la *séquence* d'instructions précédentes). Sachez juste qu'avec des portes logiques on peut réaliser facilement des mémoires de 1 bit. Et qu'en assemblant ces mémoires en cascade on obtient facilement des matrices de RAM.



Ce qu'il faut retenir dans la gestion d'une mémoire c'est qu'il y a des données à gérer, mais il faut aussi des instructions (mémoriser, effacer, lire) ainsi que des adresses indiquant l'emplacement en cours.

I.5) Bilan du I : 3 fois 3 informations à retenir

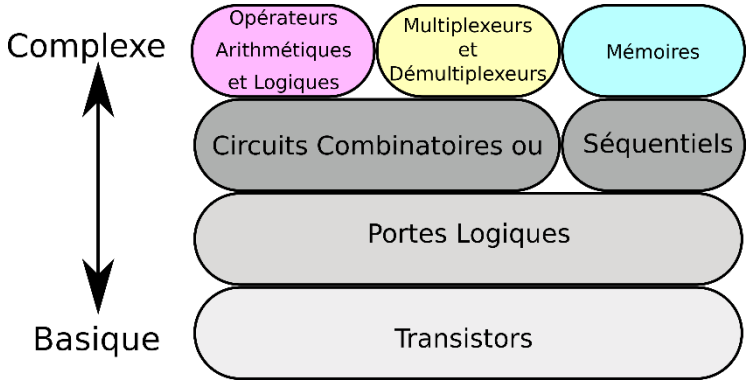
Les circuits combinatoires – qui réalisent des fonctions booléennes – peuvent être réalisés à partir de portes logiques, elles-mêmes pouvant être réalisées à partir de transistors.
En assemblant des circuits combinatoires on obtient deux catégories principales d'unités fonctionnelles :

- **Les opérateurs arithmétiques et logiques qui permettent d'effectuer du calcul binaire (addition, multiplication, opérations booléennes etc.) ainsi que des comparaisons (test d'égalité, de supériorité etc.).**
- **Les multiplexeurs et démultiplexeurs qui permettent de gérer les flux de données en agissant comme des aiguillages sur les bus de données.**

Les circuits séquentiels - pouvant également être construits à partir de portes logiques - permettent quant à eux de réaliser un autre type d'unités fonctionnelles :

- **Les mémoires.**

Ces trois sortes d'unités fonctionnelles sont les briques élémentaires qui permettent de construire une machine informatique. Pour résumer tout cela on peut regarder le schéma suivant :



Les octets - ou les mots de 16, 32 ou 64 bits – qui circulent dans ces circuits se ressemblent tous mais ils peuvent avoir – en simplifiant – trois rôles bien distincts :

- **Des valeurs issues de calculs, éventuellement à stocker en mémoire,**
- **Des adresses correspondant à des emplacements en mémoire ou à des E/S de (dé)multiplexeurs,**
- **Des instructions indiquant quelle est la tâche à effectuer.**

On peut facilement retenir ces trois rôles que peuvent avoir les mots d'une machine informatique grâce à l'instruction correspondant à une mise en mémoire : "Transférer la valeur X à l'adresse Y".
Il pourrait y avoir un mot pour l'instruction "transférer", un mot pour la valeur "X" et un mot pour l'adresse "Y" ce qui nous donne trois octets (ou mots de 16, 32, 64 bits), un pour chacun des trois rôles.

Remarque : les expressions "circuits séquentiels", "multiplexeurs" et "démultiplexeurs" ne sont pas au programme.

II Modèle d'architecture séquentielle de Von Neumann

I.1) Présentation du modèle séquentiel

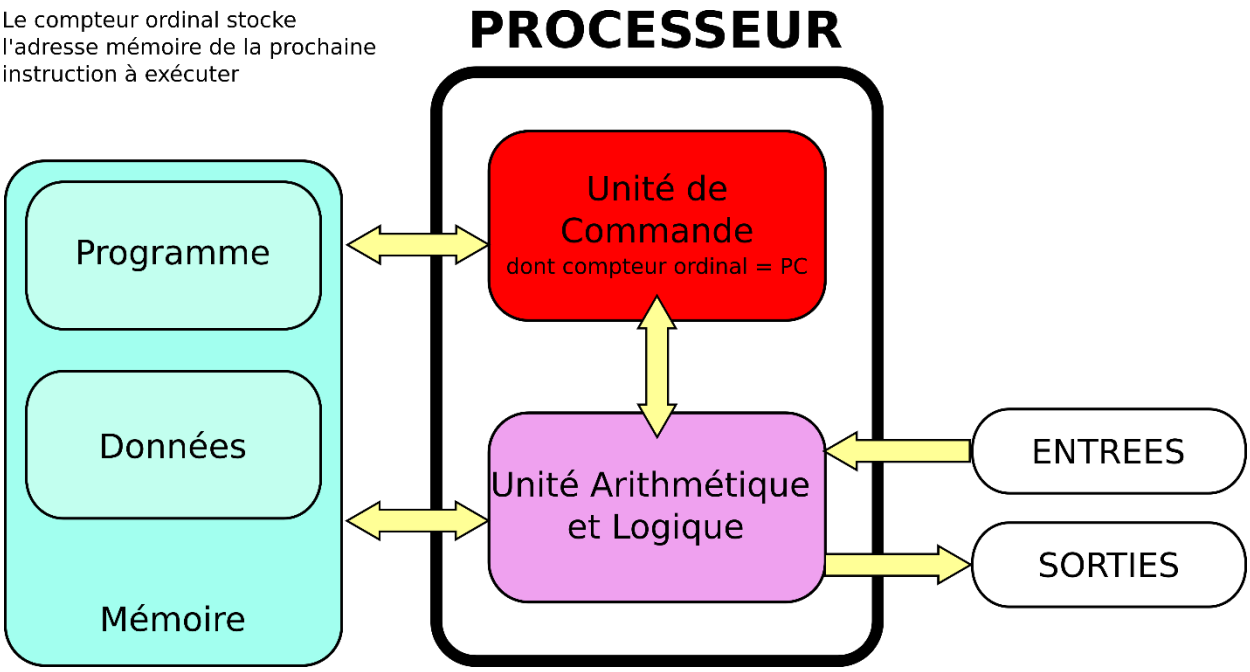
Le modèle d'architecture séquentielle de Von Neumann repose sur plusieurs choses : tout d'abord sur tout ce qui a été dit dans le bilan du I.5. On dispose de circuits électroniques capables de faire des calculs arithmétiques et logiques, capables de mémoriser des données ou encore capables d'aiguiller des données sur des bus de transfert.

La rupture créée en 1945 par ce modèle est liée à deux principes :

1. Le programme informatique à exécuter est situé dans la mémoire. Auparavant il fallait câbler physiquement chaque programme. Avec un programme stocké en mémoire on gagne en souplesse, en productivité etc. En conséquence directe les instructions (telles que mémoriser, additionner, comparer …) sont représentées exactement comme les autres données (valeurs ou adresses), typiquement par des octets (ou des mots de 16, 32 ou 64 bits). Mais cela nous l'avions déjà dit dans le bilan du I.5).

2. Le mécanisme – cadencé par une horloge – qui s'occupe de lire les instructions dans la mémoire les unes après les autres de façon *séquentielle* est séparé de la partie de la machine qui éxécute ces instructions.
   Ce mécanisme qui gère le déroulement du programme doit être capable si besoin de sauter une ou plusieurs instructions, voire de revenir en arrière dans la séquence d'instructions. En effet lorsqu'on fait un `if … then … else …`, on ne prend pas en compte certaines instructions. De façon similaire les boucles requièrent de revenir en arrière dans le programme. La partie d'une machine informatique qui s'occupe de la lecture séquentielle des instructions s'appelle "Unité de contrôle" alors que la partie qui s'occupe d'exécuter ces instructions s'appelle "Unité arithmétique et Logique". **Le fonctionnement de l'unité de contrôle est basé sur la mémorisation de l'adresse mémoire de la prochaine instruction à exécuter : cette mémorisation est effectué par le compteur ordinal ou registre PC (program counter)**

Nous pouvons maintenant présenter le modèle d'architecture séquentielle de Von Neumann ci-dessous.



On note que par rapport au schéma du I.5) les couleurs ont été respectées :
- Mémoire en bleu clair
- UAL en rose clair
- Transferts d'informations via les bus en jaune clair

b) Comment font les programmeurs et ceux qui créent des langages de programmation ?

Normalement à ce stade, on pense avoir compris. Sauf qu'il y a deux questions que certains ne vont pas manquer de se poser.

- On a vu que les instructions nécessaires pour gérer le processeur font par exemple intervenir les adresses des emplacements mémoire. Pourquoi, en python, on ne s'occupe pas des adresses des emplacements mémoire ?

- Il y a plein de langages de programmation différents et plein de processeurs différents, donc le modèle unificateur de Von Neumann voudrait dire que chaque type de processeur peut être contrôlé par n'importe quel langage de programmation ? Et réciproquement ?

En fait la gestion d'une machine informatique s'effectue par couches successives de langages de programmation. Les langages de haut niveau (comme Python) ne communiquent pas directement avec le processuer mais avec des langages intermédiaires qui à leur tour communiquent avec des langages proches du processeur. En simplifiant abusivement on a :

- La couche microarchitecture qui est la plus basique de toutes, située au cœur de la machine informatique. Pour faire simple, une microinstruction ne prend souvent qu'un tout petit nombre de cycles d'horloge.

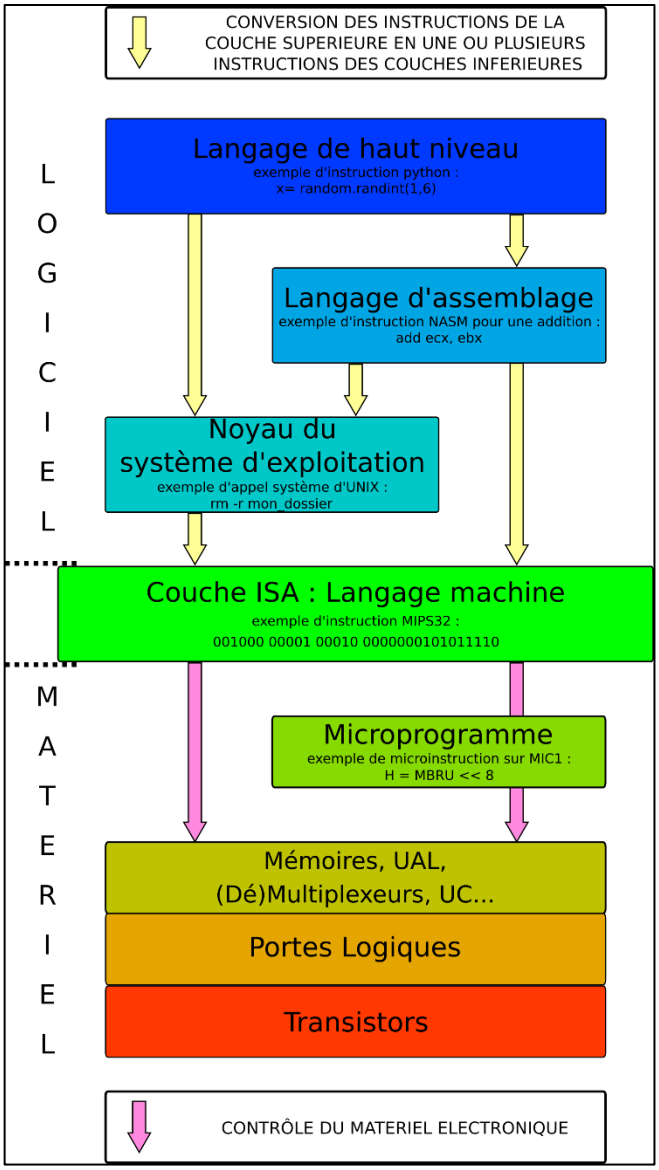
- La couche décrivant chaque famille de processeurs (jeu d'instructions ou couche ISA ou langage machine). Plusieurs machines matérielles peuvent partager le même jeu d'instructions. Un jeu d'instructions donné est ce qu'on appelle le langage machine. Il s'agit d'un langage très sommaire : les instructions ne sont pas écrites en caractères alphabétiques mais en binaire ! Chacune des instructions en langage machine correspond à plusieurs microinstructions successives.

- La couche Noyau des systèmes d'exploitation exploitant la couche ISA : ils offrent des instructions (dits « appels système » : par exemple impression, affichage, accès aux fichiers etc.) en caractères alphabétiques qui correspondent à des blocs de code en langage machine. Autrement dit un appel système correspond bien souvent à un très grand nombre d'instructions en langage machine. La couche Noyau des OS simplifie donc la tâche des programmeurs qui l'utilisent.

- La couche des langages d'assemblage. Ce sont des langages basiques pour lesquels les instructions sont en caractères alphabétiques. Ces instructions reposent pour certaines sur la couche OS (appels système) et pour d'autres directement sur la couche ISA (langage machine).

- La couche des langages d'application tels que ceux que vous connaissez (python, C++, Ocaml etc.) qui exploite la couche d'assemblage ou directement la couche Noyau des systèmes d'exploitation. Ces langages sont évolués, une seule de leurs instructions correspond parfois à plusieurs milliers de lignes de code en langage d'assemblage. Ils permettant de développer rapidement des applications.

Remarque :
Cette description en couches est très simplificatrice. Certains langages ne rentrent dans aucune de ces couches (par exemple le langage C, de bas niveau mais qui n'est pas de l'assembleur).



Exemples de couches ISA :

L'ISA ARM-V8 et ses variantes est une couche ISA utilisée dans des smartphones (iPhone avec processeurs A7 à A12, Samsung avec processeurs M1/M2/M3, smartphones avec Qualcomm Snapdragon etc.).
Les systèmes d'exploitation 64 bits suivants peuvent être installés sur cette architecture ARM-V8 : iOS, Android, Ubuntu, Debian mais aussi Windows 10.

L'ISA X86-64 est implémentée sur des processeurs AMD Athlon 64, AMD Phenom II, AMD Ryzen/Epic, Intel Pentium, Intel celeron, Intel Core i3 à Intel Core i9 etc.
De nombreux systèmes d'exploitation savent l'exploiter : BSD, Linux, macOS, Windows, PS4, Xbox One etc

Exemple des différentes versions d'un même programme selon les couches :

Voir en annexe pour un programme informatique avec ses versions :
- en java (couche de haut niveau)
- converti en assembleur IJVM (couche assemblage)
- converti en langage machine (couche ISA)
- quelques exemples d'instructions en langage machine converties en micro-code de l'architecure MIC1 (couche Microprogramme)

Vous allez remarquer que le programme de haut niveau de quelques lignes finit en programme de bas niveau de … plusieurs dizaines de lignes.

Analogie avec des recettes de cuisine :

- un langage de haut niveau pourrait comprendre ce que signifie une instruction telle que "Faire une mousse au chocolat pour 6 personnes"
- un langage assembleur pourrait comprendre des instructions plus basiques telles que "Faire fondre le chocolat au bain-marie" ou "Battre les blancs en neige" (mais ne comprendrait pas "Faire une mousse au chocolat")
- un langage en micro-instructions pourrait comprendre des instructions encore plus basiques telles que "Prendre le fouet dans la main gauche" ou "Ouvrir le robinet d'eau froide avec la main droite" (mais ne comprendrait pas "faire fondre le chocolat au bain-marie")

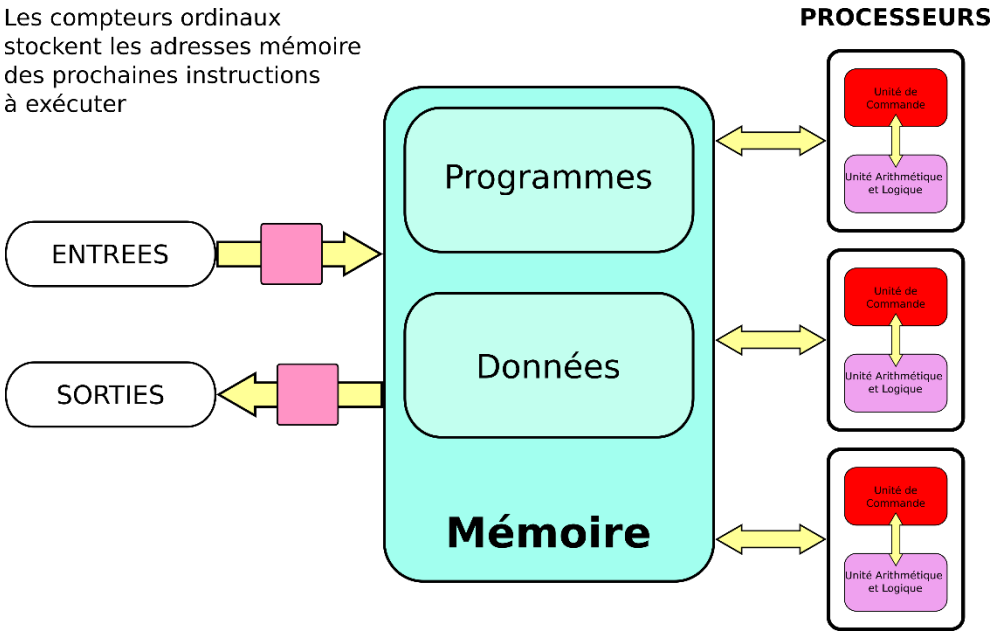
En quelque sorte c'est le niveau de détail qui change quand on change de couche.

c) Les architectures multi-processeurs

Il y aujourd'hui deux évolutions par rapport au modèle de Von-Neumann élaboré en 1945.

- Les entrées-sorties sont désormais gérées par des petits processeurs autonomes, ce qui est associé à un système de partage de la mémoire entre plusieurs programems.
- Les ordinateurs comportent maintenant des processeurs multiples à l'intérieur d'une même puce.

En conséquence c'est désormais la mémoire qui est plutôt au centre de l'ordinateur avec un haut degré de parallélisme dans les flux de données à travers les bus.



d) Bilan

Le modèle d'architecture *séquentielle* est basé sur :

- **une *mémoire* contenant des données et les programmes à exécuter**
- **une *unité arithmétique et logique* chargée d'exécuter les instructions**
- **une *unité de contrôle (ou unité de commande)* chargée de gérer le *séquençage* des instructions, en particulier les sauts conditionnels ou les boucles grâce au *compteur ordinal (ou registre PC)* qui stocke l'adresse mémoire de la prochaine instruction à exécuter.**
- **des *bus* chargés d'effectuer les transferts d'informations entre les différentes unités**

Pour chaque machine informatique, un jeu d'instructions ou langage machine – souvent partagé par plusieurs modèles différents de machines – décrit son fonctionnement. Chaque sorte de langage machine (ARM-V8, X86-64 etc.) peut être est pris en charge par des systèmes d'exploitation différents (Linux, Android, Windows 10 …). Sur un système d'exploitation donné on peut implémenter un langage assembleur (de bas niveau, proche du langage machine) ou un langage d'application (de haut niveau, tel que python).

Nous allons maintenant nous intéresser à du langage assembleur, c'est-à-dire à un langage proche du langage machine. Cela va nous permettre d'appréhender le fonctionnement des unités arithmétiques et logiques.

Annexe :

Code écrit en Java (haut niveau) :

```
int U, r, s, n;
n=0;
U=3;
r=2;
s=100;
while (U<s) {
  n +=1;
  U += r;
}
```

Converti en IJVM (assembleur):

```
.var
U
r
s
n
.end-var
BIPUSH 0
ISTORE n
BIPUSH 3
ISTORE U
BIPUSH 2
ISTORE r
BIPUSH 99
ISTORE s
BIPUSH 1
debut: NOP
IFLT fin
ILOAD U
ILOAD r
IADD
ISTORE U
ILOAD n
BIPUSH 1
IADD
ISTORE n
ILOAD s
ILOAD U
ISUB
GOTO debut
fin: POP
```

Converti en langage machine MIC 1

```
0xb6 0x00 0x01 0x00 0x01 0x00 0x04 0x10 0x00 0x36 0x04 0x10
0x03 0x36 0x01 0x10 0x02 0x36 0x02 0x10 0x63 0x36 0x03 0x10
0x01 0x00 0x9b 0x00 0x19 0x15 0x01 0x15 0x02 0x60 0x36 0x01
0x15 0x04 0x10 0x01 0x60 0x36 0x04 0x15 0x03 0x15 0x01 0x64
0xa7 0xff 0xe9 0x57 0x15 0x04 0x15 0x01
```

Exemples d'intructions en langage machine converties en microcode MIC 1

```
(0x60 - IADD)
iadd1: MAR=SP=SP-1; rd
H=TOS
MDR=TOS=MDR+H; wr; goto Main1

(0x10 - BIPUSH)
bipush1: SP=MAR=SP+1;
PC=PC+1; fetch
MDR=TOS=MBR; wr; goto Main1;

(0x15 - ILOAD)
iload1:H=LV;
MAR=MBRU+H;rd;
iload3: MAR=SP=SP+1;
PC=PC+1; fetch; wr;
TOS=MDR; goto Main1

(0x36 - ISTORE)
istore1: H=LV
MAR=MBRU+H
istore3: MDR=TOS;wr;
SP=MAR=SP-1;rd
PC=PC+1;fetch
TOS=MDR; goto Main1

(0xA7 - GOTO)
goto1:OPC=PC-1
goto2: PC=PC+1; fetch;
H=MBR <<8
H=MBRU OR H
PC=OPC+H; fetch
goto Main1
```

Les "conversions" indiquées ci-contre sont en fait des extraits d'un microprogramme d'environ 120 lignes de code. Certaines instructions plus complexes (telles que 0x9b – IFLT qui permet notamment de faire les boucles) ne sont pas données ici car elles utilisent l'intégralité du microprogramme.

On pourrait dire que ce microprogramme est le cœur de l'unité de commande.