Université d'Aix-Marseille III - Licence math-info 1ère année

I2 - Résumé du cours (1) : Architecture matérielle

Le but de cette première moitié du cours est de faire comprendre le fonctionnement interne d'un ordinateur afin de savoir ce qui passe concrètement lorsqu'un programme est exécuté.

1 L'ordinateur selon différents points de vue

Un ordinateur est constitué de composants électroniques interconnectés parmi lesquels on trouve un microprocesseur, une mémoire, etc. Chaque modèle de microprocesseur sait exécuter des programmes dans un langage qui lui est propre et qu'on appelle *langage machine*. Tous les programmes écrits dans un langage, quel qu'il soit (C, Pascal, etc), sont traduits (compilés) en un programme équivalent en langage machine afin d'être exécutable par le microprocesseur. Dans la première partie de ce cours, nous allons étudier selon 3 points de vue ce qui se passe lorsqu'un programme est exécuté : du point de vue du langage évolué dans lequel a été écrit ce programme (on utilisera le langage C), du point de vue du langage machine et du point de vue physique/électronique.

1.1 Les données selon différents points de vues

Rappel : informatique = traitement automatique de l'information. Lorsqu'on veut faire effectuer une tâche par un ordinateur, on fournit des *données en entrée* qui sont traitées par la machine pour fournir des *données en sortie* qui constituent le résultat cherché. Ce schéma **données en entrée → traitement → données en sortie** se retrouve à tous les niveaux de l'informatique : matériel, électronique, logiciel, etc.

Selon le point de vue où l'on se place (le langage évolué, le langage machine ou l'électronique), ces données sont de nature différente. Un langage évolué peut manipuler des données de différents types : entiers, flottants, chaînes de caractères, tableaux, données structurées, images, sons, etc. Le langage machine ne manipule que des entiers. Les circuits électroniques ne traitent que des états binaires. Nous allons maintenant voir le rapport entre ces différentes natures de données.

1.2 Relation entre états binaires et entiers

Les circuits électroniques d'ordinateur possèdent un certain nombre de fils électriques en entrée et en sortie. Ces circuits sont conçus pour qu'on applique sur leurs fils d'entrée deux états possibles : une tension comprise entre 0 et 1 volt (état 0) ou une tension comprise entre 2 et 5 volts (état 1). Le fait qu'il n'y ait que deux états possibles (0 ou 1) en font des *circuits logiques*. En fonction de l'ensemble des états d'entrée, le circuit va restituer un ensemble d'états binaires (correspondants aux mêmes intervalles de tensions) en sortie. Si on regroupe l'ensemble des états binaires d'entrée ou de sortie dans un certain ordre, on obtient une représentation de l'ensemble de ces états sous la forme d'un nombre binaire à plusieurs chiffres. On peut donc aussi considérer qu'un circuit est l'application d'une fonction numérique entière : un entier est fourni en entrée, une fonction lui est appliquée et on obtient l'entier résultat en sortie. De même, la mémoire principale (qui est aussi un circuit) mémorise des états électroniques binaires groupés par paquets de taille fixe que l'on considère comme des entiers.

1.3 Relations entre entiers et données des langages évolués

Quel que soit son type, toute donnée traitée par un ordinateur est *codée* sous forme d'un entier. La mémoire ne contient que des entiers et c'est le contexte d'interprétation de ces entiers qui va leur donner leur signification. Ainsi, l'entier 97 pourra représenter *selon le contexte* soit le nombre 97, soit le caractère `a` en code ASCII, soit une instruction du langage machine, soit le code de couleur d'un pixel de l'écran, etc.

Le fait que les programmes soient représentés sous forme d'entiers est essentiel et constitue ce qui différencie un ordinateur d'une machine dédiée à une fonction particulière (calculatrice, machine à laver). Un programme stocké en mémoire est aussi une donnée manipulable de la même manière que n'importe quel autre type de donnée.

2 Architecture matérielle générale d'un ordinateur

A un niveau **matériel**, on a des composants qui permettent soit de saisir physiquement, soit de traiter/transformer, soit de restituer physiquement des données. Un ordinateur est composé d'un microprocesseur, d'une mémoire principale et d'un ensemble de dispositifs d'entrée/sortie. Ces composants sont reliés par des *bus*, c'est-à-dire des ensembles de lignes électriques de communication.

2.1 Le microprocesseur

Le *microprocesseur* est le composant principal d'un ordinateur. Le microprocesseur (μP) permet d'exécuter un programme stocké dans la mémoire principale. Le μP permet aussi de communiquer avec les dispositifs d'entrée/sortie afin de fournir des données d'entrée au programme et afin que le programme transmette ses résultats à l'utilisateur.

Le μP est composé lui-même d'une *unité de commande*, d'une *unité arithmétique et logique* (UAL) et de *registres*. L'unité de commande est chargée des transferts de données et d'instructions entre le μP et la mémoire. L'UAL est chargée d'exécuter les instructions. Les registres sont en quelque sorte l'équivalent des variables simples dans les langages évolués. Ils servent à stocker des informations utiles au fonctionnement du μP.

2.2 La mémoire principale

La mémoire principale permet de stocker les programmes et les données.

L'unité d'information est le *bit*, contraction de "binary digit", c'est-à-dire chiffre binaire. Un bit ne peut prendre que la valeur 0 ou 1. Comme on veut pouvoir manipuler des informations autres que binaires, on a pris l'habitude de regrouper les bits par paquets de 8 qu'on appelle *octet*. Un octet permet donc de représenter une information qui peut prendre jusqu'à 256 états différents. Afin de traiter de l'information plus rapidement, selon les microprocesseurs les octets sont eux-même regroupés par paquets de 2, 4 ou 8 octets pour être traités en une seule fois par les instructions. Ces regroupements d'octets s'appellent des *mots*. La taille de ces mots est représentative de la puissance du microprocesseur (Z80 (Zilog), i8088 (Intel): 8 bits ; i80286, MC68000 (Motorola) : 16 bits ; i80386, MC68030, Pentium : 32 bits).

Lorsqu'on veut parler de grande quantité d'octets, on utilise les termes kilo-octet (Ko), méga-octet (Mo) ou giga-octet (Go). 1 Ko est égal à 1024 (= 2^{10} octets et non pas 1000 octets). De même, 1 Mo est égal à 1024 Ko et 1 Go à 1024 Mo.

Chaque octet de la mémoire possède une *adresse*, c'est-à-dire un numéro qui permet de le situer en mémoire. On peut faire le parallèle entre mémoire et tableau à une dimension d'entiers, entre adresse d'un octet et indice d'un entier du tableau.

La mémoire principale est divisée en deux parties : la mémoire vive (Random Access Memory, RAM) et la mémoire morte (Run Only Memory, ROM). La RAM est accessible en lecture et en écriture mais elle s'efface lorsque la machine est éteinte. La ROM n'est accessible qu'en lecture (on ne peut pas la modifier) mais elle persiste lorsque la machine est éteinte. On l'utilise notamment pour mémoriser le programme de démarrage de la machine et qui ne sert essentiellement qu'à charger le système d'exploitation dans la RAM.

2.3 Les mémoires secondaires

Comme la mémoire principale, les mémoires secondaires (ou mémoire de masse) permettent de stocker des données et des programmes. Elle permettent de mémoriser des quantités de données trop importantes

pour pouvoir toutes tenir en mémoire principale. Par rapport à la RAM et à la ROM, elles ont l'avantage d'à la fois ne pas être volatiles et d'être (souvent) accessibles en écriture (et en lecture). A quantité de stockage égale, leur prix serait plus faible qu'une RAM. Leur gros désavantage est leur temps d'accès aux données qu'elle mémorise, qui est bien supérieur à celui de la mémoire principale.

Parmi les différents types de mémoire secondaire, on trouve : les disques durs (capacité : plusieurs centaines de Go contre 1 Go ou plus pour une RAM), les disquettes (capacité : 1,44 Mo) dont l'intérêt est d'être facilement transportable, les CD-ROM (capacité : 650/700 Mo) enregistrables une seule fois (CD-R) ou réinscriptibles (CD-RW), les DVD-ROM (capacité : env 4 Go), les clés USB (capacité : jusqu'à 1 Go), etc.

3 Codage des informations sous forme d'entiers

Nous allons voir comment les types de données qu'un ordinateur traite sont représentés sous forme d'entiers.

3.1 Les notations décimales, binaires, hexadécimales

On note habituellement un entier en base 10, c'est-à-dire en utilisant 10 chiffres (de 0 à 9). Mais on peut aussi utiliser d'autres bases. Quand il pourra y avoir ambiguïté sur la base utilisée pour représenter un nombre, on utilisera la notation : $(x)_y$ qui signifie que la représentation x du nombre est en base y.

Comme l'unité d'information est le bit, on peut représenter un entier en base 2, comme un suite de bits. Ex : $(23)_{10} = (10111)_2$. Comme il peut être fastidieux de lire des entiers binaires, on utilise plutôt la notation hexadécimale (base 16) qui permet de regrouper 4 bits en un chiffre. Les chiffres hexadécimaux sont les chiffres habituels (0 à 9) et les lettres de A à F. Ex : $(111110)_2 = (3E)_{16}$.

Conversion du binaire en hexadécimal : on fait des regroupement de bits par paquets de 4 en partant de la droite. Chacun de ces paquets se traduit en un (seul) chiffre hexadécimal. Ex : $(101100100000111011)_2$ se découpe en 10 1100 1000 0011 1011, qui se convertissent en 2 C 8 3 D, pour former $(2C83D)_{16}$. La conversion inverse se fait très simplement en convertissant chaque chiffre hexadécimal en une suite de 4 bits.

Conversion du décimal en binaire : on détermine les chiffres binaires du plus à droite vers le plus à gauche en faisant une suite de divisions entières de l'entier noté en décimal. Lorsque l'entier est pair, on ajoute le chiffre 0. Quand l'entier est impair, on ajoute le chiffre 1. On s'arrête quand l'entier est nul.
Exemple :
203 → ..1
203/2=101 → ..11
101/2=50 → ..011
50/2=25 → ..1011
25/2=12 → ..01011
12/2=6 → ..001011
6/2=3 → ..1001011
3/2=1 → ..11001011
1/2=0 → 11001011

Conversion du binaire en décimal : notons b_i les chiffres binaires d'un entier x. $(b_{n-1}b_{n-2}...b_1b_0)_2 = (b_{n-1}.2^{n-1} + b_{n-1}.2^{n-1} + ... + b_1.2^1 + b_0.2^0)_{10}$. Ex : $(10110)_2 = 1 \times 16 + 0 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1 = (22)_{10}$.

Conversions entre décimal et hexadécimal : c'est le même principe qu'avec le binaire sauf qu'on est en base 16. Lorsqu'on passe du décimal à l'hexadécimal, on divise par 16 et on garde le modulo 16 pour avoir un chiffre. Lorsqu'on passe de l'hexadécimal au décimal, on additionne les puissances de 16.

3.2 Codage des nombres

Un entier naturel occupe une certaine place en mémoire en fonction de sa valeur. Lorsqu'on veut stocker des entiers naturels en mémoire, il faut prévoir la valeur maximale qu'ils peuvent avoir afin de décider du nombre d'octets nécessaire pour les mémoriser. Dans les langages évolués (tels que C), lorsqu'on déclare une

variable entière, on définit en fait le nombre d'octets nécessaires pour mémoriser sa valeur. En C, le type `char` permet de stocker un octet, le type `short` deux octets, le type `int` quatre octets. Avec un octet (resp. deux octets, resp. quatre octets), on peut représenter un entier naturel compris entre 0 et 255 (resp. 65535, resp. $2^{32} - 1$).

Si on veut représenter un entier relatif, il faut prévoir un bit pour mémoriser le signe (+ ou -) de l'entier. C'est le bit de poids fort (le bit le plus à gauche) qui mémorise le signe : 0 pour un nombre positif et 1 pour un nombre négatif. Cependant le reste des bits ne correspond pas à l'entier sans son signe. Pour retrouver cet entier, on inverse les bits et on additionne 1 au nombre obtenu. Ex : codé sur un octet, -40 se représente par 11011000 car $(40)_{10} = (00101000)_2$, 00101000 en inversant les bits donne 11010111, et en ajoutant 1 on obtient 11011000. Avec un octet (resp. deux octets, resp. quatre octets), on peut représenter un entier relatif compris entre -128 et 127 (resp. entre -32768 et 32767, resp. -2^{31} et $2^{31} - 1$. Cette représentation des entiers négatifs permet d'effectuer des additions entre entiers relatifs exactement comme si c'étaient des entiers naturels.

Pour représenter un nombre à virgule dont le nombre de chiffres après la virgule est fini, ce qu'on appelle un *nombre à virgule flottante*, on l'exprime sous la forme d'un codage binaire $b_{n-1}...b_0$. A partir des bits b_i, on peut calculer le nombre à virgule $(-1)^s.m.2^e$, où $s = b_{n-1}$ est le bit de signe, $e = b_{n-2}.2^{n-2-k} + b_{n-3}.2^{n-3-k}... + b_k.2^0 - 2^{n-k-2} + 1$ est l'*exposant* et $m = 1 + b_{k-1}.2^{-1}...b_0.2^{-k}$ sont des nombres binaires. En codant les flottants sur 4 octets (type `float` du C), on a $n = 32$ et on prend $k = 23$. En codant les flottants sur 8 octets (type `double` du C), on a $n = 64$ et on prend $k = 52$. Ex : 10101011011001010101011001000000 se décompose en 1 01010110 11001010101011001000000, ce qui permet d'obtenir $s = 1$, $e = 2^6 + 2^4 + 2^2 + 2^1$ et $m = 1 + 2^{-1} + 2^{-2} + 2^{-5} + ...$ soit environ -1.37 10^{26}.

3.3 Codage de l'information des périphériques

Afin de fournir aux programmes des données en entrée et que le programme fournisse les données-résultat en sortie, il faut que le μPcommunique avec des appareils extérieurs appelés *périphériques*. Ces appareils sont pilotés par des circuits spécialisés qui se chargent d'envoyer ou de recevoir des données du périphérique.

3.3.1 Périphériques d'entrée

Le *clavier* permet d'entrer des données sous formes de caractères. Lorsqu'on appuie sur une touche, le circuit pilotant le clavier déclenche une interruption momentanée du programme en cours d'exécution le temps de lancer une procédure qui mémorise le code des touches appuyées ou relachées. En fonction des combinaisons de touches entrées au clavier (touches de contrôle (shift, ctrl, alt) + caractère), la procédure va aussi mémoriser le nouveau code ascii correspondant dans un espace mémoire réservé ou déclencher une nouvelle procédure (ex : Ctrl+Alt+Del pour lancer le gestionnaire de programmes sous Windows).

Le code ASCII (American Standard Code for Information Interchange) est le code standard qui attribue un numéro à chaque caractère affichable. Les premiers numéros (de 0 à 31) correspondent à des caractères de contrôle (ex : 9 pour la tabulation, 13 pour le retour à la ligne). Le 32 code l'espace. Les codes 65 à 90 codent les lettres majuscules et les codes 97 à 122 codent les lettres minuscules. A la base, le code ASCII ne code que 128 caractères et n'utilise donc que 7 bits donc un octet (dont un bit est inutilisé) pour mémoriser un caractère. Le code ASCII étendu permet de coder les caractères accentués et d'autres alphabets que le latin. Il a donc besoin de deux octets pour mémoriser le caractère.

La souris permet de fournir les informations concernant les déplacements et l'état des boutons de la souris. Le pilote de la souris peut par exemple recevoir de façon cyclique 3 octets : le premier indiquant le déplacement suivant l'axe des abscisses par rapport au cycle d'avant, le deuxième le déplacement suivant l'axe des ordonnées et le troisième l'état des boutons de la souris.

3.3.2 Périphériques de sortie

L'écran permet de visualiser les résultats d'un programme sous forme textuelle ou graphique. Ce qui est affiché à l'écran peut être vu comme un tableau à deux dimensions de points colorés appelés *pixels*

(contraction de “picture element”). Les pixels sont stockés en mémoire sous forme d’entiers représentant leur couleur. Ils sont rangés par lignes.

Les imprimantes fonctionnent à peu près selon le même principe que les écrans : avec une matrice de points.

Les haut-parleurs permettent de reproduire du son. Le son consiste en des vibrations de l’air. Ces vibrations peuvent être représentées par une courbe d’oscillations sinusoïdales dont l’amplitude et la fréquence est changeante. Plus l’amplitude est grande, plus le son est fort. Plus la fréquence d’oscillation est grande, plus le son est aigu. Pour représenter une courbe continue sous la forme d’une suite d’entiers, on procède à un échantillonage : à intervalle de temps régulier, on note la valeur de la courbe en cet instant. Plus la fréquence d’échantillonnage est grande, plus la fidélité de la reproduction du son est grande mais plus la quantité de mémoire nécessaire est grande.

3.3.3 Périphériques d’entrée/sortie

En plus des mémoires de masse dont on a déjà parlé, on peut citer le modem (modulateur/démodulateur) qui permet la communication entre plusieurs machines à travers les lignes téléphoniques. Ici, il s’agit de représenter des entiers sous la forme de sons. Pour transmettre une série de bits à travers une ligne téléphonique, il suffit de transmettre un signal sinusoidal (sonore) qui pourra avoir deux amplitudes (ou deux fréquences) bien distinctes possibles et qui pourra changer à intervalle régulier. La première valeur d’amplitude correspondra au 0 et la deuxième au 1.

4 L'ordinateur en tant qu'assemblage de circuits électroniques

Nous allons voir comment on peut entièrement définir un ordinateur (μP + mémoire) grâce à un assemblage de circuits électroniques logiques très simples.

4.1 Fonctions logiques

Une fonction logique est une fonction à n variables définie sur : $\{FAUX,VRAI\}^n \rightarrow \{FAUX,VRAI\}$. Nous utiliserons F pour le FAUX et V pour le VRAI. On définit une fonction logique grâce à une *table de vérité* qui indique la valeur résultat F ou V en regard de chacune des 2^n combinaisons possibles de F et de V pour les n variables. Voici des exemples de fonctions logiques courantes :

a	NON a
F	V
V	F

a	b	a ET b
F	F	F
F	V	F
V	F	F
V	V	V

a	b	a OU b
F	F	F
F	V	V
V	F	V
V	V	V

a	b	a X-OU b
F	F	F
F	V	V
V	F	V
V	V	F

Il existe aussi les fonctions logiques NON-ET et NON-OU dont les valeurs sont les inverses de celles du ET et du OU.

Voici deux autres exemples de fonctions logiques. La première possède 3 variables x, y et z et le résultat est V ssi la majorité des variables ont pour valeur V. la seconde possède 2 variables qui codent un nombre binaire (x_1,x_0) ($x_i = V$ ssi le chiffre numéro i est à 1) et calcule (y_3, y_2, y_1, y_0) qui codent en binaire le carré du nombre en entrée.

x	y	z	majorité
F	F	F	F
F	F	V	F
F	V	F	F
F	V	V	V
V	F	F	F
V	F	V	V
V	V	F	V
V	V	V	V

x_1	x_0	y_3	y_2	y_1	y_0
F	F	F	F	F	F
F	V	F	F	F	V
V	F	F	V	F	F
V	V	V	F	F	V

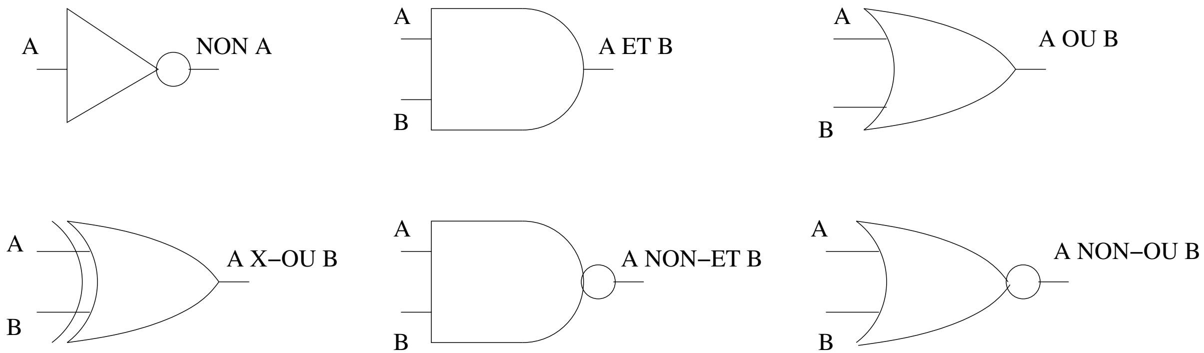
4.2 Portes logiques

Comme nous l'avons vu plus haut, un circuit électronique d'ordinateur est constitué d'un certain nombre d'entrées et de sorties binaires pouvant être à l'état 0 ou 1. Or, n'importe lequel de ces composants logiques peut être construit grâce à un assemblage de circuits logiques à 2 entrées (binaires) et une sortie (binaire) qu'on appelle *portes logiques*.

Les portes logiques existantes sont celles indiquées ci-dessous. Elles correspondent aux fonctions logiques de base en posant l'équivalence entre F et 0 et entre V et 1.

A elles seules, ces portes permettent de construire n'importe quel circuit logique quel que soit son nombre d'entrées, son nombre de sorties et la fonction logique qu'il calcule. (En fait, il est même possible de n'utiliser que des portes NON-ET ou que des portes NON-OU.)

Pour déterminer comment assembler un circuit logique à partir d'une table de vérité et des portes logiques, nous allons utiliser l'algèbre de Boole.



4.3 Algèbre de Boole

Une *fonction booléenne* est une fonction qui associe à des variables booléennes (égales à 0 ou 1) une valeur booléenne (égale à 0 ou 1). Cette algèbre définit l'addition (+) qui équivaut à un OU logique et la multiplication (.) qui équivaut au ET logique (quand on remplace F par 0 et V par 1). Ex : $0+0=0$, $0+1=1$, $1+1=1$, $0.0=0$, $0.1=0$, $1.1=1$. On a de plus l'opérateur de négation (-) qui est tel que $\overline{0}=1$ et $\overline{1}=0$.

Un certain nombre de lois permettent de transformer une formule en une autre formule équivalente :

- lois d'idempotence et d'inversion : $A.A=A$ et $A+A=A$, $A.\overline{A}=0$ et $A+\overline{A}=1$.
- La multiplication est distributive par rapport à l'addition : $A.(B+C)=A.B+A.C$.
- L'addition est distributive par rapport à la multiplication : $A+B.C=(A+B).(A+C)$.
- Lois d'absorption : $A.(A+B)=A$ et $A+A.B=A$.
- Lois de Morgan : $\overline{A.B}=\overline{A}+\overline{B}$ et $\overline{A+B}=\overline{A}\ .\ \overline{B}$.

Voici comment on peut réaliser une fonction booléenne f à partir de sa table de vérité. On construit l'expression de f en trois étapes.

- A chaque combinaison de valeurs qui rend la fonction égale à 1, on associe le produit des variables en ajoutant la barre horizontale de la négation sur les variables dont la valeur est nulle. Ex : pour une fonction à 3 variables A, B et C, si la combinaison $A=0$, $B=1$, $C=0$ donne le résultat 1 alors on y associe le produit $\overline{A}B\overline{C}$.
- Pour obtenir une expression complète de f, on fait la somme des produits déterminés précédemment. Ex : pour la fonction majorité, on obtient $f(A,B,C)=\overline{A}BC+A\overline{B}C+AB\overline{C}+ABC$.
- Quand c'est possible, on peut simplifier la formule en appliquant les lois de l'algèbre de Boole. Ex : comme $AB\overline{C}+ABC=AB(C+\overline{C})=AB$, on peut simplifier la formule de la fonction majorité par l'expression $f(A,B,C)=\overline{A}BC+A\overline{B}C+AB$.

  Si on s'autorise l'utilisation de l'opérateur $\oplus$ (équivalent du X-OU), on peut encore simplifier certaines formules, sachant que $A\oplus B=\overline{A}B+A\overline{B}$. La formule de la majorité devient $f(A,B,C)=(\overline{A}B+A\overline{B})C+AB=(A\oplus B)C+AB$.

4.4 Tableaux de Karnaugh

La méthode des tableaux de Karnaugh permet de déterminer la simplification optimale d'une somme de produits de variables booléennes. Cette méthode fonctionne pour des formules à 3 ou 4 variables différentes [1]. Elle consiste à représenter une table de vérité à deux dimensions, dont chacune représente les 4 états possibles d'un couple de variables (ou éventuellement les 2 états d'une variable pour les lignes quand la formule contient 3 variables). En ligne et en colonne, on place les 4 (ou les 2) états possibles de manière à ce que 2 états successifs ne diffèrent que par une seule valeur (ex : 00 puis 01, 11 et 10). Ensuite, on cherche à recouvrir

1. Elle est facilement extensible jusqu'à 6 variables mais nous ne traiterons pas ces cas.

les 1 de la grille par un ensemble de rectangles qui contiennent chacun un nombre de 1 égal à une puissance de 2 (1, 2, 4, 8 ou 16). Les lignes et les colonnes du tableau sont circulaires, c'est-à-dire, la case la plus à droite est voisine de la case la plus à gauche et la case la plus basse est voisine de la case la plus haute. Il faut chercher une couverture qui minimise le nombre de rectangles superposés et donc essayer d'abord de placer les plus gros rectangles possibles puis compléter avec des rectangles de plus en plus petits. Chaque rectangle correspond à l'un des produits de variables de la somme. Chaque rectangle regroupe un ensemble de combinaisons de valeurs pour un certain nombre de variable. Pour chacune de ces variables :

- si elle est toujours à 1 dans chacune des combinaisons du rectangle alors elle apparaît tel quel dans le produit correspondant.
- si elle est toujours à 0 dans chacune des combinaisons du rectangle alors c'est sa négation qui apparaît dans le produit.
- si elle est des fois à 0 et des fois à 1 alors elle n'apparaît pas dans le produit.

Les rectangles de taille 2^k d'un tableau de n variables représentent un produit de $n - k$ variables.

x	y	z	s
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

a	b	c	d	f
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

a \ bc	11	10	00	01
1	1	0	1	1
0	0	1	1	1

a.c $\overline{a}.\overline{c}$ $\overline{b}$

ab \ cd	11	10	00	01
11	1	1	1	1
10	0	0	1	0
00	1	0	0	1
01	1	1	1	1

$\overline{a}.d$ $a.\overline{c}.\overline{d}$ b

FIG. 1 – $s = \overline{b} + a.c + \overline{a}.\overline{c}$ *et* $f = b + \overline{a}.d + a.\overline{c}\overline{d}$

4.5 Réalisation des circuits à partir des fonctions booléennes

Une fois qu'on a l'expression de la fonction booléenne sous la forme d'une somme de produits, il est facile de réaliser le circuit logique correspondant. A chaque variable correspond une entrée du circuit. A toute les

variables qui apparaissent négativement on ajoute le circuit de la négation à l'entrée correspondante. On rajoute les circuits ET correspondant aux produits de l'expression booléenne. Enfin, on relie les sorties des circuits à une portes OU possédant autant d'entrées qu'il y a de produits. Le circuit de la fonction majorité est donné par le schéma de la figure 2.

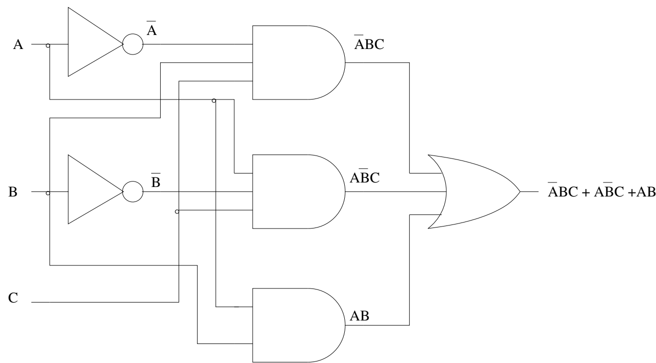


FIG. 2 – *circuit correspondant à la formule* $f(A,B,C) = \overline{A}BC + A\overline{B}C + AB$.

Si on dispose de portes X-OU, on peut simplifier le circuit comme on le voit figure 3.

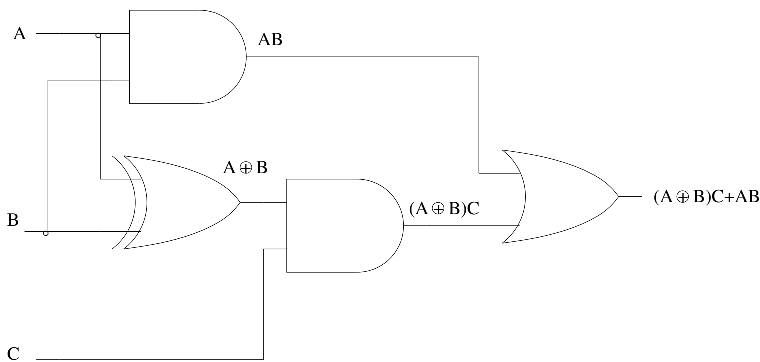


FIG. 3 – *circuit correspondant à la formule* $f(A,B,C) = (A \oplus B)C + AB$.

4.6 Les circuits logiques de base

Nous allons présenter un certain nombre de circuits logiques très utilisés dans la conception d'une machine.

4.6.1 Le décodeur

Un décodeur est un circuit logique a n entrées et 2^n sorties. L'ensemble des n entrées correspond à un nombre i (binaire de n bits). Toutes les sorties sont à 0 sauf la i^{eme} qui est à 1. Chaque sortie d'un décodeur est potentiellement relié à un autre circuit. Un décodeur sert donc à sélectionner un circuit à déclencher à partir de son numéro.

4.6.2 Le multiplexeur

Un multiplexeur est un circuit logique a $2^n + n$ entrées et une sortie. Les entrées sont réparties en 2 groupes : le groupe de sélection qui est un groupe de n entrées qui codent un nombre binaire i et un groupe

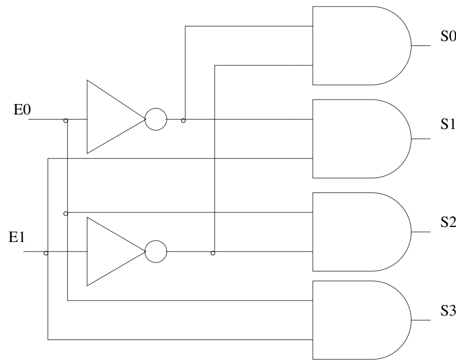


FIG. 4 – *Décodeur à 2 entrées et 4 sorties.*

d'information composé de 2^n entrées dont chacune code une information sur 1 bit. En sortie, on aura la valeur du i^{eme} bit du groupe d'information (voir figure 5).

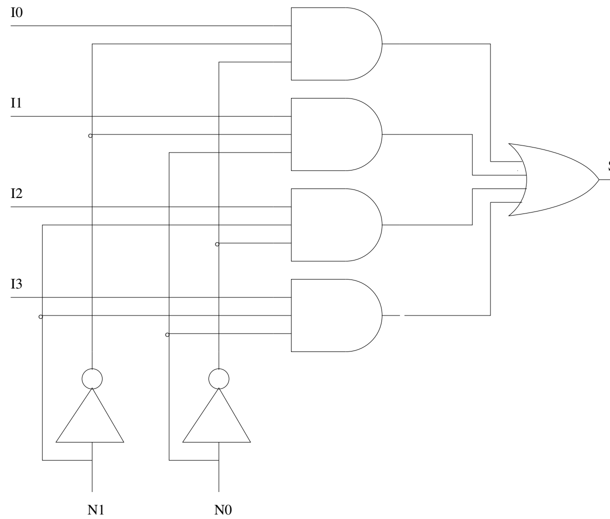


FIG. 5 – *Multiplexeur à* $4 + 2$ *entrées.*

4.6.3 L'additionneur

Un additionneur est un circuit à $2n$ entrées représentant deux entiers de n bits et $n+1$ sorties représentants la somme des deux entiers plus la retenue.

Nous allons d'abord présenter le demi-additionneur 1 bit. Il consiste à additionner 2 bits A et B et à récupérer le bit de somme S et le bit de retenue R. On détermine S et R en faisant la table de vérité de l'addition et on obtient les formules : $S = A \oplus B$ et $R = AB$. On l'appelle demi-additionneur car il n'est pas assez complet pour être utilisé pour faire des additions à plusieurs bits. En effet, lorsqu'on additionne deux entiers bit à bit, il faut tenir compte de la retenue de l'addition 1 bit précédente. Un additionneur 1 bit complet possède donc une entrée supplémentaire R_e. En tenant compte de la retenue R_e, on obtient les formules : $S = A \oplus B \oplus R_e$ et $R = AB + (A \oplus B)R_e$.

Pour faire des additions sur n bits, il suffit d'assembler n additionneurs 1 bit. Pour le i^{eme} additionneur, on place aux entrées A et B les i^{eme} bit des entiers à additionner et on relie la sortie R du $(i-1)^{eme}$

additionneur à l'entrée R_e. L'entrée R_e du premier additionneur qui reçoit la valeur 0.

4.6.4 l'unité arithmétique et logique

Grâce aux circuits que nous avons définis ci-dessus, nous allons pouvoir construire une UAL n bits simplifiée. L'UAL prend en entrée 2 entiers codés sur n bits et un entier codé sur m bits qui correspond au numéro de l'opération à appliquer sur les deux entiers.

Nous allons définir une UAL effectuant 4 opérations (OU, ET, NON et l'addition). Pour cela nous allons d'abord créer une UAL 1 bit puis en assembler 8 pour faire une UAL opérant sur des octets (voir figure 7).

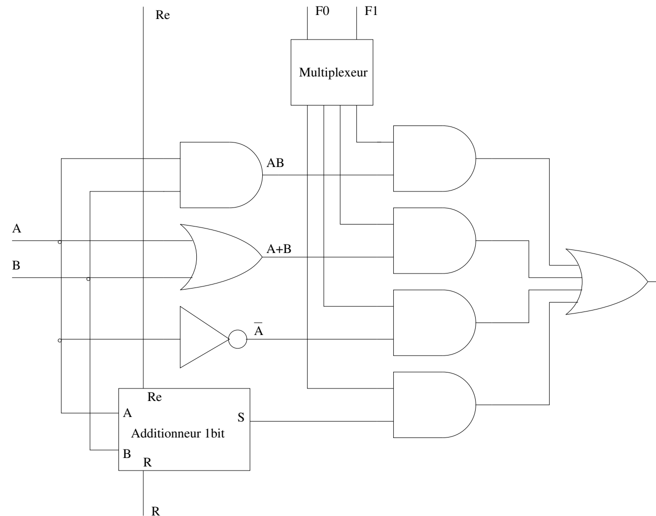


FIG. 6 – *UAL 1 bit réalisant le ET, le OU, le NON et l'addition.*

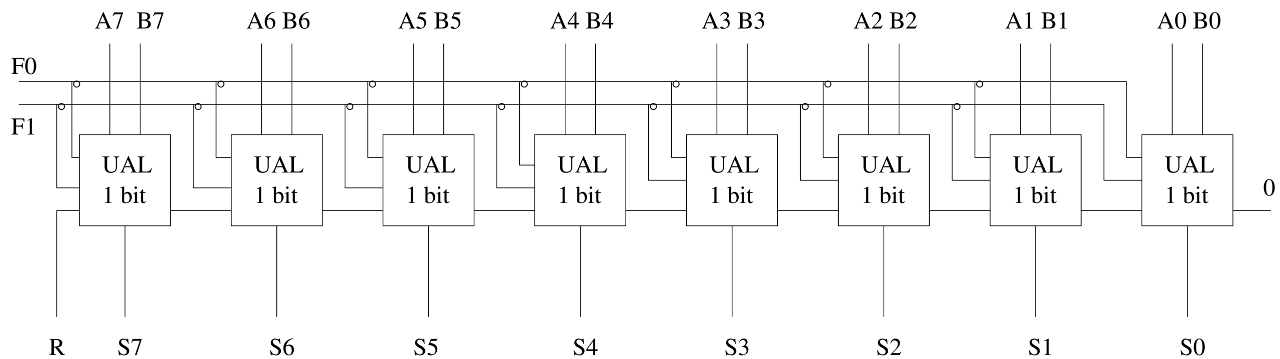


FIG. 7 – *UAL 8 bits résultant d'un assemblage de 8 UAL 1 bit.*

4.7 Réalisation des mémoires

Jusqu'à présent nous avons montré des circuits dont les états internes sont uniquement fonctions des valeurs actuelles de leurs entrées. Les circuits mémoires possèdent des états internes qui dépendent aussi des anciennes valeurs qu'ils avaient en entrée.

4.7.1 Bascule RS

Une bascule RS (RS pour Reset/Set) est le circuit de base permettant de constituer une mémoire d'un bit. Il est composé de deux portes NON-OU relié comme indiqué en figure 8. Il possède deux entrées R et S et deux sorties Q et $\overline{Q}$.

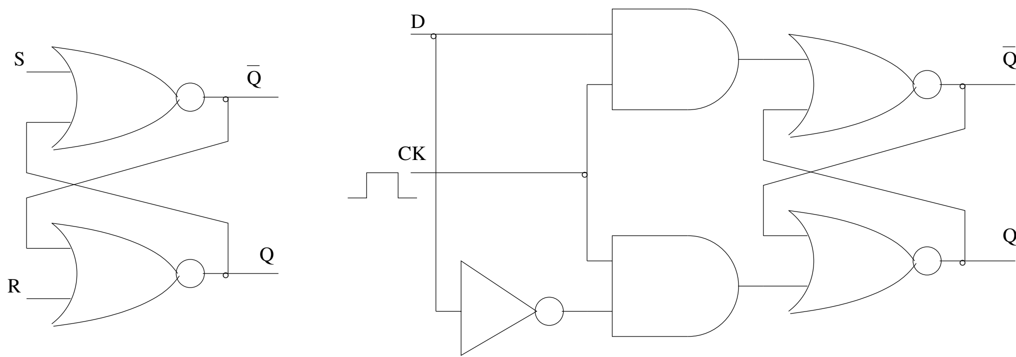


FIG. 8 – *Bascule RS et bascule D.*

La propriété fondamentale de ce circuit est que lorsque $R = S = 0$, les valeurs de Q et de $\overline{Q}$ peuvent être égales à 0 ou à 1 : si $Q = 0$ alors $\overline{Q} = 1$ et si $Q = 1$ alors $\overline{Q} = 0$.

Si on a $R = S = 0$ et que R devient égal à 1 alors Q devient (ou reste à) 0 et $\overline{Q}$ devient (ou reste) à 1. Si on a $R = S = 0$ et que S devient égal à 1 alors Q devient (ou reste à) 1 et $\overline{Q}$ devient (ou reste) à 0. Si l'entrée qui était passée à 1 revient à l'état 0 (de sorte qu'on ait à nouveau $R = S = 0$), l'état des sorties ne change pas. Il y a un cas problématique : celui où on passe directement de $R = S = 1$ (qui fait que $Q = \overline{Q} = 0$) à $R = S = 0$. En effet, on ne peut déterminer à l'avance quels vont être les valeurs de Q et $\overline{Q}$. Il faut donc que nous complétions ce circuit afin qu'il ne puisse y avoir $R = S = 1$ en entrée. C'est ce que va réaliser une bascule D.

4.7.2 Bascule D

On construit une bascule D à partir d'une bascule RS en insérant deux portes ET avant les entrées R et S. Une entrée de porte ET est nommée D. Elle est reliée à une autre entrée de l'autre porte ET en passant par une porte NON. L'entrée restante de chaque porte ET est reliée à une entrée CK. De cette façon, si $CK = 0$ alors quelle que soit la valeur de D on a $R = S = 0$ et donc les états Q et $\overline{Q}$ gardent leurs valeurs. Par contre, si $CK = 1$ alors Q prend (ou garde) la valeur de D. On peut donc dire que cette bascule mémorise l'ancienne valeur de D tant que $CK = 0$. Dès que CK passe à 1, Q prend la valeur de D.

Habituellement, CK est la sortie d'un dispositif spécial appelé *horloge interne*. Une horloge émet à intervalle régulier des tensions correspondant à l'état 1 pendant un court moment avant de revenir à une tension correspondant à l'état 0. Toutes les entrées CK sont reliées à l'horloge interne du μP. A chaque cycle d'horloge, l'état global du système constitué par l'ensemble des circuits change : chaque entrée D de bascule est recopiée sur sa sortie Q.

4.7.3 Registre

Une bascule D permet de mémoriser 1 bit. En regroupant 8, 16 ou 32 bascules D, on peut obtenir un registre 8, 16 ou 32 bits permettant de mémoriser des valeurs de 0 à 255, 65 535 ou 4 294 967 295.

4.7.4 Mémoire principale

La mémoire principale doit permettre de stocker des octets référençables grâce à leur adresse. Ces octets peuvent être écrits ou lus. Par ailleurs, cette mémoire doit être extensible donc il faut constituer la mémoire

par blocs. On aura donc les entrées suivantes : n entrées I_0, ...I_{n-1} pour stocker des mots de n bits (en général, on stocke des octets donc $n = 8$), m entrées A_0, ...A_{m-1} pour référencer un des 2^m mots, une entrée CS (Chip Select) qui est à 1 si ce bloc mémoire est sélectionné et une entrée RD (Read) qui est à 0 si on effectue une opération d'écriture d'un mot et à 1 si on effectue une opération de lecture. On aura aussi n sorties O_0, ...O_{n-1} pour récupérer un mot lors d'une lecture.

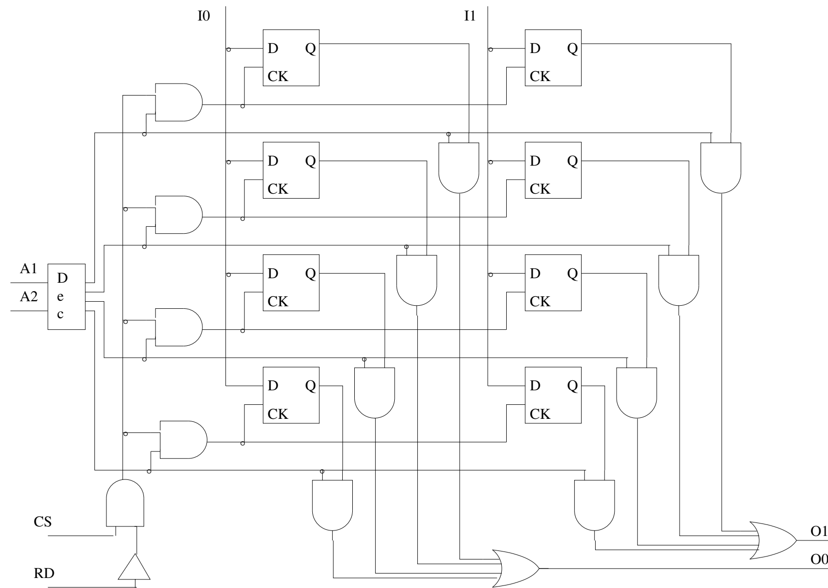


FIG. 9 – *Bloc mémoire stockant 4 mots de 2 bits. En rajoutant des colonnes, on augmente le nombre de bits des mots. En rajoutant des lignes, on augmente le nombre de mots mémorisés.*

4.7.5 Sélection d'un octet à partir de son adresse

Voici comment accéder à un octet situé dans un certain bloc mémoire à partir d'une adresse complète. Considérons qu'on a une mémoire principale de 64Ko (2^{16} octets) répartis dans 128 blocs de 512 octets. Chaque bloc a 9 entrées-adresse A_0, ...A_8. Nous avons besoin d'un registre de 16 bits pour coder une adresse. Les 7 premiers bits seront reliés à un multiplexeur dont chacune des 128 sorties sera reliée à l'entrée CS d'un des 128 blocs. Les 9 derniers bits serons reliés directement aux entrées A_0, ...A_8 de chacun des 128 blocs.

4.8 Succession d'états d'un ordinateur

Un ordinateur en train de fonctionner peut être vu comme passant cycliquement d'un état global de ses circuits à un autre. L'état suivant peut dépendre d'entrées externes fournies par les périphériques d'entrée mais aussi surtout des états internes mémorisés dans les registres et la mémoire principale.

Avant de voir comment se déroule schématiquement une instruction du point de vue électronique, nous allons voir que nous pouvons construire des machines dont l'état global suivant dépend uniquement de l'état global précédent.

Considérons un circuit avec n entrées e_1, ...e_n et m sorties s_1, ...s_m. Soient les vecteurs E =(e_1, ...e_n) et S =(s_1, ...s_m). Si le circuit est sans mémoire, il réalise une fonction logique F telle que $S = F(E)$. Si le circuit à une mémoire et qu'il est commandé par une horloge alors la sortie dépend aussi de l'état antérieur du circuit qui dépendait lui-même de l'ancienne valeur de S. Les sorties du circuits peuvent donc s'exprimer par $S_t = F(E, S_{t-1})$. Si maintenant on fait en sorte de raccorder chaque entrée du circuit à une de ses propres

sorties alors on obtient l'expression $S_t = F(S_{t-1})$. Une fois qu'on a initialisé le circuit en fixant S_0, le circuit va changer d'état à chaque cycle d'horloge sans qu'on ait besoin d'entrer d'information de l'extérieur.

Voici un exemple de circuit simple : un compteur à trois bits (voir figure 10).

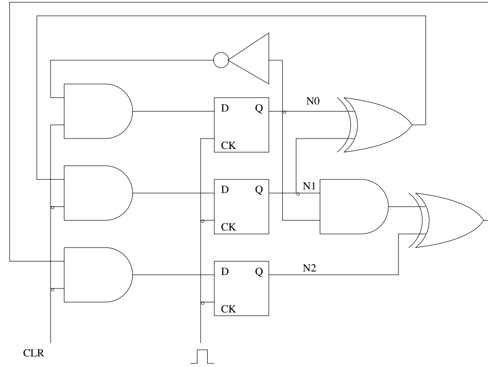


FIG. 10 – *Un compteur à 3 bits.*

Au départ, l'entrée CLR est à 0 et l'horloge démarre. Les entrées et les sorties des bascules D seront donc à zéro dès le premier cycle d'horloge et le resteront tant que CLR sera à 0. Dès qu'on passe CLR à 1, le compteur démarre. Cycle après cycle, l'état global (N_2, N_1, N_0) va passer de l'état (0, 0, 0) à l'état (0, 0, 1), puis (0, 1, 0), etc jusqu'à (1, 1, 1) puis revenir à (0, 0, 0) et recommencer.

4.9 Exécution d'une instruction

L'exécution d'une instruction sert à modifier l'état global des circuits. Pour faire un ordinateur, on peut partir d'un compteur n bits similaire à celui qu'on a défini dans la section précédente et s'en servir de compteur ordinal (CO) dont la valeur est incrémentée à chaque cycle d'horloge. Pendant le cycle, on utilise la valeur de CO comme adresse de la prochaine instruction à exécuter. Il suffit que les fils du registre CO soient reliés aux fils des blocs-mémoire pour qu'on puisse accéder à la valeur (qui code l'instruction) pointée par CO. Cette valeur est alors recopiée dans le registre d'instruction. Suivant cette valeur, on effectue soit une opération arithmétique ou logique en utilisant l'UAL, soit un transfert de mot d'un registre ou d'une case-mémoire vers un autre registre ou une autre case-mémoire.

5 L'ordinateur en tant que machine numérique

Nous allons maintenant parler du langage machine et expliquer comment un programme écrit en C peut avoir une équivalence en langage machine. Pour ceci, nous définissons un microprocesseur simplifié (inspiré du MC68000 de Motorola) que nous appellerons MP0. Il saura exécuter des programmes dans un langage appelé LM0. MP0 et LM0 n'existent pas. Ils sont définis uniquement dans le cadre de ce cours. Ils servent à illustrer à quoi ressemble un langage machine sachant que chaque modèle de microprocesseur possède un langage machine qui lui est particulier.

5.1 Les registres de MP0

MP0 ne possèdent que des registres 8 bits (c'est bien trop peu pour un vrai microprocesseur). Il ne peut donc que manipuler des octets et accéder à des adresses de 0 à 255.

Comme tous les microprocesseurs, MP0 possèdent un compteur ordinal CO, un registre d'état qui contient les bits Z et N. Quand le bit Z est à 1, cela signifie que la dernière instruction a produit un résultat égal à zéro. Quand le bit N est à 1, cela signifie que la dernière instruction a produit un résultat négatif. MP0 possède 2 registres de données D0 et D1 qui stockent chacun un octet à traiter. MP0 possède aussi 2 registres d'adresse A0 et A1 qui stockent chacun une adresse et qui servent à accéder à des octets stockés en mémoire principale. Enfin, MPO possède un pointeur de pile PP, qui est exactement comme un registre d'adresse mais dont le contenu est réservé à l'adresse de la pile.

5.2 Formats numérique et mnémonique des instructions de MP0

Chaque instruction de LM0 est codée sur 2 octets. Le premier octet code le type de l'opération et le type de son ou ses opérandes. Le second octet peut être utilisé pour coder une des opérandes. Si l'instruction n'a pas d'opérande ou qu'aucune de ses opérandes n'a besoin d'un octet pour être spécifiée alors cet octet sera 0.

Comme il n'est pas pratique de désigner une instruction du langage machine par son numéro, on lui attribuera un nom et une syntaxe. On écrira chaque instruction sous forme de *mnémonique* c'est-à-dire d'un nom qui rappelle l'opération qu'elle effectue. Ex : MOVE (transfert), ADD (addition), SUB (soustraction), JMP (saut), etc. On ajoutera l'opérande après la mnémonique (ex : JMP 120) et s'il y en a deux, on les séparera avec une virgule (ex : MOVE 100, D0).

5.3 Adressage

Les octets que manipulent les langages machines sont stockés soit dans des registres de donnée ou d'adresse, soit dans la mémoire principale. Un registre d'adresse (A0 ou A1 en LM0) sert à accéder indirectement au contenu de l'adresse qu'il contient (comme un pointeur en C). On peut aussi accéder à une adresse grâce à son numéro. Les instructions de LM0 ne permettent de désigner qu'une seule opérande au maximum par un numéro, l'autre opérande devant être un registre.

5.3.1 Adressage immédiat

Si on connait la valeur que l'on veut traiter, on peut la spécifier explicitement. En écrivant l'instruction, on préfixera la valeur avec le caractère '#'. Ex : `MOVE #3,D0` place la valeur 3 dans le registre D0.

5.3.2 Adressage direct

L'adressage direct consiste à accéder à une valeur située à une adresse dont on connait le numéro. En écrivant l'instruction, on spécifiera simplement cette opérande par le numéro de son adresse. Ex : `MOVE 100,D0` place la valeur située à l'adresse 100 dans le registre D0.

5.3.3 Adressage indirect

L'adressage indirect consiste à accéder à une valeur située à une adresse mémorisée dans un registre d'adresse. En écrivant l'instruction, on spécifiera cette indirection en mettant le registre d'adresse entre parenthèses. Ex : `MOVE (A0),D0` place la valeur située à l'adresse contenue dans le registre d'adresse A0 dans le registre D0.

5.4 Les instructions de LM0

On peut regrouper l'ensemble des instructions en 3 groupes : les instructions de transfert, les opérations arithmétiques et logiques et les instructions de saut.

5.4.1 Instructions de transfert

On utilise la mnémonique `MOVE` pour désigner les instructions de transfert d'un octet d'un emplacement (registre ou case mémoire) à un autre. Le format de l'instruction est `MOVE source,destination`. Ex : `MOVE D0,123`, `MOVE (A0),(A1)`, `MOVE #10,A1`. Si la valeur transférée est nulle, le bit Z du registre d'état passe à 1. Si la valeur transférée est négative, le bit N du registre d'état passe à 1. (Les instructions `MOVE` sont à rapprocher des instructions d'affectation des langages évolués.)

Les instructions `PSH` et `POP` respectivement empilent et dépilent un octet grâce au registre PP. L'instruction `PSH` décrémente PP et place la valeur de l'opérande à l'adresse pointée par PP. L'instruction `POP` place la valeur pointée par PP dans l'opérande et incrémente PP. Au démarrage de la machine, le registre PP contient l'adresse 255.

5.4.2 Opérations arithmétiques et logiques

Les opérations arithmétiques et logiques modifient la valeur d'une opérande ou calculent une valeur à partir de deux opérandes. En LM0, le résultat du calcul remplace la valeur de la deuxième opérande. Ex : `ADD D0,D1` fait la somme des valeurs de `D0` et `D1` et place le résultat dans `D1`. Le résultat d'une opération affecte les bits Z et N du registre d'état comme pour les instructions `MOVE`.

Les opérations existantes en LM0 sont : ADD (l'addition), SUB (la soustraction), MUL (la multiplication), DIV (la division), les opérations logiques AND, OR et NOT. De plus, l'opérateur CMP de comparaison fait la même chose que SUB mais ne modifie pas la deuxième opérande. Par contre, elle modifie les bits Z et N du registre d'état. Cette instruction s'utilise en préparation d'une instruction de saut conditionnel.

5.4.3 Instructions de saut

Les instructions de sauts permettent de modifier la valeur de CO afin d'exécuter une autre série d'instructions que celle qui suivent l'instruction courante. L'opérande de l'instruction de saut est une adresse. Ex : `JMP 136` fait que la prochaine instruction à exécuter sera celle située à l'adresse 136.

Les saut peuvent être conditionnels : ils ne sont effectués que si des conditions sur les bits Z et N du registre d'état sont réalisés. Ces sauts conditionnelles sont `JEQ`: saut si $Z = 1$, `JNE`: saut si $Z = 0$, `JGT`: saut si $Z = 0$ et $N = 0$, `JGE`: saut si $N = 0$, `JLT`: saut si $N = 1$ et `JLE`: saut si $Z = 1$ ou $N = 1$.

Enfin, il existe le saut (inconditionnel) à un sous-programme `JSR` qui empile l'adresse de l'instruction suivante (celle contenue dans CO) avant de sauter à l'adresse désignée par l'opérande. L'instruction `RTS` de retour d'un sous-programme est complémentaire. Elle dépile une adresse (normalement celle qui a été empilée lors qu'une instruction `JSR`) puis saute à cette adresse.

5.5 Equivalence d'un programme en LM0 et en C

Nous présentons maintenant comment on peut faire en LM0 les mêmes choses que l'on fait en C.

5.5.1 L'affectation de variable

Le nom d'une variable en C correspond à une adresse. Affecter une valeur à une variable correspond à faire un transfert de valeur grâce à une (ou plusieurs) instruction(s) `MOVE`. Ex : quand en C, on a l'instruction `x = 1;`, en LM0, on fait l'équivalent grâce aux deux instructions `MOVE #1,D0` puis `MOVE D0, 110` (en admettant que l'adresse de x est 110).

5.5.2 L'instruction conditionnelle `if`

L'instruction `if(...) {...} else {...}` se simule en LM0 ainsi : tester la condition du `if` et faire un saut conditionnel qui exécutera les instructions du `else` si la condition est fausse, placer à la fin des instructions du `if` un saut JMP sur l'adresse qui suit les instructions du `else`.

Exemple : (on utilise le registre D0 pour mémoriser x)

En C	En LM0
if($x > 3$)	100 : CMP #3,D0
$x = x - 5$;	102 : JLE 108
else	104 : SUB #5,D0
$x = x + 10$;	106 : JMP 110
[...]	108 : ADD #10,D0
	110 : [...]

5.5.3 La boucle `while`

L'instruction `while(...){...}` se simule en LM0 ainsi : tester la condition du while et faire un saut conditionnel sur les instructions situées après le corps du `while` si la condition est fausse, placer à la fin des instructions du corps du `while` un saut inconditionnel sur le début des instructions du test du `while`.

Exemple :

En C	En LM0
while($x > 10$)	104 : CMP #10,D0
$x = x - 1$;	106 : JLE 112
[...]	108 : SUB #1,D0
	110 : JMP 104
	112 : [...]

5.5.4 Les fonctions

Une fonction C se simule en LM0 grâce à un appel à un sous-programme (se terminant par l'instruction `RTS`) par l'instruction `JSR`. Le passage de paramètre peut se simuler en réservant un registre pour chaque paramètre d'appel et un autre registre pour le résultat. Il faudra penser à empiler les valeurs des registres réservés avant leur affectation aux valeurs des paramètres si on veut pouvoir récupérer ces valeurs en les dépilant après l'exécution du sous-programme appelé.

Exemple :

En C	En LM0
int factorielle(int n)	204 : MOVE 100,D0
{	206 : JSR 250
if($n < 2$)	208 : MOVE D1, 101
return 1;	[...]
else	250 : CMP #2,D0
return n∗fact($n - 1$);	252 : JGE 258
}	254 : MOVE #1,D1
[...]	256 : RTS
y = factorielle(x);	258 : PSH D0
[...]	260 : SUB #1,D0
	262 : JSR 250
	264 : POP D0
	266 : MUL D0,D1
	268 : RTS