


Leçon 24: Principe de fonctionnement des ordinateurs : architecture, notions d'assembleurs
Pré-requis: Représentation des nombres en binaires

I- Circuits booléens

I-1) Porte logique

Les circuits d'un ordinateur manipule des bits qui correspondent en interne à des tensions électriques.

Définition 1: Une porte logique est une fonction qui prend un ou plusieurs bits en entrée et qui renvoie un bit en sortie.

Schéma 2: 

Propriété 3: On peut composer des portes logiques, et réaliser un fil: on crée alors des circuits booléens.

Attention, on ne doit pas créer de boucles dans les circuits.

Exercice 4: Exprimer la porte OR à l'aide des portes NOT et AND.

I-2) Expressivité

Définition 5: On définit inductivement l'ensemble \mathcal{EB} des expressions booléennes par:

Cas de base: $1, \top, x \in V$ (où V est un ensemble dénombrable de variables)

Constructeurs: \neg uneaire, \wedge et \vee binaires.

Remarque 6: Cela représente exactement les circuits booléens, où seuls les fils initiaux peuvent être dupliqués.

Définition 7: Une valuation est une fonction $\sigma: V \rightarrow \{0, 1\}$

Définition 8: On définit $[\cdot]_\sigma: \mathcal{EB} \rightarrow \{0, 1\}$ inductivement sur \mathcal{EB} par: $[1]_\sigma = 0$, $[\top]_\sigma = 1$, $[x]_\sigma = \sigma(x)$ pour $x \in V$

$$[e_1 \wedge e_2]_\sigma = \max([e_1]_\sigma, [e_2]_\sigma)$$

$$[e_1 \vee e_2]_\sigma = \min([e_1]_\sigma, [e_2]_\sigma)$$

$$[\neg e_1]_\sigma = 1 - [e_1]_\sigma$$

Remarque 9: Cela revient à simuler l'exécution d'un circuit booléen.

Théorème 10: Pour toute fonction $f: \{0, 1\}^m \rightarrow \{0, 1\}$, il existe une expression booléenne e ayant pour variables $\{x_1, \dots, x_m\}$ tel que pour tout $(b_1, \dots, b_m) \in \{0, 1\}^m$, en prenant σ tel que $\sigma(x_i) = b_i$ on ait alors $[e]_\sigma = f(b_1, \dots, b_m)$.

Développement 1: Preuve du théorème 10 et discussion autour de la complexité.

Conclusion 11: Les circuits booléens permettent d'exprimer toutes les fonctions que l'on pourrait vouloir calculer.

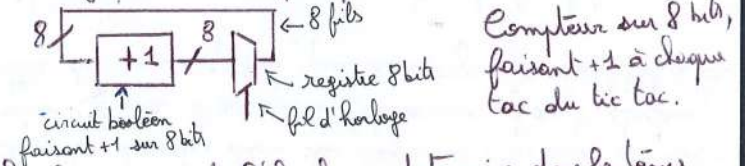
I-3) Introduction du temps

Notre ordinateur est donc composé de circuits booléens. Néanmoins, on voudrait pouvoir brancher les circuits booléens entre eux (ce qui pose problème car les portes logiques ne changent pas de valeurs instantanément et avoir de la rétroaction (ce qui est interdit)).

On introduit alors des briques de mémoire dans un ordinateur (registre) et une horloge (un tic tac). Les liens entre les différents circuits ne se font alors qu'à travers des registres, qui se mettent à jour en même temps grâce à l'horloge. Ainsi, on a jamais de réelles boucles.

Remarque 12: Les registres sont les plus petites unités de mémoire d'un ordinateur.

Exemple 13:



Remarque 14: La fréquence de l'horloge détermine donc le temps minimal pour faire une opération dans un ordinateur. C'est ce que l'on dit quand on parle de processeur 4 GHz (4 milliards de tac par secondes).

II - Modèle de Von Neuman

II-1) Le modèle

Définition 15 : Une instruction est une opération à effectuer par le processeur sur des éléments de mémoire de l'ordinateur (registre, cache, RAM, disque dur, etc...).

Principe 16 : Un ordinateur passe donc son temps à exécuter des instructions (qui modifie l'état de la mémoire).

Définition 17 : Le modèle de Von Neumann décrit le fonctionnement d'un ordinateur constitué :

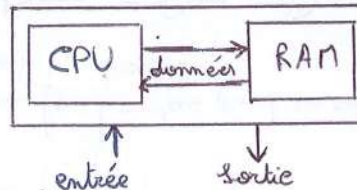
- d'un processeur qui lit les instructions en mémoire et les exécute. Il accède à la mémoire par blocs appelés mots mémoire. Pour cet accès, le processeur utilise un registre appelé compteur ordinal (Program Counter ou PC) qui contient une adresse mémoire

- La mémoire RAM adressée qui contient les programmes à exécuter et les données

- Les périphériques d'entrée (clavier, souris, disque dur) et de sortie (écran, disque dur, Haut-parleurs)

Remarque 18 : Une des spécificités de ce modèle est que les instructions sont des données comme les autres.

Schéma 19 : Modèle de Von Neumann
CPU = processeur



Définition 20 : (Cycle de Von Neumann)

- 1) Lire le mot mémoire qui commence à l'adresse PC
- 2) Interpréter ce mot mémoire comme une instruction et l'exécuter
- 3) Augmenter le PC pour passer à l'instruction suivante puis 1)

Remarque 21 : Quand vous avez plusieurs cœurs, il y a simplement plusieurs processeurs en parallèle.

II-2) Registres et mémoire adressable

Dans le modèle, la mémoire n'est pas dans le processeur. C'est la mémoire adressable (accessible par adresse).

Néanmoins, il y a aussi de la mémoire dans le processeur. Ce sont les registres. Quand un processeur veut exécuter une opération, il doit d'abord stocker les données dans des registres (les avoir en mémoire, sous la main), faire l'opération (et comme il doit stocker le résultat, le mettre dans un registre), puis renvoyer le résultat à la mémoire.

Remarque 22 : Un processeur possède un registre stockant la valeur de PC.

III - Jeu d'instruction

III-1) Définition

Définition 23 : Une instruction machine est une séquence de bits que le processeur peut interpréter et exécuter. Un jeu d'instruction (ISA) définit quelles sont les instructions supportées par le processeur, et la façon dont elles sont représentées en mémoire. L'ensemble de ces instructions machines forment le langage reconnu par le processeur, appelé langage machine.

Remarque 24 : Tous les codes dans n'importe quel langage de programmation (exemple python) doivent être traduits en langage machine pour être exécutés par le processeur.

Propriété 25 : En général, les jeux d'instructions gèrent les opérations suivantes :

- lire le contenu d'une case mémoire dans un registre, écrire le contenu d'un registre dans une case mémoire.
- opérations arithmétiques ou logiques (addition, et bit à bit, etc.)
- se déplacer dans le code du programme que l'on exécute (saute).

Remarque 26: Il existe de nombreuses ISA différentes. On peut les diviser en deux catégories principales:

CISC (complex instruction set computer): plus d'instructions pouvant faire plus de choses mais donc plus longues (ex. x86 sur la plupart des ordinateurs)

RISC (reduced instruction set computer): moins d'instructions mais plus rapides et faciles à implémenter (ex: RISC-V, ARM sur des téléphones).

II-2) Langage assembleur

Une instruction est représentée en mémoire par un code en binaire. Par exemple, si les trois premiers bits sont des 0, on doit faire une opération arithmétique, puis si les deux suivants sont des 1, on fait une addition, etc. Néanmoins, cela est très peu lisible par l'être humain.

Définition 27: Le langage assembleur représente le langage machine sous une forme lisible par un humain. C'est le langage de programmation le plus bas niveau.

Exemple 28: Exemple d'instruction classique

• `load r_i [r_j]`: met le contenu à l'adresse contenu dans r_j

dans le registre r_i

• `store r_i [r_j]`: met le contenu du registre r_i dans la case à l'adresse contenu dans r_j

• `add r_i r_j r_k` : ajoute le contenu des registres r_i et r_j pour le mettre dans le registre r_k .

• `iload r_i x` : met la valeur x dans le registre r_i

• `mv r_i r_j` : met la valeur du registre r_i dans le registre r_j

Exemple 29: Traduction en langage assembleur du code

Python `"z = x + y"`

`iload r_1 (adresse de x)`

`load r_2 [r_1]`

`iload r_1 (adresse de y)`

`load r_3 [r_1]`

`add r_2 r_2 r_3`

`iload r_1 (adresse de z)`

`store r_2 [r_1]`

Remarque 30: Pour pouvoir exécuter des boucles et les si, on introduit de nouvelles instructions:

• `bge r_1 r_2 N`: va à la ligne N si $r_1 \geq r_2$

• `hgt, ble, beg, blt` (pour $>$, \leq , $=$, $<$).

• `jump N`: va à la ligne N du code

Exemple 31: Pour while ($i < m$): $i = i + 1$

0. `iload r_1 (adresse de i)`

1. `iload r_2 (adresse de m)`

2. `load r_2 [r_1]` // contient m

3. `load r_3 [r_1]` // contient i

4. `bge r_3 r_2 9`

5. `iload r_4 1`

6. `add r_3 r_3 r_4`

7. `store r_3 [r_1]` // mettre à jour i

8. `jump 3`

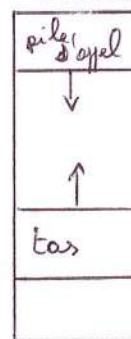
9.

Remarque 32: On peut faire beaucoup d'optimisation (par exemple en restockant i qu'à la fin). C'est d'important sujet d'études

IV- Gestion de la mémoire à plus haut niveau

Quand on exécute un programme qui n'est pas dans un langage assembleur, on a souvent besoin d'instructions de plus haut niveau, nous permettant d'allouer des variables, de faire des appels de fonctions imbriqués, etc... ce qui n'est a priori pas disponible dans le langage assembleur tel quel.

Principe 33: Lors de l'exécution d'un processus, un espace mémoire en RAM lui est réservé.



← gère les variables locales, les appels de fonctions (notamment pour la récursion), etc.
↓ adresses

← gère les données accessibles depuis tout le programme (malloc)

Développement 2:
Explication et implémentation de la pile d'appel en assembleur.