# Principes de Fonctionnement des Ordinateurs

*Version 2024*

**Olivier Bonaventure**

**janv. 22, 2024**

# Table des matières

CHAPITRE 1

# Introduction

Les ordinateurs sont au coeur d'un nombre grandissant de services dans notre société qui est de plus en plus numérique. Ce cours vise à vous apprendre les principes de base de fonctionnement des dispositifs numériques que vous utilisez tous les jours. Le syllabus est divisé en deux parties.

La première partie se focalise sur l'apprentissage de l'assembleur. L'assembleur ou langage d'assemblage est l'ensemble des instructions simples qui sont directement supportées par un microprocesseur. Plutôt que de considérer un microprocesseur réelle avec toute sa complexité, les syllabus utilise un [simulateur de microprocesseur](https://github.com/Schweigi/assembler-simulator) développé initialement par [Marco Schweighauser](https://github.com/Schweigi) et amélioré par [Nikita Tyunyayev](https://github.com/ntyunyayev). Cette première partie du syllabus vous permettre de comprendre les principes de base de la programmation d'un microprocesseur simple en assembleur.

La deuxième partie du livre abordera les aspects matériels en construisant pas à pas un microprocesseur extrêmement simple mais fonctionnel. Cette partie s'appuie sur l'excellent livre The Elements of Computing Systems écrit par Noam Nisan et Shimon Schocken et publié au MIT Press.

Ce syllabus n'est pas exhaustif, le livre de référence contient de nombreux détails qui ne sont pas abordés dans le syllabus. Par contre, la version web du syllabus est interactive, c'est-à-dire qu'à côté des concepts théoriques, vous y trouverez également de nombreux exercices qui sont supportés par inginious afin de permettre à chaque étudiant de vérifier sa compréhension de la théorie.

Nous aborderons différents aspects du fonctionnement des ordinateurs dans ce syllabus. Dans tout ordinateur, l'information est encodée sous la forme de bits. Chaque bit peut prendre deux valeurs distinctes : *0* et *1*. Leur intérêt principal est qu'il est possible de représenter n'importe quel type de données (nombre, caractères, texte, image, vidéo, son, ...) sous la forme d'une séquence de bits. Nous nous concentrerons sur la logique booléenne qui permet de manipuler ces bits. Les premiers ordinateurs ont été conçus pour effectuer des calculs numériques. Nous verrons ensuite comment représenter les nombres sous forme binaire et comment construire des circuits qui permettent de réaliser des additions, des soustractions et d'autres opérations sur les nombres entiers. Nous pourrons ensuite analyser comment un ordinateur peut mémoriser de l'information et utiliser l'information stockée en mémoire.

A ce stade, nous aurons appris les bases qui permettent de concevoir un microprocesseur simple qui pourra être programmé. Nous nous concentrerons ensuite sur l'architecture des ordinateurs et les interactions entre le microprocesseur, la mémoire et les entrées/sorties. Après avoir construit ce microprocesseur, vous pourrez voir comment il peut supporter un langage d'assemblage simple. Le cours se terminera par une explication du fonctionnement des dispositifs de stockage de données.

Une version imprimable de ce document est disponible via `/PFO.pdf`.

CHAPITRE 2

# Représentation de l'information

Le fonctionnement des ordinateurs s'appuie sur quelques principes très simples, mais qui sont utilisés à une très grande échelle. Le premier principe est que toute l'information peut s'encoder sous une forme binaire, c'est-à-dire une suite de bits. Un bit est l'unité de représentation de l'information. Un bit peut prendre deux valeurs :

— `0`
— `1`

On peut associer une signification à ces bits. Il est par exemple courant de considérer que le bit `0` représente la valeur *Faux* tandis que le bit `1` représente la valeur *Vrai*. C'est une convention qui est utile dans certains cas, mais n'est pas toujours nécessaire et peut parfois porter à confusion.

Dans un ordinateur, toutes les informations peuvent être stockées sous la forme d'une séquence de bits. La longueur de la séquence est fonction de la quantité d'information à stocker. Notre premier exemple concerne les caractères. Il est important de pouvoir représenter les différents caractères des langues écrites de façon compacte et non-ambiguë pour pouvoir stocker et manipuler du texte sur un ordinateur. Le principe est très simple. Il suffit de construire une table qui met en correspondance une séquence de bits et le caractère qu'elle représente.

## 2.1 Représentation des caractères

Pour représenter chaque caractère sous la forme d'une séquence de bits, il suffit de choisir une séquence unique qui représente un caractère. Commençons par essayer de représenter les dix chiffres de notre numérotation décimale, de `0` à `9`. Nous pouvons facilement associer une séquence binaire unique à chacun de ces chiffres. Avec deux bits, nous pouvons construire quatre séquences différentes : `00`, `01`, `10` et `11`. Avec ces deux bits, nous ne pouvons pas obtenir une séquence unique pour chaque chiffre décimale. Avec trois bits, nous pouvons construire 8 séquences différentes : `000`, `001`, `010`, `011`, `100`, `101`, `110` et `111`. Pour représenter tous les chiffres décimaux, nous avons besoin d'utiliser des séquences d'au moins 4 bits. La table ci-dessous présente une première représentation possible des chiffres décimaux.

TABLEAU 2.1 – Représentation possibles de chiffres décimaux

| Chiffre | Séquence binaire |
|---|---|
| 0 | `0000` |
| 1 | `0001` |
| 2 | `0010` |
| 3 | `0011` |
| 4 | `0100` |
| 5 | `0101` |
| 6 | `0110` |
| 7 | `0111` |
| 8 | `1000` |
| 9 | `1001` |

En utilisant cette représentation, on peut représenter n'importe quel nombre naturel comme une séquence de bits. Il suffit pour cela de représenter chaque chiffre par un bloc de quatre bits. Ainsi, le nombre *478* en notation décimale pourra être représenté par la séquence de bits `0100 0111 1000`. Dans la littérature, cette représentation est dénommé DCB, pour Décimal Codé en Binaire ou BCD pour Binary Coded Decimal en anglais.

Pour représenter les lettres de l'alphabet en plus des chiffres, il nous faut utiliser plus de bits. On peut facilement voir qu'avec $n$ bits on peut construire $2^n$ séquences distinctes. Avec 4 bits, on peut donc obtenir 16 séquences distinctes. Il faut 5 bits pour avoir 32 séquences distinctes, 6 bits pour en construire 64, ... Notre alphabet latin comprend 26 lettres. Si on veut pouvoir représenter les lettres majuscules et les chiffres sous forme binaire, nous utiliser au minimum 6 bits. Avec ces six bits, on peut représenter les 26 lettres majuscules, les 26 lettres minuscules et les 10 chiffres. Il ne nous reste ensuite plus que 2 séquences de bits pour représenter tous les autres caractères comme la ponctuation, les symboles mathématiques, ...

Parmi les tables d'encodage des caractères les plus simples, la plus connue est certainement la table US-ASCII dont la définition est notamment reprise dans **RFC 20**. Cette table associe une séquence de 7 bits (*b7* à *b1*) à un caractère particulier. Pour des raisons historiques, certains de ces caractères sont des caractères dits « de contrôle » qui ne sont pas imprimables. Ils permettaient de contrôler le fonctionnement de terminaux ou d'imprimantes. Par exemple, les caractères *CR* et/ou *LF* correspondent au retour de charriot et au passage à la ligne sur un écran ou une imprimante.

Code source 2.1 – Table des caractères ASCII

```
|-------------------------------------------------------------------|
  B  \ b7 ------------>| 0   | 0   | 0   | 0   | 1   | 1   | 1   | 1   |
   I  \  b6 ---------->| 0   | 0   | 1   | 1   | 0   | 0   | 1   | 1   |
    T  \   b5 -------->| 0   | 1   | 0   | 1   | 0   | 1   | 0   | 1   |
     S                 |-----------------------------------------------|
               COLUMN->| 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   |
|b4 |b3 |b2 |b1 | ROW  |     |     |     |     |     |     |     |     |
+---------------------+-----------------------------------------------+
| 0 | 0 | 0 | 0 | 0    | NUL | DLE | SP  | 0   | @   | P   |  `  |  p  |
|---|---|---|---|------|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 0 | 0 | 1 | 1    | SOH | DC1 | !   | 1   | A   | Q   |  a  |  q  |
|---|---|---|---|------|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 0 | 1 | 0 | 2    | STX | DC2 | "   | 2   | B   | R   |  b  |  r  |
|---|---|---|---|------|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 0 | 1 | 1 | 3    | ETX | DC3 | #   | 3   | C   | S   |  c  |  s  |
|---|---|---|---|------|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 1 | 0 | 0 | 4    | EOT | DC4 | $   | 4   | D   | T   |  d  |  t  |
|---|---|---|---|------|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 1 | 0 | 1 | 5    | ENQ | NAK | %   | 5   | E   | U   |  e  |  u  |
|---|---|---|---|------|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 1 | 1 | 0 | 6    | ACK | SYN | &   | 6   | F   | V   |  f  |  v  |
```

(suite sur la page suivante)

(suite de la page précédente)

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 7 | BEL | ETB | ' | 7 | G | W | g | w |
| 1 | 0 | 0 | 0 | 8 | BS | CAN | ( | 8 | H | X | h | x |
| 1 | 0 | 0 | 1 | 9 | HT | EM | ) | 9 | I | Y | i | y |
| 1 | 0 | 1 | 0 | 10 | LF | SUB | * | : | J | Z | j | z |
| 1 | 0 | 1 | 1 | 11 | VT | ESC | + | ; | K | [ | k | { |
| 1 | 1 | 0 | 0 | 12 | FF | FS | , | < | L | \ | l | \| |
| 1 | 1 | 0 | 1 | 13 | CR | GS | - | = | M | ] | m | } |
| 1 | 1 | 1 | 0 | 14 | SO | RS | . | > | N | ^ | n | ~ |
| 1 | 1 | 1 | 1 | 15 | SI | US | / | ? | O | _ | o | DEL |

La table US-ASCII (Code source 2.1) définit les représentations binaires suivantes :

— *0110000* correspond au caractère représentant le chiffre *0*
— *0111001* correspond au caractère représentant le chiffre *9*
— *1000001* correspond au caractère représentant la lettre *A* (majuscule)
— *0100000* correspond au caractère représentant un espace

Chaque caractère est représenté sous la forme d'une séquence de 7 bits.

Cette table avait l'inconvénient majeur de ne contenir que les représentations des caractères non-accentués de l'alphabet latin. Elle permet d'écrire du texte en anglais et dans d'autres langues européennes qui utilisent peu d'accents, mais ne permet évidemment pas de représenter tous les caractères des langues écrites sur notre planète. Au fil des années, ce problème a été résolu avec d'autres tables de correspondance dont celles qui sont adaptées aux accents utilisés par les langues européennes. Aujourd'hui, l'encodage standard des caractères se fait en utilisant le format Unicode. Une description détaillée d'Unicode sort du cadre de ce cours d'introduction, mais sachez qu'en mars 2020, la version 13.0 d'Unicode permettait de représenter 143859 caractères différents correspondant à 154 formes d'écritures. Unicode permet de représenter quasiment toutes les langues écrites connues sur notre planète. Des chercheurs ont même proposé un format Unicode permettant de supporter le Klingon, c'est-à-dire la langue écrite inventée pour la série de films Star Trek.

Avoir une représentation binaire pour les caractères permet de les stocker en mémoire, sur disque ou de les transmettre à travers un réseau. C'est important, mais il faut aussi pouvoir permettre à un humain de lire des textes produits par un ordinateur, que ce soit sur papier ou écran. Il existe de très nombreuses solutions qui permettent d'afficher ou d'imprimer des caractères. Dans ce cours d'introduction, nous nous contentons d'une solution très simple qui fonctionne en noir et blanc. Nous pourrons ajouter les couleurs lorsque nous aurons vu comment représenter des nombres dans le chapitre suivant.

Un écran et une imprimante permettent d'afficher des points à n'importe quelle position. On peut aisément se représenter un écran comme un rectangle composé de pixels. Chacun des points de cet écran est identifié par une abscisse et une ordonnée qui sont toutes les deux entières. Ainsi, un écran 1024x768 peut afficher 1024 points selon l'axe des x et 768 points selon l'axe des y.

Sur un tel écran, on peut facilement afficher des caractères. Il suffit d'avoir pour chaque caractère une table qui contient la représentation graphique de chacun des caractère à afficher sous la forme de pixels. A titre d'exemple, supposons que l'on veut afficher chaque caractère dans un carré de 8x8 pixels. Dans ce cas, on peut stocker la représentation graphique d'un caractère en noir en blanc sous la forme d'une suite de 8 bytes. Par exemple, les huit octets ci-dessous contiennent une représentation graphique du caractère *1*.

```
00001000
00011000
00101000
00001000
00001000
00001000
00001000
00111110
```

Une représentation graphique, fortement agrandie, de ce caractère est présentée dans la Fig. 2.1.

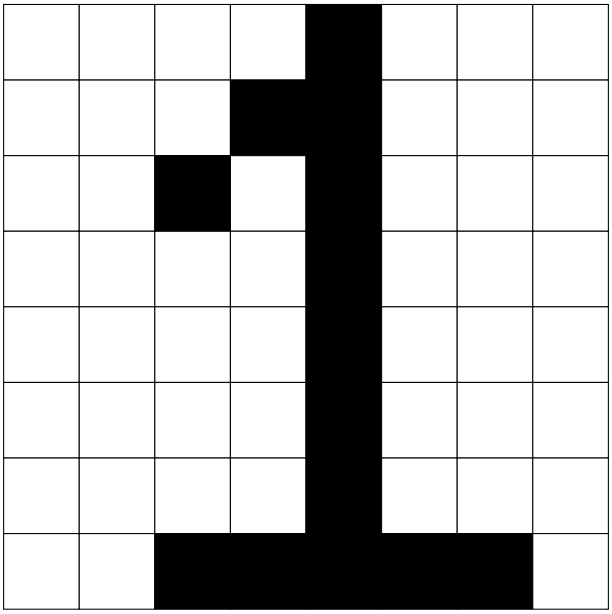


FIG. 2.1 – Un caractère sous la forme de pixels

## 2.2 Représentation des nombres naturels

Les ordinateurs ont d'abord étés conçus pour réaliser des opérations mathématiques. Il est important de pouvoir représenter des nombres dans tout ordinateur. Commençons par analyser comment représenter les nombres pour effectuer des opérations arithmétiques. Pour simplifier la présentation, nous travaillerons surtout avec des quartets dans ce chapitre. Il y a seize quartets différents :

- *0000*
- *0001*
- *0010*
- *0011*
- *0100*
- *0101*
- *0110*
- *0111*
- *1000*
- *1001*
- *1010*
- *1011*

— *1100*
— *1101*
— *1110*
— *1111*

Un tel quartet, peut se représenter de façon symbolique : $B_3B_2B_1B_0$ où les symboles $B_i$ peuvent prendre les valeurs *0* ou *1*. Dans un tel quartet, le symbole $B_3$ est appelé le bit de poids fort tandis que le symbole $B_0$ est appelé le bit de poids faible.

Cette représentation des quartets est similaire à la représentation que l'on utilise pour les nombres décimaux. Un nombre en représentation décimale peut aussi s'écrire $C_{n-1}C_{n-2}...C_2C_1C_0$. Dans cette représentation, les $C_i$ sont les chiffres de *0* à *9*. $C_0$ est le chiffre des unités, $C_1$ le chiffre correspondant aux dizaines, $C_2$ celui qui correspond aux centaines, ... Numériquement, on peut écrire que la représentation décimale $C_3C_2C_1C_0$ correspond au nombre $C_3 * 1000 + C_2 * 100 + C_1 * 10 + C_0$ ou encore $C_3 * 10^3 + C_2 * 10^2 + C_1 * 10^1 + C_0 * 10^0$ en se rappelant que $10^0$ vaut 1.

En toute généralité, la suite de chiffres décimaux $C_{n-1}C_{n-2}...C_2C_1C_0$ correspond au naturel $\sum_{i=0}^{i=n-1} C_i \times 10^i$.

A titre d'exemple, le nombre sept cent trente six s'écrit en notation décimale *736*, ce qui équivaut bien à $7 * 10^2 + 3 * 10^1 + 6 * 10^0$.

Pour représenter les nombres naturels en notation binaire, nous allons utiliser le même principe. Un nombre en notation binaire $B_{n-1}B_{n-2}...B_2B_1B_0$ représente le nombre naturel $B_{n-1}*2^{n-1}+B_{n-2}*2^{n-2}+...+B_2*2^2+B_1*2^1+B_0*2^0$. En appliquant cette règle aux quartets, on obtient aisément :

— *0000* correspond au nombre $0 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0 * 2^0$, soit *0* en notation décimale
— *0001* correspond au nombre $0 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0$, soit *1* en notation décimale
— *0010* correspond au nombre $0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0$, soit *2* en notation décimale
— *0011* correspond au nombre $0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$, soit *3* en notation décimale
— *0100* correspond au nombre $0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0$, soit *4* en notation décimale
— *0101* correspond au nombre $0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0$, soit *5* en notation décimale
— *0110* correspond au nombre $0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0$, soit *6* en notation décimale
— *0111* correspond au nombre $0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0$, soit *7* en notation décimale
— *1000* correspond au nombre $1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0 * 2^0$, soit *8* en notation décimale
— *1001* correspond au nombre $1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0$, soit *9* en notation décimale
— *1010* correspond au nombre $1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0$, soit *10* en notation décimale
— *1011* correspond au nombre $1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$, soit *11* en notation décimale
— *1100* correspond au nombre $1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0$, soit *12* en notation décimale
— *1101* correspond au nombre $1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0$, soit *13* en notation décimale
— *1110* correspond au nombre $1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0$, soit *14* en notation décimale
— *1111* correspond au nombre $1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0$, soit *15* en notation décimale

En toute généralité, la suite de bits $B_{n-1}B_{n-2}...B_2B_1B_0$ correspond au naturel $\sum_{i=0}^{i=n-1} B_i \times 2^i$.

Cette technique peut s'appliquer à des nombres binaires contenant un nombre quelconque de bits. Pour convertir efficacement un nombre binaire en son équivalent décimal, il est intéressant de connaître les principales puissances de 2 :

— $2^0 = 1$
— $2^1 = 2$
— $2^2 = 4$
— $2^3 = 8$
— $2^4 = 16$
— $2^5 = 32$
— $2^6 = 64$
— $2^7 = 128$
— $2^8 = 256$
— $2^9 = 512$
— $2^{10} = 1024$
— $2^{16} = 65536$

— $2^{20} = 1048576$ ou un peu plus d'un million
— $2^{30} = 1073741824$ ou un peu plus d'un milliard
— $2^{32} = 4294967296$ ou un peu plus de 4 milliards

Cette représentation des nombres peut se généraliser. La notation binaire utilise des puissances de *2* tandis que la notation décimale des puissances de *10*. On peut faire de même avec d'autres puissances. Ainsi, la suite de symboles $S_{n-1}S_{n-2}...S_2S_1S_0$ en base *k* où les symboles $S_i$ ont une valeur comprises entre *0* et $k-1$, correspond au naturel $\sum_{i=0}^{i=n-1} S_i \times k^i$.

En pratique, outre les notations binaires, deux notations sont couramment utilisées :
— l'octal (ou base *8*)
— l'hexadécimal (ou base *16*)

En octal, les symboles sont des chiffres de *0* à *7*. La table ci-dessous représente les correspondances entre les chiffres de *0* à *7* et les séquences de trois bits.

TABLEAU 2.2 – Représentation octale

| Chiffre | Séquence binaire |
|---|---|
| 0 | `000` |
| 1 | `001` |
| 2 | `010` |
| 3 | `011` |
| 4 | `100` |
| 5 | `101` |
| 6 | `110` |
| 7 | `111` |

En hexadécimal, les symboles sont des chiffres de *0* à *9* et les lettres de *A* à *F* sont utilisées pour représenter les valeurs de *0* à *15*. La table ci-dessous reprend la correspondances entre les 16 symboles hexadécimaux et les quartets.

TABLEAU 2.3 – Représentation hexadécimale

| Symbole | Séquence binaire |
|---|---|
| 0 | `0000` |
| 1 | `0001` |
| 2 | `0010` |
| 3 | `0011` |
| 4 | `0100` |
| 5 | `0101` |
| 6 | `0110` |
| 7 | `0111` |
| 8 | `1000` |
| 9 | `1001` |
| A | `1010` |
| B | `1011` |
| C | `1100` |
| D | `1101` |
| E | `1110` |
| F | `1111` |

On peut facilement convertir une séquence de bits en sa représentation octale ou hexadécimale. Il suffit pour cela de découper la séquence en blocs de 3 bits pour la représentation octale et en blocs de quatre bits pour la représentation hexadécimale. A titre d'exemple, la séquence de douze bits `001010011101` peut être convertie comme suit :

— en notation octale, il suffit de la découper en 4 blocs de trois bits chacun, `001 010 011 101`. Chacun de ces blocs peut ensuite être converti en notation octale, `001` correspond à `1`, `010` à `2`, `011` à `3` et `101` à `5`. La

séquence complète correspond donc à `1235`.
— en notation hexadécimale, il suffit de la découper en trois blocs de quatre bits chacun, `0010 1001 1101`. Le premier quartet correspond à `2`, le deuxième à `9` et le troisième à `D`. Le séquence complète correspond donc à `29D`.

De la même façon, les informaticiens doivent pouvoir facilement lire des séquences de bits en notation octale mais surtout hexadécimale. Pour transformer une séquence en notation hexadécimale en une séquence binaire, il suffit de remplacer chaque symbole hexadécimal par le quartet correspondant. Voici quelques exemples simples :

— `1234` est la séquence binaire `0001 0010 0011 0100`
— `0101` est la séquence binaire `0000 0001 0000 0001`
— `BEBE` est la séquence binaire `1011 1110 1011 1110`
— `CAFE` est la séquence binaire `1100 1010 1111 1110`

---

**Note :** Il est parfois intéressant d'entrer un nombre en binaire, octal ou hexadécimal dans un langage de programmation. En python3, cela se fait en préfixant le nombre avec *0b* pour du binaire, *0o* pour de l'octal et *0x* pour de l'hexadécimal. Ainsi, les lignes ci-dessous stockent toutes la valeur *23* dans la variable `n`.

```
n = 23  # décimal
n = 0b10111  # binaire
n = 0o27 # octal
n = 0x17
```

La notation adoptée dans python3 est bien plus claire que celle utilisée dans d'anciennes versions de python et des langages de programmation comme le *C*. Dans ces langages, il suffit de commencer un nombre par le chiffre zéro pour indiquer qu'il est en octal. C'était une source de très nombreuses confusions.

```
# En python2, ces deux lignes ne sont pas équivalentes
n = 23  # décimal
n = 023 # octal -> valeur décimale 19
```

---

CHAPITRE 3

# Langage d'assemblage

Nous pouvons commencer à programmer un micro-processeur qui est capable d'exécuter de petits programmes. Ce micro-processeur répond à ce que l'on appelle l"architecture de Von Neumann.

Cette architecture est composée d'un processeur (CPU en anglais) ou unité de calcul et d'une mémoire. Le processeur est un circuit électronique qui est capable d'effectuer de nombreuses tâches :

— lire de l'information en mémoire
— écrire de l'information en mémoire
— réaliser des calculs

L'architecture des ordinateurs est basée sur l'architecture dite de Von Neumann. Suivant cette architecture, un ordinateur est composé d'un processeur qui exécute un programme se trouvant en mémoire. Ce programme manipule des données qui sont aussi stockées en mémoire.

Dans un ordinateur, toutes les données et les programmes sont représentés sous la forme de séquences de bits. Pour exécuter un programme, le microprocesseur doit charger les séquences de bits qui correspondent aux instructions depuis la mémoire ainsi que les données qui y sont associées. Le programme est découpé en de très nombreuses instructions très simples, beaucoup plus simples que celles que l'on trouve dans un langage de programmation comme python. Chaque microprocesseur est caractérisé par l'ensemble des instructions qu'il peut exécuter. Les premiers microprocesseurs supportaient quelques dizaines d'instructions différentes. Les processeurs récents en supportent beaucoup plus.

Même si dans l'ordinateur chaque instruction à exécuter est représentée sous la forme d'une séquence de bits, ces séquences sont peu pratiques pour les informaticiens et informaticiennes qui doivent les utiliser pour écrire des programmes. Pour écrire de tels programmes, il est préférable de passer par le langage d'assemblage. Un langage d'assemblage est la liste de toutes les instructions simples qui sont supportées par un microprocesseur donné. Chaque microprocesseur dispose de son langage d'assemblage.

Dans ce syllabus, nous nous concentrons sur un langage d'assemblage simple qui ne correspond pas à un ordinateur réel, mais contient les instructions que l'on retrouve dans la plupart des langages d'assemblage. Son avantage principal est qu'il est très facile d'exécuter des programmes écrits dans ce langage en utilisant le simulateur disponible en ligne. Ce simulateur est utilisable depuis n'importe quel navigateur web.

## 3.1 Notre simulateur

Dans le cadre de ce syllabus, nous utilisons le [simulateur de microprocesseur](https://github.com/Schweigi/assembler-simulator) développé initialement par [Marco Schweighauser](https://github.com/Schweigi) et amélioré par [Nikita Tyunyayev](https://github.com/ntyunyayev). Ce simulateur est accessible en ligne depuis n'importe quel navigateur web via http://asm.info.ucl.ac.be . La Fig. 3.1 présente l'interface graphique de notre simulateur..

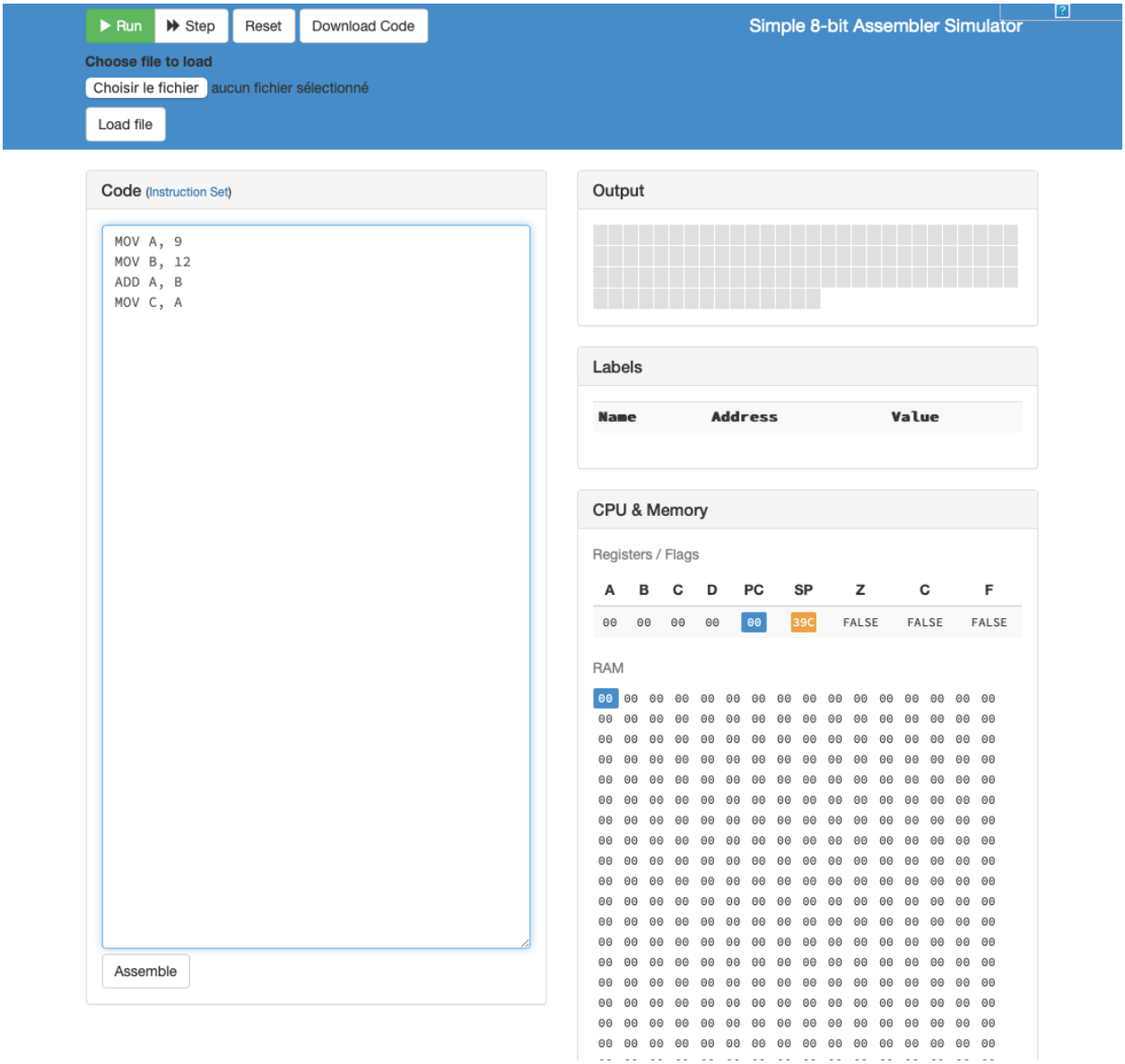



FIG. 3.1 – Capture d'écran du simulateur de processeur

Notre simulateur comprend plusieurs boutons en haut de l'écran :

- `Run` pour exécuter l'ensemble du programme pas à pas
- `Step` pour exécuter l'instruction suivante et s'arrêter
- `Reset` pour revenir au début de l'exécution du programme
- `Download Code` pour télécharger sur votre ordinateur le programme chargé sur le simulateur
- `Choisir le fichier` pour sélectionner un fichier sur votre ordinateur
- `Load file` pour charger le fichier sélectionné depuis votre ordinateur vers le simulateur

La grande boîte baptisée `Code` comprend le code en assembleur à exécuter. La boîte baptisée `Output` est un écran simplifié que vous pourrez utiliser pour afficher quelques caractères. La boîte `CPU & Memory` contient d'abord les valeurs des données stockées dans les registres et les drapeaux du processeur. La partie dénommée `RAM` représente le contenu de la mémoire. Les informations se trouvant dans les registres et la mémoire sont représentées en notation hexadécimale.

## 3.2 Un assembleur simple

Tout langage d'assemblage dépend des caractéristiques matérielles du microprocesseur qui supporte ses instructions. Notre processeur contient quatre registres que vous pouvez utiliser pour stocker des données. Ils sont identifiés par les lettres `A`, `B`, `C` et `D`. Chacun de ces registres peut stocker un bloc de 16 bits de données. Comme dans tout stockage binaire de l'information, c'est au programmeur de décider ce que représente un bloc de bits. Il peut s'agir d'un caractère, d'un nombre ou de tout autre type d'information.

Notre ordinateur comprend aussi une mémoire RAM (ou index :*Random Access Memory*) qui permet de stocker des données et des instructions. Toutes les informations stockées dans la RAM sont sous forme binaire. Tout comme pour les registres, c'est au programmeur de décider si une information se trouvant en mémoire représente un caractère, un nombre ou une instruction.

Notre processeur simple supporte quelques dizaines d'instructions que nous allons découvrir petit à petit. La première instruction est baptisée MOV. L'instruction `MOV` prend deux arguments :

— une destination
— une source

La syntaxe de base de cette instruction est : `MOV dest, src`. La destination est un identifiant de registre (`A`, `B`, `C` ou `D`) et la source peut être un identifiant de registre ou une constante. Cette constante peut être spécifiée en notation binaire, décimale, octale ou hexadécimale. Lorsque l'on spécifie une constante, c'est généralement la notation décimale qui est utilisée, mais parfois il est intéressant d'utiliser une des autres notations, notamment lorsque l'on veut spécifier une séquence de bits spécifique.

L'instruction `MOV` permet de placer un bloc de 16 bits dans un registre ou de déplacer un bloc de 16 bits d'un registre à l'autre.

Dans l'assembleur que nous utilisons, chaque instruction est représenté par un mot clé en majuscules suivi de ses arguments sur une ligne. Le caractère `;` est utilisé pour marquer le début d'un commentaire. Une ligne qui débute par le point virgule n'est donc pas considérée comme une instruction. Il en va de même pour tous les caractères qui suivent le point virgule sur une ligne quelconque.

Nous pouvons maintenant exécuter notre première séquence d'instructions. Notre microprocesseur va d'abord exécuter la première instruction. Il exécutera ensuite la deuxième et enfin la troisième.

```
1 MOV A,1
2 MOV B,2
3 MOV A,B
```

Dans l'exemple ci-dessus, la première ligne place la représentation binaire du nombre naturel `1`, c'est-à-dire `0000000000000001` dans le registre `A`. La deuxième ligne contient l'instruction qui permet de placer la représentation binaire du nombre naturel 2 dans le registre `B`. La troisième instruction permet elle de copier les 16 bits qui se trouvent dans le registre `B` (c'est-à-dire la valeur `2`) dans le registre `A`. Après l'exécution de ces trois instructions, les registres `A` et `B` contiennent tous les deux la séquence `0000000000000010`.

L'instruction `MOV`, et toutes les instructions de l'assembleur que nous utilisons, permettent de spécifier leurs arguments numérique en notation, binaire, décimale, octale et hexadécimale. Dans le cours, nous privilégierons la notation décimale qui est la plus courante, mais les autres notations sont parfois utiles lorsque l'on veut stocker un blocs de 16 bits bien particulier. Les quatre instructions ci-dessous placent toute la valeur vingt trois dans le registre `D`.

```
MOV D, 23d       ; en notation décimale
MOV D, 0x17      ; en notation hexadécimale
MOV D, 0o27      ; en notation octale
MOV D, 10111b    ; en notation binaire
```

Notre processeur peut également réaliser des opérations arithmétiques sur les données stockées dans ses registres. Les opérations arithmétiques les plus simples sont INC et DEC. Elles prennent toutes les deux comme argument un identifiant de registre. L'instruction `INC X` incrémente le nombre entier stocké dans le registre `X` d'une unité. L'instruction `DEC X` décrémente d'une unité la valeur entière stockée dans le registre `X`.

A titre d'exemple, considérons la séquence d'instructions suivante.

```
; première solution
MOV A, 7
; deuxième solution
MOV B, 6
INC B
; troisième solution
MOV C, 8
DEC C
; quatrième solution
MOV D, 7
DEC D
INC D
```

Après l'exécution de ces instructions, les quatre registres de notre processeur contiennent tous la valeur entière `7`. La première ligne est, évidemment, la meilleure solution pour placer cette valeur dans le registre `A`, mais les autres aboutissent au même résultat.

Notre processeur peut aussi additionner et soustraire les valeurs entières stockées dans des registres. L'instruction ADD prend deux arguments. Le premier est le registre qui est la destination du résultat. Le second est un registre ou une constante. Cette instruction calcule la somme entre ses deux arguments et place le résultat dans le premier argument. L'instruction SUB prend également deux arguments. Elle stocke dans son premier argument le résultat de l'opération `arg1 - arg2`.

```
MOV A, 7
MOV B, 3
ADD A, B
```

La séquence d'instructions ci-dessus place dans le registre `A` la somme entre la valeur stockée dans ce registre et celle se trouvant dans le registre `B`, c'est-à-dire la valeur `10`.

```
MOV A, 7
MOV B, 3
SUB A, B
```

La séquence d'instruction ci-dessus place dans le registre `A` le résultat de l'opération `7 - 3`, c'est-à-dire la valeur `4`.

Il est intéressant de noter que comme l'instruction `ADD` ne prend que deux arguments, il n'est pas possible, en une seule instruction, de calculer la somme entre deux registres et de la placer dans un troisième. Cela nécessite deux instructions comme dans la séquence ci-dessous qui place dans le registre `C` la somme entre les valeurs stockées dans les registres `A` et `B`.

```
MOV A, 9
MOV B, 12
ADD A, B
MOV C, A
```

Il est important de noter qu'après exécution de la séquence d'instructions ci-dessus, la valeur qui était stockée dans le registre `A` est perdue. Si cette valeur était importante pour la suite du programme, alors il est préférable d'utiliser la séquence d'instructions qui suit qui elle utilise le registre `C` comme mémoire intermédiaire.

```
MOV A, 9
MOV B, 12
MOV C, A
ADD C, B
```

Notre microprocesseur est aussi capable de réaliser des opérations de multiplication et de division. Cependant, ces opérations, qui sont nettement plus complexes à implémenter que les additions et soustractions, ne peuvent que porter sur la valeur se trouvant dans le registre `A`. Il est donc nécessaire de d'abord placer la valeur à multiplier ou diviser dans ce registre avant de pouvoir réaliser l'opération.

En utilisant l'instruction MUL, il est possible de multiplier une valeur entière stockée dans le registre `A` par une constante.

```
MOV A, 9
MUL 3
```

Après exécution du code ci-dessus, le registre `A` contient la valeur décimale `27`. Il est aussi possible de multiplier la valeur entière stockée dans le registre `A` par la valeur se trouvant dans un autre registre.

```
MOV A, 9
MOV B, 2
MUL B
```

Après exécution des instructions ci-dessus, le registre `A` contient la valeur décimale `18`. Le registre `B` contient lui toujours la valeur `2`.

L'instruction DIV s'utilise de façon similaire. Il est possible de diviser la valeur se trouvant dans le registre `A` par une constante entière.

```
MOV A, 24
DIV 3
```

Après exécution de ces instructions, le registre `A` contient la valeur `8` qui est le quotient de la division de `24` par `3`. Tout comme pour l'instruction de multiplication, il est possible de diviser la valeur stockée dans le registre `A` par une valeur entière se trouvant dans un autre registre.

```
MOV A, 35
MOV B, 7
DIV B
```

Après exécution de ces instructions, le registre `A` contient la valeur `5`. Il est important de noter que notre processeur calcule le quotient de la division entière entre le dividende stocké dans le registre `A` et le diviseur qui peut être une constante ou se trouver dans un autre registre. En python, la séquence d'instructions ci-dessous permet également de calculer le quotient de cette division entière.

```
a=35
b=7
a=a//b
```

Notre langage d'assemblage ne contient pas d'instruction permettant de calculer le reste d'une division entière. Si vous avez besoin de cette opération, vous devrez la programmer en utilisant les autres instructions du langage.

Une dernière instruction qui nous sera utile par après est l'instruction HLT. Cette instruction permet d'arrêter l'exécution du programme. Il faut pousser sur le bouton *Reset* du simulateur pour redémarrer le processeur.

## 3.3 Interaction avec la mémoire RAM

A côté des instructions de calcul telles que celles qui viennent d'être présentées, notre microprocesseur simple est aussi capable d'interagir avec la mémoire. Il existe plusieurs type de mémoires dans un ordinateur. Les deux plus simples sont les Random Access Memory (RAM) et les Read-Only Memory (ROM).

Comme son nom l'indique, une mémoire ROM est une mémoire dont le contenu ne peut qu'être lu. Le contenu de cette mémoire est écrit lors de la construction de la mémoire et ne peut jamais être modifié. Ces mémoires sont utilisées pour stocker des données ou des programmes qui ne changent jamais, comme par exemple le code qui permet de faire démarrer un ordinateur et de lancer son système d'exploitation. Une mémoire ROM peut se représenter comme dans la Fig. 3.2. Une caractéristique important des mémoires de type ROM est que leur contenu est préservé même lorsque

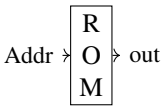



FIG. 3.2 – Une mémoire ROM

la mémoire est mise hors tension. Certaines mémoires de type ROM sont dites programmables car il est possible d'effacer et de modifier leur contenu. C'est le cas par exemple des EPROM (Electrically Programmable ROM) ou des EEPROM (Electrically Erasable and Programmable ROM). La programmation d'un tel circuit se fait en utilisant un dispositif spécialisé qui sort du cadre de ce cours.

Une mémoire ROM peut être vue comme un tableau permettant de stocker des données (dans notre ordinateur des blocs de 16 bits). Chaque élément du tableau est identifié par une adresse. A titre d'exemple, considérons une mémoire ROM qui permet de stocker 4 blocs de 16 bits.

TABLEAU 3.1 – Mémoire ROM permettant de stocker 4 blocs de 16 bits

| Adresse | Valeur |
|---|---|
| 0b11 | 0b1100100100100111 |
| 0b10 | 0b0000000000000001 |
| 0b01 | 0b0000000000000000 |
| 0b00 | 0b1111111111111111 |

La mémoire ROM représentée dans la table Tableau 3.1 contient quatre blocs de 16 bits. Le microprocesseur peut accéder à chacun de ces blocs en indiquant à la mémoire l'adresse à laquelle il est stocké. Ainsi, la donnée stockée à l'adresse `0b01` en mémoire ROM, que l'on pourrait schématiser par la notation `ROM[0b01]`, est la valeur `0b0000000000000001` ou `1` en notation décimale.

Une mémoire ROM utilise un nombre de bits d'adresse qui dépend de sa capacité. Une mémoire permettant de stocker $2^n$ blocs de données utilisera des adresses qui sont stockées sur $n$ bits. Il faut cependant noter que dans la plupart des ordinateurs, les mémoires sont organisées de façon à associer une adresse à chaque octet ou bloc de 8 bits.

Lorsque l'on doit stocker un bloc de 16 bits dans une mémoire dont l'unité de stockage est l'octet (8 bits), il faut se mettre d'accord sur la convention utilisée pour stocker les bits de poids fort et les bits de poids faible. Considérons la séquence de bits `0b 01000000 00000001`. Cette séquence de bits correspond à la valeur entière `8193`. Dans une mémoire dont l'unité de stockage est l'octet, elle peut être stockée de deux façons différentes. La première convention, baptisée big-endian stocke l'octet de poids fort (`0b01000000`) à l'adresse la plus petite comme illustré dans la table Tableau 3.2. C'est la convention qui est utilisée par notre assembleur ainsi que par certains ordinateurs actuels. Gardez cette convention en tête lorsque vous analysez le contenu d'une mémoire.

TABLEAU 3.2 – Mémoire de deux octets contenant la séquence `0b01000000 00000001` en utilisant la convention *big-endian*

| Adresse | Valeur |
|---|---|
| 0b1 | 0b00000001 |
| 0b0 | 0b01000000 |

L'autre convention, baptisée little-endian, est de stoker les bits de poids fort à l'adresse la plus élevée comme représenté dans Tableau 3.3. Cette convention est utilisée par certains ordinateurs actuels.

TABLEAU 3.3 – Mémoire de deux octets contenant la séquence `0b01000000 00000001` en utilisant la convention *little-endian*

| Adresse | Valeur |
|---|---|
| 0b1 | 0b01000000 |
| 0b0 | 0b00000001 |

Une mémoire RAM a une organisation similaire. Chaque zone de mémoire permettant de stocker un octet est identifié par une adresse. Tout comme dans une mémoire ROM, le nombre de bits utilisé pour représenter chaque adresse dépend de la capacité de la mémoire.

Dans une mémoire RAM, outre les entrées relatives aux adresses, il faut aussi avoir une entrée *load* (parfois appelée *read/write*) pour déterminer si la mémoire doit lire ou écrire une donnée et une entrée *data* permettant de charger des données dans la *RAM*. Le nombre de bits d'adresses dépend uniquement de la capacité de la mémoire. En général, une adresse correspond à un octet stocké en mémoire. L'entrée *data* quant à elle peut permettre de charger des octets, des mots de 16, 32 bits ou encore plus. La Fig. 3.3 représente une mémoire RAM de façon schématique. Cette mémoire

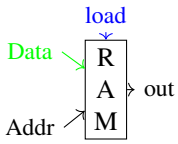



FIG. 3.3 – Une mémoire RAM

RAM peut être utilisée par notre microprocesseur comme mémoire permettant de stocker à la fois les instructions et les données. Tant les instructions que les données sont stockées sous la forme de séquences de bits. Nous analyserons plus tard comment représenter une instruction comme une séquence de bits. Pour le moment, concentrons-nous sur l'utilisation de la mémoire RAM pour stocker des données qui seront utilisées par nos programmes.

Le langage d'assemblage nous permet de précharger en mémoire RAM des constantes qui pourront ensuite être utilisées dans notre programme. Le mot clé DB permet de stocker en mémoire le mot de 16 bits qui suit le mot-clé.

A titre d'exemple, le code ci-dessous stocke le bloc de 16 bits `0b00000000 0000011` à l'adresse `0` en mémoire et le bloc `0b00000000 00000111` à l'adresse `2`.

```
DB 3
DB 7
```

**Note :** DB n'est pas une instruction

Le mot clé `DB` n'est pas une instruction du langage d'assemblage, c'est une directive qui indique à l'assembleur de simplement placer en mémoire la valeur qui suit le mot-clé `DB`. Cette valeur peut être une valeur décimale, une valeur binaire ou une valeur en notation hexadécimale. La Fig. 3.4 illustre l'utilisation de ce mot-clé `DB`. Il est intéressant de regarder le contenu de la mémoire RAM et de le mettre en parallèle avec les mots-clés `DB`.

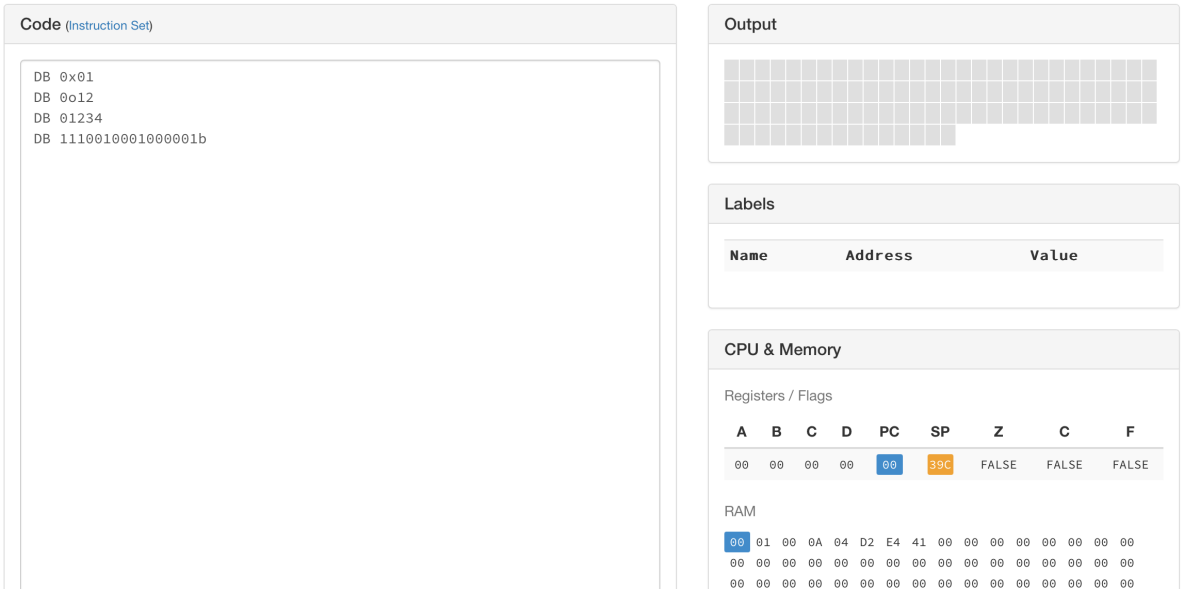



FIG. 3.4 – Exemple d'utilisation du mot-clé DB

En pratique, le mot clé `DB` sera rarement utilisé de cette façon. Dans un programme, on utilisera ce mot clé pour définir des constantes ou alors pour fixer la valeur initiale de certaines variables. Dans ces deux cas d'utilisation, il est important pour le programmeur de pouvoir connaître l'adresse mémoire correspondant à chacune de ces variables ou constantes. L'assembleur que nous utilisons permet d'associer une étiquette (label en anglais) à certaines adresses mémoire. Ces étiquettes peuvent ensuite être utilisées par les instructions du programme et elles sont automatiquement traduites par l'assembleur en l'adresse correspondante. Pour définir une étiquette, il suffit d'écrire sur une ligne une chaîne de caractères suivie par le caractère `:` et ensuite l'instruction en assembleur (souvent `DB`).

Afin d'illustrer l'utilisation de ces étiquettes, considérons la liste de mots-clés `DB` ci-dessous.

```
zero:       DB 0x0000
lln:        DB 1348
charleroi:  DB 6000
```

Cette suite de mots-clés nous permet d'initialiser en mémoire trois constantes et d'associer une étiquette à chacune de ces constantes. La première ligne place la valeur `0x0000` en mémoire. Cette valeur se trouve à l'adresse `0` et l'assembleur associe l'étiquette *zero* à cette adresse. La deuxième ligne place la valeur `1348` en mémoire à l'adresse *2* et associe l'étiquette *lln* à cette adresse. Enfin, la troisième ligne place la valeur `6000` en mémoire à l'adresse suivante (*4*) et associe l'étiquette *charleroi* à cette adresse. La Fig. 3.5 présente comment le simulateur affiche ces différentes étiquettes et les valeurs associées.

Les instructions de l'assembleur telles que `MOV` peuvent utiliser des étiquettes de deux façons différentes. Tout d'abord, si une étiquette apparaît dans une instruction en assembleur, elle est automatiquement remplacée par l'adresse mémoire à laquelle elle correspond. Si cette étiquette est placée entre crochets (`[` et `]`), alors le processeur ira chercher la donnée qui se trouve en mémoire à l'adresse de l'étiquette.

Le programme ci-dessous définit deux étiquettes : *x* et *y*. Il initialise la valeur stockée à l'adresse *x* à `3` et celle stockée à l'adresse *y* à `7` via les deux commandes `DB`. La première instruction place dans le registre `A` la valeur qui se trouve en mémoire à l'adresse de l'étiquette *x*, c'est-à-dire la valeur `3`. La deuxième instruction place dans le registre `B` la valeur qui se trouve en mémoire à l'adresse de l'étiquette *y*, c'est-à-dire la valeur `7`. Ensuite, l'instruction `ADD` place la valeur `10` dans le registre `A`.

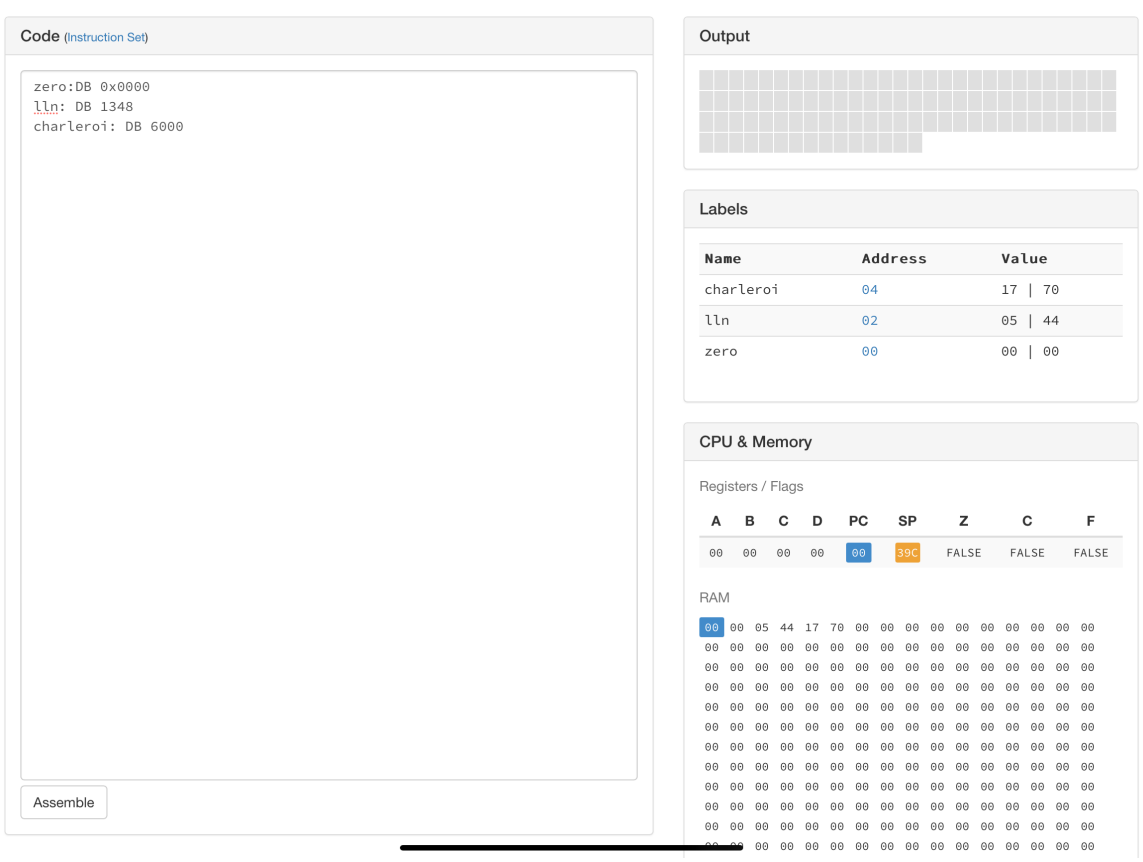



FIG. 3.5 – Exemple d'utilisation du mot-clé DB

```
MOV A, [x]
MOV B, [y]
ADD A, B
HLT
;  Variables et données du programme
x: DB 3
y: DB 7
```

Lorsque le programme ci-dessus est transformé en langage machine et stocké en mémoire, l'instruction HLT se trouve à l'adresse `0x12`. L'étiquette *x* correspond à l'adresse `0x14` et *y* à l'adresse `0x16` comme illustré dans la Fig. 3.6.

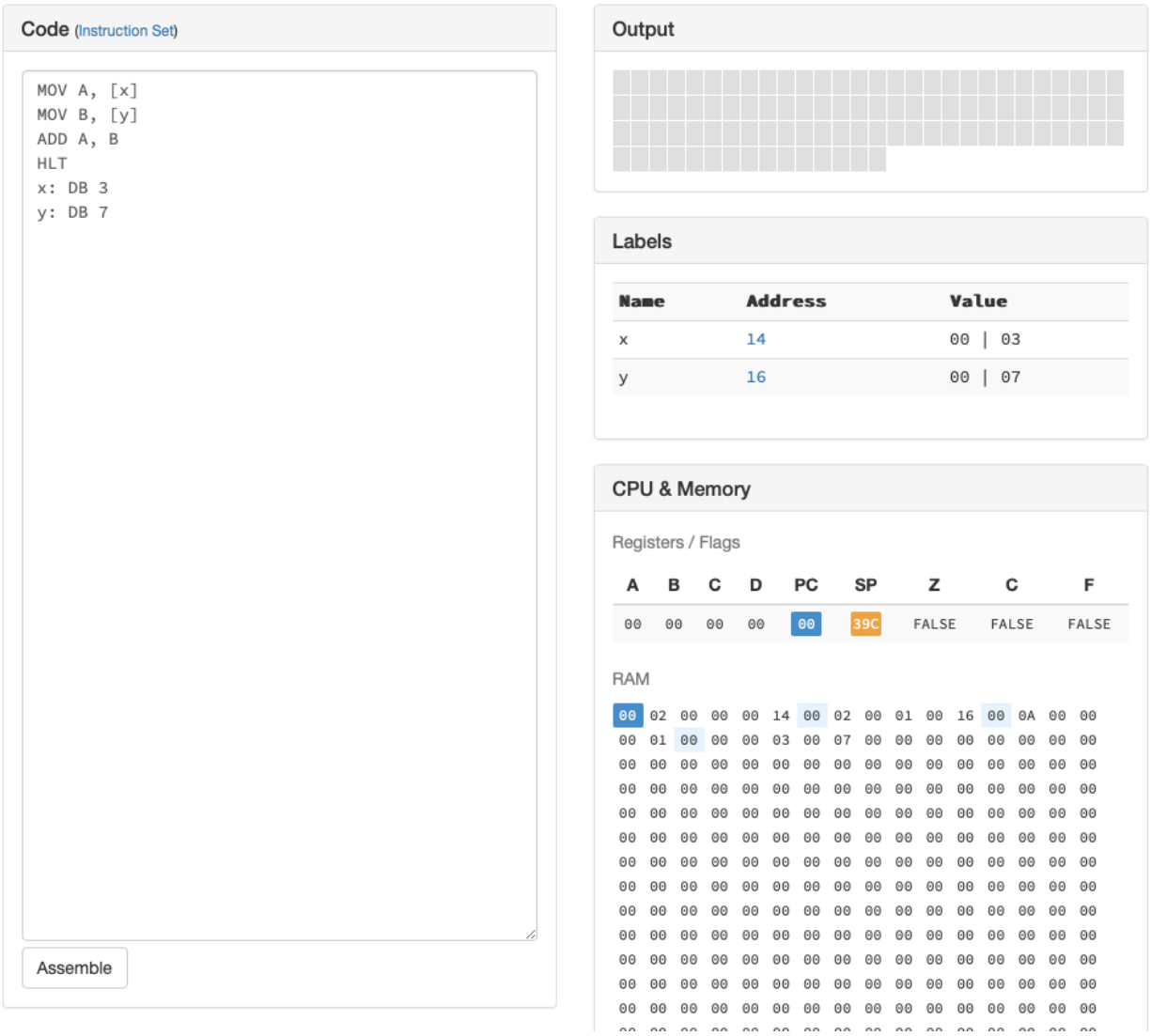



FIG. 3.6 – Adresses mémoires des étiquettes x et y

A ce stade, il est utile d'analyser un peu plus en détails la façon dont les instructions sont encodées en mémoire. Pour le processeur, une instruction en assembleur est aussi encodée sous la forme d'une séquence de bits. C'est ce que nous voyons dans la mémoire RAM présentée dans la Fig. 3.6. Analysons cette mémoire bloc de 16 bits par bloc de 16 bits. Le premier bloc, la notation `0x00 02` correspond au code opératoire (ou opcode en anglais) de l'instruction qui permet de déplacer une information se trouvant en mémoire vers un registre. Cette instruction prend deux arguments :

— un registre
— une adresse

Le bloc de 16 bits à l'adresse `0x02` qui a comme valeur `0x00 00` correspond au registre `A`. Le bloc de 16 bits à l'adresse `0x04` contient lui la valeur `0x00 14` qui est l'adresse de l'étiquette *x*. Ces 6 premiers octets (*0x00 02 00 00 00 14* `) sont la représentation binaire de l'instruction `MOV A, [14]`. Les six octets qui suivent (`0x00 02 00 01 00 16`) correspondent eux à l'instruction `MOV B, [16]`. Le code opératoire de l'instruction `ADD` est `0x00 0A` et les 6 octets `0x00 0A 00 00 00 01` représentent bien l'instruction `ADD A, B`. L'instruction `HLT` a comme code opératoire le bloc de 16 bits `0x00 00`. Le simulateur définit un code opératoire pour chaque variante d'une instruction. En voici quelques unes à titre d'illustration :

— `HLT` a comme code opératoire `0x00 00`
— `MOV` a comme code opératoire `0x00 01` lorsque ses deux arguments sont des registres
— `MOV` a comme code opératoire `0x00 06` lorsque son premier argument est un registre et le seconde une constante
— `ADD` a comme code opératoire `0x00 0A` lorsque ses deux arguments sont des registres
— `SUB` a comme code opératoire `0x00 0D` lorsque ses deux arguments sont des registres
— `INC` a comme code opératoire `0x00 12` et est suivi d'un identifiant de registre
— `DEC` a comme code opératoire `0x00 13` et est suivi d'un identifiant de registre
— `MUL` a comme code opératoire `0x00 3C` et est suivi identifiant de registre

Connaissant cette représentation des instructions en assembleur sous la forme de séquence de bits, il est possible (mais pas recommandé) d'écrire un programme assembleur en utilisant uniquement les commandes `DB` pour initialiser la mémoire. Pouvez-vous prévoir ce que fait le « programme » présenté ci-dessous ?

```
DB 0x0012
DB 0x0001
DB 0x0001
DB 0x0003
DB 0x0001
DB 0
```

Nous pouvons maintenant lister tous les arguments possibles de l'instruction MOV :

— `MOV reg1, reg2` (`reg1` et `reg2` sont des identifiants de registres) : place dans `reg1` la valeur se trouvant actuellement dans `reg2`. Le contenu de `reg2` n'est pas modifié
— `MOV reg, cst` (`reg` est un identifiant de registre et `cst` une constante) : place dans le registre `reg` la valeur `cst`
— `MOV reg, adr` (`reg` est un identifiant de registre et `adr` une adresse en mémoire ou une étiquette) : place dans `reg` l'adresse `adr`
— `MOV reg, [adr]` (`reg` est un identifiant de registre et `adr` une adresse en mémoire ou une étiquette) : place dans `reg` la valeur se trouvant en mémoire à l'adresse `adr`
— `MOV adr, reg` (`reg` est un identifiant de registre et `adr` une adresse en mémoire ou une étiquette) : place la valeur se trouvant dans le registre `reg` en mémoire à l'adresse `adr`

Les instructions `ADD` et `SUB` prennent les mêmes arguments que les quatre premiers types d'instruction `MOV` (le résultat de `ADD` et `SUB` se trouve toujours dans un registre). Les instructions `INC` et `DEC` ne prennent qu'un registre comme argument.

Les instructions `MUL` et `DIV` supportent elles trois types d'arguments :

— `MUL reg` (`reg` est un identifiant de registre) : place dans le registre `A` le résultat du produit entre la valeur se trouvant dans le registre `reg` et le registre `A`.
— `MUL cst` (`cst` est une valeur entière) : place dans le registre `A` le résultat du produit entre la valeur entière passée en argument et le registre `A`.
— `MUL [adr]` (`adr` est une adresse en mémoire ou une étiquette) : place dans le registre `A` le résultat du produit entre la valeur se trouvant à l'adresse passée en argument et le registre `A`.

L'instruction `DIV` prend également ces trois types d'arguments.

Avec ces instructions qui permettent de manipuler des données se trouvant en mémoire, il est possible de gérer des variables et de réaliser des opérations arithmétiques sur ces variables en mémoire. A titre d'exemple, considérons le programme python ci-dessous.

```
x=3
y=5
z=x+2*y
```

Lors de son exécution, ce programme place dans la variable `z` la valeur `13`. Pour écrire un programme équivalent en assembleur, nous devons d'abord réserver une zone mémoire pour stocker chacune des trois variables. Cela se fait en utilisant les trois lignes en fin de programme avec le mot-clé `DB`. Nous initialisons la variable `z` à la valeur `0`. Ces zones mémoire étant définies et initialisées, nous pouvons d'abord calculer l'expression $2 * y$ et stocker son résultat dans le registre `A`. Ensuite, il suffit d'ajouter le contenu de la variable `x` au résultat obtenu et de sauver le résultat de l'addition à l'adresse de la variable `z`.

```
MOV A, [y]      ; place la valeur de la variable y dans A
MUL 2           ; multiplie le contenu de A par 2
ADD A, [x]      ; ajoute au contenu de A la valeur de la variable x
MOV z, A        ; sauvegarde du résultat du calcul dans la variable z
HLT
;  Variables et données du programme
x: DB 3
y: DB 5
z: DB 0
```

En mathématiques, on sait que les expressions $x * x - y * y$ et $(x - y) * (x + y)$ sont équivalents. Vérifions en python et en assembleur que c'est bien le cas lorsque la variable `x` vaut `9` et la variable `y` vaut `2`.

```
x=9
y=2
z1=x*x-y*y
z2=(x-y)*(x+y)
```

Pour traduire ces lignes de python en assembleur, nous devons découper les expressions mathématiques en sous-expressions qui sont réalisables avec les instructions `ADD`, `SUB` et `MUL`. Commençons par l'expression $x * x - y * y$. Nous pouvons d'abord calculer les deux carrés et les stocker dans deux registres avant de réaliser la soustraction.

```
MOV A, [y]      ; place la valeur de la variable y dans A
MUL A           ; multiplie le contenu de A par lui-même
MOV B, A        ; sauvegarde du résultat dans le registre B
MOV A, [x]      ; place la valeur de la variable x dans A
MUL A           ; multiplie le contenu de A par lui-même
SUB A, B        ; soustraction des deux carrés
MOV z1, A       ; sauvegarde du résultat du calcul dans la variable z1
MOV A, [x]      ; place la valeur de la variable x dans A
SUB A, [y]      ; calcul de x-y
MOV B, [x]      ; place la valeur de la variable x dans B
ADD B, [y]      ; calcul de x+y
MUL B
MOV z2, A       ; sauvegarde du résultat du calcul dans la variable z2
HLT
;  Variables et données du programme
x: DB 9
y: DB 2
z1: DB 0
z2: DB 0
```

En exécutant ce programme dans le simulateur, on peut facilement vérifier que les zones mémoires étiquetées `z1` et `z2` contiennent bien le même naturel avec l'exécution du programme.

## 3.4 Instructions logiques

Comme nous l'avons indiqué précédemment, un microprocesseur manipule des séquences de bits. Outre les opérations arithmétiques que nous venons de voir, il est parfois intéressant de réaliser des opérations directement sur les séquence de bits. Cela se fait en utilisant les instructions logiques qui s'appuient sur les opérations booléennes. Une opération booléenne est une fonction qui prend en entrée 0, 1 ou plusieurs bits et retourne un résultat.

### 3.4.1 Fonctions booléennes

La fonction booléenne la plus simple est la fonction identité. Elle prend comme entrée un bit et retourne la valeur de ce bit. On peut la définir en utilisant une table de vérité qui indique la valeur du résultat de la fonction pour chaque valeur possible de son entrée. Dans la table ci-dessous, la colonne *x* contient les différentes valeurs possibles de l'entrée *x* et la valeur du résultat pour chacune des valeurs possibles de *x*.

| x | identité(x) |
|---|---|
| 0 | 0 |
| 1 | 1 |

Cette fonction n'est pas très utile en pratique. Elle nous permet d'illustrer une table de vérité simple dans laquelle il y a une valeur binaire en entrée et une valeur binaire également en sortie.

Une fonction plus intéressante est l"inverseur, aussi dénommée NOT(x) en anglais. Cette fonction prend comme entrée un bit. Si le bit d'entrée vaut *1*, elle retourne *0*. Tandis que si le bit d'entrée vaut *0*, elle retourne *1*. Cette fonction sera très fréquemment utilisée pour construire des circuits électroniques.

| x | NOT(x) |
|---|---|
| 0 | 1 |
| 1 | 0 |

Il y a encore deux fonctions que l'on peut construire avec une seule entrée binaire. La première, baptisée Toujours0, retourne toujours la valeur `0`, quelle que soit son entrée. La seconde, baptisée Toujours1 retourne toujours la valeur `1`. Voici leurs tables de vérité.

| x | Toujours0(x) |
|---|---|
| 0 | 0 |
| 1 | 0 |

| x | Toujours1(x) |
|---|---|
| 0 | 1 |
| 1 | 1 |

La logique booléenne devient nettement plus intéressante lorsque l'on considère des fonctions qui prennent plus d'une entrée.

### 3.4.2 Fonctions booléennes à deux entrées

Plusieurs fonctions booléennes classiques existent. Les premières correspondent à la conjonction (*et*) et à la disjonction (*ou*) en logique. Commençons par la fonction *AND*. Celle-ci correspond à la table de vérité suivante :

| x | y | AND(x,y) |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Cette table comprend quatre lignes qui correspondent à toutes les combinaisons possibles des deux entrées de la fonction. On remarque aisément que la fonction AND(x,y) retourne la valeur `1` uniquement lorsque ses deux entrées ont la valeur `1`. Si une des deux entrées de la fonction *AND(x,y)* a la valeur `0`, alors sa sortie est nécessairement `0`. Cette fonction est bien l'équivalent de la conjonction logique si l'on applique la convention que `0` représente la valeur *Faux*.

La fonction OR(x,y), quant à elle, est l'équivalent de la disjonction logique. Sa table de vérité est reprise ci-dessous.

| x | y | OR(x,y) |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

On remarque aisément que la fonction *OR(x,y)* correspond bien à la disjonction logique lorsque *1* représente la valeur *Vrai*. Cette fonction *OR(x,y)* ne retourne la valeur `0` que si ses deux entrées valent `0`. Dans tous les autres cas, elle retourne la valeur `1`.

Ces fonctions peuvent être combinées entre elles. Un premier exemple est d'appliquer un inverseur (opération *NOT* au résultat de la fonction *AND*). Cette fonction booléenne s'appelle généralement NAND(x,y) (*NOT AND*) et sa table de vérité est la suivante. On pourra dire que $NAND(x, y) \iff NOT(AND(x, y))$.

| x | y | NAND(x,y) |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

De même, la fonction NOR(x,y) s'obtient en inversant le résultat de la fonction *OR*. On pourra dire que $NOR(x, y) \iff NOT(OR(x, y))$.

| x | y | NOR(x,y) |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

Il est important de noter que *NOR(x,y)* n'est pas équivalent à la fonction *OR(NOT(x),NOT(y))*. La table de vérité de cette dernière fonction est reprise ci-dessous.

| x | y | NOT(x) | NOT(y) | OR(NOT(x),NOT(y)) |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |

Il existe d'autres fonctions booléennes à deux entrées qui sont utiles en pratique. Parmi celles-ci, on retrouve la fonction XOR(x,y) qui retourne la valeur *1* uniquement si une seule de ses entrées a la valeur *1*. Sa table de vérité est reprise ci-dessous. On remarquera qu'elle diffère de celle des autres fonctions booléennes que nous avons déjà présenté.

| x | y | XOR(x,y) |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Ces opérations logiques peuvent être réalisées bit à bit sur des blocs de 16 bits tels que ceux qui sont stockés dans les registres de notre processeur ou en mémoire. On peut aisément définir l'opération `NOT` sur le mot de 16 bits $b_{15}b_{14}b_{13}b_{12}b_{11}b_{10}b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0$ comme suit :

$NOT(b_{15}b_{14}b_{13}b_{12}b_{11}b_{10}b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0) =$
$not(b_{15})not(b_{14})not(b_{13})not(b_{12})not(b_{11})not(b_{10})not(b_9)not(b_8)not(b_7)not(b_6)not(b_5)not(b_4)not(b_3)not(b_2)not(b_1)not(b_0)$
où $not(...)$ est l'opération NOT appliquée à un bit définie plus haut.

De la même façon, on peut définir les opérations qui prennent deux arguments telles que `OR` ou `AND` comme suit :

$OR(a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0, b_{15}b_{14}b_{13}b_{12}b_{11}b_{10}b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0) =$
$or(a_{15}, b_{15})or(a_{14}, b_{14})or(a_{13}, b_{13})or(a_{12}, b_{12})or(a_{11}, b_{11})or(a_{10}, b_{10})or(a_9, b_9)$
$or(a_8, b_8)or(a_7, b_7)or(a_6, b_6)or(a_5, b_5)or(a_4, b_4)or(a_3, b_3)or(a_2, b_2)or(a_1, b_1)or(a_0, b_0)$ où $or(...)$ est l'opération OR appliquée à un bit définie plus haut.

Ces opérations logiques existent sous trois formes en fonction de leurs arguments :

- `OR reg1, reg2` : place dans le registre `reg1` le résultat de l'opération `OR` appliquée aux valeurs stockées dans les registres `reg1` et `reg2`
- `OR reg1, [adr]` : place dans le registre `reg1` le résultat de l'opération `OR` appliquée aux valeurs stockées dans le registre `reg1` et en mémoire à l'adresse `adr`
- `OR reg1, c` : place dans le registre `reg1` le résultat de l'opération `OR` appliquée aux valeurs stockées dans le registre `reg1` et la constante `c`

Ces instructions permettent d'utiliser ces opérations logiques sur des blocs de 16 bits. En pratique, elles s'avèrent aussi très utile lorsque l'on souhaite fixer des valeurs à certains bits en particulier. Considérons par exemple le bloc de 16 bits $b_{15}b_{14}b_{13}b_{12}b_{11}b_{10}b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0$ qui est actuellement stocké dans le registre `A`.

Si l'on souhaite forcer le bit de poids faible à la valeur `0` sans changer aucun des autres bits de ce bloc, il suffit d'exécuter l'instruction `AND A, 1111111111111110b`. Le lecteur attentif vérifiera aisément que $AND(b, 1)$ vaut $b$ et que $AND(b, 0)$ vaut toujours 0, quelle que soit la valeur du bit $b$.

L'instruction `OR` permet elle de forcer la valeur d'un bit à `1`. Ainsi `OR A, 1000000000000000b` forcera la valeur du bit de poids fort du registre `A` à `1` sans changer les valeurs des autres registres.

Le langage python supporte également les opérations booléennes bit à bit. Les principales sont listées ci-dessous :

- En python *AND(a,b)* s'écrit `a & b`
- En python *OR(a,b)* s'écrit `a | b`
- En python *NOT(a)* s'écrit `~ a`
- En python *XOR(a,b)* s'écrit `a ^ b`

## 3.5 Instructions de manipulation de bits

Notre processeur supporte également des opérations de décalage à gauche (SHL - SHift Left) et à droite (SHR - SHift Right). Ces instructions prennent deux arguments comme les opérations arithmétiques. En pratique, ces instructions sont généralement utilisées avec une constante comme second argument.

L'instruction `SHL reg, n` décale de n positions les bits se trouvant dans le registre `reg` vers la gauche. A titre d'exemple, si le registre `B` contient les bits $b_{15}b_{14}b_{13}b_{12}b_{11}b_{10}b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0$, alors après exécution de l'instruction `SHL B, 3` ce registre contiendra les bits $b_{12}b_{11}b_{10}b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0000$. De façon équivalente, si on exécute `SHR B, 5` sur les bits $b_{15}b_{14}b_{13}b_{12}b_{11}b_{10}b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0$, on obtient $00000b_{15}b_{14}b_{13}b_{12}b_{11}b_{10}b_9b_8b_7b_6b_5$.

Il est aussi possible de demander à python d'effectuer des décalages à gauche et à droite. Ainsi, `x << p` décale la représentation binaire de `x` de *p* positions vers la gauche. De la même façon, `y >> p` décale la représentation binaire de `y` de *p* positions vers la droite.

## 3.6 L'instruction de comparaison

Outre les instructions arithmétiques et logiques, notre processeur contient également une instruction de comparaison dénommée CMP. Cette instruction permet de comparer deux séquences de bits pour déterminer si elles sont égales. Elle compare la valeur se trouvant dans le registre qui est son premier argument avec son second argument qui peut être :

- un autre registre (`CMP reg1, reg2`)
- une valeur se trouvant à une adresse en mémoire (`CMP reg, [adr]`)
- une constante (`CMP reg, cst`)

Lors de son exécution, l'instruction de comparaison ne modifie pas la valeur contenue dans le registre qui est son premier argument. Elle stocke son résultat dans un drapeau (flag en anglais). Ce drapeau occupe 1 bit dans le processeur (le bit `Z`). Il est mis à la valeur *vrai* par l'instruction `CMP` si les valeurs des deux arguments sont identiques et à *faux* sinon. Dans l'exemple ci-dessous, le drapeau `Z` est mis à la valeur *faux* après exécution de la première instruction `CMP`. Ce drapeau passe à la valeur *vrai* après exécution de la seconde instruction `CMP`.

```
MOV A, 2
MOV B, 3
MOV C, 2
CMP A, B    ; Z est mis à faux
CMP A, C    ; Z est mis à vrai
```

L'instruction `CMP` n'est pas la seule à modifier le drapeau `Z`. C'est le cas pour toutes les instructions arithmétiques et logiques : `ADD`, `SUB`, `MUL`, `INC`, `DEC`, ... Après exécution de chacune de ces instructions, le drapeau `Z` est mis à *vrai* si le résultat de l'opération est le bloc de 16 bits dont tous les bits valent zéro. Lorsque l'on veut utiliser la valeur du drapeau `Z`, il faut le faire immédiatement après l'exécution de l'instruction `CMP`.

Notre processeur supporte un deuxième drapeau, Carry (report en anglais) ou C. Ce drapeau est utilisé par les opérations arithmétiques et logiques. Notre processeur stocke des données sur 16 bits dans chacun de ses registres. Lorsque l'on réalise une opération arithmétique, il est possible que le résultat nécessite plus de 16 bits pour stocker sa représentation binaire. C'est le cas par exemple pour les opérations d'addition ou de multiplication. Dans le programme ci-dessous, le drapeau `C` sera mis à *vrai* à la seconde instruction `INC` car le résultat (`65536`) doit être stocké sur 17 bits et non 16.

```
MOV A, 65534
INC A          ; C mis à faux
INC A          ; C mis à vrai
```

Il en va de même pour l'instruction de multiplication qui provoque également un dépassement de capacité (et donc fixe le drapeau `C` à *vrai* après son exécution) comme dans l'exemple ci-dessous.

```
MOV A, 40000
MUL A          ; dépassement de capacité C est mis à vrai
```

## 3.7 Le compteur de programme et les instructions de saut

Outre les registres `A`, `B`, `C` et `D`, un microprocesseur contient également un registre spécial généralement dénommé Compteur de Programme ou Program Counter (PC) en anglais. Certains documents parlent de pointeur d'instruction ou instruction pointer en anglais. Dans ce syllabus, nous utiliserons le terme PC pour parler de ce registre. Le PC stocke à tout moment l'adresse en mémoire de l'instruction à exécuter. Lors de l'exécution d'une instruction arithmétique, le PC est simplement incrémenté de façon à contenir l'adresse de l'instruction suivante. A titre d'exemple, considérons la suite d'instructions de la section précédente.

```
i1: MOV A, 2
i2: MOV B, 3
i3: MOV C, 2
i4: CMP A, B
i5: CMP A, C
```

Au démarrage de l'ordinateur, le PC est initialisé à l'adresse de la première instruction à exécuter (`0` dans notre processeur simple, mais ce n'est pas toujours le cas). Durant l'exécution de l'instruction `MOV A,2`, le PC contient l'adresse de l'étiquette `i1`. A la fin de l'exécution de cette instruction, le PC est modifié pour contenir l'adresse de l'instruction qui suit, c'est-à-dire `i2`. Cette mise à jour du compteur de programme s'effectue lors de l'exécution de toutes les instructions arithmétiques et logiques. Cela permet l'exécution séquentielle des instructions du programme.

Notre microprocesseur, comme tous les autres processeurs, supporte également des instructions qui permettent de modifier la valeur stockée dans le PC. Ce sont les instructions de saut. Il existe deux types d'instructions de saut :

- les instructions de saut *inconditionnelles* qui permettent de remplacer l'adresse stockée dans le PC par une autre adresse.
- les instructions de saut *conditionnelles* qui permettent de remplacer l'adresse stockée dans le PC par une autre adresse lorsqu'une *condition particulière est remplie*. Si la condition n'est pas remplie, l'adresse stockée dans le PC devient celle de l'instruction suivante.

L'instruction de saut inconditionnelle s'appelle JMP (pour *jump*, *saut* en anglais). Cette instruction prend un argument qui est une adresse (ou une étiquette). A titre d'illustration, considérons la suite d'instructions ci-dessous. A votre avis, l'instruction `CMP` va-t-elle mettre le drapeau `Z` à *vrai* ou *faux* ?

```
i1: MOV A, 2
i2: JMP i4
i3: MOV A, 9
i4: MOV B, 9
i5: CMP A, B
```

La première instruction place la valeur `2` dans le registre `A`. La deuxième instruction (étiquette `i2`) modifie le compteur de programme de façon à ce que l'adresse de l'instruction suivante soit celle de l'étiquette `i4`. Le processeur exécute donc ensuite l'instruction `MOV B,9` qui se trouve à l'adresse de l'étiquette `i4`. Ensuite il exécute l'instruction de comparaison qui place la valeur *faux* dans le drapeau `Z` puisque le registre `A` contient la valeur `2` et le registre `B` la valeur `9`.

L'instruction de saut inconditionnel a plusieurs utilisations comme nous le verrons bientôt. Pour rendre le code assembleur plus facile à lire, il est intéressant de définir les constantes au début du programme plutôt qu'à la fin. Comme le processeur commence par exécuter l'instruction se trouvant à l'adresse 0, nous ne pouvons pas commencer un programme par le mot clé `DB` pour définir une constante. Par contre, nous pouvons facilement associer une étiquette `start` au début « réel » de notre programme et avoir comme première instruction `JMP start`. Cette instruction peut être suivie d'une définition des différentes constantes utilisées par le programme avec une suite de mot-clés `DB` et les étiquettes associées.

A titre d'exemple, reprenons le programme python ci-dessous.

```
x=3
y=5
z=x+2*y
```

Ce programme peut être de façon plus lisible comme suit.

```
        JMP start
;  Variables et données du programme
x:      DB 3
y:      DB 5
z:      DB 0
start:  MOV A, [y]      ; place la valeur de la variable y dans A
        MUL 2           ; multiplie le contenu de A par 2
        ADD A, [x]      ; ajoute au contenu de A la valeur de la variable x
        MOV z, A        ; sauvegarde du résultat du calcul dans la variable z
        HLT
```

Nous pouvons maintenant aborder les instructions de saut conditionnelles. Ces instructions réalisent la modification du PC en fonction des valeurs des drapeaux `Z` et/ou `C`. Ces instructions prennent un seul argument : l'adresse qu'il faut placer dans le PC si la condition est remplie.

Les deux premières instructions conditionnelles sont JE (*Jump if Equal*) et JNE (*Jump if Not Equal*). Ces instructions s'utilisent après une instruction `CMP` et testent la valeur du drapeau `Z`. L'instruction `JE` modifie le PC si le drapeau `Z` contient la valeur *vrai*. L'instruction `JNE`, elle, modifie le PC lorsque le drapeau `Z` contient la valeur *faux*.

Imaginons que nous devions écrire un programme qui place la valeur `0` dans le registre `C` si les valeurs contenues dans les registres `A` et `B` sont égales et `1` sinon. Une première version de ce programme pourrait s'écrire comme suit.

```
        MOV A, 123
        MOV B, 123
        CMP A, B
        JE equal
        MOV C, 0
equal:  MOV C, 1
        HLT
```

Lors de son exécution, ce programme charge les deux valeurs dans les registres `A` et `B`. Ensuite, l'instruction `CMP` fixe la valeur du drapeau `Z`. Si ce drapeau est à la valeur *vrai*, l'instruction `JE` modifie le `PC` pour y mettre l'adresse correspondant à l'étiquette `equal` et l'instruction `MOV C,1` est exécutée. Par contre, si le drapeau est à la valeur *faux*, le processeur exécute l'instruction *MOV C,0* et place la valeur attendue dans le registre `C`. Malheureusement, l'instruction suivante est `MOV C, 1` et la valeur de ce registre est à nouveau modifiée.

On peut éviter ce problème en utilisant un saut inconditionnel après l'exécution de l'instruction `MOV C,0` comme ci-dessous.

```
        MOV A, 123
        MOV B, 123
        CMP A, B
        JE equal
ne:     MOV C, 0
        JMP suite:
equal:  MOV C, 1
suite:  HLT
```

Dans cette séquence d'instructions, le saut inconditionnel permet d'empêcher l'exécution de l'instruction se trouvant à l'adresse correspondant à l'étiquette `equal`. C'est une utilisation assez fréquente de l'instruction de saut inconditionnel comme nous le verrons bientôt.

Il existe une deuxième paire d'instructions de saut qui testent uniquement la valeur du drapeau `Z` : JZ (*Jump if Zero*) et JNZ (*Jump if Not Zero*). L'instruction JZ modifie le PC si le drapeau `Z` est à la valeur *vrai*. L'instruction JNZ réalise cette modification lorsque le drapeau `Z` contient la valeur *faux*. Ces instructions peuvent s'utiliser sans être précédées de l'instruction `CMP` comme dans l'exemple ci-dessous.

```
          MOV A, 1
          DEC A
          JZ zero
nz:       MOV C, 1
          JMP suite:
zero:     MOV C, 0
suite:    HLT
```

Les instructions conditionnelles permettent aussi de réaliser des comparaisons pour déterminer si la valeur stockée dans un registre est supérieure, inférieure, ou inférieure ou égale à celle d'un autre registre. Ces instructions s'utilisent directement après une opération `CMP reg1, reg2`. Les instructions suivantes sont supportées par notre assembleur :

- JA (*Jump Above*) : le saut conditionnel est effectué si la valeur stockée dans le premier registre argument de l'instruction `CMP` est *strictement supérieure* à la valeur stockée dans le second registre
- JB (*Jump Below*) : le saut conditionnel est effectué si la valeur stockée dans le premier registre argument de l'instruction `CMP` est *strictement inférieure* à la valeur stockée dans le second registre
- JAE (*Jump Above or Equal*) : le saut conditionnel est effectué si la valeur stockée dans le premier registre argument de l'instruction `CMP` est *supérieure ou égale* à la valeur stockée dans le second registre
- JBE (*Jump Below or Equal*) : le saut conditionnel est effectué si la valeur stockée dans le premier registre argument de l'instruction `CMP` est *inférieure ou égale* à la valeur stockée dans le second registre

A titre d'exemple, nous pouvons utiliser ces instructions conditionnelles pour implémenter un petit programme qui calcule la valeur absolue de la différence entre deux variables et place le résultat dans le registre `C`.

```
          JMP start
; déclarations des variables et constantes
x:        DB 12
y:        DB 9
start:    MOV A, [x]
          CMP A, [y]
          JBE petit
          MOV C, A
          SUB C, [y]
          JMP fin
petit:    MOV C, [y]
          SUB C, A
fin:      HLT
```

Enfin, il est aussi possible de vérifier simplement le drapeau `C` via les instructions JC (*Jump if Carry*) et JNC (*Jump if Not Carry*). Ces instructions sont utiles pour détecter un dépassement de capacité, notamment sur les opérations arithmétiques telles que la multiplication. En cas de dépassement, il est parfois préférable d'avertir l'utilisateur ou d'arrêter l'ordinateur plutôt que que continuer l'exécution du programme avec des données erronées dans un registre qui pourraient provoquer des erreurs en cascade. L'exemple ci-dessous montre comme utiliser l'instruction JNC pour vérifier si une instruction de multiplication a provoqué un dépassement de capacité.

```
          MOV A, 1000
          MOV B, 123
          MUL B
          JNC correct
; dépassement de capacité
          HLT
correct:  MOV B, D
```

(suite sur la page suivante)

(suite de la page précédente)

```
; suite du programme
```

Un autre problème mathématique qui peut survenir est lorsque l'on effectue une division par zéro. Contrairement à d'autres microprocesseurs, notre microprocesseur ne dispose pas de drapeau qui permet de détecter cette situation. Lors de l'exécution d'une instruction telle que `DIV 0`, le processeur s'arrête et affiche le message *Division by 0*. Si vous souhaitez éviter qu'un programme qui réalise une division ne s'arrête de cette façon, vous devez vous assurer de ne jamais avoir `0` comme diviseur dans un de vos quotients.

## 3.8 Les instructions conditionnelles

Les instructions de saut que nous venons de voir jouent un rôle critique dans les programmes écrits en assembleur. C'est grâce à ces instructions que l'on peut implémenter à la fois des instructions conditionnelles de type `if ... else`, mais aussi les boucles et même les appels à des fonctions et procédures comme nous le verrons plus tard.

En python, il est facile d'écrire une instruction conditionnelle. Il suffit d'utiliser le mot clé `if`, d'indiquer la condition et ensuite la séquence d'instructions à exécuter. Prenons comme exemple la recherche du maximum entre deux variables, `x` et `y`. En python, on peut affecter le maximum à la variable `max` comme suit :

```
if x>y :
   max=x
else:
   max=y
```

En assembleur, nous devons utiliser une instruction de saut conditionnelle pour obtenir le même résultat. Commençons pas déclarer nos trois variables : `x`, `y` et `max`. Ensuite nous devons comparer les valeurs des variables `x` et `y`. Pour cela, nous les plaçons dans les registres `A` et `B`. Si la valeur de la variable `x` est strictement supérieure à celle de la variable `y`, nous devons placer la valeur de `x` (qui est actuellement dans le registre `A`) dans la variable `max`. Sinon, nous plaçons la valeur de la variable `y` dans la zone mémoire correspondant à la variable `max`.

```
          JMP start
; déclarations et initialisations des variables
x:        DB 12
y:        DB 9
max:      DB 0
start:    MOV A, [x]
          MOV B, [y]
          CMP A, B
          JA xmax
          MOV max, B
          JMP fin
xmax:     MOV max, A
fin:      HLT
```

Une approche similaire peut être utilisée pour implémenter d'autres instructions conditionnelles. Le tout est de ramener toute condition à une comparaison avec la valeur *0* ou à une relation d'ordre.

Pour les conditions plus complexes, il faut parfois réécrire l'instruction conditionnelle. Prenons deux exemples en python pour illustrer cette réécriture.

```
if (a>0) AND (b<1):
   x=2
```

Dans ce cas, on peut réécrire l'instruction conditionnelle sous la forme :

```
if (a>0) :
  if (b<1) :
    x=2
```

Ces deux instructions conditionnelles imbriquées peuvent facilement s'implémenter avec les instructions de saut conditionnel que nous avons présentées. Il en va de même pour une disjonction logique. L'instruction ci-dessous :

```
if (a>0) OR (b<1):
   x=3
```

peut se réécrire de la façon suivante pour supprimer la disjonction logique.

```
if (a>0) :
  x=3
else :
  if (b<1) :
      x=2
```

Les lecteurs attentifs convertiront ces instructions conditionnelles en assembleur à titre d'exercice.

## 3.9 Les boucles

Après les opérations arithmétiques et logiques et les instructions conditionnelles, il nous reste à voir comment supporter les boucles. Python supporte deux types principaux de boucles :

— les boucles `while`
— les boucles `for`

Les boucles `while` sont les boucles les plus générales. Une boucle `for` est équivalente à une boucle d'un type particulier qui est écrite de façon compacte. Nous nous focaliserons sur les boucles `while` dans cette section. Une boucle `while` comprend toujours une condition qui comprend une expression booléenne qui est testée à chaque itération et un corps contenant une ou plusieurs instructions à exécuter.

Commençons par une boucle inutile, mais que vous avez probablement déjà rencontrée : la boucle infinie.

```
while True:
  x=x+1
```

En assembleur, cette boucle peut s'écrire en utilisant une instruction de saut inconditionnel `JMP`.

```
          JMP start
; variables et constantes
x:        DB 0
; programme
start:    MOV A, [x]
          INC A
          MOV x, A
          JMP start
```

Parfois, on écrit par inadvertance une boucle infinie en python car la condition d'arrêt de la boucle n'est jamais réalisée, même si python vérifie cette condition à chaque itération.

```
x=1
while x!=0:
  x=x+1
```

Ce programme python peut être traduit par les instructions suivantes en assembleur.

```
          JMP start
; variables et constantes
x:        DB 1
start:    MOV A, [x]
          CMP A, 0
          JZ fin
          INC A
          MOV x, A
          JMP start
fin:      HLT
```

Ce programme place la valeur de la variable `x` dans le registre `A` et regarde si elle est différente de zéro. Si cette valeur est égale à zéro, il sort de la boucle. Sinon, il incrémente la valeur du registre `A` puis sauve le résultat en mémoire à l'adresse de la variable `x`.

La sauvegarde en mémoire de la valeur de la variable `x` n'est pas nécessaire puisque cette valeur se trouve également dans le registre `A`. On peut réduire le nombre d'instructions à exécuter et donc accélérer le programme en mettant à jour la valeur de la variable `x` uniquement en fin de boucle comme présenté ci-dessous.

```
          JMP start
; variables et constantes
x:        DB 1
start:    MOV A, [x]
boucle:   CMP A, 0
          JZ fin
          INC A
          JMP boucle
fin:      MOV x, A
          HLT
```

Cette nouvelle version du programme incrémente la valeur du registre `A` à chaque itération mais ne sauvegarde la valeur du registre `A` en mémoire à l'adresse de la variable `x` qu'à la sortie de la boucle (étiquette `fin`). Ce programme est plus efficace que le précédent même si il aboutit au même résultat final.

Si vous exécutez le programme python, vous verrez qu'il ne s'arrête jamais et que vous devrez manuellement arrêter l'interpréteur python pour forcer la terminaison du programme. Si vous faites le même essai avec le programme en assembleur sur le simulateur, vous verrez que le programme en assembleur finit par s'arrêter. Cette différence de comportement s'explique par la façon dont les nombres naturels sont stockés en python et dans notre assembleur. Le langage python a été conçu de façon à pouvoir réaliser des calculs sans limitation avec tous les nombres entiers. Pour cela, le langage python adapte dynamiquement le nombre de bits utilisés pour représenter chaque nombre. En python, vous pouvez calculer avec les valeurs `1000`, `2000000`, `` `5000000000 `` ou `9000000000000` sans aucun souci.

Notre assembleur utilise 16 bits pour représenter les nombres naturels. Avec 16 bits qui peuvent prendre les valeurs `0` et `1`, il est possible de représenter $2^{16}$ nombres naturels différents. Le plus petit est `0` (ou `0b00000000 00000000`) et le plus grand `65535` ($2^{16}-1$ ou `0b11111111 11111111`). Analysons ce qu'il se passe dans la boucle. Au début, la valeur dans le registre `A` passe de `0b00000000 00000000` à puis `0b000000000 0000001`, `0b00000000 00000010`, ... Après quelque temps, le registre `A` contient la valeur `65534` ou `0b11111111 11111110`. Après incrémentation, cette valeur passe à `0b11111111 11111111`. C'est le plus grand naturel que l'on peut représenter en utilisant 16 bits. L'incrémentation suivante devrait faire passer la valeur du registre à `0b1 00000000 00000000` ou `65536`. Comme le registre `A` ne peut stocker que 16 bits, il conserve les 16 bits de poids faible, à savoir `0b00000000 0000000` ou `0` en notation décimale. Après l'exécution de cette instruction, le drapeau `C` du processeur est mis à *vrai* pour indiquer qu'il y a eu un dépassement de capacité lors de l'exécution de l'instruction `INC`, mais notre programme ne vérifie pas ce drapeau... Le nouvelle valeur stockée dans le registre `A` est numériquement égale à `0` et notre programme sauvegarde la valeur `0` en mémoire puis s'arrête.

Nous pouvons nous inspirer de cette approche pour traduire une boucle `while` en une séquence d'instructions en assembleur. Pour cela, notre programme doit :

1. Évaluer la valeur de la condition
2. Si la condition est évaluée à la valeur *vrai*, exécuter le corps de la boucle puis revenir au point 1
3. Sinon, passer à l'exécution des instructions placées juste après le corps de la boucle

Pour illustrer cette traduction, considérons la boucle ci-dessous. Après l'exécution de cette boucle, la variable `x` contient la valeur *512*.

```
x=1
n=1
while (n<10) :
   n=n+1
   x=x+x
```

Ce fragment de code peut se traduire en langage assembleur. Il faut d'abord charger la valeur de la variable `n` (ligne 5) et la comparer à `10`. Si la valeur de la variable `n` est supérieure ou égale à `10`, il faut sortir de la boucle. En général, pour implémenter une condition en assembleur, utilise l'instruction de saut qui correspond à la condition inverse puisque l'on cherche à faire un saut pour sortir de la boucle si la condition n'est pas vérifiée. Ensuite, il suffit d'incrémenter la valeur de la variable `n` puis de la sauvegarder en mémoire. On peut ensuite charger la variable `x` dans un registre et calculer `x+x`. Enfin, on utilise une instruction de saut inconditionnel `JMP` pour revenir au début de la boucle et réévaluer la condition `n<10`.

```
          JMP boucle
; variables
x:        DB 1
n:        DB 1
boucle:   MOV A, [n]
          CMP A, 10
          JAE fin
          INC A
          MOV n, A
          MOV B, [x]
          ADD B, B
          MOV x, B
          JMP boucle
fin:      HLT
```

Tout comme dans l'exemple précédent, on aurait pu réduire le nombre de transferts vers la mémoire et d'instructions en plaçant les variables `x` et `n` dans les registres `B` et `A` au début de la boucle, avant l'évaluation de la condition d'arrêt. Si on procède de cette façon, il ne faut bien entendu pas oublier de sauver les valeurs stockées dans les registres en mémoire en sortie de boucle.

```
          JMP boucle
; variables
x:        DB 1
n:        DB 1
          MOV A, [n]
          MOV B, [x]
boucle:   CMP A, 10
          JAE finb
          INC A
          ADD B, B
          JMP boucle
finb:     MOV n, A   ; sauvegarde variable n
          MOV x, B   ; sauvegarde variable x
fin:      HLT
```

En python, il existe différentes formes de boucles for. Nous nous limiterons aux boucles qui itèrent sur des naturels comme `for i in range (0,4):` ou `for x in range (1, 5, 2):`. Ces boucles peuvent facilement se

traduire sous la forme d'une boucle while en python. Ainsi, les deux boucles ci-dessous sont équivalentes.

```
for x in range(2,7):
    print(x)
```

```
x=2
while (x<=7):
  print(x)
  x=x+1
```

De la même façon, les deux boucles ci-dessous sont également équivalentes.

```
for x in range (10, 5, -2):
  print(x)
```

```
x=10
while (x>=5):
  print(x)
  x=x-2
```

Chacune de ces boucles while peut être facilement convertie en assembleur en utilisant notamment des instructions de saut.

CHAPITRE 4

# Utilisation des tableaux

Jusque maintenant, nous avons manipulé des variables entières qui sont stockées en mémoire ou dans des registres. Un ordinateur doit également pouvoir traiter des objets mathématiques tels que les vecteurs et les matrices. Ceux-ci doivent aussi pouvoir être stockés en mémoire.

Commençons par analyser la façon dont un programme peut manipuler les coordonnées *(x,y)* d'un point sur un plan. Ces coordonnées sont toutes les deux représentées sous la forme d'un naturel. Une première approche serait d'associer une variable pour l'abscisse et une autre pour l'ordonnée. C'est ce que nous faisons dans l'exemple ci-dessous avec les variables `CAx` et `CAy` pour les coordonnées du point `A`. Le programme doit vérifier si les coordonnées de deux points sont égales. Pour cela, il charge simplement les coordonnées `x` puis `y` des deux points à comparer et met la variable `eq` à `1` si les deux points sont égaux et `0` sinon.

```
        JMP start
; mis à 1 si égales, 0 sinon
eq:     DB 0
; premier point
CAx:    DB 1 ; coordonnée x
CAy:    DB 2 ; coordonnée y
; second point
CBx:    DB 1 ; coordonnée x
CBy:    DB 7 ; coordonnée y
start:  MOV A, [CAx]
        MOV B, [CBx]
        CMP A, B
        JNE diff
        MOV A, [CAy]
        MOV B, [CBy]
        CMP A, B
        JNE diff
egal:
        MOV eq, 1
        JMP fin
diff:
        MOV eq, 0
fin:    HLT
```

Malheureusement cette solution nous force à définir un très grand nombre de variables. Si on analyse comment l'assembleur place les données en mémoire, on se rend compte que les variables `CAx` et `CAy` occupent des positions consécutives en mémoire. Il en va de même pour les variables `CBx` et `CBy`. Ainsi, la mémoire initialisée par le programme ci-dessus peut se visualiser comme dans la table Tableau 4.1 où l'adresse `03` est utilisée par la variable `eq`.

TABLEAU 4.1 – Contenu de la mémoire

| adresse | valeur |
|---|---|
| 03 | 0 |
| 05 | 1 |
| 07 | 2 |
| 09 | 1 |
| 0B | 7 |

On peut profiter de cette organisation de la mémoire et déclarer nos variables en utilisant une étiquette pour chaque paire de deux entiers qui représente une coordonnée.

```
; mis à 1 si égales, 0 sinon
eq: DB 0
; premier point
CA: DB 1 ; coordonnée x
    DB 2 ; coordonnée y
; second point
CB: DB 1 ; coordonnée x
    DB 7 ; coordonnée y
```

Ces déclarations définissent deux variables : `CA` et `CB` qui utilisent chacune deux blocs consécutifs de 16 bits en mémoire. Avec ces étiquettes, nous pouvons adapter notre programme de façon à ce qu'il puisse tester l'égalité des coordonnées `x` et `y` de chaque point. Pour les coordonnées `x`, c'est facile. Il suffit de réutiliser les mêmes instructions que dans le programme précédent en adaptant le nom des variables.

```
start: MOV A, [CA]
MOV B, [CB]
CMP A, B
JNE diff
```

Pour comparer les coordonnées `y`, cette approche ne fonctionne plus car nous n'avons pas défini d'étiquette correspondant à l'adresse de la coordonnée `y` du point `CA` en mémoire. Par contre, nous savons que cette coordonnée se trouve à l'adresse qui suit celle de la coordonnée `x`. Si la coordonnée `x` se trouve à l'adresse `Adr` en mémoire, alors la coordonnée `y` se trouve à l'adresse `Adr+2` puisque sur notre processeur un entier occupe 16 bits. On voudrait pouvoir écrire les instructions suivantes :

```
MOV A, [CA+2]
MOV B, [CB+2]
CMP A, B
JNE diff
```

Dans ce programme, `CA` correspond à une adresse en mémoire et `CA+2` serait l'adresse de l'entier 16 bits qui suit celui qui se trouve à l'adresse `CA` en mémoire. Malheureusement, notre processeur ne nous permet pas de calculer une adresse de cette façon dans l'instruction `MOV`. Il permet de réaliser ce genre de calcul simple (addition ou soustraction) avec une adresse, mais uniquement si celle-ci se trouve dans un registre. On doit donc d'abord placer l'adresse `CA` dans un registre (par exemple le registre `C` avec l'instruction `MOV C, CA`). Une fois cette opération réalisée, on peut utiliser l'adresse se trouvant dans le registre `C`. Ainsi, l'instruction `MOV A, [C]` placera dans le registre `A` le bloc de 16 bits qui se trouve en mémoire à l'adresse qui se trouve actuellement dans le registre `C`. L'instruction `MOV B, [C+2]` placera dans le registre `B` le bloc de 16 bits qui se trouve actuellement en mémoire à l'adresse qui *suit*

l'adresse qui est dans le registre `C`. Enfin, l'instruction `MOV D, [C-2]` placera dans le registre `D` le bloc de 16 bits qui se trouve en mémoire à l'adresse qui *précède* celle qui est dans le registre `C`.

Nous pouvons donc écrire les instructions suivantes pour comparer les coordonnées `y`

```
MOV C, [CA]
MOV A, [C+2]
MOV D, [CB]
MOV B, [D+2]
CMP A, B
JNE diff
```

---

**Note :** Notez bien la différence entre les instructions
- `MOV A, Adr`
- `MOV A, [Adr]`

La première place dans le registre `A` l'adresse qui est son second argument. La seconde place dans le registre `A` la valeur qui est actuellement stockée en mémoire à l'adresse `Adr`.

---

Nous pouvons maintenant écrire le programme complet pour comparer les coordonnées `x` et `y` de nos deux points.

```
        JMP start
; mis à 1 si égales, 0 sinon
eq:     DB 0
; coordonnées premier point
CA:     DB 1
        DB 2
; coordonnées second point
CB:     DB 1
        DB 2
start:
        MOV A, CA
        MOV B, CB
        MOV C, [A]
        MOV D, [B]
        CMP C, D
        JNE diff
        MOV C, [A+2]
        MOV D, [B+2]
        CMP C, D
        JNE diff
egal:
        MOV eq, 1
        JMP fin
diff:
        MOV eq, 0
fin:
        HLT
```

Cette solution peut être étendue pour stocker des vecteurs ou des tableaux d'entiers dont la taille est connue. Pour stocker des coordonnées *(x,y,z)*, il nous suffit de réserver trois mots contigus en mémoire. De la même façon, si l'on doit stocker le nombre de jours dans chaque mois de l'année civile, il suffit de réserver un bloc de 12 mots consécutifs en mémoire et d'y stocker les valeurs reprises dans la Tableau 4.2.

TABLEAU 4.2 – Tableau contenant le nombre de jours dans chaque mois de l'année

| adresse | valeur |
|---|---|
| m+0 | 31 |
| m+2 | 28 |
| m+4 | 31 |
| m+6 | 30 |
| m+8 | 31 |
| m+10 | 30 |
| m+12 | 31 |
| m+14 | 31 |
| m+16 | 30 |
| m+18 | 31 |
| m+20 | 30 |
| m+22 | 31 |

Grâce à ce tableau, on peut facilement calculer le nombre de jours dans une année civile. En python, ce programme aurait pu être écrit de la façon suivante.

```
tableau=[31,28,31,30,31,30,31,31,30,31,30,31]

somme=0
i=0
while(i<12):
    somme = somme + tableau[i]
    i=i+1
```

En assembleur c'est un tout petit peu plus compliqué, mais il suffit de bien faire attention aux instructions que l'on écrit et d'être systématique. Notre programme assembleur va parcourir le tableau du nombre de jours dans chaque mois. Pour cela, nous aurons besoin de conserver l'indice du tableau `mois` qui est en cours de traitement. Nous choisissons d'utiliser le registre `C` pour stocker cette information. Il est initialisé à `0` avant d'entrer dans la boucle. Connaissant cet indice, il est possible de calculer l'adresse du *Cème* élément du tableau `` `mois ``. Pour cela, il suffit de calculer la somme entre l'adresse du tableau et le double de l'indice `C` puisque chaque entier prend 16 bits et donc deux adresses en mémoire. Nous choisissons d'utiliser le registre `A` pour stocker cette adresse car c'est le seul registre qui supporte l'opération de multiplication. Nous aurions pu aussi prendre le registre `D` et remplacer l'instruction `MUL 2` par `ADD D, D` qui donne le même résultat et en pratique est généralement plus rapide.

```
1        JMP start
2  ; variables
3  jours: DB 0
4  mois:  DB 31
5         DB 28
6         DB 31
7         DB 30
8         DB 31
9         DB 30
10        DB 31
11        DB 31
12        DB 30
13        DB 31
14        DB 30
15        DB 31
```

(suite sur la page suivante)

(suite de la page précédente)

```
start:
        MOV C, 0  ; index dans le tableau
boucle:
        MOV A, C
        MUL 2
        ADD A, mois ; adresse en mémoire du Ceme mois
        MOV B, [A]
        ADD B, [jours]
        MOV jours, B
        INC C
        CMP C,11
        JA fin
        JMP boucle
fin:
        HLT
```

De façon générale, si un tableau de naturels démarre à l'adresse *A*, alors le *ième* élément de ce tableau se trouve en mémoire à l'adresse $A + 2 * i$. Cette organisation peut également être utilisée pour stocker des matrices en mémoire. Il suffit simplement de définir une relation entre les indices d'un élément de la matrice et la zone mémoire correspondante. Les deux principales méthodes pour stocker une matrice en mémoire sont *ligne par ligne* et *colonne par colonne*.

Pour illustrer ces deux conventions, considérons la matrice à deux lignes et trois colonnes de la Fig. 4.1. La façon

$$\begin{matrix} 7 & 8 & 9 \\ 4 & 5 & 6 \end{matrix}$$

FIG. 4.1 – Une matrice entière composée de deux lignes et trois colonnes

la plus classique pour stocker une telle matrice est de le faire *ligne par ligne* comme représenté dans la Fig. 4.2. Dans cette représentation, si la matrice a *l* lignes et *c* colonnes, alors l'élément *i,j* de la matrice se trouve à l'adresse $A + i \times c + j$ en supposant que les indices des lignes et des colonnes commencent à *0*. Il est aussi possible de stocker

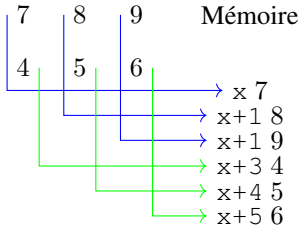



FIG. 4.2 – Stockage ligne par ligne d'une matrice

cette matrice colonne par colonne comme représenté dans la Fig. 4.3. Dans cette représentation, si la matrice a *l* lignes et *c* colonnes, alors l'élément *i,j* de la matrice se trouve à l'adresse $A + j \times l + i$ en supposant que les indices des lignes et des colonnes commencent à *0*. On est parfois amené à manipuler des tableaux de différentes tailles. Dans ce cas, il est intéressant de réserver un mot en mémoire pour stocker la taille du tableau. Tout tableau utilisant cette représentation contient donc comme premier élément sa taille. Un tableau de *n* entiers occupe donc $n + 1$ mots en mémoire.

A titre d'exemple, reprenons notre tableau avec le nombre de jours dans chaque mois. La représentation de notre tableau contient donc une entrée supplémentaire qui est sa taille (Tableau 4.3).

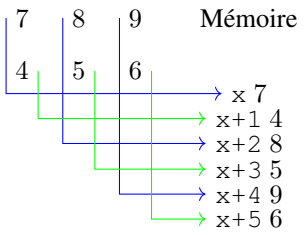



FIG. 4.3 – Stockage colonne par colonne d'une matrice

TABLEAU 4.3 – Tableau dont le premier élément est sa taille

| adresse | valeur |
|---|---|
| m | 12 |
| m+2 | 31 |
| m+4 | 28 |
| m+6 | 31 |
| m+8 | 30 |
| m+10 | 31 |
| m+12 | 30 |
| m+14 | 31 |
| m+16 | 31 |
| m+18 | 30 |
| m+20 | 31 |
| m+22 | 30 |
| m+24 | 31 |

Cette représentation a deux avantages principaux. Tout d'abord, il est possible d'écrire un programme générique qui peut parcourir tous les éléments de n'importe quel tableau comme dans l'exemple ci-dessous.

```
        JMP start
jours: DB 0
mois:
        DB 12   ; nombre d'éléments dans le tableau
        DB 31
        DB 28
        DB 31
        DB 30
        DB 31
        DB 30
        DB 31
        DB 31
        DB 30
        DB 31
        DB 30
        DB 31
start:
        MOV C, 1  ; index dans le tableau
boucle:
        MOV A, C
        MUL 2
        ADD A, mois ; adresse en mémoire du Ceme mois
        MOV B, [A]
        ADD B, [jours]
```

(suite sur la page suivante)

(suite de la page précédente)

```
        MOV jours, B
        INC C
        MOV D,[mois]
        CMP C,D
        JA fin
        JMP boucle
fin:
        HLT
```

De plus, lorsque cette représentation est utilisée dans un langage de programmation, celui-ci peut facilement vérifier que les accès aux éléments d'un tableau respectent bien les limites de ce tableau. C'est le cas avec le langage python.

---

**Note :** Convention de représentation des tableaux

Dans le cadre de ce syllabus, nous prendrons comme convention de représenter un tableau de $n$ entiers comme $n$ entiers consécutifs en mémoire mais sans indication explicite de la longueur dans le tableau lui-même. C'est la convention utilisée notamment par le langage C. Elle a l'avantage d'éviter de devoir toujours réserver un espace mémoire pour stocker la longueur du tableau alors que dans certains cas celle longueur est connue par le programme. C'était le cas notamment dans nos exemples avec les coordonnées. Dans les autres cas, la taille du tableau devra être stockée dans une autre variable qui sera elle aussi accessible au programme.

---

## 4.1 Les chaînes de caractères

Un programme informatique doit régulièrement utiliser des chaînes de caractères pour afficher des messages à l'utilisateur ou imprimer de l'information. Nous avons déjà vu comment représenter chaque caractère grâce à une table des caractères. Notre minuscule assembleur utilise un mot de 16 bits pour représenter chaque caractère. Une chaîne de caractères peut être vue comme un tableau de caractères. Elle sera composée de caractères consécutifs qui sont stockés en mémoire. En assembleur, nous pouvons stocker une chaîne de caractères en mémoire en utilisant directement le mot clé `DB` comme suit.

```
hello: DB "Hello World!"
```

Un programme devoir traiter des chaînes de caractères de tailles très différentes. Il existe deux techniques pour stocker ces chaînes de caractères en mémoire.

La première est de stocker la longueur de la chaîne suivie par les caractères qui la composent (Fig. 4.4). Cette solution permet de facilement déterminer la longueur de la chaîne de caractères puisque celle-ci est explicitement stockée en mémoire. En utilisant un mot de 16 bits pour cette longueur, on peut supporter des chaînes contenant au maximum 65535 caractères. C'est largement assez pour le minuscule ordinateur vu l'espace de mémoire dont il dispose. Afin

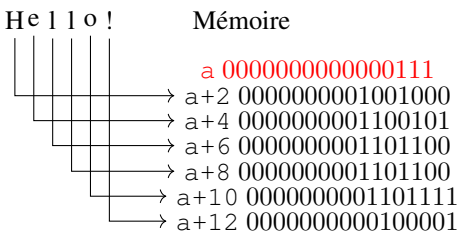



FIG. 4.4 – Représentation en mémoire de la chaîne de caractères Hello ! avec une indication explicite de longueur

d'illustrer l'utilisation de cette représentation des chaînes des caractères, considérons un petit programme qui permet

de déterminer si un caractère est présent dans une chaîne de caractères. En python, ce programme pourrait s'écrire comme suit :

```
mystr="Hello!"
c='h'
r=0 # absent
i=0
while(i<len(mystr)):
    if mystr[i]==c:
        r=1 # present
        break
    i=i+1
```

La conversion de ce programme en minuscule assembleur est présentée ci-dessous.

Notre programme a comme entrée la variable `char` contenant le caractère à rechercher et une chaîne de caractères qui est stockée en mémoire à partir de l'adresse correspondant à l'étiquette `string`. Le résultat du programme se retrouve dans la variable ``r`` en mémoire.

```
        JMP start
; Compte le nombre d'occurrences du caractère char dans la chaine string
count:  DB 0
char:   DB "o"
string:
        DB 12 ; longueur de la chaîne
        DB "Hello World!" ; Chaîne de caractères
start:
        MOV C, [char]   ; caractère à rechercher
        MOV D, 1        ; index de la position dans la chaîne
boucle:
        MOV A, D
        MUL 2
        ADD A, string
        CMP C, [A]
        JNE suite
        MOV A, [count]
        INC A
        MOV count, A
suite:
        INC D            ; incrément indice boucle
        MOV B, [string]    ; longueur de la chaîne
        CMP D, B
        JBE boucle
fin:
        HLT
```

Il existe une seconde façon de stocker les chaînes de caractères. C'est celle qui est utilisée notamment par le langage C. Ce langage utilise un caractère spécial (la valeur binaire *00000000 00000000* sur le minuscule ordinateur) pour marquer la fin de la chaîne de caractère. En assembleur, une telle chaîne de caractères peut être déclarée comme suit.

```
hello: DB "Hello!"
       DB 0  ; Fin de chaîne
```

Avec cette représentation des chaînes de caractères, le programme ne connaît pas a priori la longueur de la chaîne de caractères. Il doit la parcourir pour trouver le marqueur de fin symbolisé par la valeur `0` (et non le caractère ASCII `0`). En python, le parcours de cette chaîne peut se faire en utilisant le programme ci-dessous.

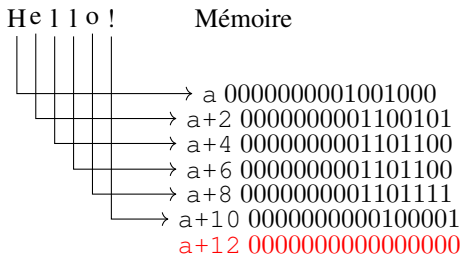



FIG. 4.5 – Représentation en mémoire de la chaîne de caractères Hello ! avec marqueur de fin

```
mystr="Hello!\0"
c='o'
r=0 # absent
i=0
while(mystr[i]!='\0'):
    if mystr[i]==c:
        r=1 # present
        break
    i=i+1
```

En assembleur, ce programme peut s'écrire comme suit.

```
        JMP start
; Compte le nombre d'occurrences du caractère char dans la chaine string
count:  DB 0
char:   DB "o"
string:
        DB "Hello World!" ; Chaîne de caractères
        DB 0              ; Marqueur de fin de chaîne
start:
        MOV C, [char]    ; caractère à rechercher
        MOV D, 0         ; index de la position dans la chaîne
boucle:
        MOV A,D
        MUL 2
        ADD A, string
        CMP C, [A]
        JNE suite
        MOV B, [count]
        INC B
        MOV count, B
suite:
        INC D             ; incrément indice boucle
        MOV B, 0
        CMP B, [A]          ; fin de chaîne ?
        JNE boucle
fin:
        HLT
```

**Note :** Convention de représentation des chaînes de caractères

Dans le cadre de ce syllabus, nous prendrons comme convention de toujours représenter les chaînes de caractères avec

la valeur `0x00 00` comme marqueur de fin de chaîne de caractères. C'est la convention qui est utilisée notamment par le langage C.

CHAPITRE 5

# Procédures et fonctions en assembleur

Notre processeur peut être programmé en langage d'assemblage grâce aux multiples instructions qu'il supporte. En théorie, ce langage d'assemblage est suffisant pour écrire n'importe quel programme. Cependant, il est fastidieux et dangereux d'écrire un programme sans le découper en fonctions et procédures qui peuvent être testées indépendamment et qui sont ensuite combinées.

Vous avez utilisé des fonctions et procédures dans le langage python sans analyser en détails comment ce langage supportait ces différentes constructions. Nous allons maintenant les analyser en nous appuyant sur le langage d'assemblage de notre processeur.

Commençons par réfléchir aux différentes opérations qu'un langage de programmation tel que python doit réaliser pour supporter différents types de fonctions. Tout d'abord, analysons comment implémenter une procédure, c'est-à-dire une fonction qui ne prend pas d'argument et ne retourne pas de résultat. Notre premier exemple simple est une procédure qui affiche de l'information à l'écran. Une telle procédure pourrait être utilisée dans un programme pour afficher le contenu d'un menu à l'écran.

```
def p():
    print("Bonjour")
```

Il est intéressant d'analyser comment un langage tel que python fait appel à une telle procédure dans un programme.

Regardons plus en détails le code ci-dessous. La première ligne initialise la variable `x` à la valeur `1`. La deuxième ligne transfère l'exécution du programme à la procédure `p()`. Le code de cette procédure est composé d'un ensemble d'instructions qui se trouvent en mémoire et vont afficher `Bonjour` à l'écran. Après l'exécution de cette procédure, le programme python retourne à l'exécution de la troisième ligne et place la valeur `2` dans la variable `x`. La quatrième ligne relance l'exécution de la procédure `p()`. Celle-ci va à nouveau exécuter les instructions qui permettent d'afficher `Bonjour` à l'écran, mais après son exécution le programme python exécutera la ligne `5`. On remarque une différence importante entre les deux invocations de la procédure `p()`. Après la première invocation, on exécute la ligne `3` du programme python. Après la deuxième invocation, on exécute la ligne `5` du programme python.

```
x=1             # ligne 1
p()             # ligne 2
x=2             # ligne 3
p()             # ligne 4
x=3             # ligne 5
```

En python, il est facile d'imprimer de l'information à l'écran. En assembleur, cette opération nécessite nettement plus d'efforts. Analysons un autre exemple en python qui utilise les variables globales. En python, une fonction utilise normalement les arguments qu'elle a reçu ou définit ses propres variables locales. Il est aussi possible de définir des variables globales, c'est-à-dire des variables qui sont stockées dans la mémoire du programme et sont accessibles à toutes les fonctions de ce programme. Cette utilisation d'une variable globale est illustrée dans le programme python ci-dessous.

```
compteur=0

def compte():
    global compteur
    compteur = compteur+1

x=1           # ligne 1
compte()      # ligne 2
x=2           # ligne 3
compte()      # ligne 4
x=3           # ligne 5
```

La variable `compteur` est une variable globale (python impose l'utilisation du mot clé `global` dans sa définition dans la procédure `compte`) qui est initialisée dans le programme principal et modifiée dans la procédure `compte`. Analysons l'exécution de ce programme pas à pas. Ce programme manipule deux variables en mémoire : `x` et `compteur`. La première ligne initialise la variable `x` à la valeur `1`. La deuxième ligne incrémente la variable `compteur` qui passe à `1`. La troisième ligne fait passer la valeur de la variable `x` à `2`. La quatrième ligne incrémente la variable `compteur` qui passe également à `2`. Enfin, la dernière ligne place la valeur `3` dans la variable `x`.

Pour bien comprendre comment une telle procédure peut être utilisée à plusieurs endroits dans un même programme, il est intéressant d'essayer de la convertir en minuscule assembleur.

Commençons par assigner une zone mémoire pour la variable `x`. Nous pouvons ensuite écrire en assembleur les lignes impaires qui correspondent aux différentes assignations de cette variable.

```
          JMP start
x:        DB 0
compteur: DB 0
start:
          MOV x, 1  ; ligne 1
          ; à compléter
          MOV x, 2  ; ligne 3
          ; à compléter
          MOV x, 3  ; ligne 5
```

Nous devons également assigner une zone mémoire pour stocker la variable `compteur`. Supposons que celle-ci soit stockée aux adresses `16` et `17`. La Fig. 5.1 présente le contenu initial de notre mémoire. Il nous faut maintenant

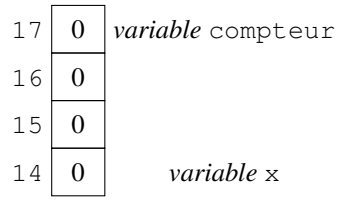



FIG. 5.1 – Contenu initial de la mémoire

pouvoir faire appel à la procédure `compte()` après l'exécution des lignes `1` et `3`. Le corps de cette procédure peut s'écrire en trois instructions en assembleur.

```
MOV A, [compteur]
INC A
MOV compteur, A
```

L'exemple ci-dessus utilise le registre `A`, mais il aurait pu aussi être écrit avec n'importe lequel des trois autres registres.

Une première approche pour inclure notre procédure dans le programme en minuscule assembleur est d'intégrer directement ces instructions *en ligne* (inline en anglais). Cette technique est parfois utilisée dans certains langages de programmation pour de très petites fonctions qui doivent s'exécuter rapidement. Elle revient à copier-coller le code de la procédure dans le programme.

```
          JMP start
x:        DB 0
compteur: DB 0
start:
          ; ligne 1
          MOV x, 1
          ; copie du code de la procédure
          MOV A, [compteur]
          INC A
          MOV compteur, A
          ; ligne 3
          MOV x, 2
          ; copie du code de la procédure
          MOV A, [compteur]
          INC A
          MOV compteur, A
          ; ligne 5
          MOV x, 3
```

Cette approche fonctionne dans notre exemple simple, mais elle a deux inconvénients majeurs. Le premier est que le code de la procédure doit être recopié à chaque invocation de la procédure dans un programme. Cela consomme inutilement de l'espace mémoire surtout si le programme appelle la procédure à de nombreux endroits. Le deuxième inconvénient est que si la procédure modifie le contenu d'un registre, elle pourrait avoir un impact non-voulu sur les instructions du programme principal. Dans notre cas, si nous utilisions le registre `A` dans le code qui se trouve entre les appels à la procédure `p`, cette valeur serait écrasée par le code de la procédure et notre programme serait en erreur.

Il est nécessaire de pouvoir isoler les instructions de la procédure dans une partie de la mémoire et d'y faire appel en exécutant un saut inconditionnel. Une première approche pourrait être la suivante. La code de la procédure `compte` est placé après l'étiquette `COMPTE` et on fait appel à la procédure en utilisant un saut inconditionnel vers cette adresse.

```
          JMP start
x:        DB 0
compteur: DB 0
start:
          ; ligne 1
          JMP COMPTE
ligne3:   ; ligne 3
          MOV x, 2
          JMP COMPTE
ligne5:   ; ligne 5
          MOV x, 3

COMPTE:
          MOV A, [compteur]
```

(suite sur la page suivante)

(suite de la page précédente)

```
        INC A
        MOV compteur, A
        JMP retour
```

Malheureusement, ce n'est pas suffisant. Après la première exécution de la procédure `compte`, l'exécution doit reprendre à l'adresse `ligne3` tandis qu'après la seconde exécution de la même procédure, il faut poursuivre l'exécution du programme principal à partir de l'adresse `ligne5`. Pour résoudre ce problème, nous devons rendre le code de la procédure plus générique. Notre procédure doit pouvoir retourner à l'adresse qui suit celle à partir de laquelle elle a été appelée. Dans notre assembleur, comme dans la plupart des assembleurs, cela se fait en utilisant deux instructions spéciales : CALL pour appeler une procédure et RET pour terminer l'exécution d'une procédure **et** retourner à l'adresse qui suit celle de l'appel. Cette adresse est appelée l"adresse de retour. L'instruction CALL la sauvegarde en mémoire et ensuite fait un saut à l'adresse qui est son unique argument. L'exécution d'une procédure se déroule comme suit :

1. Sauvegarde de l'adresse de retour en mémoire
2. Appel de la procédure (via l'instruction `JMP`)
3. Exécution du corps de la procédure
4. Récupération de l'adresse de retour
5. Saut à l'adresse de retour pour poursuivre l'exécution du programme appelant

Les deux premières opérations sont exécutées par l'instruction CALL. Les deux dernières sont exécutées par l'instruction RET.

Il est intéressant de voir comment le simulateur exécute l'instruction `CALL`. Pour cela, considérons le minuscule programme ci-dessous :

```
JMP start
fonction:
    RET
start:
    CALL fonction
```

La Fig. 5.2 présente l'état de la mémoire avant l'exécution de l'instruction `CALL`. La Fig. 5.3 montre que la pile est vide à ce moment. Le registre `SP` contient l'adresse `0x39C` comme sommet de pile/

L'instruction `CALL` se trouve à l'adresse `0x06`. En mémoire, cette instruction occupe deux blocs de 16 bits. L'instruction suivante se trouve donc à l'adresse `0x0A`. La Fig. 5.4 présente l'état de la mémoire après l'exécution de l'instruction `CALL`. On remarque que le sommet de la pile se trouve maintenant à l'adresse `0x39A`. La Fig. 5.5 montre que la pile contient bien l'adresse de retour (`0x0A`).

Grâce à l'instruction `CALL`, notre programme devient donc :

```
            JMP start
x:          DB 0
compteur:   DB 0
; ligne 1
start:
            CALL COMPTE
ligne3:                 ; ligne 3
            MOV x, 2
            CALL COMPTE
ligne5:                 ; ligne 5
            MOV x, 3

COMPTE:
            MOV A, [compteur]
            INC A
```

(suite sur la page suivante)

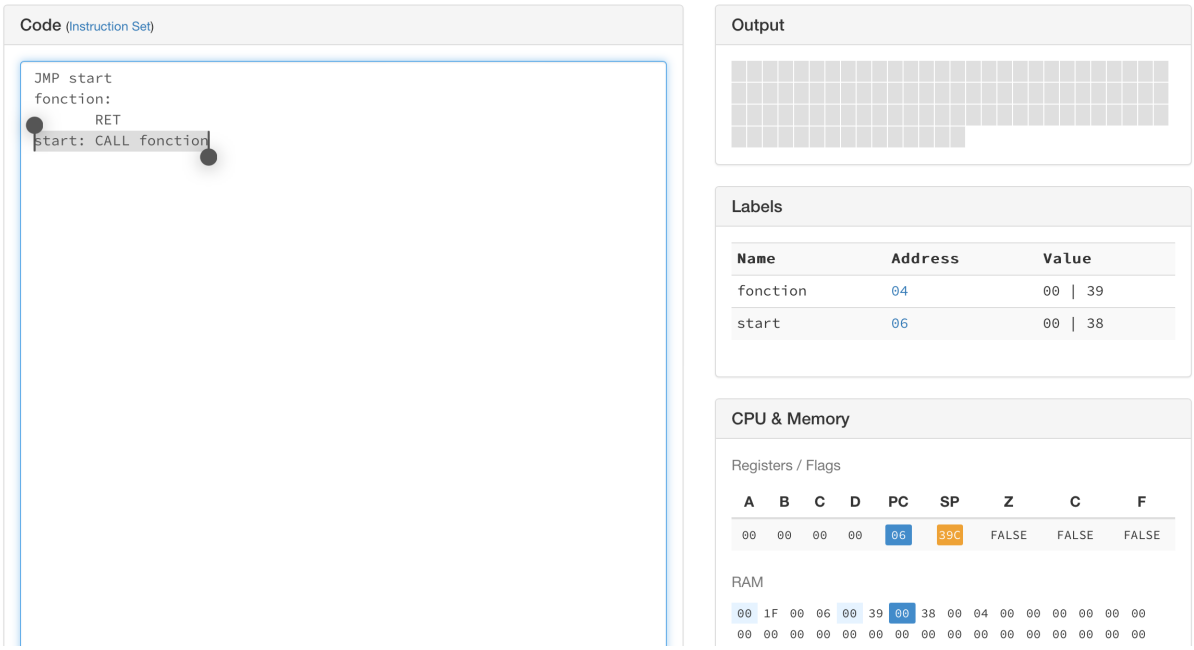



FIG. 5.2 – Etat de la mémoire avant l'exécution de l'instruction CALL

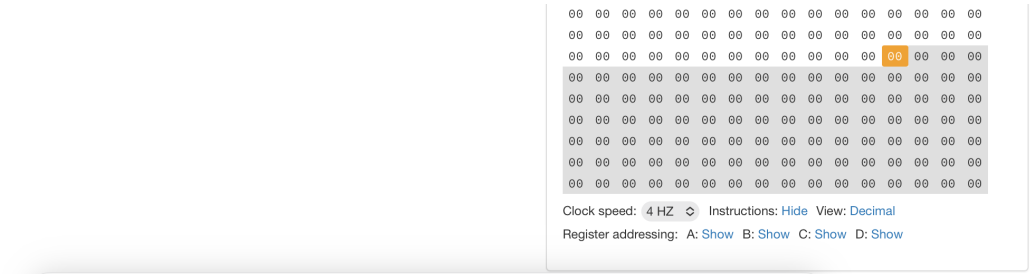



FIG. 5.3 – Etat de la pile avant l'exécution de l'instruction CALL

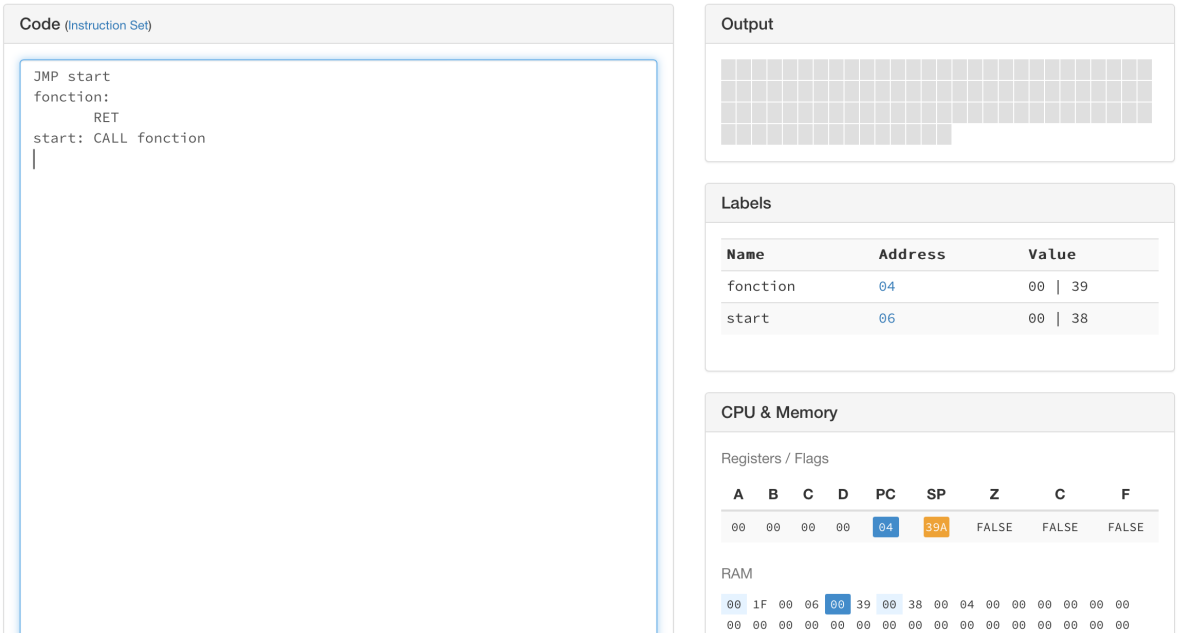



FIG. 5.4 – Etat de la mémoire après l'exécution de l'instruction CALL

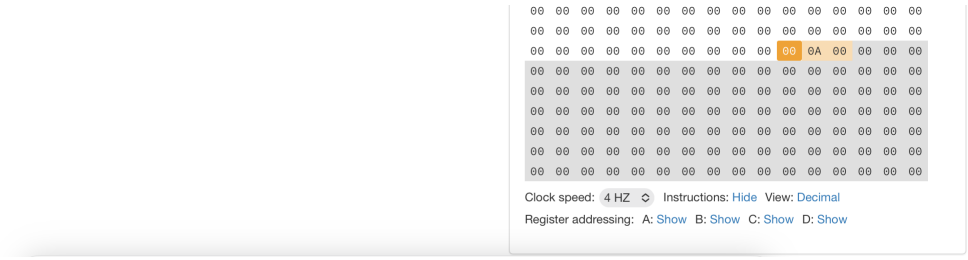



FIG. 5.5 – Etat de la pile après l'exécution de l'instruction CALL

(suite de la page précédente)

```
        MOV compteur, A
        RET
```

Il est intéressant d'observer l'évolution du processeur et de la mémoire durant l'exécution de ce programme. L'exécution de la première l'instruction `CALL COMPTE` a trois effets sur le processeur et la mémoire. Après exécution de l'instruction, le compteur de programme pointe vers l'adresse de l'étiquette `COMPTE` afin de pouvoir exécuter la première instruction de la procédure. L'exécution de l'instruction `CALL` modifie également le registre SP (Stack Pointer en anglais). Au démarrage du processeur, ce registre contient la valeur `39C` qui correspond à une adresse dans la partie « haute » de notre mémoire. Après l'exécution de l'instruction `CALL`, la valeur stockée dans le registre `SP` a été réduite de deux unités et est passé à la valeur `39A`. Si l'on observe la mémoire à l'adresse `39C`, on remarque que l'instruction `CALL` y a placé l'adresse de retour de la procédure, c'est-à-dire celle de l'étiquette `ligne3` dans notre exemple.

Le code de la procédure s'exécute et se termine par l'instruction `RET`. Celle-ci a également trois effets comme l'instruction `CALL`. Tout d'abord, elle lit en mémoire la valeur qui se trouve à l'adresse stockée dans le registre `SP`. Cette valeur est l'adresse qu'il faut placer dans le compteur de programme pour retourner à l'adresse qui suit celle de l'appel de la procédure (l'étiquette `ligne3` dans notre exemple). L'instruction `RET` modifie également la valeur stockée dans le registre `SP` en l'incrémentant de deux unités.

Le registre `SP` n'est pas utilisé par les instructions habituelles telles que `MOV` ou `ADD`. Il ne sert que pour les instructions `CALL` et `RET`. Grâce à ce registre, il est possible de maintenir en mémoire une structure de données appelée une pile (ou stack en anglais). Une pile est une structure de données permettant de stocker un nombre quelconque de données. Elle supporte deux opérations : l'ajout d'une donnée au sommet de la pile (`push` en anglais) et le retrait de la donnée se trouvant au sommet de la pile (`pop` en anglais). La pile de notre processeur est stockée en mémoire et à tout moment l'adresse du sommet de la pile se trouve dans le registre `SP`.

La pile la plus connue dans la vie de tous les jours est la pile d'assiettes. Lorsque l'on a besoin d'un assiette, on prend celle qui se trouve au sommet de la pile. Après avoir fait la vaisselle, on remet les assiettes propres au sommet de la pile également. Pour bien comprendre le fonctionnement d'une structure de données en pile en informatique, il suffit de se rappeler comment on manipule une pile d'assiettes...

---

**Note :** Comment stocker une pile de mots en mémoire ?

La solution la plus simple pour stocker et manipuler une pile de mots en minuscule assembleur est d'utiliser une zone de mémoire contiguë. Une première approche serait d'utiliser l'adresse `p` pour stocker l'élément se trouvant en bas de la pile et d'ajouter les éléments suivants aux adresses `p+2`, `p+4`, ... Pour illustrer cette approche, la Fig. 5.6 présente l'évolution d'une pile initialement vide lors de l'exécution de la séquence `push(3) ; pop ; push(2) ; push(5)` d'opérations. Avec cette approche, le sommet de la pile est toujours l'élément dont l'adresse est numériquement la plus élevée. Outre les données, une telle structure doit également stocker l'adresse de l'élément se trouvant

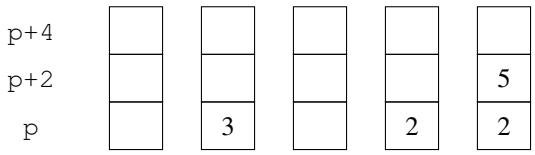



FIG. 5.6 – Évolution de la pile vers les adresses numériquement croissantes

au sommet de la pile. Après l'opération `push(3)` le sommet de la pile est à l'adresse `p`. Il est à la même adresse après l'opération `push(2)` et atteint l'adresse `p+2` après l'opération `push(5)`.

Une seconde approche est d'utiliser l'adresse `p` pour stocker le premier élément de la pile et d'ajouter les éléments suivants aux adresses `p-2`, `p-4`, ... La Fig. 5.7 illustre l'évolution d'un telle pile lors de l'exécution des opérations suivantes : `push(3) ; pop ; push(2) ; push(5)`. Avec cette approche, l'élément se trouvant au sommet de la pile est celui dont l'adresse est numériquement la plus basse. Outre les données, cette structure doit également

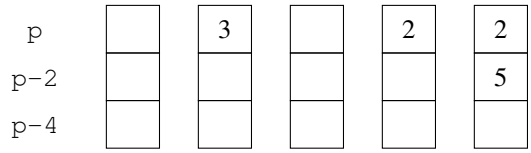



FIG. 5.7 – Évolution de la pile vers les adresses numériquement décroissantes

stocker l'adresse de l'élément se trouvant au sommet de la pile. Après l'opération `push(3)` le sommet de la pile est à l'adresse `p`. Il est à la même adresse après l'opération `push(2)` et atteint l'adresse `p-2` après l'opération `push(5)`.

Même si la première solution peut paraître la plus naturelle par analogie aux piles d'assiettes, c'est généralement la deuxième solution qui est préférée car elle facilite la gestion de la mémoire et maximise l'espace qui est disponible pour la pile sans inutilement contraindre la mémoire utilisée par un programme. C'est ce que notre assembleur fait pour les instructions `CALL` et `RET`.

---

Notre assembleur supporte également les instructions PUSH et POP . L'instruction PUSH peut prendre trois types différents d'arguments :

- un identifiant de registre
- une adresse
- une constante

L'instruction POP ne prend qu'un seul argument, un identifiant de registre. A titre d'exemple, observons l'exécution du code assembleur ci-dessous :

```
1 PUSH 7
2 MOV 122, 3
3 PUSH [122]
4 MOV A, 4
5 PUSH A
6 POP B
7 POP C
8 POP D
```

La première instructions, `PUSH 7` place la valeur `7` au sommet de la pile. La deuxième place la valeur `3` en mémoire à l'adresse `122`. La troisième instruction place la valeur qui se trouve à l'adresse `122`, c'est-à-dire `3` au sommet de la pile. La quatrième instruction, `MOV A, 4` place la valeur `4` dans le registre `A`. La cinquième instruction sauve le contenu du registre `A` sur la pile. La figure Fig. 5.8 présente l'état de la pile à ce moment. Dans cette figure, *sp* est l'adresse qui se trouve dans le registre `SP`. La première instruction `POP` place la valeur qui est actuellement au sommet

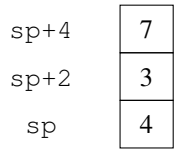



FIG. 5.8 – État de la pile après l'exécution des trois instructions ``PUSH``.

de la pile (c'est-à-dire `4`) dans le registre `B` Elle décrémente ensuite le pointeur de sommet de pile de deux unités. La deuxième instruction `POP` stocke la valeur `3` dans le registre `C`. La figure Fig. 5.9 présente l'état de la pile en mémoire à cet instant. La troisième instruction `POP` retournera la valeur `7` dans le registre `D` et le registre `SP` contiendra l'adresse du sommet de la mémoire. Si on tente d'exécuter une instruction `POP` à ce moment, le processeur affichera *Stack underflow* comme message d'erreur indiquant que la pile est vide.

Durant son exécution, la mémoire d'un de nos programmes en assembleur comprendra trois parties. Le bas de la mémoire nous servira à stocker les données, variables et constantes dont le programme a besoin. Le haut de la mémoire


FIG. 5.9 – État de la pile après l'exécution de deux instructions ``POP``.

servira lui à stocker la pile. Celle-ci démarre à une adresse haute et grandit vers le bas au fur et à mesure que l'on y stocke des adresses et des données. Nous verrons que cette pile a de nombreuses utilisations en assembleur et par extension dans les langages de programmation. Les instructions du programme se trouveront au milieu entre les données et la pile. Cette organisation de la mémoire est illustrée en Fig. 5.10. La pile est utilisée par l'instruction `CALL`

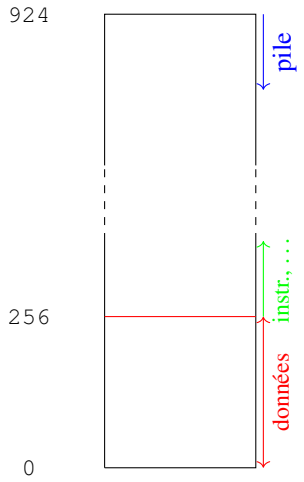



FIG. 5.10 – Organisation de la mémoire d'un programme en assembleur

pour stocker l'adresse de retour. A la fin de l'exécution de la procédure, l'instruction `RET` la récupère. Ce n'est pas la seule utilisation de la pile. Celle-ci permet aussi de stocker des données de façon temporaire. Imaginons que notre procédure `compte` est utilisée dans le programme suivant.

```
            JMP start
x:          DB 0
compteur:   DB 0
start:      MOV A, 123
            CALL COMPTE
            ADD A, [compteur]
            MOV x, A
            HLT
COMPTE:
            MOV A, [compteur]
            INC A
            MOV compteur, A
            RET
```

Ce programme est équivalent au code python suivant qui lors de son exécution place la valeur `124` dans la variable `x`.

```
compteur=0

def compte():
   global compteur
   compteur = compteur+1
```

(suite sur la page suivante)

(suite de la page précédente)

```
x=0
compte()
x=123+compteur
```

Si on exécute le programme équivalent en assembleur, on observe qu'à la fin de son exécution la variable `x` contient la valeur `2`. En observant le code assembleur, on peut facilement comprendre la raison de cette erreur. Avant l'instruction `CALL COMPTE`, nous avons placé la valeur 123 dans le registre `A`. Malheureusement, la procédure `compte` utilise le registre `A` et met sa valeur à `1` lors de l'incrémentation de la variable `compteur`. Cette utilisation du registre `A` par la procédure est un problème. D'un côté, une procédure doit pouvoir modifier des valeurs de registres pour réaliser des calculs. D'un autre côté, le programme qui appelle la procédure ne peut pas savoir quels registres vont être utilisés par la procédure.

On peut résoudre ce problème de deux façons en utilisant la pile. La première solution est de forcer le programme appelant à sauver les valeurs des quatre registres sur la pile avant d'exécuter `CALL` et de récupérer les valeurs des quatre registres sur la pile dès le retour de la procédure. Dans notre exemple, cela pourrait se faire comme suit :

```
            JMP start
x:          DB 0
compteur:   DB 0
start:      MOV A, 123
            PUSH A
            PUSH B
            PUSH C
            PUSH D
            CALL COMPTE
            POP D
            POP C
            POP B
            POP A
            ADD A, [compteur]
            MOV x, A
            HLT
COMPTE:
            MOV A, [compteur]
            INC A
            MOV compteur, A
            RET
```

Lorsque l'on exécute ce programme, la variable `x` contient bien la valeur `124` comme en python. Notez l'ordre dans lequel les valeurs des registres sont stockées (`A` puis `B` puis `C` puis `D`) et ensuite récupérées sur la pile (`D` puis `C` puis `B` puis `A`). L'ordre dans lequel on pousse les valeurs sur la pile importe peut, pour autant qu'elles soient récupérées dans l'ordre exactement inverse.

Cette approche fonctionne, mais elle implique parfois des instructions inutiles. Dans notre exemple, la procédure `compte` n'utilise que le registre `A`. Il est donc inutile de sauver les valeurs stockées dans les trois autres registres, mais le programme appelant ne connaît pas cette caractéristique de notre procédure. Une meilleure approche est de laisser à la procédure appelée la responsabilité de préserver les valeurs des registres qu'elle modifie. C'est cette approche que nous utilisons dans l'exemple suivant.

```
            JMP start
x:          DB 0
compteur:   DB 0
start:      MOV A, 123
            CALL COMPTE
            ADD A, [compteur]
            MOV x, A
```

(suite sur la page suivante)

(suite de la page précédente)

```
            HLT
COMPTE:
            PUSH A ; sauvegarde du contenu du registre A qui va être modifié
            MOV A, [compteur]
            INC A
            MOV compteur, A
            POP A  ; récupération du contenu du registre A
            RET
```

Dans ce programme, la procédure `compte` sauve la valeur du registre `A` sur la pile avant de la modifier. En pratique, le code d'une procédure commencera généralement par une sauvegarde des valeurs de registres qui peuvent être modifiés dans le corps de la procédure. Elle se terminera par les instructions `POP` correspondantes dans l'ordre inverse de celui utilisé pour stocker les données au début de la procédure.

Grâce à la pile, il est possible d'écrire des programmes qui contiennent un nombre quelconque de procédures qui s'appellent l'une l'autre et dans un ordre quelconque. La pile grandira au fur et à mesure des appels successifs à des procédures et rétrécira chaque fois qu'une procédure se termine. Il est important de noter que pour que ce système fonctionne correctement il est nécessaire que chaque procédure manipule correctement la pile. Si le sommet de la pile se situe à l'adresse `z` au début de l'exécution d'une procédure, à la fin de celle-ci la pile doit contenir exactement les mêmes informations. Si une procédure laissait la pile avec un élément en plus ou un élément en moins lorsqu'elle retourne à l'adresse de retour dans le programme appelant, alors le programme complet ne fonctionnerait plus correctement. Il faut être très rigoureux lorsque l'on écrit des programmes en langage assembleur qui manipulent la pile.

---

**Note :** Stack overflow

Les langages de programmation tels que python utilisent aussi une pile pour supporter les appels de procédures et de fonctions. C'est à l'interpréteur ou au compilateur de gérer correctement la pile. En général, le langage de programmation réserve une zone mémoire pour stocker la pile du programme. Certains langages de programmation comme python ou Java vérifient que la pile ne déborde pas lors de l'exécution d'un programme. Le cas échéant, ils lancent une exception qui indique un dépassement de pile (stack overflow en anglais) et le programme est arrêté. Pour cela, ils doivent vérifier l'état de la pile avant chaque opération `push` ou `pop`. D'autres langages de programmation comme le C ne vérifient pas la taille de la pile à chaque opération. Avec ces langages, il est possible que la pile croisse tellement qu'elle rencontre la zone contenant les instructions ou même les données du programme. Dans ce cas, le programme aura un comportement totalement incohérent. Certains problèmes de sécurité sur des programmes écrits en C exploitent ce genre de limitations du langage.

---

Nous avons utilisé la pile pour stocker les adresses de retour des procédures ainsi que pour sauvegarder temporairement les valeurs des registres. Ce n'est pas la seule utilisation de la pille. Elle va également nous permettre de supporter les fonctions auxquelles il faut passer des arguments du programme appelant vers la fonction, mais aussi récupérer des valeurs de retour. Il faut aussi permettre à une fonction d'utiliser de la mémoire pour stocker des données temporaires pendant son exécution et de libérer correctement cette mémoire après.

Revenons à un exemple simple en python pour bien comprendre les différences entre une fonction et une procédure. Notre première fonction, `f1`, prend un entier en argument et retourne un entier également. Durant son exécution, elle utilise une variable locale, `y`. La deuxième fonction, `f2` prend également un entier en argument et retourne un résultat entier. Le corps de la fonction `f2` fait deux appels à la fonction `f1` et utilise deux variables locales. Enfin, la fonction `min` prend deux arguments entiers et retourne un résultat entier. Elle utilise également une variable locale.

```
# incrémente son argument de 1
def f1(x):
    y=x+1
    return(y)
```

(suite sur la page suivante)

(suite de la page précédente)

```
# incrémente son argument de 2
def f2(x):
    y=f1(x)
    z=f1(y)
    return(z)

# retourne le minimum
def min(x,y):
    if (x<y):
        r=x
    else:
        r=y
    return(r)

print(f1(3)) # affiche 4
print(f2(5)) # affiche 7
print(min(3,5)) # affiche 3
```

Pour supporter ces différents types de fonctions, nous devons répondre à trois questions :

1. Comment un programme appelant peut-il passer les arguments à une fonction ?
2. Comment un programme appelant peut-il récupérer le résultat d'une fonction ?
3. Comment une fonction peut-elle utiliser de la mémoire pour stocker ses variables locales ?

Commençons par la première question. Avant d'appeler une fonction, il est nécessaire d'avoir d'abord calculé les valeurs des arguments que l'on doit passer à cette fonction. Une fonction peut avoir un, deux, ou un nombre quelconque d'arguments. Ceux-ci devront être placés à un endroit où la fonction pourra les récupérer. Dans notre processeur, ces arguments peuvent être mis à deux endroits différents :

— dans des registres, avec un argument par registre
— en mémoire

La première solution a l'avantage d'être simple et rapide. Il suffit d'exécuter une instruction `MOV` pour placer la valeur d'un argument au bon endroit. Malheureusement, notre processeur ne dispose que de quatre registres au total. Nous ne pourrons donc jamais supporter de fonction avec plus de quatre arguments.

La seconde solution est plus générale. Nous utilisons déjà la pile pour récupérer l'adresse de retour et on peut facilement envisager de placer des arguments sur la pile avant l'exécution d'une fonction. Il suffit pour cela d'utiliser l'instruction `PUSH` pour chaque argument à pousser sur la pile. La fonction pourra récupérer chaque argument en faisant appel à `POP` dans l'ordre inverse de celui du programme appelant.

Pour le résultat de la fonction, deux approches sont possibles. La première est d'utiliser la pile pour retourner ce résultat. La seconde est de placer le résultat de la fonction dans un registre du processeur. La première solution a l'avantage de permettre à une fonction de retourner plusieurs résultats, comme en python par exemple. La seconde est utilisée par de très nombreux langages de programmation. C'est celle que nous adoptons dans ce chapitre. Dans le cadre de ce syllabus, nous prenons la convention **qu'une fonction écrite en assembleur retournera un seul mot de 16 bits et que ce résultat sera toujours placé dans le registre ``A``**.

Pour que les fonctions et procédures écrites par un ou une informaticienne soient utilisables sans difficultés par d'autres personnes, il est important que le programme appelant et la fonction/procédure utilisent les mêmes conventions d'utilisation de la pile. Dans le cadre de ce syllabus, nous prenons les conventions suivantes pour les fonctions et procédures en assembleur :

— **le premier argument d'une fonction/procédure est toujours placé dans le registre ``D``.**
— **les deuxième, troisième, ... arguments d'une fonction/procédure sont poussés sur la pile par le programme appelant avec la séquence d'instructions ``PUSH arg2``, ``PUSH arg3``, ... avant l'instruction ``CALL``.**
— **le résultat ou valeur de retour d'une fonction est toujours placée dans le registre ``A``**
— **lors de l'appel à une fonction/procédure, le programme appelant a la garantie que les registres ``B`` et ``C`` auront la même valeur au retour de la fonction/procédure qu'avant l'appel. Cela implique que le**

**programme appelant ne doit pas sauver ces registres sur la pile avant d'appeler une fonction/procédure. Par contre, si la fonction/procédure utilise les registres ``B`` ou ``C``, elle doit préserver leurs valeurs en utilisant la pile.**

- **le corps d'une fonction/procédure peut modifier les valeurs des registres ``A`` et ``D`` à sa guise. Cela implique que si le programme appelant veut réutiliser la valeur se trouvant dans le registre ``D`` après un appel de fonction/procédure, il devra la sauver sur la pile avant d'exécuter l'instruction ``CALL``.**
- **toute fonction ou procédure qui ajoute une ou des données sur la pile, doit s'assurer qu'à la fin de son exécution la pile retrouve l'état qu'elle avait avant l'appel à la fonction/procédure.**

Nous pouvons facilement écrire le code de la fonction `f1` en appliquant ces conventions. Elle prend son argument dans le registre `D`, l'incrémente et stocke le résultat dans le registre `A`. Comme elle ne modifie pas les registres `B` et `C`, elle ne doit pas les sauver sur la pile.

```
f1:
        MOV A, D
        INC A
        RET
```

La fonction `f2` fait elle deux appels à la fonction `f1`. Pour chacun de ces appels, on doit vérifier que l'argument correct est bien placé dans le registre `D` avant d'exécuter l'instruction `CALL`. Cela peut se faire en quelques instructions.

```
f2:         CALL f1
            MOV D, A
            CALL f1
            RET
```

Nous pouvons maintenant analyser une fonction qui prend deux arguments comme celle qui calcule le minimum entre deux entiers.

Analysons d'abord comment la fonction doit être appelée depuis un programme. Son premier argument doit se trouver dans le registre `D` et le second doit être sur la pile avant d'exécuter l'instruction `CALL min`. Le premier argument pourra être calculé ou mis dans le registre `D` via une instruction `MOV`. Le second argument est lui placé sur la pile grâce à une instruction `PUSH`. A ce moment, la fonction `min` peut être appelée via l'instruction `CALL min`. Au retour de la fonction `min`, le registre `A` contiendra le minimum des deux arguments. Le programme pourra traiter cette valeur minimale comme il le souhaite. Cependant, comme le programme a modifié la pile avant d'appeler la fonction, il ne doit pas oublier de remettre la pile dans son état initial. Cela peut se faire de deux façons :

- en utilisant une instruction `POP` qui copie la valeur se trouvant au sommet de la pile et incrémente
- en incrémentant simplement le pointeur de sommet de pile (`SP`) de deux unités

La seconde solution a l'avantage de ne pas modifier de valeur de registre. C'est celle que nous utilisons dans cet exemple. Nous supposons que `arg1` et `arg2` sont des nombres naturels pour lesquels nous cherchons le minimum.

```
; appel à la fonction min
MOV D, arg1     ; premier argument
PUSH arg2       ; second argument
CALL min
ADD SP, 2       ; libération de l'argument placé sur la pile
```

Si nous souhaitons passer comme arguments à la fonction `min` des valeurs de variables, alors nous devons modifier notre code comme présenté ci-dessous.

```
; appel à la fonction min
MOV D, [adr1]   ; adresse de la première variable
PUSH [arg2]     ; adresse de la seconde variable
CALL min
ADD SP, 2       ; libération de l'argument placé sur la pile
```

Nous pouvons maintenant écrire la fonction qui calcule le minimum entre les valeurs de ses deux arguments. Le premier argument se trouve dans le registre `D`. Il est donc immédiatement disponible. Le second argument lui est sur la

pile. Il y a été placé *avant* l'appel à la fonction via l'instruction `CALL min`. La figure Fig. 5.11 présente l'état de la pile à ce moment. Pour accéder au second argument, il n'est pas intéressant d'utiliser une instruction `POP` car celle-ci ne

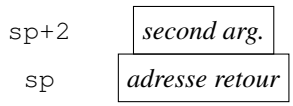



FIG. 5.11 – État de la pile au début de l'exécution de la fonction ``min``

permet que d'accéder à l'élément au sommet de la pile, c'est-à-dire l'adresse de retour. Par contre, comme nous savons que l'élément suivant sur la pile se trouve juste au-dessus de cet élément, il nous suffit d'utiliser l'instruction `MOV A, [SP+2]`. Cette instruction charge dans le registre `A` la valeur qui se trouve en mémoire à l'adresse correspondant à celle qui est stockée dans le registre `SP` plus deux unités. Comme un mot de 16 bits en mémoire consomme deux adresses, c'est bien l'adresse de l'élément suivant sur la pile et donc notre second argument. Nous plaçons cet argument dans le registre `A` car la fonction peut modifier ce registre à sa guise. Ensuite, il suffit de comparer les valeurs se trouvant dans les registres `A` et `D` et retourner le minimum dans le registre `A`. L'instruction `RET` utilise l'adresse de retour qui se trouve au sommet de la pile.

```
; calcule le minimum entre les deux valeurs passées en argument


min:
          MOV A,[SP+2] ; second argument copié dans A
          CMP A,D
          JBE finmin
          MOV A,D
finmin:   RET
```

Dans notre implémentation des fonctions `f1` et `min`, nous avons utilisé la technique du passage par valeur, c'est-à-dire que lorsqu'elle est appelée, une fonction reçoit du programme appelant les *valeurs* de ses arguments. Ces valeurs sont copiées dans le registre `D` ou sur la pile par le programme appelant et utilisées par la fonction. Cette technique est utilisée par de nombreux langages de programmation comme python lorsque l'on passe des valeurs d'un type primitif comme des réels ou des entiers à une fonction.

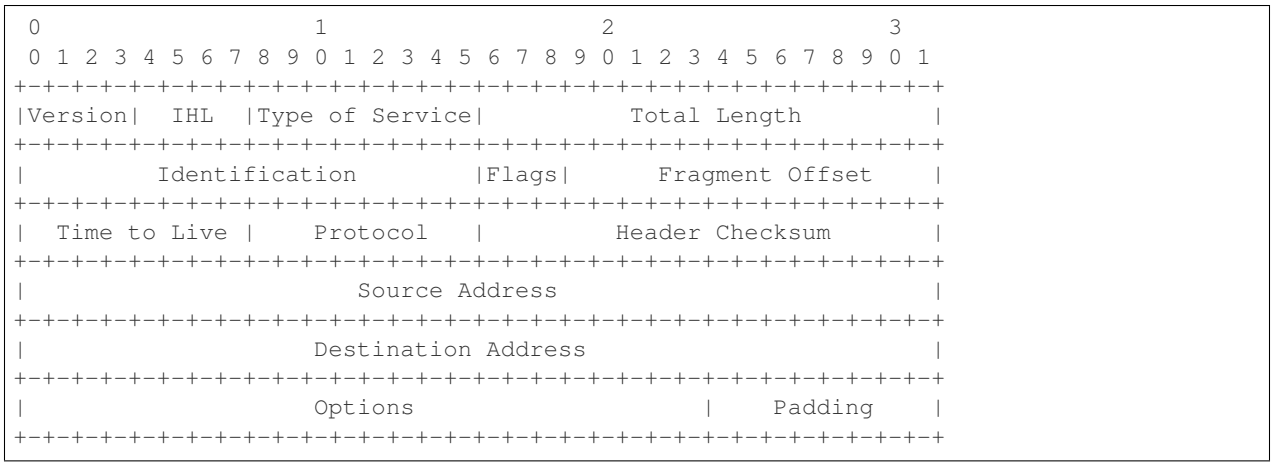
Il existe une seconde technique pour passer les arguments à une fonction. C'est le passage par référence. Dans ce cas, le programme appelant fournit à la fonction qu'il appelle une référence vers son argument. Cette référence est l'adresse en mémoire à laquelle la variable contenant l'argument est stockée. La différence fondamentale entre le passage par référence et le passage par valeur est que comme la fonction connaît l'adresse de la variable contenant son argument, elle peut modifier son contenu alors que c'est impossible avec le passage par valeur. En python, le passage par référence est utilisé lorsque l'argument passé à une fonction est une référence à un objet ou une liste. Il est possible de mixer le passage par référence et le passage par valeur dans une même fonction avec un argument entier passé par valeur et une liste passée par référence.

A titre d'illustration, la fonction `inc` ci-dessous permet d'incrémenter la variable dont l'adresse est passée par le programme appelant comme argument. Le corps de la fonction `inc` accède à l'adresse de la variable utilisée par le programme appelant et modifie sa valeur avant de terminer son exécution.

```
inc:
      MOV A, [D]
      INC A
      MOV [D], A
      RET
```

En assembleur, on stocke parfois l'information sous la forme d'une séquence de bits. Lorsque les ordinateurs communiquent sur Internet, ils s'échangent les données sous forme de paquets. Chaque paquet est composé d'une entête de quelques dizaines d'octets et suivi des données qui sont échangées. L'entête d'un paquet comprend différents champs

dont les valeurs sont fixées par l'émetteur du paquet et qui peuvent être modifiées par les nœuds du réseau (appelés routeurs). A titre d'exemple, la figure ci-dessous présente le format de l'entête d'un paquet IP version 4. Chaque ligne correspond à un mot de 32 bits et l'émetteur d'un tel paquet doit pouvoir spécifier précisément les valeurs de certains bits dans les champs tels que `Type of Service` ou `Flags`.




Notre microprocesseur utilise des mots de 16 bits. Pour traiter une entête de paquet IP, nous devrons donc la découper en blocs de 16 bits qui seront chacun stocké en mémoire ou dans un registre du microprocesseur. Pour faciliter la modification de certains bits en mémoire, nous construisons deux fonctions. Ces fonctions prennent deux arguments :

— la position du bit à modifier (qui est placée dans le registre `D` suivant notre convention). On supposera que le bit de poids faible est celui qui se trouve en position `0` tandis que le bit de poids fort est en position `15`
— le mot de 16 bits à modifier (qui est placé sur la pile suivant notre convention)

Lors de l'exécution de ces fonctions, la pile contient donc l'adresse de retour précédée du mot de 16 bits à modifier comme représenté dans la Fig. 5.12. Après exécution, nos deux fonctions retournent le mot modifié dans le registre `A`. Notre première fonction est `setbit`. Elle fixe le nième bit du mot placé sur la pile à la valeur 1. Pour construire

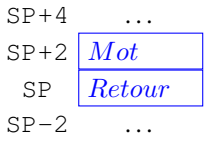



FIG. 5.12 – État de la pile avant l'exécution des fonctions setbit ou resetbit

cette fonction, il faut se rappeler un peu de logique et utiliser les instructions de décalage. En logique, on se souvient que `OR(x,1)` vaut toujours `1`, quelle que soit la valeur de `x`. Pour mettre à `1` le bit à l'index 3 d'un mot de seize bits `m`, il suffit donc de calculer `OR(0000000000001000,m)`. Il nous reste donc à trouver un façon rapide de construire la séquence de bits dont le bit en position `n` vaut `1` et tous les autres valent `0`. Pour cela, il suffit d'utiliser les opérations de décalage. En effet, `SHL(0000000000000001,1)` donne `0000000000000010`. Pour obtenir la séquence `0000000000001000` il suffit d'utiliser l'instruction `SHL(0000000000000001,3)`. En assemblant ces différentes instructions, on obtient le code ci-dessous.

```
; fonction pour fixer le nième bit d'un mot à 1
; pre: n est dans le registre D et 0<=n<15
; le mot à modifier est sur la pile, retourne le mot
setbit:
   PUSH B ; préservation de la valeur de B
   MOV B, 1
   SHL B, D
   MOV A, [SP+4]
   OR A, B
```

(suite sur la page suivante)

(suite de la page précédente)

```
   POP B  ; récupération de la valeur de B
   RET
```

De la même façon, on peut aisément construire une fonction pour mettre à zéro le nième bit d'un mot de 16 bits en se rappelant que `AND(0,x)` vaut toujours `0` et que `AND(1,y)` vaut `y`. Pour forcer la valeur du bit en position 4 du mot `m`, il suffit de calculer `AND(111111111101111,m)` et `1111111111101111` n'est rien d'autre que `NOT(0000000000010000)`. Le code de notre fonction `resetbit` se déduit facilement.

```
; fonction pour fixer le nième bit d'un mot à 0
; pre: n est dans le registre D et 0<=n<15
; le mot à modifier est sur la pile, retourne le mot
resetbit:
   PUSH B
   MOV B, 1
   SHL B, D
   NOT B
   MOV A, [SP+4]
   AND A, B
   POP B
   RET
```

A titre d'exemple, le code ci-dessous teste les fonctions `testbit` et `resetbit`.

```
; exemple d'appel à la fonction set
testset:
   MOV D, 3
   PUSH 2
   CALL setbit
   HLT
```

```
; exemple d'appel à la fonction reset
testreset:
   MOV D, 3
   PUSH 14
   CALL resetbit
   HLT
```

Prenons un autre exemple pour illustrer l'utilisation de la pile et passer plus d'arguments. Construisons la fonction qui calcule l'équivalent de la fonction `f` qui en python calcule `y=a*x+b`

```
def f (x, a, b):
    return a*x+b
```

En assembleur, nous devons passer les trois arguments à cette fonction en utilisant le registre `D` pour la valeur de `x` et la pile pour les valeurs de `a` et `b`. Nous devons définir l'ordre dans lequel nous plaçons les arguments sur la pile. Si la valeur de `a` est passée en premier et celle de `b` en second, alors lors de l'exécution de notre fonction `f`, la pile contient les valeurs représentées dans Fig. 5.13. Nous pouvons ensuite construire notre fonction en utilisant les instructions `MOV`, `MUL` et `ADD` à bon escient.

```
; calcule y=a*x+b
; x est passé dans le registre D
; a et b sont passés sur la pile
; SP   : adresse de retour
; SP+2 : b
; SP+4 : a
```

(suite sur la page suivante)

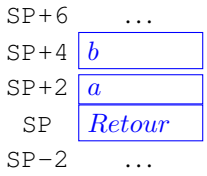



FIG. 5.13 – État de la pile durant l'exécution de la fonction f

(suite de la page précédente)

```
f:
   MOV A, [SP+4] ; récupère x
   MUL D
   ADD A, [SP+2]
   RET
```

On peut vérifier le bon fonctionnement de cette fonction en exécutant le code ci-dessous :

```
MOV D, 3
PUSH 4
PUSH 1
CALL f
; A contient 3*4+1
; ne pas oublier de libérer la pile après
ADD SP, 4
HLT
```

Comme autre exemple d'utilisation de la pile, considérons une fonction qui permet d'ajouter un entier à tous les éléments d'un tableau d'entiers. En python, une telle fonction peut s'écrire comme suit.

```
def ajouter_entier(entier, tableau):
     for i in range(len(tableau)):
             tableau[i] += entier
```

Pour traduire cette fonction en assembleur, nous devons déterminer comment passer ses arguments. Nous choisissons de mettre l'entier à ajouter dans le registre `D`. Pour passer le tableau, nous devons pousser l'adresse du premier élément sur la pile et ensuite la taille du tableau. Au début de l'exécution de la fonction `ajouter_entier`, la pile contient donc les informations reprises en Fig. 5.14. Notre fonction va utiliser les registres `B` et `C`. Nous devons donc d'abord les

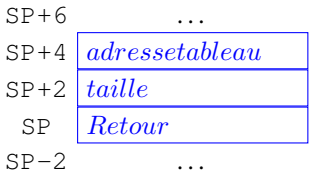



FIG. 5.14 – État de la pile au début de l'exécution de la fonction ajouter_entier

sauvegarder sur la pile. Durant l'exécution de la fonction `ajouter_entier`, la pile contiendra donc les informations reprises en Fig. 5.15. La fonction peut ensuite aisément se traduire en assembleur comme une boucle qui itère sur tous les éléments du tableau. Il faut cependant être attentif à correctement charger l'adresse du tableau et sa taille depuis la pile. Comme nos entiers sont stockés sur 16 bits, nous devons aussi incrémenter l'adresse du prochain élément de deux unités à chaque passage dans la boucle.

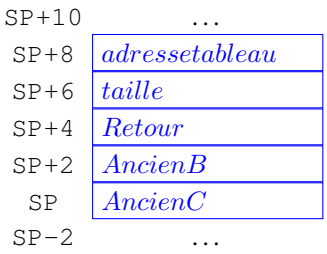



FIG. 5.15 – État de la pile pendant l'exécution de la fonction ajouter_entier

```
; ajoute son premier argument à tous les éléments du tableau
; registre D: x, valeur à ajouter
; SP+4 : adresse du premier élément du tableau
; SP+2 : nombre d'éléments
ajouter_entier:
   PUSH B ; préservation du contenu de B
   PUSH C ; préservation du contenu de C
   MOV B, [SP+8] ; adresse du tableau
   MOV C, [SP+6] ; nombre d'éléments
loop:
   MOV A, D
   ADD A, [B] ; calcule de tableau[i]+entier
   MOV [B], A ; sauvegarde du résultat en mémoire
   ADD B, 2 ; adresse suivante
   DEC C
   CMP C, 0
   JNE loop
   POP B ; récupération du contenu de B
   POP C ; récupération du contenu de C
   RET
```

Cette fonction peut être testée en créant un tableau contenant quelques entiers et en appelant la fonction comme dans l'exemple ci-dessous.

```
; création d'un tableau de 5 entiers
t: DB 1
   DB 2
   DB 3
   DB 4
   DB 5
   MOV D, 3 ; valeur à ajouter
   PUSH t   ; adresse du premier élément du tableau
   PUSH 5   ; nombre d'éléments
   CALL ajouter_entier
```

Nous devons maintenant trouver une réponse à la troisième question. Lors de son exécution, une fonction doit souvent utiliser de la mémoire, pour stocker des résultats intermédiaires de calculs ou des variables locales. C'est le cas de la fonction `fct` dans l'exemple en python ci-dessous. Celle-ci a besoin de mémoire pour réaliser les calculs qui se trouvent dans son corps. Il en va de même par exemple pour une fonction qui contiendrait une simple boucle.

```
# retourne x+2*y si x<y et y-5*x sinon
def fct(x,y):
    if (x<y):
        r=x+2*y
```

(suite sur la page suivante)

(suite de la page précédente)

```
    else:
        r=y-5*x
    return(r)
```

Chacune des variables locales d'une fonction doit être stockée à une adresse mémoire. Une première approche naïve pour résoudre ce problème serait de réserver une zone de mémoire fixe pour les variables locales utilisées par chaque fonction. Dans une implémentation en assembleur de l'exemple ci-dessus, on pourrait réserver une adresse en mémoire RAM pour la variable `r` de la fonction `fct`. Malheureusement, cette approche a deux inconvénients. Premièrement, toute la mémoire qu'une fonction peut utiliser durant son exécution doit être réservée en RAM avant de pouvoir exécuter cette fonction. Si une fonction doit utiliser un grand tableau lorsqu'elle est appelée avec une valeur spécifique d'un argument, alors la zone nécessaire pour ce tableau doit toujours être réservée, même si la fonction n'est jamais exécutée par le programme. Le deuxième inconvénient est qu'il est impossible avec cette approche de supporter une fonction `f` qui appelle une fonction `g` qui elle-même appelle une fonction `f` car le premier appel à la fonction `f` aura initialisé les « variables locales de la fonction `f` » puis fera appel à la fonction `g`. Lorsque `g` fait appel de son côté à la fonction `f`, cette seconde invocation de la fonction `f` va modifier les données stockées aux adresses en mémoire qui correspondent à ses variables locales et donc modifier les variables utilisées par la première invocation de la fonction `f`. Si ce second inconvénient peut paraître un peu théorique et hypothétique à ce stade, il est malheureusement bien réel en pratique.

On peut éviter ces deux inconvénients en utilisant la pile comme mémoire pour stocker les variables locales d'une fonction. La pile n'utilise la RAM que durant l'exécution de la fonction, il n'y a donc pas de gaspillage de mémoire comme avec la solution précédente. Dans le cas où une invocation de la fonction `f` appelle la fonction `g` qui appelle elle-même la fonction `f`, le bas de la pile contiendra les arguments, adresse de retour et variables de la première invocation de la fonction `f`. Au-dessus de ces informations, on trouvera les arguments, adresses de retour et variables locales de la fonction `g`. Enfin, les arguments, adresse de retour et variables locales de la seconde invocation de la fonction `f` sont au sommet de la pile. A la fin de son exécution, cette invocation de la fonction `f` libère la mémoire qu'elle utilise sur la pile.

La meilleure illustration de l'utilisation de la pile par les fonctions en assembleur est le support des fonctions récursives. En informatique, on parle de récursion lorsqu'une fonction s'appelle elle-même. C'est le cas par exemple de la fonction `sumn` qui permet de calculer la somme des `n` premiers naturels.

```
# Somme des n premiers naturels
def sumn(n):
    if(n==0):
        return(0)
    return(n+sumn(n-1))

print(sumn(1)) # affiche 1
print(sumn(3)) # affiche 6
```

En assembleur, cette fonction peut s'appeler en plaçant simplement la valeur de la variable `n` dans le registre `D` en utilisant le code ci-dessous.

```
MOV D, [n]
CALL sumn
; résultat dans le registre A
```

Le code de la fonction `sumn` en assembleur comprend deux parties principales : le cas de base (`n==1`) et le cas récursif. Pour la partie récursive, nous devons calculer `n+sum(n-1)`. Pour cela, il est nécessaire de placer la valeur de `n-1` dans le registre `D` pour appeler la fonction `sumn` tout en gardant la valeur de `n` pour pouvoir calculer la somme `n+sum(n-1)`. Cela nécessite d'utiliser le registre `B` à l'intérieur de notre fonction. C'est pour cette raison que nous sauvegardons ce registre via l'instruction `PUSH B` au début du code de la fonction. Nous plaçons également

une instruction `POP B` pour récupérer la valeur placée sur la pile juste avant d'exécuter l'instruction `RET`.

```
sumn:
        PUSH B    ; sauvegarde
        CMP D,1
        JNE recursif
        MOV A, D ; n==1, return(n)
        POP B    ; récupération
        RET
recursif:
        MOV B, D ; on aura besoin de n après
        DEC D
        CALL sumn ; appel récursif
rec2:
        ADD A, B  ; résultat dans A
        POP B
        RET
```

Pour bien comprendre le fonctionnement d'un tel programme récursif et son utilisation de la pile, il est intéressant d'observer son exécution pas à pas en parallèle avec l'évolution de la pile. Cette fonction est appelée par le code suivant :

```
    MOV B, 17
    MOV D, 3
    CALL sumn
RETOUR:
```

La Fig. 5.16 présente l'état de la pile lors de l'appel à la fonction `sumn` avec `3` comme argument. Par convention, le sommet de la pile se trouve en bas de la figure et utilise une police de caractères grasse. Durant son exécution, cette


FIG. 5.16 – Contenu de la pile lors de l'appel à la fonction sumn(3)

fonction sauvegarde d'abord le registre `B` puis fait appel à `sumn(2)`. La Fig. 5.17 présente l'état de l'appel au moment de cet appel. Lors de son exécution, l'invocation de la fonction `sumn` avec `2` comme argument va d'abord faire appel

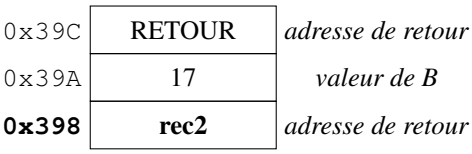



FIG. 5.17 – Contenu de la pile lors de l'appel à sumn(2)

à `sumn(1)`. La Fig. 5.18 présente l'état de la pile au moment de cet appel. Lors de son exécution, l'invocation de la fonction `sumn` avec `1` comme argument va d'abord sauvegarder le registre `B` sur la pile. La Fig. 5.19 présente l'état de la pile au moment de cet appel. Nous sommes maintenant dans l'exécution de la fonction `sumn(1)`. Celle-ci retourne la valeur `1` dans le registre `A` et retire le mot se trouvant au sommet de la pile. Elle retourne à l'adresse `RETSUMN` avec la pile dans l'état représenté à la Fig. 5.18. Grâce à cette pile, la fonction `sumn` récupère son argument (`1`) et retourne la valeur `1` qui est la somme entre la valeur du registre `B` et son argument. A la fin de son exécution, cette invocation de la fonction `sumn` retire les deux mots qui se trouvaient au sommet de la pile.

L'état de la pile est maintenant celui de la Fig. 5.17 et le registre `A` contient la valeur `1`. Nous sommes dans la dernière partie de l'invocation de la fonction `sumn(2)`. Celle-ci calcule son résultat (`3`) et retire les deux mots se trouvant au sommet de la pile avant de faire un saut à l'adresse `RETSUMN`.

| | | |
|---|---|---|
| 0x39C | RETOUR | *adresse de retour* |
| 0x39A | 17 | *valeur de B* |
| 0x398 | rec2 | *adresse de retour* |
| 0x396 | 2 | *sauvegarde du registre B* |
| **0x394** | **rec2** | *adresse de retour* |

FIG. 5.18 – Contenu de la pile lors de l'appel à sumn(1)

| | | |
|---|---|---|
| 0x39C | RETOUR | *adresse de retour* |
| 0x39A | 17 | *valeur de B* |
| 0x398 | rec2 | *adresse de retour* |
| 0x396 | 2 | *sauvegarde du registre B* |
| 0x394 | rec2 | *adresse de retour* |
| **0x392** | **1** | *sauvegarde du registre B* |

FIG. 5.19 – Contenu de la pile durant l'exécution de sumn(1)

Nous sommes maintenant dans l'invocation de la fonction `sumn(3)`. L'état de la pile est celui de la Fig. 5.16. La fonction récupère l'argument (3) et l'ajoute au résultat de la fonction appelée qu'elle a reçu dans le registre `A` (3). Le registre `D` contient maintenant le résultat final (6) de l'appel `sumn(3)`. Il ne reste plus qu'à retirer les deux mots se trouvant au sommet de la pile et retourner à l'adresse `RETOUR` dans le programme appelant.

En utilisant la même approche, on peut construire une implémentation en assembleur de la fonction qui permet de calculer la factorielle d'un naturel.

```
; Calcul de la factorielle d'un naturel, argument dans D, résultat dans A

fact:
      PUSH B
      CMP D,1
      JNE suite
      MOV A, D
      POP B
      RET
suite:
  MOV B, D
  DEC D
      CALL fact
      MUL B
      POP B
      RET
```

CHAPITRE 6

# Les structures de données

Le langage python permet de supporter différents types de structure de données dont les piles, les queues et les listes. Il est intéressant de comprendre comment ces différentes structures de données sont stockées en mémoire avant de les implémenter en assembleur. Nous avons déjà parlé de la pile qui est utilisée par notre processeur, mais une application peut aussi définir sa propre pile, indépendamment de celle que le processeur utilise pour supporter les fonctions et procédures.

Une pile utilisée par une application est une structure de données qui permet de stocker des informations et de les récupérer dans l'ordre inverse d'arrivée (dernier arrivé, premier servi ou last-in first-out en anglais). La pile a une interface de programmation qui comprend trois opérations :

- une opération permettant d'initialiser une pile vide
- une opération permettant d'ajouter un élément sur la pile
- une opération permettant de récupérer l'élément se trouvant au sommet de la pile

Considérons une pile qui permet de stocker des naturels. En python, une telle pile peut être construite en utilisant une liste. On y ajoute un élément avec l'opération `append()` et on récupère l'élément au sommet de la pile avec l'opération `pop()`.

```
>>> pile=[]
>>> pile.append(7)
>>> pile.append(9)
>>> pile.pop()
9
>>> pile.append(3)
>>> pile.append(1)
>>> pile.pop()
1
```

Il est intéressant d'observer l'évolution de cette pile en la représentant d'abord comme une pile d'assiettes. Après exécution de `pile.append(7)`, notre pile comprend la valeur `7` à son sommet (Fig. 6.1). La figure Fig. 6.2 représente


FIG. 6.1 – Représentation de la pile après exécution de ``pile.append(7)``

notre pile après exécution de `pile.append(9)`. Cette pile comprend deux éléments. La valeur `9` est au sommet de la pile et la valeur `7` le second élément. Après avoir récupéré la valeur `9` du sommet de la pile, on y ajoute ensuite deux

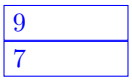



FIG. 6.2 – Représentation de la pile après exécution de ``pile.append(9)``

éléments via les opérations `pile.append(3)` et `pile.append(1)`. A ce moment, la pile contient trois éléments comme représenté dans la Fig. 6.3. Avant de supporter une telle structure de données en assembleur, il est utile de

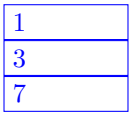



FIG. 6.3 – Représentation de la pile après exécution de ``pile.append(3)`` et ``pile.append(1)``

se demander comment elle peut être représentée en mémoire. Il faut bien entendu disposer d'espace pour stocker les naturels que l'on stocke sur la pile, mais il faut aussi mémoriser l'ordre dans lequel les opérations d'ajout à la pile ont étés effectuées pour pouvoir retourner les données stockées dans l'ordre inverse. Un première approche possible serait de réserver une zone de mémoire pour stocker celle pile et d'y stocker le nombre d'éléments se trouvant sur cette pile. Si cette zone de mémoire commence à l'adresse $p$, elle pourrait être initialisée comme dans la Fig. 6.4. Après

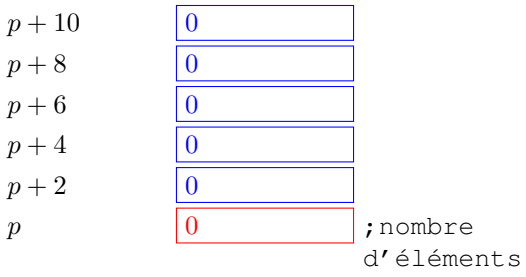



FIG. 6.4 – Stockage d'une pile dans un bloc de mémoire

exécution de `pile.append(7)`, cette pile contient un élément et la valeur `7` à son sommet (Fig. 6.5). Après avoir récupéré la valeur `9` du sommet de la pile, on y ajoute ensuite deux éléments via les opérations `pile.append(3)` et `pile.append(1)`. A ce moment, la pile contient trois éléments comme représenté dans la Fig. 6.6. Malheureusement, cette solution de stockage d'un pile souffre d'un problème majeur. Que se passe-t-il lorsque la zone mémoire allouée à la pile est remplie ? Dans note exemple, cela se produira si on veut encore ajouter trois éléments sur notre pile. Il n'y a pas de solution simple à ce problème. Pour le résoudre, il faut pouvoir déplacer la zone mémoire allouée à la pile pour la mettre dans une autre zone de la mémoire qui contient plus d'espace libre. La copie est assez facile à réaliser, mais il faut aussi modifier toutes les instructions du programme qui utilisent l'adresse de la pile puisque celle-ci change et cela c'est beaucoup plus difficile à réaliser. La pile utilisée par notre processeur évite ce problème car le processeur contient le registre `SP` et elle utilise le haut de la mémoire. Cette technique n'est pas utilisable dans un programme applicatif.

Une meilleure solution pour implémenter une pile de façon générique est d'utiliser des références. L'exemple ci-dessous montre comment construire une telle pile en python.

```
# Définition de la classe Node
class Node:
    def __init__(self, value):
```

(suite sur la page suivante)

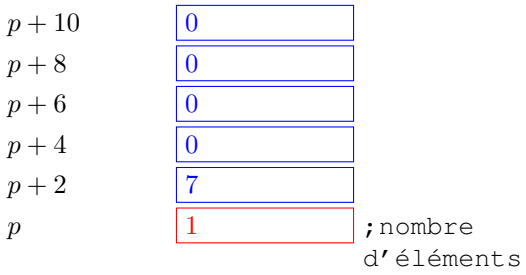



FIG. 6.5 – Stockage d'une pile dans un bloc de mémoire après exécution de ``pile.append(7)``

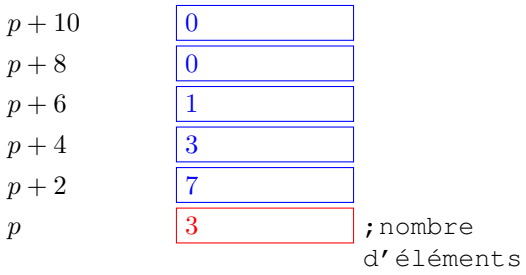



FIG. 6.6 – Stockage d'une pile dans un bloc de mémoire après exécution de ``pile.append(3)`` et ``pile.append(1)``

(suite de la page précédente)

```
        self.value = value
        self.next = None

# Définition de la classe Stack
class Stack:
    # Initialisation
    def __init__(self):
        self.top = None

    # Méthode pour empiler un élément sur la pile
    def push(self, value):
        new_node = Node(value)
        if self.top is None:
            self.top = new_node else:
            new_node.next = self.top
        self.top = new_node

    # Méthode pour dépiler un élément de la pile
    def pop(self):
        if self.top is None:
            return None
        else:
            popped_node = self.top
            self.top = self.top.next
            popped_node.next = None
            return popped_node.value

    # Méthode pour vérifier si la pile est vide
    def is_empty(self):
        return self.top is None
```

Dans cet exemple, nous utilisons une classe `Node` pour représenter chaque nœud de la liste chaînée, et une classe `Stack` pour encapsuler les opérations de la pile.

La méthode `push()` permet d'empiler un nouvel élément sur le dessus de la pile en créant un nouveau nœud et en modifiant les références pour pointer vers le nouveau nœud.

La méthode `pop()` permet de dépiler l'élément du dessus de la pile en ajustant les références pour pointer vers le nœud suivant.

La méthode `is_empty()` vérifie si la pile est vide en vérifiant si la référence *top* pointe vers `None`.

Cette implémentation peut s'utiliser par le fragment de code ci-dessous.

```
# Exemple d'utilisation de la pile
stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)
print(stack.pop()) # Résultat: 3
print(stack.pop()) # Résultat: 2
print(stack.is_empty()) # Résultat: False
```

Dans l'exemple d'utilisation, nous empilons les éléments 1, 2 et 3 sur la pile à l'aide de la méthode `push()`. Ensuite, nous dépilons les deux premiers éléments de la pile à l'aide de la méthode `pop()`. Finalement, nous utilisons la méthode `is_empty()` pour vérifier si la pile est vide.

Cette implémentation de la pile utilise une structure chaînée. Dans une pile, il est nécessaire de connaître à tout moment l'élément qui se trouve au sommet de la pile. Notre code python garde une référence vers le sommet de la pile via la variable `self.top`. Celle-ci a comme valeur `None` à la création de la pile ou lorsqu'elle est vide.

Dans de nombreux langages de programmation on appelle cette adresse un pointeur. Lorsque l'on créée une pile, celle-ci est vide et le pointeur du sommet de pile ne peut pas indiquer l'adresse d'un élément de la pile. En assembleur on utilise la valeur `NULL` pour indiquer un pointeur qui ne pointe vers rien. En mémoire, ce pointeur `NULL` correspondra à l'adresse `0`.

En assembleur, nous pouvons également stocker l'équivalent de l'information contenue dans chaque instance de la classe `Node`, c'est-à-dire :

— la valeur (le naturel) stockée en mémoire
— l'adresse de l'instance suivante de la classe `Node` sur la pile ou `NULL` si on est en fin de pile.

Nous utiliserons une notation pointée pour indiquer les deux parties d'un élément d'une pile. Si `e` est notre élément, alors `e_val` sera la valeur du naturel de cet élément et `e_ptr` contiendra l'adresse de l'élément suivant sur la pile. Sur base de cette notation, nous pouvons reprendre notre exemple en python et analyser comment les différents éléments sont stockés en mémoire. La pile est initialisée en plaçant la valeur `0`, correspondant au pointeur `NULL`, à l'adresse ($p$) correspondant au pointeur de sommet de pile. Ensuite, nous ajoutons `7` sur la pile avec l'opération `p.push(7)`. L'élément correspondant se trouve à l'adresse $x$ sur la Fig. 6.7. En assembleur, une telle structure chaînée peut être

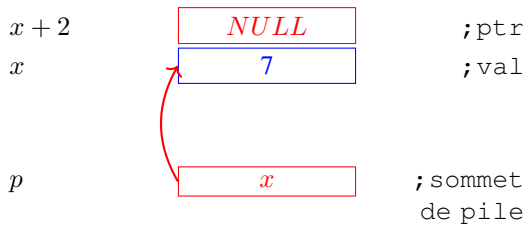



FIG. 6.7 – Stockage d'une pile dans une structure chaînée après exécution de ``pile.push(7)``

écrite en mémoire en utilisant les instructions suivantes.

```
p:      DB n1_val    ; le pointeur vers le sommet de la pile
n1_val: DB 7         ; le premier naturel stocké sur la pile
n1_ptr: DB 0         ; pointeur NULL, pas de successeur
```

La Fig. 6.8 représente l'état de la pile en mémoire après exécution de l'opération `p.push(9)` en supposant que l'élément correspondant soit stocké en mémoire à l'adresse $z$. En assembleur, une telle structure chaînée peut être

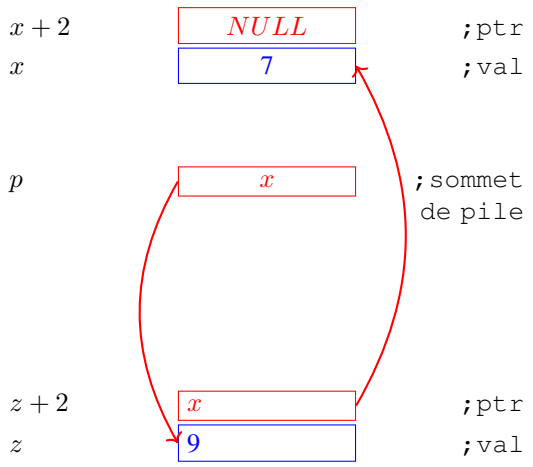



FIG. 6.8 – Stockage d'une pile dans une structure chaînée après exécution de ``pile.push(9)``

écrite en mémoire en utilisant les instructions suivantes.

```
p: DB n2_val         ; le pointeur vers le sommet de la pile
n1_val: DB 7         ; le deuxième naturel stocké sur la pile
n1_ptr: DB 0         ; pointeur NULL, pas de successeur
n2_val: DB 9         ; le premier naturel stocké sur la pile
n2_ptr: DB n1_val    ; pointeur vers le successeur
```

Après avoir récupéré la valeur `9` du sommet de la pile, on y ajoute ensuite deux éléments via les opérations `pile.push(3)` et `pile.push(1)`. A ce moment, la pile contient trois éléments comme représenté dans la Fig. 6.9. Cette structure chaînée peut facilement s'adapter aux stockages d'autres types de données que des naturels. A titre d'exemple, considérons des chaînes de caractères qui sont terminées par un marqueur de fin valant `0`. On peut facilement construire une pile de prénoms en conservant un pointeur de sommet de pile et en ayant dans chaque élément de la pile un pointeur vers la chaîne de caractères stockée et un pointeur vers l'élément suivant sur la pile.

A titre d'exemple, considérons la pile de prénoms suivante en python :

```
# Exemple d'utilisation de la pile
pile = Stack()
pile.push("Louise")
pile.push("Claire")
pile.push("Dominique")
```

Si la chaîne de caractères `Louise` est stockée à l'adresse $l$, la chaîne `Claire` à l'adresse $c$ et la chaîne `Dominique` à l'adresse $d$, alors en mémoire cette pile peut être organisée comme dans la Fig. 6.10. Pour ne pas alourdir la figure, seule la chaîne de caractères `Louise` est représentée dans la figure avec son marqueur de fin. Nous pouvons maintenant construire une implémentation en assembleur qui permet d'ajouter et de retirer un naturel d'une pile. Tout comme l'implémentation en python, notre implémentation en assembleur utilise des nœuds qui sont composés de deux zones mémoires contiguës de 16 bits chacune :

— `n_val` : le naturel stocké sur le pile
— `n_ptr` : un pointeur vers le successeur de l'élément sur la pile ou `NULL` (`0`) pour indiquer la fin de pile

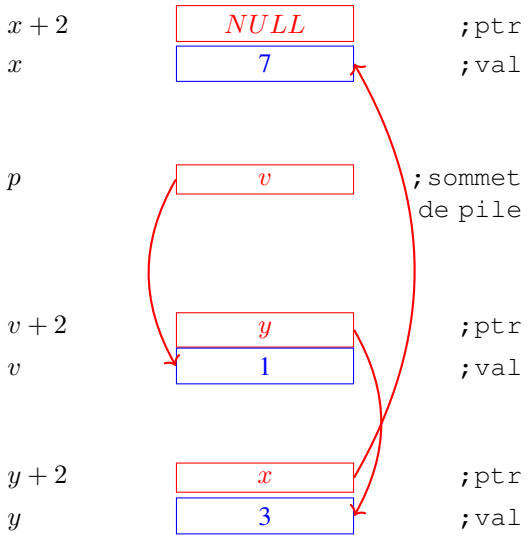



FIG. 6.9 – Stockage d'une pile dans une structure chaînée après exécution de ``pile.push(3)`` suivi de ``pile.push(1)``

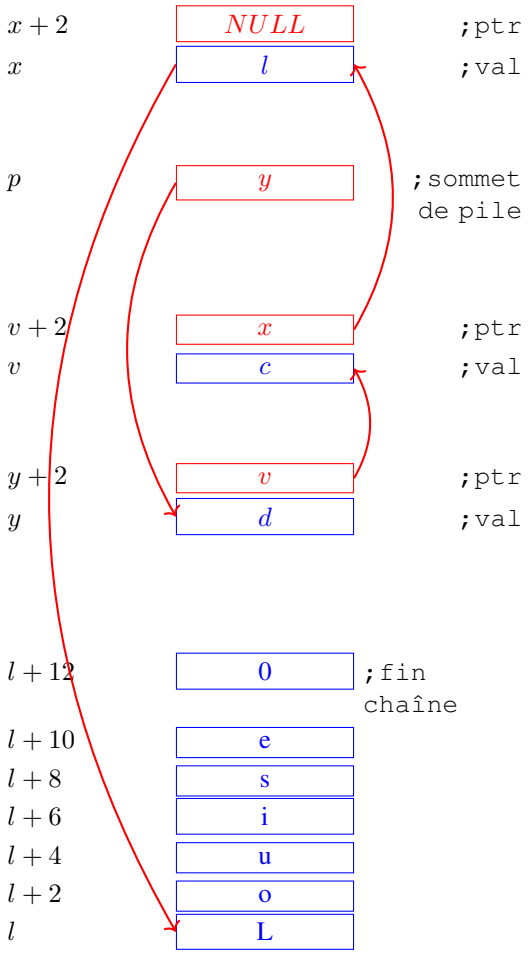



FIG. 6.10 – Stockage d'une pile dans une structure chaînée après exécution de pile.push(« Louise ») suivi de pile.push(« Claire ») et pile.push(« Dominique »)

Nous utilisons la variable `p` pour stocker un pointeur vers l'adresse du nœud qui se trouve au sommet de la pile (ou `NULL` si la pile est vide). Cette variable est initialisée à la valeur `0` puisque la pile est initialement vide.

Nous pouvons commencer par la fonction `push` qui permet d'ajouter un élément au sommet de la pile. Notre fonction `push` prend trois arguments :

- la valeur entière à ajouter sur la pile qui est placée dans le registre `D`
- l'adresse de la variable contenant l'adresse du sommet de la pile
- comme la fonction doit créer un nouveau nœud, nous devons aussi lui indiquer l'adresse mémoire de ce nouveau nœud

Cette fonction utilise les registres `B` et `C`. Ils sont donc sauvegardés sur la pile du processeur au début de la fonction. Durant l'exécution de la fonction `push`, la pile du programme contient donc les informations reprises en Fig. 6.11.

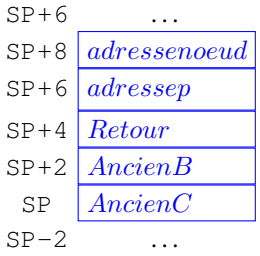



FIG. 6.11 – État de la pile pendant l'exécution de la fonction push

```
; push
; premier argument la valeur à ajouter dans D
; [SP+4] deuxième argument, l'adresse du sommet de la pile
; [SP+2] troisième argument, l'adresse du nœud à ajouter
push:
    PUSH B
    PUSH C
    MOV B, [SP+8]     ; adresse pointeur de pile, premier sur stack
    MOV C, [SP+6]     ; adresse (val) du nœud à ajouter
    ; ajout de la valeur
    MOV [C], D  ; sauvegarde dans le nouveau nœud
    ADD C, 2    ; adresse de l'élément _ptr du nœud
    MOV B, [B]  ; adresse de l'ancien sommet de la pile
    MOV [C], B  ; sauvegarde dans l'élément _ptr du nouveau nœud
    ; mise à jour du pointeur de somme de pile
    MOV C, [SP+6]
    MOV B, [SP+8]
    MOV [B], C
    POP B
    POP C
    RET
```

Nous pouvons ensuite implémenter la fonction `pop` qui retire l'élément se trouvant au sommet de la pile. Cette fonction prend un argument, l'adresse de la variable qui contient l'adresse du sommet de la pile. Le code de cette fonction est assez simple. Il utilise le registre `B` comme registre temporaire. Sa valeur est donc placée sur la pile au début de la fonction et récupérée à la fin. Au début de la fonction, nous devons d'abord tester si la pile est vide. C'est le cas si la variable qui stocke l'adresse du sommet de pile contient la valeur `NULL` (`0`). Notre fonction récupère ensuite la valeur se trouvant au sommet de la pile et met à jour le pointeur de sommet de pile passé en argument pour qu'il pointe vers le nœud se trouvant maintenant au sommet. Elle remet à zéro le nœud qui a été effacé.

```
pop:
    PUSH B
    PUSH C
    MOV A, [D] ; si pile vide, retourne 0
    CMP A, 0
    JE fin_pop
    MOV B, [D] ; adresse de l'élément au sommet de la pile
    MOV A, [B] ; valeur à retourner
    ADD B, 2   ; adresse de l'élément ptr du nœud
    MOV C, [B]
    MOV [D], C ; nouveau sommet de pile
    MOV [B], 0 ; mise à zéro de l'élément
    SUB B, 2 ; ptr est au-dessus de val
    MOV [B], 0 ; mise à zéro du pointeur
 fin_pop:
    POP C
    POP B
    RET
```

Pour tester ces deux fonctions, nous pouvons construire une petite pile en mémoire en utilisant les instructions `DB` à bon escient. Pour cela, il suffit de se rappeler qu'un nœud occupe deux blocs de 16 bits consécutifs en mémoire. L'exemple ci-dessous contient une pile contenant deux nœuds. Celui du sommet contient la valeur `3` et son pointeur indique comme successeur le nœud se trouvant à l'adresse `n1_val` qui contient la valeur `7`. Ce second nœud n'a pas de successeur. Les nœuds `n3`, `n4` et `n5` sont vides.

```
JMP start:
p: DB n2_val ; pile
n1_val: DB 7
n1_ptr: DB 0
n2_val: DB 3
n2_ptr: DB n1_val
n3_val: DB 0
n3_ptr: DB 0
n4_val: DB 0
n4_ptr: DB 0
n5_val: DB 0
n5_ptr: DB 0
```

Sur cette pile, on peut faire appel à la fonction `pop` en lui passant l'adresse de la variable `p` comme argument dans le registre `D`.

```
; exemple d'appel à pop
MOV D, p
CALL pop
```

L'appel à la fonction assembleur `push` est un peu plus compliqué puisqu'il faut lui passer l'entier à ajouter, l'adresse du sommet de la pile et l'adresse d'un nœud vide. L'exemple ci-dessous ajoute la valeur `42` sur notre pile.

```
; ajout de la valeur 42 sur la pile
MOV D, 42
PUSH p
PUSH n5_val
CALL push
```

---

**Note :** Gestion de la mémoire

En python, lorsque l'on écrit `new_node = Node(value)`, on réserve une zone mémoire pour stocker le nouveau

nœud. Cela se fait en appelant une fonction de gestion de la mémoire qui sort du cadre de ce cours. C'est pour cette raison que notre fonction `push`, et d'autres exemples que nous verrons ensuite, reçoivent l'adresse de la zone mémoire à utiliser. Vous verrez dans d'autres cours comment il est possible d'écrire des programmes pour gérer la mémoire. De la même façon, la fonction `pop` devrait libérer la mémoire du nœud qu'elle retire de la pile afin que celle-ci soit disponible pour d'autres parties du programme.

## 6.1 Liste chaînée

Nous pouvons maintenant construire une liste chaînée et écrire quelques fonctions pour manipuler une telle liste. Nous choisissons d'utiliser trois blocs de 16 bits consécutifs pour stocker les informations suivantes sur la liste :

— le nombre d'éléments dans la liste (`len`)
— l'adresse du dernier élément de la liste (`tail`, mis à `NULL` si la liste est vide)
— l'adresse du premier élément de la liste (`head`, mis à `NULL` si la liste est vide)

La Fig. 6.12 représente ces trois blocs de données en mémoire. Ce bloc de mémoire peut être initialisé par la fonction


FIG. 6.12 – Entête de la liste initialisée

`init_list` qui prend comme argument l'adresse du bloc.

```
; initialise une liste
; D: adresse du bloc de mémoire
init_list:
   MOV [D], 0   ; initialisation du pointeur du premier élément
   MOV [D+2], 0 ; initialisation du pointeur du dernier élément
   MOV [D+4], 0 ; initialisation de l'indication de longueur
   RET
```

Un nœud de notre liste contiendra deux éléments :

— la valeur stockée (`val`)
— le pointeur vers le nœud successeur (`next`)

La Fig. 6.13 représente un nœud de notre liste contenant la valeur `17`. Ce nœud est le dernier de la liste puisqu'il n'a pas de successeur. Nous pouvons maintenant visualiser comment une telle liste peut être stockée en mémoire. La Fig. 6.14


FIG. 6.13 – Élément de la liste contenant la valeur 17 et Entête de la liste initialisée

et la Fig. 6.15 représentent deux organisations en mémoire possible d'une liste de deux éléments contenant la valeur `42` suivie par la valeur `17`. Tout comme nous l'avons fait dans la section précédente pour tester notre implémentation des fonctions de manipulation d'une pile, nous pouvons facilement construire en mémoire une liste chaînée telle que celle représentée en Fig. 6.15.

```
n1_val: DB 17
n1_next: DB 0
l_head: DB n2_val
```

(suite sur la page suivante)

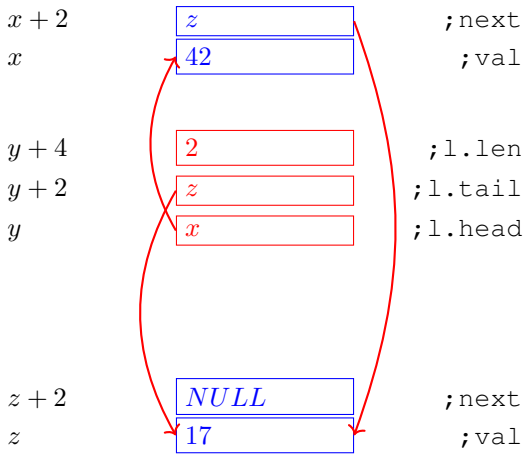



FIG. 6.14 – Représentation en mémoire d'une liste contenant la valeur ``42`` suivie de ``17``

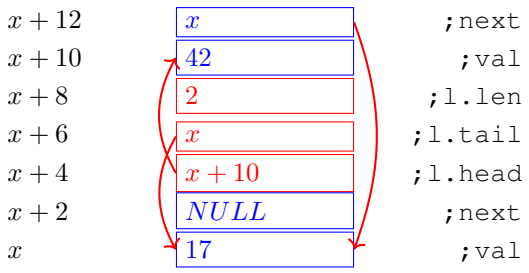



FIG. 6.15 – Une autre organisation possible de la liste contenant ``42`` suivi de ``17`` en mémoire

(suite de la page précédente)

```
l_tail: DB n1_val
l_len: DB 2
n2_val: DB 42
n2_next: DB n1_val
```

Nous utiliserons cette structure de liste pour implémenter plusieurs fonctions. La première, baptisée `add_head` ajoute un nouvel entier en début de liste. Cette fonction prend trois arguments :

— l'adresse de la structure contenant la longueur de la liste et les deux pointeurs vers le début et la fin de la liste (dans le registre `D`)
— la valeur à ajouter (sur la pile, `SP+4`)
— l'adresse d'un nœud vide (sur la pile, `SP+2`)

La Fig. 6.16 présente graphiquement l'ajout d'un nœud dans une telle liste.

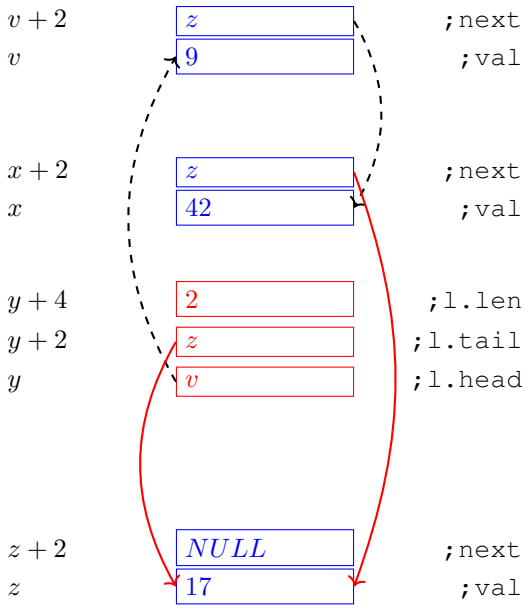



FIG. 6.16 – Ajout du nœud contenant la valeur ``9`` en tête de liste

```
; Ajout d'un nouvel élément en tête de liste
; D: adresse du descripteur de liste
; [SP+4]: valeur à ajouter
; [SP+2]: adresse du nœud vide à utiliser
; retourne dans A l'adresse du nœud ajouté
add_head:
   PUSH B ; sauvegarde
   PUSH C ; sauvegarde
   MOV A, [SP+6] ; adresse nœud à ajouter
   MOV C, [SP+8] ; valeur à ajouter
   MOV [A], C ; valeur placée dans le nœud à ajouter
   MOV B, [D] ; adresse du premier nœud de l'ancienne liste
   MOV C, [SP+6] ; adresse du nœud à ajouter
   ADD C, 2  ; C contient l'adresse de l'élément next du nouveau nœud
   MOV [C], B ; next pointe vers l'ancien premier nœud
   MOV  C, [SP+6] ; adresse du nouveau nœud
   MOV [D], C     ; descripteur head pointe vers le nouveau nœud
```

(suite sur la page suivante)

(suite de la page précédente)

```
   MOV B, [D+4] ; adresse de len dans le descripteur
   INC B
   MOV [D+4], B ; sauvegarde en mémoire
   POP C ; récupération
   POP B ; récupération
   RET
```

De la même façon, on pourra facilement écrire une fonction `add_tail` qui ajoute un élément en fin de liste en utilisant le pointeur de fin de liste. La Fig. 6.17 présente graphiquement l'ajout d'un nœud en fin de liste. Regardons

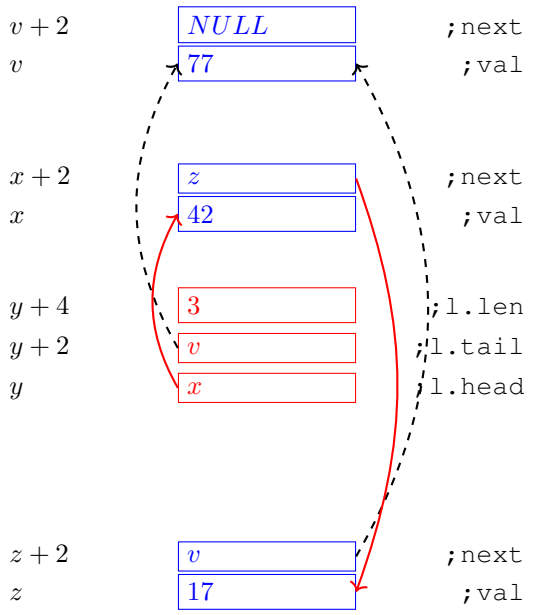



FIG. 6.17 – Ajout du nœud 77 en fin de liste

maintenant comment parcourir cette liste. Le parcours d'une liste est une opération importante sur les listes. Comme exemple, considérons la fonction `sum` qui calcule la somme de tous les éléments présents dans une liste. Cette fonction prend un seul argument dans le registre `D`, l'adresse du descripteur de liste. Elle retourne la somme calculée dans le registre `A`.

Cette fonction commence par vérifier si la liste est vide. Pour cela, elle regarde si le pointeur `head` vaut zéro (adresse `NULL`). Dans ce cas, elle retourne la valeur zéro dans le registre `A`. Ensuite, elle parcoure la liste est bouclant tant que le pointeur `next` des éléments parcourus est différent de `NULL` et accumule la somme des éléments dans le registre `A`.

```
; calcul de la somme des éléments d'une liste
; D: adresse du descripteur de liste
sum:
   ; si la liste est vide, retourne 0
   MOV A, [D] ; D est l'adresse de head
   CMP A, 0
   JNE suite
   RET
suite:
   PUSH B
   MOV A, 0
   MOV B, [D] ; adresse du premier nœud
boucle:
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ADD A, [B] ; valeur du premier nœud
    ADD B, 2 ; adresse du pointeur next
    MOV B, [B] ; pointeur next
    CMP B, 0
    JNE boucle
    POP B
    RET
```

On peut bien entendu construire d'autres opérations sur de telles structures chaînées. Plusieurs exemples vous serons présentés durant les travaux pratiques.

CHAPITRE 7

# Logique booléenne

Le fonctionnement des ordinateurs s'appuie sur quelques principes très simples, mais qui sont utilisés à une très grande échelle. Le premier principe est que toute l'information peut s'encoder sous une forme binaire, c'est-à-dire une suite de bits. Un bit est l'unité de représentation de l'information. Un bit peut prendre deux valeurs :

— `0`
— `1`

On peut associer une signification à ces bits. Il est par exemple courant de considérer que le bit `0` représente la valeur *Faux* tandis que le bit `1` représente la valeur *Vrai*. C'est une convention qui est utile dans certains cas, mais n'est pas toujours nécessaire et peut parfois porter à confusion.

Avec ces deux valeurs booléennes, il est intéressant de définir des opérations. Une opération booléenne est une fonction qui prend en entrée 0, 1 ou plusieurs bits et retourne un résultat.

## 7.1 Fonctions booléennes

La fonction la plus simple est la fonction identité. Elle prend comme entrée un bit et retourne la valeur de ce bit. On peut la définir en utilisant une *table de vérité* qui indique la valeur du résultat de la fonction pour chaque valeur possible de son entrée. Dans la table ci-dessous, la colonne *x* contient les différentes valeurs possibles de l'entrée *x* et la valeur du résultat pour chacune des valeurs possibles de *x*.

| x | identité(x) |
|---|---|
| 0 | 0 |
| 1 | 1 |

Cette fonction n'est pas très utile en pratique. Elle nous permet d'illustrer une table de vérité simple dans laquelle il y a une valeur binaire en entrée et une valeur binaire également en sortie.

Une fonction plus intéressante est l'inverseur, aussi dénommée *NOT* en anglais. Cette fonction prend comme entrée un bit. Si le bit d'entrée vaut *1*, elle retourne *0*. Tandis que si le bit d'entrée vaut *0*, elle retourne *1*. Cette fonction sera très fréquemment utilisée pour construire des circuits électroniques utilisés dans les ordinateurs.

| x | NOT(x) |
|---|---|
| 0 | 1 |
| 1 | 0 |

Il y a encore deux fonctions que l'on peut construire avec une seule entrée binaire. La première, baptisée *Toujours0*, retourne toujours la valeur `0`, quelle que soit son entrée. La seconde, baptisée *Toujours1* retourne toujours la valeur `1`. Voici leurs tables de vérité.

| x | Toujours0(x) |
|---|---|
| 0 | 0 |
| 1 | 0 |

| x | Toujours1(x) |
|---|---|
| 0 | 1 |
| 1 | 1 |

La logique booléenne devient nettement plus intéressante lorsque l'on considère des fonctions qui prennent plus d'une entrée.

### 7.1.1 Fonctions booléennes à deux entrées

Plusieurs fonctions booléennes classiques existent. Les premières correspondent à la conjonction (*et*) et à la disjonction (*ou*) en logique. Commençons par la fonction *AND*. Celle-ci correspond à la table de vérité suivante :

| x | y | AND(x,y) |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Cette table comprend quatre lignes qui correspondent à toutes les combinaisons possibles des deux entrées de la fonction. On remarque aisément que la fonction *AND(x,y)* retourne la valeur `1` uniquement lorsque ses deux entrées ont la valeur `1`. Si une des deux entrées de la fonction *AND(x,y)* a la valeur `0`, alors sa sortie est nécessairement `0`. Cette fonction est bien l'équivalent de la conjonction logique si l'on applique la convention que `0` représente la valeur *Faux*.

La fonction *OR(x,y)*, quant à elle, est l'équivalent de la disjonction logique. Sa table de vérité est reprise ci-dessous.

| x | y | OR(x,y) |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

On remarque aisément que la fonction *OR(x,y)* correspond bien à la disjonction logique lorsque *1* représente la valeur *Vrai*. Cette fonction *OR(x,y)* ne retourne la valeur `0` que si ses deux entrées valent `0`. Dans tous les autres cas, elle retourne la valeur `1`.

Ces fonctions peuvent être combinées entre elles. Un premier exemple est d'appliquer un inverseur (opération *NOT* au résultat de la fonction *AND*). Cette fonction booléenne s'appelle généralement *NAND* (*NOT AND*) et sa table de vérité est la suivante. On pourra dire que $NAND(x, y) \iff NOT(AND(x, y))$.

| x | y | NAND(x,y) |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

De même, la fonction *NOR* s'obtient en inversant le résultat de la fonction *OR*. On pourra dire que $NOR(x, y) \iff NOT(OR(x, y))$.

| x | y | NOR(x,y) |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

Il est important de noter que *NOR(x,y)* n'est pas équivalent à la fonction *OR(NOT(x),NOT(y))*. La table de vérité de cette dernière fonction est reprise ci-dessous.

| x | y | NOT(x) | NOT(y) | OR(NOT(x),NOT(y)) |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |

Il existe d'autres fonctions booléennes à deux entrées qui sont utiles en pratique. Parmi celles-ci, on retrouve la fonction *XOR(x,y)* qui retourne la valeur *1* uniquement si une seule de ses entrées a la valeur *1*. Sa table de vérité est reprise ci-dessous. On remarquera qu'elle diffère de celle des autres fonctions booléennes que nous avons déjà présenté.

| x | y | XOR(x,y) |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

### 7.1.2 Algèbre booléenne

Ces fonctions booléennes ont des propriétés importantes que l'on peut facilement démontrer en utilisant des tables de vérité.

- $AND(1, x) \iff x$
- $AND(0, x) \iff 0$
- $OR(1, x) \iff 1$
- $OR(0, x) \iff x$

A titre d'exemple, regardons la table de vérité de la dernière propriété :

| x | 0 | OR(0,x) |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

Dans certains cas, on peut être amené à appliquer une fonction booléenne à deux entrées identiques ou l'une inverse de l'autre. En utilisant les tables de vérité, on peut aisément démontrer que :

— $AND(x,x) \iff x$
— $OR(x,x) \iff x$
— $AND(NOT(x),x) \iff 0$
— $OR(NOT(x),x) \iff 1$

A titre d'exemple, regardons la table de vérité de la dernière propriété :

| x | NOT(x) | OR(NOT(x),x) |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 0 | 1 |

Les opérations *AND* et *OR* sont commutatives et associatives comme les opérations arithmétiques d'addition et de multiplication.

— $AND(x,y) \iff AND(y,x)$ (commutativité)
— $OR(x,y) \iff OR(y,x)$ (commutativité)
— $AND(x,AND(y,z)) \iff AND(AND(x,y),z)$ (associativité)
— $OR(x,OR(y,z)) \iff OR(OR(x,y),z)$ (associativité)

Ces lois d'associativité sont importantes car elles vont nous permettre de facilement construire des fonctions booléennes qui prennent un nombre quelconque d'entrées en utilisant des fonctions à deux entrées comme briques de base.

La distributivité est une autre propriété qui relie les fonctions *AND* et *OR*.

— $AND(x,OR(y,z)) \iff OR(AND(x,y),AND(x,z))$ (distributivité)
— $OR(x,AND(y,z)) \iff AND(OR(x,y),OR(x,z))$ (distributivité)

Lorsque l'on ajoute la fonction *NOT*, on obtient deux autres propriétés utiles en pratique.

— $AND(x,OR(NOT(x),y)) \iff AND(x,y)$
— $OR(x,AND(NOT(x),y)) \iff OR(x,y)$

Enfin, les trois opérations *AND*, *OR* et *NOT* sont reliées entre elles par les lois de *De Morgan*. On peut facilement démontrer, par exemple en utilisant des tables de vérité, que :

— *NOT( OR(x,y) ) = AND ( NOT(x), NOT(y) )*
— *NOT( AND(x,y) ) = OR ( NOT(x), NOT(y) )*

Ce lois sont très utiles lorsque l'on doit manipuler des fonctions booléennes.

### 7.1.3 Fonctions booléennes à plus de deux entrées

En utilisant l'associativité, on peut facilement construire des fonctions à plus de deux entrées. Ainsi, la fonction *AND* à trois entrées $AND(x,y,z) \iff AND(x,AND(y,z)) \iff AND(AND(x,y),z)$. Sa table de vérité est sans surprise la suivante.

| x | y | z | AND(x,y,z) |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 |

De la même façon, on peut obtenir la fonction *OR* à plus de deux entrées : $OR(x,y,z) \iff OR(x,OR(y,z)) \iff OR(OR(x,y),z)$.

En plus de ces fonctions booléennes classiques, il est possible de construire deux autres fonctions qui sont très utiles en pratique. La première est le multiplexeur qui permet de « sélectionner » une valeur d'entrée. La table de vérité du multiplexeur est reprise ci-dessous.

| x | y | sel | out |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 |

On remarque aisément que la sortie du multiplexeur dépend de l'entrée marquée *sel* (pour sélecteur). Lorsque *sel* vaut *0*, la sortie du multiplexeur est égale à sa première entrée (*x*). Lorsque *sel* vaut *1*, sa sortie est égale à sa seconde entrée (*y*). On peut résumer ceci avec la table de vérité ci-dessous :

| sel | out |
|---|---|
| 0 | x |
| 1 | y |

La fonction duale du multiplexeur est le démultiplexeur. Un démultiplexeur a deux entrées, *in* et *sel* et deux sorties, *x* et *y*. Son comportement est le suivant :

— lorsque l'entrée *sel* vaut *0*, alors la sortie *x* a la même valeur que l'entrée *in* tandis que la sortie *y* vaut *0*
— lorsque l'entrée *sel* vaut *1*, alors la sortie *y* a la même valeur que l'entrée *in* tandis que la sortie *x* vaut *0*

La table de vérité correspondant au démultiplexeur est présentée ci-dessous.

| in | sel | x | y |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

Tant le multiplexeur que le démultiplexeur peuvent s'implémenter en utilisant des portes *AND*, *OR* et des inverseurs. Prenons comme exemple le multiplexeur. Nous verrons dans la section suivante qu'il est possible de l'implémenter en utilisant une fonction *OR* à quatre entrées et des fonctions *AND* à trois entrées.

## 7.2 Synthèse de fonctions booléennes

L'intérêt des fonctions booléennes est qu'il est possible de concevoir des fonctions booléennes pour supporter n'importe quelle table de vérité. Prenons comme exemple la fonction *XOR* qui retourne *1* lorsque ses deux entrées sont différentes et *0* sinon. Sa table de vérité est reprise ci-dessous.

| x | y | XOR(x,y) |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Pour réaliser une telle fonction, il suffit de se trouver une combinaison de fonctions *AND*, *OR* et *NOT* qui produit la même table de vérité. Une façon mécanique de produire cette fonction est de remarquer que la sortie d'une fonction *AND* ne vaut *1* que lorsque ses deux entrées sont à *1*. Examinons la deuxième ligne de la table de vérité de la fonction *XOR*. Celle-ci indique que cette fonction doit valoir *1* lorsque *x* vaut *0* et *y* vaut *1*. Avec des fonctions *AND* et des inverseurs, on peut obtenir les tables de vérité suivantes :

| x | y | AND(x,y) |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| x | y | AND(NOT(x),y) |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

| x | y | AND(x,NOT(y)) |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| x | y | AND(NOT(x),NOT(y)) |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

Deux de ces fonctions *AND* peuvent être combinées avec une fonction *OR*. Un premier exemple est de combiner les deux premières fonctions, *AND(x,y)* et *AND(NOT(x),y)* pour construire la fonction *OR(AND(x,y),AND(NOT(x),y)*. Sa table de vérité est la suivante.

| x | y | OR(AND(x,y),AND(NOT(x),y) |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

On remarque aisément que la fonction combinée vaut *1* uniquement lorsque *x* vaut *1* et *y* vaut *1* ou lorsque *x* vaut *0* et *y* vaut *1*.

**En revenant à notre fonction *XOR*, on se rend aisément compte qu'elle doit valoir *1* dans uniquement deux cas :**

— *x* vaut *1* et *y* vaut *0*
— *x* vaut *0* et *y* vaut *1*

Dans tous les autres cas, la fonction *XOR* doit retourner *0*. Le premier cas peut s'implémenter en utilisant la fonction *AND(x,NOT(y))* tandis que le second correspond à la fonction *AND(NOT(x),y)*. Ces deux fonctions peuvent se combi-

ner comme suit : *OR(AND(x,NOT(y)), AND(NOT(x),y))*. En construisant la table de vérité, on se convainc facilement que $OR(AND(x, NOT(y)), AND(NOT(x), y)) \iff XOR(x, y)$.

En pratique, il est possible de construire n'importe quelle fonction booléenne en combinant avec la fonction *OR*, autant de fonctions *AND* qu'il y a de lignes de la table de vérité dont la sortie vaut *1*.

A titre d'exemple, considérons la fonction *F* dont la table de vérité est reprise ci-dessous.

| x | y | F(x,y) |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Cette fonction peut s'implémenter comme étant la combinaison des trois fonctions *AND* suivantes :

— *AND(NOT(x),NOT(y))*
— *AND(NOT(x),y)*
— *AND(x,NOT(y))*

Et donc, $OR(AND(NOT(x), NOT(y)), AND(NOT(x), y), AND(x, NOT(y)) \iff F(x, y)$. Cependant, cette implémentation n'est pas la plus efficace du point de vue du nombre de fonctions *AND*. Il y a d'autres réalisations possibles. Une première implémentation équivalente est de remarquer que lorsque *x* vaut *0*, la fonction *F(x,y)* vaut toujours *1*. On peut donc simplifier cette fonction comme étant *OR(NOT(x), AND(x,NOT(y))*. On peut aisément se rendre compte que cette fonction booléenne a la même table de vérité que la fonction *F(x,y)*. Mathématiquement, on peut noter que $OR(AND(NOT(x), NOT(y)), AND(NOT(x), y)) \iff NOT(x)$.

Cette implémentation de la fonction *F(x,y)* n'est pas la plus compacte. On remarque aisément que cette fonction vaut *0* uniquement lorsque ses deux entrées valent *1*. Dans tous les autres cas, elle vaut *1*. Cela nous rappelle la fonction *NAND* ou $NOT(AND(x, y)) \iff F(x, y)$.

Dans le cadre de ce cours, nous nous focaliserons sur la synthèse de fonctions booléennes qui sont correctes, c'est-à-dire qui produisent une table de vérité donnée, mais qui n'utilisent pas nécessairement un nombre minimal de fonctions de base. Différentes techniques existent pour minimiser de telles fonctions booléennes, mais elles correspondent plus à un cours d'électronique digitale qu'à un cours d'introduction au fonctionnement des ordinateurs.

## 7.3 Représentations graphiques

Lorsque l'on travaille avec des fonctions booléennes, on peut soit utiliser les symboles comme *AND*, *OR*, *NOT*, soit utiliser des symboles graphiques. Ceux-ci sont très utilisés pour construire de petits circuits. La Fig. 7.1 représente l'inverseur ou la fonction *NOT*. La fonction *OR* est présentée schématiquement dans la Fig. 7.2 et la fonction *AND* dans la Fig. 7.3. La fonction *XOR* a aussi sa représentation graphique. Celle-ci est présentée dans la Fig. 7.4. Dans

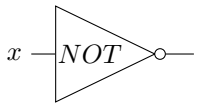



FIG. 7.1 – Représentation graphique d'une fonction NOT

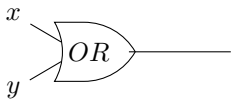



FIG. 7.2 – Représentation graphique d'une fonction OR

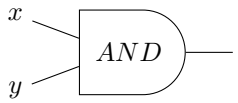



FIG. 7.3 – Représentation graphique d'une fonction AND

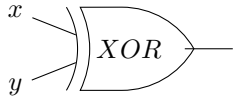



FIG. 7.4 – Représentation graphique d'une fonction XOR

de nombreux circuits, on retrouve des inverseurs. Ainsi, la fonction *NAND* est finalement une fonction *AND* suivie d'un inverseur comme représenté sur la Fig. 7.5. Cette inversion est symbolisée par un petit rond. Il en va de même pour la fonction *NOR* (Fig. 7.6). Les multiplexeurs et démultiplexeurs ont aussi leur représentation graphique. Le

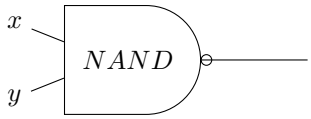



FIG. 7.5 – Représentation graphique d'une fonction NAND

livre les représente en utilisant un triangle comme dans la Fig. 7.7. De la même façon, on peut également représenter le démultiplexeur de façon graphique comme représenté dans la Fig. 7.8. Il est évidemment possible de combiner plusieurs fonctions booléennes pour supporter des fonctions plus avancées. A titre d'exemple, considérons la fonction d'égalité qui vaut *1* lorsque ses deux entrées sont égales et *0* sinon. Voici sa table de vérité.

| x | y | EQ(x,y) |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Cette fonction peut être réalisée en utilisant deux fonctions *AND*, une fonction *OR* et des inverseurs (Fig. 7.9). Un autre exemple est la fonction *XOR* dont nous avons déjà parlé précédemment. Celle-ci peut s'implémenter en utilisant deux inverseurs, deux fonctions *AND* et une fonction *OR* comme représenté dans la Fig. 7.10. Avec un multiplexeur, il est possible de construire un circuit « programmable » qui, en fonction de la valeur de son entrée *sel*, calcule soit la fonction *AND*, soit la fonction *OR*. Ce circuit est représenté dans la Fig. 7.11.

## 7.4 Un langage de description de circuits logiques

Les représentations graphiques sont très utiles pour permettre à des électroniciens de discuter de circuits électroniques, mais de nos jours ils travaillent généralement en utilisant des langages informatiques qui permettent de décrire ces circuits électroniques sous la forme de commandes. L'avantage de ces langages est qu'ils peuvent facilement être utilisés dans des logiciels de simulation ou d'analyse de circuits. C'est ce que nous ferons dans le cadre de ce cours avec le langage HDL proposé par les auteurs du livre Building a Modern Computer from First Principles.

Il existe de nombreux langages qui permettent de décrire de façon précise des fonctions booléennes et des circuits électroniques de façon générale [1] . Une description détaillée de ces langages sort du cadre de ce cours. Nous nous

1. Voir par exemple https://en.wikipedia.org/wiki/Hardware_description_language

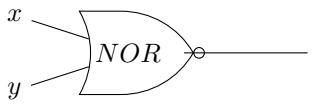



FIG. 7.6 – Représentation graphique d'une fonction NOR

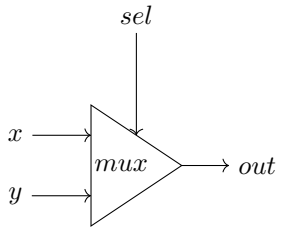



FIG. 7.7 – Un multiplexeur à deux entrées

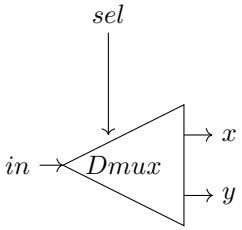



FIG. 7.8 – Un démultiplexeur à deux sorties

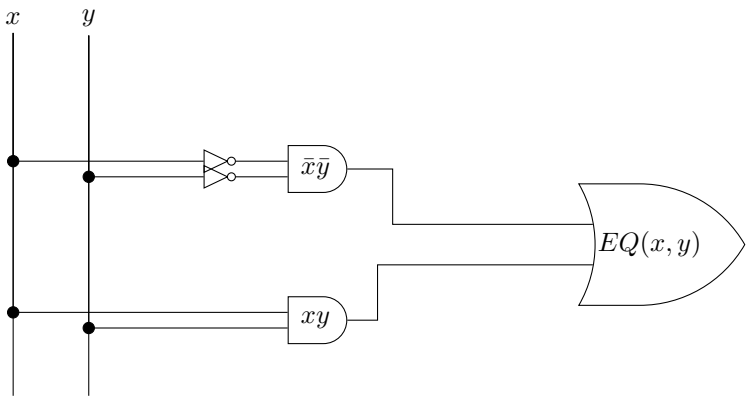



FIG. 7.9 – Représentation graphique d'un circuit qui réalise la fonction EQ

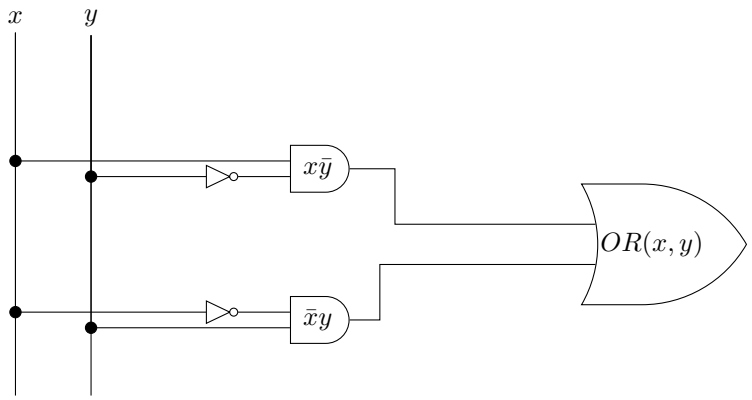



FIG. 7.10 – Représentation graphique d'un circuit qui réalise la fonction XOR

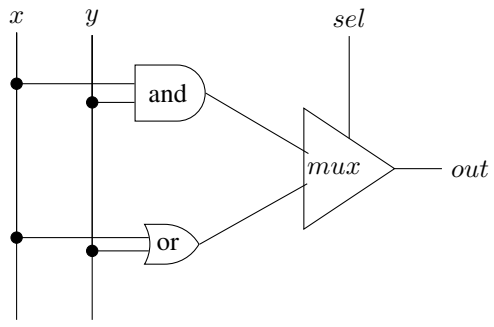



FIG. 7.11 – Un circuit programmable

contenterons de voir celui qui est utilisé par les simulateurs du livre de référence.

**Quatre types de fichiers sont utilisés par le simulateur :**

- les fichiers de description de circuits (nom de fichier se terminant par `.hdl`)
- les fichiers qui définissent les tests à réaliser sur les circuits (nom de fichier se terminant par `.tst`)
- les fichiers contenant les sorties d'un circuit obtenues lors de l'exécution d'un fichier de test (nom de fichier se terminant par `.out`)
- les fichiers contenant les sorties attendues d'un circuit (nom de fichier se terminant par `.cmp`)

Le langage de description de circuits permet de construire des fonctions booléennes en réutilisant les fonctions de base. Ce langage s'utilise un peu comme un langage de programmation. Dans le langage HDL, un circuit est défini sous la forme d'une liste de commandes, avec généralement une commande par ligne.

Comme dans tout langage de programmation, HDL permet d'inclure des commentaires. HDL utilise une convention similaire à des langages de programmation tels que C ou Java. En HDL, il y a deux façons de définir un commentaire. La première est d'utiliser les caractères *//*. Tous les caractères qui suivent *//* sur une ligne sont un commentaire qui ne sera pas lu par le simulateur. Il est aussi possible d'écrire de longs commentaires qui couvrent plusieurs lignes. Dans ce cas, le commentaire débute par les caractères */** et couvre tout le texte jusqu'à **/*. Le texte ci-dessous présente ces deux types de commentaires.

```
// Un commentaire sur une seule ligne

 /*
  * un commentaire sur plusieurs lignes
  */
```

Le langage HDL comprend différents mots-clés que l'on retrouve dans toute description de circuits. Le premier est le mot clé *CHIP* qui permet donner un nom au circuit électronique que l'on décrit dans le fichier. Il est préférable d'utiliser comme nom du circuit le même nom que celui du fichier. Le livre recommande d'utiliser un nom commençant par une majuscule pour les circuits que l'on crée. La définition d'un circuit commence après l'accolade ouvrante (*{*) et se termine à l'accolade fermante (*}*).

```
/*
 * Commentaire expliquant ce que fait le circuit
 */
CHIP Nom {
  // définition complète du circuit
}
```

A l'intérieur de la définition d'un circuit, on peut utiliser différents mots-clés :

- *IN* permet de lister un ensemble d'entrées
- *OUT* permet de lister un ensemble de sorties

Ces deux mots-clés sont utilisés au début de la description d'un circuit. Chaque entrée et chaque sortie doit avoir un nom différent. Par convention, on utilisera un nom écrit en minuscules et commençant par une lettre pour les entrées et les sorties. Les noms des entrées/sorties doivent être séparés par des virgules et la liste des entrées/sorties doit se terminer par un point-virgule (*;*).

```
IN a,b,c;  // Trois entrées appelés a, b et c
OUT out1, out2; // Deux entrées baptisées out1 et out2
```

Après avoir spécifié les entrées/sorties, il faut indiquer les différentes fonctions qui sont utilisées par le circuit. Le mot-clé *PARTS :* marque le début de la définition des fonctions logiques. L'exemple ci-dessous présente un squelette de circuit en HDL.

```
// Un commentaire
CHIP Nom {   // Le nom du circuit doit être le même que le nom du fichier
   IN ...  // les entrées du circuit
```

(suite sur la page suivante)

(suite de la page précédente)

```
   OUT ... // les sorties du circuit

   PARTS:   // les composantes du circuit
     // description des différentes parties du circuit
} // marque la fin de la définition du circuit Nom
```

HDL peut être utilisé pour construire de nombreuses fonctions booléennes en s'appuyant sur les fonctions existantes. Le simulateur supporte différentes fonctions de base dont :

— la fonction *Nand* qui est la fonction primitive pour de très nombreux circuits électroniques
— la fonction *And*
— la fonction *Or*
— la fonction *Not* ou l'inverseur

En utilisant l'inverseur, il est possible de construire un circuit électronique qui ne fait rien du tout avec deux inverseurs. Ce circuit prend une entrée nommée *a* et la connecte à un inverseur. La sortie de cet inverseur a comme nom *nota*. Elle est connecté à l'entrée du second inverseur.

```
// un circuit qui ne fait rien
CHIP Rien {
    IN a;      // Le circuit a une entrée que l'on nomme a dans ce fichier
    OUT out;  // Le circuit a une sortie que l'on nomme out dans ce fichier
    //
    PARTS:
    Not(in=a, out=nota);   // premier inverseur connecté à l'entrée a, sa sortie est␣
↪appelée nota
    Not(in=nota, out=out); // second inverseur connecté à la sortie du premier, sa␣
↪sortie est reliée à out
}
```

Graphiquement, ce circuit peut être représenté comme dans la Fig. 7.12. Un autre exemple est de construire un circuit

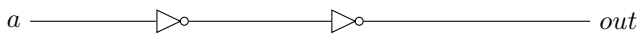



FIG. 7.12 – Représentation graphique du circuit qui ne fait rien

qui implémente la fonction *AND* avec trois entrées en utilisant des fonctions *AND* à deux entrées.

```
/*
 * Une circuit AND à trois entrées
 */
CHIP And3 {
    IN a,b,c;   // Les trois entrées
    OUT out;    // La sortie du circuit
    //
    PARTS:
    And(a=a, b=b, out=and1);   // première fonction AND
    And(a=and1, b=c, out=out); // seconde fonction AND
}
```

Un exemple plus complexe est de construire une implémentation de la fonction *XOR* sur base des fonctions *AND*, *OR* et *NOT*.

```
/*
 * Une circuit XOR à deux entrées
 */
CHIP Xor {
```

(suite sur la page suivante)

(suite de la page précédente)

```
    IN a,b;
    OUT out;

    PARTS:
    Not(in=a, out=nota);
    Not(in=b, out=notb);
    And(a=a, b=notb, out=w1);
    And(a=nota, b=b, out=w2);
    Or(a=w1, b=w2, out=out);
}
```

Les fichiers *HDL* contiennent la description du circuit électronique. Ils seront utilisés pour les différents projets de ce cours. Outre le langage HDL, le simulateur proposé dans le livre de référence supporte également un langage qui permet de définir les tests que chaque circuit doit supporter. Ces tests sont très importants car ils définissent de façon précise les sorties attendues de chaque circuit. Prenons comme exemple les tests pour la fonction *NOT*. Ceux-ci sont définis dans le fichier *Not.tst* du premier projet. La fonction *Not* a une entrée baptisée *in* et une sortie baptisée *out*.

```
// This file is part of www.nand2tetris.org
// and the book "The Elements of Computing Systems"
// by Nisan and Schocken, MIT Press.
// File name: projects/01/Not.tst

load Not.hdl,                    // charge la description de l'inverseur
output-file Not.out,             // les valeurs de la sortie out sont sauvées dans le␣
↪fichier Not.out
compare-to Not.cmp,              // les valeurs de la sortie out seront comparées au␣
↪contenu du fichier Not.cmp
output-list in%B3.1.3 out%B3.1.3; // format des données dans le fichier de sortie

set in 0,      // pour ce test, on fixe la valeur de in à 0
eval,          // on exécute le simulateur
output;        // on sauvegarde le résultat

set in 1,      // pour ce test, on fixe la valeur de in à 0
eval,          // on exécute le simulateur
output;        // on sauvegarde le résultat
```

Ce test charge le fichier contenant la description du circuit (*Not.hdl*). Il définit ensuite le fichier de sortie comme étant *Not.out*. Le fichier référence auquel le résultat de la simulation devra être comparé est le fichier *Not.cmp*. La commande *output-list* indique qu'il faut créer une colonne avec la valeur de l'entrée *in* suivie d'une colonne avec la valeur de la sortie *out* dans le fichier *Not.out*.

Dans la deuxième partie de la suite de test, la commande *set* permet de fixer les valeurs des différentes entrées. Comme le circuit n'a qu'une entrée, il suffit de deux commandes *set* pour couvrir toutes les possibilités.

Le fichier *Not.cmp* reprend les résultats attendus lors de l'exécution du circuit qui implémente l'inverseur. Dans ce cas, il s'agit de la table de vérité complète de l'inverseur. Pour des circuits plus simples, ce fichier ne contiendra que les valeurs attendues pour les tests réalisés.

```
|  in   |  out  |
|   0   |   1   |
|   1   |   0   |
```

Vous trouverez de nombreux autres exemples de fichiers de test dans l'archive relative au premier projet : https://www.nand2tetris.org/project01 ainsi qu'une présentation détaillée du langage HDL sur le site du livre.

### 7.4.1 Les fonctions « multi-bits »

Maintenant que nous avons vu les fonctions logiques de base, nous pouvons nous préparer à construire les circuits qui seront les briques de base d'un microprocesseur. Avant cela, il nous reste deux concepts importants à discuter.

Durant la première semaine, nous avons vu comment une fonction booléenne pouvait traiter des entrées valant *0* ou *1*. Souvent, les circuits électroniques sont amenés à traiter plusieurs données simultanément. Le livre appelle ces circuits les circuits « multi-bits ».

L'autre point que nous devons aborder sont les fonctions primitives. Durant la première semaine, nous avons travaillé avec *AND*, *OR* et *NOT*. Ces fonctions sont faciles à comprendre et utiliser. Pour des raisons technologiques, les circuits électroniques n'utilisent pas ces fonctions comme des fonctions primitives mais plutôt les fonctions *NAND* ou *NOR* dans certains cas. Nous verrons que la fonction *NAND* est une fonction primitive qui permet d'implémenter n'importe quelle fonction booléenne.

Les fonctions multi-bits sont simplement des fonctions qui sont appliquées de la même façon à plusieurs entrées. Le circuit de la Fig. 7.13 applique la fonction *NOT* à quatre entrées baptisées *x[0]*, *x[1]*, *x[2]* et *x[3]*. Les sorties sont *out[0]*, *out[1]*, *out[2]* et *out[3]*. Il est aussi possible de construire des versions multi-bits des fonctions *AND* et *OR*.

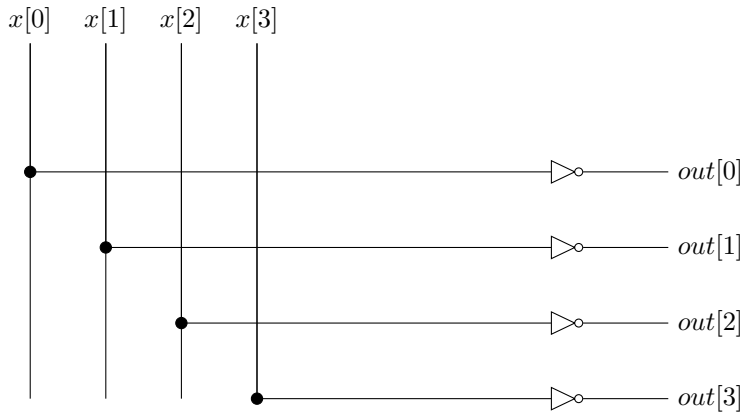



FIG. 7.13 – Représentation graphique d'un circuit NOT 4-bits

Ces deux circuits sont représentés dans les figures Fig. 7.14 et Fig. 7.15. De la même façon, on peut construire des

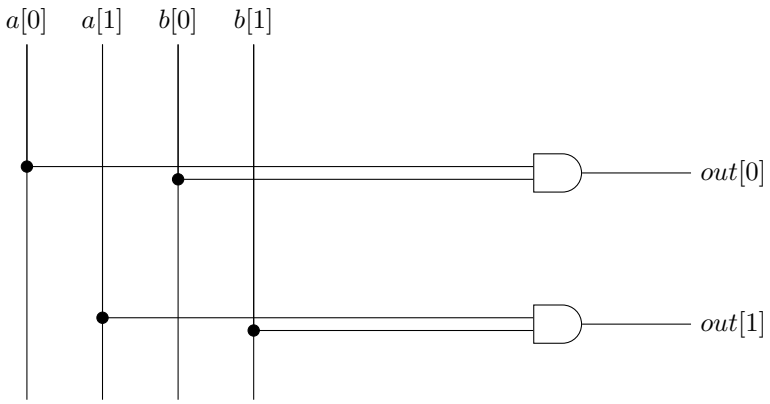



FIG. 7.14 – Représentation graphique d'un circuit AND 2-bits

multiplexeurs et des démultiplexeurs à k-bits.

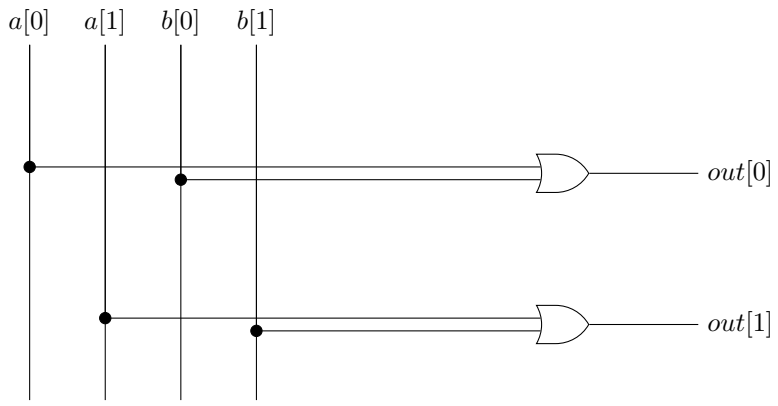



FIG. 7.15 – Représentation graphique d'un circuit OR 2-bits

### 7.4.2 La fonction universelle *NAND*

La fonction *NAND* joue un rôle particulier dans de nombreux circuits électroniques car elle sert d'élément de base à la réalisation d'autres fonctions. Un point particulier est que la fonction *NAND* permet de facilement obtenir un inverseur. Ainsi, $NAND(x, x) \iff NOT(x)$.

| x | NAND(x,x) |
|---|---|
| 0 | 1 |
| 1 | 0 |

Sur base de cette fonction *NAND*, on peut aussi facilement construire la fonction *AND* puisque $AND(x, y) \iff NAND(NAND(x, y), NAND(x, y))$. On peut s'en convaincre en construisant la table de vérité de cette fonction

| x | y | NAND(x,y) | NAND(x,y) | NAND( NAND(x,y), NAND(x,y) ) |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 |

## 7.5 Compléments sur les fonctions booléennes

Dans les chapitres précédents, nous avons couvert les bases de la construction des fonctions booléennes en utilisant les fonctions *AND*, *OR* et *NOT*. Il existe de nombreuses fonctions de ce type. La plupart de ces fonctions manipulent des séquences de bits. Certaines de ces séquences de bits servent à représenter de l'information d'un type particulier.

### 7.5.1 Représentation binaire de l'information

Dans un ordinateur, toutes les informations peuvent être stockées sous la forme d'une séquence de bits. La longueur de la séquence est fonction de la quantité d'information à stocker. Notre premier exemple concerne les caractères. Il est important de pouvoir représenter les différents caractères des langues écrites de façon compacte et non-ambiguë pour pouvoir stocker et manipuler du texte sur un ordinateur. Le principe est très simple. Il suffit de construire une table qui met en correspondance une séquence de bits et le caractère qu'elle représente.

Parmi les tables d'encodage des caractères les plus simples, la plus connue est certainement la table US-ASCII dont la définition est notamment reprise dans **RFC 20**. Cette table associe une séquence de 7 bits (*b7* à *b1*) à un caractère particulier. Pour des raisons historiques, certains de ces caractères sont des caractères dits « de contrôle » qui ne sont pas imprimables. Ils permettaient de contrôler le fonctionnement de terminaux ou d'imprimantes. Par exemple, les caractères *CR* et/ou *LF* correspondent au retour de charriot et au passage à la ligne sur un écran ou une imprimante.

Code source 7.1 – Table des caractères ASCII

| B \ b7 ----------> | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|
| I \ b6 ----------> | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| T \ b5 --------> | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| S | | | | | | | | |
| COLUMN-> | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| b4 b3 b2 b1 ROW | | | | | | | | |
| 0 0 0 0 0 | NUL | DLE | SP | 0 | @ | P | ` | p |
| 0 0 0 1 1 | SOH | DC1 | ! | 1 | A | Q | a | q |
| 0 0 1 0 2 | STX | DC2 | " | 2 | B | R | b | r |
| 0 0 1 1 3 | ETX | DC3 | # | 3 | C | S | c | s |
| 0 1 0 0 4 | EOT | DC4 | $ | 4 | D | T | d | t |
| 0 1 0 1 5 | ENQ | NAK | % | 5 | E | U | e | u |
| 0 1 1 0 6 | ACK | SYN | & | 6 | F | V | f | v |
| 0 1 1 1 7 | BEL | ETB | ' | 7 | G | W | g | w |
| 1 0 0 0 8 | BS | CAN | ( | 8 | H | X | h | x |
| 1 0 0 1 9 | HT | EM | ) | 9 | I | Y | i | y |
| 1 0 1 0 10 | LF | SUB | * | : | J | Z | j | z |
| 1 0 1 1 11 | VT | ESC | + | ; | K | [ | k | { |
| 1 1 0 0 12 | FF | FS | , | < | L | \ | l | \| |
| 1 1 0 1 13 | CR | GS | - | = | M | ] | m | } |
| 1 1 1 0 14 | SO | RS | . | > | N | ^ | n | ~ |
| 1 1 1 1 15 | SI | US | / | ? | O | _ | o | DEL |

La table US-ASCII (Code source 2.1) définit les représentations binaires suivantes :

— *01100000* correspond au caractère représentant le chiffre *0*
— *01101001* correspond au caractère représentant le chiffre *9*
— *10000011* correspond au caractère représentant la lettre *A* (majuscule)
— *01000000* correspond au caractère représentant un espace

Cette table avait l'inconvénient majeur de ne contenir que les représentations des caractères non-accentués de l'alphabet latin. Elle permet d'écrire du texte en anglais et dans d'autres langues européennes qui utilisent peu d'accents, mais ne permet évidemment pas de représenter tous les caractères des langues écrites sur notre planète. Au fil des années, ce problème a été résolu avec d'autres tables de correspondance dont celles qui sont adaptées aux accents utilisés par les langues européennes. Aujourd'hui, l'encodage standard des caractères se fait en utilisant le format Unicode. Une description détaillée d'Unicode sort du cadre de ce cours d'introduction, mais sachez qu'en mars 2020, la version 13.0 d'Unicode permettait de représenter 143859 caractères différents correspondant à 154 formes d'écritures. Unicode permet de représenter quasiment toutes les langues écrites connues sur notre planète. Des chercheurs ont même proposé un format Unicode permettant de supporter le Klingon, c'est-à-dire la langue écrite inventée pour la série de films Star Trek.

Avoir une représentation binaire pour les caractères permet de les stocker en mémoire, sur disque ou de les transmettre à travers un réseau. C'est important, mais il faut aussi pouvoir permettre à un humain de lire des textes produits par un ordinateur, que ce soit sur papier ou écran. Il existe de très nombreuses solutions qui permettent d'afficher ou d'imprimer des caractères. Dans ce cours d'introduction, nous nous contentons d'une solution très simple qui fonctionne en noir et blanc. Nous pourrons ajouter les couleurs lorsque nous aurons vu comment représenter des nombres dans le chapitre suivant.

Un écran et une imprimante permettent d'afficher des points à n'importe quelle position. On peut aisément se représenter un écran comme un rectangle composé de pixels. Chacun des points de cet écran est identifié par une abscisse et une ordonnée qui sont toutes les deux entières. Ainsi, un écran 1024x768 peut afficher 1024 points selon l'axe des x et 768 points selon l'axe des y.

Sur un tel écran, on peut facilement afficher des caractères. Il suffit d'avoir pour chaque caractère une table qui contient la représentation graphique de chacun des caractère à afficher sous la forme de pixels. A titre d'exemple, supposons que l'on veut afficher chaque caractère dans un carré de 8x8 pixels. Dans ce cas, on peut stocker la représentation graphique d'un caractère en noir en blanc sous la forme d'une suite de 8 bytes. Par exemple, les huit octets ci-dessous contiennent une représentation graphique du caractère *1*.

```
00001000
00011000
00101000
00001000
00001000
00001000
00001000
00111110
```

Une représentation graphique, fortement agrandie, de ce caractère est présentée dans la Fig. 2.1.

### 7.5.2 Fonctions booléennes sur les séquences de bits

De nombreuses fonctions manipulent des séquences de bits. Nous verrons dans le prochain chapitre comment représenter des nombres sous la forme d'une séquence de bits et comment réaliser différentes opérations arithmétiques sur ces séquences de bits. Ces fonctions sont dites combinatoires car ce sont des fonctions dont le résultat dépend uniquement des valeurs d'entrée. Dans cette section, nous abordons d'abord les fonctions combinatoires qui permettent de déplacer des bits dans une séquence. Nous considérons trois types de fonctions :

— les fonctions de décalage (à droite ou à gauche)
— les fonctions de rotation (à droite ou à gauche)
— les fonctions de masquage permettant de forcer certains bits à la valeur *0* ou *1*

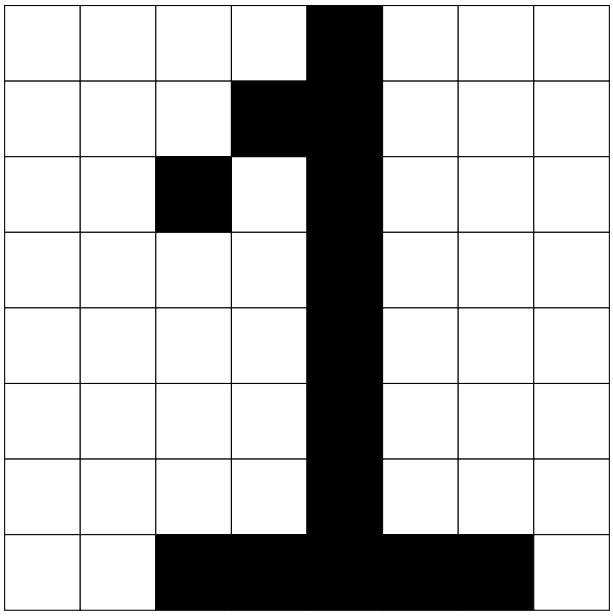


FIG. 7.16 – Un caractère sous la forme de pixels

Chacune de ces fonctions travaille sur une séquence de *n* bits, $b_{n-1}b_{n-2}...b_2b_1b_0$. Dans une telle séquence, nous avons vu que $b_{n-1}$ était le bit de poids fort tandis que $b_0$ est le bit de poids faible. Ces opérations sont généralement appliquées à des séquences de 8, 16, 32 ou 64 bits

Plusieurs fonctions de décalage sont possibles. La plus simple est la fonction de décalage d'un bit vers la droite. Cette fonction prend comme entrée la séquence de bits $b_{n-1}b_{n-2}...b_2b_1b_0$ et retourne comme résultat la séquence $0b_{n-1}b_{n-2}...b_2b_1$. Tous les bits sont décalés d'une place vers la droite. Il existe une variante de cette fonction de décalage qui retourne $b_{n-1}b_{n-1}b_{n-2}...b_2b_1$ pour la séquence d'entrée $b_{n-1}b_{n-2}...b_2b_1b_0$. Elle est parfois utilisée pour certaines manipulations des nombres entiers.

De la même façon, la fonction de décalage d'une place vers la gauche prend comme entrée la séquence de bits $b_{n-1}b_{n-2}...b_2b_1b_0$ et retourne comme résultat $b_{n-2}...b_2b_1b_00$.

Ces deux fonctions peuvent se généraliser. Plutôt que de décaler la séquence de bits d'une place vers la gauche ou vers la droite, on peut la décaler de *p* places où *p* est aussi une entrée de la fonction. Ainsi, lorsque l'on décale de deux places vers la droite la séquence $b_{n-1}b_{n-2}...b_2b_1b_0$, on obtient la séquence $00b_{n-1}b_{n-2}...b_2$. Il en va de même pour le décalage vers la gauche.

Dans certaines applications, il est utile de pouvoir forcer la valeur d'un bit particulier à *0* ou *1*. Pour illustrer ces interactions, considérons deux exemples sur base de la représentation des caractères et l'utilisation de pixels. Dans la table US-ASCII, les lettres majuscules sont représentées par des chaînes de bits dont les deux bits de poids forts sont à *10* tandis que pour les minuscules, ces deux bits de poids forts sont à *11*. Si on observe les séquences de bits pour chaque caractère, on remarque que les 4 bits de poids faible sont identiques pour la majuscule et la minuscule d'une lettre. Ainsi, pour la lettre *E*, on utilise les séquence *1000101* en majuscules et *1100101* en minuscules. Si une séquence de 7 bits représente une lettre majuscules, alors on peut facilement la convertir en minuscules en forçant le deuxième bit de poids fort à la valeur *1*. Sachant que la fonction booléenne *OR* retourne toujours *1* lorsqu'au moins une de ses deux entrées vaut *1*, on peut transformer une majuscule en minuscule en calculant *OR* avec la séquence *0100000*. Si la représentation du caractère initiale est $b_6b_5b_4b_3b_2b_1b_0$, alors la fonction *OR 0100000* retournera $b_61b_4b_3b_2b_1b_0$. De la même façon, on peut forcer un bit à zéro en utilisant la fonction *AND*. Par exemple, pour transformer une minuscule en majuscule en utilisant le masque *1011111*.

Lorsqu'un ordinateur doit transmettre ou stocker de l'information encodée sous la forme d'une séquence de bits, il doit parfois pouvoir s'assurer que l'information qui est reçue ou lue est bien identique à celle qui a été envoyée ou écrite.

Un exemple classique de l'utilisation de ces techniques concerne les sondes spatiales qui sont envoyées pour explorer les planètes du système solaire voire explorer au-delà de notre système solaire. Ces sondes collectent de nombreuses informations qu'elles doivent envoyer par radio vers la Terre. Différentes techniques, qui sortent du cadre de ce cours, permettent d'envoyer des séquences de bits par radio. Malheureusement, les transmissions radio peuvent être perturbées par différents phénomènes naturels dont les émissions du soleil par exemple. Suite à ces perturbations, une séquence de bits envoyée par une sonde spatiale peut être reçue de façon incorrecte par la station d'écoute se trouvant au sol. Vu les capacités de la sonde spatiale et les délais de transmission entre les confins du système solaire et la Terre, il est impossible de demander à la sonde spatiale de stocker de l'information pour pouvoir la retransmettre au cas où elle ne serait pas reçue correctement par la station d'écoute sur la Terre. A titre d'exemple, la distance entre Mercure et la Terre varie entre 77 millions de kilomètres et 222 millions de kilomètres. La lumière, qui est la façon la plus rapide de transmettre de l'information, se propage à une vitesse de 300.000 kilomètres par seconde. Cela signifie que lorsque Mercure est proche de la Terre, un signal émis par une sonde autour de Mercure met au moins 256 secondes pour atteindre la Terre. Pour les sondes Voyager 1 et Voyager 2 qui explorent les confins du système solaire, les délais sont encore plus grands. En octobre 2020, un signal radio émis par Voyager 1 mettait près de 21 heures pour atteindre la Terre.

Plusieurs techniques ont étés proposées pour faire face à des erreurs dans la transmission de séquences de bits. Certaines permettent de détecter des erreurs dans l'information reçue. D'autres, plus complexes, permettent de récupérer certaines erreurs de transmission.

Les techniques de détection les plus simples sont les techniques dite *de parité*. L'idée est très simple. Pour pouvoir détecter si une erreur de transmission a affecté une séquence de bits, il suffit d'encoder ces séquences de bits de façon à pouvoir facilement distinguer une séquence valide d'une séquence invalide. Les techniques de parité séparent les séquences de bits en deux moitiés. La première contient les séquences valides qui sont émises par l'émetteur. La seconde contient des séquences qui peuvent être obtenues des première après une erreur de transmission.

La technique de parité paire fonctionne comme suit. Une séquence de *n+1* bits, $b_{n-1}b_{n-2}...b_2b_1b_0p$ est valide si elle contient un nombre pair de bits ayant la valeur *1* et invalide sinon. Lorsqu'un émetteur veut envoyer *n* bits, il doit calculer la valeur du bit de poids faible de façon à ce que la séquence des *n+1* bits contienne un nombre pair de bits à la valeur *1*.

Il est utile de prendre quelques exemples pour bien comprendre comment cette technique fonctionne. Considérons les caractères représentés sur 7 bits. Une parité peut être associé à chacun de ces caractères.

— la parité paire de *01100000* sera *0*
— la parité paire de *01101001* sera *0*
— la parité paire de *10000011* sera *1*

Considérons une sonde spatiale qui envoie la séquence de bits composée de ces trois caractères avec leur parité paire, c'est-à-dire : *011000000 011010010 100000111*. La station d'écoute pourra recalculer le bit de parité qui est placé dans le bit de poids faible de chaque octet pour vérifier qu'il n'y a pas eu d'erreur de transmission. Si par contre la station d'écoute reçoit *011000001 111010010 100000111*, elle pourra vérifier que les deux premiers octets sont incorrects tandis que le troisième est correct. Cette technique de parité permet de détecter les erreurs de transmission qui modifient la valeur de un (et un seul bit) dans la séquence de bits couverte par la parité. En pratique, l'émetteur envoie les bits et calcule la valeur du bit de parité pendant l'envoi de ces bits. Le receveur fait l'inverse pour vérifier que la parité de la séquence reçue est correcte.

CHAPITRE 8

# Arithmétique binaire

Dans le chapitre précédent, nous avons vu comment un ordinateur pouvait représenter des caractères et des images sous la forme d'un séquence de symboles binaires ou bits. Dans ce chapitre, nous nous focaliserons sur la façon dont il est possible de représenter les nombres entiers et ensuite de réaliser des opérations arithmétiques simples (addition et soustraction) sur ces nombres.

## 8.1 Représentation des nombres naturels

Commençons par analyser comment représenter les nombres pour effectuer des opérations arithmétiques. Pour simplifier la présentation, nous travaillerons surtout avec des quartets dans ce chapitre. Il y a seize quartets différents :

- *0000*
- *0001*
- *0010*
- *0011*
- *0100*
- *0101*
- *0110*
- *0111*
- *1000*
- *1001*
- *1010*
- *1011*
- *1100*
- *1101*
- *1110*
- *1111*

Un tel quartet, peut se représenter de façon symbolique : $B_3B_2B_1B_0$ où les symboles $B_i$ peuvent prendre les valeurs *0* ou *1*. Dans un tel quartet, le symbole $B_3$ est appelé le bit de poids fort tandis que le symbole $B_0$ est appelé le bit de poids faible.

Cette représentation des quartets est similaire à la représentation que l'on utilise pour les nombres décimaux. Un nombre en représentation décimale peut aussi s'écrire $C_{n-1}C_{n-2}...C_2C_1C_0$. Dans cette représentation, les $C_i$ sont

les chiffres de *0* à *9*. $C_0$ est le chiffre des unités, $C_1$ le chiffre correspondant aux dizaines, $C_2$ celui qui correspond aux centaines, ... Numériquement, on peut écrire que la représentation décimale $C_3C_2C_1C_0$ correspond au nombre $C_3 * 1000 + C_2 * 100 + C_1 * 10 + C_0$ ou encore $C_3 * 10^3 + C_2 * 10^2 + C_1 * 10^1 + C_0 * 10^0$ en se rappelant que $10^0$ vaut 1.

En toute généralité, la suite de chiffres $C_{n-1}C_{n-2}...C_2C_1C_0$ correspond au naturel $\sum_{i=0}^{i=n-1} C_i \times 10^i$.

A titre d'exemple, le nombre sept cent trente six s'écrit en notation décimale *736*, ce qui équivaut bien à $7 * 10^2 + 3 * 10^1 + 6 * 10^0$.

Pour représenter les nombres naturels en notation binaire, nous allons utiliser le même principe. Un nombre en notation binaire $B_{n-1}B_{n-2}...B_2B_1B_0$ représente le nombre naturel $B_{n-1}*2^{n-1}+B_{n-2}*2^{n-2}+...+B_2*2^2+B_1*2^1+B_0*2^0$. En appliquant cette règle aux quartets, on obtient aisément :

- *0000* correspond au nombre $0 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0 * 2^0$, soit *0* en notation décimale
- *0001* correspond au nombre $0 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0$, soit *1* en notation décimale
- *0010* correspond au nombre $0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0$, soit *2* en notation décimale
- *0011* correspond au nombre $0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$, soit *3* en notation décimale
- *0100* correspond au nombre $0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0$, soit *4* en notation décimale
- *0101* correspond au nombre $0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0$, soit *5* en notation décimale
- *0110* correspond au nombre $0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0$, soit *6* en notation décimale
- *0111* correspond au nombre $0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0$, soit *7* en notation décimale
- *1000* correspond au nombre $1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0 * 2^0$, soit *8* en notation décimale
- *1001* correspond au nombre $1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0$, soit *9* en notation décimale
- *1010* correspond au nombre $1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0$, soit *10* en notation décimale
- *1011* correspond au nombre $1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$, soit *11* en notation décimale
- *1100* correspond au nombre $1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0$, soit *12* en notation décimale
- *1101* correspond au nombre $1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0$, soit *13* en notation décimale
- *1110* correspond au nombre $1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0$, soit *14* en notation décimale
- *1111* correspond au nombre $1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0$, soit *15* en notation décimale

En toute généralité, la suite de bits $B_{n-1}B_{n-2}...B_2B_1B_0$ correspond au naturel $\sum_{i=0}^{i=n-1} B_i \times 2^i$.

Cette technique peut s'appliquer à des nombres binaires contenant un nombre quelconque de bits. Pour convertir efficacement un nombre binaire en son équivalent décimal, il est intéressant de connaître les principales puissances de 2 :

- $2^0 = 1$
- $2^1 = 2$
- $2^2 = 4$
- $2^3 = 8$
- $2^4 = 16$
- $2^5 = 32$
- $2^6 = 64$
- $2^7 = 128$
- $2^8 = 256$
- $2^9 = 512$
- $2^{10} = 1024$
- $2^{16} = 65536$
- $2^{20} = 1048576$ ou un peu plus d'un million
- $2^{30} = 1073741824$ ou un peu plus d'un milliard
- $2^{32} = 4294967296$ ou un peu plus de 4 milliards

Cette représentation des nombres peut se généraliser. La notation binaire utilise des puissances de *2* tandis que la notation décimale des puissances de *10*. On peut faire de même avec d'autres puissances. Ainsi, la suite de symboles $S_{n-1}S_{n-2}...S_2S_1S_0$ en base *k* où les symboles $S_i$ ont une valeur comprises entre *0* et $k-1$, correspond au naturel $\sum_{i=0}^{i=n-1} S_i \times k^i$.

En pratique, outre les notations binaires, deux notations sont couramment utilisées :

- l'octal (ou base *8*)

— l'hexadécimal (ou base *16*)

En octal, les symboles sont des chiffres de *0* à *7*. En hexadécimal, les symboles sont des chiffres de *0* à *9* et les lettres de *A* à *F* sont utilisées pour représenter les valeurs de *0* à 15.

---

**Note :** Il est parfois intéressant d'entrer un nombre en binaire, octal ou hexadécimal dans un langage de programmation. En python3, cela se fait en préfixant le nombre avec *0b* pour du binaire, *0o* pour de l'octal et *0x* pour de l'hexadécimal. Ainsi, les lignes ci-dessous stockent toutes la valeur *23* dans la variable `n`.

```
n = 23  # décimal
n = 0b10111  # binaire
n = 0o27 # octal
n = 0x17
```

La notation adoptée dans python3 est bien plus claire que celle utilisée dans d'anciennes versions de python et des langages de programmation comme le *C*. Dans ces langages, il suffit de commencer un nombre par le chiffre zéro pour indiquer qu'il est en octal. C'était une source de très nombreuses confusions.

```
# En python2, ces deux lignes ne sont pas équivalentes
n = 23  # décimal
n = 023 # octal -> valeur décimale 19
```

---

## 8.2 Opérations arithmétiques sur les nombres binaires

Sur base de cette représentation binaire des nombres naturels, il est possible de réaliser toutes les opérations arithmétiques. La première que nous aborderons est l'addition. Avant de travailler en binaire, il est intéressant de se rappeler comment l'addition se réalise en calcul écrit. Considérons comme premier exemple $123 + 463$.

```
  1 2 3   << premier naturel
+ 4 6 3   << second naturel
-------
  5 8 6
```

Pour des nombres simples comme celui repris ci-dessus, l'addition s'effectue « chiffre par chiffre ». Vous avez aussi appris qu'il faut parfois faire des reports lorsqu'une addition « chiffre par chiffre » donne un résultat qui est supérieur à 10. C'est le cas lorsque l'on cherche à ajouter *456* à *789*.

```
  1  1  1      << reports
     4  5  6   << premier naturel
+    7  8  9   << second naturel
------------
  1  2  4  5
```

L'intérêt de cette approche est que l'addition avec des nombres en représentation binaire peut se faire de la même façon. Considérons quelques exemples avec des naturels représentés sur 4 bits.

```
   0  0  1  0   << premier nombre binaire (2 en décimal)
+  0  1  0  1   << second nombre binaire (5 en décimal)
-------------
   0  1  1  1   << 7 en décimal
```

On vérifie aisément que $2+5 = 7$. Comme avec l'addition des naturels, il est aussi possible d'avoir des reports lorsque l'on réalise une addition entre des nombres binaires. L'exemple ci-dessous réalise l'addition $2 + 7$.

```
   1  1  0  0   << reports
   0  0  1  0   << premier nombre binaire (2 en décimal)
+  0  1  1  1   << second nombre binaire  (7 en décimal)
-------------
   1  0  0  1
```

Tout comme avec l'addition des naturels, le report est aussi possible avec le bit de poids fort. En toute généralité, lorsque l'on additionne deux quartets, la notation binaire du résultat devra parfois être stockée sur 5 bits et non 4. L'exemple ci-dessous illustre ce cas.

```
  1  1  1          << reports
     1  0  1  0    << premier nombre binaire (10 en décimal)
+    0  1  1  1    << second nombre binaire  (7 en décimal)
---------------
  1  0  0  0  1
```

En utilisant la représentation binaire, il est possible de construire des fonctions booléennes qui permettent de réaliser l'opération d'addition. Commençons par considérer l'addition entre deux bits. En tout généralité, cette addition peut donner comme résultat un nombre stocké sur deux bits, le bit de poids fort (*report*) et le bit de poids faible (*somme*). Si les deux bits à additionner sont *a* et *b*, on peut facilement vérifier que cette addition correspond à la table de vérité ci-dessous.

| a | b | report | somme |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

Cette table de vérité correspond à ce que l'on appelle un demi-additionneur (half-adder en anglais). On l'appelle demi-additionneur car en général, un bit du résultat de l'addition binaire est le résultat de l'addition de trois bits et non deux : les deux bits des nombres à additionner et le bit de report de l'étage précédent.

| a | b | r | report | somme |
|---|---|---|---|---|
| 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 |

Cette table de vérité correspond à ce que l'on appelle un additionneur complet ( full-adder en anglais). Il s'agit d'une fonction booléenne à trois entrées (*a*, *b* et *r*) et deux sorties (*report* et *somme*). Comme toutes les fonctions booléennes que nous avons vu dans les chapitres précédents, celle-ci peut facilement s'implémenter en utilisant des fonctions *AND*, *OR* et des inverseurs.

Vous développerez les circuits correspondants à ces additionneurs dans le cadre du deuxième projet. Un point important à noter est que l'additionneur complet peut facilement remplacer un demi-additionneur en mettant son entrée *r* à zéro. Dans ce cas, sa table de vérité est la suivante :

| a | b | r | report | somme |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |

Cet additionneur sera important dans le cadre de ce cours. La Fig. 8.1 le représente schématiquement sous la forme d'un rectangle avec trois entrées (*a*, *b* et *r*) et deux sorties (*report* et *somme*). Le plus intéressant est que ces additionneurs

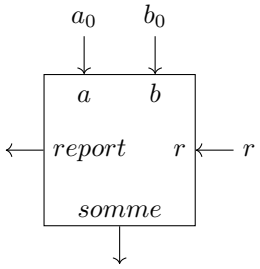



FIG. 8.1 – Un additionneur complet

peuvent se combiner en cascade pour construire un additionneur qui est capable d'additionner deux nombres binaires sur *n* bits. La Fig. 8.2 présente un additionneur qui travaille avec deux quartets, *a* et *b*. Pour des raisons graphiques,

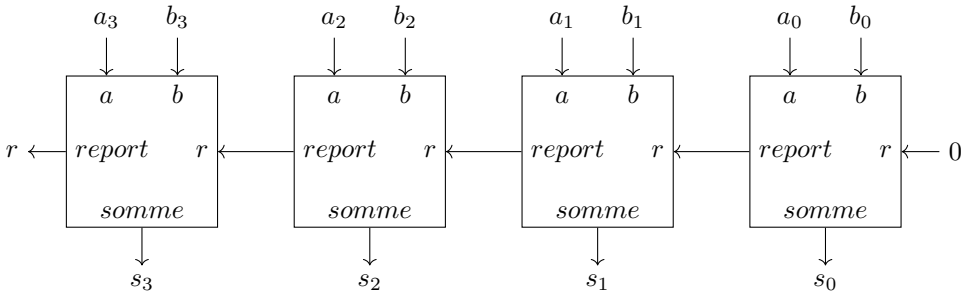



FIG. 8.2 – Avec quatre additionneurs, on peut additionner des quartets

il est compliqué de dessiner un additionneur pour des octets ou des mots de 16 ou 32 bits, mais le même principe s'applique. On peut donc facilement construire un additionneur qui prend en entrées deux nombres encodés sur *n* bits et retourne un résultat encodé sur *n* bits avec un report éventuel.

L'additionneur que nous venons de construire prend comme entrées les bits des deux nombres à additionner. Dans ce circuit, le report de l'additionneur qui correspond au bit de poids faible est mis à *0*. Que se passerait-il si cette entrée *r* était mise à la valeur *1* ? Le circuit calculerait le résultat de l'addition $a + b + 1$.

En informatique, on doit très souvent incrémenter une valeur entière, par exemple à l'intérieur de boucles. Si *a* est la valeur à incrémenter, on peut grâce à nos quatre additionneurs incrémenter cette valeur en forçant les entrées $b_i$ à 0 et le report du bit de poids faible à *1*. Ce circuit est représenté dans le schéma de la Fig. 8.3.

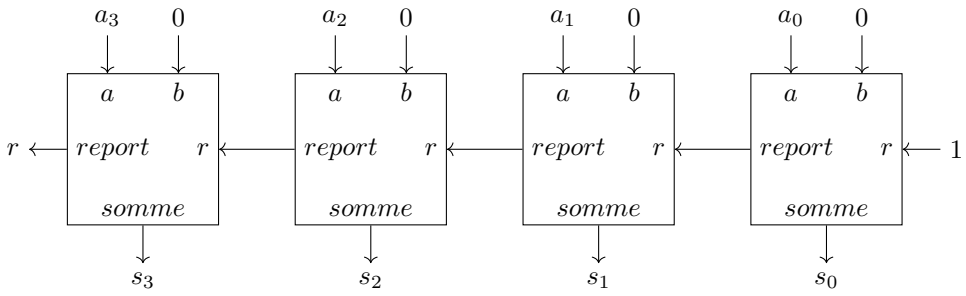



FIG. 8.3 – Un circuit pour incrémenter un quartet

## 8.3 Représentation des nombres entiers

La solution présentée dans la section précédente permet de facilement représenter les nombres naturels qui sont nuls ou strictement positifs. En pratique, les ordinateurs doivent aussi pouvoir représenter les nombres négatifs et effectuer des soustractions. Différentes solutions sont envisageables pour représenter ces nombres entiers.

Une première approche serait d'utiliser un bit du nombre binaire pour indiquer explicitement si le nombre est positif ou négatif. A titre d'exemple, considérons une représentation sur 4 bits et utilisons le bit de poids fort pour indiquer le signe (*0* pour un nombre positif et *1* pour un nombre négatif). Avec cette convention, nous pourrions représenter les nombres suivants :

- *0 000* représente le nombre *+0*
- *0 001* représente le nombre *+1*
- *0 010* représente le nombre *+2*
- *0 011* représente le nombre *+3*
- *0 100* représente le nombre *+4*
- *0 101* représente le nombre *+5*
- *0 110* représente le nombre *+6*
- *0 111* représente le nombre *+7*
- *1 000* représente le nombre *-0*
- *1 001* représente le nombre *-1*
- *1 010* représente le nombre *-2*
- *1 011* représente le nombre *-3*
- *1 100* représente le nombre *-4*
- *1 101* représente le nombre *-5*
- *1 110* représente le nombre *-6*
- *1 111* représente le nombre *-7*

Nous aurions pu aussi choisir d'utiliser le bit de poids faible pour indiquer le signe du nombre entier. Avec cette convention, nous pourrions représenter les nombres suivants :

- *000 0* représente le nombre *+0*
- *000 1* représente le nombre *-0*
- *001 0* représente le nombre *+1*
- *001 1* représente le nombre *-1*
- *010 0* représente le nombre *+2*
- *010 1* représente le nombre *-2*
- *011 0* représente le nombre *+3*
- *011 1* représente le nombre *-3*
- *100 0* représente le nombre *+4*
- *100 1* représente le nombre *-4*
- *101 0* représente le nombre *+5*
- *101 1* représente le nombre *-5*
- *110 0* représente le nombre *+6*

- *110 1* représente le nombre *-6*
- *111 0* représente le nombre *-7*
- *111 1* représente le nombre *-7*

Ces deux conventions permettent de représenter les entiers de *-7* à *+7*. Malheureusement, ces deux représentations ont deux inconvénients majeurs. Premièrement, elles utilisent deux nombres binaires différents pour représenter la valeur nulle. De plus, il est difficile de construire des circuits électroniques qui permettent de facilement manipuler de telles représentations des nombres entiers.

La solution à ce problème est d'utiliser la notation en complément à deux. Pour représenter les nombres entiers en notation binaire, nous adaptons la représentation utilisée pour les nombres naturels. Le nombre binaire $B_{n-1}B_{n-2}...B_2B_1B_0$ représente le nombre entier $(-1)*B_{n-1}*2^{n-1}+B_{n-2}*2^{n-2}+...+B_2*2^2+B_1*2^1+B_0*2^0$. Il est important de noter que le facteur *(-1)* est appliqué uniquement au bit de poids fort. En appliquant cette règle aux quartets, on obtient aisément :

- *0000* représente le nombre *0*
- *0001* représente le nombre *1*
- *0010* représente le nombre *2*
- *0011* représente le nombre *3*
- *0100* représente le nombre *4*
- *0101* représente le nombre *5*
- *0110* représente le nombre *6*
- *0111* représente le nombre *7*
- *1000* représente le nombre $-8+0 \rightarrow -8$
- *1001* représente le nombre $-8+1 \rightarrow -7$
- *1010* représente le nombre $-8+2 \rightarrow -6$
- *1011* représente le nombre $-8+3 \rightarrow -5$
- *1100* représente le nombre $-8+4 \rightarrow -4$
- *1101* représente le nombre $-8+5 \rightarrow -3$
- *1110* représente le nombre $-8+6 \rightarrow -2$
- *1111* représente le nombre $-8+7 \rightarrow -1$

On remarque aisément qu'il n'y a qu'une seule chaîne de bits qui représente la valeur nulle et que celle-ci correspond à la chaîne de bits dans laquelle tous les bits sont à *0*. C'est un avantage important par rapport aux représentations précédentes. Par contre, il existe un nombre négatif qui n'a pas d'opposé dans une représentation utilisant un nombre fixe de bits. C'est inévitable sachant qu'avec *n* bits on ne peut représenter que $2^n$ nombres distincts. Le site web https://integer.exposed/ vous permet de facilement expérimenter avec les représentations binaires des nombres entiers sur 8, 16, 32 ou même 64 bits.

Une propriété intéressante de la notation en complément à deux est que tous les nombres négatifs ont leur bit de poids fort qui vaut *1*. C'est une conséquence de la façon dont ces nombres sont représentés et pas un *bit de signe* explicite comme dans les représentations précédentes.

Enfin, l'avantage principal de cette représentation est que l'on va pouvoir assez facilement construire les circuits qui permettent d'effectuer des opérations arithmétiques sur ces nombres. Un premier avantage de la représentation en complément à deux, est qu'il est possible de réutiliser notre additionneur sans aucune modification pour additionner des entiers. Considérons comme premier exemple $(-6)+(-1)$.

```
   1  1         << reports
   1  0  1  0   << premier nombre binaire : -6
+  1  1  1  1   << second nombre binaire  : -1
-------------
1  1  0  0  1
```

Le quartet *1001* est bien la représentation du nombre négatif *-7*. Comme second exemple, prenons $(-2)+(-3)$. Le résultat de l'addition bit à bit est *1011* qui est le quartet qui représente le nombre entier *-5*.

```
   1            << reports
   1  1  1  0   << premier nombre binaire : -2
```

(suite sur la page suivante)

(suite de la page précédente)

```
+  1  1  0  1   << second nombre binaire  : -3
-------------
1  1  0  1  1
```

On peut maintenant se demander comment calculer l'opposé d'un nombre en représentation binaire. Une première approche est de déterminer la table de vérité de cette opération qui prend comme entrée *n* bits et retourne un résultat sur *n* bits également. A titre d'exemple, considérons des nombres binaires sur 3 bits.

| a2 | a1 | a0 | b2 | b1 | b0 | Commentaire |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | *opposé(0)=0* |
| 0 | 0 | 1 | 1 | 1 | 1 | *opposé(1)=-1* |
| 0 | 1 | 0 | 1 | 1 | 0 | *opposé(2)=-2* |
| 0 | 1 | 1 | 1 | 0 | 1 | *opposé(3)=-3* |
| 1 | 0 | 0 | ? | ? | ? | *-4* n'a pas d'opposé |
| 1 | 0 | 1 | 0 | 1 | 1 | *opposé(-3)=3* |
| 1 | 1 | 0 | 0 | 1 | 0 | *opposé(-2)=2* |
| 1 | 1 | 1 | 0 | 0 | 1 | *opposé(-1)=1* |

Sur base de cette table de vérité, on pourrait facilement construire un circuit qui calcule l'opposé d'un nombre sur n bits en utilisant des fonctions *AND*, *OR* et *NOT* ou uniquement des fonctions *NAND* comme durant le premier projet. Cependant, on peut faire beaucoup mieux en réutilisant l'additionneur dont nous disposons déjà. Si on observe la table de vérité ci-dessus, on remarque que l'on peut calculer l'opposé d'un nombre binaire en deux étapes :

a. inverser tous les bits de ce nombre en utilisant l'opération *NOT*

b. incrémenter d'une unité le nombre binaire obtenu

La première opération est facile à réaliser en utilisant la fonction *NOT*. La seconde peut se réaliser en utilisant notre additionneur avec un report du bit de poids faible initialisé à *1*. Schématiquement, le circuit à construire pour calculer l'opposé du quartet *a* est donc celui de la Fig. 8.4. Si on sait facilement calculer l'opposé d'un nombre, et additionner

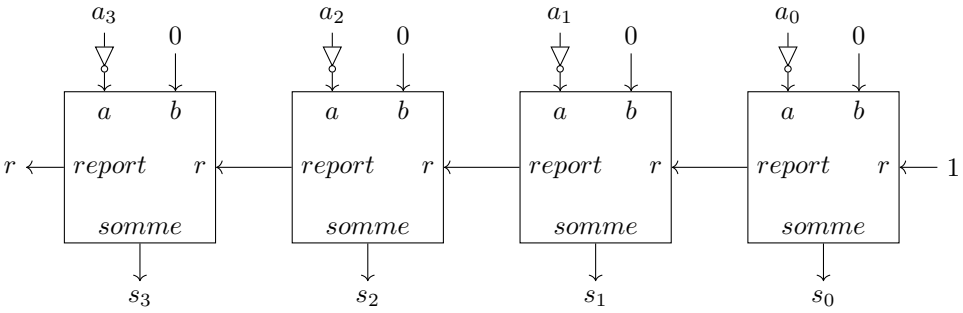



FIG. 8.4 – Calcul de l'opposé d'un quartet

deux nombres, il devient possible de réaliser la soustraction. Pour calculer $a - b$, il suffit de calculer $a + (-b)$. Le circuit de la Fig. 8.5 réalise la soustraction $b - a$. Notez que le report du bit de poids faible est mis à *1* et que les bits $a_i$ sont inversés.

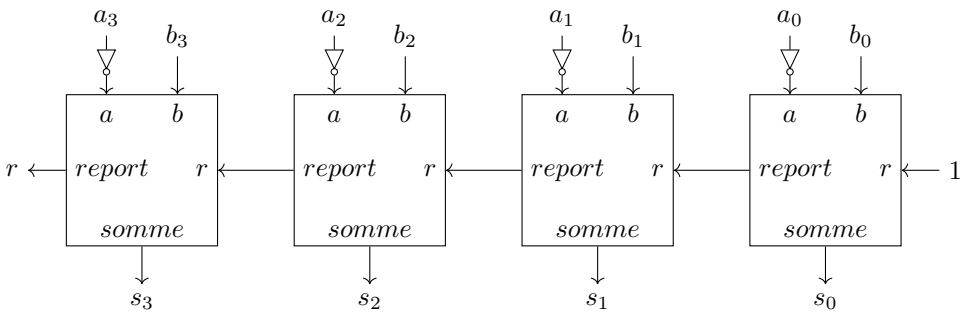



FIG. 8.5 – Soustraction : b-a

## 8.4 Unité Arithmétique et Logique

Cet additionneur joue un rôle important dans les microprocesseurs utilisés par un ordinateur. Souvent, il n'est pas utilisé seul, mais plutôt à l'intérieur d'une Unité Arithmétique et Logique (*Arithmetic and Logic Unit* (ALU) en anglais). Ce circuit constitue le coeur d'un ordinateur au niveau du calcul. Il combine les principales fonctions de manipulations de séquences de bits. Dans le projet précédent, vous avez construit un premier circuit programmable : le multiplexeur. Celui-ci a deux entrées sur *n* bits et un signal de contrôle qui permet de sélectionner en sortie la valeur de la première ou de la seconde entrée. L'ALU va plus loin car elle prend deux signaux sur *n* bits en entrée (*x* et *y*) et plusieurs signaux de contrôle qui permettent de sélectionner l'opération à effectuer et à envoyer vers les fils de sortie. L'ALU proposée dans le livre permet de réaliser les 18 opérations reprises dans la Tableau 8.1.

TABLEAU 8.1 – Signaux de contrôle de l'ALU

| Opération | Commentaire |
|---|---|
| *0* | La sortie vaut toujours la représentation binaire de *0* |
| *1* | La sortie vaut toujours la représentation binaire de *1* |
| *-1* | La sortie vaut toujours la représentation binaire de *-1* |
| *x* | La sortie est égale à l'entrée *x* |
| *y* | La sortie est égale à l'entrée *y* |
| *NOT(x)* | La sortie est égale à l'entrée *x* inversée |
| *NOT(y)* | La sortie est égale à l'entrée *y* inversée |
| opposé(x) | La sortie est égale à l'opposé de l'entrée *x* |
| opposé(y) | La sortie est égale à l'opposé de l'entrée *y* |
| incrément(x) | La sortie vaut $x + 1$ |
| incrément(y) | La sortie vaut $y + 1$ |
| décrément(x) | La sortie vaut $x - 1$ |
| décrément(y) | La sortie vaut $y - 1$ |
| addition | La sortie vaut $x + y$ |
| soustraction | La sortie vaut $x - y$ |
| soustraction | La sortie vaut $y - x$ |
| *AND* | La sortie vaut *AND(x,y)* |
| *OR* | La sortie vaut *OR(x,y)* |

Certaines ALUs vont plus loin et supportent d'autres opérations, mais supporter toutes ces opérations va déjà nécessiter un peu de travail dans le cadre de notre deuxième projet. Nous avons déjà vu comment effectuer quasiment chacune de ces opérations en utilisant des fonctions booléennes. Pour les combiner dans un seul circuit, il suffira d'utiliser des multiplexeurs et de choisir des signaux permettant de les contrôler. L'ALU du livre de référence utilise six signaux de contrôle :

— *zx* : l'entrée *x* est mise à *0*
— *zy* : l'entrée *y* est mise à *0*

— *nx* : l'entrée *x* est inversée
— *ny* : l'entrée *y* est inversée
— *f* : permet de choisir entre le résultat de l'additionneur (*1*) et de la fonction *AND* pour la sortie
— *no* : permet d'inverser ou non le résultat

Outre le résultat qui est encodé sur 16 bits, l'ALU retourne également deux drapeaux :

— *zr* qui est mis à *1* si le résultat de l'ALU vaut zéro et *0* sinon
— *ng* qui est mis à *1* si le résultat de l'ALU est négatif et *0* sinon

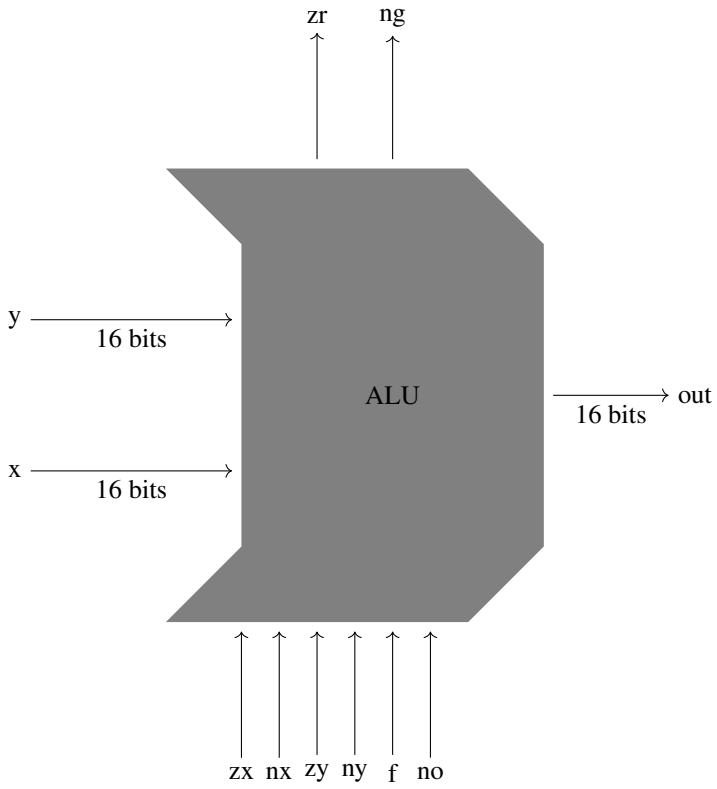



FIG. 8.6 – Unité Arithmétique et Logique (ALU)

Les drapeaux *zr* et *ng* méritent un peu d'explication. Sur base de la représentation des nombres entiers, il est facile de vérifier si la représentation binaire d'un nombre entier vaut *0*. Il suffit de vérifier que tous ses bits valent *0*. Pour calculer la valeur de *ng*, c'est encore plus simple, il suffit de retourner la valeur du bit de poids fort du résultat puisqu'en notation complément à 2, celui-ci vaut *1* pour tous les entiers négatifs.

Les signaux de contrôle ont chacun une signification particulière.

— lorsque le signal *zx* est mis à *1*, l'entrée *x* est remplacée par la valeur *0*
— lorsque le signal *zy* est mis à *1*, l'entrée *y* est remplacée par la valeur *0*
— lorsque le signal *nx* est mis à *1*, alors l'entrée *x* est inversée (opération *NOT*) avant d'être utilisée
— lorsque le signal *ny* est mis à *1*, alors l'entrée *y* est inversée (opération *NOT*) avant d'être utilisée
— lorsque le signal *no* est mis à *1*, alors la sortie *out* est inversée (opération *NOT*) après l'exécution de l'opération demandée

Ces signaux de contrôle sont connectés à des multiplexeurs qui permettent de choisir entre un signal, l'inverse de ce signal ou la constante *0*.

Enfin, le signal de contrôle *f* permet de connecter soit le résultat de l'additionneur ou soit celui d'une fonction *AND* à la sortie.

Les éléments principaux de l'ALU sont donc des inverseurs, la constante *0*, des multiplexeurs, un additionneur 16 bits et une fonction *AND* à 16 bits.

La construction complète de cette ALU nécessite l'utilisation de quelques astuces et propriétés de la représentation binaire des nombres entiers. Le livre suggère d'utiliser les signaux de contrôle d'une façon particulière.

Pour calculer *0*, il faut mettre *zx*, *zy* et *f* à *1*. Cela revient donc à calculer l'opération $0 + 0$.

Pour calculer *1*, il faut mettre tous les signaux de contrôle` (*zx*, *zy*, *nx*, *ny*, *f* et *no* à *1*). Voyez-vous pourquoi cette addition suivie d'une inversion donne bien comme résultat la valeur *1* ?

Pour calculer *-1*, il faut mettre cinq signaux de contrôle (*zx*, *zy*, *nx*, et *f*) à *1*. Les signaux *ny* et *no* sont mis à *0*. En mettant *zx* et *nx* à *1*, l'entrée *x* de l'ALU contient la valeur *-1*. Comme *zy* est mis à *1*, l'ALU calcule $-1 + 0$. On aurait pu obtenir le même résultat avec d'autres valeurs des signaux de contrôle. Lesquels ?

Pour retourner *x* comme sortie, il faut mettre *zy* à *1*. On calcule donc le résultat de $x + 0$.

Pour retourner *y* comme sortie, il faut mettre *zx* à *1*. On calcule donc le résultat de $0 + y$.

Pour calculer *NOT(x)*, il y a deux approches possibles. La première est de mettre *zy* à *1* et *nx* à *1*. La seconde est de mettre uniquement *zy* et *nx* à *1*. Dans le premier cas, on calcule $x + 0$ et on inverse le résultat. Dans le second cas, on calcule $NOT(x) + 0$. On peut raisonner de façon similaire pour le calcul de *NOT(y)*.

Pour calculer *-x*, le livre suggère de mettre quatre signaux à *1* : *zy*, *ny*, *f* et *no*. Cela revient à calculer l'opération $NOT(ADD(x, 11..11))$. Regardons avec des nombres encodés sur trois bits le résultat de cette opération.

| x2 | x1 | x0 | ADD(x,111) | NOT(ADD(x,111)) |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 1 1 1 | 0 0 0 |
| 0 | 0 | 1 | 1 0 0 0 | 1 1 1 |
| 0 | 1 | 0 | 1 0 0 1 | 1 1 0 |
| 0 | 1 | 1 | 1 0 1 0 | 1 0 1 |
| 1 | 0 | 0 | 1 0 1 1 | 1 0 0 << pas d'opposé |
| 1 | 0 | 1 | 1 1 0 0 | 0 1 1 |
| 1 | 1 | 0 | 1 1 0 1 | 0 1 0 |
| 1 | 1 | 1 | 1 1 1 0 | 0 0 1 |

On obtient bien le résultat attendu.

Pour calculer $x - 1$, les signaux *zy* et *ny* et *f* sont mis à *1*. Le circuit calcule donc *ADD(x,11..11)* ce qui correspond bien à *x-1*. De même, on peut calculer $y - 1$.

Pour calculer *x+1*, seul *zx* est mis à zéro, tous les autres signaux de contrôle sont mis à *1*. Le circuit calcule donc *NOT(ADD(NOT(x),11...11))*. Regardons avec des nombres de trois bits le résultat de cette opération.

| x2 | x1 | x0 | NOT(x) | ADD(NOT(x),111)) | NOT(...) | Commentaire |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 1 1 | 1 1 1 0 | 0 0 1 | << 0+1=1 |
| 0 | 0 | 1 | 1 1 0 | 1 1 0 1 | 0 1 0 | << 1+1=2 |
| 0 | 1 | 0 | 1 0 1 | 1 1 0 0 | 0 1 1 | << 2+1=3 |
| 0 | 1 | 1 | 1 0 0 | 1 0 1 1 | 1 0 0 | << 3+1=4 |
| 1 | 0 | 0 | 0 1 1 | 1 0 1 0 | 1 0 1 | << -4+1=-3 |
| 1 | 0 | 1 | 0 1 0 | 1 0 0 1 | 1 1 0 | << -3+1=-2 |
| 1 | 1 | 0 | 0 0 1 | 1 0 0 0 | 1 1 1 | << -2+1=-1 |
| 1 | 1 | 1 | 0 0 0 | 0 1 1 1 | 0 0 0 | << -1+1=0 |

Pour la même raison, pour calculer *y+1*, seul *zy* est mis à zéro, tous les autres signaux de contrôle sont mis à *1*.

Pour calculer *x+y*, seul *f* doit être mis à *1*. Pour calculer *x-y*, *nx*, *f* et *no* sont mis à *1*. On doit donc calculer *NOT(ADD(NOT(x),y))*. Vous pouvez vous en convaincre en reconstruisant la table de vérité. De même pour calculer *y-x*, seuls les signaux *no*, *f* et *ny* sont mis à *1*.

Enfin, pour implémenter l'opération *OR* en utilisant l'ALU, on se souviendra des lois de De Morgan et il suffira de mettre les signaux *nx*, *ny* et *no* à *1*. Pour calculer *AND(x,y)*, tous les signaux de contrôle sont mis à *0*.

CHAPITRE 9

# Compléments d'arithmétique

Avant d'aborder d'autres opérations arithmétiques que l'addition et la division, il est intéressant de voir comment python supportent les nombres en notation binaire. Python supporte à la fois les conversions de décimal en binaire et vice-versa ainsi que les fonctions booléennes.

En python, on peut facilement entrer un nombre en représentation binaire en le préfixant par *0b* et l'inverse avec la fonction *bin* comme dans l'exemple ci-dessous.

```
a=0b00100111
print(a)    # affiche 39
print(bin(39)) # affiche 0b100111
```

Le langage python supporte également les opérations booléennes bit à bit. Les principales sont listées ci-dessous :

- En python *AND(a,b)* s'écrit `a & b`
- En python *OR(a,b)* s'écrit `a | b`
- En python *NOT(a)* s'écrit `~ a`
- En python *XOR(a,b)* s'écrit `a ^ b`

Il est aussi possible de demander à python d'effectuer des décalages à gauche et à droite. Ainsi, `x << p` décale la représentation binaire de `x` de *p* positions vers la gauche. De la même façon, `y >> p` décale la représentation binaire de `y` de *p* positions vers la droite.

Ces notations nous seront utiles pour présenter certains algorithmes dont ceux de la multiplication et de la division.

## 9.1 Multiplication des naturels

Dans le chapitre précédent, nous avons vu les opérations de base qui sont l'addition et la soustraction. Pour supporter la multiplication, nous pourrions construire une table de vérité et utiliser des portes *AND*, *OR* et *NOT*. Malheureusement, ce serait assez fastidieux pour supporter une multiplication sur 32 bits. Nous allons travailler comme pour l'addition, c'est-à-dire essayer de séparer la multiplication en une suite de calculs simples. Pour l'addition, nous avions pu travailler sur des opérations sur un bit. Malheureusement nous ne pourrons pas faire de même pour la multiplication. Par contre, il est assez facile de se rendre compte qu'une multiplication est une série d'additions. Comme nous savons déjà comment construire ces additions, nous allons pouvoir nous appuyer sur elles pour construire des circuits permettant de multiplier deux nombres entiers.

L'opération de multiplication $a \times b$ prend deux arguments. Le premier, $a$ est appelé le multiplicateur. Le second, $b$ est appelé le multiplicande. Le résultat de la multiplication est appelé le produit. La multiplication se définit sur base de l'addition :

$$a \times b = \overbrace{b + b + ... + b}^{a\ fois}$$

La multiplication et la division étant des opérations complexes, le livre de référence a choisi des les supporter par du logiciel. Il est intéressant de construire ces algorithmes simples en python de façon à bien comprendre comment ces opérations sont réalisées. Les ordinateurs modernes contiennent bien entendu des circuits électroniques qui implémentent ces opérations arithmétiques de façon efficace.

Pour l'opération de multiplication, un point important à prendre en compte est que la multiplication de deux nombres encodés sur *n* bits retourne un nombre qui peut nécessiter jusqu'à $2 \times n$ bits. Pour s'en convaincre, il suffit de considérer les naturels encodés sur 8 bits. Le carré du plus grand de ces naturels, 11111111 (255 en décimal), vaut 65025 dont la représentation binaire est 1111111000000001.Lorsque l'on calcule $A_{n-1}A_{n-2}...A_2A_1A_0 \times B_{m-1}B_{m-2}...B_2B_1A_0$, le résultat est stocké sur $m + n$ bits.

Avant d'aborder la multiplication en général, il est intéressant de considérer la multiplication par une puissance de 10. En notation décimale, pour multiplier le nombre $C_{n-1}C_{n-2}...C_2C_1C_0$ par $10^k$, il suffit d'insérer k fois le chiffre *0* à droite du nombre de façon à obtenir $C_{n-1}C_{n-2}...C_2C_1C_0\ \overbrace{0..0}^{k\ fois}$.

Avant d'aborder la multiplication binaire, regardons le cas particulier de la multiplication d'un nombre par 2. Si $B_nB_{n-1}..B_2B_1B_0$ est un naturel en notation binaire, alors on peut facilement calculer le double de ce naturel en décalant tous les bits d'une position vers la gauche. Mathématiquement, on pourrait écrire que $2 \times B_nB_{n-1}..B_2B_1B_0 = B_nB_{n-1}..B_2B_1B_00$. Cette relation est correcte et peut s'étendre à toute puissance positive de *2*. Ainsi, $2^k \times B_{n-1}B_{n-2}..B_2B_1B_0 = B_{n-1}B_{n-2}..B_2B_1B_0\overbrace{00..0}^{k}$.

---

**Note :** Les décalages sont plus rapides que les multiplications

Les opérations de décalage beaucoup plus rapide que la multiplication entière, mais il faut les utiliser correctement. Lorsque l'on manipule des nombres stockés sur un nombre fixe de bits, il faut être attentif à deux points. Tout d'abord, le résultat de la multiplication doit pouvoir être stocké sur le même nombre de bits que l'opérande. Ainsi, si l'on travaille en représentant des naturels sur 8 bits, alors on peut calculer $2 \times 37$ en décalant *00100101* vers la gauche, ce qui donne 01001010. Par contre, le décalage de *00100101* de trois positions vers la gauche donne comme résultat 00101000, c'est-à-dire 40 en notation décimale.

Le deuxième problème est que cette technique ne fonctionne pas avec tous les entiers signés. Considérons cette fois les quartets. Le quartet *1011* représente la valeur $-5$ en notation décimale. Si on décale ce quartet d'une position vers la gauche, on obtient *0110* qui correspond à la valeur positive 6. Les décalages sont donc à utiliser avec précautions.

---

Sur base de la définition de la multiplication comme une séquence d'additions, on pourrait utiliser un algorithme simple comme celui qui est présenté ci-dessous.

```
def mult(multiplicande,multiplicateur):
    produit=0
    for i in range(multiplicateur):
        produit = produit + multiplicande
    return produit
```

Cet algorithme est inefficace lorsque le multiplicateur est grand. Son temps d'exécution augmente avec le multiplicateur. Comme la multiplication est commutative, on pourrait l'accélérer en comparant les deux facteurs à multiplier comme dans le code ci-dessous.

```
def mult(multiplicande,multiplicateur):
    produit=0
    if (multiplicateur<multiplicande) :
        for i in range(multiplicateur):
            produit = produit + multiplicande
    else:
        for i in range(multiplicande):
            produit = produit + multiplicateur
    return produit
```

Cette approche reste inefficace. Essayons de l'améliorer. Prenons comme exemple la multiplication $123 \times 456$ en notation décimale. Celle-ci peut également s'écrire $123 \times (6 \times 10^0 + 5 \times 10^1 + 4 \times 10^2)$. Pour simplifier la discussion, considérons le cas simple où le multiplicande, bien qu'étant en notation décimale, ne contient que des chiffres *1* et *0*.

Lorsque l'on calcule $123 \times 101$, on calcule en fait $123 \times (1 \times 10^0 + 0 \times 10^1 + 1 \times 10^2)$. En distribuant, on obtient $123 \times 1 \times 10^0 + 123 \times 0 \times 10^1 + 123 \times 1 \times 10^2$. Cette séquence d'additions peut se représenter graphiquement comme dans Fig. 9.1. Cette représentation nous permet d'insister sur deux points importants de la réalisation de cette multiplication « en calcul écrit ». Premièrement, à chaque étape on multiplie le multiplicande par un chiffre du multiplicateur. Deuxièmement, multiplier le multiplicande par une puissance de dix revient à le décaler vers la gauche. Prenons un second exemple en notation binaire avec deux naturels sur 4 bits : 11 en notation décimale dont

```
   123
 * 101
000123
000000
+ 012300
 012423
```

FIG. 9.1 – Une multiplication en notation décimale

la représentation binaire est 1011 et 10 dont la représentation binaire est 1010. Leur produit vaut 110 en notation décimale. Lorsque l'on multiplie ces deux quartets, on obtient un résultat qui est stocké sur un octet. Le résultat est obtenu par une succession d'additions sur 8 bits. Il s'obtient en utilisant un algorithme qui fonctionne comme suit :

0. Initialisation. Le résultat est initialisé à 0.
1. Etape 0. L'algorithme examine le bit de poids faible du multiplicateur ($B_0$). Celui-ci valant zéro, on ajoute la valeur $0 \times 2^0 \times 00001011 = 00000000$ au résultat intermédiaire.
2. Etape 1. L'algorithme examine le bit $B_1$ du multiplicateur. Celui valant *1*, nous devons ajouter au résultat intermédiaire la valeur $1 \times 2^1 \times 00001011$. En notation binaire, les multiplications par une puissance de deux se réalisent facilement via un décalage vers la gauche. Dans ce cas, $2^1 \times 00001011 = 00010110$. Le résultat intermédiaire vaut maintenant 00010110.
3. Etape 2. L'algorithme examine le bit $B_2$ du multiplicateur. Celui-ci valant zéro, on ajoute la valeur $0 \times 2^2 \times 00001011 = 00000000$ au résultat intermédiaire.
4. Etape 3. L'algorithme examine le bit de poids fort ($B_3$) du multiplicateur. Celui valant *1*, nous devons ajouter au résultat intermédiaire la valeur $1 \times 2^3 \times 00001011$ soit 01011000. Le résultat intermédiaire vaut maintenant 01101110. C'est le résultat final.

Fig. 9.2 présente la succession d'additions de façon plus lisible. Pour implémenter cette addition dans un programme python, nous pouvons utiliser le principe décrit ci-dessus avec trois variables :

— le multiplicande
— le multiplicateur
— le produit intermédiaire

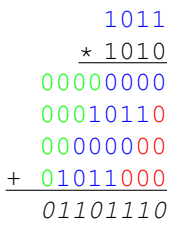



FIG. 9.2 – Une multiplication en notation binaire

Pour multiplier le multiplicateur à chaque étape, il suffit de le décaler d'un bit vers la gauche. De la même façon, pour pouvoir tester successivement les différents bits du multiplicande, il suffit de le décaler d'un bit vers la droite à chaque étape. Le programme ci-dessous présente cet algorithme en python.

```
def lowest_order_bit(x):
    return x & 0b0001

def mult(multiplicande,multiplicateur):
    resultat=0b00000000
    for i in range(4):
        if lowest_order_bit(multiplicateur) == 1:
            resultat = resultat + multiplicande
        multiplicande = multiplicande << 1
        multiplicateur = multiplicateur >>1
    return resultat
```

Cet algorithme est beaucoup plus efficace que notre première solution naïve. Le nombre d'additions qui sont calculées dépend uniquement du nombre de bits utilisés pour représenter chacun des nombres à additionner. Sur un ordinateur, ce nombre de bits est une constante.

Il est facile d'étendre cet algorithme pour supporter les entiers positifs et négatifs. Le plus simple est de d'abord déterminer le signe du résultat et ensuite d'utiliser l'algorithme ci-dessous pour multiplier les valeurs absolues des nombres. Pour rappel, la multiplication de deux nombres de même signe donne un résultat positif tandis que le résultat est négatif si ils sont de signes opposés.

### 9.1.1 Exercices

1. En utilisant l'algorithme ci-dessus, calculez $7 * 9$.
2. En utilisant l'algorithme ci-dessus, calculez $(-4) * (-5)$.

2. En utilisant l'algorithme ci-dessus, calculez $(-8) * (11)$.

---

**Note :** Représentation des entiers en python

Les ordinateurs utilisent un nombre fixe de bits pour stocker les entiers. En notation en complément à deux, le plus grand nombre positif que l'on peut stocker sur 32 bits est $2^{31} - 1$, soit 2147483641. Si une opération arithmétique retourne une résultat qui est supérieur à 2147483641, celui-ci ne pourra plus être stocké sur 32 bits. La plupart des processeurs indiquent alors un problème de dépassement de capacité qui peut être traité par le logiciel qui réalise le calcul. Si ce dépassement n'est pas traité, le calcul sera erroné.

Le langage python ne souffre pas de ce problème car ce langage utilise un nombre variable de bits pour stocker les nombres entiers. Il ajuste le nombre de bits nécessaire en fonction du nombre à stocker. On peut observer ce

comportement en utilisant la fonction sys.getsizeof du module sys. Cette fonction retourne la zone mémoire occupée par un type primitif ou un objet en python.

Grâce à cette fonction, on peut observer qu'un programme python utilise *28* octets pour stocker un entier mais que la zone mémoire nécessaire augmente avec la valeur de cet entier. Au-delà de $2^{30}$, un entier occupe 32 bytes en python et la représentation du nombre $2^{900}$ nécessite *148* octets en mémoire.

Cette adaptation dynamique de la taille des entiers dans python permet de réaliser des calculs exacts avec les nombres entiers, quel que soit le nombre considéré. Tous les langages de programmation ne sont pas aussi précis. Vous verrez l'an prochain qu'en Java et en C par exemple les entiers sont stockés sur un nombre fixe de bits, ce qui vous posera différents problèmes liés à des dépassements de capacité.

## 9.2 Division euclidienne

La quatrième opération arithmétique de base sur les naturels est la division euclidienne. Cette division prend deux arguments : un *dividende* et un *diviseur*. Elle retourne deux entiers : un quotient et un reste. Formellement, la relation entre ces trois entiers est : $dividende = diviseur \times quotient + reste$. Nous nous concentrerons sur la division euclidienne appliquée aux naturels même si elle peut évidemment s'appliquer aux entiers positifs et négatifs.

**Note :** Division entière en python

Le langage python permet de réaliser les divisions entières de différentes façons. Si les variable `x` et `y` contiennent des nombres entiers, alors `x / y` et `x // y` (depuis python 3.5) retournent le quotient de la division euclidienne. Le reste de la division euclidienne s'obtient en utilisant l'expression `x % y`. Il est aussi possible d'utiliser la fonction divmod(a,b) qui retourne le quotient et le reste de la division entière entre `a` et `b`.

Avant d'aborder cette division euclidienne, il est intéressant de discuter le cas particulier de la division par deux ou par une puissance de 2. En représentation binaire, la division par deux d'un naturel revient à décaler sa représentation binaire d'une position vers la droite. A titre d'exemple, considérons le quartet $0110$ qui représente le nombre *6* en notation décimale. Lorsque l'on décale ce quartet d'une position vers la droite, on obtient la séquence $0011$ qui est bien la représentation binaire de *3*. Ce décalage vers la droite fonctionne également pour les puissances de deux. Ainsi, pour obtenir le quotient de la division du nombre décimal *109* représenté par l'octet $01101101$ par $2^3$, il suffit de décaler la séquence de bits de trois positions vers la droite. Ce décalage donne $00001101$ qui est bien la représentation de *13*.

**Note :** Division rapide d'un entier par une puissance de deux

Le décalage fonctionne pour les naturels, mais pas pour les entiers en notation en complément à deux. Pour s'en rendre compte, considérons la valeur *-5* dont la représentation binaire est $11111011$. Si on décale cette représentation binaire de deux positions vers la droite, on obtient $00111110$ qui est la représentation binaire du nombre $+62\ldots$

On pourrait imaginer de résoudre ce problème en insérant des bits de poids fort avec la valeur *1* plutôt que *0*. Dans notre exemple, cela donnerait la séquence binaire $11111110$ qui correspond à la valeur $-2$. C'est plus proche de la valeur attendue mais malheureusement incorrect. Soyez prudents lorsque vous utilisez des décalages pour remplacer des multiplications ou des divisions.

Pour illustrer la division euclidienne, considérons l'opération $1011/10$ en notation décimale. Lorsque l'on réalise cette division en calcul écrit, on réalise en fait une suite de soustractions. Analysons ce calcul étape par étape. Chaque étape nous permet d'obtenir un chiffre du quotient. Le calcul démarre en initialisant le reste à la valeur du dividende. Nous allons d'abord déterminer la valeur du chiffre des centaines du quotient, $Q_2$. Pour cela, nous essayons de soustraire $1 \times 10^2 \times diviseur$ du reste (Fig. 9.3). Comme son résultat est positif et vaut *11*, le chiffre des centaines du quotient est connu et il vaut *1*. Le reste est mis à jour à la valeur *11*. L'étape suivante est de déterminer le chiffre des dizaines du quotient. Pour cela, nous essayons de soustraire $1 \times 10^1 \times diviseur$ du reste (Fig. 9.4). Le résultat étant négatif,

```
  1011
- 1000
    11
```

$Q_2 = 1$
reste = 011

FIG. 9.3 – Première étape de la division décimale

le chiffre des dizaines du quotient doit valoir *0*. Nous pouvons maintenant réaliser la troisième soustraction pour

```
   011
- 0100
 négatif
```

$Q_1 = 0$
reste = 011

FIG. 9.4 – Deuxième étape de la division décimale

déterminer le chiffre des unités du quotient. Pour cela, nous essayons de soustraire $1 \times 10^0 \times diviseur$ du reste (Fig. 9.5). Ce résultat est positif, le chiffre des unités du quotient vaut donc *1* et le reste de notre division également. Essayons maintenant de transposer cette méthode au calcul des divisions binaires. Pour cela, considérons la division

```
   011
-  10
    1
```

$Q_0 = 1$
reste = 1

FIG. 9.5 – Première étape de la division décimale

entière de 46 par 5. Cette division euclidienne retourne comme quotient la valeur 9 avec un reste de 1.

A chaque étape, on va déterminer la valeur d'un bit du quotient en commençant par le bit de poids fort. La première étape est d'essayer de soustraire $2^4 \times diviseur$ du dividende. En notation binaire, cette valeur s'obtient facilement en décalant le diviseur de 4 positions vers la gauche. La soustraction réalisée dans la Fig. 9.6 retourne un résultat négatif. Cela indique que le bit $Q_4$ du quotient doit valoir *0*. La deuxième étape (Fig. 9.7) nous permet de déterminer la valeur du bit $Q_3$ de notre quotient. Celui-ci vaudra *1* si en soustrayant $2^3 \times diviseur$ on obtient un résultat positif. C'est le cas. Le bit $Q_3$ est donc mis à la valeur *1* et le reste prend la valeur du résultat. La troisième étape nous permet de déterminer la valeur du bit $Q_2$ du quotient. Pour cela, on essaye de soustraire $2^2 \times diviseur$ de notre dividende intermédiaire. Le résultat de cette soustraction est négatif (Fig. 9.8) et $Q_2$ prend donc la valeur zéro. La troisième étape nous permet de déterminer la valeur du bit $Q_1$ du quotient. Pour cela, on essaye de soustraire $2^1 \times diviseur$ de notre dividende intermédiaire. Le résultat de cette soustraction est négatif (Fig. 9.9) et $Q_2$ prend donc la valeur zéro. La dernière étape (Fig. 9.10) nous permet de déterminer la valeur du bit $Q_0$ de notre quotient. Celui-ci vaudra *1* si en soustrayant $2^0 \times diviseur$ du dividende intermédiaire on obtient un résultat positif. C'est le cas. Le bit $Q_0$ est donc mis à la valeur *1* et le dividende intermédiaire prend la valeur du reste de notre calcul. Le résultat final de notre division en binaire est donc :

— $quotient = 01001$
— $reste = 0001$

Cette procédure peut également s'écrire en python comme nous l'avons fait pour la multiplication. Une version de cet algorithme permettant de diviser un naturel représenté sur 8 bits par un naturel représenté sur quatre bits est repris dans le code ci-dessous. Cet algorithme peut être étendu pour supporter des nombres encodés sur un plus grand nombre de bits.

```
def div(dividende,diviseur):
    quotient=0b0000
    reste = dividende
    diviseur = diviseur << 4
    for i in range(4+1):
```

(suite sur la page suivante)

```
  00101110
- 01010000
    négatif
```

$Q_4 = 0$

FIG. 9.6 – Première étape de la division binaire

```
  00101110
- 00101000
  00000110
```

$Q_3 = 1$
reste = 00000110

FIG. 9.7 – Deuxième étape de la division binaire

```
  00000110
- 00010100
    négatif
```

$Q_2 = 0$

FIG. 9.8 – Troisième étape de la division binaire

```
  00000110
- 00001010
    négatif
```

$Q_1 = 0$

FIG. 9.9 – Quatrième étape de la division binaire

```
  00000110
- 00000101
  00000001
```

$Q_0 = 1$
Reste=00000001

FIG. 9.10 – Dernière étape de la division binaire

(suite de la page précédente)

```
        r=reste-diviseur
        if( r > 0 ):
            reste=r
            quotient = quotient << 1
            quotient = quotient | 0b0001
        else:
            quotient = quotient << 1
            quotient = quotient & 0b1110

        diviseur = diviseur >> 1
    return quotient, reste
```

La plupart des ordinateurs utilisent des circuits logiques pour calculer efficacement les divisions euclidiennes. Ces circuits permettent de diviser des entiers, positifs et négatifs. Le fonctionnement de ces circuits sort du cadre de ce cours d'introduction.

---

**Note :** Division par zéro en python

La division euclidienne n'est pas définie lorsque le diviseur vaut zéro. Pourtant, il peut arriver que l'utilisateur demande par inadvertance de réaliser une division par zéro. Dans la plupart des langages de programmation, une telle division par zéro provoque une exception. Cette exception correspond à un signal généré par le matériel pour avertir d'une erreur lors de l'exécution d'un programme. Ce signal est intercepté par le système d'exploitation. Un système d'exploitation est un logiciel spécialisé qui contrôle les interactions entre les programmes qui s'exécutent sur l'ordinateur et le matériel. Parmi les systèmes d'exploitation les plus connus, on peut citer Microsoft Windows, Linux, MacOS, ... Le système d'exploitation avertit ensuite le programme qui a tenté d'effectuer cette division par zéro. Par défaut, le système d'exploitation termine l'exécution du programme en erreur, mais il est possible dans certains langages de programmation de traiter ces erreurs de division par zéro. C'est le cas en python via le mécanisme d'exceptions. Python définit l'exception `ZeroDivisionError` qui correspond exactement à ce cas de figure.

Code source 9.1 – Division par zéro en python

```
a = ...
b = ...
try:
  a / b
except ZeroDivisionError:
  print("Erreur")
```

---

## 9.3 Opérations sur les réels

Les entiers sont des nombres importants, mais ce ne sont pas les seuls types de nombres avec lesquels nous devons réaliser des opérations mathématiques. Les réels sont nettement plus importants dans de très nombreux domaines scientifiques. Les réels sont d'ailleurs les nombres que nous manipulons le plus fréquemment, que ce soit dans la vie de tous les jours pour représenter des montants en Euros ou pour réaliser des calculs scientifiques. Les constantes mathématiques importantes comme $\pi$ (3.141592653589793) ou $e$ (2.718281828459045) sont des réels.

Quasiment tous les ordinateurs construits depuis les années 1980s ont adopté la norme IEEE 754 pour représenter les nombres réels et réaliser des opérations mathématiques sur ces nombres. Cette norme peut être vue comme la façon d'utiliser sur un ordinateur la notation scientifique à laquelle vous avez étés habituée durant vos études secondaires. Lorsque l'on doit représenter des réels très grands ou très petits, on exprime le réel sous la forme d'une mantisse et d'un exposant. La notation standard est $\pm m \times 10^p$ où $m$ est appelée la mantisse et doit être dans l'intervalle $[1, 10[$ et $p$ est l'exposant. L'avantage de la notation scientifique est qu'elle permet de manipuler efficacement de grands et de

petits nombres comme le nombre d'Avogadro, $N_A = 6.02214076 \times 10^{23}$ ou la masse de l'électron, $9.109 \times 10^{-31}$. Formellement, il n'y a pas de représentation pour le nombre *0* en utilisant la notation scientifique, mais tout le monde utilise le chiffre *0* dans ce cas.

La norme IEEE 754 permet à l'ordinateur de représenter les réels en utilisant une notation binaire qui est inspirée de la notation scientifique. Cette représentation est souvent appelée la représentation en virgule flottante. Dans cette représentation, tout nombre réel est de la forme $(-1)^s 1.mmmm \times 2^{eeee}$ où tant la mantisse (*mmm*) que l'exposant (*eeee*) sont en notation binaire.

Il est intéressant d'analyser plus en détails la représentation de la partie fractionnaire d'un nombre en binaire. Formellement, le nombre $1.B_{(-1)}B_{(-2)}B_{(-3)}...B_{(-n)}$ correspond à la valeur numérique $(1 + B_{(-1)} \times 2^{-1} + B_{(-2)} \times 2^{-2} + B_{(-3)} \times 2^{-3} + ...B_{-n} \times 2^{-n})$. On peut donc aisément convertir en nombre binaire en notation fractionnaire en sa version décimal. Ainsi, $1.1010$ correspond au nombre décimal $1 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} = 1.625$

### 9.3.1 Exercices

1. Quel est le nombre réel qui correspond à $1.0110$ en notation binaire ?

2. Si on utilise 4 bits pour représenter la partie fractionnaire du nombre $1.B_{(-1)}B_{(-2)}B_{(-3)}B_{(-4)}$, quels sont le plus petit réel et le plus grand réel que l'on peut représenter ?

3. Sans convertir les nombres $A = 1.00110$ et $B = 1.010001$, quelle relation pouvez-vous identifier entre ces deux séquences de bits ?
   — $A = B$
   — $A \neq B$
   — $A < B$
   — $A > B$
4. Quelle est la valeur décimale qui correspond au nombre binaire fractionnaire $1.1111111111111111$ ?

La norme IEEE 754 définit deux représentations pour les réels :
— la représentation en simple précision
— la représentation en double précision
Ces deux représentations diffèrent au niveau de nombre de bits qui sont utilisés pour représenter l'exposant et la partie fractionnaire du nombre en virgule flottante. En simple précision, la partie fractionnaire est encodée sur 23 bits et l'exposant sur 8. En double précision, la partie fractionnaire est encodée sur 52 bits et l'exposant sur 11 bits. Dans les deux cas, un bit est utilisé pour indiquer si le nombre est positif ou négatif. Un nombre en simple précision occupe donc 32 bits tandis qu'un nombre en double précision occupe 64 bits.

Ces deux représentations utilisent quelques astuces dans l'encodage des nombres réels. La première astuce est que si tout nombre réel est exprimé sous la forme $(-1)^s \times 1.mmmm \times 2^{eeee}$, alors il n'est pas nécessaire d'inclure le premier *1* dans la représentation binaire du nombre en virgule flottante. C'est une optimisation intéressante car elle libère un bit dans la représentation binaire de ces nombres. Cependant, il y a un nombre important que l'on ne peut pas représenter sous la forme $(-1)^s 1.mmmm \times 2^{eeee}$ : *zéro*. La norme IEEE 754 contourne cette difficulté en réservant la séquence de bits $00000..00$ pour représenter la valeur zéro.

La deuxième astuce est que le bit de poids fort de la représentation binaire contient le signe du nombre réel, *1* pour les nombres négatif et *0* pour les nombres positifs. Même si la notation en complément à deux ne contient pas de bit explicite de signe, tous les nombres entiers négatifs ont aussi leur bit de poids fort à *1*.

La troisième astuce concerne l'exposant. Pour faciliter le tri des nombres en virgule flottante sur base de leur séquence binaire, l'exposant est placé dans les bits de poids fort, juste après le bit de signe. Si l'exposant étant représenté en utilisant la notation en complément à deux, alors une séquence de bits commençant par $011111111...$ correspondrait à une valeur numériquement inférieure à $000000001...$ ce qui rendrait ces tris compliqués. La solution choisie par la notation IEEE 754 est d'encoder les exposants en utilisant un biais de 127 en simple précision (et de 1023 en double

précision). Avec ce biais, l'exposant *-1* est encodé comme la séquence de bits *0111 1110* qui correspond à la valeur décimale *126*.

La notation complète utilisée par la norme IEEE 754 est donc $(-1)^{Signe} \times (1 + Fraction) \times 2^{Exposant-Biais}$.

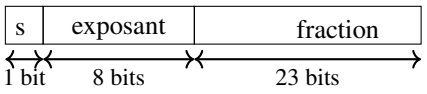



FIG. 9.11 – Notation IEEE 754 en simple précision

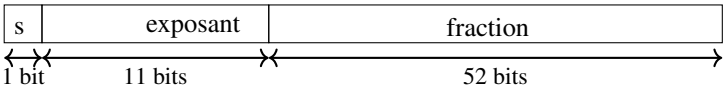



FIG. 9.12 – Notation IEEE 754 en double précision

**Note :** Python et la norme IEEE 754

Comme la plupart des langages de programmation, python supporte la norme IEEE 754. Par défaut, python utilise la notation en double précision pour tous les calculs avec des réels. La librairie python contient deux fonctions intéressantes qui permettent d'explorer la notation IEEE 754 :

- float.hex() permet de convertir un réel en notation hexadécimale sous la forme `[sign] ['0x'] integer ['.' fraction] ['p' exponent]`
- float.fromhex() réalise la conversion inverse

Ces deux fonctions permettent d'explorer différents nombres réels en virgule flottante.

```
print((0.0).hex())   # affiche 0x0.0p+0
print((4.0).hex())    # affiche 0x1.0000000000000p+2
print((1.25).hex())   # affiche 0x1.4000000000000p+0
print((1000000000.0).hex()) # affiche 0x1.dcd6500000000p+29
print(float.fromhex('1.8p+0')) # affiche 1.5
print(float.fromhex('1.fffffffffp+0')) # affiche 1.999999999985448
```

### 9.3.2 Exercices

1. Quel est le plus petit nombre positif que l'on peut représenter en double précision ?

2. Quel est le plus petit nombre négatif que l'on peut représenter en double précision ?

Lorsque l'on réalise des opérations mathématiques sur les nombres en virgule flottante, il se peut que le résultat soit trop grand ou trop petit pour être représenté en utilisant la notation IEEE 754. Dans ce cas, le circuit électronique va générer une exception ou interruption. Ce signal sera intercepté par la système d'exploitation qui avertira le programme du problème détecté.

Toutes les opérations arithmétiques peuvent être réalisées avec la notation en virgule flottante. Cependant, la notation en virgule flottante pose plusieurs problèmes qui sont liés au nombre de bits pour encoder la mantisse et l'exposant dont il est important d'être conscient. Afin de les illustrer, considérons d'abord une addition en utilisant la notation scientifique : $9.998 \times 10^2 + 2.789 \times 10^{-1}$. Nous supposerons que notre représentation décimale nous permet uniquement de stocker 4 chiffres décimaux.

La première étape pour réaliser cette addition est de ramener les deux nombres à la même puissance de dix. Nous devons donc ramener $2.789 \times 10^{-1}$ sous la forme $x \times 10^2$. Notre addition est donc $9.998 \times 10^2 + 0.003 \times 10^2$ où $0.003 \times 10^2$ est l'arrondi de $2.789 \times 10^{-1}$. Cette opération a provoqué une première perte de précision.

Nous pouvons maintenant additionner les mantisses de nos deux nombres : $9.998 + 0.003 = 10.001$. Le résultat de notre addition est $10.001 \times 10^2$, soit $1.0001 \times 10^3$. Malheureusement, ce résultat contient cinq chiffres décimaux alors que notre représentation ne permet qu'en stocker 4. Nous devons donc à nouveau arrondir la mantisse. Le résultat final de notre addition en virgule flottante $9.998 \times 10^2 + 1.789 \times 10^{-2} = 1.000 \times 10^3$. Le résultat obtenu par ce calcul est à comparer au résultat exact : $1000.0789$.

En pratique, l'ordinateur utilisera la représentation binaire des nombres pour réaliser les opérations mathématiques, mais des problèmes similaires vont se poser : la mantisse et l'exposant contiennent chacun un nombre finis de bits. A chaque étape d'un calcul, il faut potentiellement réaliser un arrondi pour que le résultat tienne dans la représentation en virgule flottante choisie. En simple précision, sachant que l'on utilise des nombres encodés sur 32 bits, on peut représenter au maximum $2^{32} = 4294967296$ réels différents. Vu la façon dont séquences de bits sont encodées, on remarque aisément que la moitié de ces nombres sont dans l'intervalle $[-1, 1]$ et l'autre moitié sert à représenter des réels dont la valeur absolue est comprise entre 1 et $2^{127}$. Dans cet intervalle, nous ne pouvons représenter que $2^{30}$ réels différents parmi l'infinité de réels qui existent.

Pour illustrer les imprécisions liées aux nombres en virgule flottante, il est intéressant de calculer les puissances de *3*. Si l'on calcule $3^{33}$ comme une multiplication d'entiers, on obtient $5559060566555523$ comme résultat. Le résultat est identique lorsque l'on calcule cette valeur avec une multiplication de réels : $5559060566555523.0$. Par contre, si l'on multiplie ce dernier nombre par 3.0, on obtient $1.6677181699666568e+16$ comme résultat alors que la valeur exacte est $16677181699666569$. Les erreurs relatives augmentent pour de plus grands nombres. Ainsi, $3^{50}$ vaut $717897987691852588770249$ lorsque le calcul est réalisé avec des entiers. En virgule flottante, le résultat obtenu est $7.178979876918526e+23$.

---

**Note :** Les opérations en virgule flottante ne sont pas toujours associatives

En mathématique, vous avez l'habitude d'utiliser la propriété d'associativité qui implique que $(a+b)+c = a+(b+c)$. Cette propriété est très pratique car elle vous permet de réaliser une opération arithmétique dans l'ordre dans lequel vous le souhaitez. En virgule flottante, cette propriété n'est pas toujours vérifiée, notamment lorsque l'on utilise des nombres réels de valeur très différentes. L'exemple ci-dessous en python devrait vous convaincre :

```
import math

a=1.234 * math.pow(10,56)
b=-a
c=5.678 * math.pow(10,-23)

print(a+b+c) # affiche 5.678e-23
print(a+(b+c)) # affiche 0.0
```

Il existe des techniques qui permettent de réaliser des calculs les plus précis possibles en virgule flottante. Certaines d'entre elles seront présentées dans le cours d'algorithmique numérique.

---

Les dépassements de capacité et les divisions par zéro peuvent provoquer des exceptions en python lorsque l'on travaille avec des réels.

```
import math

a = math.exp(1000)  # provoque OverflowError: math range error
b = 24.0 / 0.0 # provoque ZeroDivisionError: float division by zero
```

En python, le nombre `float('inf')` est utilisé pour représenter une valeur infinie. Elle pourrait être utilisée en cas de dépassement de capacité comme dans la fonction ci-dessous :

```
import math

def myexp(x):
    try:
        return math.exp(x)
    except OverflowError:
        return float('inf')
```

CHAPITRE 10

# Mémoire

Dans les deux chapitres précédents, nous avons travaillé sur des fonctions combinatoires, c'est-à-dire des fonctions dont le résultat dépend uniquement de leurs entrées. C'est une simplification de la réalité. Dans un circuit électronique, c'est un signal électrique qui représente les valeurs *0* et *1*. Il y a différentes façons de représenter des valeurs binaires avec un signal électrique. Une des plus simples est de convenir d'utiliser un potentiel positif, par exemple *+5V* pour représenter la valeur binaire *1* et *0V* pour la valeur zéro. La Fig. 10.1 représente un tel signal électrique qui vaut initialement *1*, puis passe pendant un certain temps à *0* avant de revenir à la valeur *1*. Un tel signal électrique ne se

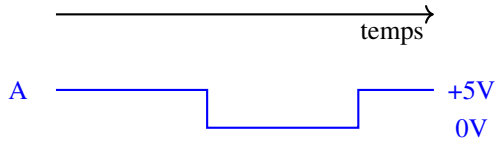



FIG. 10.1 – Signal électrique

propage pas instantanément dans un circuit électronique. En pratique, il circule à une vitesse proche de celle de la lumière. A titre d'illustration, considérons que ce signal se propage à une vitesse de 200.000 km/sec. Cela implique que le signal parcourt 1 mètre en 5 nanosecondes.

**Note :** Unités de mesure du temps

En informatique, on doit souvent manipuler des fractions de secondes. Il est important de bien connaître les fractions standard de la seconde.

| nom | abréviation | durée en secondes |
|---|---|---|
| seconde | $s$ | 1 |
| milliseconde | $ms$ | $10^{-3}$ |
| microseconde | $\mu s$ | $10^{-6}$ |
| nanoseconde | $ns$ | $10^{-9}$ |
| picoseconde | $ps$ | $10^{-12}$ |

Dans un circuit électronique, il n'est pas impossible que le signal dans deux parties du circuit suive des chemins de longueurs différentes. Considérons la situation représentée en Fig. 10.2. Imaginons que le signal *C* doit parcourir un chemin plus long que celui des signaux *A* et *B*. Initialement, les signaux *A* et *B* valent *0*. Après quelque temps, les signaux *A* et *C* passent à la valeur *1*, mais le signal *C* est un peu retardé par rapport au signal *A*. La Fig. 10.3 présente l'évolution de ces signaux et leur valeur juste avant les portes *AND* et *OR*. Remarquez que le signal *C* est un peu retardé par rapport au signal *A*. Le même raisonnement s'applique lorsque l'on prend en compte le fait qu'une porte logique ne réagit pas instantanément à une modification de son signal d'entrée. En analysant le signal de sortie (Fig. 10.3), on

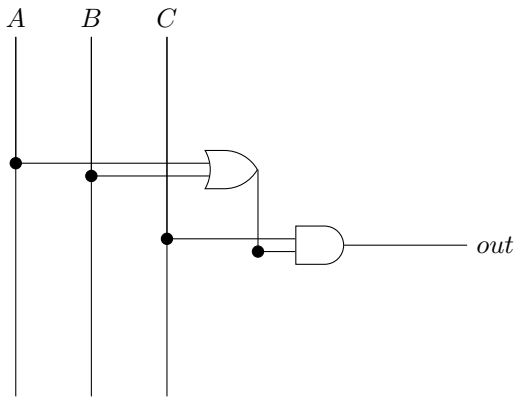



FIG. 10.2 – Un circuit simple à trois entrées

remarque que les délais différents pour les signaux *A* et *B* ont provoqué un court changement de valeur dans le signal de sortie. Cela peut poser des problèmes si ce signal doit ensuite passer dans d'autres circuits et un seul processeur peut contenir des millions de portes logiques.

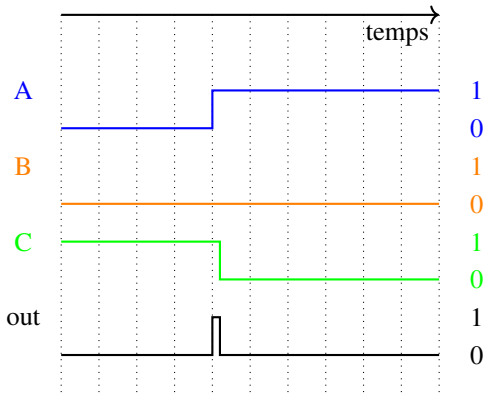



FIG. 10.3 – Evolution des signaux d'entrée et de sortie

## 10.1 Le signal d'horloge

Pour éviter ces problèmes, la plupart des ordinateurs utilisent un signal d'horloge qui régule le fonctionnement des différents circuits qui sont utilisés. Ce signal d'horloge est un signal périodique, c'est-à-dire un signal qui répète sa valeurs à des intervalles réguliers. Les fonctions trigonométriques sont des exemples de signaux périodiques. En informatique on travaille avec des signaux binaires. On dira qu'un signal $S(t)$ sera périodique si il existe un réel $P$ qui est tel que : $\forall t, S(t+P) = S(t)$. $P$ est appelé la période du signal et s'exprime en secondes. La Fig. 10.4 présente un exemple de signal binaire périodique aussi appelé signal d'horloge. La période d'un signal périodique s'exprime en

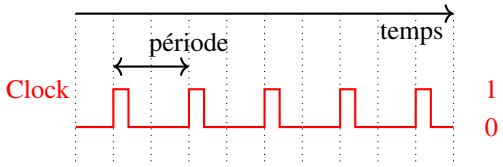



FIG. 10.4 – Signal d'horloge d'un ordinateur

secondes. Souvent, plutôt que de donner la période du signal on préfère indiquer sa fréquence. La fréquence ($f$) d'un signal est définie comme étant l'inverse de sa période : $f = \frac{1}{P}$. Si la période est exprimée en secondes, alors la fréquence est exprimée en Hz (Hertz, du nom du découvreur des ondes électromagnétiques). En pratique, on rencontrera plus fréquemment des fréquences exprimées en $MHz$ et $GHz$.

---

**Note :** Unités de mesure de la fréquence

| fréquence | abréviation | durée d'une période (s) |
|---|---|---|
| hertz | $Hz$ | 1 |
| kilohertz | $kHz$ | $10^{-3}$ |
| Mégahertz | $MHz$ | $10^{-6}$ |
| Gigahertz | $GHz$ | $10^{-9}$ |
| Térahertz | $THz$ | $10^{-12}$ |

---

Un tel signal d'horloge permet de contrôler le fonctionnement des circuits combinatoires en forçant ceux-ci à ne retourner leur résultat que lorsque le signal d'horloge est à la valeur *1*. Cela peut se réaliser en ajoutant simplement une porte *AND* qui est combinée avec le signal de sortie comme représenté en Fig. 10.5. Grâce à ce signal d'horloge et

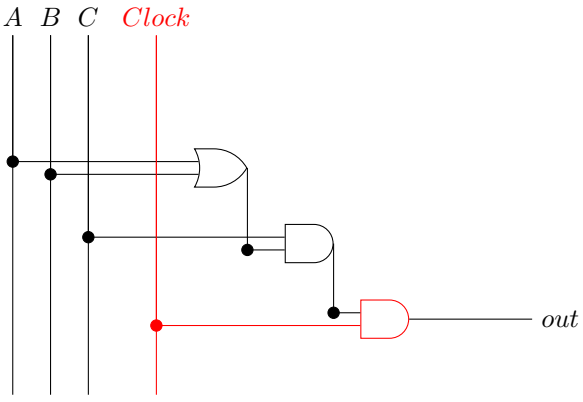



FIG. 10.5 – Un circuit simple à trois entrées contrôlé par une horloge

à la porte *AND* que nous avons ajoutés, nous pouvons maintenant observer (Fig. 10.6) que la valeur du signal de sortie

(*out*) ne se modifie pas malgré le délai dans le signal *C*. En pratique, on choisira la période de l'horloge de façon à ce qu'elle soit supérieur à la différence de délais de propagation dans le circuit électronique. On veillera également à ce que le signal d'horloge lui-même soit acheminé suivant le chemin le plus court vers tous les circuits qu'il contrôle. L'horloge va jouer un rôle très important dans le fonctionnement des ordinateurs comme nous le verrons dans les

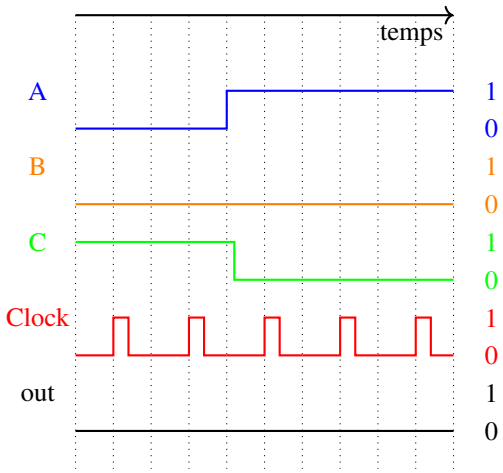



FIG. 10.6 – Evolution des signaux d'entrée et de sortie

prochains chapitres. Un autre élément essentiel du fonctionnement des ordinateurs est la possibilité de mémoriser une information. Si le fonctionnement de l'ordinateur est rythmé par un signal d'horloge, comment peut-on mémoriser une valeur binaire d'un cycle d'horloge à l'autre ?

## 10.2 La mémorisation d'un bit

Le livre de référence construit cet élément de mémoire en démarrant d'un data flip-flop (DFF). Ce DFF est un circuit qui prend deux entrées : *in* et un signal d'horloge et a une sortie : *out*. Ce circuit est conçu de façon à ce que sa sortie au cycle d'horloge *t* corresponde à la valeur de l'entrée au cycle d'horloge *t-1*. Ce circuit est représenté en Fig. 10.7. Pour comprendre le fonctionnement de ce circuit, il est intéressant d'analyser comment sa sortie évolue en fonction

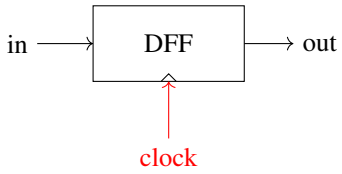



FIG. 10.7 – Un data flip-flop

de son entrée et du signal d'horloge. Lorsque le signal d'entrée change, le signal de sortie attend le prochain cycle de l'horloge pour changer de valeur. On observe donc un décalage dans le temps entre le signal d'entrée et le signal de sortie. Ce décalage est intéressant dans certains applications, mais il serait nettement plus utile de pouvoir mémoriser un bit d'informatique dans un flip-flop de ce type. On pourrait se dire que pour mémoriser une information pendant plusieurs cycles d'horloge, il suffira de prendre la sortie d'un data flip-flop et de la connecter à son entrée comme en Fig. 10.9. Malheureusement, un tel circuit pose deux problèmes. Premièrement, puisque sa sortie dépend avec un délai de son entrée, il n'est pas possible de le forcer à mémoriser une valeur donnée à un instant donné. Deuxièmement, au niveau électronique, il est compliqué de connecter deux signaux simultanément sur une entrée puisque cela revient à créer un court-circuit au niveau électrique ...

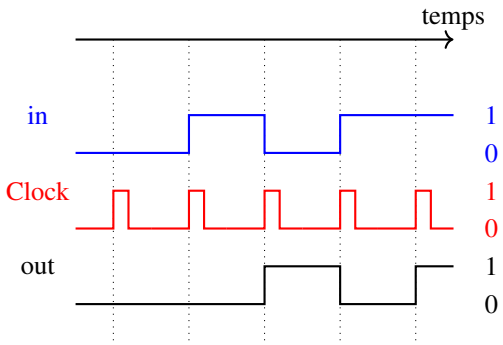



FIG. 10.8 – Data flip-flop - exemple

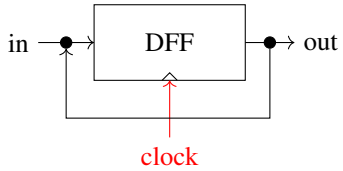



FIG. 10.9 – Un circuit pour mémoriser un bit ?

La solution pour résoudre ce problème est d'utiliser un multiplexeur en amont du flip-flop pour choisir entre le signal d'entrée *in* et le signal de sortie qui est bouclé comme entrée pour le flip-flop. Ce multiplexeur est commandé par un signal *load* qui permet de forcer le chargement du bit du signal *in* dans le flip-flop. Lorsque *load* vaut *1*, le signal *in* est mémorisé par le flip-flop durant le cycle d'horloge. Lorsque *load* vaut *0*, le flip-flop reçoit sa sortie en entrée et celle-ci est conservée pour le cycle d'horloge suivant. Ce registre est présenté en Fig. 10.10. Cette mémoire d'un bit va jouer

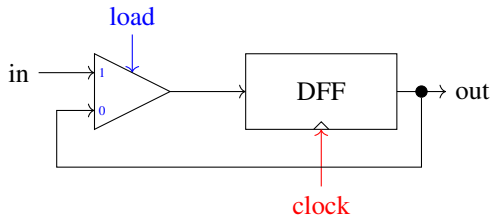



FIG. 10.10 – Un registre permettant de mémoriser un bit

un rôle très important dans la construction de tous les éléments de mémoire d'un ordinateur. Pour pouvoir la réutiliser dans d'autres circuits, nous allons lui choisir une représentation standard (Fig. 10.11). Dans la Fig. 10.11, le triangle rouge rappelle la présence du signal d'horloge qui est présent dans tous les circuits de mémoire. Pour simplifier les prochaines représentations graphiques, nous le retirerons souvent, mais si il restera bien présent en réalité.

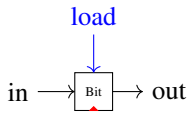



FIG. 10.11 – Une mémoire pour un bit

## 10.3 Un registre pour mémoriser un quartet

Nous pouvons maintenant utiliser cet élément de mémoire pour construire un registre qui permet de mémoriser la valeur d'un quartet. Ce circuit a six entrées :

- le signal d'horloge
- le signal *load*
- le bit $B_3$ du quartet à mémoriser
- le bit $B_2$ du quartet à mémoriser
- le bit $B_1$ du quartet à mémoriser
- le bit $B_0$ du quartet à mémoriser

et quatre sorties :

- le bit $Out_3$ du quartet mémorisé
- le bit $Out_2$ du quartet mémorisé
- le bit $Out_1$ du quartet mémorisé
- le bit $Out_0$ du quartet mémorisé

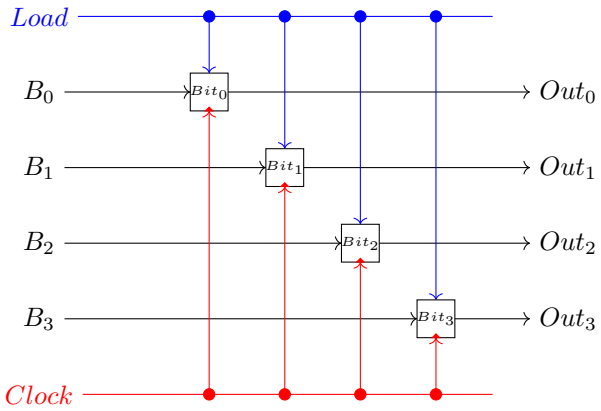



FIG. 10.12 – Un registre à 4 bits

De la même façon, on peut construire des registres qui permettent de stocker un octet ou un mot de 16, 32 voire même 64 bits. Dans la suite de ce chapitre, nous représenterons un tel registre sous la forme d'un rectangle.

De tels registres s'utilisent généralement en groupe. Un microprocesseur contient plusieurs registres et une mémoire peut stocker des millions ou même des milliards d'octets. A titre d'illustration, considérons un bloc de registre qui stocke quatre bits. Ce bloc de registres comprend bien entendu quatre registres qui stockent chacun un bit. Outre le signal d'horloge (non représenté en Fig. 10.12), nous devons connecter le signal *load*, les 4 bits d'entrée et les 4 bits de sortie à cet ensemble de registres. Le signal d'horloge peut être directement connectés à chacun de nos quatre registres. Pour la connexion des bits d'entrée et des bits de sortie, nous devons trouver une solution qui nous permet d'identifier le registre dans lequel nous souhaitons effectuer une opération de lecture ou d'écriture. Pour cela, nous devons identifier chacun de nos registres avec un numéro. Le premier registre a *0* comme identifiant, le deuxième *1*, le troisième *2* et le dernier *3*. Comme nous nous avons 4 identifiants, il nous suffit de deux signaux binaires pour encoder la valeur de l'identifiant du registre concerné. Ces deux signaux s'ajoutent au bloc de registre représenté en Fig. 10.13. Ils doivent nous permettre de sélectionner le registre dans lequel l'information arrivant est écrite ou lue en fonction de la valeur du signal *load*. Cet identifiant est généralement appelé une adresse. Dans notre exemple, nous avons 4

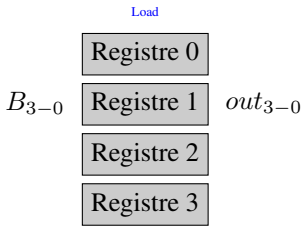



FIG. 10.13 – Eléments d'un registre à 4 bits

adresses possibles qui sont encodées sur deux bits.

Commençons par analyser l'opération de lecture à travers notre bloc de quatre registres. A chaque cycle d'horloge, chaque registre envoie sur sa sortie la valeur qu'il a stocké. Pour choisir comme sortie globale du bloc de 4 registres une de ces valeurs, il nous suffit d'utiliser un multiplexeur auquel nos quarte registres sont connectés. Ce multiplexeur est commandé par les deux bits d'adresse. Il est représenté sur la droite de la Fig. 10.14.

Analysons maintenant l'opération d'écriture dans un de nos quatre registres. La valeur à enregistrer arrive via les signaux $B_3B_2B_1B_0$. Elle peut être connectée à nos quatre registres. L'important est de pouvoir activer le signal *load* uniquement sur le registre dans lequel l'information doit être stockée. Lorsque l'adresse est *00* en binaire, le signal *load* doit activer le registre *0*. De même, c'est le registre *3* qui doit être activé pour l'adresse *11* en binaire. Nous avons déjà résolu un problème similaire il y a quelques chapitres en utilisant un démultiplexeur. Celui-ci est connecté à l'entrée *load* et commandé par les deux bits d'adresse. Ses quatre sorties sont attachées aux quatre entrées *load* de nos registres. Ce démultiplexeur est représenté dans la partie gauche de Fig. 10.14. Ce schéma général peut se reproduire

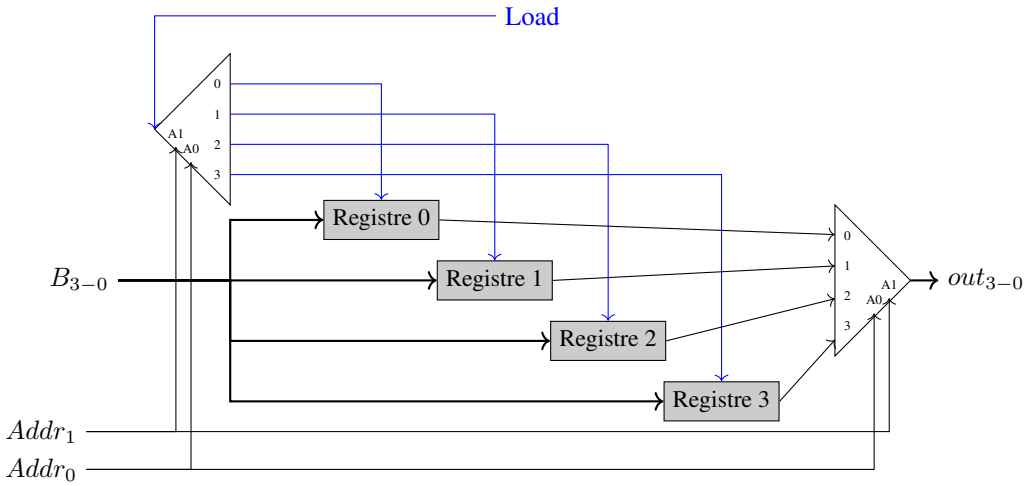



FIG. 10.14 – Un registre à 4 bits

sans difficulté pour des mémoires de plus grande capacité. La seule limitation sera technologique et liée au nombre de registres et de multiplexeurs/démultiplexeurs que l'on pourra placer sur une surface donnée.

A titre d'exemple, regardons comment construire un bloc de huit registres. Ce bloc doit avoir en entrée les signaux suivants :

— les données à mémoriser ($B_3B_2B_1B_0$ pour des quartets)
— le signal d'horloge (non représenté sur les figures)
— le signal *load*
— 3 bits pour indiquer l'adresse du registre où il faut lire/écrire

Pour construire cette mémoire contenant huit registres, nous pouvons partir du bloc de quatre registres que nous venons de construire. Celui-ci peut être schématisé comme en Fig. 10.15. Grâce à ce bloc de quatre registres, nous pouvons

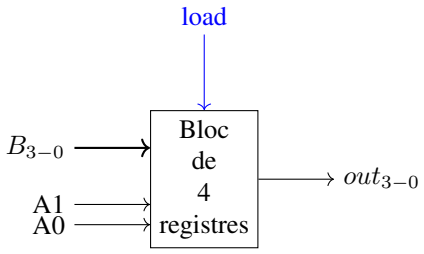



FIG. 10.15 – Représentation schématique d'un bloc de 4 registres

facilement construire notre bloc de huit registres. Il suffit de considérer que l'un des blocs de registres correspond aux adresses 0 à 3 et le second aux adresses allant de 4 à 7. En notation binaire, les adresses correspondant au premier bloc vont de 000 à 011 tandis que celle du second bloc vont de 100 à 111. On peut donc utiliser le bit de poids fort de l'adresse ($A_2$) pour choisir entre le premier bloc de registres et le second. Pour l'opération de lecture, il suffit de connecter un multiplexeur connecté aux sorties et de le commandé en utilisant le bit de poids fort de l'adresse. Ce bit de poids fort doit aussi commander le démultiplexeur se trouvant sur la gauche de Fig. 10.16 pour acheminer le signal *load* vers le *bloc 0* ou le *bloc 1*. Ce schéma général peut se reproduire sans difficulté pour des mémoires de plus grande

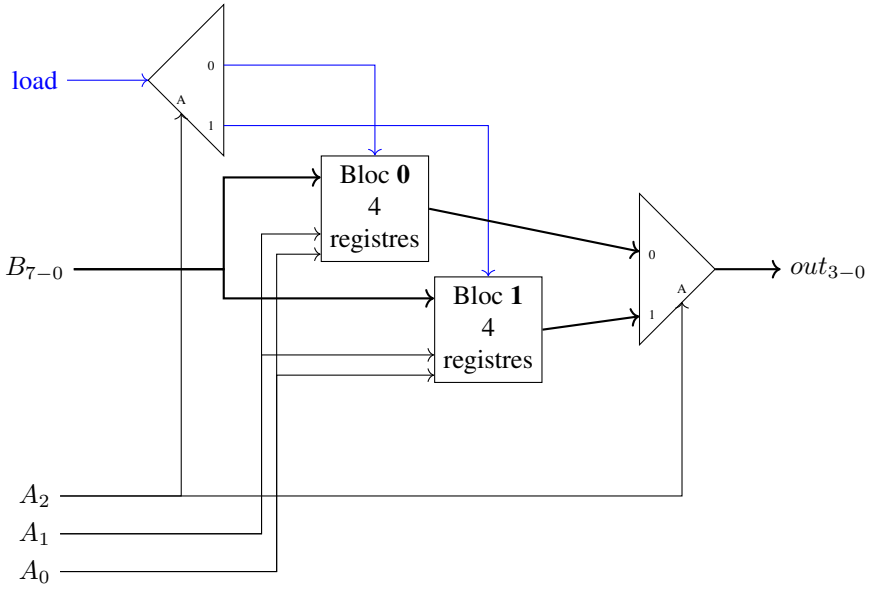



FIG. 10.16 – Un bloc de 8 registres

capacité. La seule limitation sera technologique et liée au nombre de registres et de multiplexeurs/démultiplexeurs que l'on pourra placer sur une surface donnée.

### 10.3.1 Exercice

Il est souvent nécessaire de compter le nombre de cycles d'horloge qui se sont écoulés depuis un instant donné. Parmi les circuits que vous devez réaliser pour cette mission, l'on retrouve un compteur. Celui que vous devez réaliser a une sortie sur 16 bits et quatre entrées :

— un entier sur 16 bits
— un signal de contrôle *load*
— un signal de contrôle *inc*
— un signal de contrôle *reset*

Ces différents signaux de contrôle permettent de forcer le compteur à réaliser certaines opérations. Si *reset* est mis à *1* durant un cycle d'horloge, alors la sortie du compteur doit valoir *0* durant le cycle suivant. Ce signal de contrôle permet donc de réinitialiser le compteur.

Si *inc* est mis à *1* durant un cycle d'horloge, alors la sortie durant le cycle d'horloge suivant sera celle du cycle d'horloge courant incrémentée d'une unité. C'est le mode de fonctionnement normal du compteur.

Si *load* est mis à *1* durant un cycle d'horloge, alors le compteur lit la valeur en entrée et c'est cette valeur qui sera retournée sur la sortie du compteur durant le cycle d'horloge suivant.

La Fig. 10.17 présente l'évolution dans le temps d'un compteur à deux bits ($out_1$ est le bit de poids fort et $out_0$ le bit de poids faible) en fonction des différents signaux de contrôle. On suppose dans cet exemple que les deux signaux d'entrée sont mis à *1* ainsi que $out_1$ et $out_0$. Durant le premier cycle d'horloge, tous les signaux de contrôle sont à *0* et la sortie

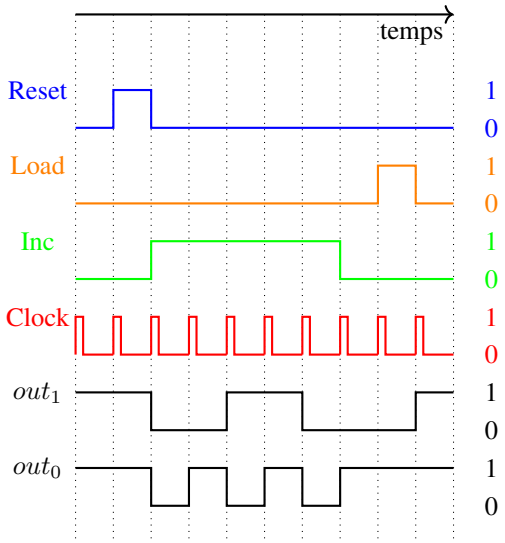



FIG. 10.17 – Evolution de la sortie du compteur en fonction du temps

garde donc sa valeur initiale. Durant le second cycle d'horloge, le signal de contrôle *reset* est activé. Cela provoque une réinitialisation des sorties $out_1$ et $out_0$ à *0*, mais celle-ci n'est visible qu'autre troisième cycle d'horloge. Durant ce troisième cycle d'horloge, le signal de contrôle *Inc* est activé. Le compteur commence à s'incrémenter. Durant le quatrième cycle, le compteur retourne la valeur binaire *01*. Durant le sixième cycle, il retourne la valeur binaire *11* qui est la valeur maximale pour un compteur sur deux bits. Comme le signal de contrôle *Inc* reste à *1* le compteur repasse à la valeur binaire *00* durant le cycle suivant. Durant le septième cycle, *Inc* est toujours activé. C'est pour cette raison que le compteur retourne la valeur binaire *01* durant le huitième cycle d'horloge. Le signal *Inc* étant désactivé durant ce cycle, le compteur ne modifie pas sa valeur qui reste inchangée pour le neuvième cycle d'horloge. Enfin, durant le dernière cycle d'horloge sur Fig. 10.17, on observe le résultat de l'activation du signal *Load* sachant que les deux entrées du compteur sont mises à *1*.

1. Quels sont, à votre avis, les circuits de base qui sont nécessaires pour construire un tel compteur ? Pensez aux différents circuits que vous avez construit durant les dernières semaines.

## 10.4 Les mémoires RAM et ROM

Les mémoires utilisées dans un ordinateur peuvent être divisées en plusieurs classes. La première distinction est entre les mémoires de type ROM (*Read-Only Memory*) et de type RAM (*Random Access Memory*). Comme son nom l'indique, une mémoire ROM est une mémoire dont le contenu ne peut qu'être lu. Le contenu de cette mémoire est écrit lors de la construction du circuit et elle ne peut jamais être modifiée. Ces mémoires sont utilisées pour stocker des données ou des programmes qui ne changent jamais, comme par exemple le code qui permet de faire démarrer un ordinateur et de lancer son système d'exploitation. Une mémoire ROM peut se représenter comme dans la Fig. 3.2. Une caractéristique important des mémoires de type ROM est que leur contenu est préservé même lorsque la mémoire

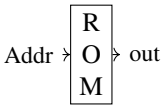



FIG. 10.18 – Une mémoire ROM

est mise hors tension. Certaines mémoires de type ROM sont dites programmables car il est possible d'effacer et de modifier leur contenu. C'est le cas par exemple des EPROM ou des EEPROM. La programmation d'un tel circuit se fait en utilisant un dispositif spécialisé.

Dans une mémoire RAM, outre les entrées relatives aux adresses, il faut aussi avoir une entrée *load* (parfois appelée *read/write*) pour déterminer si la mémoire doit lire ou écrire une donnée et une entrée *data* permettant de charger des données dans la *RAM*. Le nombre de bits d'adresses dépend uniquement de la capacité de la mémoire. En général, une adresse correspond à un octet stocké en mémoire. L'entrée *data* quant à elle peut permettre de charger des octets, des mots de 16, 32 bits ou encore plus. La Fig. 3.3 représente une mémoire RAM de façon schématique.

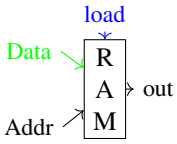



FIG. 10.19 – Une mémoire RAM

### 10.4.1 Exercice

1. En utilisant uniquement les portes logiques de base *AND*, *OR* et *NOT*, pourriez-vous construire une mémoire *ROM* de 4 octets qui contient les valeurs suivantes :
   — à l'adresse *00* : *11110000*
   — à l'adresse *01* : *10101010*
   — à l'adresse *10* : *00001111*
   — à l'adresse *11* : *01010101*

Une mémoire RAM est dite volatile. Elle ne préserve son contenu que tant qu'elle est sous tension. L'ensemble des données stockées dans une RAM disparaît dès que celle-ci est mise en tension. Il existe deux grandes familles de mémoires RAM :

— les SRAM ou mémoires RAM statiques
— les DRAM ou mémoires RAM dynamiques

En simplifiant fortement la technologie utilisée par ces deux grandes familles de mémoire RAM, on peut dire que dans une SRAM, une valeur binaire correspond à la présence ou l'absence d'un courant électrique. Pour cette raison, une mémoire SRAM consomme en permanence de l'électricité et cela limite la densité de ces mémoires, c'est-à-dire le nombre de bits que l'on peut stocker sur une surface donnée. Dans une mémoire DRAM, les bits sont stockés comme

une charge électrique présente dans un minuscule condensateur. Comme la charge d'un condensateur décroît naturellement avec le temps, il est nécessaire de réécrire régulièrement (on parle généralement de rafraîchir) les données qui sont stockées en mémoire DRAM. Ce rafraîchissement est réalisé automatiquement par un circuit électronique spécialisé. Les mémoires DRAM consomment moins d'électricité que les mémoires de type SRAM. Cela leur permet d'être beaucoup plus denses et moins coûteuses pour une même quantité de données. Par contre, les mémoires DRAM sont généralement plus lentes que les mémoires SRAM.

Les mémoires RAM jouent un rôle extrêmement important dans le fonctionnement d'un ordinateur comme nous les verrons dans les prochains chapitres. Durant les dernières décennies, elles ont fortement évolué. Sans entrer dans trop de détails technologiques, il est intéressant d'analyser trois éléments de performance de ces dispositifs de mémoire. Pour cela, nous nous basons sur les données reprises dans le livre Computer Architecture: A Quantitative Approach écrit par John Hennessy et David Patterson. Ce livre va bien au-delà des concepts qui sont vus dans ce cours, mais c'est un des livres de référence du domaine. Son premier chapitre reprend plusieurs chiffres très intéressant que nous analysons.

Une première métrique pour analyser l'évolution des mémoires RAM est de regarder leur capacité. Celle-ci s'exprime généralement en Mbits par puce. En 1980, date de la sortie de l'IBM PC-AT, une puce de mémoire DRAM contenait 64 Kbits. Cette capacité a été quadruplée en 1983 et ensuite portée à 1 Mbits en 1986. En 2000, une puce de mémoire contenait 256 Mbits. En 2016, une puce de mémoire DDR4 a une capacité de 4096 Mbits. En 33 ans, la capacité de mémoire RAM d'une ordinateur de bureau standard a donc été multipliée par 64000 ! La Fig. 10.20 résume cette évolution. La deuxième métrique que l'on peut utiliser pour comparer des mémoires est de regarder le débit auquel il

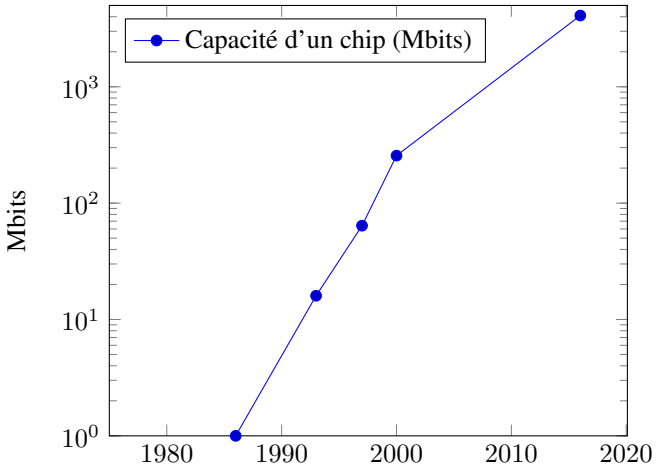



FIG. 10.20 – Evolution de la capacité des DRAMs

est possible de lire des données depuis une telle mémoire. Ce débit s'exprime en MBytes/s. En 1980, celui-ci était de seulement 13 MBytes/s. En 2000, il est passé à 1600 MBytes/s et en 2016 il a atteint 27000 MBytes/s. L'amélioration en performance reste importante, mais nettement moindre que pour la capacité des mémoires. En 33 ans, le débit ne s'est amélioré que d'un facteur d'environ 2000. Cela reste impressionnant évidemment (Fig. 10.21). La dernière métrique importante pour une mémoire RAM est son temps d'accès, c'est-à-dire le temps qui s'écoule entre le moment où l'on place une adresse en entrée de la mémoire et le moment où la valeur stockée à cette adresse est disponible. En 1980, il fallait 225 ns pour accéder à une information stockée en mémoire DRAM. En 2000, ce temps d'accès était passé à 52 ns. En 2016, les mémoires DDR4 affichent des temps d'accès de 30 ns. En 33 ans, on n'a donc gagné qu'un facteur 7 du point de vue du temps d'accès aux mémoires RAM (Fig. 10.22). Malheureusement, les limitations technologiques ont fait qu'il n'a pas été possible d'améliorer les temps d'accès des mémoires RAM aussi rapidement que leur capacité ou leur débit. Nous aurons l'occasion de discuter à la fin du cours de l'impact de ces temps d'accès relativement élevés.

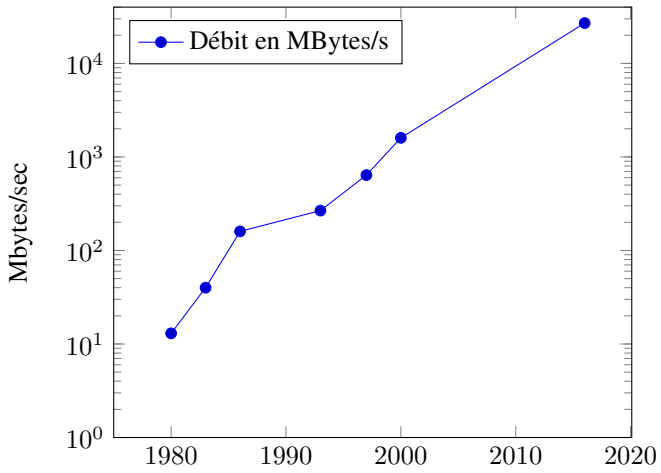



FIG. 10.21 – Evolution du débit des mémoires RAM

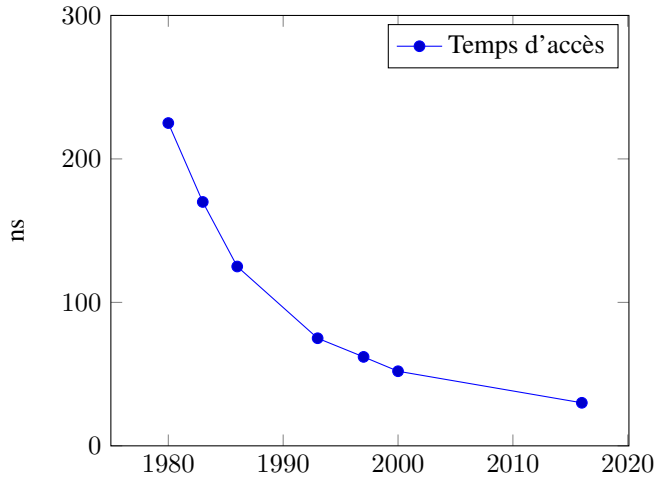



FIG. 10.22 – Evolution du temps d'accès des DRAMs

## 10.5 La construction d'un data flip-flop

Le livre a choisi de prendre le data flip-flop comme élément de base pour la construction de tous les dispositifs de mémoire. En pratique, un tel flip-flop peut aussi se construire en utilisant des portes logiques standard. Il existe différentes réalisations de tels flip-flops. Nous en considérons deux afin de comprendre leur fonctionnement. Le flip-flop le plus simple est le flip-flop RS comprenant une porte *AND*, une porte *OR* et un inverseur. Ce circuit très simple

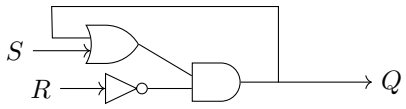



FIG. 10.23 – Représentation graphique d'un flip-flop RS AND-OR

utilise une porte *AND* et une porte *OR*. Il comporte deux entrées : *S* et *R* et a comme sortie *Q*. Pour analyser le comportement de ce circuit, commençons par discuter de ce qu'il se passe lorsque *S* et *R* valent *0*. Dans ce cas, la sortie de la porte *OR* vaut la valeur de *Q*. Il en va de même pour celle de la sortie de la porte *AND* puisque sa deuxième entrée est mise à *1*. Quelle que soit la valeur initiale de *Q*, celle-ci est conservée lorsque *R* et *S* valent *0*.

Essayons maintenant de faire passer *S* à la valeur *1* tout en gardant *R* à *0*. Si *Q* valait initialement *0*, alors la sortie *Q* passe à *1* et cette valeur reste stable. Si *Q* valait initialement *1*, alors sa valeur reste à *1*. On utilise généralement le nom *Set* pour l'entrée *S* car elle permet de faire passer la valeur de *Q* à *1*.

Analysons maintenant ce qu'il se passe si *R* passe à *1*. Dans ce cas, la sortie *Q* va nécessairement passer à *0* puisque la seconde entrée de la porte *AND* est mise à *0*. Cette valeur restera quelle que soit la valeur de *S* (*0* ou *1*). La deuxième entrée de ce flip-flop est généralement appelée l'entrée *Reset* car elle force une mise à zéro de la sortie. Il est important de noter que la valeur de *Q* reste conservée par le flip-flop lorsque *R* et *S* valent *0*.

Notre second circuit est le latch SR. Ce circuit utilise deux portes *NOR* et a deux entrées : *R* et *S*. Une caractéristique importante de ce circuit est qu'il existe une boucle entre la sortie d'un porte *NOR* et l'entrée de l'autre porte. Par ce circuit, *R* et *S* sont les entrées tandis que *Q* et $\overline{Q}$ sont les sorties

Ce circuit est assez inhabituel. N'essayez pas de le tester avec le simulateur du livre. Par contre, il est intéressant d'analyser comment ce circuit fonctionne.

Commençons par analyser le cas où *R* et *S* valent *0*. Supposons qu'initialement *Q* valait *0* et $\overline{Q}$ valait *1*. Dans ce cas, la sortie de la port *NOR* supérieure reste à *0* tandis que la sortie de la porte *NOR* inférieure reste à *1*. Si par contre *Q* valait *1* et $\overline{Q}$ valait *0*, alors *Q* reste à *1* et $\overline{Q}$ reste à *0*. On dit que lorsque *R* et *S* valent *0*, la sortie du flip-flop reste stable. Cela revient à dire que notre flip-flop garde sa valeur.

Regardons maintenant ce qu'il se passe lorsque *R* vaut *1* tandis que *S* reste à *0*. Si *Q* valait initialement *1* tandis que $\overline{Q}$ valait *0*, alors la sortie de la porte *NOR* supérieure va passer à *0*. Cette valeur va revenir dans la porte *NOR* inférieure et forcer un passage à *1* de la sortie $\overline{Q}$. Lorsque cette sortie revient dans la porte *NOR* supérieure, elle force sa sortie à *0*. Si *Q* valait initialement *0* (et $\overline{Q}$ valait *1*), rien ne change. On dit que l'entrée *R* est l'entrée *Reset* car elle permet de forcer la sortie *Q* à passer à *0*.

Regardons maintenant ce qu'il se passe lorsque *R* reset à *0* tandis que *S* passe à *0*. Si *Q* valait initialement *0* tandis que $\overline{Q}$ valait *1*, alors la sortie de la porte *NOR* supérieure va passer à *1*. Cette valeur va revenir dans la porte *NOR* inférieure et forcer un passage à *0* de la sortie $\overline{Q}$. Lorsque cette sortie revient dans la porte *NOR* supérieure, elle force sa sortie à *1*. Si *Q* valait initialement *1* (et $\overline{Q}$ valait *0*), rien ne change. On dit que l'entrée *S* est l'entrée *Set* car elle permet de forcer la sortie *Q* à passer à *1*.

Lorsque *R* et *S* valent simultanément *1*, les sorties *Q* et $\overline{Q}$ passent à *0* toutes les deux.

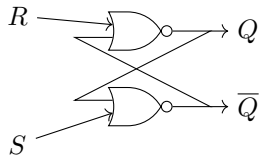



FIG. 10.24 – Représentation graphique d'un latch SR utilisant des portes NOR

### 10.5.1 Exercices

1. Il est aussi possible de construire le flip-flop RS AND-OR en connectant la sortie *Q* à la sortie de la porte *OR*. Quel est le comportement de ce flip-flop dans ce cas ?

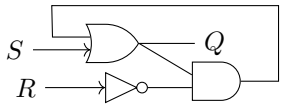



FIG. 10.25 – Variante du flip-flop RS AND-OR

2. Le latch SR peut-être construit en utilisant des portes *NOR* comme présenté ci-dessus. Il est aussi possible de construire un circuit du même type avec des portes *NAND* (Fig. 10.26). Expliquez le fonctionnement de ce circuit.

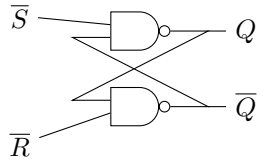



FIG. 10.26 – Représentation graphique d'un latch SR utilisant des portes NOR

CHAPITRE 11

# Langage d'assemblage

Avec la mémoire et l'ALU nous avons les briques de base qui vont nous permettre de construire un micro-processeur qui sera capable d'exécuter de petits programmes. Ce micro-processeur répond à ce que l'on appelle l''architecture de Von Neumann.

Cette architecture est composée d'un processeur (CPU en anglais) ou unité de calcul et d'une mémoire. Le processeur est un circuit électronique qui est capable d'effectuer de nombreuses tâches :

— lire de l'information en mémoire
— écrire de l'information en mémoire
— réaliser des calculs

L'architecture des ordinateurs est basée sur l'architecture dite de Von Neumann. Suivant cette architecture, un ordinateur est composé d'un processeur qui exécute un programme se trouvant en mémoire. Ce programme manipule des données qui sont aussi stockées en mémoire.

Dans notre minuscule ordinateur, toutes les informations sont stockées sous la forme de nombres binaires. Le livre a fait le choix d'utiliser des mots de 16 bits comme unité de base pour les calculs et la mémoire. On pourrait dire que notre minuscule ordinateur est un ordinateur « 16 bits ». Ce choix a plusieurs conséquences sur les données qui sont traitées par ce minuscule processeur :

— les entiers sont représentés en utilisant la notation binaire en complément à deux sur 16 bits
— chaque caractère ASCII est également stocké sous la forme d'un nombre sur 16 bits

Notre minuscule processeur ne supporte pas les nombres réels. L'utilisation de 16 bits pour représenter chaque caractère constitue un gaspillage de la mémoire puisqu'il suffit d'utiliser 8 bits pour représenter les caractères ASCII. Cependant, ce gaspillage de mémoire permet de simplifier fortement l'implémentation de notre minuscule processeur comme vous le verrez dans le prochain projet. On ne peut pas gagner de tous les points de vue.

Les ordinateurs actuels sont basés sur d'autres choix. Les entiers sont encodés sur 32 ou 64 bits tandis que les caractères sont soit encodés sur 8 bits lorsque l'on utilise la représentation ASCII historique soit sur 16 bits pour la représentation Unicode.

Le minuscule ordinateur construit dans le livre de référence a d'autres caractéristiques particulières qui simplifient sa réalisation mais ne correspondent pas nécessairement aux ordinateurs actuels. Ce minuscule ordinateur utilise deux mémoires séparées :

— une mémoire dite mémoire d'instructions contenant le code des programmes à exécuter
— une mémoire dite mémoire de données contenant les données à traiter

Ces deux mémoires ont chacune une capacité de 16384 mots de 16 bits. La plupart des ordinateurs actuels utilisent une mémoire qui contient indifféremment les données et le code machine des programmes. La mémoire d'instructions de notre minuscule ordinateur est une mémoire de type ROM. Elle est initialisée au lancement de l'ordinateur avec le programme à exécuter mais ne peut pas être modifiée par un programme. La mémoire de données elle est une mémoire de type RAM dans laquelle les programmes peuvent lire et écrire des données.

Une autre différence entre le minuscule ordinateur et un ordinateur actuel est la façon dont on accède aux données en mémoire. Le minuscule ordinateur peut uniquement lire ou écrire un mot de 16 bits à la fois à une adresse donnée en mémoire de données. Un ordinateur actuel peut lire et écrire un octet en mémoire, un mot de 16, 32 ou 64 bits voire beaucoup plus dans certains cas.

Outre ces deux mémoires, notre minuscule processeur dispose de deux registres :

— le premier, baptisé `D` est utilisé pour stocker un mot de 16 bits qui est lu depuis la mémoire ou résulte d'un calcul réalisé par l'ALU. Son nom reflète le fait qu'il stocke des données (Data en anglais)

— le second, baptisé `A`. Il a un double rôle. Tout d'abord, va il servir à stocker une donnée sur 16 bits comme le registre `D`. Son deuxième rôle est de contenir une adresse dans la mémoire de données pour permettre le chargement d'une donnée depuis cette mémoire. C'est pour cette raison qu'il est appelé le registre `A` (comme adresse).

Ces deux registres `A` et `D` sont schématiquement connectés à l'ALU qui est le coeur de notre minuscule processeur. Cela permet d'utiliser l'ALU pour réaliser différents calculs sur ces deux registres (Fig. 11.1). A côté de ces deux

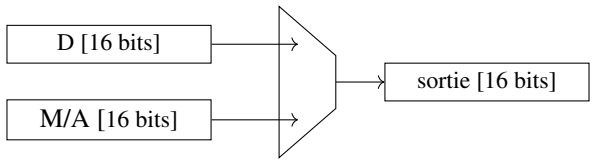



FIG. 11.1 – Les registres A, D et l'ALU

registres qui sont associés aux données, notre minuscule processeur contient également un registre baptisé `PC` (pour *Program Counter* ou compteur de programmes). Ce registre contient l'adresse de l'instruction qui est exécutée par le minuscule processeur. Nous verrons plus tard comment celui-ci est utilisé.

## 11.1 Les instructions du minuscule processeur

Avant de construire le minuscule processeur dans le projet suivant, nous devons d'abord comprendre quelles sont les instructions que celui-ci peut exécuter. Il supporte deux types d'instructions qui sont toutes les deux encodées sous la forme d'un mot de 16 bits.

### 11.1.1 L'instruction de type A

L'instruction la plus simple du minuscule microprocesseur est l'instruction de type *A* (où *A* est l'abréviation de adresse). Cette instruction permet simplement de charger un nombre binaire sur 15 bits dans le registre `A`. Dans les logiciels fournis avec le livre de référence, cette instruction s'écrit `@` suivi de la valeur à placer dans le registre `A`. La valeur passée comme argument de cette instruction de type *A* est obligatoirement un entier positif. Nous verrons plus tard comment indiquer une constante négative.

```
@1          // charge la valeur 1 dans A
@123        // charge la valeur 123, i.e. 1111011 en binaire dans A
```

Vous pouvez télécharger cet exemple depuis `asm/ex0.asm`.

Cette instruction a trois utilisations en pratique. Tout d'abord, elle permet de charger une valeur constante dans le registre `A`. Mais surtout elle est utilisée avec les instructions de type C pour soit indiquer une adresse mémoire à

laquelle une donnée doit être chargée soit une adresse mémoire où un saut doit être réalisé si une condition est vérifiée. Nous y reviendrons.

Comme toutes les instructions, l'instruction de type A est encodée sous la forme d'un mot de 16 bits. L'encodage est extrêmement simple :

— le bit de poids fort est mis à `0`
— les quinze bits de poids faible sont la valeur de l'argument de l'instruction en binaire

C'est à cause de l'encodage de l'instruction dans un mot de 16 bits que la constante qui est passée en argument doit être encodée sur 15 bits.

### 11.1.2 L'instruction de type C

Cette instruction est l'instruction « à tout faire » du minuscule processeur. Son nom vient de l'initiale de Compute (calculer). C'est elle qui permet d'utiliser toutes les fonctionnalités de l'ALU mais aussi d'implémenter des instructions conditionnelles et des boucles comme nous le verrons par après.

Plutôt que de présenter directement toutes les possibilités de cette instruction, nous allons la construire petit à petit sur base d'exemples illustratifs. Une première utilisation de l'instruction de type *C* est de charger des données depuis la mémoire vers un registre ou d'un registre vers la mémoire. Cette variante de l'instruction *C* s'écrit généralement sous la forme $dest = calcul$. Nous verrons plus tard comment réaliser un calcul en utilisant l'ALU. Commençons par observer le fonctionnement de cette instruction. La partie gauche de l'instruction de type *C* indique l'endroit où le résultat de notre calcul doit être stocké. La première destination possible est le registre *D*. Une deuxième destination possible est le registre *A*. Enfin, la troisième destination possible pour le résultat d'un calcul de l'ALU est la mémoire. Dans le minuscule assembleur, ceci est représenté en utilisant le symbole *M*. Ce symbole est un raccourci pour représenter le mot de 16 bits en mémoire se trouvant à l'adresse contenue dans le registre *A*. Ces trois destinations peuvent être combinées entre elles. La partie gauche de l'instruction de type *C* peut contenir les symboles suivants :

— `D` le résultat du calcul doit être stocké dans le registre `D`
— `A` le résultat du calcul doit être stocké dans le registre `A`
— `M` le résultat du calcul doit être stocké dans la mémoire à l'adresse qui se trouve actuellement dans le registre `A`
— `MD` le résultat du calcul doit être stocké dans le registre `D` et dans la mémoire à l'adresse qui se trouve actuellement dans le registre `A`
— `AM` le résultat du calcul doit être stocké dans le registre `A` et dans la mémoire à l'adresse qui se trouve actuellement dans le registre `A`
— `AD` le résultat du calcul doit être stocké dans le registre `A` et dans le registre `D`
— `AMD` le résultat du calcul doit être stocké dans le registre `A`, le registre `D` et dans la mémoire à l'adresse qui se trouve actuellement dans le registre `A`

Il est aussi possible d'avoir une instruction de type *C* qui ne modifie ni les registres `A`/`D` ni la mémoire. Nous en parlerons plus tard.

La partie droite de l'instruction de type *C* permet de spécifier le calcul à réaliser. Une première possibilité est de prendre la valeur d'un registre ou d'une zone mémoire sans demander à l'ALU de réaliser un calcul particulier. Les trois calculs les plus simples à réaliser correspondent aux symboles `A`, `D` et `M` :

— `D` le résultat du calcul est la valeur stockée dans le registre `D`
— `A` le résultat du calcul est la valeur stockée dans le registre `A`
— `M` le résultat du calcul est la donnée qui se trouve en mémoire à l'adresse qui se trouve actuellement dans le registre `A`

Nous pouvons maintenant explorer ces différentes instructions. Supposons que la mémoire contient les valeurs reprises dans Tableau 11.1.

TABLEAU 11.1 – Contenu de la mémoire

| adresse | valeur |
|---|---|
| 0 | 9 |
| 1 | 2 |
| 2 | 4 |
| 3 | 1 |

Commençons par le code qui permet de charger une donnée en mémoire.

```
@1     // place l'adresse 1 dans le registre A
D=M    // lit la donnée à l'adresse 1 en mémoire et la place dans D
```

Après exécution de ces deux instructions, le registre `D` contient la valeur qui se trouvait en mémoire à l'adresse *1*, c'est-à-dire *2*.

Vous pouvez télécharger cet exemple depuis `asm/ex1.asm`.

Notre deuxième exemple montre qu'il est aussi possible de charger le registre `A` avec une valeur stockée en mémoire.

```
@1     // place l'adresse 1 dans le registre A
A=M    // lit la donnée à l'adresse 1 en mémoire et place donc (2) dans A
D=M    // lit la donnée à l'adresse 2 en mémoire (4) et la place dans D
```

Vous pouvez télécharger cet exemple depuis `asm/ex2.asm`.

Notre troisième exemple montre comment déplacer une information en mémoire.

```
@3     // place l'adresse 3 dans le registre A
AD=M   // lit la donnée à l'adresse 3 en mémoire (1) et la place dans A et D
@0     // place l'adresse 0 dans le registre A
M=D    // sauve la donnée se trouvant dans D en mémoire à l'adresse se trouvant dans␣
↪A (0)
```

Vous pouvez télécharger cet exemple depuis `asm/ex3.asm`.

Nous pouvons maintenant utiliser ces instructions pour réaliser des initialisations de variables comme dans un langage de haut niveau comme python. En python cette initialisation s'écrit comme en Code source 11.1

Code source 11.1 – Initialisation de variables en python

```
a=1
b=42
```

Avant de pouvoir initialiser des variables en assembleur, nous devons d'abord définir l'adresse en mémoire à laquelle chaque variable est stockée. Par convention, le minuscule processeur réserve les adresses de *0* à *15* en mémoire de données pour certaines utilisations particulières. Nous pouvons donc stocker nos variables à partir de l'adresse *16*. Nous pouvons par exemple placer la variable `a` à l'adresse *16* et la variable *b* à l'adresse *17*. Dans un programme en assembleur, on définit généralement une table des symboles qui associent une adresse à chaque variable du programme. Dans notre exemple, cette table des symboles pourrait être celle du Tableau 11.2.

TABLEAU 11.2 – Table des symboles

| adresse | variable |
|---|---|
| 16 | a |
| 17 | b |
| 18 | — |
| ... | |

Pour initialiser ces variables, la séquence d'instruction à utiliser est la suivante. Premièrement, il faut charger la valeur *1* dans le registre `D`. Ensuite il faut charger dans le registre `A` l'adresse de la variable *a* (16 dans notre exemple) pour pouvoir sauver le contenu du registre `D` à cette adresse en mémoire. On fait de même pour l'initialisation de la variable `b`.

Code source 11.2 – Initialisation de variables en assembleur

```
@1    // valeur 1 pour l'initialisation
D=A
@16   // adresse de la variable a
M=D
@42   // valeur 42 pour l'initialisation
D=A
@17   // adresse de la variable b
M=D
```

Vous pouvez télécharger cet exemple depuis `asm/ex4.asm`.

Dans un programme python, il est parfois nécessaire d'échanger le contenu de la variable `a` avec celui de la variable `b`. En python, cela peut se faire de deux façons. La première solution est d'utiliser une variable intermédiaire (Code source 11.3)

Code source 11.3 – Echange de contenu de variables en python

```
x=a
a=b
b=x
```

Pour simplifier la vie du programmeur, python permet de cacher la création d'une variable temporaire et supporte la forme compacte reprise en Code source 11.4.

Code source 11.4 – Echange de contenu de variables en python (forme compacte)

```
a,b = b, a
```

Pour faire la même opération en langage assembleur, nous devons aussi passer par une zone mémoire intermédiaire. Dans notre exemple, l'adresse `18` est inutilisée. Nous pouvons donc y placer le contenu de la variable `b` avant d'y copier le contenu de la variable `a` comme dans le programme en python. La code assembleur est présenté en Code source 11.5.

Code source 11.5 – Echange du contenu de variables en assembleur

```
@17   // variable b
D=M
@18   // variable x
M=D
@16   // variable a
D=M
@17   // variable b
M=D
@18   // variable x
D=M
@16   // variable b
M=D
```

Vous pouvez télécharger cet exemple depuis `asm/ex5.asm`.

Continuons notre exploration des instructions de type *C*. L'ALU de notre minuscule processeur est aussi capable de produire les constantes suivantes :

- `0`
- `1`
- `-1`

Ces constantes peuvent apparaître dans la partie de droite d'une instruction de type *C*. A titre d'exemple le Code source 11.6 initialise le contenu de la variable `x` à la valeur *1* et celui de la variable `y` à *-1*.

Code source 11.6 – Initialisation de variables

```
@20   // variable x
M=1
@21   // variable y
M=-1
```

Vous pouvez télécharger cet exemple depuis `asm/ex6.asm`.

Notre minuscule ALU peut aussi réaliser des calculs sur un registre ou une valeur lue en mémoire. La partie de droite d'une instruction de type *C* peut en effet contenir les symboles suivants :

- `!D` : le résultat de l'ALU sera le résultat de l'application de l'opération *NOT* à tous les bits du contenu du registre `D`
- `!A` : le résultat de l'ALU sera le résultat de l'application de l'opération *NOT* à tous les bits du contenu du registre `A`
- `!M` : le résultat de l'ALU sera le résultat de l'application de l'opération *NOT* à tous les bits du mot lu en mémoire à l'adresse contenue dans le registre `A`
- `-D` : le résultat de l'ALU sera l'opposé du contenu du registre `D`
- `-A` : le résultat de l'ALU sera l'opposé du contenu du registre `A`
- `-M` : le résultat de l'ALU sera l'opposé du mot lu en mémoire à l'adresse contenue dans le registre `A`
- `D+1` : le résultat de l'ALU sera le contenu du registre `D` incrémenté de *1*

— `A+1` : le résultat de l'ALU sera le contenu du registre `A` incrémenté de *1*
— `M+1` : le résultat de l'ALU sera le mot lu en mémoire à l'adresse contenue dans le registre `A` incrémenté de *1*
— `D-1` : le résultat de l'ALU sera le contenu du registre `D` décrémenté de *1*
— `A-1` : le résultat de l'ALU sera le contenu du registre `A` décrémenté de *1*
— `M-1` : le résultat de l'ALU sera le mot lu en mémoire à l'adresse contenue dans le registre `A` décrémenté de *1*

Enfin, il est possible d'utiliser l'ALU pour effectuer des opérations arithmétiques (addition et soustraction) et logiques (*AND* et *OR*) avec les registres `A` et `D` ainsi que le mot lu en mémoire à l'adresse se trouvant dans le registre `A`. Le minuscule processeur supporte six opérations arithmétiques.

— `A+D` : le résultat de l'ALU sera le résultat de l'addition du contenu du registre `D` et du contenu du registre `A`
— `D+M` : le résultat de l'ALU sera le résultat de l'addition du contenu du registre `D` et du mot lu en mémoire à l'adresse contenue dans le registre `A`
— `A-D` : le résultat de l'ALU sera le résultat de la soustraction du contenu du registre `A` moins le contenu du registre `D`
— `D-A` : le résultat de l'ALU sera le résultat de la soustraction du contenu du registre `D` moins le contenu du registre `A`
— `D-M` : le résultat de l'ALU sera le résultat de la soustraction du contenu du registre `D` moins le mot lu en mémoire à l'adresse contenue dans le registre `A`
— `M-D` : le résultat de l'ALU sera le résultat de la soustraction du mot lu en mémoire à l'adresse contenue dans le registre `A` moins le contenu du registre `D`

Les dernières opérations supportées par l'ALU sont les opération logiques.

— `D&A` : le résultat de l'ALU sera le résultat de l'opération logique *AND* appliquée au contenu du registre `D` et au contenu du registre `A`
— `D|A` : le résultat de l'ALU sera le résultat de l'opération logique *OR* appliquée au contenu du registre `D` et au contenu du registre `A`
— `D&M` : le résultat de l'ALU sera le résultat de l'opération logique *AND* appliquée au contenu du registre `D` et au mot lu en mémoire à l'adresse contenue dans le registre `A`
— `D|M` : le résultat de l'ALU sera le résultat de l'opération logique *OR* appliquée au contenu du registre `D` et au mot lu en mémoire à l'adresse contenue dans le registre `A`

Avec ces 28 opérations, nous pouvons maintenant réaliser de très nombreuses opérations arithmétiques et logiques. Dans un programme informatique, il est très courant de devoir incrémenter ou décrémenter une variable. Dans notre minuscule langage d'assemblage, cette opération peut se réaliser de différentes façons. Une première solution pour incrémenter une variable est d'y ajouter la constante `1`. Supposons que notre variable soit stockée à l'adresse 20.

Code source 11.7 – Incrémentation de la valeur d'une variable en assembleur

```
@20   // adresse de la variable
D=M   // chargement de la valeur de la variable
@1    // constante 1
D=D+A // addition
@20   // adresse de la variable
M=D   // sauvegarde du résultat en mémoire
```

Vous pouvez télécharger cet exemple depuis `asm/ex7.asm`.

Il existe une solution nettement plus compacte et plus efficace (Code source 11.8).

Code source 11.8 – Incrémentation de la valeur d'une variable en assembleur

```
@20   // adresse de la variable
M=M+1
```

Vous pouvez télécharger cet exemple depuis `asm/ex7b.asm`.

Il en va de même pour décrémenter la valeur d'une variable (Code source 11.9).

Code source 11.9 – Décrémentation d'une variable en assembleur

```
@20   // adresse de la variable
M=M-1
```

Vous pouvez télécharger cet exemple depuis `asm/ex7c.asm`.

Le minuscule langage d'assemblage permet de réaliser des opérations mathématiques plus complexes. il est en effet possible de combiner des additions et des soustractions. Supposons que *A*, *B* et *C* sont des variables entières et qu'il faut calculer $A + B - C$ et stocker le résultat dans la variable :math`X`. Pour cela, il faut d'abord fixer les adresses mémoires dans lesquelles ces variables sont stockées (Tableau 11.3).

TABLEAU 11.3 – Table des symboles

| adresse | variable |
|---|---|
| 21 | A |
| 22 | B |
| 23 | C |
| 25 | X |

Code source 11.10 – X=A+B-C en assembleur minuscule

```
@21   // adresse de la variable A
D=M   // chargement de la valeur de la variable
@22   // adresse de la variable B
D=D+M // addition, D contient A+B
@23   // adresse de la variable C
D=D-M // soustraction, D contient A+B-C
@25   // adresse de la variable X
M=D   // sauvegarde du résultat en mémoire
```

Vous pouvez télécharger cet exemple depuis `asm/ex8.asm`.

On peut également utiliser les instructions de notre langage d'assemblage pour calculer l'opposé d'un nombre. Si la variable est stockée à l'adresse `20` et que son opposé doit être stocké à l'adresse `24`, une première solution est de procéder comme dans le Code source 11.11.

Code source 11.11 – Calcul de l'opposé

```
@20   // adresse de la variable
D=-M  // calcul de l'opposé
@24   // adresse de la variable B
M=D   // sauvegarde du résultat en mémoire
```

Vous pouvez télécharger cet exemple depuis `asm/ex9.asm`.

### 11.1.3 Exercices

1. Proposez deux façons pour initialiser la variable `X` qui est stockée à l'adresse `23` à la valeur `17`.
2. Avec le minuscule langage d'assemblage, comment faire pour initialiser une variable à la valeur `-2` ?
3. Que font les instructions en assembleur minuscule ci-dessous ?

```
@20
D=!M
D=D+1
@25
M=D
```

4. Avec le minuscule assembleur, l'initialisation d'une variable se fait normalement avec une instruction de type *A* :

```
@1234   // valeur
D=A
@16     // adresse variable
M=D
```

   Cependant, comme l'instruction de type *A* est encodée sur 16 bits, il n'y a que 15 bits de disponibles pour encoder cette valeur. Comment feriez-vous pour traduire l'assignation `x=50000` en minuscule assembleur ?

5. Le minuscule assembleur supporte les opérations logiques *AND* et *OR* de l'ALU. Certains langages de programmation supportent également l'opération *XOR*. Comment feriez-vous pour implémenter l'opération *XOR* en minuscule assembleur ?

Toutes les instructions de type *C* sont encodées sous la forme d'un mot de 16 bits qui a la structure suivante :

$$111ac_1c_2c_3c_4c_5c_6d_1d_2d_3j_1j_2j_3$$

Dans cette structure, le bit de poids fort mis à *1* permet au minuscule processeur de distinguer une instructions de type *A* (dont le bit de poids fort est mis à *0*) d'une instruction de type *C*. Les deux bits suivants ne sont pas utilisés par le minuscule processeur. Ensuite, les bits $a$ et $c_i$ servent à spécifier les différentes instructions que nous avons présenté ci-dessus. Le livre de référence contient la spécification complète de ces instructions. En voici quelques unes à titre d'exemples. Pour les instructions arithmétiques et logiques, les bits de poids faible ($j_1j_2j_3$ sont mis à *0*).

— l'instruction `M=D+1` a comme encodage `1 1 1 0 0 1 1 1 1 1 0 0 1 0 0 0`. Dans cet encodage, `0 0 1 1 1 1 1` représente le membre de droite (`D+1`) et `0 0 1` le membre de gauche de l'instruction
— l'instruction `D=D+1` a comme encodage `1 1 1 0 0 1 1 1 1 1 0 1 0 0 0 0`. Dans cet encodage, `0 0 1 1 1 1 1` représente le membre de droite (`D+1`) et `0 1 0` le membre de gauche de l'instruction
— l'instruction `AMD=A-D` a comme encodage `1 1 1 0 0 0 0 1 1 1 1 1 1 0 0 0`. Dans cet encodage, `0 0 0 0 1 1 1` représente le membre de droite (`A-D`) et `1 1 1` le membre de gauche de l'instruction

Le rôle des autres bits qui composent cette instruction sera détaillé plus tard.

## 11.2 Les instructions de saut

Pour exécuter un programme, notre minuscule processeur doit charger une nouvelle instruction à chaque cycle d'horloge. Il le fait en utilisant le registre `PC`. Celui-ci est initialisé à la valeur `0` lorsque le minuscule processeur démarre. A chaque cycle d'horloge, le minuscule processeur réalise les opérations suivantes :

— lecture de l'instruction se trouvant à l'adresse qui est stockée dans le registre `PC`
— décodage de l'instruction lue en mémoire
— exécution de l'instruction lue en mémoire
— mise à jour du registre `PC`

L'exécution de toutes les instructions que nous avons vues jusque maintenant se termine par l'incrémentation du contenu du registre `PC`. Cela permettra à notre minuscule processeur de charger automatiquement l'instruction suivante lors du prochain cycle d'horloge.

L'encodage de l'instruction de type *C* implique que les trois bits de poids faible ($j_1 j_2 j_3$) restent disponibles. Ceux-ci vont nous permettre de supporter les instructions conditionnelles (`if ... else`) et les boucles. Pour comprendre comment ces instructions sont supportées en langage d'assemblage, nous devons d'abord comprendre comment fonctionne le compteur de programme (ou Program Counter - PC en anglais). Ce compteur de programme est un registre qui fait partie de notre minuscule processeur et qui contient à tout instant l'adresse de l'instruction que le minuscule processeur exécute. Reprenons le code du calcul de l'opposé (Code source 11.11). Ce code contient quatre instructions. Il est stocké dans la mémoire d'instructions (Tableau 11.4).

TABLEAU 11.4 – Instructions du calcul de l'opposé en mémoire d'instructions

| adresse | instruction |
|---|---|
| 51 | @20 |
| 52 | D=-M |
| 53 | @24 |
| 54 | M=D |

Pour exécuter ces instructions en mémoire d'instructions, le *PC* prend d'abord la valeur *51*. Le minuscule processeur exécute à ce moment l'instruction `@20`. A la fin de l'exécution de cette instruction, le *PC* est incrémenté d'une unité et passe à *52*. Il exécute ensuite l'instruction `D=-M`. A la fin de l'exécution de cette instruction, le *PC* passe à la valeur *53* et ainsi de suite.

Les trois bits de poids faible de l'instruction de type *C* permettent d'influencer la façon dont le contenu du *PC* est modifié à la fin de l'exécution de l'instruction en cours. Lorsque ces trois bits valent *0 0 0*, le *PC* est incrémenté d'une unité. Si par contre ces trois bits valent *1 1 1*, le *PC* prend la valeur qui se trouve dans le registre *A* pour réaliser un saut (*jump* en anglais). Pour comprendre l'utilisation de ces sauts, revenons aux instructions qui nous permettent d'incrémenter une variable en mémoire (Code source 11.8). Supposons que notre variable est stockée à l'adresse `22` en mémoire de données et que notre séquence d'instructions commence à l'adresse `71` en mémoire d'instructions.

TABLEAU 11.5 – Un programme qui ne s'arrête jamais

| adresse | instruction |
|---|---|
| 71 | @22 |
| 72 | M=M+1 |
| 73 | @71 |
| 74 | 0;JMP |

Vous pouvez télécharger cet exemple depuis `asm/ex10.asm`.

Exécutons le programme représenté en Tableau 11.5 instruction par instruction en supposant que la mémoire de données contient initialement la valeur *0* à l'adresse `22`. Les instructions suivantes sont exécutées :

— exécution de l'instruction à l'adresse `71`, chargement de la valeur *22* dans le registre *A*, *PC* passe à `72`

— exécution de l'instruction à l'adresse `72`, incrémentation de la valeur stockée en mémoire à l'adresse se trouvant dans le registre *A*. L'adresse *22* en mémoire de données contient maintenant *1*. *PC* passe à `73`
— exécution de l'instruction à l'adresse `73`, chargement de la valeur *71* dans le registre *A*, *PC* passe à `74`
— exécution de l'instruction à l'adresse `74`, le *PC* prend la valeur stockée dans le registre *A* (`71`)
— exécution de l'instruction à l'adresse `71`, chargement de la valeur *22* dans le registre *A* (1), *PC* passe à `72`
— exécution de l'instruction à l'adresse `72`, incrémentation de la valeur stockée en mémoire à l'adresse se trouvant dans le registre *A*. L'adresse *22* en mémoire de données contient maintenant *2*. *PC* passe à `73`
— exécution de l'instruction à l'adresse `73`, chargement de la valeur *71* dans le registre *A*, *PC* passe à `74`
— exécution de l'instruction à l'adresse `74`, le *PC* prend la valeur stockée dans le registre *A* (`71`)
— ...

Ce programme ne s'arrêtera jamais. Il est équivalent au code python suivant.

```
while True:
  x=x+1
```

## 11.3 Les instructions de saut conditionnel

L'instruction de saut (`0;JMP`) est très fréquente en assembleur. Elle permet d'effectuer un saut qui est dit non-conditionnel car la valeur du *PC* est toujours modifiée. A côté de cette instruction, notre minuscule langage d'assemblage supporte plusieurs instructions de saut conditionnel. Ces instructions modifient la valeur du *PC* uniquement si une condition particulière est vérifiée. Le langage d'assemblage du minuscule processeur supporte six instructions de saut conditionnel :

— `JEQ` (Jump if EQual to 0). Avec cette instruction, le saut est réalisé uniquement si le résultat du calcul fait par l'ALU est nul.
— `JNE` (Jump if Not Equal to 0). Avec cette instruction, le saut est réalisé uniquement si le résultat du calcul fait par l'ALU est différent de zéro.
— `JGT` (Jump if Greater Than 0). Avec cette instruction, le saut est réalisé uniquement si le résultat du calcul fait par l'ALU est strictement positif.
— `JLT` (Jump if Lower Than 0). Avec cette instruction, le saut est réalisé uniquement si le résultat du calcul fait par l'ALU est strictement inférieur à 0.
— `JGE` (Jump if Greater than or Equal to 0). Avec cette instruction, le saut est réalisé uniquement si le résultat du calcul fait par l'ALU est supérieur ou égal à 0.
— `JLE` (Jump if Lower than or Equal to 0). Avec cette instruction, le saut est réalisé uniquement si le résultat du calcul fait par l'ALU est inférieur ou égal à 0.

Avec ces six instructions, il est possible de supporter les instructions conditionnelles et les boucles avec le minuscule langage d'assemblage. Commençons par les instructions conditionnelles. Supposons que l'on veuille mettre dans la variable `y` la valeur absolue de la variable `x`. En python, une première approche pourrait être celle du programme ci-dessous.

```
y=x
if (x<0):
  y=-x
# y contient abs(x)
z=0
```

Une première solution pour traduire ces trois lignes de python est de les traduire le plus littéralement possible.

TABLEAU 11.6 – Calcul de la valeur absolue en minuscule assembleur

| adresse | instruction |
|---|---|
| 41 | @22 // x |
| 42 | D=M |
| 43 | @23 // y |
| 44 | MD=D |
| 45 | @49 |
| 46 | D ;JLT |
| 47 | @51 |
| 48 | 0 ;JMP |
| 49 | @23 |
| 50 | M=-D |
| 51 | @24 // z |
| 52 | M=0 |

Vous pouvez télécharger cet exemple depuis `asm/ex11a.asm`.

Il est intéressant d'analyser l'exécution du programme du Tableau 11.6 pas à pas. Les instructions aux adresses `41` et `42` placent la valeur de la variable `x` dans le registre *D*. Les deux instructions suivantes sauvent le contenu de ce registre dans la variable `y`. L'instruction à l'adresse `45` charge l'adresse `49` dans le registre *A*. Cette adresse est celle de la première instruction correspondant au corps du `if`. L'instruction suivante va elle comparer le contenu du registre *D* avec *0*. Si le registre *D* est strictement négatif, alors l'adresse se trouvant dans le registre *A*, c'est-à-dire `49` est placée dans le compteur de programme. Dans ce cas, le programme exécutera le corps de l'instruction conditionnelle. Si par contre le contenu du registre *D* est positif ou nul, nous ne devons pas exécuter le corps de la boucle, mais directement passer à l'instruction qui initialise la variable `z` à partir de l'adresse `51`. C'est le rôle de l'instruction de saut inconditionnel aux adresses `47` et `48`. L'instruction à l'adresse `49` est celle du corps de l'instruction conditionnelle. A la fin de son exécution on peut exécuter l'instruction qui suit l'instruction conditionnelle.

En y réfléchissant un peu, on peut réduire le nombre d'instructions conditionnelles dans ce programme en utilisant une instruction `JGE` (Tableau 11.7).

TABLEAU 11.7 – Calcul de la valeur absolue en minuscule assembleur

| adresse | instruction |
|---|---|
| 41 | @22 // x |
| 42 | D=M |
| 43 | @23 // y |
| 44 | M=D |
| 45 | @49 |
| 46 | D ;JGE |
| 47 | @23 |
| 48 | M=-D |
| 49 | ... |

Vous pouvez télécharger cet exemple depuis `asm/ex11.asm`.

Il est intéressant d'analyser l'exécution du programme du Tableau 11.7 pas à pas. Les instructions aux adresses `41` et `42` placent la valeur de la variable `x` dans le registre *D*. Les deux instructions suivantes sauvent le contenu de ce registre dans la variable `y`. L'instruction à l'adresse `45` charge l'adresse `49` dans le registre *A*. L'instruction suivante va elle comparer le contenu du registre *D* avec *0*. Si le registre *D* est positif ou nul, alors l'adresse se trouvant dans le registre *A*, c'est-à-dire `49` est placée dans le compteur de programme. Sinon, les instructions aux adresses `47` et `48` sont exécutées. Par rapport au code python, on remarque que l'on prend comme condition pour l'instruction assembleur l'inverse de la condition du code python. En effet, la condition de l'instruction conditionnelle en python doit être vérifiée pour que l'instruction `y=-x` soit exécutée. En assembleur, on place la cible du saut après l'exécution

des instructions qui se trouvent dans le corps du `if` en python. Analysons une seconde variante du calcul de la valeur absolue.

```
if (x>0):
  y=x
else:
  y=-x
# y contient abs(x)
```

Une première approche pour traduire ce code python en minuscule assembleur serait de procéder comme dans la Tableau 11.8.

TABLEAU 11.8 – Essai de calcul de la valeur absolue en minuscule assembleur

| adresse | instruction |
|---|---|
| 41 | @22 // x |
| 42 | D=M |
| 43 | @47 |
| 44 | D ;JLE |
| 45 | @23 // y |
| 46 | M=D |
| 47 | @23 // y |
| 48 | M=-D |
| 49 | ... |

Vous pouvez télécharger cet exemple depuis `asm/ex12.asm`.

Malheureusement, cette solution est incorrecte car elle place toujours la valeur de la variable `-x` dans la variable `y` quel que soit son signe. Lorsque `x` est négatif, l'exécution passe directement à l'instruction se trouvant à l'adresse `47` et sauve la valeur de `-x` dans la variable `y`. Cependant, si `x` est positif, après avoir copié `x` dans la variable `y` (instructions aux adresses `45` et `46`), le minuscule processeur exécute les instructions aux adresses `47` et `48` et sauve donc la valeur de `-x` dans la variable `y`. On peut éviter ce problème en utilisant un saut inconditionnel après le corps du `if ...` (Tableau 11.9).

TABLEAU 11.9 – Calcul de la valeur absolue en minuscule assembleur

| adresse | instruction |
|---|---|
| 41 | @22 // x |
| 42 | D=M |
| 43 | @47 |
| 44 | D ;JLE |
| 45 | @23 // y |
| 46 | M=D |
| 47 | @51 |
| 48 | 0 ;JMP |
| 49 | @23 // y |
| 50 | M=-D |
| 51 | ... |

Vous pouvez télécharger cet exemple depuis `asm/ex12b.asm`.

Dans l'exemple de la Tableau 11.9, le saut inconditionnel des instructions aux adresses `47` et `48` garantit que les instructions des adresses `49` et `50` ne seront pas exécutées lorsque `x` est positif.

Un approche similaire peut être utilisée pour implémenter d'autres instructions conditionnelles. Le tout est de ramener

toute condition à une comparaison avec la valeur *0* ou à une comparaison de signe. Ainsi, pour comparer si deux variables contiennent la même valeur, il suffira de calculer une soustraction et ensuite de vérifier si le résultat est nul. Il en va de même pour vérifier si deux variables contiennent des valeurs différentes.

Pour les conditions plus complexes, il faut parfois réécrire l'instruction conditionnelle. Prenons deux exemples en python pour illustrer cette réécriture.

```
if (a>0) AND (b<1):
   x=2
```

Dans ce cas, on peut réécrire l'instruction conditionnelle sous la forme :

```
if (a>0) :
  if (b<1) :
    x=2
```

Ces deux instructions conditionnelles imbriquées peuvent facilement s'implémenter avec les instructions de saut conditionnel que nous avons présenté. Il en va de même pour une disjonction logique. L'instruction ci-dessous :

```
if (a>0) OR (b<1):
   x=3
```

peut se réécrire de la façon suivante pour supprimer la disjonction logique.

```
if (a>0) :
  x=3
else :
  if (b<1) :
      x=2
```

A nouveau, les deux instructions conditionnelles ci-dessous peuvent facilement s'implémenter avec les instructions conditionnelles de notre minuscule langage d'assemblage.

Lorsque l'on utilise le langage d'assemblage, il peut être fastidieux de devoir indiquer les valeurs numériques des adresses des variables ainsi que des adresses des sauts. Heureusement, l'assembleur du minuscule processeur vous permet d'utiliser des symboles qui correspondent à ces adresses. Avec ces symboles, notre exemple du calcul de la valeur absolue (Tableau 11.9) peut s'écrire comme suit :

```
// valeur absolue
   @x    // variable, adresse choisie par l'assembleur
   D=M
   @SUITE // adresse calculée par l'assembleur
   D;JLE
   @y    // variable, adresse choisie par l'assembleur
   M=D
   @SUITE
   0;JMP
   @y
   M=-D
(SUITE)
//  ...
```

Vous pouvez télécharger cet exemple depuis `asm/ex13.asm`.

Dans ce code, l'assembleur construit automatiquement la table des symboles permettant de sauver les variables `x` et `y`. Il détermine aussi l'adresse en mémoire de l'instruction qui correspond à l'étiquette `(SUITE)` et remplace cette étiquette par l'adresse correspondante dans le code. Cela simplifie l'écriture de programmes en minuscule assembleur.

### 11.3.1 Exercices

1. Convertissez en minuscule assembleur l'instruction conditionnelle suivante :

```
if(x!=y):
  a=x+y
else:
  a=x-y
```

2. Convertissez en minuscule assembleur le code python ci-dessous :

```
if(x>a) and (x<b) :
  i=1
else:
  i=0
```

3. Convertissez en minuscule assembleur le code python ci-dessous :

```
if(x==a) or (x>b) :
  y=a
else:
  y=-1
```

## 11.4 Les boucles

Après les opérations arithmétiques et logiques et les instructions conditionnelles, il nous reste à voir comment supporter les boucles. Python supporte deux types principaux de boucles :

— les boucles `while`
— les boucles `for`

Les boucles `while` sont les boucles les plus générales. Un boucle `for` est généralement une boucle d'un type particulier qui est écrite de façon compacte. Nous nous focaliserons sur les boucles `while` dans cette section. Une boucle `while` comprend toujours une condition qui est une expression booléenne et un corps comprenant une ou plusieurs instructions à exécuter. Nous avons déjà vu que la boucle infinie

```
while True:
  x=x+1
```

pouvait être traduite dans notre minuscule assembleur par les instructions reprises en Tableau 11.10.

TABLEAU 11.10 – Une boucle infinie

| adresse | instruction |
|---|---|
| 71 | @22 |
| 72 | M=M+1 |
| 73 | @71 |
| 74 | 0;JMP |

Vous pouvez télécharger cet exemple depuis `asm/ex14.asm`.

Nous pouvons nous inspirer de cette approche pour traduire une boucle `while` en une séquence d'instructions en minuscule assembleur. Pour cela, notre programme doit :

1. Évaluer la valeur de la condition
2. Si la condition s'évalue à True, exécuter le corps de la boucle puis revenir au point 1
3. Sinon, passer à l'exécution des instructions placées juste après le corps de la boucle

Pour illustrer cette traduction, considérons la boucle ci-dessous. Après l'exécution de cette boucle, la variable `x` contient la valeur *512*.

```
x=1
n=1
while (n<10) :
  x=x+x
  n=n+1
```

Le code assembleur correspondant est présenté ci-dessous. L'étiquette `(DEBUT)` correspond à la première instruction. Nous initialisons ensuite les variables `x` et `n` à la valeur *1* dans les deux mots de mémoire que l'assembleur leur a réservé. L'étiquette `(DBOUCLE)` correspond à l'adresse de la première instruction de notre boucle. Les quatre instructions qui suivent placent dans le registre `D` le résultat de $n - 10$. Cela nous permet ensuite de comparer cette valeur avec *0*. Si $n - 10 \geq 0$, alors la condition de notre boucle n'est pas vérifiée et nous devons en sortir. C'est le rôle de l'instruction `JGE` qui placera l'adresse de l'étiquette `(FBOUCLE)` dans le compteur de programme. Sinon, les six instructions suivantes permettent de placer `x+x` dans la variable `x` et ensuite d'incrémenter la variable `n`. Les deux dernières instructions permettent de revenir à l'adresse de l'étiquette `(DBOUCLE)` pour faire l'itération suivante dans la boucle.

```
(DEBUT)
        @x
        M=1
        @n
        M=1
(DBOUCLE)
        @10
        D=A
        @n
        D=M-D
        @FBOUCLE
        D;JGE
        @x
        D=M
        @x
        M=D+M
        @n
        M=M+1
        @DBOUCLE
        0;JMP
(FBOUCLE)
```

Le programme en minuscule assembleur est téléchargeable via `asm/boucle.asm`.

### 11.4.1 Exercices

1. Écrivez un programme en assembleur pour calculer la somme des *n* premiers naturels.
2. Le reste de la division euclidienne entre deux naturels `a % b` peut s'obtenir en faisant une série de soustractions. En python, cela peut s'écrire comme suit

   ```
   # place dans r le reste de la division euclidienne a/b
   r=a
   while(r>=b):
      r=r-b
   ```

   Convertissez ce programme python en une suite d'instructions en minuscule assembleur.

Python, comme d'autres langages de programmation, support les mode clés `break` et `continue` qui peuvent être utilisé à l'intérieur de boucles. Prenons comme exemple la boucle ci-dessous.

```
x=9
while(x<a):
   x=x+b
   if(x>c):
     x=c
     break
```

Ce fragment de code en python peut être traduit en minuscule assembleur par les instructions ci-dessous (téléchargeable via `asm/boucle-break.asm`). La traduction en assembleur de ce fragment de code montre que l'instruction `break` est traduite comme un saut inconditionnel qui permet de sortir de la boucle.

```
      @9
      D=A
      @x
      M=D
(DBOUCLE)
      @a
      D=M
      @x
      D=M-D
      @FBOUCLE
      D;JGE       // while(x<a)
      @b
      D=M
      @x
      M=D+M
      @x            // x=x+b
      D=M
      @c
      D=D-M
      @FINIF
      D;JLE       // if(x>c)
      @c
      D=M
      @x
      M=D         // x=c
      @FBOUCLE
      0;JMP       // break
(FINIF)
      @DBOUCLE
      0;JMP
(FBOUCLE)
```

Python supporte aussi l'instruction `continue` qui permet de continuer l'exécution de la boucle sans exécuter les instructions se trouvant après cette instruction. La code ci-dessous est un exemple de l'utilisation de `continue` en python.

```
x=7
while(x<a):
   if(x<c):
     x=x+1
     continue
   x=x+b
```

A nouveau, la traduction de ce code en minuscule assembleur fait appel à un saut inconditionnel pour supporter l'instruction `continue`, mais cette fois-ci vers l'étiquette `(DBOUCLE)` qui correspond au début de la boucle.

```
      @7
      D=A
      @x
      M=D
(DBOUCLE)
      @a
      D=M
      @x
      D=M-D
      @FBOUCLE
      D;JGE     // while(x<a)
      @x
      D=M
      @c
      D=D-M
      @SUITE
      0;JGE     // if (x<c)
      @x
      M=M+1     // x=x+1
      @DBOUCLE
      D;JMP
(SUITE)
      @b
      D=M
      @x
      M=D+M     // x=x+b
      @DBOUCLE
      0;JMP
(FBOUCLE)
```

Ce programme en minuscule assembleur est téléchargeable via `asm/boucle-continue.asm`.

CHAPITRE 12

# Tests de programmes en langage d'assemblage

Le langage d'assemblage est un langage de très bas niveau car il manipule directement la mémoire et les registres du minuscule processeur. Même si il ne supporte que deux types d'instructions, il est suffisamment expressif pour permettre des logiciels complexes. Tout comme pour les langages de programmation de plus haut niveau comme python, il est très important de bien tester et de vérifier le bon fonctionnement des programmes écrits en langage d'assemblage.

Pour ces tests, le livre de référence propose un simulateur du minuscule processeur qui peut s'utiliser de deux façons :

— exécution pas à pas d'un programme via l'interface graphique
— exécution « en batch » d'un programme et d'une suite de tests qui y est associée

L'exécution pas à pas est très pratique pour bien comprendre le fonctionnement d'un programme en langage d'assemblage ou détecter des erreurs durant son développement. L'interface graphique (Fig. 12.1) du simulateur est assez intuitive.

Le menu `File` permet de charger un programme en langage d'assemblage. Celui-ci peut avoir été écrit avec un éditeur de texte ou être du code machine qui a été produit par l'assembleur fourni avec le livre. Lorsque le simulateur du minuscule CPU charge un programme en langage d'assemblage, il vérifie d'abord sa syntaxe et affiche un message d'erreur en rouge en cas de problème. Ces messages d'erreur ne sont pas toujours explicites. Quand un tel message apparaît, il peut être utile de charger le programme dans l'assembleur de façon à vérifier sa syntaxe et le corriger si nécessaire.

Le programme chargé apparaît dans le tableau de gauche qui représente la ROM du minuscule ordinateur. Il est possible de modifier les instructions se trouvant dans ce tableau, mais pas de sauver la ROM modifiée dans un fichier. La mémoire RAM contenant les données est représentée par le second tableau. La partie droite de la fenêtre du simulateur représente l'écran graphique et le clavier.

Les trois registres du minuscule ordinateur sont représentés par des boites. La première est le `PC` qui se trouve en bas à gauche. Le registre `A` qui contient l'adresse en RAM à laquelle il faut lire les données se trouve en dessous de la RAM. Enfin, la partie droite de la fenêtre représente l'ALU avec le contenu du registre `D` juste au-dessus.

Il est possible d'exécuter un programme en minuscule langage d'assemblage de trois façons différentes. La première est l'exécution pas à pas. En cliquant sur la flèche bleue simple, on simule un cycle d'horloge et donc l'exécution d'une instruction. Cela permet d'observer l'exécution de petits programmes et l'effet de chaque instruction sur les différents registres et la mémoire. Cette exécution pas à pas reste fastidieuse pour de grands programmes.

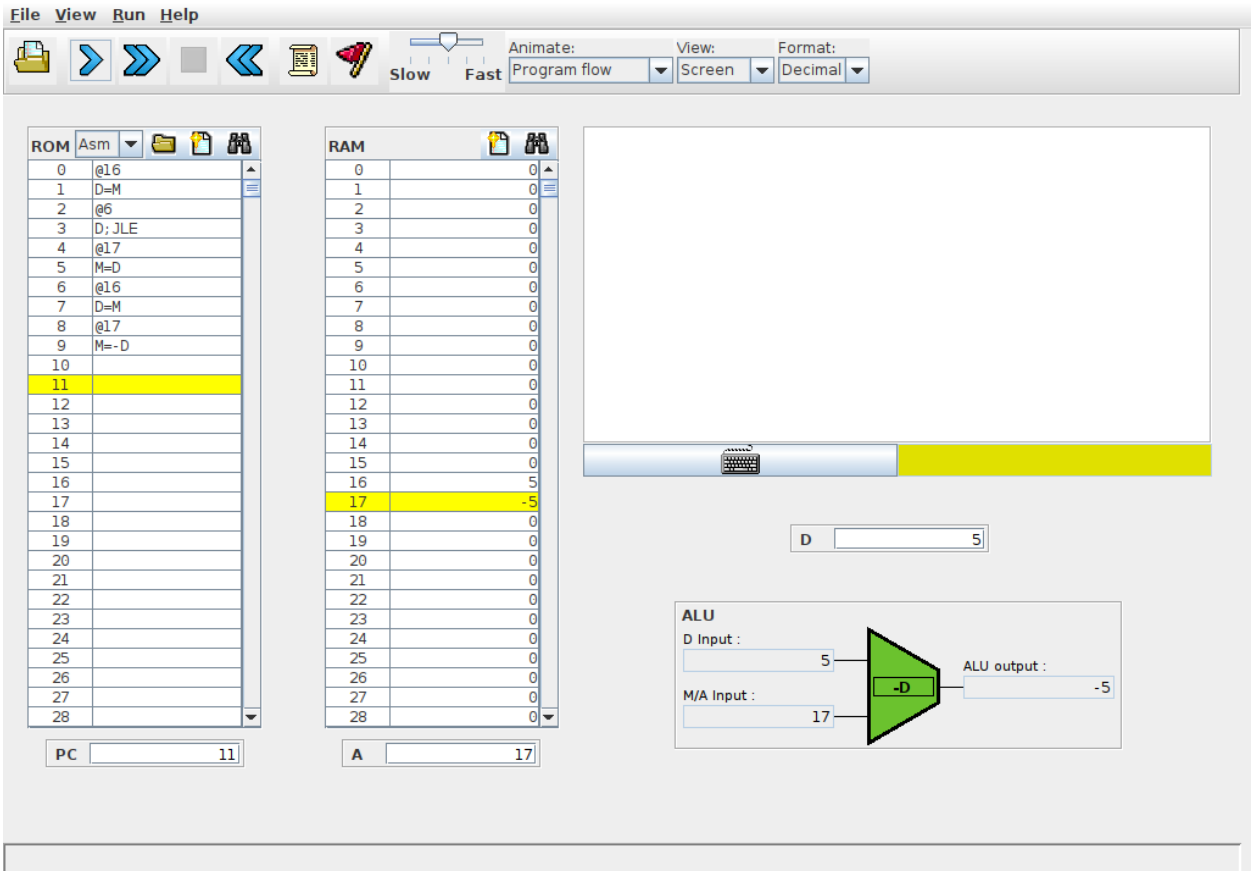



FIG. 12.1 – Simulateur interactif du minuscule processeur

La seconde méthode pour exécuter un programme est de définir des conditions d'arrêt ( breakpoints en anglais). Ces conditions permettent de spécifier quand l'exécution du programme doit s'arrêter. Ces conditions peuvent être :

— une valeur du `PC`
— un nombre de cycles d'horloge
— une valeur particulière dans le registre `A` ou le registre `D`
— une valeur stockée à une adresse en mémoire (cela permet de prendre en compte les valeurs des variables stockées en mémoire)

Grâce à ces conditions, il est possible de lancer l'exécution d'un programme et de passer au mode pas à pas dans la région du code qui est la plus intéressante. Plusieurs conditions d'arrêt peuvent être définies. Le simulateur les évalue lorsqu'il exécute chaque instruction et s'arrête dès qu'une condition est vérifiée.

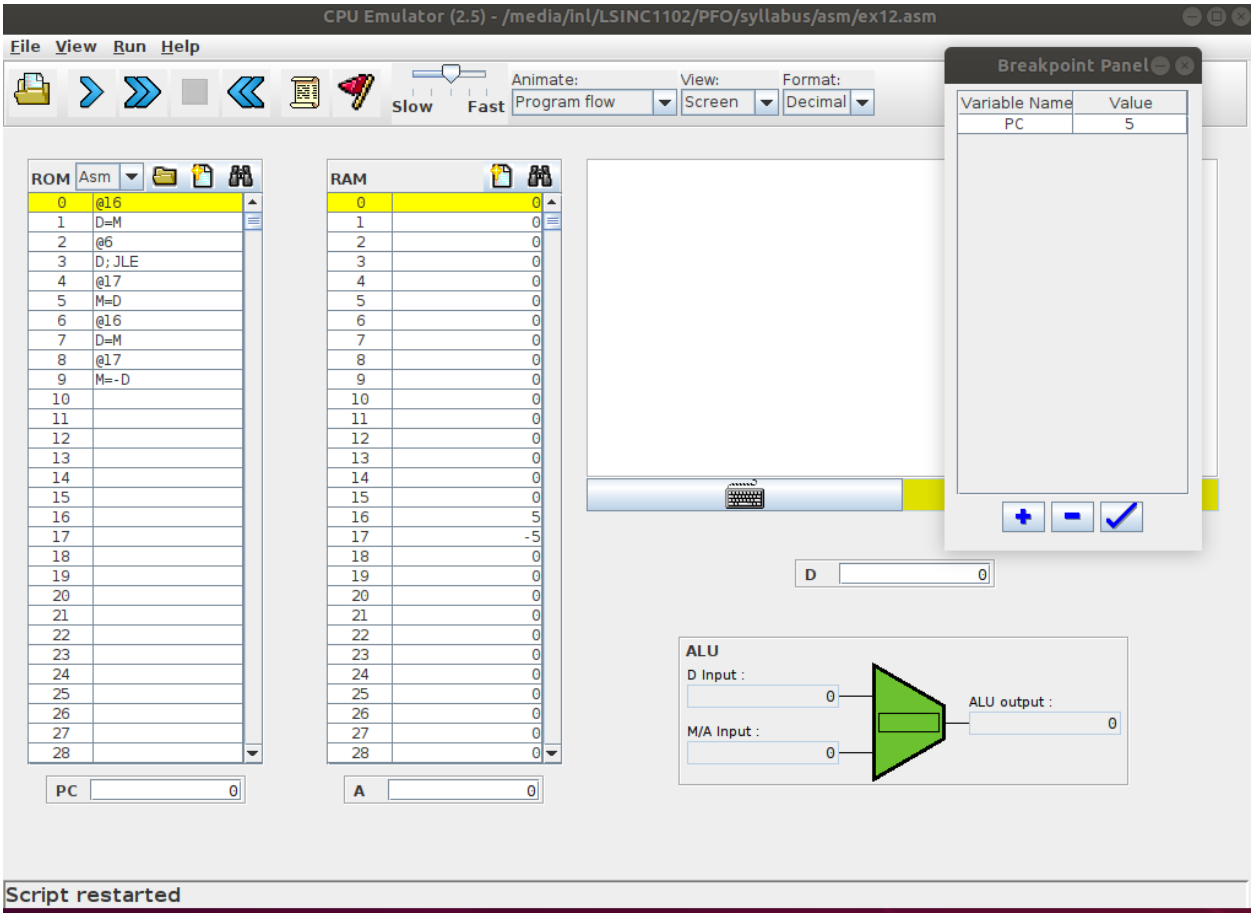



FIG. 12.2 – Breakpoints avec le simulateur interactif du minuscule processeur

La troisième méthode et d'écrire un script qui contrôle l'exécution du simulateur. Ces scripts sont un extension de ceux que vous avez déjà utilisé pour les circuits logiques. Ils permettent d'initialiser certaines zones de la mémoire et d'analyser le résultat de l'exécution d'un programme. Il est très utile de construire une script de test avant d'écrire un programme en assembleur.

Pour illustrer l'utilisation de ces scripts, reprenons le programme qui permet de calculer la valeur absolue d'un nombre entier. Nous avons vu précédemment que le code repris ci-dessous ne fonctionnait pas correctement

```
@x
D=M
@LABEL
D;JLE
```

(suite sur la page suivante)

(suite de la page précédente)

```
   @y
   M=D
(LABEL)
   @x
   D=M
   @y
   M=-D
```

Ce programme a comme entrée un entier qui est stocké à l'adresse `16` (`@x`)en mémoire RAM. La valeur absolue calculée se trouve à l'adresse `17` (`@y`). La première étape pour tester un tel programme est de définir les résultats attendus pour chaque exécution du programme. A la fin de chaque exécution, nous devons vérifier les valeurs se trouvant en mémoire aux adresses `16` et `17`.

```
| RAM[16]   | RAM[17]   |
|       0   |       0   |
|       1   |       1   |
|       7   |       7   |
|      -3   |       3   |
|      -1   |       1   |
```

Ce fichier est téléchargeable via `asm/abs.cmp`.

Les cinq lignes du fichier ci-dessus correspondent aux différents cas d'utilisation de notre programme de calcul de la valeur absolue. Un programme qui passe ces cinq tests devrait calculer la valeur absolue correctement.

Nous pouvons maintenant écrire notre script de test. Celui-ci contient d'abord une initialisation qui déclare le nom du fichier en langage machine à exécuter (`abs.hack` dans notre exemple), le fichier de sortie (`abs.out` dans notre exemple) et le format des données de sortie.

Ensuite, nous pouvons définir chaque test en initialisant le registre `PC` à `0` et en plaçant les valeurs souhaitées en RAM. La commande `ticktock` permet de faire passer un cycle d'horloge et donc d'exécuter une instruction. La commande `repeat x { ... }` répète `x` fois les instructions se trouvant dans le bloc `{ ... }`. Enfin, la commande `ouput` sauve dans le fichier de sortie les données définies dans la commande `output-list`. Un exemple complet est repris ci-dessous.

```
// Script de test du programme de calcul de la valeur absolue
load abs.hack,          // le fichier contenant le programme en langage machine
output-file abs.out,    // le fichier de sortie
//compare-to abs.cmp,     // les résultats attendus
output-list RAM[16]%D2.6.2 RAM[17]%D2.6.2;  // les deux valeurs

// premier test, abs(0)=0
set PC 0,       // démarrage du minuscule processeur
set RAM[16] 0, // valeur d'entrée
set RAM[17] 0; // initialisation de la mémoire
repeat 20 {     // vingt cycles d'horloge devraient suffire
  ticktock;
}
output; // Sauvegarde dans le fichier sortie

// deuxième test, abs(1)=1
set PC 0,
set RAM[16] 1,
set RAM[17] 1;
repeat 20 {
  ticktock;
}
```

(suite sur la page suivante)

(suite de la page précédente)

```
output;

// troisième test, abs(7)=7
set PC 0,
set RAM[16] 7,
set RAM[17] 7;
repeat 20 {
  ticktock;
}
output;

// quatrième test, abs(-3)=3
set PC 0,
set RAM[16] -3,
set RAM[17] 3;
repeat 20 {
  ticktock;
}
output;

// cinquième test, abs(-1)=1

set PC 0,
set RAM[16] -1,
set RAM[17] 1;
repeat 20 {
  ticktock;
}
output;
```

Lors de son exécution, le script retourne le résultat de l'exécution de notre programme. Une comparaison avec les valeurs attendues nous indique clairement que notre implémentation est erronée.

```
| RAM[16]   | RAM[17]   |
|        0  |        0  |
|        1  |       -1  |
|        7  |       -7  |
|       -3  |        3  |
|       -1  |        1  |
```

Dans le cadre des projets, nous vous encourageons à écrire d'abord le script de test avant d'écrire vos programmes et pas l'inverse. N'hésitez pas à écrire des petits scripts de test pour de petites parties de votre programme afin des les valider une après l'autre.

CHAPITRE 13

# Langage d'assemblage : compléments

Dans ce chapitre, nous allons d'abord voir comment notre minuscule ordinateur peut interagir avec le monde extérieur (écran et clavier) et ensuite comment manipuler des tableaux et des chaînes de caractères stockés en mémoire.

## 13.1 Entrées-sorties

Un ordinateur doit interagir avec son environnement. Les ordinateurs actuels comprennent de très nombreux dispositifs pour interagir avec les humains et le monde extérieur via des capteurs, clavier, souris, écran, … Le minuscule ordinateur se limite à deux dispositifs : un écran qui est son unique dispositif de sortie et un clavier qui est son unique dispositif d'entrée. Les principes que l'on va présenter pour ces deux dispositifs sont génériques et peuvent s'appliquer à d'autres dispositifs d'entrée ou de sortie. En anglais, on parle généralement de dispositifs d'I/O pour Input/Output.

Commençons par le clavier qui est le dispositif le plus simple. Un clavier peut s'interfacer de différentes façons avec un ordinateur. On peut voir un clavier comme une sorte de matrice dans laquelle chaque touche correspond à une position dans la matrice. Lorsqu'un utilisateur pousse sur une touche, l'élément correspondant de la matrice est mis à une valeur convenue. Si l'utilisateur pousse sur plusieurs touches, les positions correspondantes de la matrice sont modifiées. Cela permet de supporter des claviers avec des touches telles que *shift* ou *ctrl* dont la pression modifie le caractères correspondant à une autre touche.

Le minuscule ordinateur prend une approche beaucoup plus simple. Il ne représente pas les touches tapées par l'utilisateur mais retourne directement le mot de 16 bits qui correspond au caractère tapé par l'utilisateur. Il reste cependant à déterminer comment un programme peut accéder à ce caractère. Pour cela, le minuscule ordinateur utilise la technique des entrées/sorties mappées en mémoire (memory-mapped I/O en anglais). Cette technique est à la fois très simple, mais aussi très fréquemment utilisée pour supporter de très nombreux dispositifs d'entrée-sortie.

Le clavier du minuscule ordinateur comprend un registre qui contient le code ASCII du caractère sur lequel l'utilisateur tape actuellement sur le clavier. Si l'utilisateur ne tape pas sur le clavier, celui-ci contient la valeur `0`. En outre, le minuscule ordinateur définit les caractères de contrôle repris dans le Tableau 13.1.

TABLEAU 13.1 – Caractères de contrôle

| Touche | Code ASCII |
|---|---|
| retour à la ligne | 128 |
| backspace | 129 |
| flèche gauche | 130 |
| flèche haut | 131 |
| flèche droite | 132 |
| flèche bas | 133 |
| home | 134 |
| end | 135 |
| page up | 136 |
| page down | 137 |
| insert | 138 |
| delete | 139 |
| escape | 140 |
| f1-f12 | 141-152 |

Les concepteurs du minuscule ordinateur ont réservé une adresse mémoire pour ce registre du clavier : l'adresse `24576` (`0x6000` en hexadécimal). La mémoire du minuscule ordinateur a été conçue de façon à ce que lorsqu'un programme demande à lire le mot se trouvant à cette adresse, il lit le contenu du registre du clavier.

Le programme ci-dessous présente un exemple simple de lecture de caractères depuis le clavier. Le compteur `c` compte simplement le nombre de fois qu'une touche a été pressée.

```
        @c
        M=0
(LOOP)
        @24576 // keyboard
        D=M
        @LOOP
        D;JEQ
        @c
        M=M+1
        @LOOP
        0;JMP
```

Ce programme peut être téléchargé via le lien `asm/keyb.asm`.

Lorsque l'on exécute ce programme en utilisant le simulateur du minuscule CPU, on observe facilement que le compteur n'est incrémenté qu'à condition que la touche soit pressée au moment où le programme lit le mot à l'adresse `24576` en mémoire. Dès que l'utilisateur arrête de pousser sur un touche, ce mot revient à la valeur `0`. Cela implique que sur le minuscule ordinateur, il est nécessaire de consulter très régulièrement l'information stockée à cette adresse pour réagir à la pression d'une touche sur le clavier. C'est le rôle notamment du système d'exploitation, mais cela sort du cadre de ce cours.

Cette technique de lecture de données sous la forme d'une boucle qui lit en permanence l'information mappée en mémoire à une adresse donnée s'appelle le polling. Elle a l'avantage d'être très rapide puisqu'il suffit d'attendre le temps d'exécution de quelques instructions pour que la donnée soit disponible dans le programme. Elle est encore utilisée de nos jours lorsqu'il est nécessaire de réagir très rapidement sur certains dispositifs d'entrée. Malheureusement, elle souffre d'un inconvénient majeur. Le processeur doit en permanence exécuter un programme qui consulte les adresses mappées en mémoire pour voir de l'information est disponible. Pour un dispositif tel que le clavier via lequel l'utilisateur pousse sur quelques touches chaque seconde, il n'est pas souhaitable que le processeur consacre une bonne partie de sa puissance de calcul pour simplement vérifier si une touche a été pressée.

Pour éviter ce problème, les ordinateurs actuels supportent aussi les entrées-sorties par interruption. Les détails de cette technique sortent du cadre de ce cours introductif. En simplifiant, l'idée de base des interruptions est la suivante.

On ajoute sur le minuscule processeur un signal de contrôle baptisé interruption. Ce signal est connecté aux dispositifs d'entrée-sortie. Lorsqu'une nouvelle information est disponible sur un dispositif, celui-ci met le signal d'interruption à `1`. Après l'exécution de chaque instruction, le processeur vérifie la valeur du signal d'interruption. Si celui-ci vaut `0`, il continue l'exécution du programme en cours. Par contre, si le signal d'interruption vaut `1`, le processeur sauvegarde la valeur actuelle du `PC` et passe à l'exécution d'un programme spécial dédié au traitement des interruptions. Ce programme, qui fait généralement partie du système d'exploitation, consulte les différents dispositifs d'entrée-sortie pour voir quelle information est disponible et la traite rapidement. Ensuite, il récupère l'ancienne valeur du `PC` et relance automatiquement l'exécution du programme qui avait été interrompu par l'interruption à l'adresse de l'instruction où il s'était arrêté. Un programme de traitement des interruptions doit être écrit avec précautions car il ne peut perturber le programme qui s'exécutait au moment de l'interruption.

Nous pouvons maintenant étudier l'écran comme exemple de dispositif de sortie. Tout comme pour le clavier, celui-ci utilise la technique des entrées-sorties mappées en mémoire. L'écran du minuscule ordinateur est un écran rectangulaire en noir et blanc de 256 pixels de haut et 512 pixels de large. Il est représenté par un bloc de 8192 adresses en mémoire à partir de l'adresse `16384` (`0x4000` en hexadécimal) en RAM. La valeur de chaque pixel est encodé sur un bit (`1` pour un pixel noir et `0` pour un pixel blanc). Voici un premier exemple qui remplit l'écran en noir en parcourant tous les pixels et toute la mémoire correspondant à l'écran.

```
        @pixel
        M=-1
        @16384   // screen
        D=A
        @pos
        M=D
        @8192
        D=A
        @count
        M=D
(LOOP)
        @pixel
        D=M
        @pos
        A=M
        M=D
        @pos
        M=M+1
        @count
        MD=M-1
        @LOOP
        D;JGT
```

Ce programme peut être téléchargé via le lien `asm/screen.asm`.

En écrivant une donnée en mémoire, on peut donc afficher un pixel à l'écran. L'adresse `16384` correspond au pixel se trouvant dans le coin supérieur gauche de l'écran. Si on attribue les coordonnées `(0,0)` à ce point et que l'axe des ordonnées (*y*) est croissant vers le bas tandis que celui des abscisses (*x*) est croissant vers la droite, alors dans ce repère, le pixel en position *(x,y)* corresponde à l'adresse mémoire $16384 + y \times 32 + x/16$ où / est le quotient de la division entière. Il y a donc 16 pixels qui sont encodés dans le même mot de 16 bits en mémoire. Dans celui-ci, le bit $x\%16$ est celui qui correspond à notre pixel.

En utilisant cette représentation binaire des pixels, il est possible de dessiner des caractères et autres formes géométriques à l'écran. Commençons par écrire un petit caractère que vous reconnaîtrez rapidement. Le caractère dessiné en Fig. 13.1 occupe huit lignes de huit pixels chacune. La première contient l'octet `0`. La deuxième l'octet `00100100` en notation binaire. Les troisième et quatrième contiennent également l'octet `0`. La cinquième contient l'octet `01000010`, la sixième `00100100` et la septième `00111100`. La dernière contient à nouveau l'octet `0`. Pour afficher ce caractère à l'écran, il faut se souvenir que si un pixel se trouve à l'adresse `A`, alors le pixel qui se trouve en dessous de lui est à l'adresse `A+32` puisque chaque ligne de notre écran comprend 512 pixels qui sont encodés sur 32

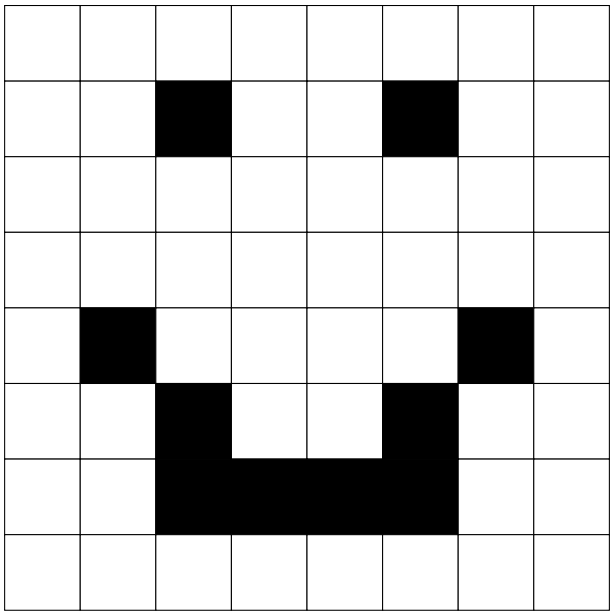


FIG. 13.1 – Un caractère sous la forme de pixels

mots de 16 bits chacun.

Ce caractère peut être affiché à l'écran en utilisant les instructions suivantes.

```
        @pos
        M=0
        @20000   // position du caractère l'écran
        D=A
        @pos
        M=D
        @0   // première ligne
        D=A
        @pos
        A=M
        M=D
        @32
        D=A
        @pos
        M=M+D
        @36  // deuxième ligne
        D=A
        @pos
        A=M
        M=D
        @32
        D=A
        @pos
        M=M+D
        @0   // troisième ligne
        D=A
        @pos
        A=M
        M=D
        @32
```

(suite sur la page suivante)

(suite de la page précédente)

```
        D=A
        @pos
        M=M+D
        @0   // quatrième ligne
        D=A
        @pos
        A=M
        M=D
        @32
        D=A
        @pos
        M=M+D
        @68  // cinquième ligne
        D=A
        @pos
        A=M
        M=D
        @32
        D=A
        @pos
        M=M+D
        @36  // sixième ligne
        D=A
        @pos
        A=M
        M=D
        @32
        D=A
        @pos
        M=M+D
        @60  // septième ligne
        D=A
        @pos
        A=M
        M=D
        @32
        D=A
        @pos
        M=M+D
        @0   // huitième ligne
        D=A
        @pos
        A=M
        M=D
        @32
        D=A
        @pos
        M=M+D
```

Ce programme peut être téléchargé via le lien `asm/screen.asm`.

Notre dernier exemple portera sur le dessin d'un rectangle. Pour simplifier ce dessin, nous supposerons que la longueur de notre rectangle est un multiple de 32 pixels. Notre rectangle sera défini par trois paramètres :

- l'adresse mémoire à laquelle il débute
- sa longueur (un multiple de 32 pixels)
- sa hauteur (en pixels)

Ce programme comprendra deux boucles imbriquées. La première va permettre d'afficher une ligne horizontale noire

de l'adresse `A` à l'adresse `A+long` où `long` est la longueur du rectangle. Cette boucle se trouvera à l'intérieur d'une boucle qui incrémente la position verticale de la ligne de façon à dessiner les `haut` lignes de notre rectangle.

```
// rectangle
        @noir
        M=-1
        @17996   // coin supérieur gauche
        D=A
        @coin
        M=D
        @16      // longueur
        D=A
        @long
        M=D
        @12
        D=A
        @haut   // hauteur
        M=D
        @x
        M=0
        @y
        M=0
        @coin
        D=M
        @addr
        M=D
(BOUCLEH)
        @addr
        D=M
        @addrx
        M=D
        @long
        D=M
        @countx
        M=D
(BOUCLEL)
        @noir
        D=M
        @addrx
        A=M
        M=D
        @addrx
        M=M+1
        @countx
        MD=M-1
        @BOUCLEL
        D;JGT
        @32
        D=A
        @addr
        M=M+D
        @haut
        MD=M-1
        @BOUCLEH
        D;JGT
```

CHAPITRE 14

# Le minuscule ordinateur

Nous avons maintenant tous les composants qui sont nécessaires pour construire notre minuscule ordinateur. Celui-ci sera composé de :

— un minuscule processeur supportant le langage d'assemblage décrit dans les chapitres précédents
— une mémoire RAM qui contiendra les données manipulées par le minuscule processeur
— une mémoire ROM qui contiendra les programmes exécutés par le minuscule processeur
— un clavier qui nous servira d'exemple de dispositif d'entrée
— un écran qui nous servira d'exemple de dispositif de sortie

Ces différents composants interagissent entre eux lors de l'exécution de programmes. Le minuscule processeur lit des données en mémoire RAM ainsi que des instructions en mémoire ROM. Il est aussi capable de lire le code ASCII d'une touche poussée sur le clavier. Le minuscule processeur est aussi capable d'écrire de l'information en mémoire RAM et d'afficher des pixels à l'écran.

Pour que les différents composants de notre minuscule ordinateur puissent interagir entre eux, il est nécessaire qu'ils soient reliés par des fils électriques permettant d'échanger des données et des adresses. Même si notre minuscule ordinateur ne dispose que d'une mémoire, d'un écran et d'un clavier, connecter directement le minuscule CPU à chacun de ces dispositifs nécessiterait un trop grand nombre de pins sur le minuscule CPU. Ce problème a été résolu par l'industrie informatique en utilisant ce que l'on appelle un bus. Un bus est un ensemble de lignes de communications qui facilite l'échange d'information entre dispositifs se trouvant dans un ordinateur donné ou qui sont connectés à cet ordinateur. Au fil des années, l'industrie a développé de nombreux types de bus standardisés qui permettent à des vendeurs différents de produire des composants et cartes d'extension qui peuvent être connectés à des microprocesseurs différents. Parmi les bus de communication les plus connus, on peut citer :

— le bus ISA utilisé sur les premiers IBM PCs
— le bus PCI utilisé par de nombreux PC
— le bus SCSI souvent utilisé pour connecter des dispositifs de stockage et d'entrées/sorties
— le bus SATA utilisé pour connecter de nombreux dispositifs de stockage comme des disques durs ou des lecteurs SSD

Un tel bus de communication permet de connecter plusieurs composants sur le même canal de communication. Comme plusieurs composants sont connectés sur ce canal de communication, il est possible que plusieurs d'entre eux cherchent à envoyer de l'information simultanément. Le spécification du bus définit comment ces conflits sont gérés, mais cela sort du cadre de ce cours introductif. Le point important est qu'un tel bus permet à plusieurs composants de communiquer efficacement en minimisant le nombre de pins utilisées sur chacun de ces composants. Chacun de ces bus définit précisément les différents types de signaux qui peuvent être échangés à travers lui. En pratique, ces signaux sont généralement de trois types :

— des données brutes
— des adresses
— des informations de contrôle comme un signal d'horloge, un signal indiquant une opération d'écriture ou de lecture, etc.

Chaque bus permet l'échange de ces trois types d'information entre tous les composants qui y sont connectés. La Fig. 14.1 décrit l'organisation générale d'un tel bus.

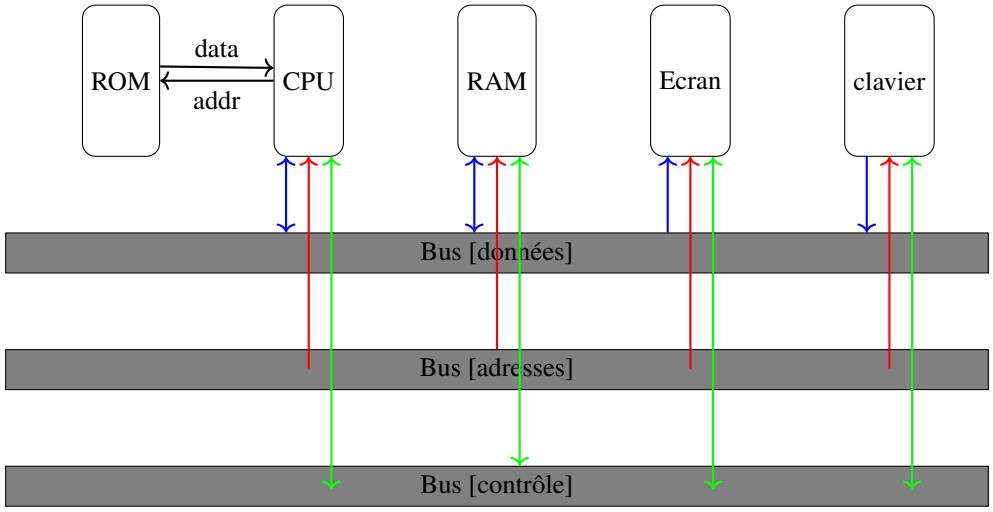



FIG. 14.1 – Architecture du minuscule ordinateur

**Note :** Le livre de référence a choisi, pour simplifier la réalisation des circuits électroniques, une architecture Harvard dans laquelle le microprocesseur est connecté à deux mémoires distinctes :

— une mémoire de type ROM contenant les instructions
— une mémoire de type RAM contenant les données

Ce choix simplifie la réalisation du minuscule ordinateur, mais poserait plusieurs problèmes à un microprocesseur actuel. Premièrement, en stockant le programme à exécuter dans une ROM, on force l'exécution du même programme, cela limite fortement la flexibilité de l'ordinateur. On pourrait bien entendu remplacer cette mémoire ROM par une mémoire de type RAM. Si l'on faisait cette modification, il faudrait également que l'on modifie le microprocesseur pour lui ajouter des instructions qui lui permettent d'écrire dans la mémoire contenant les instructions. Ce n'est pas le cas actuellement. Un autre problème lié à l'utilisation de deux mémoires séparées est qu'il est nécessaire de placer sur le microprocesseur des connexions d'adresse et de données pour la mémoire de données et la mémoire d'instruction. Cela double le nombre de connexions qui doivent être installées sur le microprocesseur. Si l'on veut construire le minuscule processeur sous la forme d'une puce électronique, il faudrait prévoir 16 fils pour recevoir l'instruction, 15 fils pour l'adresse en mémoire ROM mais aussi 16 fils pour l'adresse en mémoire RAM et 16 fils pour la donnée venant de cette mémoire. En combinant les mémoires de données et d'instruction, on divise par deux le nombre de fils qui doivent être connectés au microprocesseur. C'est très important au niveau de leur construction.

Les ordinateurs actuels utilisent l"architecture de von Neumann dans laquelle les programmes et les données sont stockées dans la même mémoire. Cette architecture avait été proposée par John von Neumann en 1945.

## 14.1 Le minuscule CPU

Pour pouvoir construire notre minuscule CPU il est important de bien identifier les différents signaux d'entrée qu'il va devoir traiter ainsi que les valeurs de sortie qu'il va produire. Ces signaux sont naturellement liés aux instructions que notre CPU va exécuter.

Le premier signal d'entrée de notre CPU sera un mot de 16 bits contenant l'instruction à exécuter en binaire (le livre utilise *instruction* comme nom pour cet ensemble de 16 bits). Cette entrée sera lue à chaque cycle d'horloge par notre CPU pour décoder l'instruction courante. Cela nous permettra d'exécuter une instruction de type `A` ou une instruction telle que `D=D+1`. Ce n'est cependant pas suffisant car certaines instructions font référence à un mot de 16 bits se trouvant à l'adresse contenue dans le registre `A`. C'est le cas d'une instruction telle que `D=M-1`. Pour supporter ces instructions, notre minuscule CPU devra, durant certains cycles d'horloge, lire le contenu d'un mot en mémoire à l'adresse se trouvant dans le registre `A`. Le livre utilise *inM* comme nom pour cet ensemble de 16 bits.

Nous pouvons maintenant réfléchir aux sorties du minuscule CPU. La valeur calculée par son ALU peut être stockée en mémoire. C'est le cas lors de l'exécution d'instructions telles que `M=D+1` ou `M=M+D`. Cela nécessite un ensemble de seize lignes de sortie que le livre nomme *outM*. Outre la valeur calculée par l'ALU, notre CPU doit aussi pouvoir spécifier une adresse mémoire à laquelle la donnée doit être écrite. Cela nécessite quinze bits puisque la mémoire RAM ne contient que $2^{15}$ mots de 16 bits. Le livre utilise le nom *addressM* pour cette sortie. Ces deux sorties sont connectées à la mémoire RAM, mais elles ne sont pas suffisantes. Il nous reste un petit détail à régler. Lors de l'exécution d'une instruction telle que `M=D-1`, la valeur émise sur les signaux *outM* doit être stockée à l'adresse correspondant à la sortie *addressM*. Par contre, lors de l'exécution de l'instruction *D=A+1*, aucune information ne doit être stockée en mémoire RAM, même si une valeur (éventuellement *0*) est émise sur les signaux *outM* et *addressM*. Pour éviter tout risque de confusion au niveau de la mémoire, notre minuscule CPU définit un signal de contrôle baptisé *writeM* qui est mis à *1* lorsque la valeur se trouvant sur *outM* doit être écrite en mémoire à l'adresse *addressM* et *0* sinon.

Les interactions entre le minuscule CPU et le reste de l'ordinateur sont maintenant presque complètes. Il nous reste à gérer le chargement des instructions depuis la mémoire ROM. Comme celle-ci contient $2^{15}$ mots, nous avons besoin de 15 bits de sortie, baptisées *PC* sur notre minuscule CPU. Cette sortie sera naturellement connectée à la mémoire ROM qui retourne l'instruction lue sur les lignes *instruction* de notre CPU. La sortie *PC* sera directement connectée au registre `PC` de notre CPU. Il nous reste un dernier détail à régler. En cas de problème comme une boucle infinie ou un comportement bizarre, il est utile d'équiper notre minuscule ordinateur d'un signal *reset*. Sur une machine réelle, celui-ci serait par exemple relié à un bouton poussoir qui est connecté au minuscule CPU. Lorsque ce signal d'entrée passe à *1*, le minuscule CPU doit automatiquement arrêter l'exécution du programme en cours et redémarrer à l'instruction se trouvant à l'adresse `0`. Il nous suffira pour cela de forcer une initialisation à *0* du registre *PC* lorsque le signal d'entrée *reset* est mis à *1*.

La Fig. 14.2 résume les signaux d'entrée et de sortie du minuscule CPU. Nous avions précédemment construit la

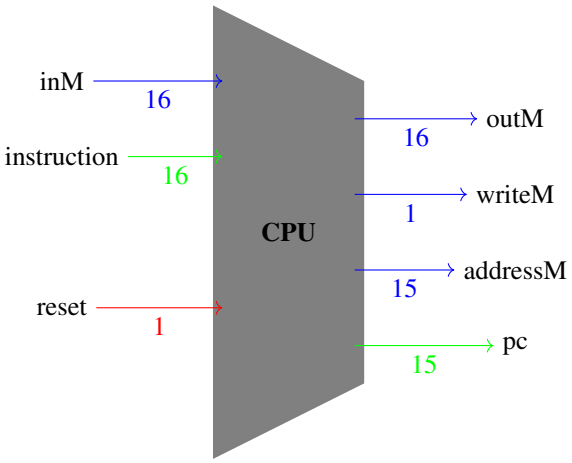



FIG. 14.2 – Minuscule CPU

mémoire RAM que nous pouvons connecter à notre minuscule CPU. Notre mémoire avait une capacité de 16K mots de 16 bits. Elle utilise 14 bits d'adresse (entrée *address*). Elle dispose aussi d'une entrée sur 16 bits (*in*). Le mot de 16 bits présent sur cette entrée est écrit en mémoire RAM lorsque le signal de contrôle *loadRAM* est mis à *1*. Enfin, la mémoire dispose d'une sortie (*out*) sur seize bits également.

Nous pouvons maintenant connecter la mémoire RAM avec le minuscule CPU. Il suffit pour cela de relier la sortie *addressM* du CPU à l'entrée *address* de notre mémoire RAM. De même, la sortie *outM* du CPU doit être connectée à l'entrée *in* de la mémoire RAM. La sortie de la mémoire RAM doit elle être reliée à l'entrée *inM* du minuscule CPU. Il nous reste enfin à relier la sortie *writeM* du minuscule CPU à l'entrée *loadRAM* de notre RAM. Cette interconnexion est représentée en Fig. 14.3. Il nous faudra ensuite ajouter l'écran et le clavier pour compléter notre ordinateur. Il ne

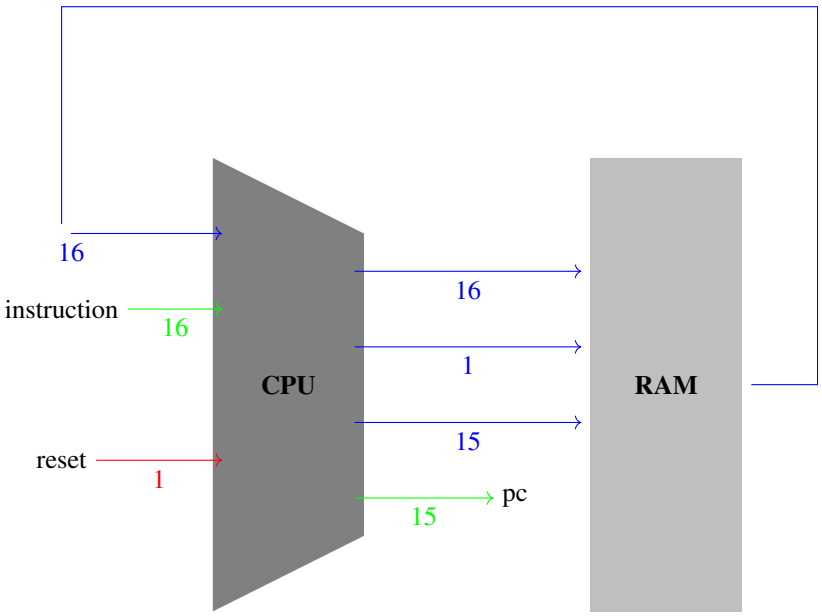



FIG. 14.3 – Connexions entre le minuscule CPU et la RAM

nous reste plus qu'à relier le minuscule CPU à la mémoire ROM. Pour cela, il suffit de relier la sortie de la ROM à l'entrée *instruction* du CPU et la sortie *pc* du CPU à l'entrée *address* de cette ROM. Les interconnexions entre le minuscule CPU et les mémoires sont représentées en Fig. 14.4.

## 14.2 Construction du minuscule CPU

Avant de commencer à construire le minuscule CPU, nous devons d'abord réfléchir à la façon dont celui-ci va exécuter les instructions qui se trouvent en mémoire ROM. Notre objectif est de pouvoir exécuter une instruction se trouvant en mémoire ROM durant chaque cycle d'horloge. Durant chacun de ces cycles d'horloge, notre minuscule processeur devra procéder comme représenté sur la Fig. 14.5. Premièrement, le minuscule processeur doit charger (*fetch* en anglais) l'instruction à exécuter à l'adresse contenue dans le registre `PC`. Ensuite, il faut décoder cette instruction. Enfin, il faut exécuter cette instruction et par exemple charger ou sauver un mot en mémoire. Nous pouvons maintenant commencer la construction du minuscule CPU. Pour cela, nous pouvons réutiliser les circuits construits dans les précédents chapitres :

- une ALU
- un registre `A`
- un registre `D`
- un registre pour stocker la valeur du `PC`

Chacun de ces éléments de base pourra être utilisé lors de l'exécution d'une instruction particulière. Pour rappel notre ALU dispose de huit entrées et trois sorties. Les entrées sont :

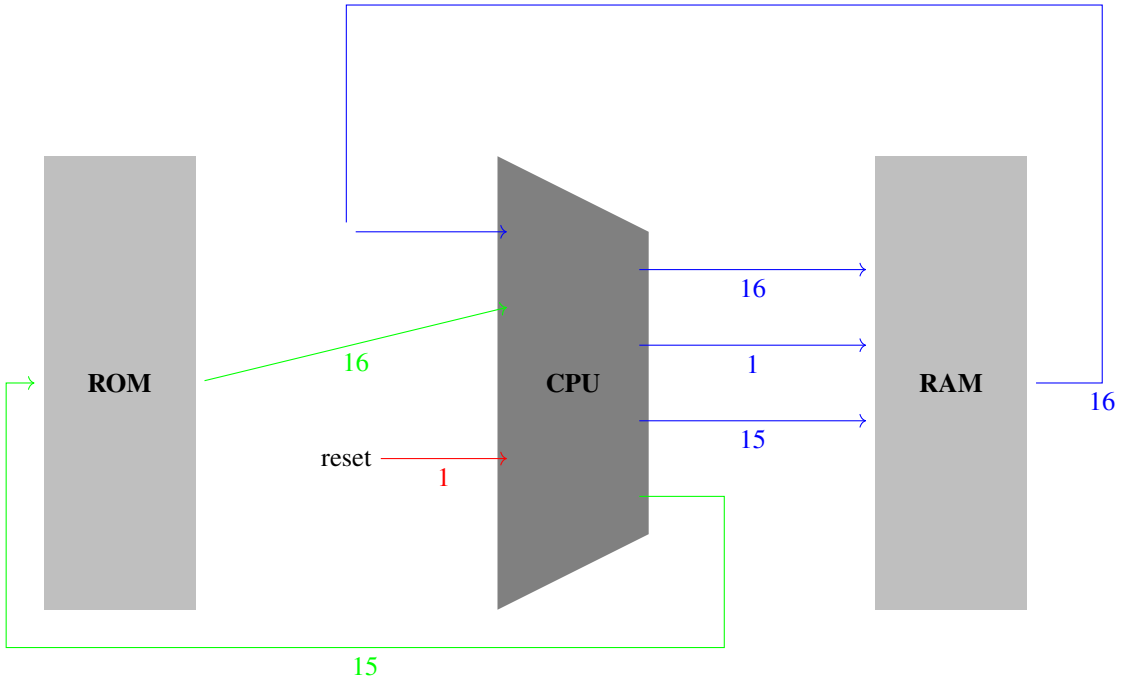



FIG. 14.4 – Connexions entre le minuscule CPU et les mémoires

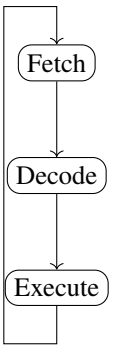



FIG. 14.5 – Le cycle Fetch-Decode-Execute

— le premier mot de seize bits (*x*)
— le second mot de seize bits (*y*)
— le signal de contrôle *zx* qui indique si l'entrée *x* doit être mise à zéro
— le signal de contrôle *zy* qui indique si l'entrée *y* doit être mise à zéro - le signal de contrôle *nx* qui indique si l'entrée *x* doit être inversée
— le signal de contrôle *ny* qui indique si l'entrée *y* doit être inversée
— le signal de contrôle *f* qui permet de choisir entre le résultat de l'additionneur et celui de la porte *AND* comme sortie de l'ALU
— le signal de contrôle *no* qui détermine si la sortie doit être inversée ou non

Les trois sorties de l'ALU sont :

— le mot de seize bits qui est le résultat du calcul
— le signal de contrôle *zr* qui est mis à *1* si le résultat du calcul est égal à zéro
— le signal de contrôle *ng* qui est mis à *1* si le résultat du calcul est négatif

Il ne nous reste plus qu'à connecter ces différents composants ensemble de façon à pouvoir supporter toutes les instructions que nous avons présentées dans les chapitres précédents. La Fig. 14.6 présente un schéma bloc de notre minuscule CPU que nous allons compléter petit à petit.

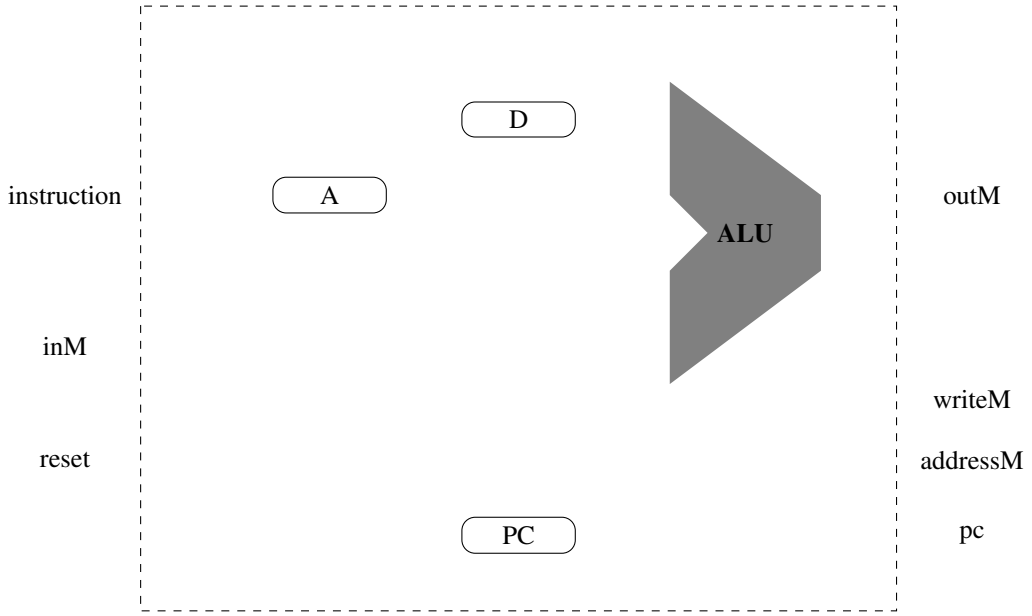



FIG. 14.6 – Composition du minuscule CPU

**Les deux registres `A` et `D` permettent de stocker un mot de seize bits. Ils ont chacun deux entrées et une sortie :**

— une entrée *in* sur 16 bits
— une sortie *out* sur 16 bits
— un signal de contrôle *load* qui doit être mis à *1* pour que le registre mémorise l'information présente sur son entrée *in*

Le registre `PC` est lui plus complexe. Dans le troisième projet, nous avons vu que ce registre avait quatre entrées :

— un mot de 16 bits contenant une nouvelle valeur à stocker (*in*)
— un signal de contrôle *inc* qui détermine si le contenu du `PC` doit être incrémenté
— un signal de contrôle *reset* qui initialise son contenu à `0`
— un signal de contrôle *load* qui force le chargement de la valeur se trouvant sur l'entrée *in*

Ce registre a une sortie sur 16 bits baptisée *out*. Dans un premier temps, considérons uniquement l'incrémentation et la réinitialisation de ce registre. Pour cela, il nous suffit de connecter la valeur *1* à l'entrée *inc* du *PC* et son signal de contrôle *reset* au signal extérieur. Nous verrons ultérieurement comment utiliser les autres signaux de contrôle de ce registre, mais nous avons déjà un registre `PC` qui s'incrémente à la fin de l'exécution de chaque instruction.

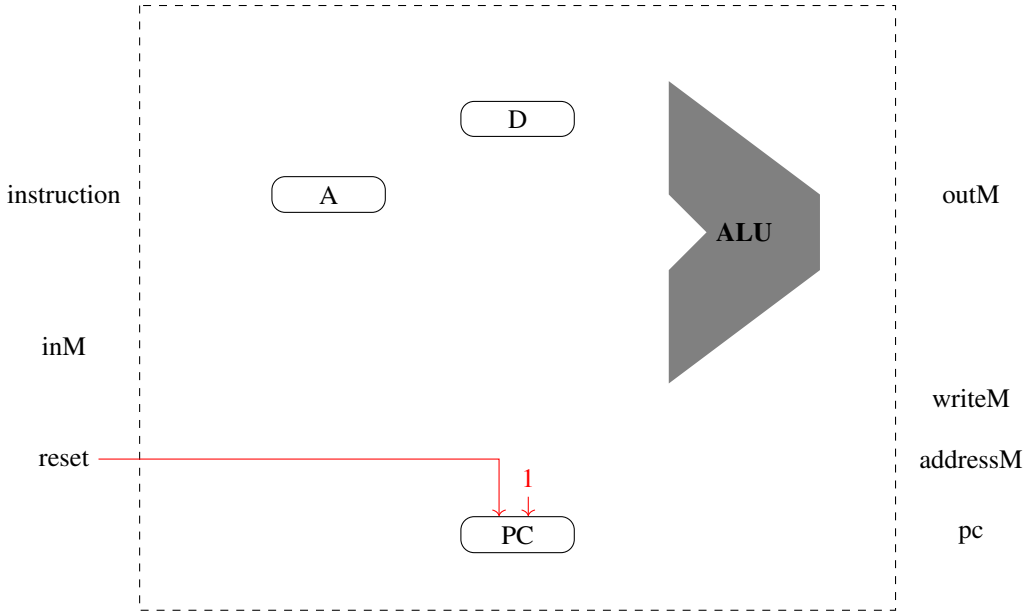



FIG. 14.7 – Un registre PC de base

### 14.2.1 Les instructions

Pour poursuivre la construction de notre CPU, nous devons maintenant analyser plus en détails les différentes instructions qu'il doit exécuter. Chaque instruction de notre minuscule CPU est encodée sous la forme d'un mot de 16 bits. Comme indiqué précédemment, ce CPU supporte deux types d'instructions :

— les instructions de type `A` qui permettent de charger la valeur se trouvant dans les quinze bits de poids faible de l'instruction dans le registre `A`
— les instructions de type `C` qui comprennent toutes les autres instructions

Notre minuscule CPU utilise le bit de poids fort de l'instruction pour déterminer si il s'agit d'une instruction de type `A` (bit de poids fort mis à *0*) ou de type `C` (bit de poids fort mis à *1*).

Commençons par analyser les instructions de type `A`. Une de ces instructions permet de charger dans le registre `A` la valeur correspondant aux quinze bits de poids faible du mot de seize bits contenant l'instruction. Pour supporter cette instruction, nous devons donc :

— mettre le signal de contrôle *load* du registre `A` à *1* lorsque le bit de poids fort de l'instruction lue en mémoire ROM a bien la valeur *0*
— connecter les quinze bits de poids faible de l'instruction lue en mémoire ROM sur l'entrée *in* du registre `A`

Pour mettre à *1* le signal de contrôle de registre `A` lorsque le bit de poids fort de l'instruction vaut *0*, il suffit de faire passer ce bit dans un inverseur avant de le connecter à l'entrée *load* du registre `A`. Pour supporter les instructions de type `C`, il est nécessaire de s'intéresser plus en détails à la façon dont elles sont encodées en binaire. Le format de ces instructions est repris ci-dessous.

$$111\overbrace{ac_1c_2c_3c_4c_5c_6}^{calcul}\ \overbrace{d_1d_2d_3}^{destination}\ \overbrace{j_1j_2j_3}^{saut}$$

Les seize bits de cette instruction sont découpés en trois parties :

— les sept bits *calcul* spécifient le type de calcul à réaliser
— les trois bits *destination* spécifient l'endroit où le résultat du calcul doit être stocké
— les trois bits de poids faible sont utilisés pour les instructions de saut

Parmi les bits de *calcul*, le bit *a* joue un rôle particulier. Lorsqu'il vaut *1*, le calcul fait par l'ALU utilise une donnée lue en mémoire RAM à l'adresse contenue dans le registre `A`. Sinon, l'ALU réalise son calcul sur base des constantes *0* et *1* ainsi que du contenu des registres `A` et/ou `D`. Nous devons donc prévoir la possibilité d'amener une donnée lue en mémoire à l'une des entrées de la minuscule ALU. En pratique, le livre a choisi de connecter la sortie du registre `D`

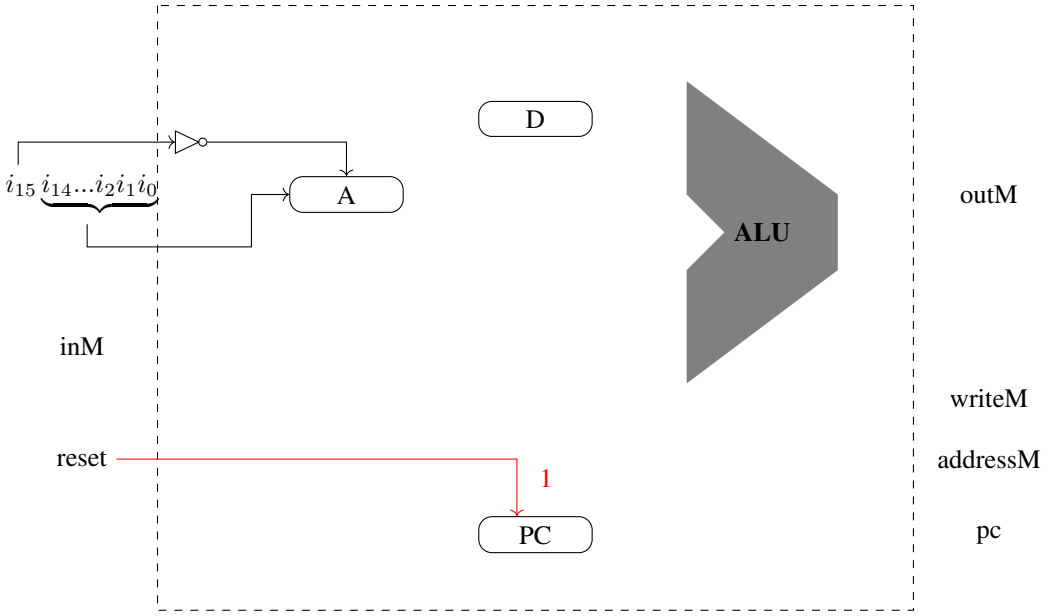



FIG. 14.8 – Support de l'instruction de type A

à l'entrée *x* de l'ALU et de connecter la sortie du registre `A` ou la donnée lue en mémoire à l'adresse contenue dans le registre `A` à l'entrée *y*. Pour réaliser cette lecture en mémoire, nous devons donc connecter la sortie du registre `A` à la sortie *addressM* du minuscule *CPU*. La seconde entrée de la minuscule ALU doit elle être la donnée lue en mémoire lorsque le bit *a* de l'instruction vaut *1* et sinon ce doit être le contenu du registre `A`. Pour implémenter ce choix, il suffit d'utiliser un multiplexeur qui est commandé par le bit *a* de l'instruction de type `C`. Ces connexions sont illustrées en Fig. 14.9. Nous pouvons maintenant analyser plus en détails les différentes instructions de type `C` pour voir comment les implémenter. Pour chacune de ces instructions, la procédure à suivre est la suivante. Tout d'abord, il faut extraire des bits $c_1c_2c_3c_4c_5c_6$ les informations qui permettent de choisir les bonnes valeurs pour les entrées et les signaux de contrôle de l'ALU. Ensuite, il faudra faire de même pour la destination du résultat du calcul réalisé par la minuscule ALU en utilisant les bits $d_1d_2d_3$. Pour cela, nous devons analyser en détails les valeurs de ces différents bits dans les instructions qui nous intéressent. Dans le minuscule CPU, les formats de ces bits ont été choisies de façon à faciliter la réalisation des circuits qui permettent de décoder chaque instruction. Le Tableau 14.1, extrait du livre de référence, présente l'encodage des bits $c_i$ pour les instructions de type `C` lorsque le bit *a* est mis à 0.

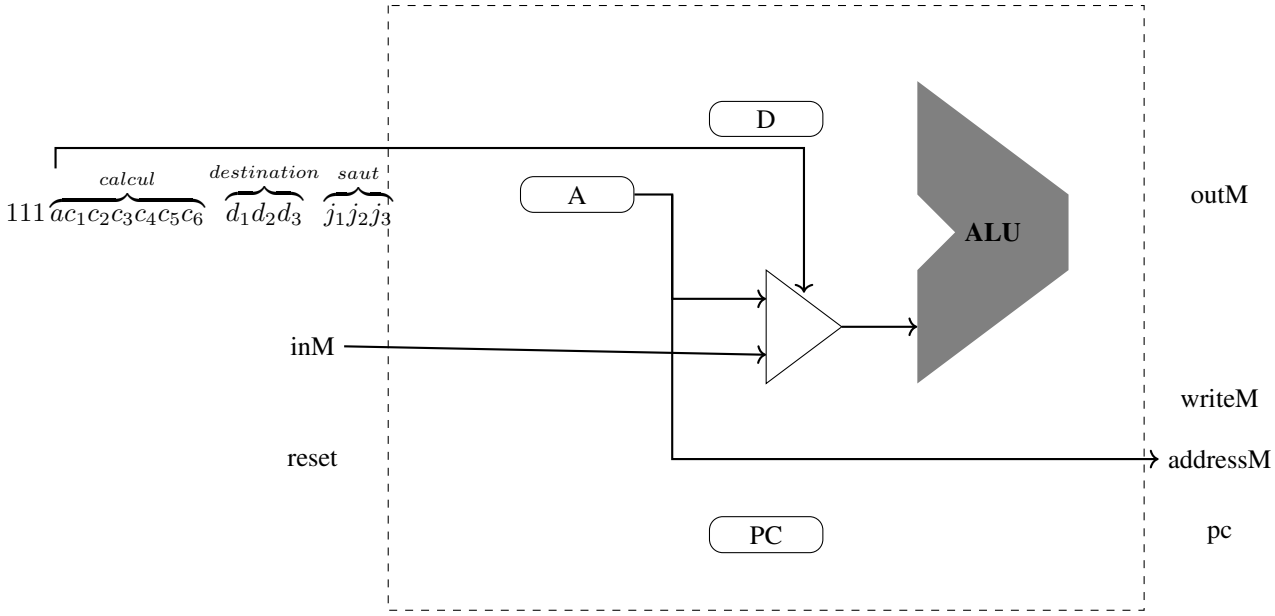



FIG. 14.9 – Utilisation du bit a des instructions de type C

TABLEAU 14.1 – Valeurs des bits calcul des instructions de type C lorsque le bit a est à 0

| Calcul | $c_1$ | $c_2$ | $c_3$ | $c_4$ | $c_5$ | $c_6$ |
|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| -1 | 1 | 1 | 1 | 0 | 1 | 0 |
| D | 0 | 0 | 1 | 1 | 0 | 0 |
| A | 1 | 1 | 0 | 0 | 0 | 0 |
| !D | 0 | 0 | 1 | 1 | 0 | 1 |
| !A | 1 | 1 | 0 | 0 | 0 | 1 |
| -D | 0 | 0 | 1 | 1 | 1 | 1 |
| -A | 1 | 1 | 0 | 0 | 1 | 1 |
| D+1 | 0 | 1 | 1 | 1 | 1 | 1 |
| A+1 | 1 | 1 | 0 | 1 | 1 | 1 |
| D-1 | 0 | 0 | 1 | 1 | 1 | 0 |
| A-1 | 1 | 1 | 0 | 0 | 1 | 0 |
| D+A | 0 | 0 | 0 | 0 | 1 | 0 |
| D-A | 0 | 1 | 0 | 0 | 1 | 1 |
| A-D | 0 | 0 | 0 | 1 | 1 | 1 |
| D&A | 0 | 0 | 0 | 0 | 0 | 0 |
| D\|A | 0 | 1 | 0 | 1 | 0 | 1 |

Lorsque le bit *a* est mis à *1*, la seconde entrée de la minuscule ALU est la donnée lue en mémoire. Dans ce cas, seules les instructions du Tableau 14.2 sont valides.

TABLEAU 14.2 – Valeurs des bits calcul des instructions de type C lorsque le bit a est à 1

| Calcul | $c_1$ | $c_2$ | $c_3$ | $c_4$ | $c_5$ | $c_6$ |
|---|---|---|---|---|---|---|
| M | 1 | 1 | 0 | 0 | 0 | 0 |
| !M | 1 | 1 | 0 | 0 | 0 | 1 |
| -M | 1 | 1 | 0 | 0 | 1 | 1 |
| M+1 | 1 | 1 | 0 | 1 | 1 | 1 |
| M-1 | 1 | 1 | 0 | 0 | 1 | 0 |
| D+M | 0 | 0 | 0 | 0 | 1 | 0 |
| D-M | 0 | 1 | 0 | 0 | 1 | 1 |
| M-D | 0 | 0 | 0 | 1 | 1 | 1 |
| D&M | 0 | 0 | 0 | 0 | 0 | 0 |
| D\|M | 0 | 1 | 0 | 1 | 0 | 1 |

Pour compléter la description des instructions de type `C`, le Tableau 14.3 présente les valeurs des bits $d_1d_2d_3$ qui encodent la destination du calcul réalisé par la minuscule ALU.

TABLEAU 14.3 – Valeurs des bits destination des instructions de type C

| Destination | $d_1$ | $d_2$ | $d_3$ |
|---|---|---|---|
| aucune | 0 | 0 | 0 |
| M | 0 | 0 | 1 |
| D | 0 | 1 | 0 |
| MD | 0 | 1 | 1 |
| A | 1 | 0 | 0 |
| AM | 1 | 0 | 1 |
| AD | 1 | 1 | 0 |
| AMD | 1 | 1 | 1 |

**En observant cette table, on remarque aisément que :**

- le résultat du calcul de l'ALU est stocké dans le registre `A` lorsque le bit $d_1$ vaut *1*
- le résultat du calcul de l'ALU est stocké dans le registre `D` lorsque le bit $d_2$ vaut *1*
- le résultat du calcul de l'ALU est stocké en mémoire RAM lorsque le bit $d_3$ vaut *1*

Nous devons donc relier la sortie de la minuscule ALU à la sortie *outM*, mais aussi aux entrées de registres `D` et `A`. Pour le registre `D`, cette connexion ne posera pas de problème. Par contre, pour le registre `A`, nous devons nous rappeler que nous y avons déjà connecté les quinze bits de poids faible de l'instruction lue en mémoire ROM pour supporter les instructions de type `A`. Comme nous avons deux entrées possibles pour le registre `A`, il nous suffit des les connecter à un multiplexeur qui est placé devant l'entrée de ce registre. Ce multiplexeur sera commandé par le bit de poids fort de l'instruction. Lorsque ce bit vaut *0* (instruction de type `A`), il doit sélectionner son entrée avec les 15 bits de poids faible de l'instruction. Sinon, il sélectionne l'entrée provenant de la sortie de l'ALU. Pour simplifier les schémas, nous présentons maintenant les bits de contrôle de façon symbolique. Le registre `A` doit charger la valeur en entrée dans deux cas :

- on exécute une instruction de type `A` et donc le bit $i_{15}$ est à *0* comme expliqué précédemment
- on exécute une instruction de type `C` dont le bit $d_1$ vaut *1*

Il nous suffit donc d'utiliser le signal $OR(d_1, NOT(i_{15}))$ pour contrôler le registre `A` et $NOT(i_{15})$ pour le multiplexeur se trouvant en amont du registre `A`. Le registre `D` lui devra sauvegarder son entrée lorsque le bit $d_2$ vaut *1*. Le dernier cas est celui d'une sauvegarde du résultat de l'ALU en mémoire. Dans ce cas, il faut que signal *writeM* du minuscule CPU soit mis à *1*. Il suffit pour cela de simplement relier le bit $d_3$ de l'instruction directement à cette sortie. La Fig. 14.10 décrit cette partie du minuscule CPU. Nous pouvons maintenant nous concentrer sur la partie calcul des instructions de type `C`. Nous nous limiterons à illustrer comment quelques unes de ces instructions peuvent être implémentées. Les étudiants sont invités à construire le minuscule CPU entièrement comme exercice.

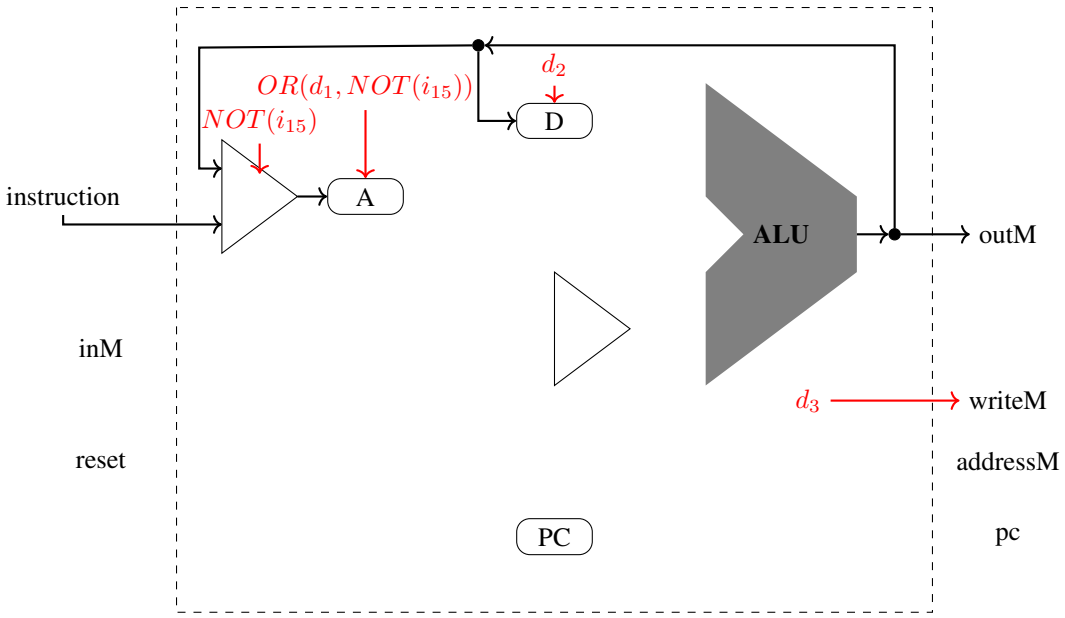



FIG. 14.10 – Choix de la destination du calcul de l'ALU

Commençons par utiliser l'ALU pour calculer la constante *0*. Lorsque nous avons construit la minuscule ALU, cette valeur était obtenue en utilisant les signaux de contrôle suivants :

- *zx=1*
- *nx=0*
- *zy=1*
- *ny=0*
- *f=1*
- *no=0*

La minuscule ALU doit réaliser cette opération pour l'instruction suivante :

- $c_1 = 1$
- $c_2 = 0$
- $c_3 = 1$
- $c_4 = 0$
- $c_5 = 1$
- $c_5 = 0$

Pour supporter cette instruction, il nous suffit donc de relier le bit $c_1$ à l'entrée *zx* de la minuscule ALU, le bit $c_2$ à l'entrée *zy*, ...

Analysons maintenant comment calculer la somme entre le registre `D` et le registre `A` ou la valeur lue en mémoire. Pour réaliser cette opération d'addition, nous devons fixer les valeurs suivants aux signaux de contrôle de la minuscule ALU :

- *zx=0*
- *nx=0*
- *zy=0*
- *ny=0*
- *f=1*
- *no=0*

**Notre minuscule ALU doit réaliser cette opération pour l'instruction suivante :**

- $c_1 = 0$
- $c_2 = 0$
- $c_3 = 0$

— $c_4 = 0$
— $c_5 = 1$
— $c_5 = 0$

En continuant l'analyse, on remarque aisément que les bits $c_1$ à $c_6$ extraits de l'instruction correspondent exactement aux bits de contrôle de la minuscule ALU. Il suffit donc d'extraire les valeurs de ces bits de l'instruction lue en mémoire et des les connecter sur les entrées de la minuscule ALU. Pour supporter toutes les instructions du minuscule CPU, il

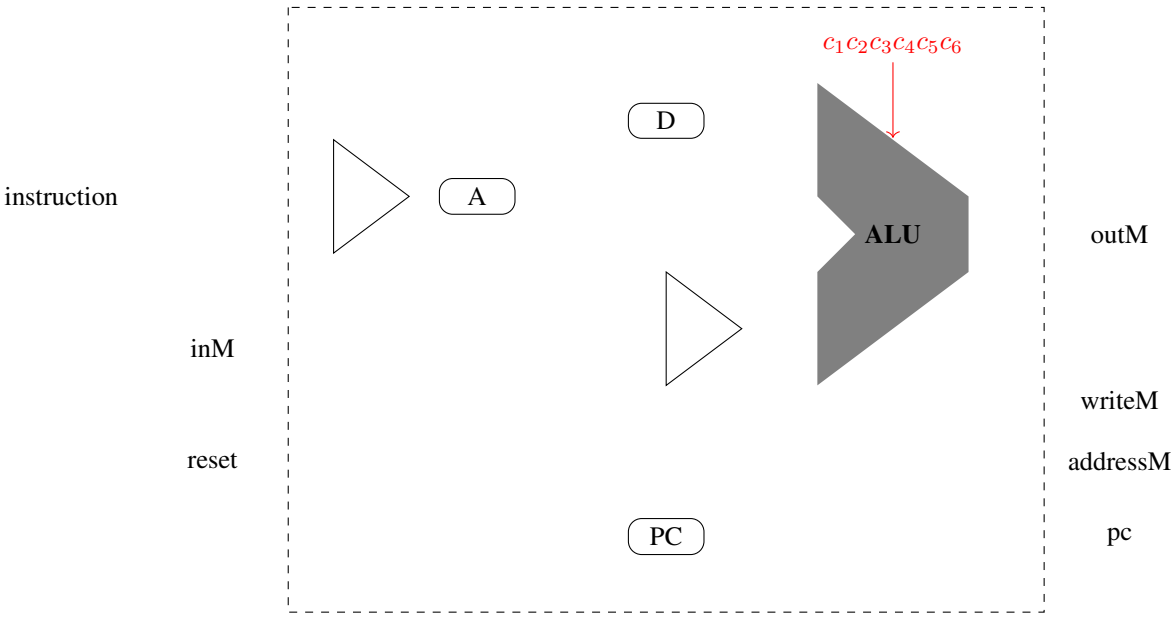



FIG. 14.11 – Connexion des bits de calcul de l'instruction à la minuscule ALU

nous reste à analyser les instructions de saut qui permettent de modifier le contenu du registre `PC`. Le type de saut est encodé dans les trois bits de poids faible de l'instruction. Nous pouvons distinguer trois types de sauts :

— pas de saut à réaliser lorsque les trois bits de poids faible de l'instruction valent *000*
— saut inconditionnel à l'adresse se trouvant dans le registre `A` lorsque les trois bits de poids faible de l'instruction valent *111*
— saut conditionnel pour les autres valeurs des bits de poids faible

Le Tableau 14.4 présente les différents types de sauts qui sont supportés par le minuscule CPU.

TABLEAU 14.4 – Valeurs des bits de poids faible des instructions de type C

| Saut | $j_1$ | $j_2$ | $j_3$ |
|---|---|---|---|
| — | 0 | 0 | 0 |
| JGT | 0 | 0 | 1 |
| JEQ | 0 | 1 | 0 |
| JGE | 0 | 1 | 1 |
| JLT | 1 | 0 | 0 |
| JNE | 1 | 0 | 1 |
| JLE | 1 | 1 | 0 |
| JMP | 1 | 1 | 1 |

Nous avons précédemment expliqué comment le registre `PC` pouvait être mis à jour en l'absence de saut. Nous devons maintenant repartir de ce premier circuit et analyser comment il doit être modifié pour prendre en compte les différentes

instructions de saut. Tout d'abord, il faut remarquer que le contenu du registre PC doit être incrémenté, c'est-à-dire que son entrée *inc* doit être à *1* et son entrée *load* à zéro lorsque l'on exécute une instruction de type A (bit de poids fort du mot contenant l'instruction mis à *0*) ou une instruction de type C (bit de poids fort mis à *1*) qui n'est pas un saut (bits $j_1 j_2 j_3$ à *0*). L'entrée *inc* de notre registre PC doit donc être $OR(i_{15}, AND(NOT(j_1), NOT(j_2), NOT(j_3)$.

Nous devons maintenant analyser les conditions dans lesquelles le registre PC doit charger la valeur venant du registre A. Ces conditions dépendent à la fois de l'instruction en cours d'exécution et du résultat de la minuscule ALU et plus particulièrement des valeurs de drapeaux *zr* et *ng*. Pour rappel, *zr* est mis à *1* lorsque le résultat de l'ALU est nul. Le drapeau *ng* indique un résultat négatif de l'ALU. Nous sommes en fait face à la construction d'un circuit logique qui a cinq entrées :

— le bit $j_1$
— le bit $j_2$
— le bit $j_3$
— le drapeau *zr*
— le drapeau *ng*

Ce circuit logique va avoir comme sortie la valeur du signal de contrôle *load* du registre PC. Pour construire le circuit logique correspondant, il suffit de construire sa table de vérité (Tableau 14.5). Cette table de vérité aura donc 32 lignes. Pour construire cette table de vérité, il faut se souvenir du fonctionnement des différents instructions de saut et les conditions qui doivent être remplies pour que le contenu du PC prenne la valeur du registre A.

Le premier cas correspond aux instructions dont les trois bits de poids faible sont à zéro. Dans ce cas, *load* est toujours à zéro quelles que soient les valeurs de *zr* et *ng*.

Le deuxième cas correspond à l'instruction inconditionnelle JMP (bits de poids faible à *1*). Dans ce cas, *load* est toujours mis à *1*, quelles que soient les valeurs de *zr* et *ng*.

Le troisième cas est celui de l'instruction JGT. Lors de l'exécution de cette instruction, le bit de contrôle *load* doit être mis à *1* lorsque *zr=0* et *ng=0*. Sinon, il est mis à *0*.

Le quatrième cas correspond à l'instruction JEQ. Dans ce cas, le bit *load* doit être mis à *1* lorsque *zr=1* et *ng=0*. Sinon, il est mis à *0*.

Le cinquième cas est celui de l'instruction JGE. Pour cette instruction, le bit *load* doit être mis à *1* lorsque *ng* vaut *0*, quelle que soit la valeur de *zr*

Le sixième cas est celui de l'instruction JLT. Lors de l'exécution de cette instruction, le bit de contrôle *load* doit être mis à *1* lorsque *zr=0* et *ng=1*. Sinon, il est mis à *0*.

La septième instruction est JNE. Pour cette instruction, le bit de contrôle *load* doit valoir *1* pour autant que *zr* soit mis à *0*.

Le dernier cas est celui de l'instruction JLE. Lors de l'exécution de cette instruction, le bit de contrôle *load* doit être mis à *1* lorsque *ng=0*, quelle que soit la valeur de *zr*.

TABLEAU 14.5: Table de vérité du calcul du signal de contrôle

| Saut | *zr* | *ng* | $j_1$ | $j_2$ | $j_3$ | *load* |
|---|---|---|---|---|---|---|
| — | 0 | 0 | 0 | 0 | 0 | 0 |
| JGT | 0 | 0 | 0 | 0 | 1 | 1 |
| JEQ | 0 | 0 | 0 | 1 | 0 | 0 |
| JGE | 0 | 0 | 0 | 1 | 1 | 1 |
| JLT | 0 | 0 | 1 | 0 | 0 | 0 |
| JNE | 0 | 0 | 1 | 0 | 1 | 1 |
| JLE | 0 | 0 | 1 | 1 | 0 | 1 |
| JMP | 0 | 0 | 1 | 1 | 1 | 1 |

suite sur la page suivante

Tableau 14.5 – suite de la page précédente

| | | | | | | |
|---|---|---|---|---|---|---|
| — | 0 | 1 | 0 | 0 | 0 | 0 |
| JGT | 0 | 1 | 0 | 0 | 1 | 0 |
| JEQ | 0 | 1 | 0 | 1 | 0 | 0 |
| JGE | 0 | 1 | 0 | 1 | 1 | 0 |
| JLT | 0 | 1 | 1 | 0 | 0 | 1 |
| JNE | 0 | 1 | 1 | 0 | 1 | 1 |
| JLE | 0 | 1 | 1 | 1 | 0 | 0 |
| JMP | 0 | 1 | 1 | 1 | 1 | 1 |
| — | 1 | 0 | 0 | 0 | 0 | 0 |
| JGT | 1 | 0 | 0 | 0 | 1 | 0 |
| JEQ | 1 | 0 | 0 | 1 | 0 | 1 |
| JGE | 1 | 0 | 0 | 1 | 1 | 1 |
| JLT | 1 | 0 | 1 | 0 | 0 | 0 |
| JNE | 1 | 0 | 1 | 0 | 1 | 0 |
| JLE | 1 | 0 | 1 | 1 | 0 | 1 |
| JMP | 1 | 0 | 1 | 1 | 1 | 1 |
| — | 1 | 1 | 0 | 0 | 0 | 0 |
| JGT | 1 | 1 | 0 | 0 | 1 | 0 |
| JEQ | 1 | 1 | 0 | 1 | 0 | 0 |
| JGE | 1 | 1 | 0 | 1 | 1 | 0 |
| JLT | 1 | 1 | 1 | 0 | 0 | 0 |
| JNE | 1 | 1 | 1 | 0 | 1 | 0 |
| JLE | 1 | 1 | 1 | 1 | 0 | 0 |
| JMP | 1 | 1 | 1 | 1 | 1 | 1 |

Le Tableau 14.5 contient la table de vérité complète du circuit permettant de calculer le signal de contrôle nécessaire pour supporter les instructions de saut. Il suffit maintenant de transformer cette table de vérité en un circuit logique. Cette transformation est laissée aux étudiants à titre d'exercice. Il est possible de réaliser ce circuit en utilisant peu de fonctions logiques.

# CHAPITRE 15

# Ordinateurs actuels

Le livre de référence et les chapitres précédents nous ont permis de voir les éléments principaux du fonctionnement d'un ordinateur qui est capable d'exécuter des programmes simples écrits en langage d'assemblage. Le minuscule ordinateur est complètement fonctionnel et le livre de référence l'utilise pour développer des logiciels qui permettent de l'exploiter pleinement.

L'approche choisie par le livre de référence est pédagogique. L'ordinateur construit fonctionne mais il est loin d'être équivalent aux ordinateurs et aux microprocesseurs qui existent de nos jours. En une septantaine d'années environ, les ordinateurs et les microprocesseurs ont fait d'immenses progrès. Il est impossible de les lister tous dans ce cours introductif. Vous aurez plus tard l'occasion d'analyser ces techniques avancées plus en détails notamment dans les cours de Master. Cependant, il y a certaines contraintes technologiques auxquelles il est intéressant que vous soyez déjà sensibilisé.

La complexité d'un microprocesseur se mesure d'abord grâce au nombre de transistors qui le composent. En fonction de la technologie utilisée, il faut compter que quelques transistors sont nécessaires pour construire une porte logique de type NAND ou NOR. A partir de ces portes logiques, il est possible de construire un ordinateur complet comme nous l'avons vu. La Fig. 15.1 présente l'évolution du nombre de transistors que contiennent les microprocesseurs commerciaux depuis l'intel 4004 jusqu'au récent Apple M1. En cinquante ans, on est passé d'un microprocesseur comprenant 2300 transistors à une puce qui en comprend plus de 16 milliards. La capacité de l'industrie électronique de concentrer de plus en plus de transistors sur de petites surfaces est une des raisons de son succès. En 1965, Gordon Moore, un des cofondateurs du fabricant de circuits électroniques intel, avait prédit que le nombre de composants que l'on peut intégrer dans un circuit électronique allait doubler chaque année durant la prochaine décennie. En 1975, il a revu ses prévisions et ramené cette croissance à un doublement tous les deux ans. Depuis, cette prévision est connue sous le nom de la loi de Moore. Sur base de la loi de Moore, on pourrait penser que l'industrie informatique continue son évolution sans difficulté depuis les début des années 1970s et qu'il en sera toujours de même. Ce n'est pas tout à fait correct. Il y a certaines contraintes technologiques qui ont un impact sur l'architecture des ordinateurs et l'évolution de leurs performances. L'analyse de cette évolution et des techniques qui permettent d'améliorer les performances des ordinateurs sort du cadre de ce cours introductif. Il y a cependant certains points sur lesquels il est important que vous soyez déjà conscientisés.

Le minuscule processeur utilise une horloge pour rythmer son fonctionnement. Toutes les instructions qu'il supporte doivent s'exécuter durant un cycle d'horloge, que ce soit l'instruction `M=A+M` qui nécessite une lecture en mémoire, une écriture en mémoire et une addition ou l'instruction `D=0` qui est nettement plus simple. Cette hypothèse facilite grandement la réalisation du minuscule ordinateur, mais les microprocesseurs réels ont des instructions qui ne prennent pas toutes le même temps. Certaines s'exécutent en un seul cycle d'horloge, comme une addition entre deux registres.

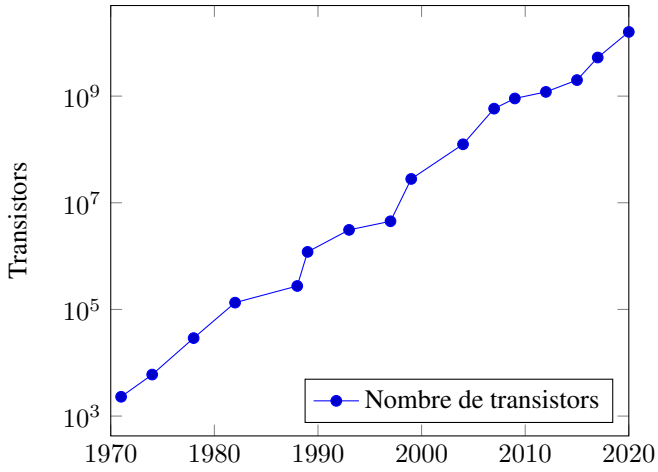



FIG. 15.1 – Evolution du nombre de transistors contenus dans les processeurs commerciaux

D'autres utilisent plusieurs cycles d'horloge voire des dizaines de cycles d'horloge comme des opérations de division ou de multiplication ou des opérations de calcul avec des réels représentés en virgule flottante.

La vitesse de l'horloge d'un ordinateur a souvent été présentée, notamment dans des actions de marketing, comme la métrique la plus importante au niveau des performances. De ce point de vue, il est intéressant de suivre l'évolution des microprocesseurs du fabricant intel qui publie de nombreuses données historiques sur son site web. La Fig. 15.2 présente l'évolution du cycle d'horloge des processeurs intel durant les cinq dernières décennies. Jusqu'aux environs

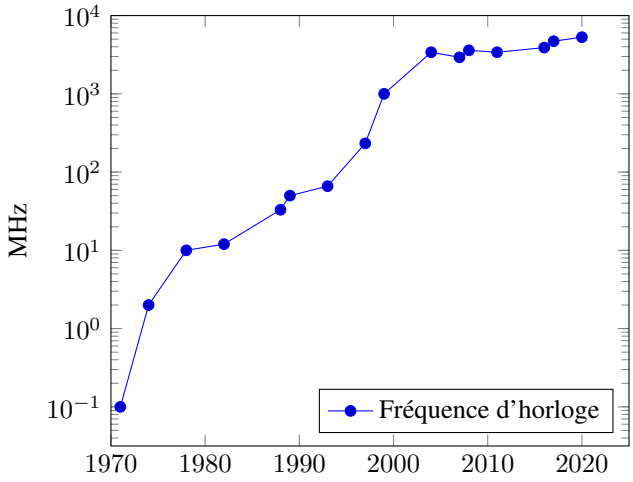



FIG. 15.2 – Evolution de la fréquence d'horloge des processeurs intel

de l'année 2000, la fréquence d'horloge des microprocesseurs a régulièrement augmenté. Les premiers processeurs fonctionnaient à des fréquences de quelques centaines de kHz. En 1978, le 8086 atteignait les 10 MHz. En 1999, l'intel Pentium atteignait 1 GHz. Depuis, la plupart des processeurs sont restés aux alentours de 2 à 5 GHz. Les contraintes technologiques font qu'il est difficile aujourd'hui de construire des microprocesseurs qui supportent des fréquences d'horloge supérieures à 4-5 GHz. Face à cette limitation technologique, les fabricants de processeurs ont dû trouver des solutions pour exécuter plus d'instructions sans augmenter la fréquence d'horloge des microprocesseurs.

Les deux principales technologies sont l'hyperthreading et l'utilisation de plusieurs coeurs sur un même processeur. L'hyperthreading a été introduit au début des années 2000. Cette technologie permet à un système d'exploitation d'exécuter deux programmes simultanément sur le même processeur. Ces deux programmes ont chacun accès à des registres qui leurs sont propres et leurs accès en mémoire sont entrelacés. La deuxième technique est d'installer sur

un processeur unique plusieurs coeurs, c'est-à-dire plusieurs unités de calcul qui sont chacune capables d'interagir avec la mémoire et d'exécuter des programmes. Chacun de ces coeurs dispose d'un ensemble de registres qui lui est propre. Il peut donc exécuter un programme différent. Il est aussi possible d'écrire les programmes de façon à ce que plusieurs parties de chaque programme puissent s'exécuter en parallèle sur le même coeur ou sur des coeurs différents. Cette technique de programmation sort du cadre de ce cours. Elle sera abordée en deuxième bachelier en utilisant les langages de programmation Java et C.

La plupart des microprocesseurs actuels utilisent plusieurs coeurs. En voici quelques exemples :

— l"intel core 2 duo, introduit en 2006, comprenait deux coeurs
— l"AMD K10, introduit en 2007, comprenait quatre coeurs
— l"intel Xeon 7400, introduit en 2008, était composé de six coeurs
— le Sparc T3, introduit en 2010, était composé de 16 coeurs
— l "intel Xeon Westmere, introduit en 2011, comprenait 10 coeurs
— l"intel Xeon Phi, introduit en 2012, comprend 61 coeurs
— le SPARC M7, introduit en 2015, comprend 32 coeurs
— le Qualcomm Snapdragon 850, qui équipe de nombreux smartphones, contient huit coeurs
— l"AMD Epyc supporte 32 coeurs
— l"Apple A14 Bionic, qui équipe les iPhones 12, contient six coeurs de calcul

A côté du microprocesseur principal, les ordinateurs actuels utilisent des microprocesseurs spécialisés pour certaines opérations. En termes de performances, les applications les plus demandeuses sont souvent les applications graphiques. Les premières cartes graphiques permettaient d'afficher des pixels individuels à l'écran comme nous l'avons fait avec le minuscule ordinateur. Au fil des années, les besoins ont augmenté et les cartes graphiques ont commencé à supporter des instructions qui permettent d'afficher des lignes, des caractères puis des objets 3-D etc. Aujourd'hui les cartes graphiques performantes sont équipées de GPU ou Graphics Processing Units. Un GPU peut être vu comme un petit ordinateur spécialisée dans les calculs nécessaires pour afficher des informations à l'écran. Ces GPUs contiennent des dizaines ou des centaines de coeurs qui supportent en langage d'assemblage spécialisé. Ils contiennent parfois autant de mémoire RAM que l'ordinateur dans lequel ils sont installés.

Si l'arrivée de l'hyperthreading et des processeurs multicoeurs a permis de continuer à augmenter les performances sans augmenter les fréquences d'horloge des microprocesseurs, il y a un autre problème auquel les fabricants de microprocesseurs doivent encore faire face. Un microprocesseur doit en permanence interagir avec la mémoire, pour charger les instructions à exécuter mais aussi pour lire et écrire les données qu'il manipule. Dans les années 1970s, le CPU était plus lent que les mémoires DRAM et celles-ci pouvaient fournir rapidement les instructions et données demandées par le CPU. Malheureusement, dans le courant des années 1980s, la tendance s'est inversée. La vitesse des processeurs s'est améliorée plus rapidement que les temps d'accès aux mémoires de type DRAM. La Fig. 15.3, basée sur des données de l'excellent livre Computer Systems: A Programmer's Perspective de Randal E. Bryant et David R. O'Hallaron décrit clairement ce problème. En 1985, il était encore possible de faire attendre le processeur pour accéder aux données de la DRAM sans trop affecter les performances, mais depuis le milieu des années 1990s, ce n'est plus envisageable. En 1995, le temps d'accès à la DRAM était de 70 nsec alors qu'un microprocesseur ne mettait que 6 nsec pour exécuter une instruction. Une première solution pour pallier à ce problème était de remplacer les mémoires DRAM par des SRAM. En effet, cette technologie a des temps d'accès qui sont nettement plus courts comme illustré sur la Fig. 15.4. Si les SRAMs sont satisfaisantes au niveau des temps d'accès, elles ont un inconvénient majeur : leur capacité limitée. Il est économiquement impossible de construire un ordinateur qui n'utiliserait que de la mémoire de type SRAM. La solution qui a été trouvée par l'industrie informatique pour résoudre ce problème a été l'introduction des mémoires caches. Une mémoire cache est une mémoire SRAM de faible capacité qui s'intercale entre le CPU et la mémoire DRAM. Une mémoire cache ne fonctionne pas comme une mémoire RAM. Une mémoire RAM est un peu comme un tableau dans un langage de programmation comme python. En python, on peut accéder à un élément de ce tableau en utilisant son index. Dans une mémoire RAM, on accède à une donnée en fournissant son adresse. Chaque zone de la mémoire est identifiée par une adresse unique et une mémoire RAM supporte autant d'adresses qu'il y a d'éléments qu'elle peut stocker en mémoire.

Une mémoire cache est une mémoire qui est dite associative. Une cache stocke des couples *adresse, donnée*. Elle fonctionne un peu comme un dictionnaire en langage python. Lorsqu'elle reçoit une adresse, elle parcourt rapidement l'ensemble des couples *adresse, donnée* qu'elle a mémorisé. Si l'adresse demandée s'y trouve, elle retourne la donnée qui y est associée au processeur et arrête de demander cette adresse à la mémoire RAM. Sinon, elle attend simplement

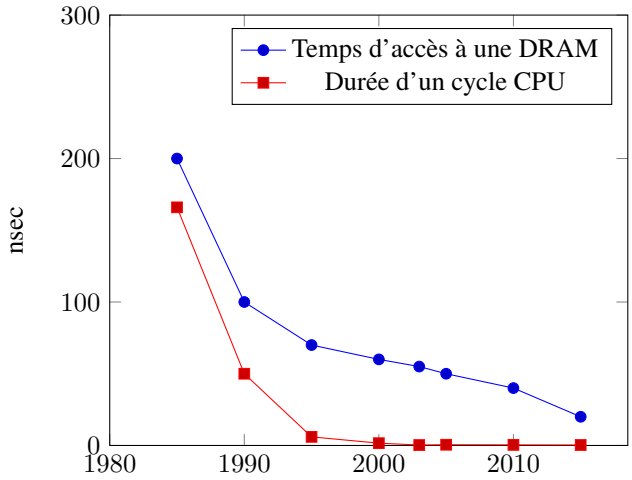



FIG. 15.3 – Au fil des années, le gap entre la durée d'un cycle CPU (en nsec) et un temps d'accès à la DRAM n'a fait qu'augmenter

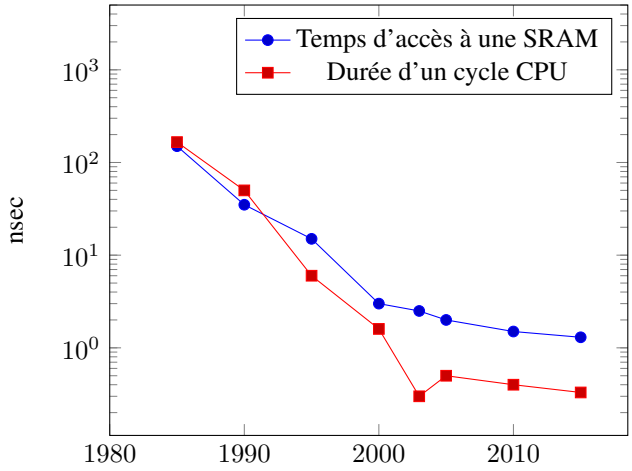



FIG. 15.4 – Gap entre la durée d'un cycle CPU (en nsec) et un temps d'accès à la SRAM

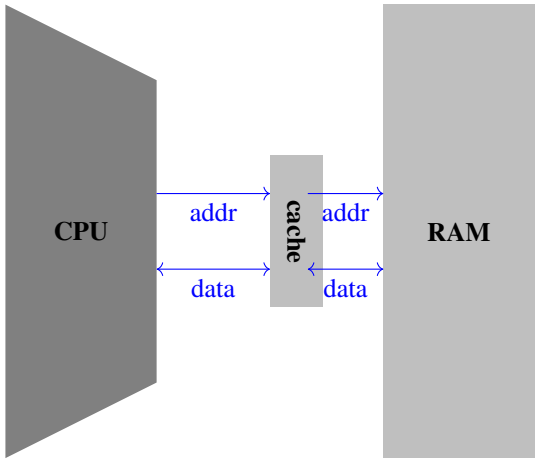



FIG. 15.5 – La cache s'interpose entre le CPU et la mémoire RAM

que la mémoire, plus lente, retourne la donnée demandée au processeur. Lorsque la mémoire RAM retourne la valeur demandée par le processeur, celle-ci passe par la mémoire cache qui en profite pour mémoriser ce nouveau couple *adresse,donnée*. Comme la capacité de la mémoire cache est limitée, il est possible qu'elle doivent supprimer un ancien couple pour avoir la place pour stocker le nouveau couple.

Une analyse détaillée du fonctionnement des mémoires cache sort du cadre de ce cours. La Fig. 15.6 présente l'évolution de la taille des mémoires cache sur les processeurs intel durant les trente dernières années. On est passé de quelques KBytes à quelques dizaines de MBytes, soit une capacité décuplée chaque décennie. Vu la différence au

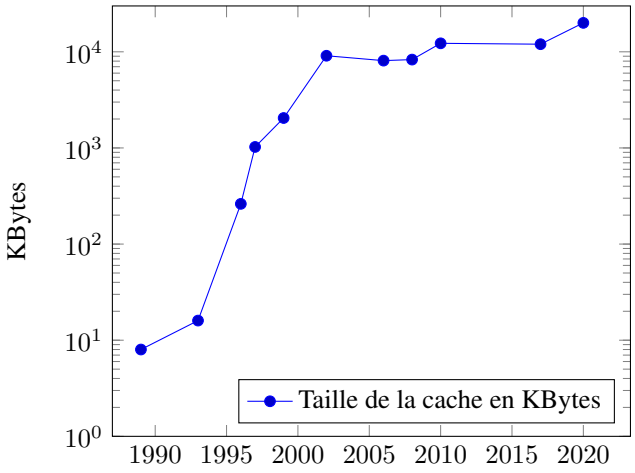



FIG. 15.6 – Evolution de la taille totale de la cache sur les processeurs intel

niveau des temps d'accès entre les mémoires caches et la DRAM, il peut être intéressant pour certains types de programmes qui échangent beaucoup de données avec la mémoire de traiter des blocs de données qui tiennent à l'intérieur de la cache. Vous aborderez ces techniques dans d'autres cours du bachelier et en master.

Pour terminer, notre discussion des ordinateurs actuels, il est intéressant d'analyser rapidement les dispositifs de stockage. Pour exécuter un programme, il faut d'abord le charger en mémoire RAM depuis un disque dur ou un lecteur SSD. Les données que le programme manipule sont aussi également stockées sur ce disque dur ou ce lecteur SSD. Sans entrer dans les détails du fonctionnement de ces dispositifs de stockage (ce sera l'objet du cours de systèmes informatiques), il est utile d'avoir en tête les performances de ces dispositifs. Un tel dispositif de stockage est conçu pour stocker des blocs de données qui sont généralement lus par le système d'exploitation. Les premiers disques durs datent de la fin des années 1950 avec l'"IBM 350 qui avait une capacité de 3.75 MBytes. Les disques durs actuels peuvent stocker plusieurs TBytes de données et il est possible de construire des armoires de stockage qui regroupent des centaines ou des milliers de tels disques durs. La capacité de ces disques durs n'a fait qu'augmenter au fil des années. Malheureusement, tout comme les DRAMs, les temps d'accès n'ont pas étés réduits aussi rapidement (voir Fig. 15.7 également extraite du livre Computer Systems: A Programmer's Perspective ). C'est lié à la technologie utilisée pour construire ces dispositifs de stockage.

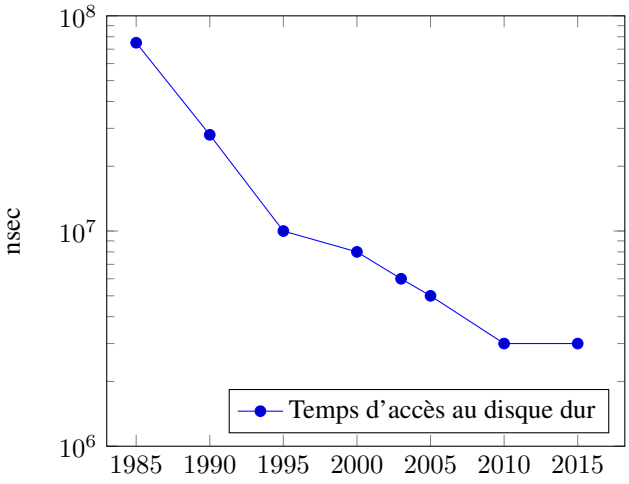



FIG. 15.7 – Evolution du temps d'accès aux disques durs

# CHAPITRE 16

# Systèmes de stockage de données

Le dernier composant d'un ordinateur que nous analysons sont les dispositifs permettant de stocker des données et des programmes. De nombreuses technologies ont étés utilisées au fil des années pour stocker de l'information et des programmes de façon à ce que ces informations puissent être réutilisées après un redémarrage de l'ordinateur ou sur un autre ordinateur. Contrairement à la mémoire RAM, ces systèmes peuvent conserver l'information stockée même lorsque l'ordinateur est déconnecté du réseau électrique. Ces systèmes utilisent des technologies très variées pour stocker l'information. Une description détaillée de toutes ces technologies sort du cadre de ce syllabus.

Les plus anciennes technologies sont les cartes perforées. Ces cartes sont antérieures aux premiers ordinateurs puisqu'elles étaient initialement utilisées sur les métiers à tisser, les orgues de barbarie ou les pianos mécaniques. C'est en utilisant de telles cartes perforées que Charles Babbage a pu construire sa machine analytique qui est un ancêtre des ordinateurs.

Une carte perforée est une feuille de papier rigide sur laquelle on encode l'information en y faisant des trous. Ces cartes perforées sont lues par des dispositifs mécaniques ou optiques qui détectent les différents trous. Les cartes perforées courantes avaient 80 colonnes et douze lignes. Ces cartes ont servi jusque dans les années 1970s et 1980s pour soumettre des travaux et des données à des ordinateurs dans des centres de calcul. De nos jours, les cartes perforées ne sont plus utilisées.

De nombreux systèmes de stockage utilisent les propriétés magnétiques ou optiques de la matière pour stocker de l'information. Parmi les systèmes utilisant les propriétés magnétiques, on peut citer :

— les cassettes
— les lecteurs de bandes
— les disques souples (floppy disks)
— les disques durs (hard disks)

Parmi les systèmes utilisant les propriétés optiques de la matière, on peut noter :

— les CDs
— les DVDs

D'autres systèmes plus récents utilisent des dispositifs électroniques pour stocker de l'information. Parmi ces systèmes, on peut citer :

— les cartes mémoires
— les lecteurs SSD
— les mémoires non-volatiles (NVM)

Les informaticiens et informaticiennes interagissent rarement directement avec tous ces dispositifs de stockage. La plupart des programmes s'exécutent dans le cadre d'un système d'exploitation comme Linux, Windows, MacOS, …

Ce système d'exploitation est un logiciel spécialisé qui gère l'ensemble de ressources matérielles d'un ordinateur et offre aux programmes applicatifs des abstractions qui leur permettent d'utiliser de la même façon un disque SSD et un CD, même si ces deux dispositifs utilisent des technologies très différentes. Pour cela, les informaticiens et informaticiennes qui ont conçu ces systèmes d'exploitation ont développé des abstractions qui facilitent les opérations de lecture et d'écriture sur des dispositifs de stockage. Vous avez l'habitude d'utiliser des fichiers pour stocker vos données et programmes. Ces fichiers et programmes sont généralement organisés dans des répertoires pour faciliter leur accès et regrouper logiquement les fichiers qui font partie d'un même projet ou d'un même cours. Ces fichiers et ces répertoires sont des abstractions qui ont été introduites par les informaticiens et informaticiennes qui ont construit ces systèmes d'exploitation.

Les répertoires et fichiers sont regroupés dans ce que l'on appelle en informatique un système de fichiers. Un système de fichiers est un ensemble de répertoires et de fichiers qui sont stockés sur un ou plusieurs dispositifs de stockage. Un répertoire est un ensemble contenant zéro, un ou plusieurs autres répertoires, zéro, un ou plusieurs autres fichiers. Dans un système de fichiers, les répertoires sont organisés sous la forme d'un arbre. Cet arbre commence par un répertoire spécial appelé la racine. Cette racine contient un ou plusieurs autres répertoires et parfois des fichiers. Chacun de ces répertoires, et tous les autres répertoires du système de fichiers, a un répertoire parent. Le seul répertoire qui n'a pas de parent est le répertoire qui se trouve à la racine du système de fichiers et s'appelle le répertoire racine.

Un fichier est simplement une séquence d'octets qui est identifié par un nom. Un fichier contient généralement un nombre positif d'octets, mais il est possible d'avoir des fichiers vides qui ne contiennent aucun octet. Un système d'exploitation associé généralement à un fichier des métadonnées comme sa date de création ou de dernière modification, des permissions, des informations sur le propriétaire, ... Les métadonnées associées à un fichier peuvent varier d'un système d'exploitation à l'autre.

Les systèmes d'exploitation permettent aux logiciels d'accéder aux données qui sont stockées dans les fichiers à l'intérieur des répertoires. Chaque système d'exploitation contient des fonctions qui permettent aux applications d'accéder simplement au contenu des fichiers. A titre d'exemple, python fournit différentes fonctions qui permettent de manipuler les fichiers et répertoires. Avant de pouvoir utiliser un fichier, un programme doit l'ouvrir en utilisant la fonction `open()`. Lors de l'appel à cette fonction, le système d'exploitation vérifie notamment si le fichier existe et si le programme dispose des permissions nécessaires pour accéder au fichier. Les fonctions les plus connues pour accéder à un tel fichier sont `read()` et `write()`. Ces deux fonctions accèdent au fichier de façon linéaire. Lorsque un programme ouvre un fichier, le système d'exploitation associe à ce fichier ouvert une « tête de lecture ». Cette « tête de lecture » est un entier qui indique la position du fichier à laquelle la prochaine opération `read()` et `write()` sera réalisée. A l'ouverture du fichier, la « tête de lecture » indique le début du fichier (c'est-à-dire la position `0`). Après avoir lu `n` octets, cette « tête de lecture » vaut `n` et référence donc le n+1 ième octet du fichier. Si le programme appelle la fonction `write()` à ce moment, les données seront écrites à partir du n+1 ième octet. Outre les fonctions `read()` et `write()`, python supporte des fonctions qui permettent de modifier directement la tête de lecture associée à un fichier ouvert. C'est notamment le cas de la fonction `seek()` qui permet de déplacer la « tête de lecture » de façon absolue ou relative. Pour la lecture de fichiers contenant du texte, la fonction `seek()` est peu utilisée. Par contre, `seek()` est très importante pour des logiciels qui doivent manipuler des fichiers contenant des images, des sons ou même des bases de données. Un système de fichier doit pouvoir fournir un support efficace aux applications qui accèdent à des fichiers de façon séquentielle en utilisant `read()`/`write()` mais aussi d'autres applications qui combinent ces deux fonctions avec `seek()` pour un accès direct aux fichiers. En python, le module `os` fournit différentes fonctions qui permettent d'accéder directement aux répertoires comme `os.scandir()` par exemple.

Les systèmes d'exploitation utilisent des abstractions qui représentent les caractéristiques des principaux dispositifs de stockage afin de pouvoir utiliser le même code pour manipuler de très nombreux dispositifs. Le logiciel est alors organisés en couches. La couche supérieure expose des structures de données et des fonctions qui correspondent à cette abstraction. L'implémentation de cette couche peut varier d'un dispositif à l'autre, mais l'avantage est que le code de gestion des fichiers et répertoire est le même quel que soit le dispositif de stockage. Nous utiliserons une telle abstraction dans le cadre de ce cours. Les dispositifs de stockage permettent d'écrire ou de lire des blocs d'octets de taille fixe. Un dispositif de stockage contient un nombre fixe de blocs et chaque bloc est identifié par un numéro unique. On pourrait représenter l'utilisation d'un dispositif de stockage par deux fonctions :

```
def read_block(pos):
```

(suite sur la page suivante)

(suite de la page précédente)

```
  # Lit le contenu du bloc numéro pos et retourne le bloc complet

def write_block(bloc, pos):
  # Remplace le bloc numéro pos avec le bloc passé en argument
```

Les dispositifs de stockage utilisent différentes tailles de blocs. Plus la capacité de stockage est important, plus la taille des blocs est élevée. Sur un disque souple, les blocs de 512 octets étaient courants. Sur disque dur, les blocs de 4.096 octets sont très souvent utilisés. Dans le cadre de ce cours, nous représenteront un dispositif de stockage sous la forme d'une grille. Chaque cellule de la grille correspond à un bloc. Notre modèle de dispositif de stockage est illustré dans la figure ci-dessous. Chaque cellule correspond à un bloc et il contient le numéro du bloc. Dans un dispositif de stockage

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |
| 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |

FIG. 16.1 – Modèle d'un dispositif de stockage

réel, seuls certains blocs sont utilisés pour stocker les fichiers et les répertoires. Les blocs du début du système de stockage sont généralement utilisés pour stocker des paramètres du système de fichier. Dans les exemples qui suivent, nous réserverons les dix premiers blocs à cette utilisation. En pratique, le nombre de ces blocs « spéciaux » varie d'un système de fichiers à l'autre. Généralement, le ou les premiers blocs d'un système de stockage sont utilisés pour placer le programme d'amorçage, c'est-à-dire le code qui est chargé automatiquement au lancement de l'ordinateur et qui charge généralement le système de stockage. Dans nos exemples, nous réservons les deux premiers blocs du système de stockage à cette utilisation. Ils seront représentés en rouge dans les figures.

Les systèmes de fichiers sont organisés sous la forme d'un arborescence comme illustré en Fig. 16.2. Les rectangles représentent des fichiers tandis que les rectangles arrondis sont les répertoires. Le premier répertoire est considéré comme la racine du système de fichiers. Il contient parfois quelques fichiers, mais souvent des répertoires. Chacun de ces répertoires peut lui-même contenir des fichiers et/ou des répertoires. Dans le cadre de ce chapitre, nous utiliserons une arborescence simple pour illustrer les différentes notions. Cette arborescence est reprise dans la figure ci-dessous avec pour chaque fichier l'indication de sa taille en octets. Ce système de fichiers contient un fichier (`f1`) et trois répertoires (`bin`, `tmp` et `lib`) dans le répertoire racine. Le sous-répertoire `bin` contient deux fichiers (`sh` et `echo`). Le sous-répertoire `lib` ne contient ni fichier si sous-répertoire. Le sous-répertoire `tmp` contient un fichier (`file`) et un répertoire (`rep`). Le sous-répertoire `rep` contient un sous-répertoire nommé `sub` qui contient lui-même un fichier nommé `file`. Dans un système de fichiers, il est possible (et même fréquent) d'avoir différents fichiers et/ou répertoires qui ont le même nom mais se trouvent à des endroits différents de l'arborescence. C'est possible car pour le système de fichiers, le nom d'un fichier ou d'un répertoire est toujours son *nom complet* depuis la racine.

Code source 16.1 – Noms complets des fichiers du système de fichiers d'exemple

```
/  (répertoire racine)
/bin (répertoire)
/bin/sh (fichier, 2400 octets)
/bin/echo (fichier, 600 octets)
/tmp (répertoire)
/tmp/file (fichier, 0 octet)
/tmp/rep (répertoire)
/tmp/rep/sub (répertoire)
/tmp/rep/sub/file (fichier, 123 octets)
/lib (répertoire)
/f1 (fichier, 600 octets)
```

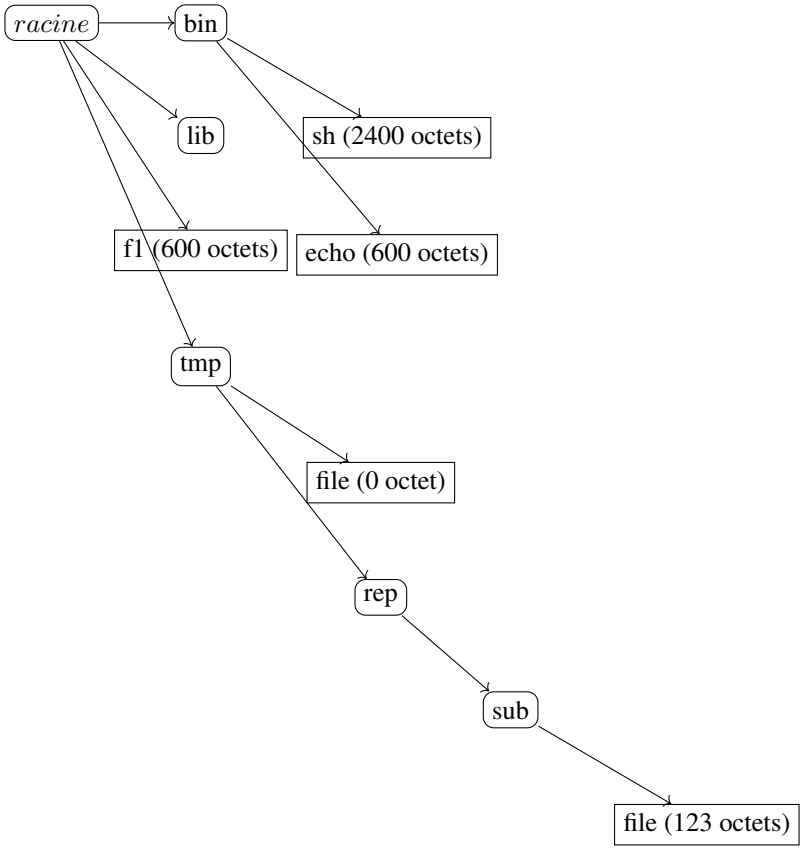



FIG. 16.2 – Arborescence des répertoires et fichiers

Dans l'exemple ci-dessus, nous avons utilisé le caractère `/` pour séparer les noms de sous-répertoire et de fichiers. C'est la convention qui est utilisée par les systèmes de fichiers dérivés du systèmes d'exploitation Unix. D'autres systèmes d'exploitation utilisent d'autres conventions. Par exemple, les systèmes d'exploitation produits par Microsoft utilisent plutôt le caractère `\`. La convention utilisée importe peu, par contre elle implique que le caractère qui est utilisé comme séparateur ne peut pas se trouver à l'intérieur d'un nom de répertoire ou de fichier.

Il existe un très grand nombre de systèmes de fichiers. Une page wikipedia dédiée aux systèmes de fichiers en liste plusieurs dizaines. Nous nous concentrerons sur deux exemples importants :

— le système de fichiers *FAT* utilisé par le système d'exploitation MSDOS
— le système de fichiers *étendu* utilisé par le système d'exploitation Linux

Ces deux systèmes de fichiers ont une structure assez différente et ils ont tous les deux étés largement déployés. L'objectif principal de ces deux systèmes de fichiers est de stocker les fichiers, les répertoires mais aussi l'arborescence qui représente le système de fichiers sur un dispositif de stockage composé de blocs.

La partie la plus simple dans le design d'un système de fichiers est le stockage des fichiers. Dans notre exemple, nous avons cinq fichiers à stocker :

— /bin/sh (fichier, 2400 octets)
— /bin/echo (fichier, 600 octets)
— /tmp/file (fichier, 0 octet)
— /tmp/rep/sub/file (fichier, 123 octets)
— /f1 (fichier, 600 octets)

Si l'on suppose que le système de fichiers utilise des blocs de 512 octets, le premier fichier, `/bin/sh` occupera 5 blocs, `/bin/echo` en occupera 2, `/tmp/file` n'en occupera aucun, `/tmp/rep/sub/file` utiliser 1 bloc et enfin les données de `/f1` nécessiteront deux blocs sur le système de fichiers. Pour simplifier la comparaison des deux systèmes de fichiers, nous supposerons que les dix blocs qui contiennent les données de ces fichiers seront stockés dans les blocs numérotés de 30 à 39. La Fig. 16.3 décrit une organisation possible des blocs contenant les données de ces fichiers. Chaque fichier est composé de blocs qui sont contigus, mais cette contrainte n'est imposée que sur les dispositifs de stockage tels que les CD-ROMs et DVDs où les données sont écrites une seule fois. Dans un système de stockage où des fichiers sont créés, modifiés et supprimés de façon dynamique, les blocs qui composent un fichier ne sont pas nécessairement contigus. Sur base de ces exemples, on remarque aisément qu'un fichier correspondant à une

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |
| /b | i | n | /s | h | /bin/e | cho | /t...le | f | l |

FIG. 16.3 – Blocs contenant les données des fichiers

séquence d'octets est en fait stocké sur le dispositif de stockage comme une liste de blocs. Ainsi, dans l'exemple de la Fig. 16.4, le fichier `/bin/sh` est composé du bloc `39` suivi du bloc `35` puis du bloc `36` puis du bloc `33` et enfin du bloc `38`. Le fichier `/bin/echo` correspond lui au bloc `31` suivi du bloc `32`. Le fichier `/f1` débute lui par le bloc `34` et se termine par le bloc `30`. La liste correspondant à chaque fichier évolue à chaque fois que l'on modifie les fichiers, notamment lors des créations, suppressions et des ajouts de données dans des fichiers. La figure Fig. 16.5 représente graphiquement les listes chaînées qui correspondent à ces différents fichiers. Le système de fichiers doit stocker les listes ordonnées correspondant à chacun de ces fichiers. Ces listes doivent se trouver sur le dispositif de stockage car elles doivent pouvoir être lues par l'ordinateur. Une première possibilité serait d'ajouter à l'intérieur de chaque bloc le numéro de son successeur dans la liste et `-1` en fin de fichier. Pour cela, il faudrait réserver une partie du bloc pour stocker le numéro du bloc suivant. Si les blocs ont une capacité de 512 octets, on pourrait par exemple réserver 4 octets

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |
| 1 | /bin/e | cho | /s | f | i | n | /t...le | h | /b |

FIG. 16.4 – Autre organisation possible des blocs contenant les données des fichiers

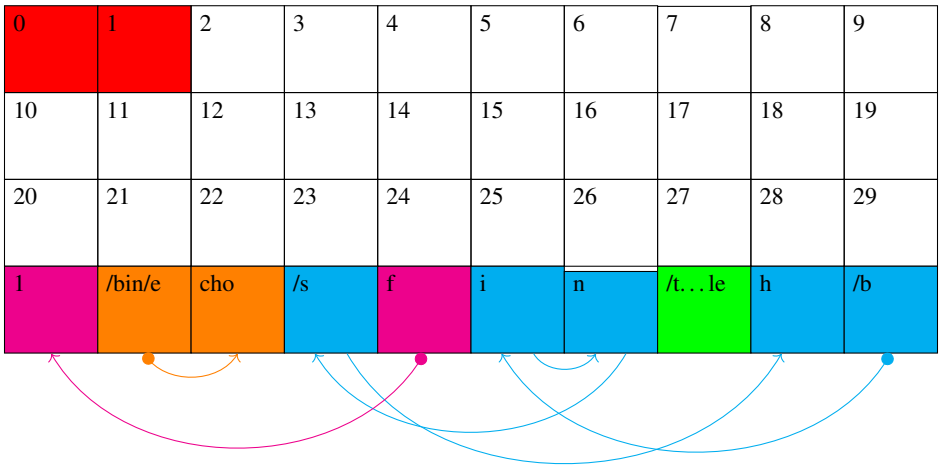



FIG. 16.5 – Liste chaînée et autre organisation des blocs contenant les données des fichiers

pour encoder le numéro du bloc suivant. Une telle organisation est représentée dans Fig. 16.6. Malheureusement, cette

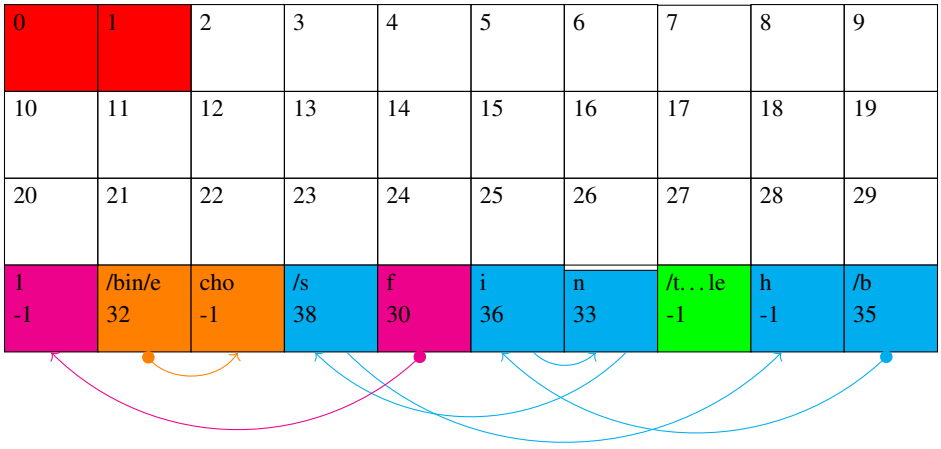



FIG. 16.6 – Ajout d'une référence vers le bloc suivant dans chaque bloc

approche souffre de plusieurs problèmes. Tout d'abord, elle réduit la taille des blocs. Un bloc contient $k$ octets de moins où $k$ est le nombre d'octets nécessaire pour encoder un numéro de blocs (typiquement 4 pour un disque dur). De plus, en stockant la liste chaînée directement dans les blocs, on force le système de fichiers à lire tous les blocs qui composent un fichier pour pouvoir y faire des accès directs. Considérons un fichier qui est édité par l'utilisateur ou créé par un programme. Lorsque la taille de ce fichier grandit, le système de fichiers doit allouer un nouveau bloc au fichier et l'ajouter à la liste chaînée. Si le fichier rétrécit suite à une opération d'édition, le système de fichiers va devoir « remonter » la liste chaînée pour retrouver les blocs à retirer. Cela peut nécessiter de reparcourir tout le fichier depuis son premier bloc ce qui sera coûteux en nombre d'accès au dispositif de stockage. Sur des dispositifs lents comme des disques souples ou durs, cela peut avoir un impact très important au niveau des performances. Pour ces raisons, les systèmes de fichiers ont choisi d'autres astuces pour stocker les listes chaînées qui représentent les différents fichiers et répertoires.

## 16.1 La table d'allocation des fichiers

La table d'allocation des fichiers (File Allocation Table - FAT en anglais) est une table qui permet de stocker de façon compacte toutes les listes chaînées qui représentent tous les fichiers (et répertoires) du système de fichiers. Cette table d'allocation représente de façon compacte l'ensemble des listes chaînées qui correspondent à nos quatre fichiers. La Fig. 16.7 représente à la fois l'index des lignes de la table et le numéro du bloc stocké à chaque ligne. Cette table indique pour le bloc d'index `i` le numéro du bloc qui le suit dans le fichier auquel il appartient. La valeur de `-1` est réservée pour indiquer qu'un bloc est le dernier bloc d'un fichier. On pourrait aussi réserver d'autres valeurs comme par exemple `-2` pour indiquer que le bloc n'est actuellement pas utilisé pour stocker des données ou un répertoire ou `-3` pour indiquer que le bloc a été marqué comme invalide par le dispositif de stockage et qu'il ne doit donc jamais être utilisé.

Dans la Fig. 16.7, le fichier `/f1` commence par le bloc `34` et se termine au bloc `30`. Le fichier `/bin/echo` commence lui au bloc `31` et se termine au bloc `32`. Le fichier `/tmp/rep/sub/file` n'utilise que le bloc `37`. Enfin, le fichier `/bin/sh` commence au bloc `39` et utilise ensuite les blocs `35`, `36` et `33` pour se terminer au bloc `38`.

La table d'allocation contient toutes les informations concernant les fichiers. Elle est critique pour pouvoir accéder aux données stockées sur le disque. Elle est stockée directement sur le dispositif de stockage dans une suite de blocs qui sont contigus. Le nombre de blocs nécessaire pour stocker la table d'allocation des fichiers dépend de deux paramètres : la taille du dispositif de stockage (en blocs) et le nombre d'octets utilisés pour encoder les numéros de blocs. La première version du système d'exploitation MS-DOS de Microsoft utilisait 1.5 octets (12 bits) pour identifier chaque

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |
| 1 | /bin/e | cho | /s | f | i | n | /t. . . le | h | /b |

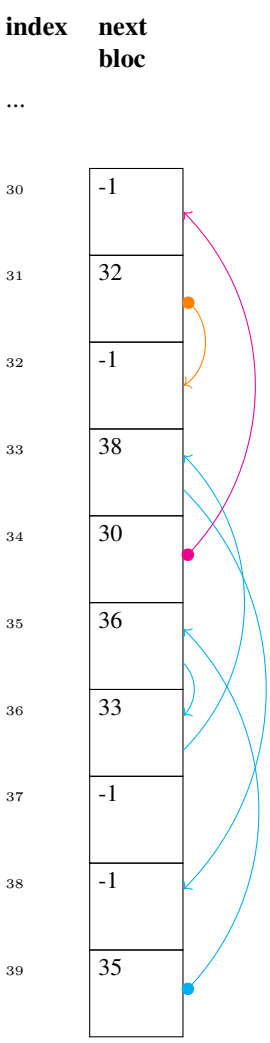



FIG. 16.7 – Les listes chaînées correspondant à des fichiers peuvent être stockées dans une table d'allocation

bloc. Les dernières versions du système de fichier FAT utilisaient 4 octets (32 bits) pour encoder les numéros de blocs. Dans un bloc de 512 octets, on pouvait donc stocker une table d'allocation des fichiers pour un dispositif comprenant 128 blocs. Dans nos exemples, nous supposerons que l'on utilise cette version du système de fichiers FAT et que la table d'allocation contient 1024 entrées. Si chaque bloc a une capacité de 512 octets, cette table d'allocation peut donc gérer des dispositifs de stockage d'une capacité de 512 Ko. La table d'allocation nécessite 8 blocs consécutifs. Elle est généralement placée dans des blocs dont le numéro est bien connu. Dans notre système de fichier d'exemple, nous avons choisi de la placer dans les blocs `2` à `9`.

La table d'allocation est consultée par le système d'exploitation lors de chaque accès à un fichier. Si chaque accès à un fichier devait nécessiter aussi des accès aux blocs qui contiennent la table d'allocation, les performances seraient fortement réduites. En pratique, les systèmes d'exploitation qui utilisent une table d'allocation des fichiers chargent en mémoire une copie de la table d'allocation des fichiers au démarrage. Tous les accès en lecture à la table d'allocation se font directement en mémoire qui est nettement plus rapide que le dispositif de stockage. Par contre, lors des opérations qui modifient la table d'allocation des fichiers, comme la création ou l'édition d'un fichier, cette modification est d'abord effectuée en mémoire et ensuite copiée sur le dispositif de stockage. Cette écriture sur le dispositif de stockage est nécessaire pour pouvoir récupérer le dernier état du système de fichiers si l'ordinateur s'arrêtait à cet instant en raison par exemple d'une panne d'électricité.

---

**Note :** La table d'allocation ne peut pas être perdue

La table d'allocation contient toutes les informations concernant les fichiers. Une erreur de lecture dans un des blocs contenant la table d'allocation rendrait inaccessible tous les fichiers affectés par cette erreur. Pour cette raison, les systèmes de fichiers qui utilisent une table d'allocation maintiennent généralement deux copies de cette table de façon à pouvoir parer à toute erreur de lecture. Cela complexifie le système de fichiers puisqu'il faut s'assurer de toujours stocker les mêmes informations dans les deux copies de la table, mais c'est une assurance très utile pour éviter de perdre des données.

---

Grâce à la table d'allocation des fichiers, nous pouvons stocker sur un dispositif de stockage des fichiers de taille quelconque, mais nous n'avons pas pas encore défini comment stocker les répertoires. Dans un système de fichiers, un répertoire est simplement un fichier ayant un format particulier. Un répertoire peut être vu comme un tableau qui contient plusieurs entrées. Chaque entrée a une structure particulière. A titre d'exemple, considérons le format des répertoires utilisés par MS-DOS version 2.0. Ce système de fichiers utilisait des entrées de répertoires qui sont encodées sur 32 octets. 16 de ces entrées pouvaient donc se trouver dans un bloc de 512 octets. La Fig. 16.8 présente le format de cette entrée de répertoire. Chaque entrée de répertoire comprend plusieurs informations. Les 8 premiers

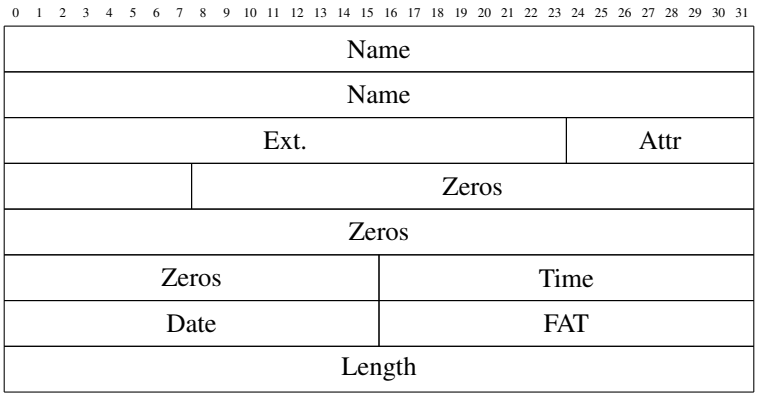



FIG. 16.8 – Une entrée d'un répertoire MS-DOS

caractères contiennent le nom du fichier. Ils sont suivis de trois caractères qui contiennent l'extension du nom de fichier. Sous MS-DOS, les trois caractères de l'extension sont utilisés pour indiquer le type de fichier. Ainsi, l'extension `.EXE` correspond à un programme exécutable, `.BAT` un script batch, `.DOC` un document pour le traitement de textes Word.

Il faut noter que le caractère `.` qui sépare le nom de l'extension ne se trouve pas explicitement dans l'entrée d'un répertoire. L'octet *Attr* (Attribut) contient des bits qui indiquent si l'entrée correspond à un répertoire ou un fichier, si il est accessible en lecture ou en écriture, ... Les champs *Time* et *Date* indiquent de quand date le fichier. Les deux derniers champs d'un répertoire sont très importants. Le premier, `FAT` indique le numéro du premier bloc contenant le fichier. Sur base de ce numéro de bloc, le système d'exploitation peut utiliser la table d'allocation des fichiers pour accéder aux autres blocs qui le composent. Le dernier champ est la longueur totale du fichier en octets. Ce champ est nécessaire pour indiquer où le fichier se termine dans le dernier bloc stocké sur le disque comme les fichiers ont rarement une longueur qui est un multiple de la taille des blocs.

Nous pouvons maintenant construire le répertoire racine de notre système de fichiers et voir comment il aurait été stocké en MS-DOS. Celui-ci contient les fichiers et répertoires suivants :

```
/bin (répertoire)
/tmp (répertoire)
/lib (répertoire)
/f1 (fichier, 600 octets)
```

Nous devons donc stocker 4 entrées dans ce répertoire. Les trois premières correspondent aux répertoires `/bin`, `/tmp` et `/lib`. La dernière correspond au fichier `/f1`. L'entrée du répertoire correspondant au fichier `f1` aura comme champ `FAT` le numéro de bloc `34`. Nous devons aussi stocker sur le dispositif de stockage le répertoire racine et les trois sous-répertoires `/bin`, `/tmp` et `/lib`. Chaque répertoire est un fichier contenant des entrées de répertoire. Le répertoire racine est un peu particulier car il doit pouvoir être lu au démarrage du système de fichiers. Il est placé par convention dans un bloc dont le numéro est bien connu. Dans notre système de fichiers d'exemple, nous supposons que le répertoire racine commence toujours au bloc `10`. Certains systèmes de fichiers de type `FAT` utilisent un répertoire racine de taille fixe, ce qui limite le nombre de fichiers dans le répertoire racine. D'autres systèmes de fichiers supportent un répertoire racine de taille variable.

Chacun des sous-répertoires du répertoire racine est stocké sous la forme d'un fichier spécial contenant des entrées de répertoire. Le répertoire racine contient donc les champs suivants :

- Name : `bin`, Attr : répertoire, FAT : `11`, Length : 64
- Name : `tmp`, Attr : répertoire, FAT : `12`, Length : 64
- Name : `lib`, Attr : répertoire, FAT : `13`, Length : 0
- Name : `f1`, Attr : fichier, FAT : `34`, Length : 600

Le répertoire `/bin` ne contient que deux fichiers. Il contient les champs suivants :

- Name : `sh`, Attr : fichier, FAT : `39`, Length : 2400
- Name : `echo`, Attr : fichier, FAT : `31`, Length : 600

Le répertoire `lib` est vide. Le répertoire `tmp` contient un fichier et un sous-répertoire, c'est-à-dire :

- Name : `file`, Attr : fichier, FAT : 0, Length : 0
- Name : `rep`, Attr : répertoire, FAT : `14`, Length : 0

Le sous-répertoire `rep` contient le sous-répertoire `sub` :

- Name : `sub`, Attr : répertoire, FAT : `15`, Length : 32

Le dernier répertoire est le répertoire `sub` qui contient le fichier `file` de 123 octets :

- Name : `file`, Attr : fichier, FAT : `37`, Length : 123

Nos répertoires sont donc stockés dans les blocs `10` à `15`. En pratique, il est fréquent qu'une zone du système de stockage soit réservée aux blocs qui contiennent les répertoires. Cela permet notamment de précharger certains répertoires en mémoire afin d'accélérer les accès aux fichiers ultérieurement. Cela facilite aussi la récupération d'erreurs lorsque l'ordinateur est arrêté avant que le système d'exploitation n'aie pu écrire les dernières modifications au système de fichiers sur le dispositif de stockage.

Grâce aux champs `FAT` qui se trouvent dans les répertoires, nous avons construit une arborescence de fichiers et de répertoire. Il y a quatre branches qui partent de la racine. Les trois premières correspondent aux répertoires `bin`, `tmp` et `lib`. La quatrième branche est celle du fichier `f1`. Cette arborescence est illustrée dans la Fig. 16.8 avec le numéro du premier bloc de chaque fichier ou répertoire.

---

**Note :** Vérification d'un système de fichiers corrompu

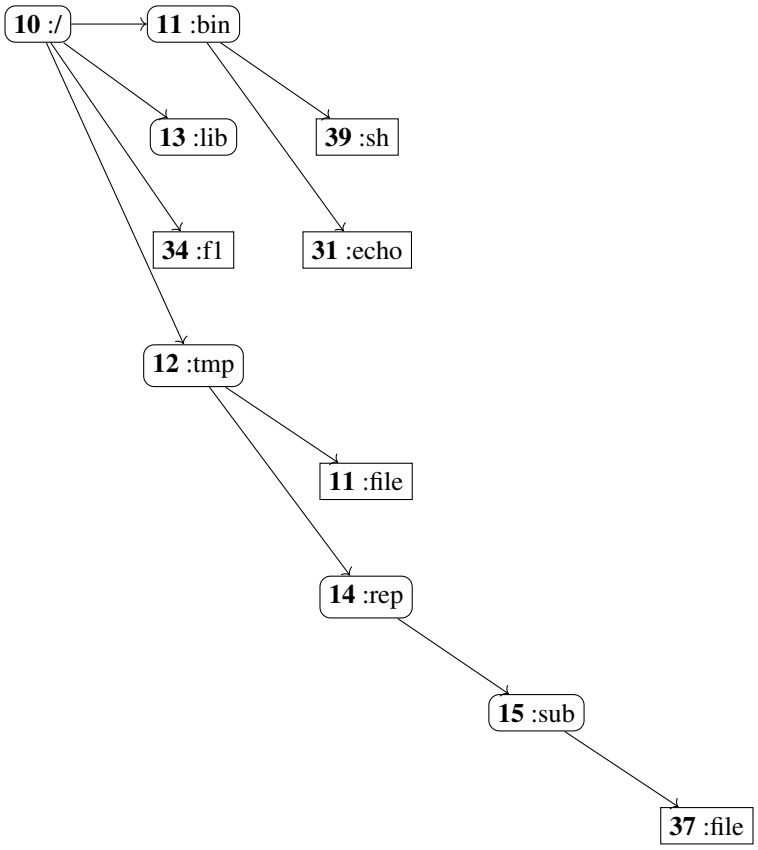



FIG. 16.9 – Arborescence des répertoires et fichiers

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |
| 1 | /bin/e | cho | /s | f | i | n | /t. . . le | h | /b |

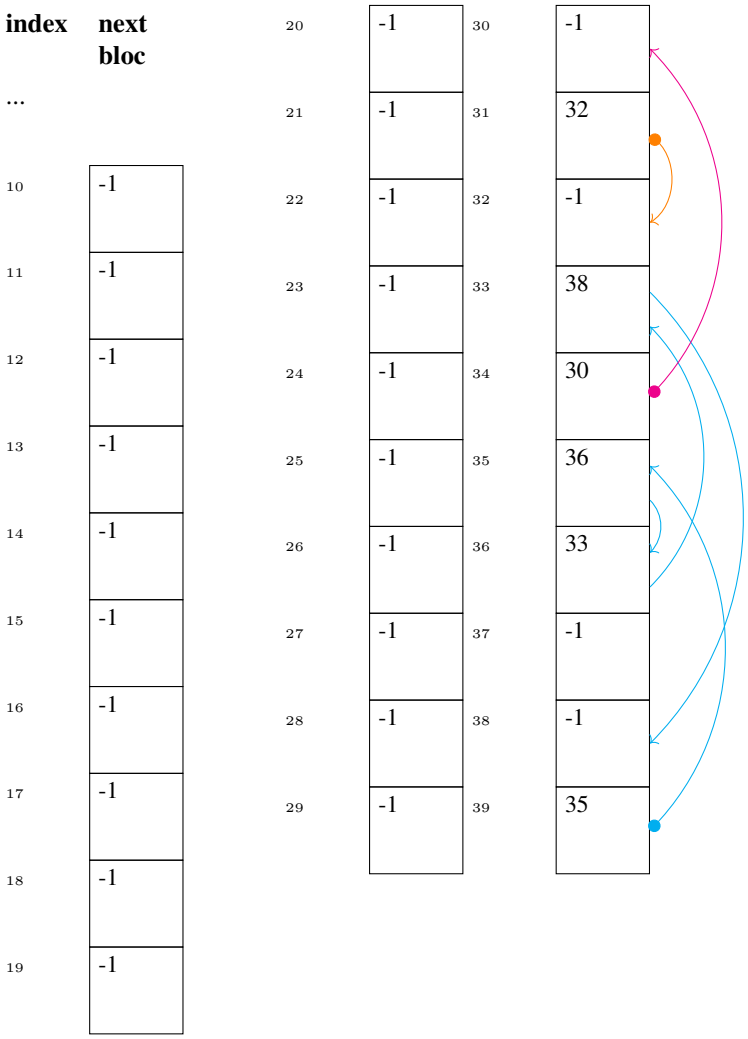



FIG. 16.10 – Le système de fichiers complet et sa table d'allocation des fichiers. La FAT occupe les blocs 2 à 9. Les répertoires sont dans les blocs 10 à 15. Les fichiers occupent les blocs 30 à 39.

Durant son utilisation, un système de fichiers doit pouvoir gérer les lectures et écritures classiques de fichiers et de répertoires et toutes les modifications de blocs qui y sont associées. Il doit aussi pouvoir faire face à des événements inattendus tout en préservant le plus possible les données se trouvant sur le dispositif de stockage. Pour un système de fichiers utilisant une table d'allocation des fichiers, les problèmes les plus pénalisants sont les erreurs de lecture qui rendent un bloc inutilisable et les pannes de courant.

## 16.2 Les inodes

La table d'allocation est une des techniques qui permet de construire un système de fichiers, mais c'est loin d'être la seule. Depuis les années 1970s les systèmes d'exploitation dérivés de Unix (dont Linux, MacOS et Android notamment) utilisent une technique basée sur les *inodes*. L'origine du nom *inode* n'est pas clairement établie. Il semble que ce sera la contraction des mots *index* et *node* (noeud en français). L'idée intuitive de l'inode est d'y stocker la liste des numéros des blocs qui composent un fichier. On pourrait intuitivement voir l'inode comme une sorte de table des matières des blocs qui composent un fichier.

Un *inode* est une zone du dispositif de stockage qui contient la liste des blocs qui composent un fichier. A titre d'exemple, considérons le même système de fichiers que celui de la section précédente et supposons que les fichiers sont stockés dans les mêmes blocs. Le fichier `/bin/sh` est composé des blocs `39`, `35`, `36`, `38` et `33`. Ces cinq blocs sont listés dans cet ordre dans l'inode correspondant à ce fichier. Grâce à cet *inode*, on connaît la position exacte de tous les blocs qui composent le fichier. Il en va de même pour les fichiers `f1`, `/bin/echo` et `/tmp/rep/sub/file` comme illustré dans la Fig. 16.11. A titre d'illustration sur l'utilisation des inodes, considérons le système de fichiers ext2 utilisé dans les premières versions du système d'exploitation Linux. Ce système de fichiers est inspiré des systèmes de fichiers Unix.

Un système de fichiers *ext2* est composé d'une suite de blocs. Les premiers blocs sont des blocs de contrôle qui contiennent de l'information sur le système de fichiers et sa structure. La plupart des blocs sont les blocs qui contiennent les données relatives aux fichiers et aux répertoires. Les premiers blocs d'un système de fichiers *ext2* contiennent les paramètres principaux du système de fichiers comme la taille des blocs, le nombre d'inodes, ... Une description détaillée du contenu du *Super Block* et des *FS descriptors* sort du cadre de ce cours. Après ces blocs de contrôle, un système de fichiers *ext2* contient deux bitmaps : le bitmap des block et le bitmap des inodes. Ensuite on retrouve la table des inodes. Cette table contient tous les inodes du système de fichiers. Tous les blocs de contrôle sont initialisés lors de la réaction (le formatage) du système de fichiers. La taille des bitmaps, des blocs et de la table des inodes sont fixées à ce moment. Ces tailles ne changeront jamais durant la vie du système de fichiers. Tout comme pour le système de fichiers utilisant une table d'allocation, commençons par analyser comment les répertoires sont encodés. Nous verrons ensuite plus en détails le contenu des inodes et terminerons par le rôle des deux bitmaps.

Dans le système de fichiers *ext2*, les entrées d'un répertoire ont une longueur variable. Chaque entrée d'un répertoire comprend quatre informations :

— le numéro de l'inode correspondant au fichier/répertoire sur 32 bits
— un entier sur 16 bits indiquant la longueur en octets de cette entrée du répertoire
— un entier sur 16 bits indiquant la longueur du nom de fichier/répertoire
— une chaîne de caractère contenant le nom du fichier/répertoire

Cette organisation des répertoires permet de supporter les fichiers et répertoires dont le nom contient un nombre quelconque de caractères. Contrairement au système de fichiers MS-DOS le répertoire ne contient pas d'attributs ni d'information sur la longueur des fichiers/répertoires. Dans le système de fichiers *ext2*, toutes ces informations sont stockées dans les inodes. La Fig. 16.13 représente cette arborescence avec en gras les *inodes* associés à chaque fichier/répertoire. Pour comprendre comment cette arborescence est stockée en utilisant le système de fichiers *ext2*, nous devons décrire plus en profondeur le contenu d'un inode. Dans un système de fichiers *ext2*, un *inode* contient différentes informations. Les plus importantes sont les suivantes :

— le mode du fichier/répertoire
— l'identifiant du propriétaire du fichier
— l'identifiant du groupe propriétaire du fichier
— la longueur du fichier (en octets)

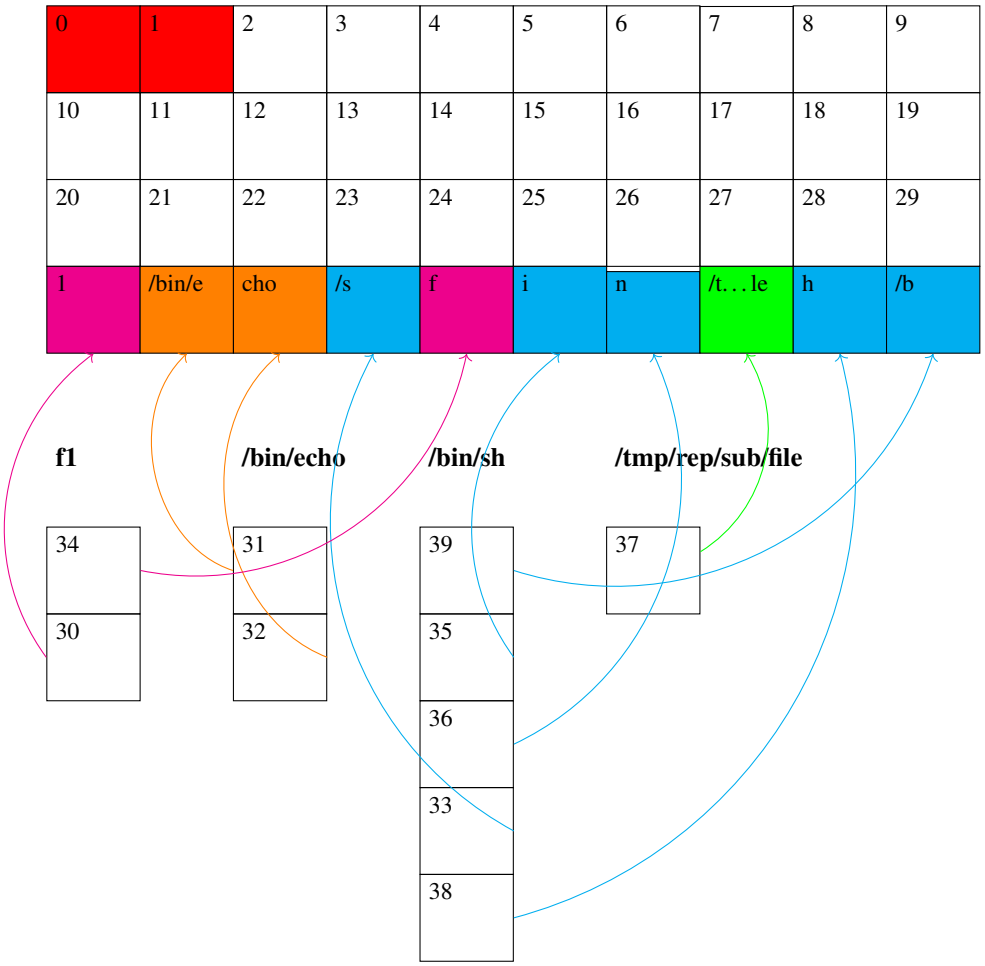



FIG. 16.11 – Les inodes stockent les listes chaînées correspondant à chaque fichier

| Super | FS | Block | Bitmap | Inode | Bitmap | Inode | Table | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |
| 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |

FIG. 16.12 – Un système de fichiers `ext2` contient un superblock, un FS descriptor, un bitmap des blocks, un bitmap des inodes, une table des inodes et des blocs de données

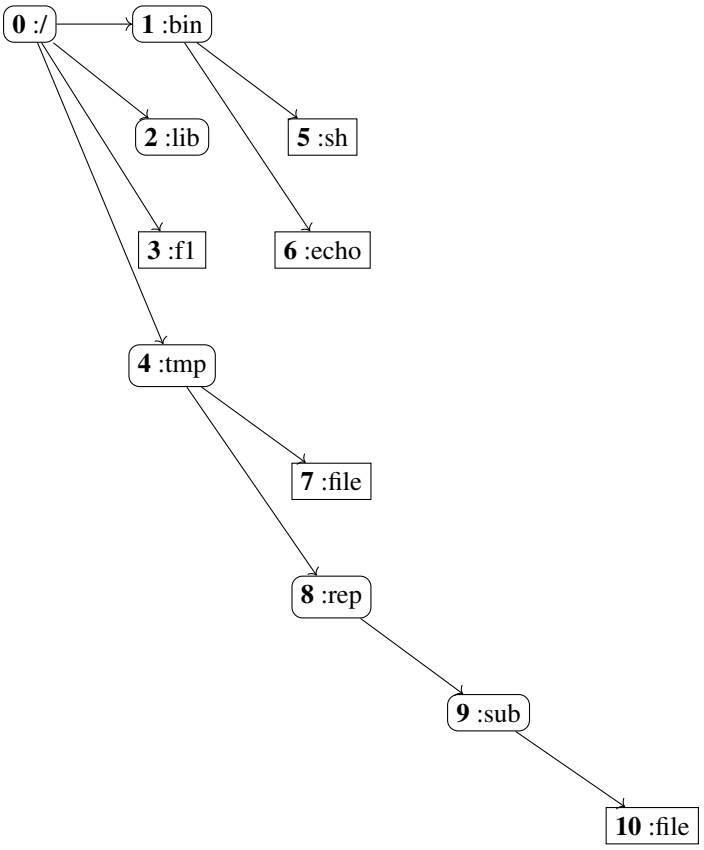



FIG. 16.13 – Arborescence des répertoires et fichiers de notre système `ext2`

— les dates de création, de dernier accès et de dernière modification du fichier
— le nombre de blocs du fichier
— le nombre de liens vers le fichier
— la liste des blocs qui composent le fichier

L'inode ne contient donc pas le nom du fichier. Celui-ci est uniquement spécifié dans le répertoire qui le contient. Le *mode* indique si il s'agit d'un fichier ou d'un répertoire. Ce champ contient également les attributs telles que les permissions de lecture, d'écriture et d'exécution sur le fichier. Comme Unix et Linux sont des systèmes d'exploitation multi-utilisateurs, l'inode comprend également les identifiants de l'utilisateur et du groupe qui sont propriétaires du fichier. Le système d'exploitation vérifie ces informations pour autoriser ou non une opération de lecture ou d'écriture sur le fichier en fonction des permissions de l'utilisateur qui exécute le programme. La longueur du fichier est spécifiée en nombre de blocs et en octets. La longueur en blocs est utile pour accéder facilement à la liste des blocs tandis que la longueur en octets est nécessaire lorsque le dernier bloc d'un fichier/répertoire est lu. Les dates sont encodées sous la forme d'un entiers de 32 bits qui contient le nombre de secondes depuis le premier janvier 1970. Le champ le plus important de l'inode est la liste des blocs.

---

**Note :** Le deuxième bug de l'an 2000

Les premières versions du système d'exploitation Unix datent du début des années 1970s. Quand les concepteurs de Unix ont réfléchi à une technique efficace pour encoder les dates de création des fichiers, ils ont opté pour un entier de 32 bits qui représente le nombre de secondes entre la date courante et une date de référence qu'ils ont arbitrairement fixé au premier janvier 1970. Sur un entier 32 bits, on peut stocker au maximum 4294967296 valeur différentes. En utilisant le premier janvier 1970 comme référence, la date la plus ancienne que l'on peut encoder est le 13 décembre 1901. Malheureusement la date la plus avancé que l'on pourra encoder est le mardi 19 janvier 2038. Les 32 bits du timestamp Unix ne permettent pas d'encoder de date postérieure à cette date. Ce problème est connu par les informaticiens comme étant le problème de l'année 2038. Il fait suite au problème de l'an 2000 où de nombreuses applications informatiques qui encodaient les années sur deux chiffres ont du être adaptées suite au passage du millénaire. Pour résoudre le problème de l'année 2038, il faut éviter d'encoder les timestamps sur 32 bits et préférer un encodage sur 64 bits. Plusieurs systèmes d'exploitation ont déjà fait le pas et mis à jour leurs systèmes de fichiers et autres structures de données.

---

Les concepteurs des systèmes de fichiers qui utilisent des inodes sont confrontés à une difficulté. D'un côté, il est important que chaque inode soit encodé en utilisant un nombre fixe d'octets pour faciliter l'accès direct à chaque inode sur le système de stockage. D'un autre côté, l'inode doit contenir la liste des blocs qui composent le fichier/répertoire. Une première approche serait de dire qu'un *inode* doit pouvoir contenir la liste des tous les blocs du fichier de taille maximale. Sur un système de stockage de 1 TBytes utilisant des blocs de 1024 octets, le plus grand fichier peut contenir un milliard de blocs. Si chaque bloc est identifié par un entier de 32 bits, cela signifierait qu'il faudrait réserver 4 milliards d'octets dans chaque *inode* au cas où cet *inode* correspondrait à un fichier gigantesque, alors que la quasi totalité des fichiers sont assez petits. Les systèmes de fichiers utilisant les inodes résolvent ce problème en utilisant un inode de petite taille, par exemple 128 octets. Cette taille permet de stocker 4 inodes directement dans un bloc de 512 octets. Elle permet aussi de stocker directement dans l'inode les numéros des premiers blocs du fichier. Pour des petits fichiers, qui ne contiennent quelque milliers d'octets, le système de fichiers peut récupérer les identifiants de blocs directement de l'inode. Pour les plus gros fichiers, il faut cependant utiliser un mécanisme plus complexe. En pratique, l'inode contient *n* entrées qui pointent directement vers des numéros de blocs. Ces entrées sont généralement appelées les pointeurs directs. L'entrée suivante (*n+1*) pointe elle vers un bloc qui contient des numéros de blocs. Cette entrée est généralement appelée pointeur avec une indirection simple. Si les numéros de blocs sont encodés sur 32 bits et qu'un bloc contient 512 octets, alors en utilisant l'inode plus ce premier bloc on peut encoder $n + 128$ numéros de blocs. La Fig. 16.14 illustre l'utilisation de ce type de pointeur. L'entrée suivante de l'inode est le pointeur avec une indirection double. Ce pointeur contient le numéro d'un bloc qui contient des numéros de blocs indirects. Via cette entrée, on peut donc référencer $128^2$ blocs du fichier. Pour les très longs fichiers, il reste le pointeur d'indirection triple. Ce pointeur contient le numéro d'un bloc qui contient des pointeurs d'indirection double. Via ce dernier pointeur, il est possible d'encoder $128^3$ blocs. La figure ci-dessous illustre comment, grâce à un pointeur indirect, un inode peut référencer un fichier composé de 16 blocs, même si il ne contient que douze pointeurs directs. Nous pouvons maintenant revenir à notre système de fichiers d'exemple. Celui-ci utilise dix inodes qui sont stockés

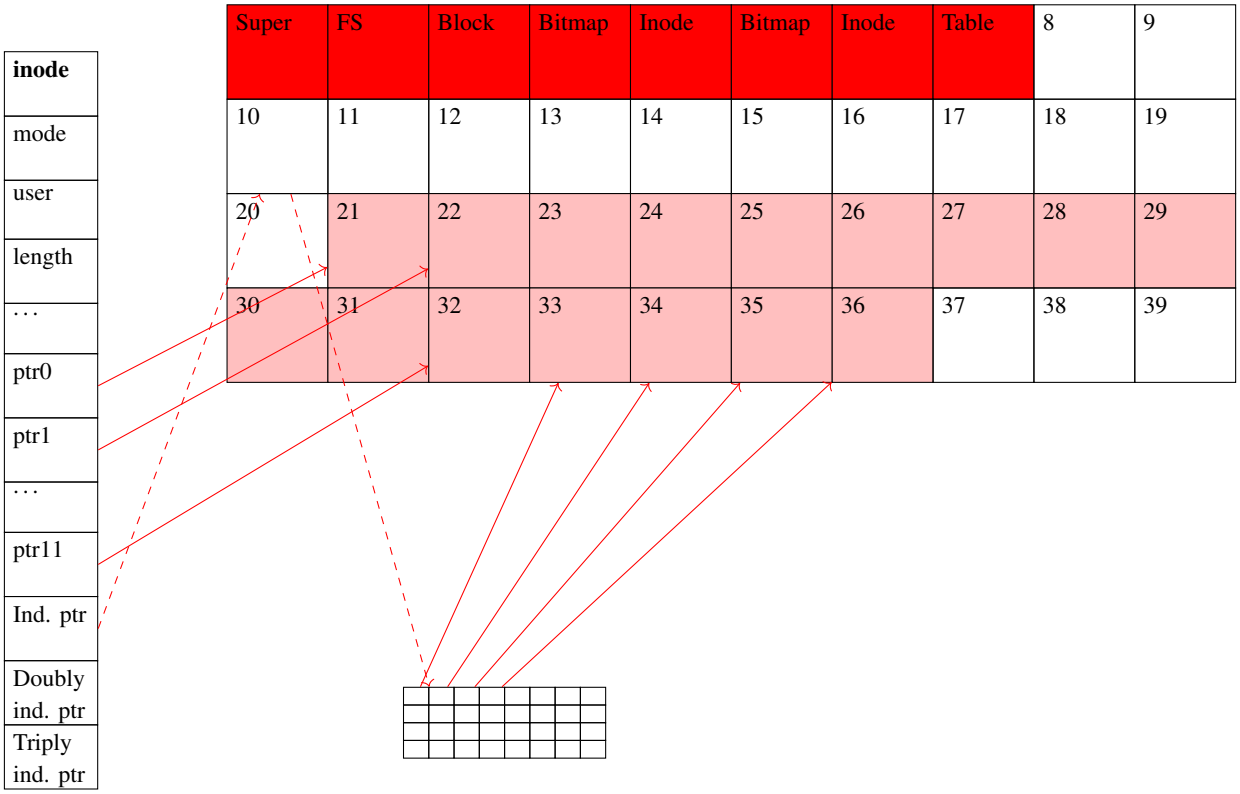



FIG. 16.14 – Un inode référençant un fichier de 16 blocs via un bloc indirect

dans la table des inodes. Il est représenté dans la Fig. 16.15. Il nous reste maintenant à expliquer le rôle des deux

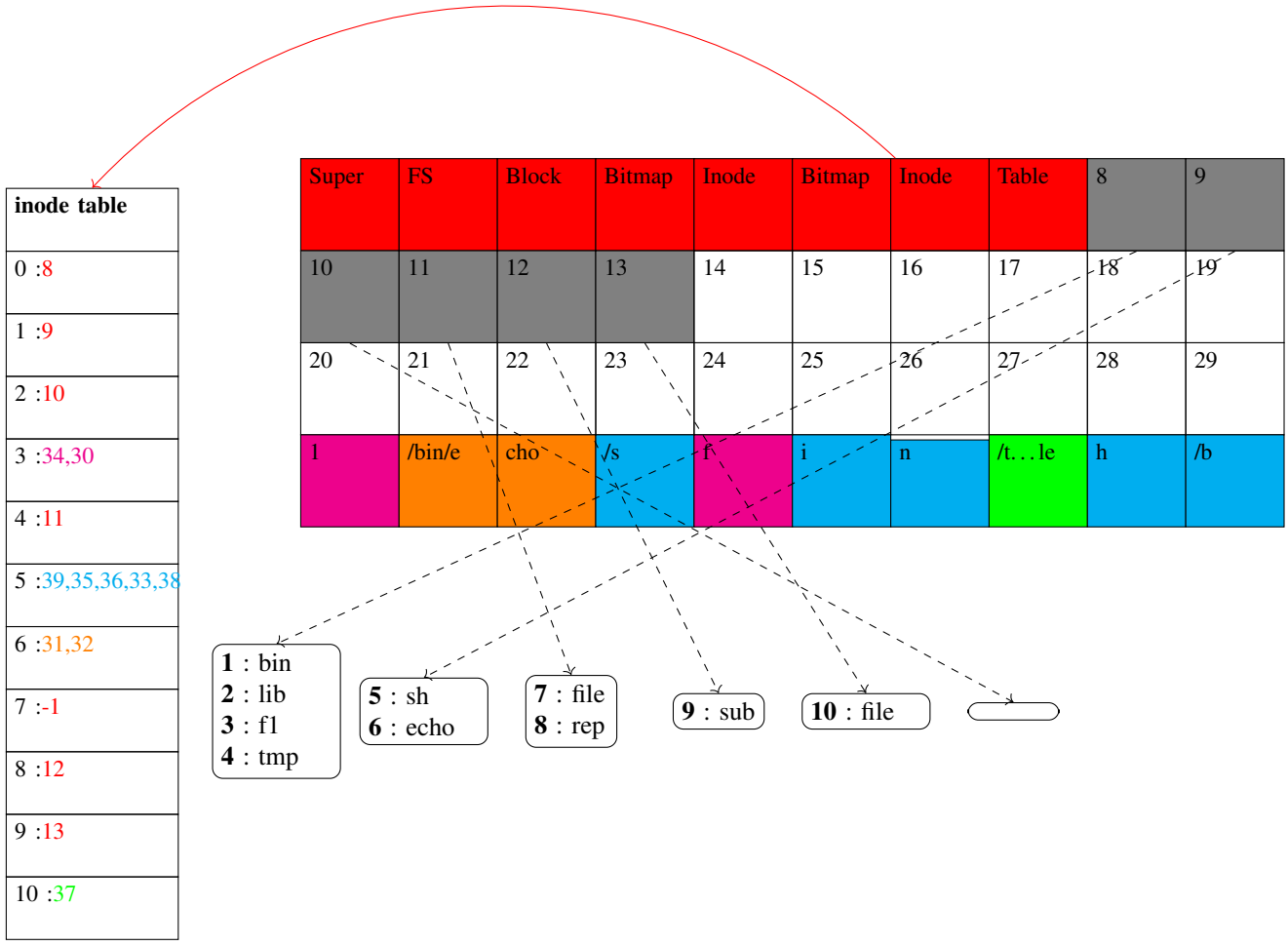



FIG. 16.15 – Notre système de fichiers d'exemple en format `ext2`

blocs qui contiennent les bitmaps. Dans un système de fichiers de type *ext2*, il est nécessaire de savoir si un bloc de données est utilisé par un fichier/répertoire ou libre. Il en va de même pour les inodes. Une première solution pour conserver cette information serait de maintenir une liste chaînée avec les numéros des blocs de données libres ou occupés. Malheureusement, une telle liste prendrait de la place sur le dispositif de stockage et pourrait être difficile à manipuler. Une solution plus efficace pour conserver cette information est d'utiliser un *bitmap*. Le bitmap des inodes est une structure de données simple qui utilise un bit pour indiquer si un inode est libre ou occupé. La structure contient autant de bits qu'il n'y a d'inodes dans le système de fichiers. Dans notre système de fichiers d'exemple, nous avons une dizaine d'inodes et 32 blocs. Si l'on s'en réfère à la Fig. 16.15, le bitmap des inodes contiendrait la chaîne de bits suivante, en supposant que `1` corresponde à un inode occupé et que le bit de poids fort corresponde à l'inode `0` :

Code source 16.2 – Bitmap des inodes

```
11111111 11000000
```

Pour le bitmap des blocs on procède de la même façon. Le bit correspondant à un bloc est mis à `1` lorsqu'il est occupé par un fichier/répertoire et `0` sinon. Dans notre système de fichiers d'exemple, le bitmap des blocs serait :

Code source 16.3 – Bitmap des blocs

```
11111100 00000000 00000011 11111111
```

Maintenant que nous avons décrit les principaux éléments du système de fichiers *ext2*, il est intéressant de voir plus en détails toutes les opérations qui doivent être réalisées sur le système de fichiers pour lire des données, écrire des données et créer un fichier.

Commençons par la lecture des données dans un fichier. Pour accéder aux données d'un fichier, le système de fichier doit d'abord accéder à l'inode qui le décrit. Cet inode est référencé dans l'entrée du répertoire qui correspond au fichier. Avant de lire le fichier, il faut vérifier qu'il s'agit bien d'un fichier et que l'utilisateur dispose des permissions de lecture sur ce fichier. Cette information est présente dans le champ `mode` de l'inode. Ensuite, le système de fichiers va parcourir la liste des pointeurs directs et indirects pour pouvoir accéder aux différents blocs de données qui composent le fichier. Grâce au champ `length` de l'inode, le système de fichiers pourra arrêter la lecture des données au dernier octet utile du fichier. Il faut aussi mettre à jour la date de dernier accès au fichier dans l'inode.

Les opérations d'écriture dans un fichier sont assez similaires sauf lorsqu'il faut ajouter un nouveau bloc à un fichier existant. Dans ce cas, le système de fichier va d'abord consulter le bitmap des blocs pour trouver un bloc libre. Pour améliorer les performances du système de fichiers, il est généralement utile de placer les blocs d'un fichier dans des zones contiguës, mais si les blocs qui suivent ceux utilisés par le fichier sont déjà occupés, rien n'empêche le système de fichiers de choisir un bloc dans une autre partie du dispositif de stockage. Une fois ce bloc choisi, il faut le référencer dans l'inode du fichier et mettre à jour la longueur du fichier (en blocs et en octets). Si il reste un pointeur direct de libre dans l'inode, il suffit de le modifier pour référencer le nouveau bloc. Sinon, il peut être nécessaire d'obtenir un nouveau bloc pour stocker un pointeur indirect, doublement indirect ou triplement indirect. Dans les trois cas, cela nécessite de trouver un nouveau bloc via le bitmap des blocs, mettre à jour ce bitmap et référencer ce bloc correctement. Il faut aussi mettre à jour les dates de dernière modification et d'accès au fichier dans l'inode.

Les opérations de création de fichier sont les plus complexes. Pour créer un nouveau fichier et y stocker des données, il faut d'abord trouver un inode de libre. Cela se fait en consultant le bitmap des inodes. On peut ensuite commencer à remplir l'inode avec les informations relatives au propriétaire du fichier, ... Ce fichier peut maintenant être référencé dans un répertoire. Si le bloc qui contient le répertoire est incomplet, il suffit d'ajouter l'entrée au répertoire. Si le bloc est complet, il faut consulter le bitmap des blocs pour trouver un nouveau bloc de libre, le marquer comme occupé, et l'ajouter dans l'inode du répertoire. Ensuite, il faut trouver dans le bitmap des blocs les blocs libres nécessaires au nouveau fichier et les référencer dans l'inode de ce nouveau fichier.

---

**Note :** Un même fichier peut se retrouver dans plusieurs répertoires

Les systèmes de fichiers utilisant les inodes ont une caractéristique que nous n'avons pas encore analysé. Comme une entrée de répertoire contient un nom de fichier/répertoire et le numéro de l'inode correspondant, il est possible qu'un même fichier soit accessible depuis plusieurs répertoires différents. Ce n'est pas une erreur, mais une optimisation introduite par les concepteurs de Unix. Sous Unix/Linux, c'est la commande ln qui permet de créer des liens vers des fichiers. Si l'on reprend l'exemple de la Fig. 16.15 et que l'on crée un lien vers le fichier `echo` dans les répertoires `/tmp` et `lib`, on obtiendra l'arborescence reprise en Fig. 16.16 et le système de fichiers sera modifié comme décrit dans la Fig. 16.17. Dans ce système de fichier, les noms `/bin/echo`, `/tmp/echo` et `/lib/echo` correspondent tous les trois au fichier qui est référencé par l'inode `6`.

Pour pouvoir gérer ces liens, le système de fichiers utilise le champ *nombre de liens vers le fichier* qui se trouve dans l'inode. Quand un fichier est créé, ce champ est initialisé à la valeur `1`. Cette valeur indique qu'il existe un seul lien vers le fichier qui est représenté par cet inode. Si un deuxième répertoire fait référence à cet inode, alors ce champ passera à la valeur `2`. Ce champ est important lors des opérations de suppression des fichiers. Quand un fichier est effacé d'un répertoire, le système de fichiers décrémente d'abord le champ contenant le nombre de liens vers le fichier. Si celui-ci passe à zéro, alors l'inode et les blocs utilisés par le fichier sont marqués comme libres et les bitmaps sont mis à jour. Sinon, cela signifie que le fichier est encore référencé dans au moins un autre répertoire. Il est important de noter que lorsque

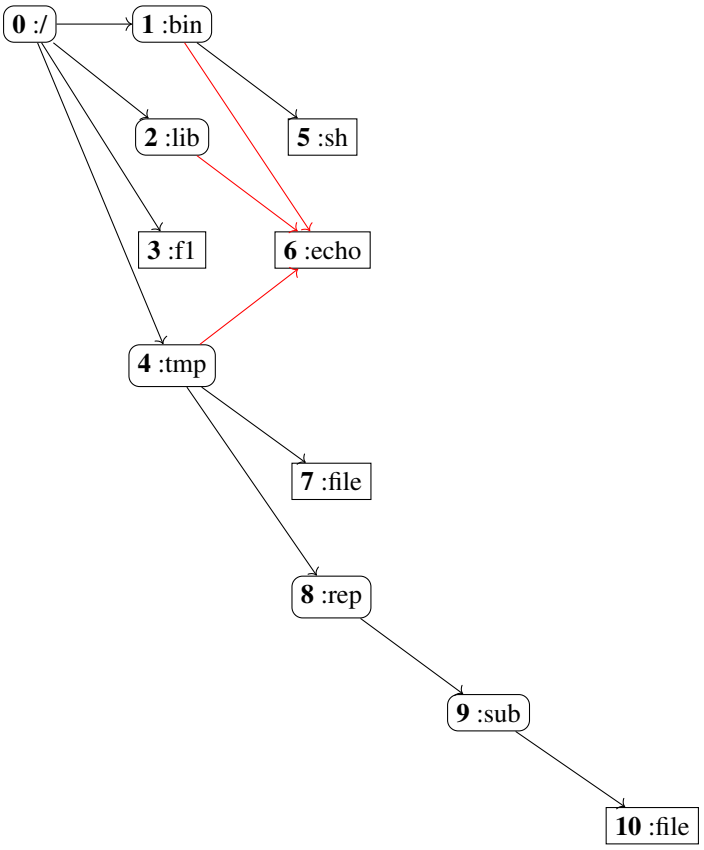



FIG. 16.16 – Arborescence des répertoires et fichiers de notre système `ext2`

plusieurs liens existent vers le même fichier, tout accès en écriture à ce fichier est directement visible depuis tous les répertoires qui y font référence.

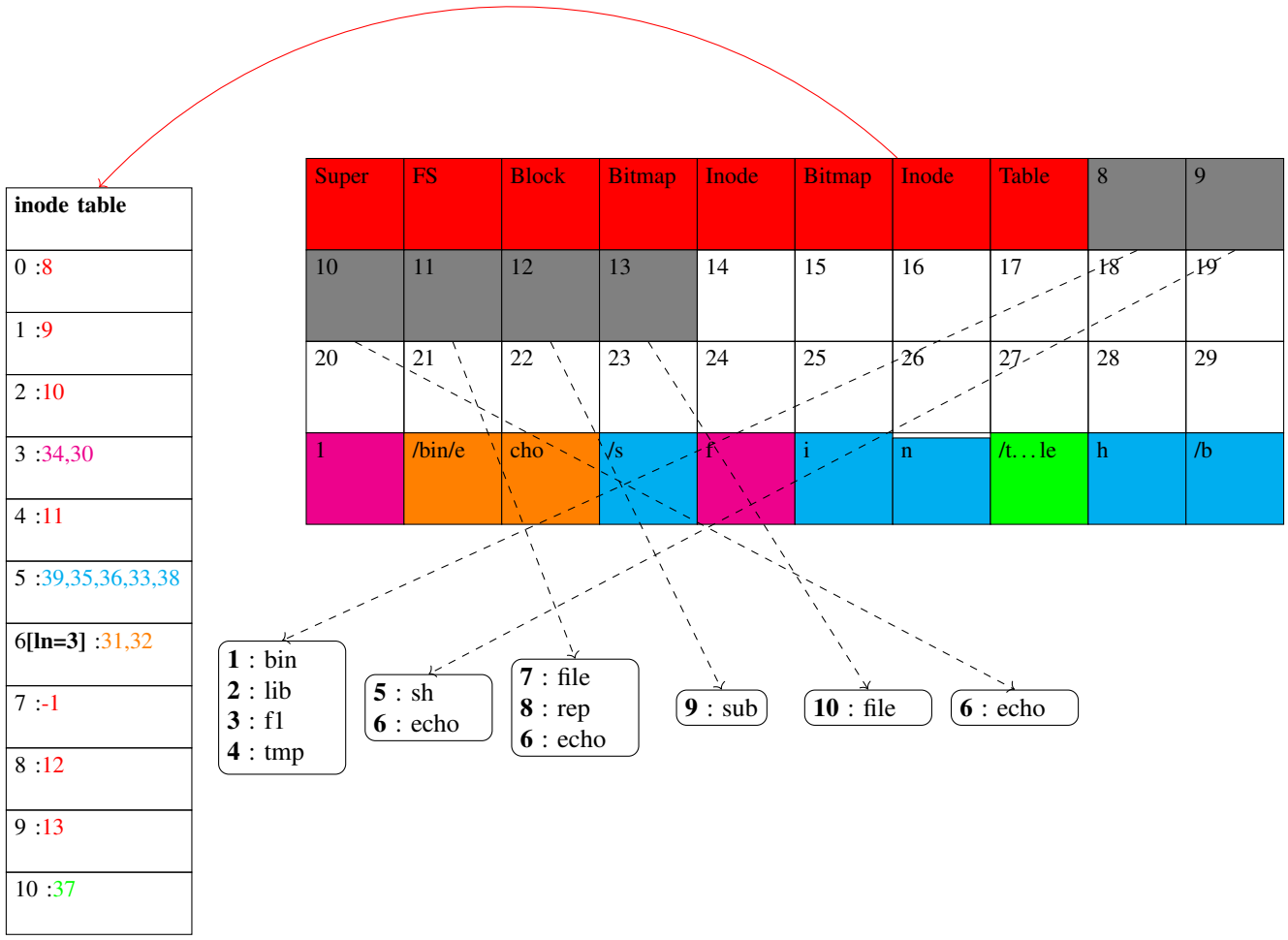



FIG. 16.17 – Notre système de fichiers d'exemple en format `ext2` avec deux liens pour le fichier ``echo``

Tout comme pour le système de fichiers utilisant une table d'allocation, un système utilisant des inodes doit pouvoir survivre à plusieurs types de perturbations. Les premiers problèmes à considérer sont la perte d'un bloc. Si un bloc du système de fichiers devient illisible, quel impact cela peut-il avoir sur le système de fichiers. La seconde classe de problèmes est un système de fichiers qui se trouve dans un état incohérent car l'ordinateur qui le gérait a été arrêté brusquement alors qu'il était en train d'écrire sur le dispositif de stockage les modifications de blocs relatives à une opération. Tous ces problèmes ajoutent une couche importante de complexité aux systèmes de fichiers et à leur implémentation dans les systèmes d'exploitation. Chaque système d'exploitation contient des utilitaires spécialisés pour récupérer partiellement ou totalement un système de fichiers incohérent. C'est le cas notamment de fsck sous Linux ou de chkdsk sous Windows.

Commençons par analyser l'impact de la perte d'un bloc. Les deux premiers blocs, le Super Block et le FS descriptor, sont critiques car ils contiennent les paramètres du système de fichiers. Ils définissent notamment le nombre de blocs sur le dispositif de stockage, le nombre d'inodes et la taille de chaque bloc. Si ces blocs deviennent inutilisables, le système de fichiers l'est aussi. Pour faire face à ce risque, la solution la plus fréquente est de stocker une copie de ces deux blocs dans une autre partie du disque. Cette copie doit évidemment être mise à jour à chaque modification des blocs primaires.

Les bitmaps jouent un rôle important dans le fonctionnement du système de fichiers. Si une partie du bitmap des blocs devenait inaccessible, alors le système de fichiers ne saurait plus quels blocs sont libres. Un utilitaire spécialisé pourrait récupérer cette panne en parcourant les blocs contenant les inodes pour voir quels inodes sont utilisés. Cela suppose que lorsqu'un fichier est effacé son inode est marqué comme étant libre sur le disque et dans le bitmap des inodes. Cela peut se faire par exemple en mettant certains champs de l'inode à une valeur de référence comme 0.

Si le bitmap des blocs est affecté par une erreur, la situation est un peu différente. Il faudra dans ce cas parcourir tous les inodes pour voir quels sont les blocs du dispositif de stockage qui sont référencés et reconstruire le bitmap des blocs. Ce bitmap peut ensuite être recopié dans une autre partie du dispositif de stockage. Si un bloc contenant un répertoire est devenu illisible, les fichiers et les sous-répertoires de ce répertoire ne seront plus accessibles. Il faudra faire appel à un utilitaire spécialisé pour les récupérer. Si un bloc contenant des données est illisible, le fichier correspondant devient malheureusement aussi illisible.

Les incohérences du système de fichier sont plus complexes. Elles dépendent de l'ordre dans lequel les opérations de modification au système de fichiers sont réalisées et également du moment auquel l'ordinateur est arrêté. Pour récupérer ces incohérences, les logiciels spécialisés procèdent généralement en deux phases. Tout d'abord, ils doivent lire tous les blocs de contrôle (Super Block, FS descriptor, Inode Bitmap, Block Bitmap et Inode Table) du système de fichiers. Ensuite, il faut vérifier la cohérence entre d'abord le bitmap des inodes et l'information lue dans la table des inodes. En cas d'incohérence, ce sont les inodes qui seront considérés comme valides. L'étape suivante est de comparer le bitmap des blocs avec les blocs référencés dans les inodes. La troisième étape sera d'analyser l'arborescence des répertoires et fichiers. Cette structure doit être un arbre et non un graphe contenant des cycles. Si un cycle est détecté, il devra être supprimé généralement avec l'aide de l'utilisateur. En comparant la liste des inodes et les inodes référencés dans les répertoires, il est possible que l'on trouve un ou des inodes qui ne sont pas repris dans l'arborescence. Ces fichiers et répertoires seront recréés avec un nom générique et placé dans le répertoire `/lost+found`. L'administrateur du système de fichiers pourra consulter le contenu de ces fichiers pour déterminer si ils doivent être conservés ou peuvent être effacés du système de fichiers.

CHAPITRE 17

# Glossaire

**loi de Moore** Prédiction de Gordon Moore indiquant que les nombre de composants d'un circuit intégré double tous les deux ans. Voir notamment https://en.wikipedia.org/wiki/Moore%27s_law

**architecture de type Harvard**

**architecture Harvard** Organisation d'un ordinateur qui utilise des mémoires séparées pour les programmes et les données. Cette architecture avait été proposé pour l'ordinateur Mark I conçu à l'université de Harvard. La plupart des ordinateurs actuels utilisent l'architecture de von Neuman. Voir https://en.wikipedia.org/wiki/Harvard_architecture

**architecture de von Neumann** Organisation d'un ordinateur qui utilise une mémoire pour stocker à la fois les programmes et les données. Cette architecture est utilisée par la plupart des ordinateurs actuels. Voir https://en.wikipedia.org/wiki/Von_Neumann_architecture

**système d'exploitation** Logiciel permet de contrôler l'utilisation du matériel (mémoire, processeur, entrées-sorties) par les programmes applicatifs. Les systèmes d'exploitation courant sont Windows, MacOS et Linux.

**GPU**

**Graphics Processing Unit** Un GPU est un ensemble de circuits électroniques qui sont spécialisés dans les calculs nécessaires pour afficher de l'information à l'écran. Ils excellent aussi pour l'édition de séquences vidéo et l'apprentissage automatique. Voir https://en.wikipedia.org/wiki/Graphics_processing_unit

**inline**

**fonction inline** Ce dit d'une fonction dont le corps est exécuté à l'intérieur d'un code existant.

**mémoire cache** TODO

**bus** TODO

**pile** Structure de données permettant de stocker un nombre quelconque de données. Elle supporte deux opérations : l'ajout d'une donnée au sommet de la pile et le retrait de la donnée se trouvant au sommet de la pile.

**passage par valeur** TODO

**passage par référence** TODO

**c** TODO

**java** TODO

**python** TODO

**adresse de retour** TODO

**récursion** TODO

CHAPITRE 18

# Indices et tables

— genindex
— modindex
— search

# Index

## L



## M



## N



## O



## P



## Q



## R



## S



## T



## U



## W



## X