LSINC1102

Version 2020

déc. 07, 2020

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 cours 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, le syllabus est interactif, 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.

Le cours est divisé en cinq missions qui abordent chacune un aspect particulier du fonctionnement des ordinateurs. Chaque mission dure deux semaines. Durant la première semaine, nous détaillerons les principes théoriques et vérifierons via des exercices simples qu'ils sont bien compris par tous les étudiants. Ceux-ci seront mis en oeuvre durant la seconde partie de la mission via un mini-projet qui sera réalisé de façon individuelle.

Nous aborderons cinq aspects différents 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. Durant la première mission, 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 durant la deuxième mission comment représenter les nombres sous forme binaire et ensuite comment construire des circuits qui permettent de réaliser des additions, des soustractions et d'autres opérations sur les nombres entiers. La troisième mission se concentrera sur comment un ordinateur peut mémoriser de l'information et utiliser l'information stockée en mémoire.

Avec ces trois premières missions, nous aurons appris les bases qui permettent de concevoir un microprocesseur simple qui pourra être programmé. Nous commencerons par construire un langage machine qui permet de programmer ce microprocesseur. Grâce à un simulateur, nous pourrons tester de petits programmes sur notre futur microprocesseur avant de le construire. La mission suivante se concentrera 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 définir un langage d'assemblage qui est plus facile à utiliser pour les programmeurs que le langage machine.

Une version imprimable de ce document est disponible via `LSINC1102.pdf`.

CHAPITRE 2

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 considéré 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.

2.1 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
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.

2.1.1 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

Exercices

1. Construisez la table de vérité de la fonction booléenne à quatre entrées $AND(x, OR(y, AND(z, a)))$
2. Construisez la table de vérité de la fonction booléenne à trois entrées $OR(AND(NOT(x), y, NOT(z)), AND(x, NOT(y), z))$
3. Construisez la table de vérité de la fonction booléenne à quatre entrées $AND(x, OR(y, AND(z, a)))$

2.1.2 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.

Exercices

1. En utilisant une table de vérité, démontrez que $AND(x, OR(NOT(x), y)) \iff AND(x, y)$
2. En utilisant une table de vérité, démontrez que $OR(x, AND(NOT(x), y)) \iff OR(x, y)$
3. En utilisant une table de vérité, démontrez la première loi de De Morgan $NOT(OR(x, y)) = AND(NOT(x), NOT(y))$
4. En utilisant une table de vérité, démontrez la deuxième loi de De Morgan $NOT(AND(x, y)) = OR(NOT(x), NOT(y))$
5. Considérons la fonction booléenne $OR(AND(NOT(x), y), AND(y, NOT(z)), AND(y, z), AND(x, AND(NOT(y), NOT$ Pouvez-vous simplifier cette fonction en utilisant uniquement une fonction booléenne *AND* à deux entrées, une fonction *OR* à deux entrées et un inverseur ?

6. Même question pour la fonction $AND(OR(x, y), OR(x, z))$

7. Même question pour la fonction $OR(x, AND(x, y), AND(x, NOT(y), z))$

2.1.3 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.

2.2 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 *DIFF* qui retourne *1* lorsque ses deux entrées sont différentes et *0* sinon. Sa table de vérité est reprise ci-dessous.

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

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.

2.2.1 Exercices

1. En utilisant uniquement des fonctions *AND*, *OR* et *NOT*, réalisez un multiplexeur.
2. En utilisant uniquement des fonctions *AND*, *OR* et *NOT*, réalisez un démultiplexeur.

2.3 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. 2.1 représente l'inverseur ou la fonction *NOT*. La fonction *OR* est présentée schématiquement dans la Fig. 2.2 et la fonction *AND* dans la Fig. 2.3. La fonction *XOR* a aussi sa représentation graphique. Celle-ci est présentée dans la Fig. 2.4. Dans

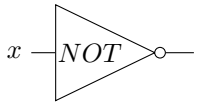

Fig. 2.1 – Représentation graphique d'une fonction NOT

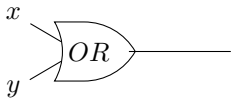

Fig. 2.2 – Représentation graphique d'une fonction OR

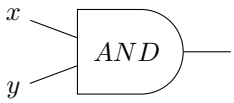

Fig. 2.3 – Représentation graphique d'une fonction AND

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. 2.5. Cette inversion est symbolisée par un petit rond. Il en va de même pour la fonction *NOR* (Fig. 2.6). Les multiplexeurs et démultiplexeurs ont aussi leur représentation graphique. Le livre les représente en utilisant un triangle comme dans la Fig. 2.7. De la même façon, on peut également représenter le démultiplexeur de façon graphique comme représenté dans la Fig. 2.8. 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

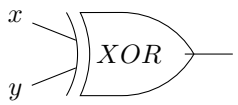

Fig. 2.4 – Représentation graphique d'une fonction XOR

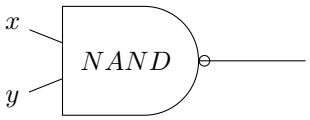

Fig. 2.5 – Représentation graphique d'une fonction NAND

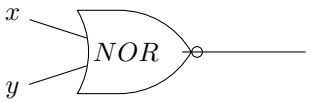

Fig. 2.6 – Représentation graphique d'une fonction NOR

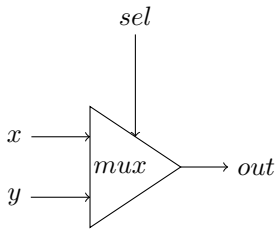

Fig. 2.7 – Un multiplexeur à deux entrées

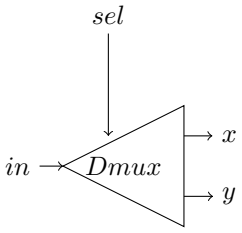

Fig. 2.8 – Un démultiplexeur à deux sorties

Cette fonction peut être réalisée en utilisant deux fonctions *AND*, une fonction *OR* et des inverseurs (Fig. 2.9). Un

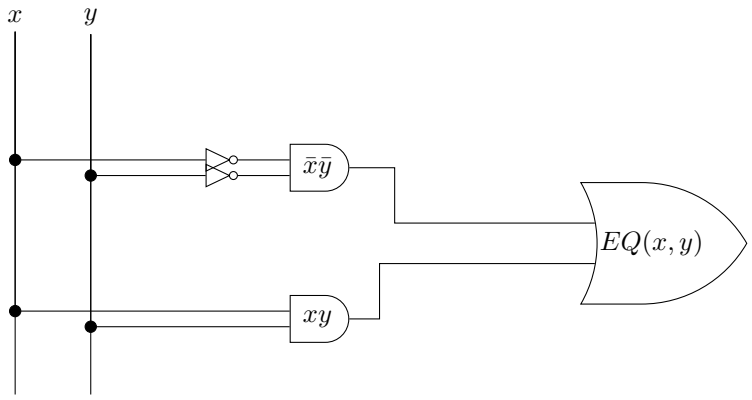

Fig. 2.9 – Représentation graphique d'un circuit qui réalise la fonction EQ

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. 2.10. Avec un multiplexeur,

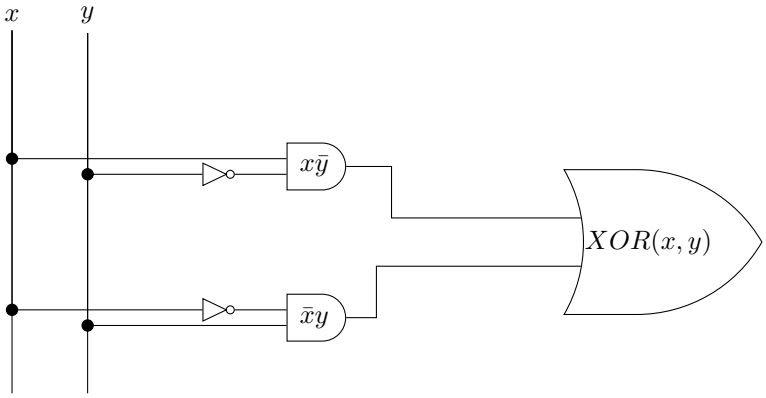

Fig. 2.10 – Représentation graphique d'un circuit qui réalise la fonction XOR

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. 2.11.

2.3.1 Exercices

1. Quelle est la table de vérité qui correspond au circuit représenté dans la Fig. 2.12 ?

2. Quelle est la table de vérité qui correspond au circuit de la Fig. 2.13 ?

2.4 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.

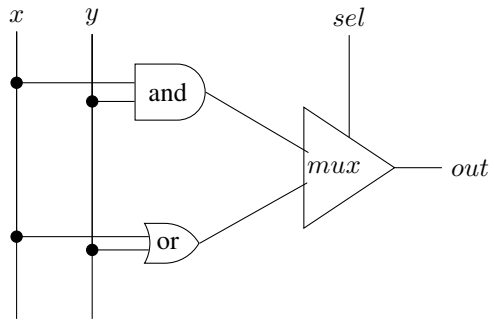

Fig. 2.11 – Un circuit programmable

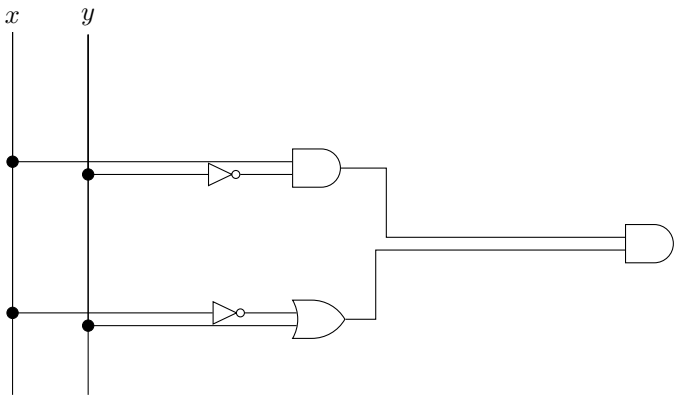

Fig. 2.12 – Un circuit simple à deux entrées

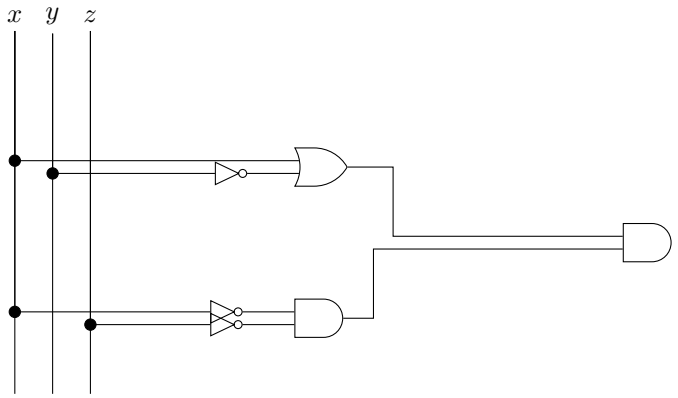

Fig. 2.13 – Un circuit simple à trois entrées

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 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 *.out*)

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.

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

```
// Un commentaire
CHIP Nom {   // Le nom du circuit doit être le même que le nom du fichier
   IN ...  // les entrées du circuit
   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. 2.14. Un autre exemple est de construire un circuit

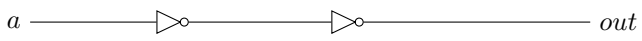

Fig. 2.14 – Représentation graphique du circuit qui ne fait rien

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
```

(suite sur la page suivante)

(suite de la page précédente)

```
 */
CHIP Xor {
    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

2.4.1 Exercices

1. Avec un multiplexeur, il est possible de construire des circuits « programmables », c'est-à-dire des circuits pour lesquels une des entrées permet de choisir la fonction calculée. Considérons le circuit hypothétique représenté dans la Fig. 2.15 :

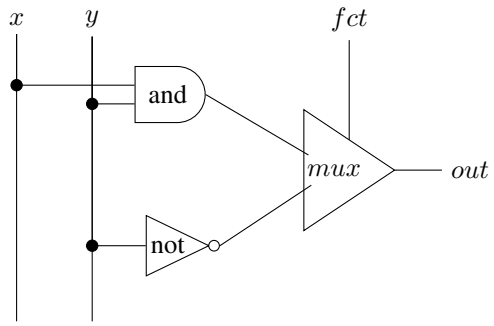

Fig. 2.15 – Un exemple de circuit programmable

Construisez d'abord la table de vérité de ce circuit et ensuite proposez une suite de test qui permet de valider qu'une implémentation de ce circuit est correcte.

CHAPITRE 3

Préparation au premier projet

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 technologies, 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.

3.1 Les fonctions « multi-bits »

Les fonctions multi-bits sont simplement des fonctions qui sont appliquées de la même façon à plusieurs entrées. Le circuit de la Fig. 3.1 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*. Ces deux circuits sont représentés dans les figures Fig. 3.2 et Fig. 3.3. De la même façon, on peut construire des multiplexeurs et des démultiplexeurs à k-bits.

3.2 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

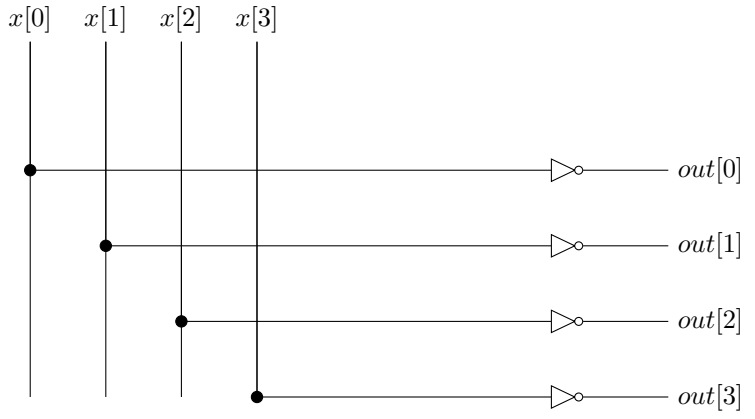

Fig. 3.1 – Représentation graphique d'un circuit NOT 4-bits

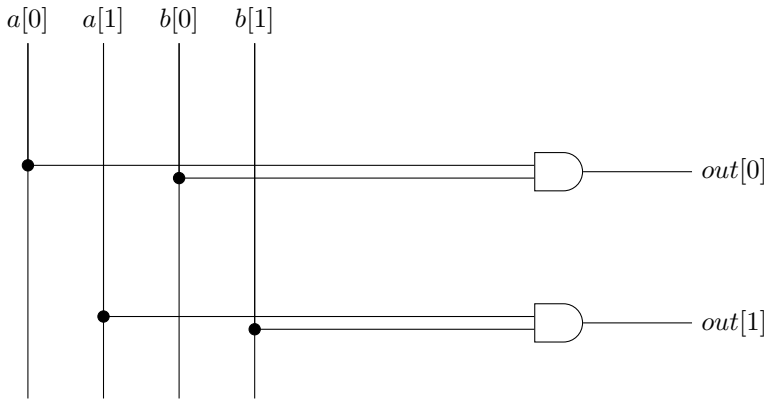

Fig. 3.2 – Représentation graphique d'un circuit AND 2-bits

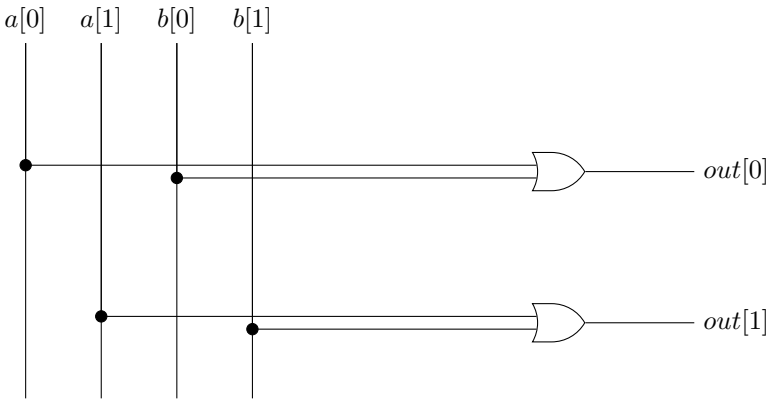

Fig. 3.3 – Représentation graphique d'un circuit OR 2-bits

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

3.2.1 Exercices

La fonction *NAND* est une fonction de base qui permet d'implémenter toutes les autres fonctions booléennes.

1. En appliquant les lois de De Morgan, il est aussi possible de construire la fonction *OR* en se basant uniquement sur la fonction *NAND*.

2. En utilisant uniquement des fonctions *NAND*, implémentez les fonctions suivantes : - *XOR* - *NOR*
3. La fonction *NOR* est également une fonction universelle qui permet d'implémenter n'importer quelle fonction logique. En utilisant uniquement une fonction *NOR*, implémentez les fonctions suivantes :
   — inverseur (*NOT*)
   — *OR*
   — *AND*
   — *XOR*
   — *NAND*

Pour rappel, la table de vérité de la fonction *NOR* est la suivante :

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

3.3 Premier projet

Votre premier projet dans le cadre de ce cours est de construire les circuits de base de l'on retrouve dans tout ordinateur en utilisant exclusivement des fonctions *NAND*. Ces circuits sont :

— *NOT* (une entrée), voir https://inginious.info.ucl.ac.be/course/LSINC1102/Not
— *AND* (deux entrées), voir https://inginious.info.ucl.ac.be/course/LSINC1102/And
— *OR* (deux entrées), voir https://inginious.info.ucl.ac.be/course/LSINC1102/Or
— *XOR* (deux entrées), voir https://inginious.info.ucl.ac.be/course/LSINC1102/Xor
— Multiplexeur (deux entrées et sélecteur), voir https://inginious.info.ucl.ac.be/course/LSINC1102/Mux
— Démultiplexeur (une entrée et sélecteur, une sortie), voir https://inginious.info.ucl.ac.be/course/LSINC1102/Dmux

Construisez ces différents circuits dans l'ordre indiqué en réutilisant pour chaque circuit votre circuit précédent.

Dans la suite du cours, vous devrez aussi utiliser des circuits qui manipulent des mots de 16 bits. Vous devez donc construire les circuits :

— *NOT16* (16 entrées et 16 sorties), voir https://inginious.info.ucl.ac.be/course/LSINC1102/Not16
— *AND16* (16 entrées et 16 sorties), voir https://inginious.info.ucl.ac.be/course/LSINC1102/And16
— *OR16* (16 entrées et 16 sorties), voir https://inginious.info.ucl.ac.be/course/LSINC1102/Or16

— Multiplexeur16 (2 fois 16 entrées, un sélecteur et 16 sorties), voir https://inginious.info.ucl.ac.be/course/LSINC1102/Mux16

En outre, vous devez également construire les circuits suivants :

— une fonction *OR* avec 8 entrées et une sortie (*Or8Way*), voir https://inginious.info.ucl.ac.be/course/LSINC1102/Or8Way
— un multiplexeur avec 4 entrées sur 16 bits, un sélecteur sur 2 bits et 16 sorties (*Mux4Way16*), voir https://inginious.info.ucl.ac.be/course/LSINC1102/Mux4Way16
— un multiplexeur avec 8 entrées sur 16 bits, un sélecteur sur 3 bits et 16 sortie (*Mux8Way16*), voir https://inginious.info.ucl.ac.be/course/LSINC1102/Mux8Way16
— un démultiplexeur une entrée sur 16 bits, un sélecteur sur 2 bits et 4 sorties sur 16 bits (*DMux4Way*), voir https://inginious.info.ucl.ac.be/course/LSINC1102/DMux4Way
— un démultiplexeur une entrée sur 16 bits, un sélecteur sur 3 bits et 8 sorties sur 16 bits (*DMux8Way*), voir https://inginious.info.ucl.ac.be/course/LSINC1102/DMux8Way

Ce projet est également décrit en ligne sur le site nand2tetris.org/project01.

La date limite pour ce projet est fixée au lundi 12 octobre 2020 à 18h00. Vous devez déposer toutes vos solutions aux exercices pour cette date sur https://inginious.info.ucl.ac.be/course/LSINC1102/ Vous aurez un retour sur votre projet durant la séance de travaux pratiques du vendredi 16 octobre 2020.

CHAPITRE 4

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.

4.1 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 4.1 – 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 |
```

(suite sur la page suivante)

(suite de la page précédente)

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 4.1) 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 dans 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. 4.1.

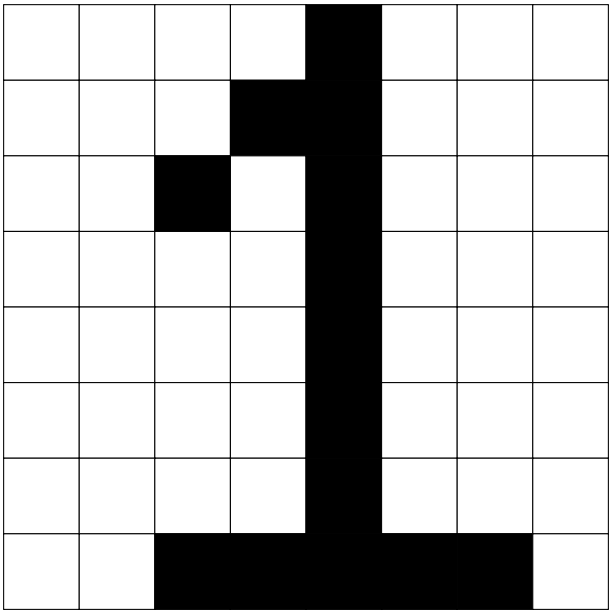

Fig. 4.1 – Un caractère sous la forme de pixels

4.2 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 deux 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*

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.

4.2.1 Exercices

1. Écrivez la table de vérité de la fonction de décalage permettant de décaler bloc de quatre bits (nibble en anglais et index :*quartet* ou index :*semi-octet* en français d'une place vers la droite. Implémentez ensuite cette fonction en utilisant uniquement des fonctions *AND*, *OR* et *NOT*.
2. Écrivez la table de vérité de la fonction de décalage permettant de décaler un quartet (4 bits) d'une place vers la gauche. Implémentez ensuite cette fonction en utilisant uniquement des fonctions *AND*, *OR* et *NOT*.
3. Écrivez la table de vérité de la fonction de décalage permettant de décaler un quartet (4 bits) de *p* places vers la droite. Pour écrire cette date de vérité, on utilisera deux bits pour représenter l'entrée *p* est les séquences de deux bits suivantes pour représenter les entiers de *0* à *3*.
   — *00* représente l'entier 0
   — *01* représente l'entier 1
   — *10* représente l'entier 2
   — *11* représente l'entier 3

   Implémentez ensuite cette fonction en utilisant uniquement des fonctions *AND*, *OR* et *NOT*.
4. Faites de même pour le décalage de *p* places vers la droite.

Les fonctions de décalage sont utiles pour certaines manipulations sur les bits dans une séquence. Malheureusement, elles résultent en une perte d'information puisque un ou des bits de poids faible sont perdus lors d'un décalage vers la droite. Les fonctions de rotation évitent ce problème. Elles peuvent notamment servir à construire des algorithmes pour crypter (et décrypter) des données stockées sous forme binaire.

La rotation la plus simple est la rotation d'une place vers la droite. Cette fonction prend en entrée une séquence de bits $b_{n-1}b_{n-2}...b_2b_1b_0$ et retourne la séquence $b_0b_{n-1}b_{n-2}...b_2b_1$. D'une façon similaire, dans un décalage à gauche d'une place, lorsque la fonction reçoit la séquence $b_{n-1}b_{n-2}...b_2b_1b_0$ en entrée, elle retourne la séquence $b_{n-2}...b_2b_1b_0b_{n-1}$. Tout comme pour les fonctions de décalage, les fonctions de rotation peuvent recevoir une seconde entrée qui est le nombre de places de rotation.

4.2.2 Exercices

1. Écrivez la table de vérité de la fonction qui réalise la rotation d'une place vers la gauche d'un quartet (4 bits). Implémentez ensuite cette fonction en utilisant uniquement des fonctions *AND*, *OR* et *NOT*.

2. Écrivez la table de vérité de la fonction qui réalise la rotation d'un quartet (4 bits) de *p* places vers la gauche. Pour écrire cette table de vérité, on utilisera deux bits pour représenter l'entrée *p* est les séquences de deux bits suivantes pour représenter les entiers de *0* à *3*.
   — *00* représente l'entier *0*
   — *01* représente l'entier *1*
   — *10* représente l'entier *2*
   — *11* représente l'entier *3*

   Implémentez ensuite cette fonction en utilisant uniquement des fonctions *AND*, *OR* et *NOT*.

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 explore 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.

4.2.3 Exercices

1. Écrivez la table de vérité d'une fonction qui prend une séquence de trois bits en entrée et retourne un bit de parité paire.
2. Écrivez la table de vérité d'une fonction qui prend une séquence de trois bits en entrée et retourne un bit de parité impaire.
3. Écrivez la table de vérité d'une fonction qui prend en entrée un quartet dont le bit de poids faible contient une parité paire et retourne *1* si ce quartet est valide et *0* sinon.
4. Écrivez la table de vérité d'une fonction qui prend en entrée un quartet dont le bit de poids faible contient une parité impaire et retourne *1* si ce quartet est valide et *0* sinon.

CHAPITRE 5

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.

5.1 Représentation des nombres naturels

Commençons par analyser comme 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 $7 * 10^2 + 3 * 10^2 + 6 * 10^0$.

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
```

5.1.1 Exercices

1. Quel est le nombre décimal qui correspond au nombre binaire *1001100* ?
2. Quel est le nombre décimal qui correspond au nombre binaire *00001101110* ?
3. Comment peut-on facilement reconnaître si un nombre en notation binaire est :
   — pair
   — impair
4. **Quel est le plus grand nombre naturel que l'on peut représenter en utilisant :**
   — un nombre binaire sur 4 bits
   — un nombre binaire sur 6 bits
   — un nombre binaire sur 8 bits
5. Considérons le nombre naturel en représentation binaire $N = B_{n-1}B_{n-2}...B_2B_1B_0$. Construisons le nombre *M* dans lequel on ajoute un bit de poids fort à 0, c'est-à-dire $M = 0B_{n-1}B_{n-2}...B_2B_1B_0$. Quelle relation y-a-t-il entre *N* et *M* ?
   — $N > M$
   — $N < M$
   — $N = M$
6. Considérons le nombre naturel en représentation binaire $N = B_{n-1}B_{n-2}...B_2B_1B_0$. Construisons le nombre *P* dans lequel on ajoute un bit de poids faible à 0, c'est-à-dire $P = B_{n-1}B_{n-2}...B_2B_1B_00$. Quelle relation y-a-t-il entre *N* et *P* ?
   — $N > P$
   — $N < P$
   — $N = P$
7. Combien de nombres naturels différents peut-on représenter avec un nombre décimal sur :
   — 8 bits (c'est-à-dire un byte ou un octet)
   — 16 bits
   — 32 bits

5.2 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 case 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 $2+5 = 4$. 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. 5.1 le représente schématiquement sous la forme d'un rectangle avec (*a*, *b* et *r*) et deux sorties (*report* et *somme*). Le plus intéressant est que ces additionneurs peuvent

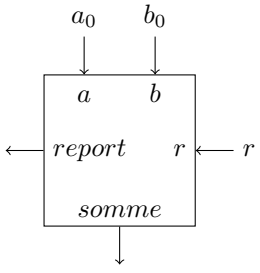

Fig. 5.1 – Un additionneur complet

se combiner en cascade pour construire un additionneur qui est capable d'additionner deux nombres binaires sur *n* bits. La Fig. 5.2 présente un additionneur qui travaille avec deux quartets, *a* et *b*. Pour des raisons graphiques, il est

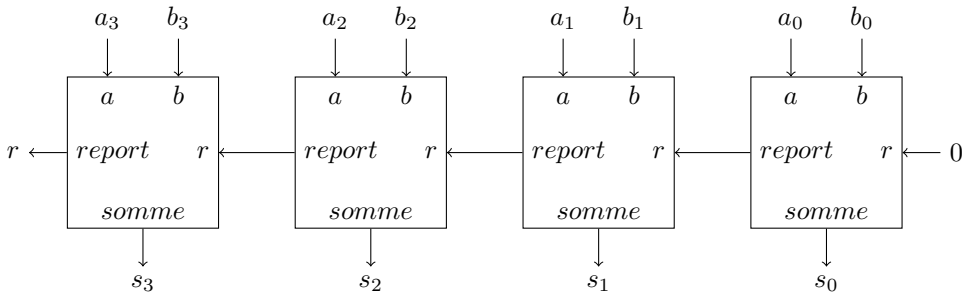

Fig. 5.2 – Avec quatre additionneurs, on peut additionner des quartets

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* 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. 5.3.

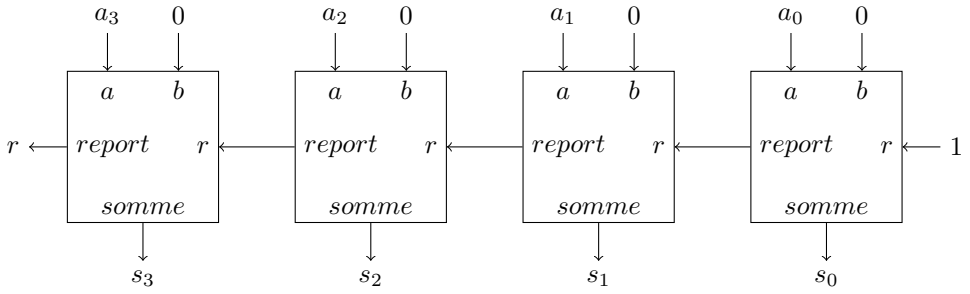

Fig. 5.3 – Un circuit pour incrémenter un quartet

5.3 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 la présence du facteur *(-1)* qui est appliqué 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.

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 de d'effectuer des opérations arithmétique 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
+  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	b0	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. 5.4. Si on sait facilement calculer l'opposé d'un nombre, et additionner deux nombres, il devient possible de réaliser la soustraction. Pour calculer $a - b$, il suffit de calculer $a + (-b)$. Le

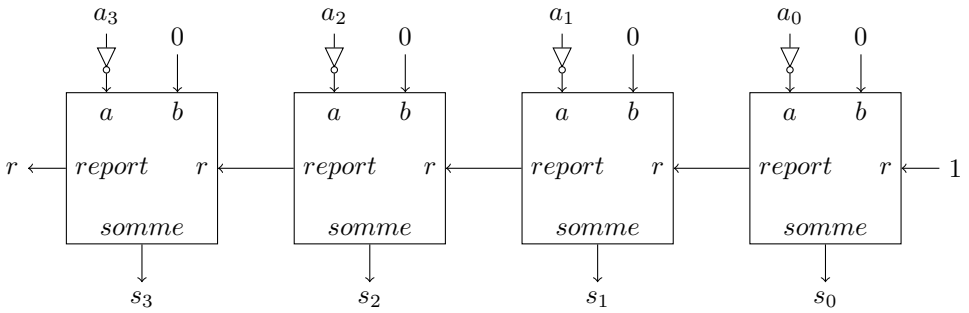

Fig. 5.4 – Calcul de l'opposé d'un quartet

circuit de la Fig. 5.5 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.

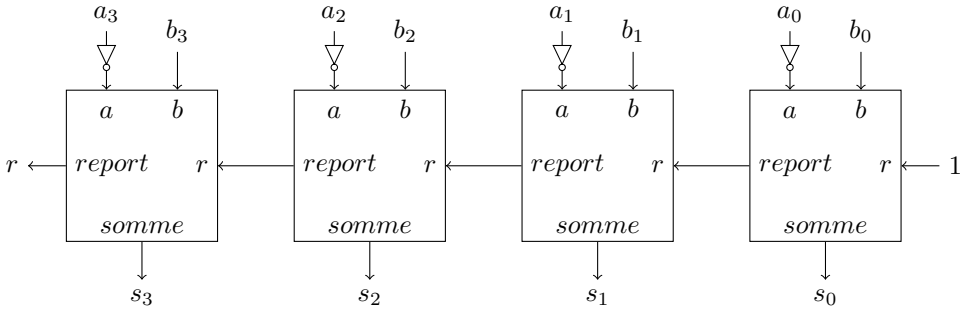

Fig. 5.5 – Soustraction : b-a

5.3.1 Exercices

1. Quel est le nombre décimal qui correspond au nombre binaire *1001100* ?
2. Quel est le nombre décimal qui correspond au nombre binaire *00001101110* ?
3. Comment peut-on facilement reconnaître si un nombre en notation binaire est :
   — pair
   — impair
4. Quels sont les plus petit et plus grand nombres entiers que l'on peut représenter en utilisant un nombre binaire sur 8 bits
5. Considérons le nombre binaire $N = B_{n-1}B_{n-2}...B_2B_1B_0$ en notation en complément à deux. Construisons le nombre de *n+1* bits baptisé *M* dans lequel on ajoute un bit de poids fort mis à la valeur *0*, c'est-à-dire $M = 0B_{n-1}B_{n-2}...B_2B_1B_0$. Quelle relation y-a-t-il entre les valeurs de *N* et *M* ?
   — $N > M$
   — $N < M$
   — $N = M$
   — $N \neq M$
6. Considérons le nombre binaire $N = B_{n-1}B_{n-2}...B_2B_1B_0$ en notation en complément à deux. Construisons le nombre encodé sur *n+1* bits *M* dans lequel on ajoute un bit de poids fort mis à la valeur *1*, c'est-à-dire $M = 1B_{n-1}B_{n-2}...B_2B_1B_0$. Quelle relation y-a-t-il entre *N* et *M* ?
   — $N > M$
   — $N < M$

— $N = M$
— $N \neq M$

5.4 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 5.1.

Tableau 5.1 – 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 le 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

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

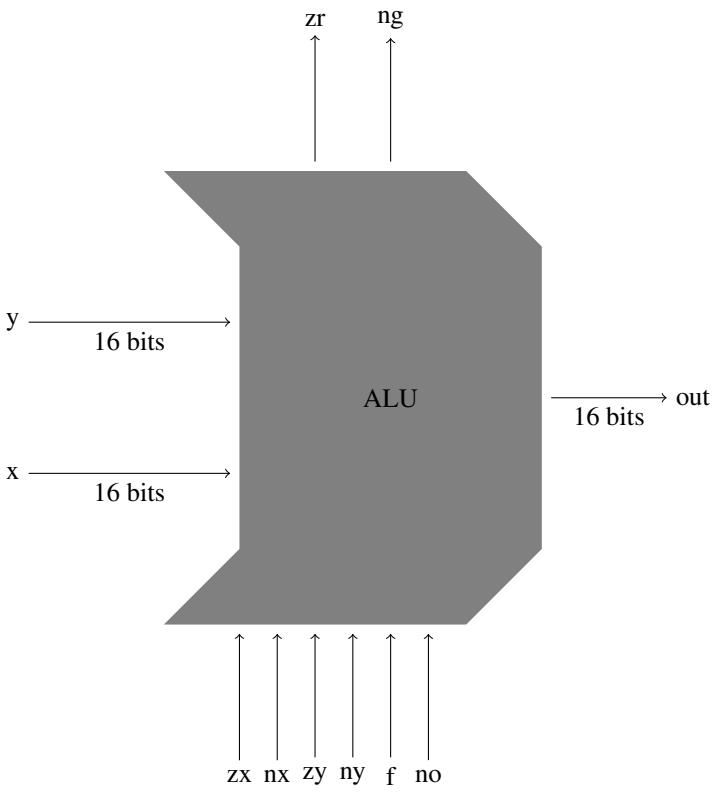

Fig. 5.6 – Unité Arithmétique et Logique (ALU)

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 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 6

Deuxième projet

Ce projet est à rendre par groupe de deux étudiants pour le lundi 26 octobre 2020 à 18h00 sur inginious.

1. Construisez un demi-additionneur sur un bit, https://inginious.info.ucl.ac.be/course/LSINC1102/HalfAdder
2. Construisez un additionneur complet sur un bit, https://inginious.info.ucl.ac.be/course/LSINC1102/FullAdder
3. Construisez un additionneur sur 16 bits, https://inginious.info.ucl.ac.be/course/LSINC1102/Add16
4. Construisez un circuit permettant d'incrémenter un nombre sur 16 bits, https://inginious.info.ucl.ac.be/course/LSINC1102/Inc16
5. Construisez l'ALU qui a été présentée en détails dans la section précédente, https://inginious.info.ucl.ac.be/course/LSINC1102/ALU

CHAPITRE 7

Compléments d'arithmétique

Avant d'aborder d'autres opérations arithmétiques que l'addition et la division, il est intéressant 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.

7.1 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 en utilisant 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 décimal, 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

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 2×37 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 123×456 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 123×101, 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. 7.1. 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. 7.1 – 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. 7.2 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 va

— le multiplicande
— le multiplicateur

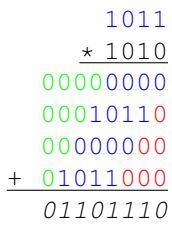

Fig. 7.2 – Une multiplication en notation binaire

— le produit intermédiaire

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.

7.1.1 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.

7.2 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 d'un 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. 7.3). 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

```
  1011
- 1000
    11
```

$Q_2 = 1$
reste = 011

Fig. 7.3 – Première étape de la division décimale

du quotient. Pour cela, nous essayons de soustraire $1 \times 10^1 \times diviseur$ du reste (Fig. 7.4). Le résultat étant négatif, 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. 7.4 – 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. 7.5). 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. 7.5 – 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. 7.6 retourne un résultat négatif. Cela indique que le bit Q_4 du quotient doit valoir *0*. La deuxième étape (Fig. 7.7) 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. 7.8) 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. 7.9) et Q_2 prend donc la valeur zéro. La dernière étape (Fig. 7.10) 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.

(suite sur la page suivante)

```
  00101110
- 01010000
    négatif
```

$Q_4 = 0$

Fig. 7.6 – Première étape de la division binaire

```
  00101110
- 00101000
  00000110
```

$Q_3 = 1$
reste = 00000110

Fig. 7.7 – Deuxième étape de la division binaire

```
  00000110
- 00010100
    négatif
```

$Q_2 = 0$

Fig. 7.8 – Troisième étape de la division binaire

```
  00000110
- 00001010
    négatif
```

$Q_1 = 0$

Fig. 7.9 – Quatrième étape de la division binaire

```
  00000110
- 00000101
  00000001
```

$Q_0 = 1$
Reste=00000001

Fig. 7.10 – Dernière étape de la division binaire

(suite de la page précédente)

```
def div(dividende,diviseur):
    quotient=0b0000
    reste = dividende
    diviseur = diviseur << 4
    for i in range(4+1):
        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 7.1 – Division par zéro en python

```
a = ...
b = ...
try:
  a / b
except ZeroDivisionError:
  print("Erreur")
```

7.3 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 π (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 scientifiques à laquelle vous avez étés habitués 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, 9.109×10^{-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 $1 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} = 1.75$

7.3.1 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 continent 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 $(-1)^{Signe} \times (1 + Fraction) \times 2^{Exposant-Biais}$.

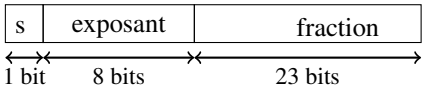

Fig. 7.11 – Notation IEEE 754 en simple précision

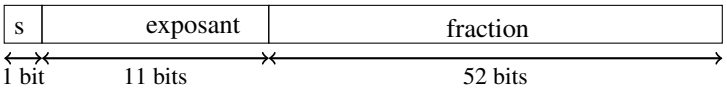

Fig. 7.12 – Notation IEEE 754 en double précision

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
```

7.3.2 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 2.789×10^{-1} sous la forme $x \times 10^2$. Notre addition est donc $9.998 \times 10^2 + 0.003 \times 10^2$ où 0.003×10^2 est l'arrondi de 2.789×10^{-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 10.001×10^2, soit 1.0001×10^3. 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
```

(suite sur la page suivante)

(suite de la page précédente)

```
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 8

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. 8.1 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

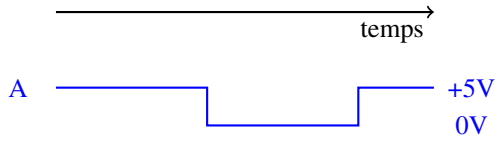

Fig. 8.1 – Signal électrique

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	μs	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. 8.2. 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. 8.3 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*. En analysant le signal de sortie (Fig. 8.3), on remarque que les délais différents pour

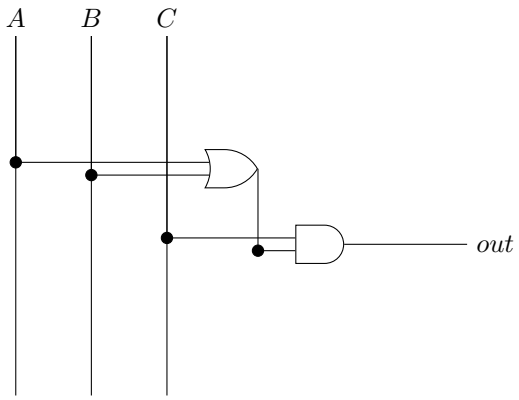

Fig. 8.2 – Un circuit simple à trois entrées

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.

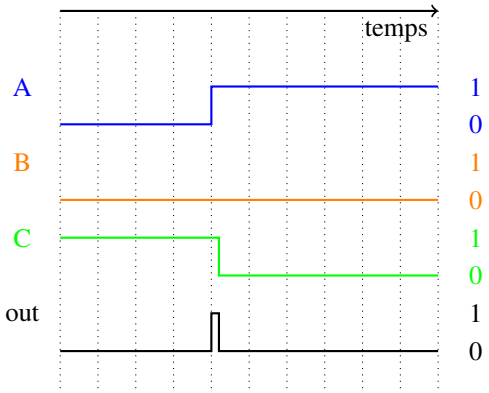

Fig. 8.3 – Evolution des signaux d'entrée et de sortie

8.1 Le signal d'horloge

Pour éviter ces problèmes, la plupart des ordinateurs utilisez 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. 8.4 présente un exemple de signal binaire périodique aussi appelé signal d'horloge. La période d'un signal périodique s'exprime en secondes. Souvent, plutôt que de donner la période du signal on préfère indiquer sa fréquence. La fréquence (f) d'un

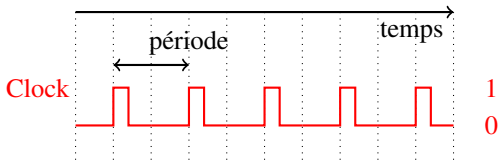

Fig. 8.4 – Signal d'horloge d'un ordinateur

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. 8.5. Grâce à ce signal d'horloge et

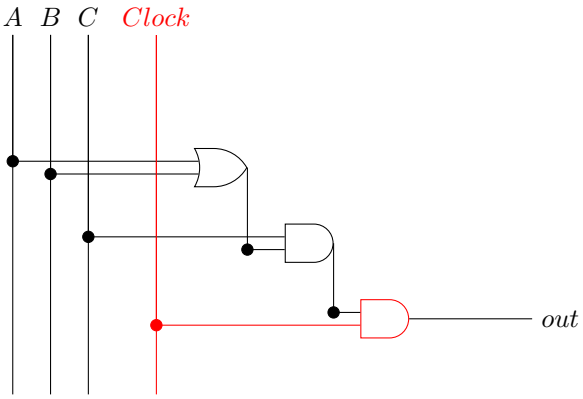

Fig. 8.5 – Un circuit simple à trois entrées contrôlé par une horloge

à la porte *AND* que nous avons ajouté, nous pouvons maintenant observer (Fig. 8.6) 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 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 ?

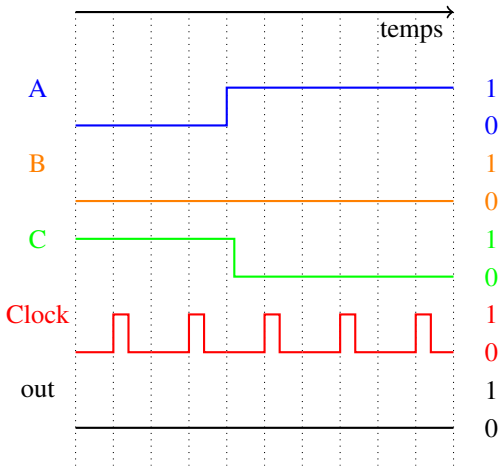

Fig. 8.6 – Evolution des signaux d'entrée et de sortie

8.2 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. 8.7. Pour comprendre le fonctionnement de ce circuit, il est intéressant d'analyser comment sa sortie évolue en fonction

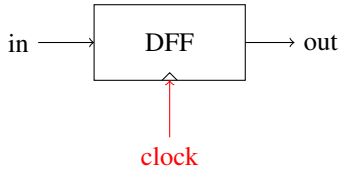

Fig. 8.7 – Un data flip-flop

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

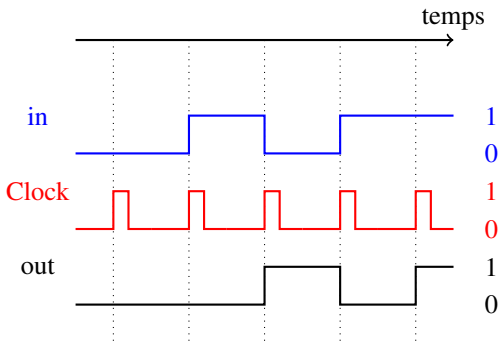

Fig. 8.8 – Data flip-flop - exemple

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. 8.9. Malheureusement, un tel circuit pose deux problèmes. Premièrement, puisque sa sortie dépend avec un délai

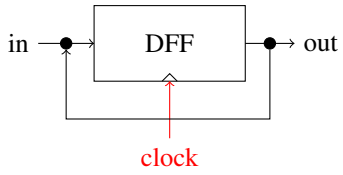

Fig. 8.9 – Un circuit pour mémoriser un bit ?

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 ...

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. 8.10. Cette mémoire d'un bit va jouer

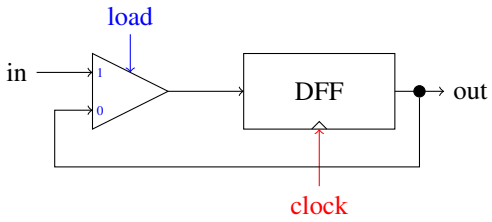

Fig. 8.10 – Un registre permettant de mémoriser un bit

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. 8.11). Dans la Fig. 8.11, le triangle

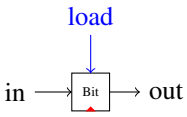

Fig. 8.11 – Une mémoire pour un bit

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.

8.3 Un registre pour mémoriser un quartet

Nous pouvons maintenant utiliser cet élément de mémoire pour construire une 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

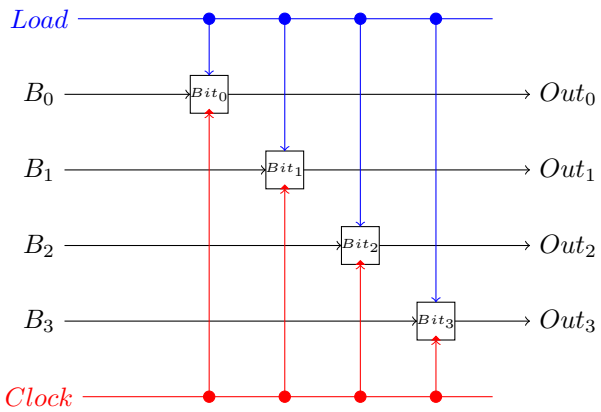

Fig. 8.12 – Un registre à 4 bits

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 octets. Ce bloc de registres comprend bien entendu quatre registres qui stockent chacun un octet. Outre le signal d'horloge (non représenté en Fig. 8.12), nous devons connecter le signal *load*, les 4 bits d'entrée et les 4 bits de sortie à cet semble 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

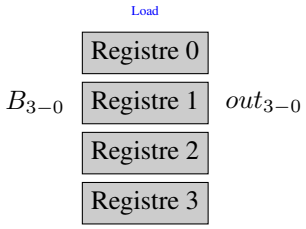

Fig. 8.13 – Eléments d'un registre à 4 bits

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. 8.12. 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 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. 8.14.

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. 8.14. Ce schéma général peut se reproduire

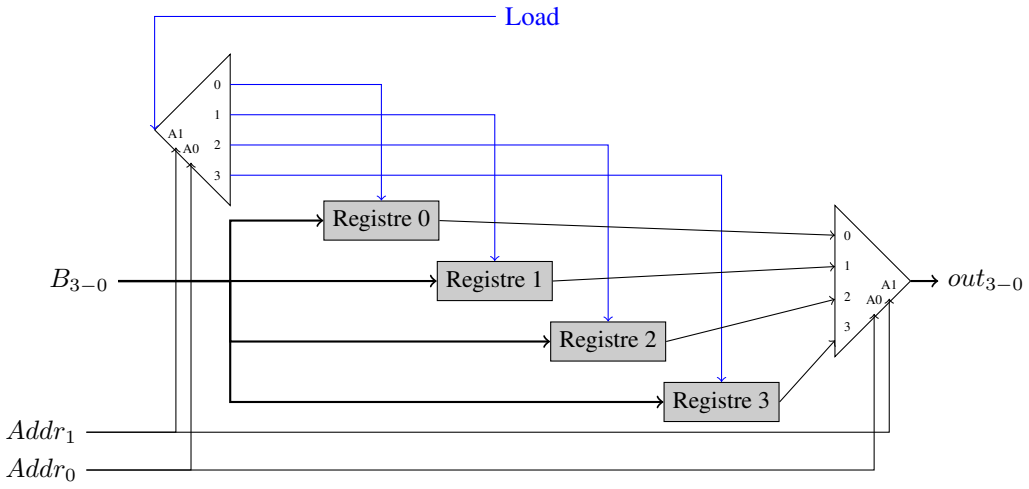

Fig. 8.14 – Un registre à 4 bits

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. 8.15. Grâce à ce bloc de quatre registres, nous pouvons

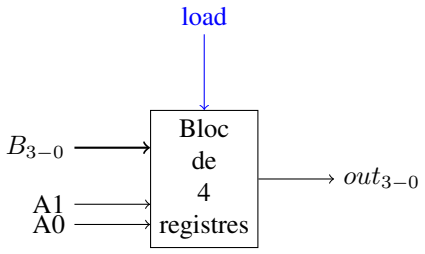

Fig. 8.15 – Représentation schématique d'un bloc de 4 registres

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. 8.16 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

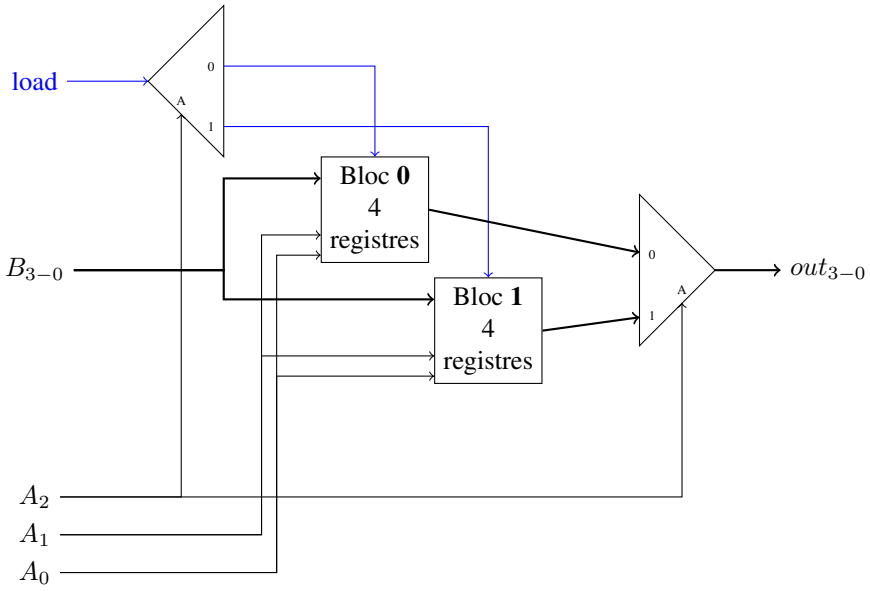

Fig. 8.16 – Un bloc de 8 registres

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.

8.3.1 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. 8.17 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 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

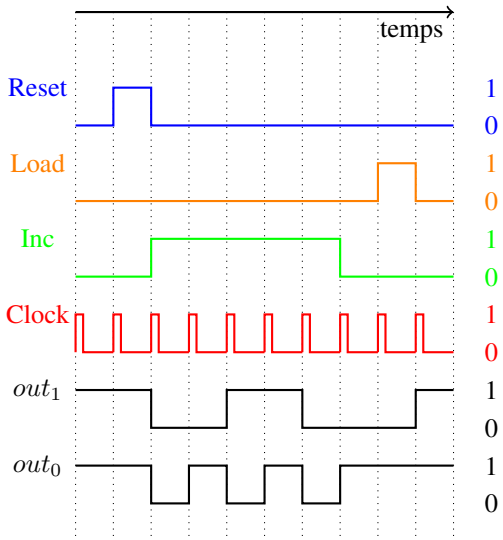

Fig. 8.17 – Evolution de la sortie du compteur en fonction du temps

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. 8.17, 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.

8.4 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. 8.18. Une caractéristique important des mémoires de type ROM est que leur contenu est préservé même lorsque la mémoire

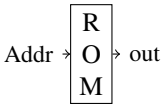

Fig. 8.18 – Une mémoire ROM

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. 8.19 représente une mémoire RAM de façon schématique.

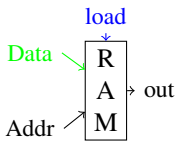

Fig. 8.19 – Une mémoire RAM

8.4.1 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. 8.20 résume cette évolution. La deuxième métrique que l'on peut utiliser pour comparer des mémoires est de regarder le débit auquel il 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 200, 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. 8.21). La dernière métrique

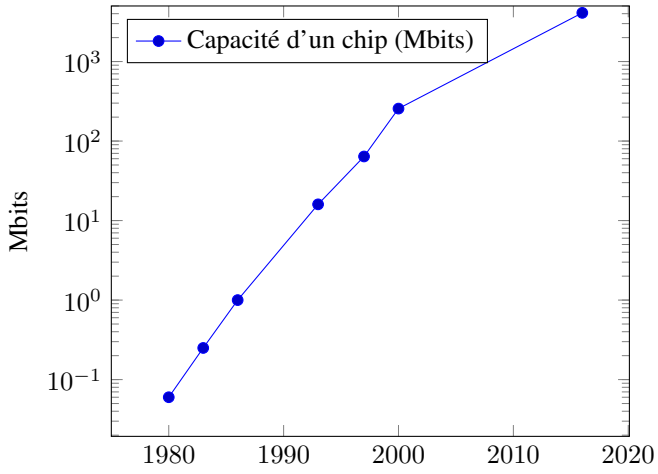

Fig. 8.20 – Evolution de la capacité des DRAMs

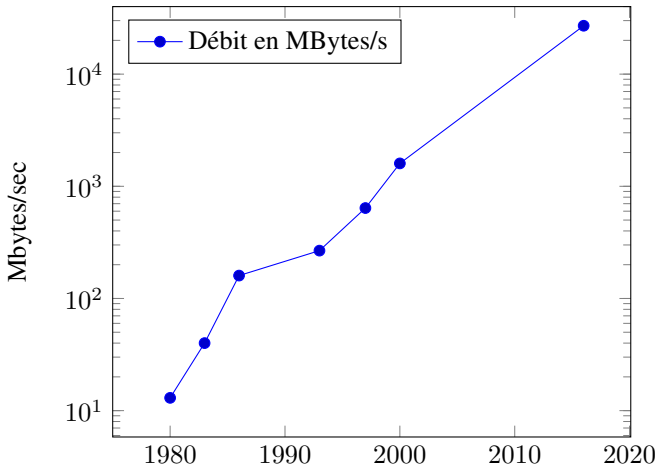

Fig. 8.21 – Evolution du débit des mémoires RAM

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. 8.22). 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.

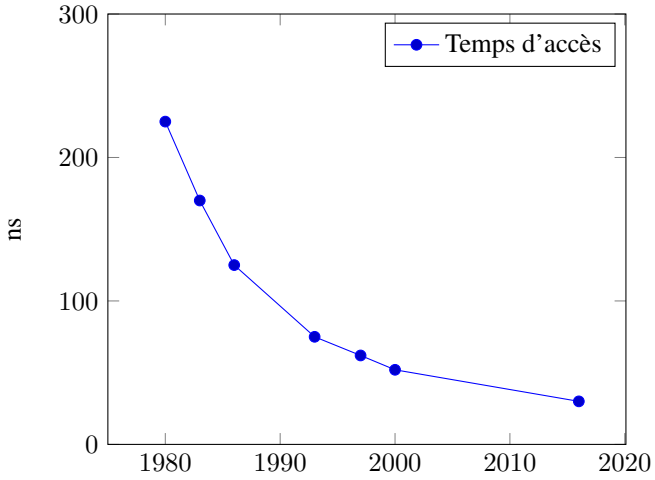

Fig. 8.22 – Evolution du temps d'accès des DRAMs

8.5 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

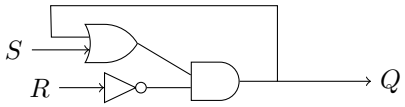

Fig. 8.23 – Représentation graphique d'un flip-flop RS AND-OR

utilise une port *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 flip-flop est le flip-flop 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.

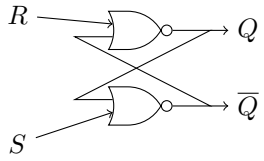

Fig. 8.24 – Représentation graphique d'un flip-flop SR utilisant des portes NOR

8.5.1 Exercices

1. Il est aussi possible de construire le flip-flop SR 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 ?

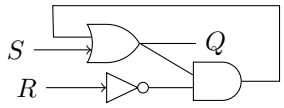

Fig. 8.25 – Variante du flip-flop SR AND-OR

2. Le flip-flop 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. 8.26). Expliquez le fonctionnement de ce circuit.

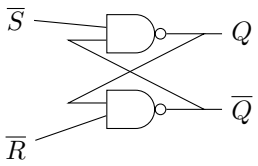

Fig. 8.26 – Représentation graphique d'un flip-flop SR utilisant des portes NOR

CHAPITRE 9

Troisième projet

Ce projet est à rendre par groupe de deux étudiants pour le lundi 9 novembre 2020 à 18h00 sur inginious.

1. Construisez un circuit permettant de stocker un bit, https://inginious.info.ucl.ac.be/course/LSINC1102/Bit
2. Construisez un circuit permettant d'implémenter un registre à 16 bits, https://inginious.info.ucl.ac.be/course/LSINC1102/Register
3. Construisez un circuit permettant de supporter une mémoire RAM comprenant 8 registres de 16 bits chacun, https://inginious.info.ucl.ac.be/course/LSINC1102/RAM8
4. Construisez un circuit permettant de supporter une mémoire RAM comprenant 64 registres de 16 bits chacun, https://inginious.info.ucl.ac.be/course/LSINC1102/RAM64
5. Construisez un circuit permettant de supporter une mémoire RAM comprenant 512 registres de 16 bits chacun, https://inginious.info.ucl.ac.be/course/LSINC1102/RAM512
6. Construisez un circuit permettant d'implémenter un compteur de programme, https://inginious.info.ucl.ac.be/course/LSINC1102/PC

CHAPITRE 10

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 Neuman.

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
— 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.

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. 10.1). A côté de ces deux

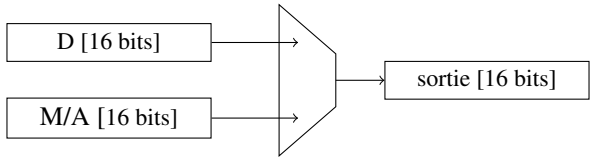

Fig. 10.1 – Les registres A, D et l'ALU

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.

10.1 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.

10.1.1 L'instruction de type A

L'instruction la plus simple du minuscule microprocesseur est l'instruction de type *A*. 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 sont vérifiées. 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.

10.1.2 L'instruction de type C

Cette instruction est l'instruction « à tout faire » du minuscule processeur. 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 10.1.

Tableau 10.1 – 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 (2) et la place 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 1 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 10.1

Code source 10.1 – 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 associe une adresse ç chaque variable du programme. Dans notre exemple, cette table des symboles pourrait être celle du Tableau 10.2.

Tableau 10.2 – 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 10.2 – 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 10.4)

Code source 10.3 – 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 10.4.

Code source 10.4 – 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 10.5.

Code source 10.5 – 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 10.6 initialise le contenu de la variable `x` à la valeur *1* et celui de la variable `y` à *-1*.

Code source 10.6 – 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 10.7 – 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 10.8).

Code source 10.8 – 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 10.9).

Code source 10.9 – 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 10.3).

Tableau 10.3 – Table des symboles

adresse	variable
21	A
22	B
23	C
25	X

Code source 10.10 – 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 10.11.

Code source 10.11 – 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`.

10.1.3 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
```

(suite sur la page suivante)

(suite de la page précédente)

```
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

10.2 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 vu 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* que les trois bits de poids faible ($j_1j_2j_3$) 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 10.11). Ce code contient quatre instructions. Il est stocké dans la mémoire d'instructions (Tableau 10.4).

Tableau 10.4 – 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 fort 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 un variable en mémoire (Code source 10.8). 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 10.5 – 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 10.5 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
```

10.3 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 10.6 – Calcul de la valeur absolue en minuscule assembleur

adresse	instruction
41	@22 // x
42	D=M
43	@23 // y
44	DM=D
45	@49
46	D ;JLT
47	@51
48	0 ;JMP
49	@23
50	M=-D
51	@24 // z
51	M=0

Vous pouvez télécharger cet exemple depuis `asm/ex11a.asm`.

Il est intéressant d'analyser l'exécution du programme du Tableau 10.5 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`. Ces 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 10.7).

Tableau 10.7 – 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 10.5 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 instruction 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 10.8.

Tableau 10.8 – 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 10.9).

Tableau 10.9 – 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 10.9, 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 10.9) 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.

10.3.1 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
```

10.4 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 10.10.

Tableau 10.10 – 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`.

10.4.1 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):
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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
```

(suite sur la page suivante)

(suite de la page précédente)

```
      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 11

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. 11.1) 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.

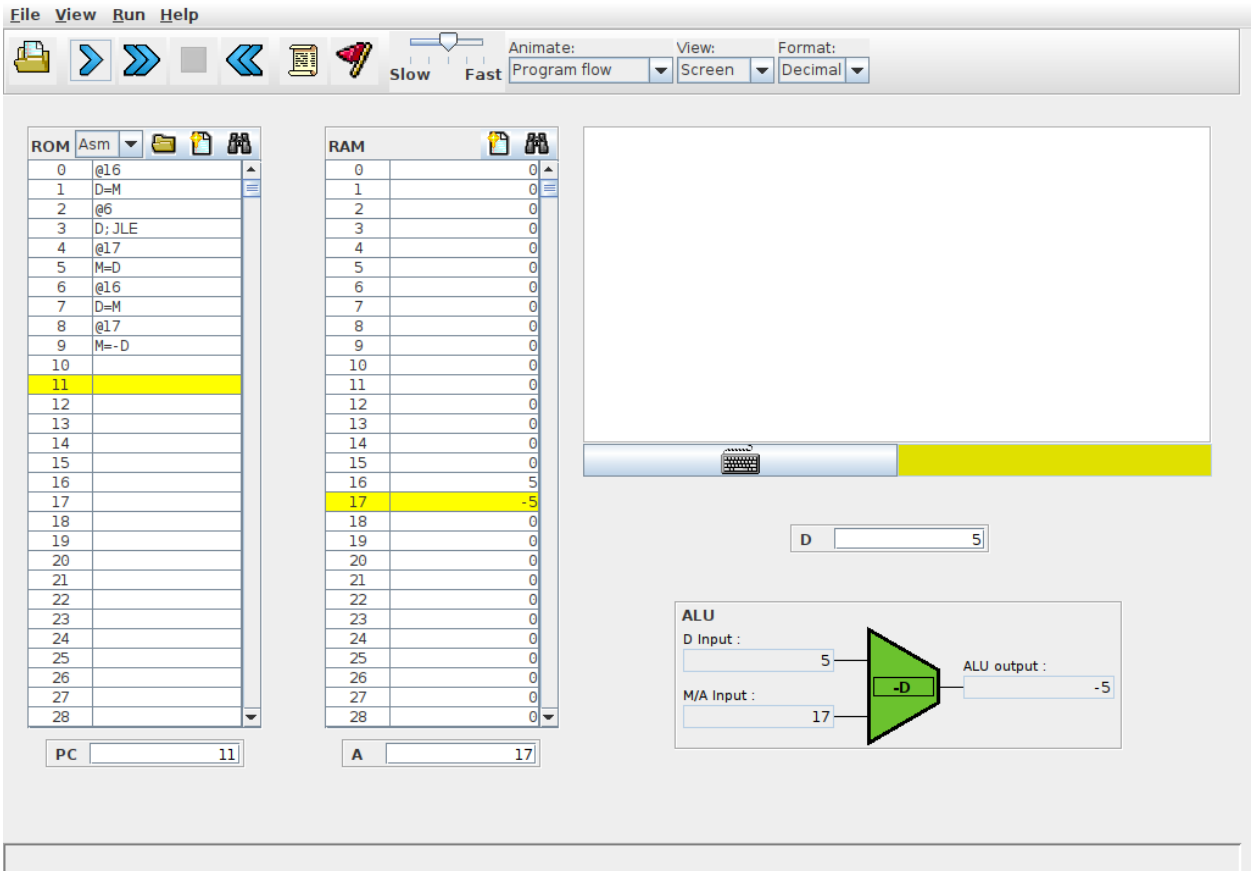

Fig. 11.1 – Simulateur interactif du minuscule processeur

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.

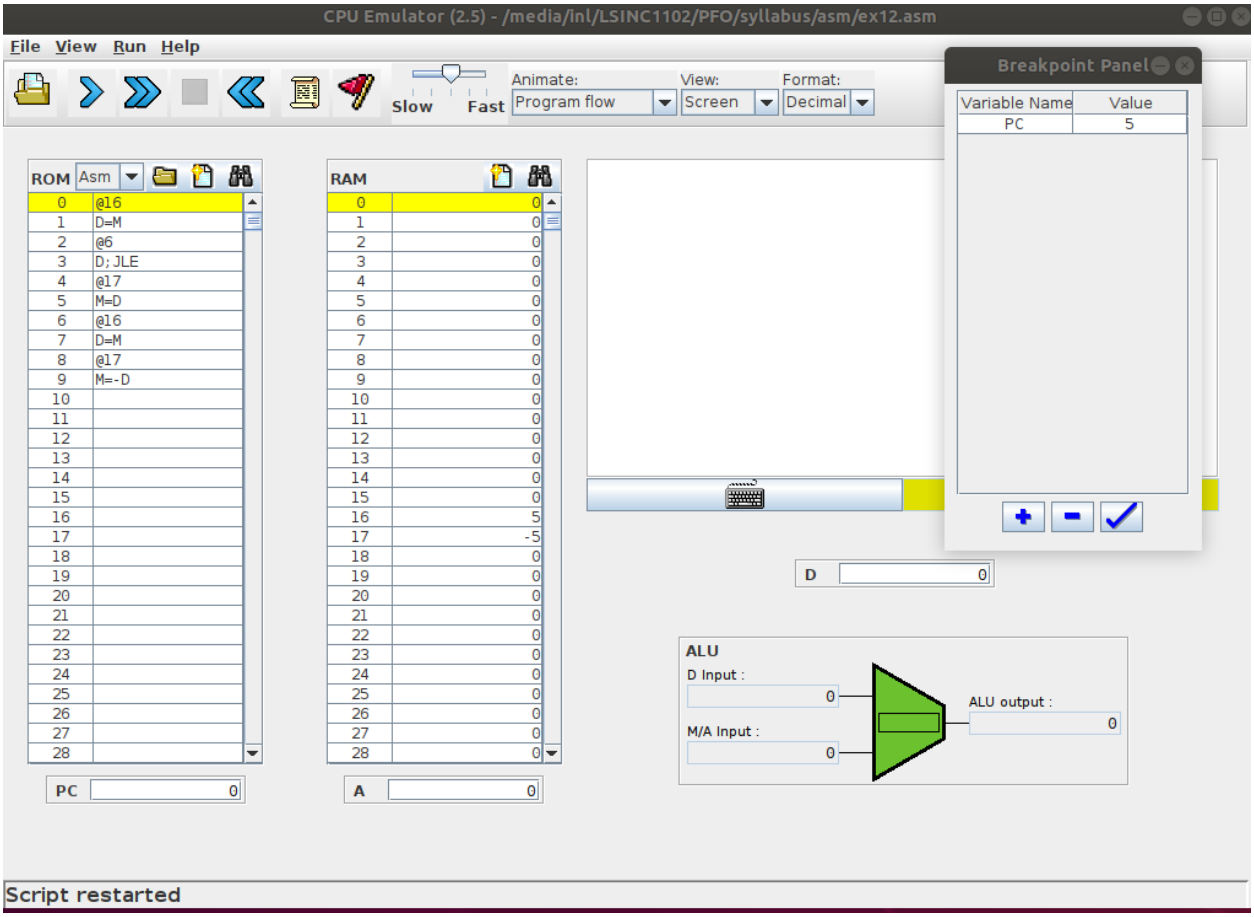

Fig. 11.2 – Breakpoints avec le simulateur interactif du minuscule processeur

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 calcule 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 12

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.

12.1 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 12.1.

Tableau 12.1 – 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. 12.1 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

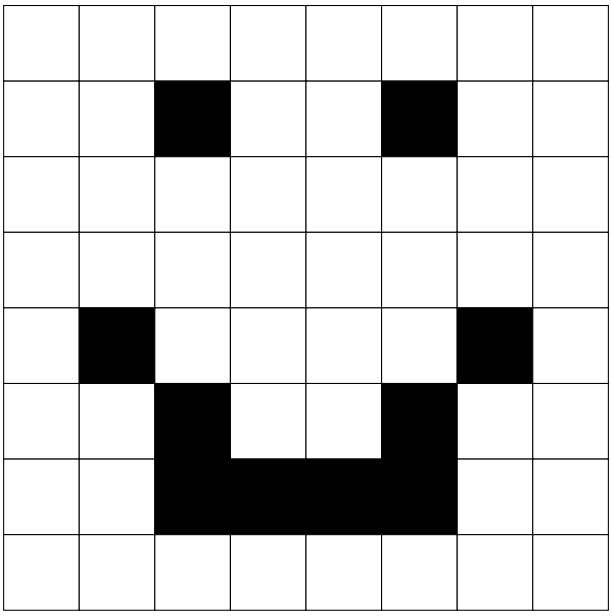

Fig. 12.1 – Un caractère sous la forme de pixels

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
```

12.2 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-u doivent pouvoir être stockés en mémoire.

Commençons par analyser la façon dont un programme peut manipuler les coordonnées *(x,y)* d'un pixel à l'écran. Ces coordonnées sont toutes les deux représentées sous la forme d'un nombre entier. Une première approche serait d'associer une variable pour l'abscisse et une autre pour l'ordonnée. Malheureusement cette solution nous force à définir un très grand nombre de variables. Une autre possibilité est de dire que ces coordonnées constituent une paire d'entiers et que cette paire peut être stockée en mémoire en utilisant deux adresses consécutives. Par exemple, on peut prendre la convention que l'adresse `C` contiendra la valeur de l'abscisse tandis que l'adresse `C+1` contiendra la valeur de l'ordonnée. La Tableau 12.2 présente deux de ces coordonnées en mémoire. La première, *(3,7)* est stockée aux adresses `16` et `17`. La seconde, *(6,4)* occupe les adresses `18` et `19`.

Tableau 12.2 – Contenu de la mémoire

adresse	valeur
16	3
17	7
18	6
19	4

Sur base de cette représentation, on peut écrire un petit programme assembleur qui permet de vérifier si deux coordonnées sont identiques.

```
        @16
        D=M
        @18
        D=D-M
        @DIFF
        D;JNE
        @16
        A=A+1
        D=M
        @18
        A=A+1
        D=D-M
        @DIFF
        D;JNE
(EGAL)
        @20
        M=1
        @FIN
        0;JMP
(DIFF)
        @20
        M=0
(FIN)
```

Vous pouvez télécharger ce programme via `asm/coord-eq.asm`.

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 le Tableau 12.3.

Tableau 12.3 – Tableau contenant le nombre de jours dans chaque mois de l'année

adresse	valeur
m+0	31
m+1	28
m+2	31
m+3	30
m+4	31
m+5	30
m+6	31
m+7	31
m+8	30
m+9	31
m+10	30
m+11	31

Sur base de ce tableau de douze nombres, on peut ensuite facilement écrire une programme qui calcule le nombre de jours durant une année en additionnant les nombres présents dans ce tableau. Ce programme est téléchargeable depuis `asm/mois-annee.asm`.

```
        @16   // somme
        M=0
        @17   // i variable boucle
        M=0
(BOUCLE)
        @20   // début tableau
        D=A
        @17
        A=M+D
        D=M   // charge i ème élément du tableau
        @16
        M=D+M
        @17
        M=M+1
        @12
        D=A
        @17
        D=D-M
        @BOUCLE
        D;JGT
```

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
```

Notre programme en assembleur contient une construction qui mérite d'être analysée plus en détails.

```
@20   // début tableau
D=A
@17   // variable contenant i
A=M+D
D=M   // charge le i ème élément du tableau
```

Tout d'abord, les deux premières lignes permettent de placer dans le registre `A` l'adresse du début de notre tableau. L'adresse `17` est celle de notre variable de boucle. L'instruction suivante, `A=M+D`, calcule une adresse en mémoire (le résultat est stocké dans le registre `D`). Cette adresse est la somme entre la valeur actuelle du registre `D`, c'est-à-dire l'adresse du début de notre tableau, et la valeur qui a été lue en mémoire à l'adresse `17`, c'est-à-dire la valeur de notre compteur. Nous plaçons donc dans le registre `A` l'adresse du *ième* élément de notre tableau. L'instruction `D=M` qui suit nous permet donc de charger la valeur de cet élément dans le registre `D`. Cette valeur peut ensuite être utilisée dans les calculs.

Une construction similaire peut être utilisée pour initialiser à `0` ou `1` la valeur du *ième* élément du tableau.

```
@20   // début tableau
D=A
@17   // variable contenant i
A=M+D
M=1   // initialise le i ème élément du tableau
```

Si l'on veut modifier la valeur du *ième* élément d'un tableau, il faut procéder en deux étapes. Il faut d'abord calculer l'adresse de cette élément et la stocker dans une adresse mémoire temporaire. Ensuite, on peut modifier la valeur de l'élément se trouvant à cette adresse.

De façon générale, si un tableau d'entier démarre à l'adresse A, alors le *ième* élément de ce tableau se trouve en mémoire à l'adresse $A+i$. 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. 12.2. La façon

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

Fig. 12.2 – 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. 12.3. 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 $A + i \times c + j$ en supposant que les indices des lignes et colonnes commencent à *0*. Il est aussi possible de stocker cette

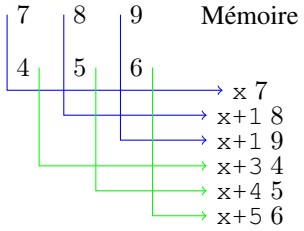

Fig. 12.3 – Stockage ligne par ligne d'une matrice

matrice colonne par colonne comme représenté dans la Fig. 12.4. 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 colonnes commencent à *0*. On est parfois amener à manipuler des tableaux de différentes tailles. Dans ce cas, il est

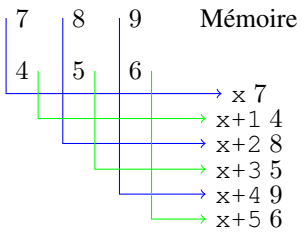

Fig. 12.4 – Stockage colonne par colonne d'une matrice

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 12.4).

Tableau 12.4 – Tableau dont le premier élément est sa taille

adresse	valeur
m	12
m+1	31
m+2	28
m+3	31
m+4	30
m+5	31
m+6	30
m+7	31
m+8	31
m+9	30
m+10	31
m+11	30
m+12	31

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

```
        @16   // somme
        M=0
        @17   // i variable boucle
        M=0
(BOUCLE)
        @19   // début tableau
        D=A+1 // on veut le premier élément, pas la taille
        @17
        A=M+D
        D=M   // charge i ème élément du tableau
        @16
        M=D+M
        @17
        M=M+1
        @19
        D=M   // taille du tableau
        @17
        D=D-M
```

(suite sur la page suivante)

(suite de la page précédente)

```
        @BOUCLE
        D;JGT
```

Ce programme est téléchargeable depuis `asm/mois-annee2.asm`.

De plus, il est facile dans un programme ou un langage de programmation de vérifier que les accès aux éléments d'un tableau respectent bien les limites de ce tableau.

12.3 Utilisation des chaînes de 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 donc composée de caractères consécutifs qui sont stockés en mémoire. Un programme peut être amené à 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. 12.5). 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. A titre d'exemple, considérons un petit programme

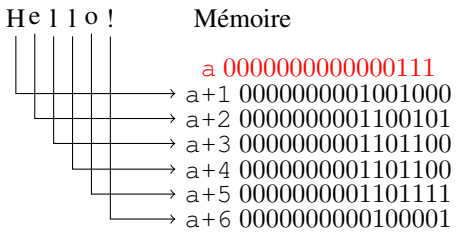

Fig. 12.5 – Représentation en mémoire de la chaîne de caractères Hello ! avec une indication explicite de longueur

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

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

La conversion de ce programme en minuscule assembleur est présentée ci-dessous. Vous pouvez le télécharger via `asm/charin.asm`.

Ce programme a comme entrée la variable `c` et une chaîne de caractères qui est stockée en mémoire à partir de l'adresse `29`. Le résultat du programme se retrouve dans la variable `r` en mémoire.

```
    @104 // caractère à tester
    D=A
    @c
```

(suite sur la page suivante)

(suite de la page précédente)

```
     M=D
     @i
     M=0
     @r
     M=0
(BOUCLE)
     @i
     D=M
     @29    // longueur
     D=D-M  // len(str) -i
     @FIN
     D;JGE
     @29
     D=A+1  // i+1 car la première adresse contient longueur
     @i
     A=M+D  // adresse de str[i]
     D=M
     @c
     D=M-D  // str[i] - c
     @TROUVE
     D;JEQ
     @i
     M=M+1
     @BOUCLE
     0;JMP
(TROUVE)
     @r
     M=1
(FIN)
```

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. Avec cette représentation des chaînes de caractères, le programme ne connaît

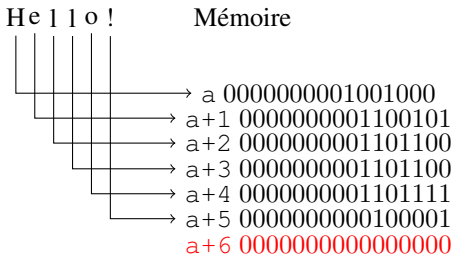

Fig. 12.6 – Représentation en mémoire de la chaîne de caractères Hello ! avec mrqueur de fin

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`. En python, le parcours de cette chaîne peut se faire en utilisant le programme ci-dessous.

```
str="Hello!\0"
c='o'
r=0 # absent
i=0
while(str[i]!='\0'):
    if str[i]==c:
```

(suite sur la page suivante)

(suite de la page précédente)

```
        r=1 # present
        break
    i=i+1
```

La conversion de ce programme en minuscule assembleur est présentée ci-dessous. Vous pouvez le télécharger via `asm/charin-c.asm`.

```
     @104 // caractère à tester
     D=A
     @c
     M=D
     @i
     M=0
     @r
     M=0
(BOUCLE)
     @30 // string
     D=A
     @i
     A=M+D  // adresse de str[i]
     D=M
     @FIN
     D;JEQ
     @c
     D=M-D  // str[i] - c
     @TROUVE
     D;JEQ
     @i
     M=M+1
     @BOUCLE
     0;JMP
(TROUVE)
     @r
     M=1
(FIN)
```

CHAPITRE 13

Quatrième projet

L'objectif de ce quatrième projet, qui se fera de façon individuelle, est de démontrer votre connaissance de la programmation en langage d'assemblage. Vous devrez écrire deux petits programmes dans ce langage pour le lundi 30 novembre 2020 à 18h00. Ce projet vaut trois points et sera le dernier projet côté pour le cours cette année.

1. Implémentez un programme en langage d'assemblage qui permet de calculer le résultat de la multiplication entre deux naturels. Ce programme est à déposer sur inginious : https://inginious.info.ucl.ac.be/course/LSINC1102/MultAsm Cet exercice compte pour un point sur les trois points de ce projet.
2. Vous avez maintenant appris les bases vous permettant d'écrire un petit programme en langage d'assemblage. En utilisant ces connaissances, soyez créatifs et proposez un programme non trivial qui traite des données sous la forme d'un tableau, d'une matrice ou de chaînes de caractères. Le type de traitement est laissé à votre choix. Inventez un traitement qui est utile et écrivez d'abord un petit programme python qui réalise un traitement de votre choix. Écrivez un script de test qui permettra de valider votre programme. Ensuite, traduisez ce programme en minuscule langage d'assemblage et utilisez votre script de test pour démontrer le bon fonctionnement de votre programme. Justifiez vos choix éventuels dans les commentaires de votre programme et du script de test. Cette seconde partie du projet comptera pour 2 points, le programme et le script de test auront le même poids.

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.

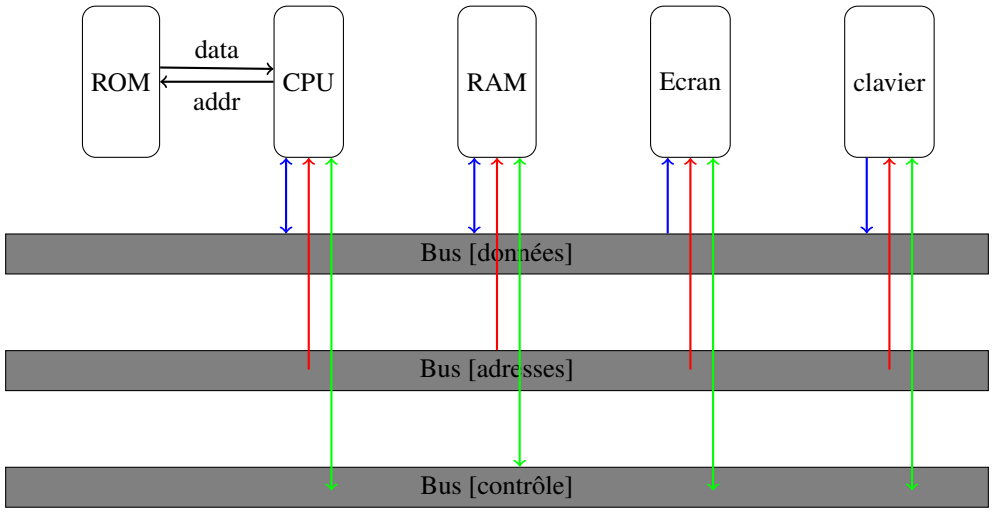

Fig. 14.1 – Architecture du minuscule ordinateur

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. Une 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

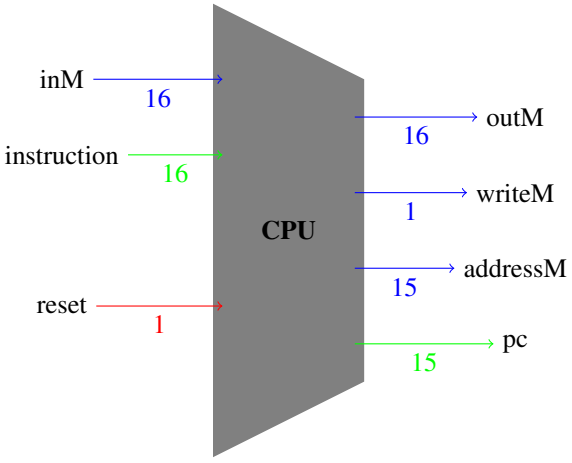

Fig. 14.2 – Minuscule CPU

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

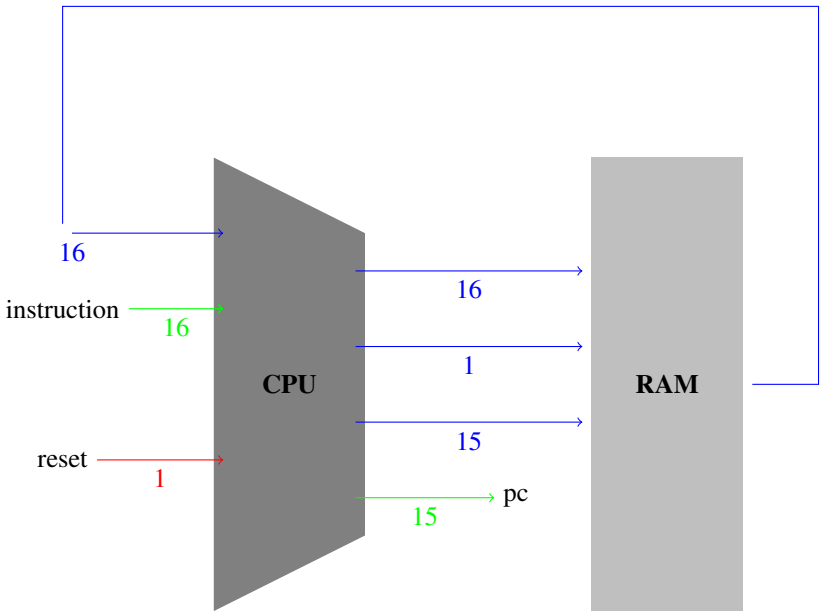

Fig. 14.3 – Connexions entre le minuscule CPU et la RAM

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 exécution 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 :

- 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

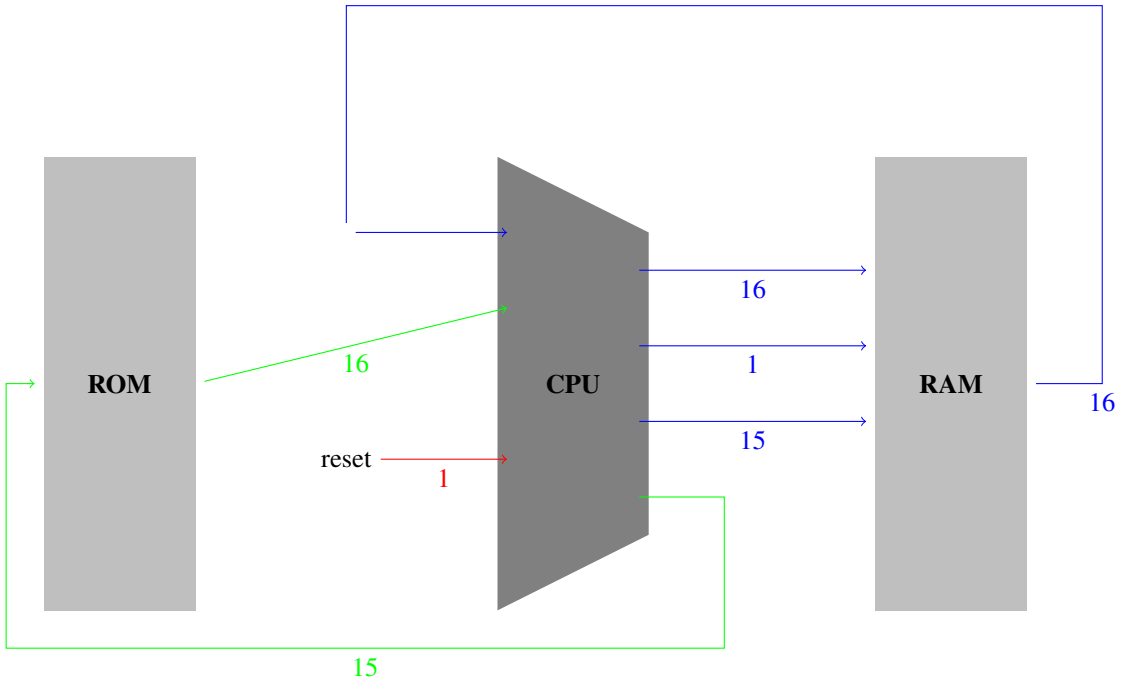

Fig. 14.4 – Connexions entre le minuscule CPU et les mémoires

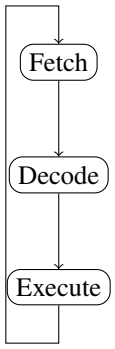

Fig. 14.5 – Le cycle Fetch-Decode-Execute

— 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 *zr* 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 présenté dans les chapitres précédents. La Fig. 14.6 présente un schéma bloc de notre minuscule CPU que nous allons compléter petit à petit.

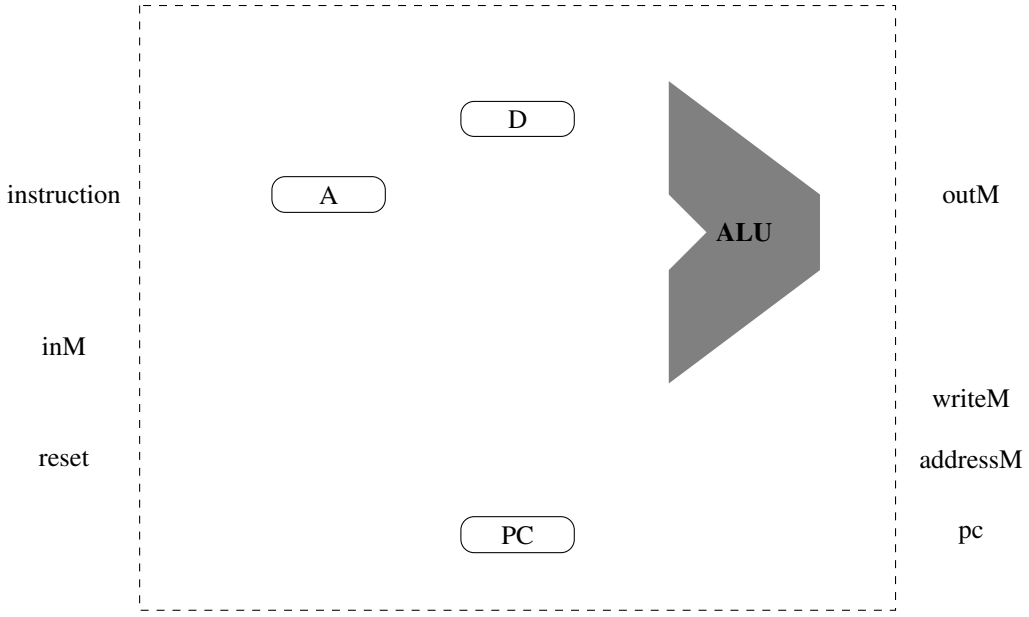

Fig. 14.6 – Composition du minuscule CPU

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.

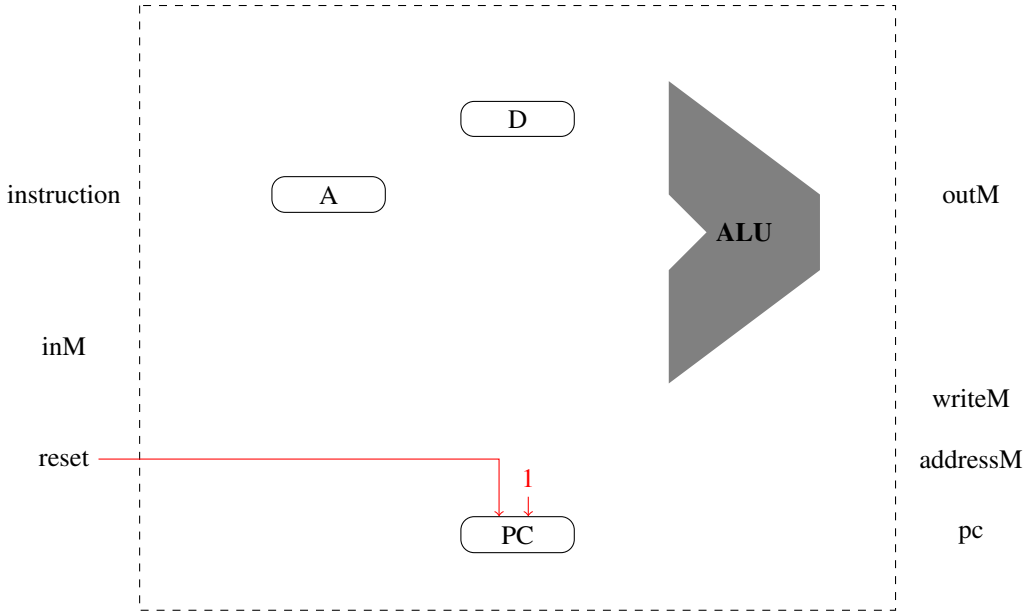

Fig. 14.7 – Un registre PC de base

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 *in* du registre `A` à *1* lorsque le bit de poids faible 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 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`

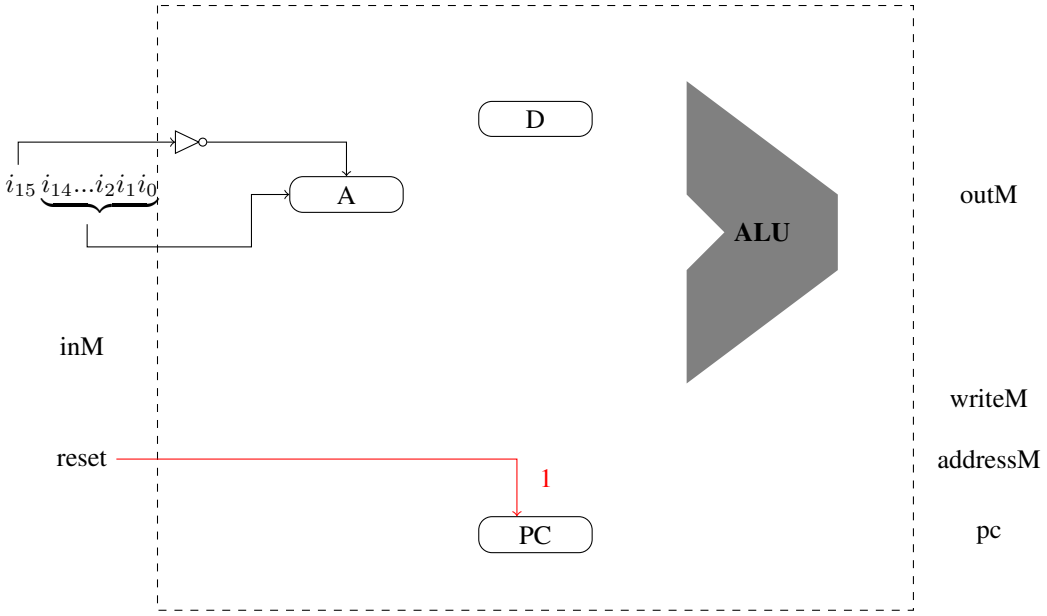

Fig. 14.8 – Support de l'instruction de type A

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.

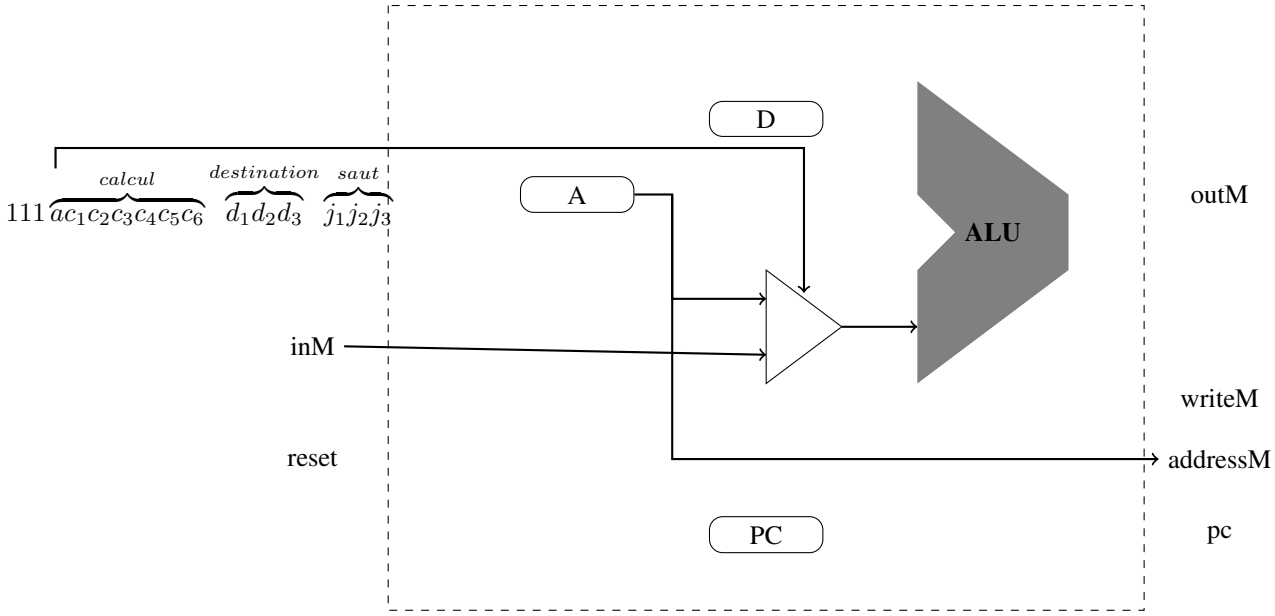

Fig. 14.9 – Utilisation du bit a des instructions de type C

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.

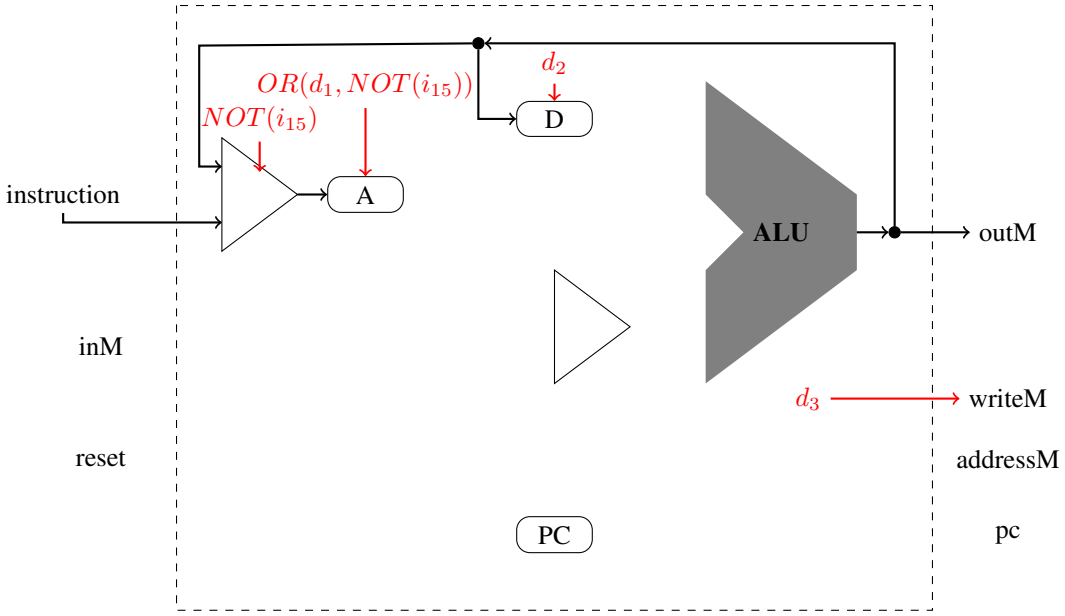

Fig. 14.10 – Choix de la destination du calcul de l'ALU

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

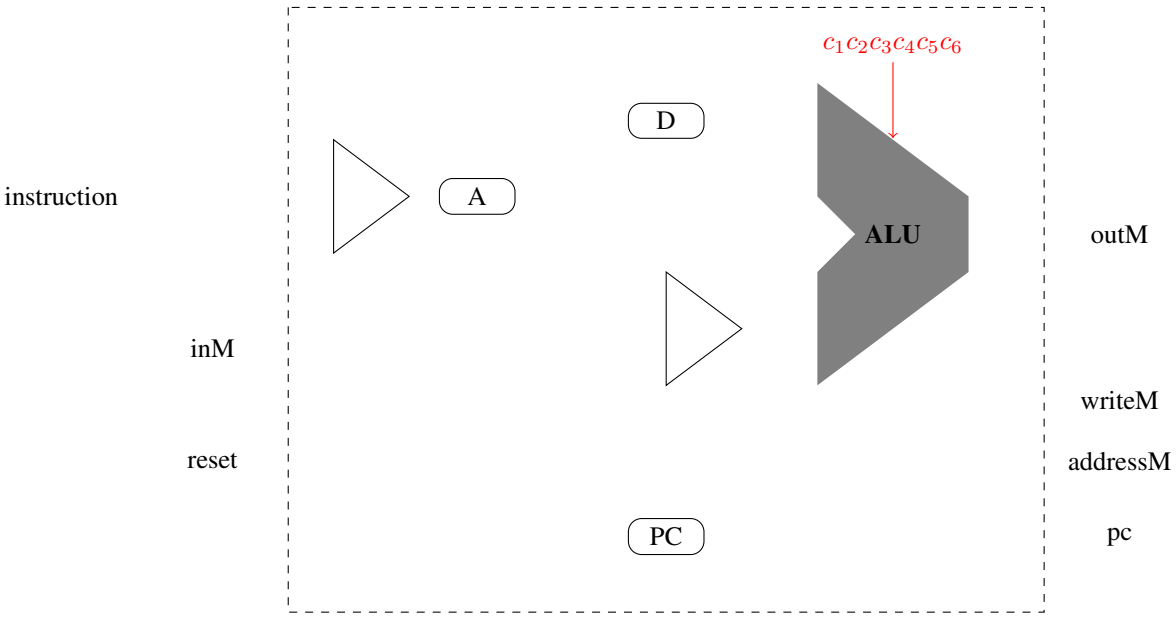

Fig. 14.11 – Connexion des bits de calcul de l'instruction à la minuscule ALU

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 su 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'une 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 difficultés 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.

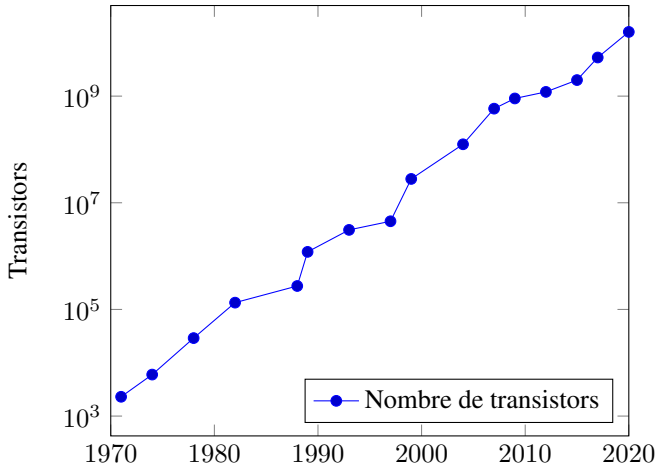

Fig. 15.1 – Evolution du nombre de transistors contenus dans les processeurs commerciaux

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

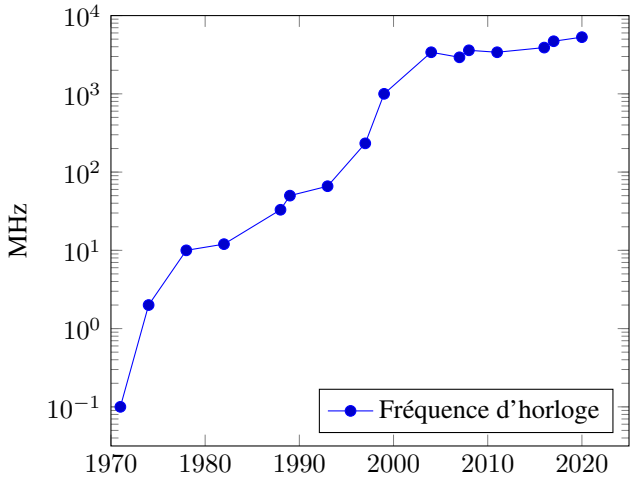

Fig. 15.2 – Evolution de la fréquence d'horloge des processeurs intel

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 programme 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 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 a 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

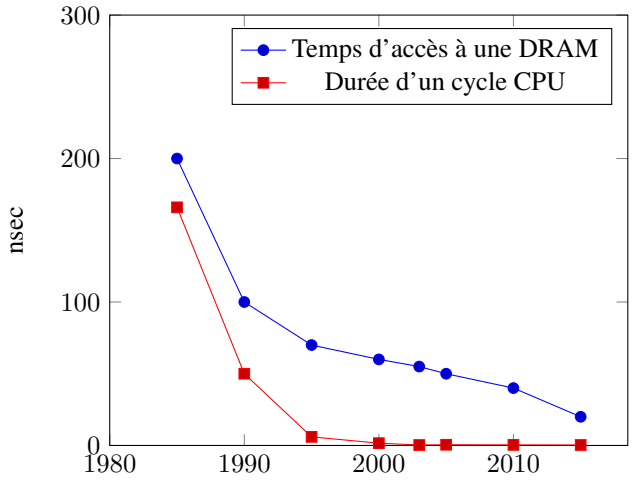

Fig. 15.3 – Au fil des années, le gap entre la durée d'un cycle CPU (en nsec) et un temps d'accès à la DRAM n'a fait qu'augmenter

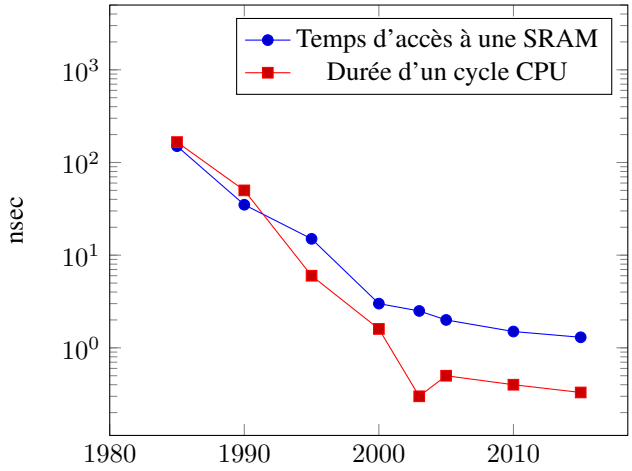

Fig. 15.4 – Gap entre la durée d'un cycle CPU (en nsec) et un temps d'accès à la SRAM

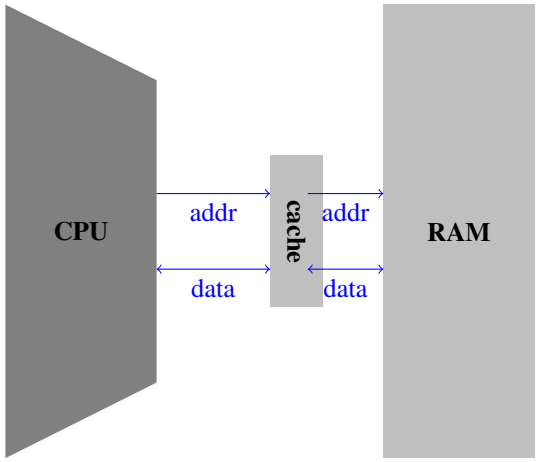

Fig. 15.5 – La cache s'interpose entre le CPU et la mémoire RAM

que la mémoire, plus lente, retourne la donnée demandé 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

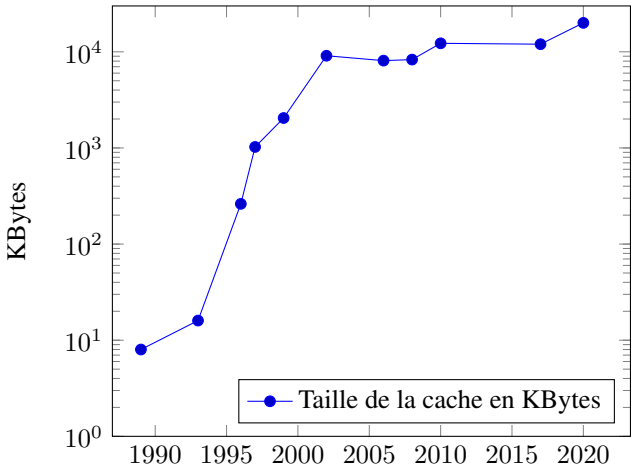

Fig. 15.6 – Evolution de la taille totale de la cache sur les processeurs intel

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.

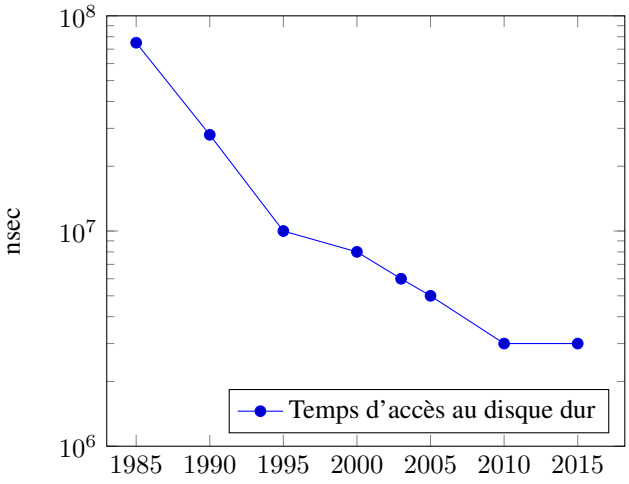

Fig. 15.7 – Evolution du temps d'accès aux disques durs

CHAPITRE 16

Fonctions en assembleur

Le minuscule ordinateur 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 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 du minuscule CPU.

Commençons par réfléchir aux différentes opérations qu'un langage de programmation tel que python doit réaliser pour supporter différents types de fonctions. Commençons par analyser 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 simplement 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 une telle procédure dans un programme. Regardons plus en détails le code ci-dessous. La première ligne initialise la variable `a` 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 `a`. 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 important 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.

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

En python, il est facile d'imprimer de l'information à l'écran. En minuscule 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'un variable globale est illustrée dans le programme python ci-dessous.

```
compteur=0

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

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

La variable `compteur` est une variable globale (python impose l'utilisation du mot clé `global` dans la définition de la procédure `compte`) qui est modifiée dans la procédure `compte`. Analysons l'exécution de ce programme pas à pas. Ce programme manipule deux variables en mémoire : `a` et `compteur`. La première ligne initialise la variable `a` 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 `a` 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 `a`.

Pour bien comprendre cette différence lors de l'exécution des deux procédures, il est intéressant d'analyser comment il est possible d'implémenter un tel programme en utilisant le minuscule assembleur.

Commençons par assigner une zone mémoire pour la variable `a`, par exemple l'adresse `16`. Nous pouvons ensuite écrire en assembleur les lignes impaires qui correspondent aux différentes assignations de cette variable.

```
// ligne 1
@1
D=A
@a
M=D
// à compléter
// ligne 3
@2
D=A
@a
M=D
// à compléter
// ligne 5
@1
D=A
@a
M=D
```

Nous devons également assigner une zone mémoire pour stocker la variable `compteur`. Supposons que celle-ci soit stockée à l'adresse `17`. La Fig. 16.1 présente le contenu initial de notre mémoire. Il nous faut maintenant exécuter la procédure `compte()` après l'exécution des lignes `1` et `3`. Le corps de cette procédure peut s'écrire en deux instructions en minuscule assembleur.

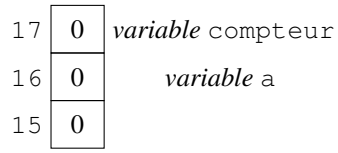

Fig. 16.1 – Contenu de la mémoire

```
@compteur
M=M+1
```

Une première approche pour intégré 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.

```
   // ligne 1
   @1
   D=A
   @a
   M=D
   // ligne 2
   @compteur    // procédure compte
   M=M+1        // procédure compte
   // ligne 3
   @2
   D=A
   @a
   M=D
   // ligne 4
   @compteur    // procédure compte
   M=M+1        // procédure compte
   // ligne 5
   @3
   D=A
   @a
   M=D
```

Ce programme est téléchargeable depuis `asm/procedure-ex1.asm`.

Cette approche fonctionne dans notre exemple simple, mais elle a deux inconvénient majeurs. Le premier est que le code de la procédure est recopié plusieurs fois en mémoire. Cela consomme inutilement de l'espace mémoire surtout si le programme appelle la procédure à de nombreuses reprises. Le deuxième inconvénient est que si la procédure modifie le contenu des registres `A` ou `D`, elle pourrait avoir un impact non-voulu sur les instructions du programme principal.

Il serait préférable 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.

```
(LIGNE1)
   @1  // ligne 1
   D=A
   @a
   M=D
   @COMPTE // ligne 2
   0;JMP
```

(suite sur la page suivante)

(suite de la page précédente)

```
(LIGNE3)
   @2  // ligne 3
   D=A
   @a
   M=D
   // ligne 4
   @COMPTE
   0;JMP
(LIGNE5)
   @3    // ligne 5
   D=A
   @a
   M=D
(COMPTE)
   @compteur
   M=M+1
   @retour
   0;JMP
```

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. Celle-ci doit pouvoir retourner, via une instruction `JMP` à l'adresse de l'instruction qui suit celle à partir de laquelle est a été appelée. Malheureusement, il n'est pas possible d'extraire cette information des registres du minuscule CPU durant l'exécution de notre procédure. Le programme appelant devra donc calculer cette adresse, souvent appelée *adresse de retour* et la sauvegarder à un endroit où la procédure peut la récupérer. Sur le minuscule ordinateur, le seul endroit où nous pouvons stocker de l'information est dans la mémoire. Une solution simple est de réserver une adresse en mémoire pour sauver cette adresse de retour. Dans ce cas, l'appel à un 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

Il nous reste à traduire cela en minuscule assembleur. L'appel à une procédure peut se réaliser en utilisant les instructions suivantes :

```
// juste avant l'appel à la procédure
  @SUIVANT  // sauvegarde de l'adresse qui suit l'appel
  D=A       // ...
  @retour   // ...
  M=D             // dans la zone mémoire "retour"
  @COMPTE   // appel de la procédure
  0;JMP     // suite
(SUIVANT)   // adresse de l'instruction qui suit l'appel à la procédure
```

La fin de la procédure doit également être adaptée. Il faut d'abord charger l'adresse de retour depuis la mémoire avant d'exécuter le `JMP`. Cela se fait en utilisant la séquence d'instructions suivante :

```
(COMPTE)
  @compteur
  M=M+1
  @retour   // adresse où est stockée l'adresse de retour
```

(suite sur la page suivante)

(suite de la page précédente)

```
  A=M       // chargement de l'adresse se trouvant en mémoire
  0;JMP     // saut vers l'adresse de retour
```

Le programme complet intègre ces deux modifications.

```
   @compteur
   M=0
(LIGNE1)
   @1  // ligne 1
   D=A
   @a
   M=D
   @LIGNE3  // sauvegarde de l'adresse de retour
   D=A
   @retour
   M=D
   @COMPTE  // appel de la procédure
   0;JMP                  // adresse 11
(LIGNE3)                  // adresse 12
   @2  // ligne 3
   D=A
   @a
   M=D
   @LIGNE5  // sauvegarde de l'adresse de retour
   D=A
   @retour
   M=D
   @COMPTE  // appel de la procédure
   0;JMP                  // adresse 21
(LIGNE5)                  // adresse 22
   @3    // ligne 5
   D=A
   @a
   M=D                    // adresse 25
   @FIN
   0;JMP
(COMPTE)
   @compteur
   M=M+1
   @retour
   A=M
   0;JMP
(FIN)
```

Ce programme est téléchargeable depuis `asm/procedure-ex3.asm`. Il est intéressant d'observer l'évolution de la mémoire durant l'exécution de ce programme. La Fig. 16.2 présente l'état de la mémoire lors de l'exécution de l'instruction `0;JMP` se trouvant à l'adresse 11. A ce moment, l'adresse `16` contient le compteur initialisé à zéro, l'adresse `17` contient la variable `a` initialisée à `1`. L'adresse de retour est l'adresse de l'instruction qui suit l'instruction `0;JMP`, c'est-à-dire la valeur `12`. Elle est stockée en mémoire à l'adresse `18`. Après l'exécution de cette instruction, le programme va exécuter le code de la procédure. Celui-ci va incrémenter la valeur de la variable `compteur` et ensuite revenir au programme principal à l'instruction se trouvant à l'adresse `12`. La Fig. 16.3 présente l'état de la mémoire à ce moment. L'exécution du programme se poursuit. Lors de l'exécution de la seconde instruction `0;JMP` du programme principal, le contenu de la mémoire est tel que représenté en Fig. 16.4. En pratique, une telle procédure n'est pas isolée et il est courant qu'une procédure fasse appel à une autre procédure. Analysons ce cas de figure en utilisant l'exemple ci-dessous en python.

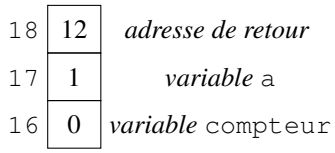

Fig. 16.2 – Contenu de la mémoire durant l'exécution de l'instruction se trouvant à l'adresse 11

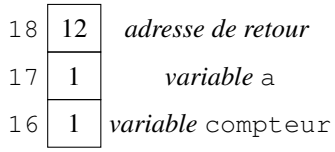

Fig. 16.3 – Contenu de la mémoire durant l'exécution de l'instruction se trouvant à l'adresse 12

```
compteur=0

def oppose():
    global compteur
    compteur = -compteur


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


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

Dans ce programme, la procédure `compte()` incrémente le compteur et la procédure `oppose()` inverse le signe de ce compteur. Après exécution de la ligne `3`, la variable compteur contient la valeur `-1`. A la fin du programme, cette variable contient la valeur `0`. Nous pouvons maintenant essayer de convertir ce petit programme en minuscule assembleur.

```
   @compteur
   M=0
(LIGNE1)
   @1  // ligne 1
   D=A
   @a
   M=D
   @LIGNE3  // sauvegarde de l'adresse de retour
   D=A
   @retour
   M=D
   @COMPTE  // appel de la procédure
```

(suite sur la page suivante)

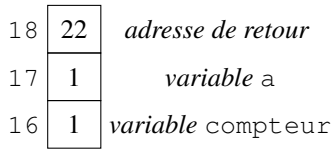

Fig. 16.4 – Contenu de la mémoire durant l'exécution de l'instruction se trouvant à l'adresse 21

(suite de la page précédente)

```
   0;JMP                  // adresse 11
(LIGNE3)                  // adresse 12
   @2  // ligne 3
   D=A
   @a
   M=D
   @LIGNE5  // sauvegarde de l'adresse de retour
   D=A
   @retour
   M=D
   @COMPTE  // appel de la procédure
   0;JMP                  // adresse 21
(LIGNE5)                  // adresse 22
   @3    // ligne 5
   D=A
   @a
   M=D                    // adresse 25
   @FIN
   0;JMP
(COMPTE)
   @compteur
   M=M+1
   @SUIVANT  // sauvegarde de l'adresse de retour
   D=A
   @retour
   M=D
   @OPPOSE   // appel de la procédure
   0;JMP
(SUIVANT)
   @retour                 // adresse 36
   A=M
   0;JMP
(OPPOSE)
   @compteur               // adresse 39
   M=-M
   @retour
   A=M
   0;JMP
(FIN)
```

Ce programme (téléchargeable via `asm/procedure-ex4.asm`) utilise les mêmes variables en mémoire que le programme précédent. Son exécution démarre de la même façon. Il est intéressant d'observer le premier appel à la procédure `compte`. Au début de son exécution la mémoire contient les valeurs représentées en Fig. 16.5. La procédure incrémente la valeur de la variable `compteur`. Ensuite elle fait appel à la procédure `oppose()`. Pour cela, elle sauvegarde l'adresse qui suite celle de l'instruction `0;JMP`, dans ce cas l'adresse `36`. Elle exécute ensuite l'instruction `0;JMP`. A ce stade, la mémoire contient les valeurs représentées dans la Fig. 16.6. La procédure `oppose()` s'exécute

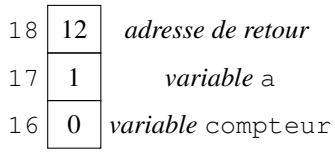

Fig. 16.5 – Contenu de la mémoire durant la première instruction de la procédure compte

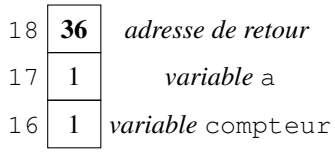

Fig. 16.6 – Contenu de la mémoire durant la première instruction de la procédure compte

et place la valeur `-1`. Il ne lui reste plus qu'à retourner au programme appelant, c'est à dire dans le corps de la procédure `compte()`. L'adresse `36` est donc chargée de la mémoire et l'instruction `0;JMP` est exécutée. Les trois instructions qu'il reste à exécuter dans la procédure `compte()` sont les suivantes :

```
(SUIVANT)
    @retour                 // adresse 36
    A=M
    0;JMP
```

Ces instructions vont recharger la valeur stockée à l'adresse de retour, c'est-à-dire `36` puis l'instruction `0;JMP` va redémarrer l'exécution des instructions à cette adresse. Nous avons malheureusement écrit une boucle infinie...

Le problème que nous observons est dû au fait qu'en sauvegardant l'adresse de retour dans la procédure `compte()` pour supporter l'appel à la procédure `oppose()`, on écrase l'adresse qui permet à la procédure `compte()` de retourner au bon endroit du programme appelant. Pour supporter une procédure qui appelle une autre procédure, nous avons besoin de stocker deux adresses de retour. Mais il est aussi possible qu'une procédure appelle une procédure qui elle-même appelle une autre procédure, ... En toute généralité la succession des appels de procédure n'est pas nécessairement bornée. De plus, une procédure `p1()` peut appeler une procédure `p2()`, mais cette procédure `p2()` peut aussi être appelée par la procédure `p4()` qui elle-même est une procédure appelée par `p5()` qui est appelé par la procédure `p1()`. Il suffit pour s'en convaincre de regarder le nombre d'appels à des fonctions de type `print` qu'un programme peut contenir.

Avant d'appeler une procédure, nous devons trouver une moyen pour sauvegarder l'adresse de retour de cette procédure. Lors de son exécution, notre procédure peut aussi sauvegarder des adresses de retour, mais à la fin de son exécution nous devons pouvoir récupérer l'adresse de retour que nous avions sauvegardé. Cela correspond au fonctionnement d'un *pile* (*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).

Note : Une pile d'adresses en mémoire

La solution la plus simple pour stocker une pile d'adresses et la manipuler en minuscule assembleur est d'utiliser une zone de mémoire contiguë. Une première approche est d'utiliser l'adresse `p` pour stocker le premier élément de la pile et d'ajouter les éléments suivants aux adresses `p+1`, `p+2`, ... Pour illustrer cette approche, la figure Fig. 16.7 l'évolution de la pile lors de l'exécution des opérations suivantes : `push(3) ; pop ; push(2) ; 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-1`, `p-2`, ... La figure Fig. 16.7 illustre l'évolution d'un telle pile lors de l'exécution des opérations suivantes : `push(3) ; pop ; push(2) ; push(5)`. En pratique, c'est 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

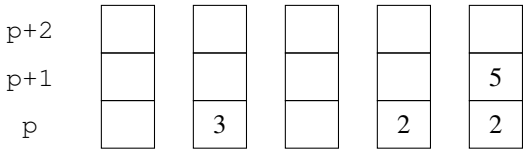

Fig. 16.7 – Evolution de la pile en démarrant à une adresse basse

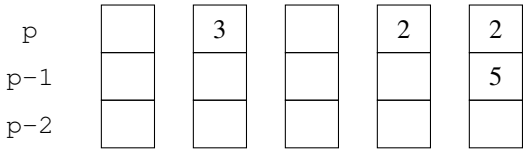

Fig. 16.8 – Evolution de la pile en démarrant à une adresse haute

contraindre la mémoire utilisée par un programme.

Avant de modifier notre programme pour y intégrer une pile, il est intéressant de réfléchir à la façon dont on peut implémenter une pile d'entiers en utilisant le minuscule assembleur. Les deux opérations que nous devons supporter sont :

- `push(x)` place au sommet de la pile la valeur `x`. Après l'exécution de `push`, la pile contient un élément de plus.
- `y=pop()` récupère la valeur se trouvant au sommet de la pile et la retire. Cette opération ne peut être réalisée que sur une pile qui contient au moins un élément. Après l'exécution de `pop`, la pile contient un élément de moins.

Pour implémenter ces deux opérations, nous devons pouvoir stocker les différents entiers qui se trouvent dans la pile et savoir quel est celui qui se situe au sommet, celui qui se trouve juste en dessous, ... Nous adoptons la représentation de la pile qui démarre aux adresses hautes et réservons 10 mots en mémoire pour stocker notre pile en supposant qu'elle ne contiendra jamais plus d'éléments. Nous supposons également que cette zone mémoire commence à l'adresse 1000. De plus, la variable `pile` contient à tout moment l'adresse de l'élément se trouvant au sommet de la pile.

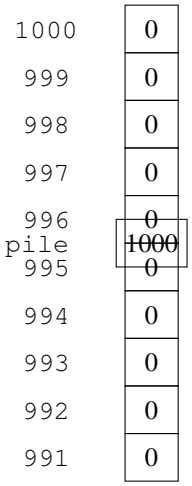

Fig. 16.9 – Pile pouvant stocker dix éléments

Avant d'écrire les opérations `push` et `pop` en minuscule assembleur, nous devons encore déterminer où doit se trouver la valeur que `push` sauve sur le stack et où l'opération `pop` va sauver l'entier qui se trouvait au sommet de la pile. Dans ces deux cas, nous choisissons le registre `D`. Celui-ci devra contenir l'entier à écrire sur la pile avant d'exécuter le code correspondant à l'opération `push`. De façon similaire, l'opération `pop` se terminera en laissant la valeur se trouvant au sommet de la pile dans le registre `D`. A tout moment, la variable `pile` contiendra l'adresse de l'élément se trouvant au sommet de la pile.

Nous pouvons maintenant écrire les instructions pour implémenter l'opération `push`. Il faut tout d'abord décrémenter la variable qui contient le sommet de la pile pour « faire de la place » pour ce nouvel élément. Comme notre pile grandit en mémoire vers les adresses basses, il suffit pour cela de décrémenter l'adresse contenue dans la variable `pile`. Nous devons ensuite sauvegarder la valeur se trouvant dans le registre `D` à cette adresse.

```
// push
        // D contient l'entier à sauver au sommet de la pile
@pile
M=M-1   // libération d'une place au sommet de la pile
@pile
A=M     // l'adresse contenue dans la variable pile est placée dans A
M=D     // sauvegarde de la valeur contenue dans le registre D
```

L'opération `pop` est assez proche. Nous devons d'abord récupérer la valeur se trouvant à l'adresse contenue dans le registre `pile` et la placer dans le registre `D`. Ensuite, il faut indiquer que l'élément qui était au sommet de la pile ne s'y trouve plus. Pour cela, il suffit d'incrémenter l'adresse contenue dans la variable `pile` puisque notre pile grandit vers les adresses basses.

```
// pop

@pile
A=M         // chargement de l'adresse contenue dans la variable pile
D=M         // sauvegarde de la donnée se trouvant au sommet de la pile
@pile
M=M+1       // libération de la place correspondant au sommet de la pile
            // la donnée lue au sommet de la pile est dans le registre D
```

Le programme ci-dessous (téléchargeable depuis `asm/pile-exemple.asm`) illustre le fonctionnement d'une telle pile.

```
    // exemple d'utilisation de la pile
    // celle-ci démarre à l'adresse 1000
    @1000
    D=A
    @pile
    M=D
    // push (2)
    @2
    D=A
    @pile
    M=M-1
    @pile
    A=M
    M=D
    // push (7)
    @7
    D=A
    @pile
    M=M-1
    @pile
```

(suite sur la page suivante)

(suite de la page précédente)

```
    A=M
    M=D
    // pop
    @pile
    A=M
    D=M
    @pile
    M=M+1
    // push (9)
    @9
    D=A
    @pile
    M=M-1
    @pile
    A=M
    M=D
    // pop
    @pile
    A=M
    D=M
    @pile
    M=M+1
    // pop
    @pile
    A=M
    D=M
    @pile
    M=M+1
```

Durant son exécution, la mémoire d'un de nos programmes en minuscule assembleur comprendra trois parties. Tout d'abord, le bas de la mémoire (adresses de `0` à `15`) est réservé pour stocker des valeurs particulières. Le livre en définit plusieurs dans les derniers chapitres. Nous en discuterons quelques unes prochainement. La deuxième partie de la mémoire contient les variables, tableaux et chaînes de caractères utilisés par le programme. Dans le minuscule assembleur, cette zone démarre à l'adresse `16` et peut grandir vers le haut si le programme a besoin de plus de mémoire (nous ne discuterons pas ce cas de figure dans ce cours introductif, mais vous en entendrez parler l'an prochain). La dernière partie de la mémoire est réservée à 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.

Pour pouvoir utiliser cette pile, nous devons d'abord convenir de l'adresse qui correspond au sommet de la pile. Nous prenons la convention de débuter la pile à l'adresse la plus haute en mémoire, `16383`.

Nous pouvons maintenant réécrire notre programme en assembleur en utilisant cette pile. Le livre réserve l'adresse `0` en mémoire de données pour stocker l'adresse du sommet de la pile. Grâce à cette pile, nous pouvons réécrire les fonctions `compte` et `oppose`.

La fonction `oppose` est la plus simple. Elle calcule d'abord l'opposé de la variable `compteur`. Ensuite, elle récupère sur le pile l'adresse de retour et incrémente l'adresse du sommet de pile. Elle se termine par un saut à cette adresse de retour.

La fonction `compte` est un peu plus compliqué puisqu'elle contient une appel à la fonction `oppose` et qu'elle doit retourner au programme appelant. Elle incrémente d'abord la valeur du compteur. Ensuite, elle sauve l'adresse de retour sur la pile avant d'appeler la procédure `oppose`. Au retour de celle-ci, elle récupère l'adresse de retour qui est au sommet de la pile pour retourner au programme appelant.

Le programme complet est repris ci-dessous. Il est téléchargeable depuis `asm/procedure-pile.asm`.

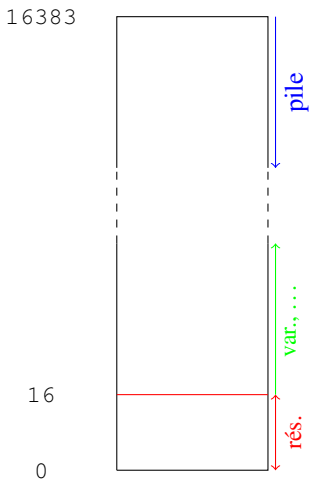

Fig. 16.10 – Organisation de la mémoire de données d'un programme

```
   @16383 // sommet choisi pour notre pile
   D=A
   @SP    // 0 par convention dans le livre
   M=D
   @compteur
   M=0
(LIGNE1)
   @1  // ligne 1
   D=A
   @a
   M=D
   @SP
   M=M-1    // libération d'une place au sommet de la pile
   @LIGNE3  // sauvegarde de l'adresse de retour
   D=A
   @SP
   A=M
   M=D
   @COMPTE  // appel de la procédure
   0;JMP                  // adresse 18
(LIGNE3)                  // adresse 19
   @2  // ligne 3
   D=A
   @a
   M=D
   @SP
   M=M-1    // libération d'une place au sommet de la pile
   @LIGNE5  // sauvegarde de l'adresse de retour
   D=A
   @SP
   A=M
   M=D
   @COMPTE  // appel de la procédure
   0;JMP                  // adresse 31
(LIGNE5)                  // adresse 32
   @3    // ligne 5
   D=A
```

(suite sur la page suivante)

(suite de la page précédente)

```
   @a
   M=D                      // adresse 35
   @FIN
   0;JMP
(COMPTE)
   @compteur
   M=M+1
   @SP
   M=M-1    // libération d'une place au sommet de la pile
   @SUIVANT  // sauvegarde de l'adresse de retour
   D=A
   @SP
   A=M
   M=D
   @OPPOSE   // appel de la procédure
   0;JMP
(SUIVANT)
   @SP                  // adresse 48
   A=M
   D=M
   @SP
   M=M+1
   A=D
   0;JMP
(OPPOSE)
   @compteur            // adresse 56
   M=-M
   @SP                  // adresse 58
   A=M
   D=M
   @SP
   M=M+1
   A=D
   0;JMP
(FIN)
```

Grâce à cette 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 procédure 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 `A` au début de l'exécution d'une procédure, à la fin de celle-ci la pile dont 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. Pour s'en rendre compte, il suffit de prendre le programme ci-dessous et de l'exécuter après avoir par exemple remplacé une des instructions qui modifie la pile dans les fonctions `compte` ou `oppose`. 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 procédure. 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 lors de chaque opération `push` et `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 données du programme. Dans ce cas, le programme aura un comportement totalement incohérent. Certains attaques 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. En assembleur, et par extension dans la plupart des langages de programmation, la pile joue un rôle fondamental. C'est grâce à la pile que nous allons pouvoir également supporter les fonctions pour lesquelles 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 ce qu'il se passe avec des fonctions. 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 une variable locale.

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

# 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 ses variables locales ?

Commençons par la première question. Avant d'appeler une fonction, il est nécessaire 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 en mémoire dans une zone qui est accessible à la fonction lors de son exécution. Notre fonction doit déjà récupérer l'adresse de retour sur la pile. Les arguments de la fonction seront également placés sur la pile, dans l'ordre dans lequel ils sont passés à la fonction. Durant son exécution, la fonction pourra facilement récupérer ses arguments depuis la pile.

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. Par convention, une fonction

écrite en minuscule assembleur retournera un seul mot de 16 bits et celui-ci sera placé dans le registre `D` qui est le seul registre vraiment utilisable sur le minuscule processeur.

Commençons par la fonction `f1` pour illustrer le passage des arguments et de la valeur de retour d'une fonction simple. Pour le passage des arguments, nous devons convenir de l'ordre dans lequel ceux-ci et l'adresse de retour doivent être placés sur la pile. Le livre de référence choisit de placer d'abord les arguments et ensuite l'adresse de retour. Lors d'un appel à la fonction `f1(7)`, la pile ressemblera à la Fig. 16.11 où `sp` est l'adresse du sommet de la pile. Pour récupérer

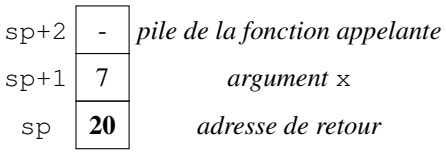

Fig. 16.11 – Contenu de la pile lors de l'appel à la fonction ‘‘f1‘‘

son argument, la fonction `f1` doit donc lire l'information se trouvant à l'adresse `sp+1` tandis que l'adresse de retour se trouve au sommet de la pile. Le programme complet est téléchargeable via `asm/f1.asm`.

```
     @16383    // initialisation pile
     D=A
     @SP
     M=D
     @SP       // libération de la place pour l'argument
     M=M-1
     @7        // argument
     D=A
     @SP
     A=M
     M=D
     @SP       // libération de la place pour l'adresse de retour
     M=M-1
     @RETOUR   // adresse de retour
     D=A
     @SP
     A=M
     M=D
     @F1      // appel à la fonction
     0;JMP
(RETOUR)
     // D contient le résultat de f1(7)
     @FIN
     0;JMP
(DEBUT)
// Implémentation simple en minuscule assembleur de la fonction f1
// def f1(x):
//     y=x+1
//    return(y)
(F1)
     @SP
     A=M+1   // adresse de l'argument x
     D=M+1   // calcul de y=x+1 et sauvegarde dans D
     @SP     // récupération de l'adresse de retour
     M=M+1   // et suppression de l'argument
     M=M+1   // et de l'adresse de retour de la pile
     A=M     // adresse du sommet de la pile après f1
     A=A-1   // adresse de l'argument de f1 sur la pile
```

(suite sur la page suivante)

(suite de la page précédente)

```
     A=A-1   // adresse de l'adresse de retour sur la pile
     A=M     // chargement de l'adresse de retour dans A
     0;JMP   // retour au programme appelant
(FIN)
```

Il est intéressant d'examiner un peu plus en détails l'implémentation de la fonction `f1`. Tout d'abord, notre fonction doit récupérer son argument. Celui-ci étant le deuxième élément sur la pile, il suffit de mettre son adresse dans le registre `A`. C'est ce que font les deux premières instructions de notre fonction. Après l'exécution de ces deux instructions, le registre `A` contient l'adresse de la zone mémoire contenant l'argument de la fonction `f1`. Pour incrémenter cet argument, il nous suffit de calculer `M+1`. Comme le résultat de ce calcul est aussi la valeur de retour de la fonction `f1`, il suffit de le stocker dans le registre `D`. En trois lignes nous avons donc implémenté le corps de la fonction. Il nous reste mettre à récupérer l'adresse de retour de la fonction `f1` sur la pile et de retirer cette adresse ainsi que l'argument pour que la fonction appelante retrouve une pile propre. Toutes ces opérations doivent se faire sans utiliser le registre `D` puisque celui-ci contient déjà la valeur retournée par notre fonction. Les trois instructions suivantes (`@SP`, `M=M+1` et `M=M+1`) modifient l'adresse du sommet de pile et « suppriment » donc l'adresse et l'argument qui s'y trouvent. Il est utile de noter qu'il suffit d'incrémenter l'adresse contenue dans `SP` pour retirer un élément sur la pile. Il n'est pas nécessaire de remplacer la valeur qui s'y trouve par zéro. Mettre cette valeur à zéro serait une perte de temps et de performance. Après l'exécution de ces trois instructions, `SP` contient la bonne valeur. Il nous reste à récupérer l'adresse de retour. Si `X` est l'adresse actuelle du sommet de la pile, alors l'adresse de retour se trouver à l'adresse `X-2`. Les quatre instructions qui suivent permettent de récupérer cette adresse et de la stocker dans le registre `A` pour pouvoir exécuter un saut vers cette adresse sans utiliser le registre `D`.

Nous pouvons maintenant regarder une fonction qui prend deux arguments comme celle qui calcule le minimum.

```
     @x         // force l'assembleur à réserver l'adresse 16 pour x
     @y         // force l'assembleur à réserver l'adresse 17 pour y
     @z         // force l'assembleur à réserver l'adresse 18 pour z
     @16383     // initialisation pile
     D=A
     @SP
     M=D
     @SP        // libération de la place pour le second argument
     M=M-1
     @y         // argument
     D=M
     @SP
     A=M
     M=D
     @SP        // libération de la place pour le premier argument
     M=M-1
     @x         // argument
     D=M
     @SP
     A=M
     M=D
     @SP        // libération de la place pour l'adresse de retour
     M=M-1
     @RETOUR    // adresse de retour
     D=A
     @SP
     A=M
     M=D
     @MIN       // appel à la fonction
     0;JMP
(RETOUR)
     // D contient le résultat de min(x,y)
```

(suite sur la page suivante)

(suite de la page précédente)

```
     @z
     M=D
     @FIN
     0;JMP
(DEBUT)
// Implémentation simple en minuscule assembleur de la fonction min
// def min(x,y):
//     if(x<y):
//         return(x)
//     return(y)
(MIN)
     @SP
     A=M+1    // adresse de l'argument y
     D=M      // sauvegarde dans D
     @SP
     A=M+1
     A=A+1    // adresse de l'argument x
     D=D-M    // y-x
     @RETY
     D;JGT
     @SP      // saut conditionnel a échoué, D doit contenir y
     A=M+1    // adresse de l'argument y
     D=M      // valeur de retour
     @RET
     0;JMP
(RETY)
     @SP      // saut conditionnel a réuissi, D doit contenir x
     A=M+1    // adresse de l'argument y
     A=A+1    // adresse de l'argument x
     D=M      // valeur de retour
     @RET
     0;JMP
(RET)
     @SP      // récupération de l'adresse de retour
     M=M+1    // et suppression du premier argument
     M=M+1    // et suppression du second argument
     M=M+1    // et de l'adresse de retour de la pile
     A=M      // adresse du sommet de la pile après min
     A=A-1    // adresse du premier argument sur la pile
     A=A-1    // adresse du deuxième argument sur la pile
     A=A-1    // adresse de l'adresse de retour sur la pile
     A=M      // chargement de l'adresse de retour dans A
     0;JMP    // retour au programme appelant
(FIN)
```

L'implémentation de cette fonction est assembleur est téléchargeable via `asm/min.asm`. Le script de test est téléchargeable via `asm/min.tst`. Par rapport à la fonction `f1` que nous avons présenté précédemment, il faut regarder comment chaque argument est récupéré de la pile. A la fin de l'exécution de la fonction `min`, il faut retirer les trois éléments qui avaient étés placés sur la pile par le programme appelant. De cette façon, le programme qui a appelé la fonction `min` retrouve la pile dans l'état dans lequel elle se trouvait avant l'appel à la fonction.

Note : Utilisation de la pile par une fonction

Une fonction telle que `f1` ou `min` qui utilise la pile doit respecter plusieurs principes pour garantir le bon fonctionnement du programme qui l'a appelée :

1. La fonction ne peut accéder aux éléments qui ont été placés sur la pile *avant* ses arguments et son

adresse de retour. Les informations qui se trouvent dans le bas de la pile sont nécessaires à l'exécution d'autres fonctions et ne peuvent en aucun cas être modifiées par la fonction.

2. La fonction peut rajouter des éléments sue la pile, soit directement soit en appelant d'autres fonctions. Cela implique que l'adresse du sommet de pile peut se modifier *durant* l'exécution de la fonction. Quelles que soient les modifications qu'elle fait à la pile, la fonction doit garantir qu'à l'issue de *n'importe laquelle* de ses exécutions la pile retrouvera l'état qu'elle avait avant l'appel à la fonction.

Si un de ces principes n'est pas respecté par une fonction, le programme qui utilise cette fonction risque d'avoir un comportement erratique voire totalement incorrect. C'est le développeur d'une fonction en assembleur qui doit garantir la correction de sa fonction pour toutes ses exécutions possibles. Comme dans des langages de programmation de plus haut niveau, les tests les plus exhaustifs possibles sont une excellente façon de vérifier le bon fonctionnement d'un fonction.

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 des arguments qu'elle doit utiliser. Ces valeurs sont copiées 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 un 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 cors de la fonction `inc` accède à la variable utilisée par le programme appelant et modifie sa valeur avant de terminer son exécution.

```
     @x        // variable
     @r        // résultat
     @16383    // initialisation pile
     D=A
     @SP
     M=D
     @SP       // libération de la place pour l'argument
     M=M-1
     @x        // adresse de la variable passée en argument
     D=A
     @SP
     A=M
     M=D
     @SP       // libération de la place pour l'adresse de retour
     M=M-1
     @RETOUR   // adresse de retour
     D=A
     @SP
     A=M
     M=D
     @INC      // appel à la fonction
     0;JMP
(RETOUR)
     // D contient le résultat de inc(x)
```

(suite sur la page suivante)

(suite de la page précédente)

```
     @r
     M=D
     @FIN
     0;JMP
// Implémentation simple en minuscule assembleur de la fonction inc qui
// incrémente la valeur stockée à l'adresse passée en argument
// exemple de passage par référence - bien spécifier quand c'est le cas
(INC)
     @SP
     A=M+1   // élément de la pile contenant l'argument
     D=M     //
     A=D
     M=M+1   // incrémentation de la variable
     @SP
     A=M+1   // adresse contenant l'argument
     M=D     // mise à jour de la valeur de la variable passée en argument
     @SP     // récupération de l'adresse de retour
     M=M+1   // et suppression de l'argument
     M=M+1   // et de l'adresse de retour de la pile
     A=M     // adresse du sommet de la pile après inc
     A=A-1   // adresse de l'argument de inc sur la pile
     A=A-1   // adresse de l'adresse de retour sur la pile
     A=M     // chargement de l'adresse de retour dans A
     0;JMP   // retour au programme appelant
(FIN)
```

Le programme complet (`asm/inc.asm`) et le script de test (`asm/inc.tst`) sont téléchargeables.

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
    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`, celle-ci va modifier 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 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 comment 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
```

Cette fonction peut être traduite en minuscule assembleur. Il y a trois parties distinctes dans cette fonction récursive. La première, généralement baptisée le cas de base, est la partie du code qui calcule la valeur de retour lorsque l'argument de la fonction est nul. Les premières instructions récupèrent l'argument de la pile et le placent dans le registre `D`. Si celui-ci est nul, il contient déjà la valeur de retour (`0`) et les instructions suivantes retirent simplement l'argument et l'adresse de retour de la pile et permettent de retourner au programme appelant.

La deuxième partie de la fonction `sumn` est l'appel récursif (à partir de l'étiquette `RECURSION`). Dans ce cas, nous commençons par décrémenter l'argument qui avait préalablement été placé dans le registre `D`. Cette valeur calculée est celle qui est passée à l'appel `sumn(n-1)`. Cette valeur est placée sur la pile et l'adresse de retour (`RETSUMN`) également.

La troisième partie de la fonction `sumn` est le retour de l'appel récursif (après `RETSUMN`). Nous recevons dans le registre `D` le résultat du calcul de `sumn(n-1)`. Il nous reste à y additionner la valeur de l'argument qui se trouve sur la pile. Cette addition est réalisée par les trois premières instructions de cette partie. Ensuite, il suffit de retirer l'argument et l'adresse de retour de la pile pour retourner au programme appelant.

```
    @x        // variable
    @r        // résultat
    @16383    // initialisation pile
    D=A
    @SP
    M=D
    @SP       // libération de la place pour l'argument
    M=M-1
    @x        // adresse de la variable passée en argument
    D=M
    @SP
    A=M
    M=D
    @SP       // libération de la place pour l'adresse de retour
    M=M-1
    @RETOUR   // adresse de retour
    D=A
    @SP
    A=M
```

(suite sur la page suivante)

(suite de la page précédente)

```
     M=D
     @SUMN      // appel à la fonction
     0;JMP
(RETOUR)
     // D contient le résultat de SUMN(x)
     @r
     M=D
     @FIN
     0;JMP
// Implémentation simple en minuscule assembleur de la fonction récursive
// sumn qui retourne la somme des n premiers naturels
(SUMN)
     @SP
     A=M+1   // élément de la pile contenant l'argument
     D=M     //
     @RECURSION
     D;JNE
             // argument == 0, D contient déjà 0
     @FINSUMN
     0;JMP
(RECURSION)
             // D contient l'argument, calculer n-1 puis appeler sumn
     D=D-1
     @SP       // libération de la place pour l'argument
     M=M-1
     @SP
     A=M
     M=D
     @SP       // libération de la place pour l'adresse de retour
     M=M-1
     @RETSUMN   // adresse de retour
     D=A
     @SP
     A=M
     M=D
     @SUMN      // appel à la fonction
     0;JMP
(RETSUMN)
                // retour de sumn(n-1), reste à récupérer n et n+D
     @SP
     A=M+1      // élément de la pile contenant l'argument
     D=D+M      // calcul de la valeur de retour
(FINSUMN)
             // libération de la pile et retour
     @SP     // récupération de l'adresse de retour
     M=M+1   // et suppression de l'argument
     M=M+1   // et de l'adresse de retour de la pile
     A=M     // adresse du sommet de la pile après SUMN
     A=A-1   // adresse de l'argument de SUMN sur la pile
     A=A-1   // adresse de l'adresse de retour sur la pile
     A=M     // chargement de l'adresse de retour dans A
     0;JMP   // retour au programme appelant
↪
(FIN)
```

Ce programme (`asm/sumn.asm`) et son script de test (`asm/sumn.tst`) sont téléchargeables.

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 et de voir l'évolution de la pile. La Fig. 16.12 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 est l'élément le plus en bas de la figure et en gras. Durant son exécution, cette fonction fait appel à `sumn(2)`. La Fig. 16.13 présente l'état de

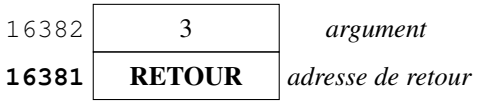

Fig. 16.12 – Contenu de la pile lors de l'appel à la fonction sumn(3)

l'appel au moment de cet appel. Lors de son exécution, l'invocation de la fonction `sumn` avec `2` comme argument

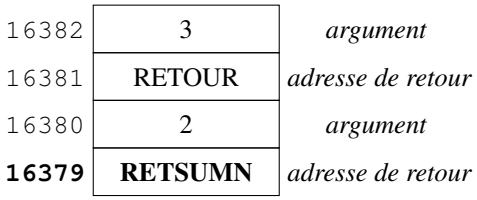

Fig. 16.13 – Contenu de la pile lors de l'appel à sumn(2)

va d'abord faire appel à `sumn(1)`. La Fig. 16.14 présente l'état de l'appel au moment de cet appel. Lors de son

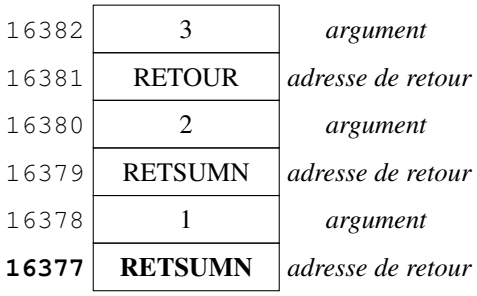

Fig. 16.14 – Contenu de la pile lors de l'appel à sumn(1)

exécution, l'invocation de la fonction `sumn` avec `1` comme argument va d'abord faire appel à `sumn(0)`. La Fig. 16.15 présente l'état de l'appel au moment de cet appel. Nous sommes maintenant dans l'éxécution de la fonction `sumn(0)`. Celle-ci retourne la valeur `0` dans le registre `D` et retire les deux mots se trouvant au sommet de la pile. Elle retourne à l'adresse `RETSUMN` avec comme pile celle représentée à la Fig. 16.14. Avec cette pile, la fonction `sumn` récupère l'argument valant `1` et retourne la valeur `1` qui est la somme entre la valeur récupérée dans le registre `D` 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.

La pile contient maintenant les informations reprises en Fig. 16.13 et le registre `D` 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`.

Nous sommes maintenant dans l'invocation de la fonction `sumn(3)` avec la pile présentée en Fig. 16.12. La fonction récupère son argument (`3`) sur la pile et l'ajoute au résultat de la fonction appelée qu'elle a reçu dans le registre `D`. Le registre `D` contient 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.

16382	3	*argument*
16381	RETOUR	*adresse de retour*
16380	2	*argument*
16379	RETSUMN	*adresse de retour*
16378	1	*argument*
16377	RETSUMN	*adresse de retour*
16376	0	*argument*
16375	**RETSUMN**	*adresse de retour*

Fig. 16.15 – Contenu de la pile lors de l'appel à sumn(0)

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

Q

R

S

U