Cours condensé avec exercices corrigés

Licence d'informatique • DUT • Écoles d'ingénieurs

ARCHITECTURE DE L'ORDINATEUR

Portes logiques, circuits combinatoires, arithmétique binaire, circuits séquentiels et mémoires. Exemple d'architecture.

Robert Strandh
Irène Durand

DUNOD

ARCHITECTURE DE L'ORDINATEUR

Portes logiques, circuits combinatoires, arithmétique binaire, circuits séquentiels et mémoires. Exemple d'architecture.

Robert Strandh
Professeur à l'université Bordeaux 1

Irène Durand
Maître de conférences à l'université Bordeaux 1

DUNOD

Illustration de couverture : contexture, digitalvision®

ISBN 2 10 049214 4

Table des matières

Introduction

L'architecture des ordinateurs constitue un vaste sujet en constante évolution. En 20 ans, la vitesse d'un processeur standard a été multipliée par un facteur 1 000 et le coût par unité de performance a plongé d'un facteur 100 000. Pour ces raisons, tout livre basé sur une architecture particulière risque d'être obsolète six mois après sa sortie.

Ce livre n'est pas destiné aux concepteurs de processeurs de demain, mais aux informaticiens actuels et futurs. Ce n'est pas le rôle d'un informaticien de concevoir des architectures. Les détails d'une architecture particulière, ou même de l'architecture en général, ne sont pas forcément utiles. L'informaticien a plutôt besoin d'un *modèle* de fonctionnement de l'ordinateur qui lui donne une bonne idée de la performance de son programme et de l'impact de modifications données du programme sur sa performance.

Un tel modèle nécessite certaines connaissances sur le fonctionnement d'un ordinateur telles que :

– le fonctionnement du mécanisme d'appel de fonction ;
– la façon dont des paramètres sont transmis d'une fonction à l'autre ;

– l'allocation et la libération d'espace pour les variables locales ;
– l'organisation du cache ;
– la manière dont les processus légers sont implémentés ;
– la méthode de changement de contexte ;
– etc.

Ces notions ne changent pas aussi rapidement que les détails des architectures modernes.

Dans ce livre, nous présentons un modèle cohérent du point de vue d'un informaticien. Parfois, ce modèle est une approximation grossière de la réalité. Parfois, il est même complètement faux, mais néanmoins utile pour prévoir le comportement d'un programme.

Ce livre a deux avantages considérables par rapport aux autres ouvrages du domaine :

– il ne nécessite pas de prérequis autres que des notions de mathématiques de niveau lycée ;
– aucune étape de la construction de l'ordinateur n'est omise ; en théorie, le lecteur pourrait donc à la fin de ce livre construire un ordinateur selon les principes qui y sont présentés.

Cet ouvrage est organisé en trois parties. La première partie, *Notions de base*, introduit trois types de circuits : circuits combinatoires, circuits séquentiels et mémoires. Plusieurs exemples de circuits classiques sont exposés (compteurs, multiplexeurs, circuits pour l'arithmétique binaire). Dans la deuxième partie, *Exemple d'architecture*, les notions de base introduites dans la première partie sont utilisées pour construire un ordinateur simple, mais complet. L'ordinateur initial n'est capable d'exécuter que des programmes très simples. Puis, cet ordinateur est étendu avec des fonctionnalités telles que les sauts conditionnels, les sous-programmes récursifs et les interruptions. Dans la troisième partie, *Sujets avancés*, nous introduisons des notions comme la mémoire cache, l'adressage virtuel et la multiprogrammation.

PARTIE 1

NOTIONS DE BASE

Chapitre 1

Notions préalables

BITS ET OCTETS

Ce chapitre introduit les notions de bit, octet, kilo-octet... Le lecteur déjà familier avec celles-ci peut passer directement à la lecture du chapitre suivant.

Pour exprimer et manipuler les nombres entiers, nous utilisons habituellement leur représentation en base 10 avec les dix chiffres 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Matériellement, on peut coder facilement deux valeurs différentes par une tension électrique : une tension haute représentera l'une des deux valeurs et une tension basse, l'autre valeur. De ce fait, les ordinateurs utilisent principalement la base 2 et donc les deux chiffres 0, 1. Cette unité d'information qui peut valoir 0 ou 1 est appelée *bit*.

Une suite de 8 bits constitue un *octet* (en anglais : *byte*). C'est généralement la plus petite quantité d'information manipulable par un programmeur. Avec un octet on peut représenter 2^8, soit 256 valeurs différentes.

Vient ensuite la notion de *mot* (en anglais : *word*). Autrefois un mot était constitué de deux octets (16 bits) et on parlait d'ordinateur 16 bits. La taille du mot ayant évolué avec la technologie, on trouve

maintenant couramment des mots de 32 bits (4 octets) et même 64 bits (8 octets).

Vu que la taille du mot n'est pas a priori bien fixée, toutes les quantités d'information seront exprimées à partir de l'octet qui est, quant à lui, de taille bien définie (8 bits).

Un *kilo-octet* (Ko) correspond à $2^{10} = 1\,024$ octets. Un *mega-octet* (Mo) correspond à $2^{20} = 1\,048\,576$ octets ou encore 1 024 Ko.

Suite à la rapide évolution de la taille des mémoires, nous utilisons maintenant couramment le *giga-octet* (Go) qui correspond à 2^{30} octets, soit 2^{20} Ko, soit 2^{10} Mo. Est apparu récemment, le *tera-octet* (To) qui correspond à 2^{40} octets, soit 2^{30} Ko, soit 2^{20} Mo, soit 2^{10} Go.

EXERCICES

Exercice 1.1. Aujourd'hui les techniques de compression d'images numériques animées permettent d'utiliser un débit de 4 Mbit/s pour transmettre un film avec une bonne qualité d'image. Quelle est dans ces conditions, la capacité minimale d'un DVD *(Digital Video Disc)* contenant un film d'une durée de deux heures ? La réponse sera donnée en octets, en utilisant les abréviations standard Ko, Mo et Go.

Exercice 1.2. Les tailles des fichiers seront données en Ko, Mo ou Go. Il existe une multitude de formats différents pour stocker un son. Entre autres :

* Le format standard non compressé (format `Wave`), où le son est :
  - enregistré en stéreo,
  - l'amplitude du signal sonore codée sur 16 bits,
  - échantillonné à 44,1 kHz (1 Hz = 1 valeur par seconde)
* Le format `MP3`, format `Wave` compressé pour pouvoir être joué en ne lisant que 128 kbits/s tout en conservant un signal sonore de bonne qualité.

(1) Quelle est la taille d'un fichier `Wave` standard de 3 minutes ?

(2) Quelle est la taille d'un fichier `MP3` de 3 minutes ?

(3) Quel est le taux de compression du format MP3 ?

(4) Un étudiant désire télécharger un morceau au format MP3. La taille du fichier correspondant est de 11,7 Mo. Son modem ne lui autorise qu'un débit moyen de 28 kbits/s. Combien de temps mettra-t-il pour télécharger le fichier ?

Chapitre 2

Portes

Un ordinateur est construit à partir de blocs de base appelés *portes logiques* ou simplement *portes*.

Une porte est un circuit ayant au moins une (et souvent plusieurs) *entrée* et exactement une *sortie*. Les valeurs des entrées et de la sortie sont les valeurs logiques *vrai* et *faux*. Dans le domaine de l'architecture de l'ordinateur, il est fréquent d'utiliser 1 pour *vrai* et 0 pour *faux*. Une porte n'a pas de *mémoire* ; la valeur de sa sortie dépend uniquement des valeurs présentes sur ses entrées. Par conséquent, il est possible de décrire complètement le comportement d'une telle porte par le biais d'une *table de vérité* (voir section 3.1).

Il est courant de considérer trois types fondamentaux de portes : porte-*et*, porte-*ou* et porte-*non* (ou *inverseur*).

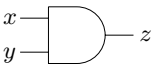
2.1 PORTES FONDAMENTALES

2.1.1 Porte-et

Une porte-*et* peut avoir un nombre arbitraire d'entrées. Sa sortie vaut 1 si et seulement si *toutes* les entrées valent 1. Donc, la sortie vaut 0 si et seulement si au moins une des entrées vaut 0. L'appellation *« et »* traduit le fait que la sortie vaut 1 si la première

entrée vaut 1 *et* la deuxième entrée vaut 1 *et* ... *et* la n-ième entrée vaut 1.

Dans les diagrammes représentant des circuits (portes et interconnexions entre les portes), la porte-*et* est dessinée de la manière suivante :



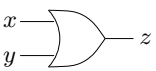
Voici la table de vérité d'une porte-*et* avec deux entrées :

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

2.1.2 Porte-ou

Comme la porte-*et*, la porte-*ou* peut avoir un nombre arbitraire d'entrées. La sortie vaut 1 si et seulement si *au moins* une des entrées vaut 1. Autrement dit, la sortie vaut 0 si et seulement si *toutes* les entrées valent 0. L'appellation *« ou »* vient du fait que la sortie vaut 1 si la première entrée vaut 1 *ou* la deuxième entrée vaut 1 *ou* ... *ou* la n-ième entrée vaut 1.

Dans les diagrammes de circuits, la porte-*ou* est dessinée de la façon suivante :



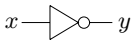
Voici la table de vérité d'une porte-*ou* avec deux entrées :

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

2.1.3 Inverseur

Un *inverseur* a exactement une entrée et une sortie. Sa sortie vaut 1 si et seulement si l'entrée vaut 0. Sinon la sortie vaut 0. Autrement dit, la valeur de la sortie est exactement l'inverse de la valeur de l'entrée.

Dans les diagrammes de circuits, l'inverseur est dessiné de la façon suivante :



Voici la table de vérité de l'inverseur :

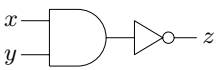
x	y
0	1
1	0

2.2 PORTES COMBINÉES

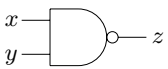
Souvent, il est pratique de combiner plusieurs fonctions de base dans une seule porte plus complexe, par exemple afin de conserver l'espace de dessin dans un diagramme de circuits. Dans cette section, nous présentons quelques portes combinées fréquemment utilisées avec leur table de vérité.

2.2.1 Porte-non-et

La porte-*non-et* est une porte-*et* avec un inverseur sur la sortie. Donc, au lieu de dessiner l'enchaînement de portes suivant,



on dessine une porte-*et* avec un cercle sur la sortie, comme ceci :

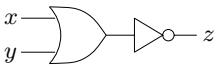


Comme la porte-*et*, la porte-*non-et* peut avoir un nombre arbitraire d'entrées. La table de vérité de la porte *non-et* est similaire à celle de la porte-*et* sauf que toutes les valeurs de la sortie sont inversées :

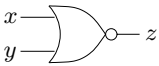
x	y	z
0	0	1
0	1	1
1	0	1
1	1	0

2.2.2 Porte-non-ou

La porte-*non-ou* est une porte-*ou* avec un inverseur sur la sortie. Donc, au lieu de dessiner l'enchaînement de portes suivant,



on dessine une seule porte-*ou* avec un cercle sur la sortie comme ceci :



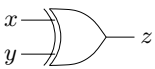
De même que la porte-*ou*, la porte-*non-ou* peut avoir un nombre arbitraire d'entrées. La table de vérité de la porte-*non-ou* est similaire à celle de la porte-*ou* sauf que la valeur de la sortie est inversée :

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

2.2.3 Porte-ou-exclusif

La porte-*ou-exclusif* est apparentée à la porte-*ou*. Elle peut avoir un nombre arbitraire d'entrées. Sa sortie vaut 1 si et seulement si *exactement une des entrées* vaut 1. Sinon la sortie vaut 0.

On dessine la porte-*ou-exclusif* comme ceci :



Voici la table de vérité d'une porte-*ou-exclusif* avec deux entrées :

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

2.3 COMBIEN EXISTE-T-IL DE PORTES ?

« Combien existe-t-il de types de portes différents » et « comment les appelle-t-on ? ».

Pour répondre partiellement à ces deux questions intéressantes, nous nous limitons à des portes à n entrées. La table de vérité d'une telle porte comporte 2^n lignes. La porte est complètement définie par les valeurs de la colonne de la table décrivant les valeurs de la sortie. On peut voir cette colonne comme un nombre binaire écrit avec 2^n chiffres binaires. Combien de nombres binaires peut-on représenter avec 2^n chiffres ? La réponse est 2^{2^n} car il y a 2^k suites différentes de longueur k et si $k = 2^n$, alors cela donne 2^{2^n} suites différentes. En particulier, si $n = 2$, on a 16 types différents de portes à 2 entrées.

La plupart de ces portes n'ont pas de nom et ne sont pas utilisées. Pour nous en rendre compte, nous allons examiner les 16 possibilités. Chaque entrée de la table suivante correspond à une *colonne* de sortie différente (de haut en bas) :

- `0000` la sortie de cette porte vaut toujours 0 ; on dit que la porte « ignore » ses entrées ; sa réalisation ne nécessite aucun circuit : il suffit de laisser les entrées non connectées et de connecter la sortie à la constante 0.
- `0001` correspond à la porte-*et* décrite précédemment.
- `0010` correspond à une porte-*et* avec un inverseur sur la deuxième entrée.
- `0011` cette porte ignore sa deuxième entrée et la valeur de sa sortie est identique à celle de sa première entrée ; sa réalisation ne nécessite aucun circuit : il suffit de laisser la deuxième entrée non connectée et de connecter la sortie à la première entrée.
- `0100` correspond à une porte-*et* avec un inverseur sur la première entrée.
- `0101` cette porte ignore sa première entrée et la valeur de sa sortie est identique à celle de la deuxième entrée ; sa réalisation ne nécessite aucun circuit : il suffit de laisser la première entrée non connectée et de connecter la sortie à la deuxième entrée.
- `0110` correspond à la porte-*ou-exclusif* décrite précédemment.
- `0111` correspond à la porte-*ou* décrite précédemment.
- `1000` correspond à la porte-*non-ou* décrite précédemment.

– `1001` correspond à une porte-*ou-exclusif* avec un inverseur sur la sortie.
– `1010` cette porte peut être construite avec un inverseur sur la deuxième entrée et avec la première entrée non connectée.
– `1011` correspond à une porte-*ou* avec un inverseur sur sa deuxième entrée.
– `1100` cette porte peut être construite avec un inverseur sur la première entrée et la deuxième entrée non connectée.
– `1101` correspond à une porte-*ou* avec un inverseur sur la première entrée.
– `1110` correspond à la porte-*non-et* décrite précédemment
– `1111` la sortie de cette porte vaut toujours 1 : cette porte ignore ses entrées ; sa réalisation ne nécessite aucun circuit. Il suffit de laisser les entrées non connectées et de connecter la sortie à la constante 1.

Comme on peut le constater, plusieurs de ces portes sont inutiles.

2.4 TOUT FAIRE AVEC UN SEUL TYPE DE PORTE

Il est possible de construire n'importe quel type de porte en n'utilisant que des portes-*non-et*. Pour cela, on observe d'abord qu'un inverseur est la même chose qu'une porte-*non-et* à une seule entrée. Puis, on observe qu'une porte-*et* peut être construite grâce à une porte-*non-et* avec un inverseur sur la sortie. Finalement, une porte-*ou* peut être construite grâce à une porte-*non-et* avec un inverseur sur chaque entrée.

Avec certaines technologies, il est en fait *plus facile* de construire une porte-*non-et* que n'importe quel autre type de porte. Dans ce cas, la porte-*non-et* est considérée comme le bloc de base et les autres types doivent être construits à partir de celle-ci.

De même, toutes les portes peuvent être construites à partir de portes-*non-ou* exclusivement.

L'inverseur est une porte-*non-ou* à une seule entrée. La porte-*ou* est une porte-*non-ou* avec un inverseur sur la sortie. Finalement, la porte-*et* peut être construite grâce à une porte-*non-ou* avec un inverseur sur chaque entrée.

2.5 RÉCAPITULATIF

Bien qu'une porte puisse être arbitrairement compliquée, désormais, lorsque nous utilisons le mot *porte*, nous parlerons de l'une des portes-*et*, *ou*, *inverseur*, *non-et*, *non-ou* et parfois *ou-exclusif*.

Pourquoi exclure les autres portes ? La raison principale est que nous souhaitons que le nombre de portes d'un circuit reflète le coût de sa fabrication. Si on autorise des portes arbitrairement compliquées, on peut alors construire un circuit combinatoire arbitrairement compliqué avec une seule porte.

La restriction ci-dessus n'est pas idéale car, pour que le nombre de portes reflète le coût, il faudrait tenir compte de la technologie de fabrication. Chaque technologie permet de fabriquer plus ou moins facilement les différentes portes. C'est dans le but de rester indépendant de la technologie choisie que nous utilisons cette approximation.

EXERCICES

Exercice 2.1. Réaliser une porte-*non-et* à trois entrées à partir de portes-*ou* à deux entrées et d'inverseurs.

Exercice 2.2. En n'utilisant que des portes-*non-ou* à deux entrées, réaliser une porte-*et* à trois entrées.

Exercice 2.3. Peut-on construire toutes les portes à deux entrées à partir de portes-*ou* et portes-*et* à deux entrées exclusivement ?

Chapitre 3

Circuits combinatoires

Un *circuit combinatoire* est une porte généralisée (voir chapitre 2). De manière générale, un tel circuit a m entrées et n sorties.

Un circuit combinatoire avec n sorties peut toujours être construit sous la forme de n circuits différents possédant chacun une seule sortie. Pour cette raison, la méthode de construction proposée ici est valable uniquement pour un circuit à une seule sortie. En réalité, cette méthode n'est pas forcément optimale. En effet, on peut souvent utiliser une même porte calculant un signal intermédiaire dans plusieurs circuits, mais cette possibilité n'est en général pas prise en compte ici.

Lorsque nous construisons un circuit à partir d'une spécification quelconque, nous essayons toujours d'obtenir le meilleur résultat possible. Le problème est la définition de « meilleur résultat ». Dans certaines applications, on souhaite minimiser le nombre de portes logiques (ou plutôt le nombre de transistors). Dans d'autres applications, on peut vouloir minimiser le retard du signal afin d'obtenir un circuit le plus rapide possible, ou encore réduire la consommation d'énergie. En général, un mélange de ces critères doit être appliqué.

La minimisation du nombre de portes logiques d'un circuit est un problème difficile du point de vue de la complexité de calcul. Les

programmes d'ordinateurs conçus pour ce type de réduction utilisent des *heuristiques* pour améliorer les temps de calcul. Dans ce texte, nous ne nous intéressons pas à l'optimalité des circuits. Nous ne présentons que des méthodes générales, qui fonctionnent pour n'importe quel circuit, mais qui ne sont pas souvent optimales.

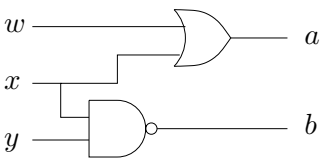
Différents moyens permettant de spécifier le comportement exact d'un circuit combinatoire ; ce sont par exemple les expressions logiques (voir chapitre 4) ou, comme présenté dans la section suivante, les *tables de vérité*.

3.1 TABLE DE VÉRITÉ

Une table de vérité est une énumération complète de toutes les combinaisons des valeurs des entrées du circuit avec, pour chacune d'elles, la valeur des sorties associées.

3.1.1 Description d'un circuit existant

Lorsqu'elle est utilisée afin de décrire un circuit existant, la table de vérité contient des valeurs 0 et 1 pour les sorties. Supposons par exemple que nous souhaitions établir la table de vérité du circuit suivant :



Afin d'y parvenir, il suffit de calculer, pour chaque combinaison de valeurs des entrées, la valeur de la sortie. On obtient alors la table suivante :

w	x	y	a	b
0	0	0	0	1
0	0	1	0	1
0	1	0	1	1
0	1	1	1	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	0

3.1.2 Spécification d'un circuit à construire

Lorsque la table de vérité est utilisée pour spécifier un circuit à construire, on peut avoir des combinaisons de valeurs des entrées pour lesquelles les sorties correspondantes ne sont pas spécifiées, peut-être parce que cette combinaison des entrées ne peut pas se produire dans l'application visée. Nous indiquons une *sortie non spécifiée* par le caractère « - ».

Par exemple, supposons que nous souhaitions un circuit à quatre entrées, interprétées comme deux entiers naturels binaires, chacun sur deux bits, et deux sorties interprétées comme le quotient sur deux bits des deux entiers en entrée. Puisque le quotient n'est pas défini lorsque le dénominateur vaut 0, la valeur de la sortie est indifférente dans ce cas. Parmi les 16 combinaisons de la table, 4 correspondent au cas du dénominateur nul. Voici la table :

x_1	x_0	y_1	y_0	z_1	z_0
0	0	0	0	-	-
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	-	-
0	1	0	1	0	1
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	-	-
1	0	0	1	1	0
1	0	1	0	0	1
1	0	1	1	0	0
1	1	0	0	-	-
1	1	0	1	1	1
1	1	1	0	0	1
1	1	1	1	0	1

Des sorties non spécifiées peuvent permettre de diminuer considérablement le nombre de transistors nécessaires à la construction du circuit. L'explication est simple : si nous sommes libres de choisir la valeur de sortie dans une situation particulière, nous choisissons celle qui donne le moins de portes logiques.

Dans ce livre, le problème de la minimalité n'est pas pris en compte. Mais pour des raisons conceptuelles, il est intéressant de ne pas préciser les valeurs des sorties quand elles n'ont pas d'importance.

3.2 CONCEPTION D'UN CIRCUIT COMBINATOIRE

Avec notre méthode, pour réaliser un circuit combinatoire à m entrées et n sorties, il est nécessaire de disposer d'une table de vérité. Si celle-ci n'est pas donnée, à savoir si la spécification du circuit à construire n'est pas sous la forme d'une table de vérité, il faut commencer par la créer. Un circuit à m entrées et une seule sortie sera construit pour chacune des n *colonnes* de sortie de la table.

Le circuit construit a une structure très régulière. Il est organisé en deux *couches*, la première couche ayant au plus 2^m portes-*non-et*, chacune à m entrées, et la deuxième une seule porte *non-et* avec autant d'entrées que de portes dans la première couche.

Pour chaque ligne de la table telle que la sortie vaut 1, nous mettons une porte-*non-et* à m entrées. Chaque signal d'entrée valant 1 sur la ligne en question est branché directement sur une entrée de la porte-*non-et*, et chaque signal d'entrée valant 0 est branché directement sur la porte-*non-et* à travers un *inverseur*.

Puis, la sortie de chaque porte-*non-et* de la première couche est branchée sur une entrée de la porte-*non-et* de la deuxième couche. Ceci termine la construction.

Voici un exemple d'utilisation de cette méthode de construction de circuit combinatoire. Nous commençons par la table de vérité suivante (dans laquelle « - » indique une valeur sans importance) :

x	y	z	a	b
0	0	0	-	0
0	0	1	1	1
0	1	0	1	-
0	1	1	0	0
1	0	0	0	1
1	0	1	0	-
1	1	0	-	-
1	1	1	1	0

La première étape consiste à choisir des valeurs pour les sorties non spécifiées. Avec notre méthode simple, la meilleure solution est de systématiquement mettre 0 à la place de « - ». Voici la table ainsi obtenue :

x	y	z	a	b
0	0	0	0	0
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	0

Maintenant, il faut construire deux circuits différents, chacun ayant une seule sortie. Le premier correspond à la colonne marquée a dans la table, et le deuxième correspond à la colonne marquée b.

Pour la colonne a, la première couche contiendra trois portes-*non-et* (car il y a trois lignes telles que la valeur de la colonne a est égale à 1). Chaque porte aura trois entrées, correspondant aux signaux x, y et z. La deuxième couche aura une porte-*non-et* à trois entrées, car il y a trois portes-*non-et* dans la première couche.

Le circuit complet pour la colonne a est illustré par la figure 3.1.

Pour la colonne b, on obtient deux portes-*non-et* chacune avec trois entrées pour la première couche, et une porte-*non-et* avec deux entrées pour la deuxième couche. Il y a deux portes pour la première couche, car il y a deux lignes dans la table de vérité telles que la valeur de la colonne b est égale à 1. La porte de la deuxième couche aura deux entrées car il y a deux portes dans la première couche.

Le circuit complet pour la deuxième colonne est illustré par la figure 3.2.

Maintenant, il ne reste plus qu'à combiner les deux circuits en un seul comme présenté à la figure 3.3.

Ce circuit est correct mais pas optimal dans le sens qu'il existe un circuit qui fonctionne de manière identique avec moins de portes. En fait, puisque les deux colonnes de la table de vérité ont la valeur 1

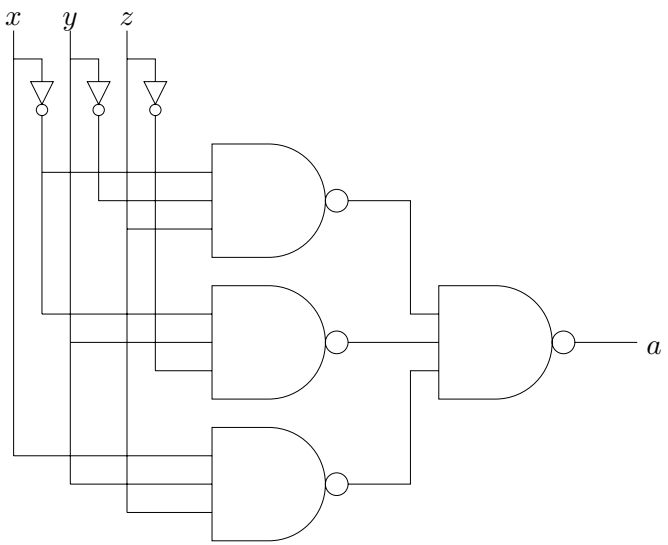


Figure 3.1 : Circuit combinatoire pour *a*

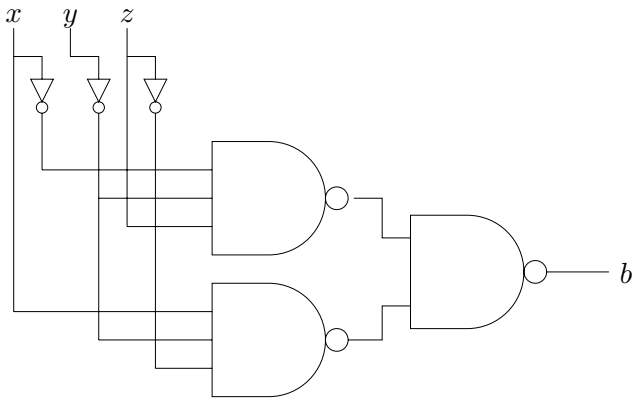


Figure 3.2 : Circuit combinatoire pour *b*

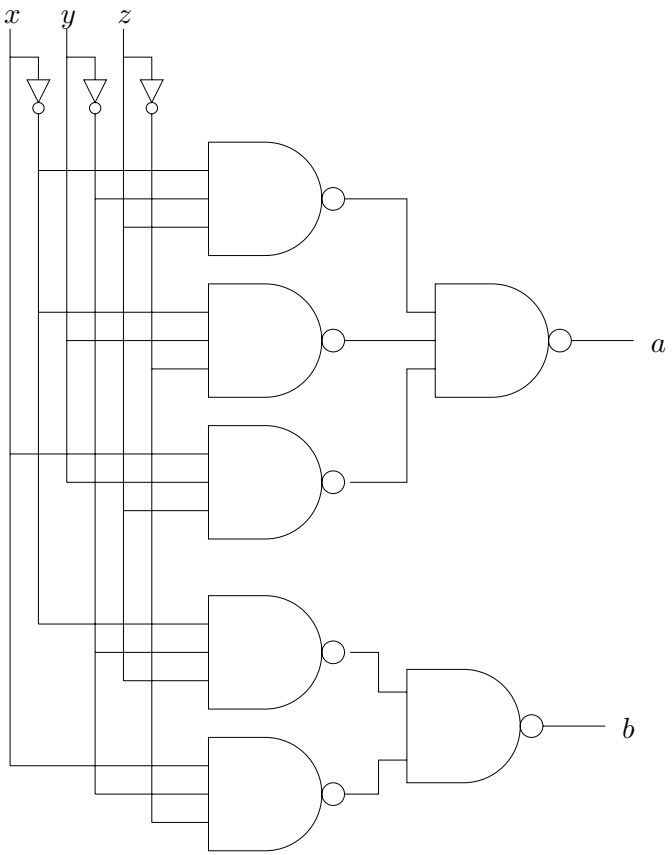


Figure 3.3 : Circuit complet

pour les valeurs $x = 0$, $y = 0$ et $z = 1$ des entrées, il est clair qu'une seule porte-*non-et* correspondant à cette combinaison de valeurs sur les entrées peut être partagée par les deux sous-circuits. On obtient alors le circuit illustré par la figure 3.4.

Ce n'est pas la seule façon de diminuer la complexité d'un circuit. Dans certains cas, on peut diminuer le nombre de portes-*non-et* de la première couche ou encore le nombre d'entrées de certaines de ces portes. Finalement, on peut souvent encore diminuer le nombre de portes si on est prêt à accepter un nombre de couches supérieur à 2.

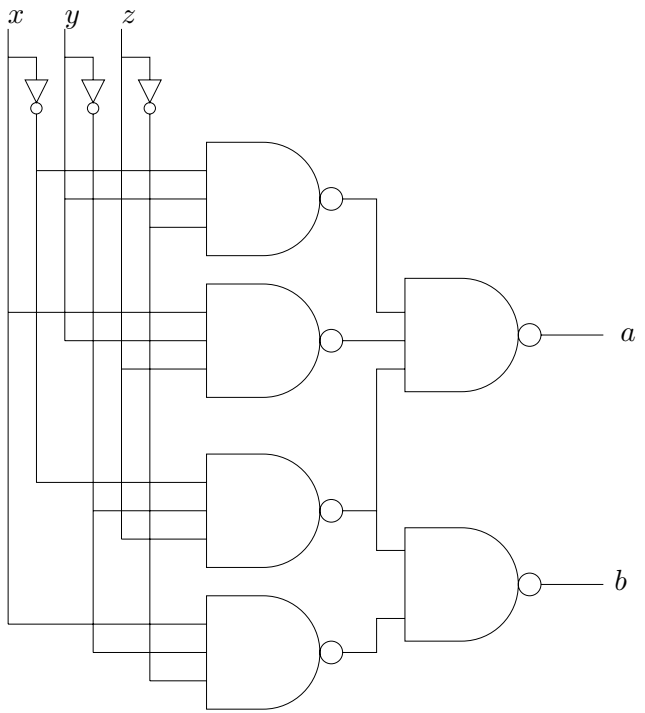


Figure 3.4 : Circuit complet simplifié

3.3 RÉCAPITULATIF

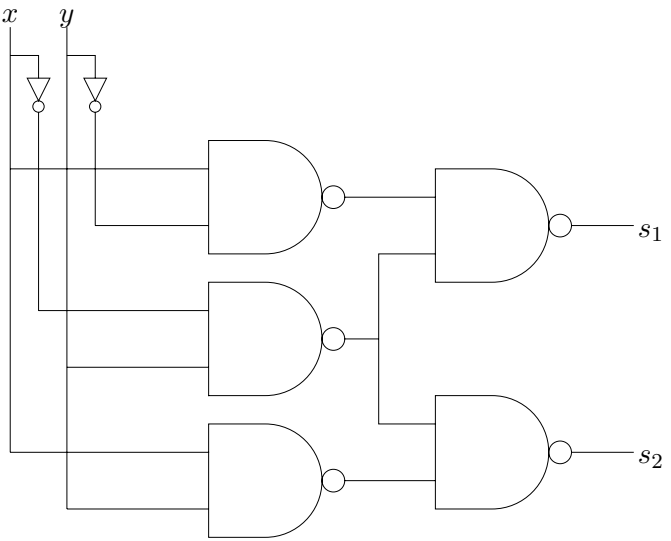
Un circuit combinatoire à m entrées et n sorties réalise n fonctions logiques à m arguments. Une table de vérité permet de décrire un tel circuit soit pour la conception d'un nouveau circuit, soit pour l'analyse d'un circuit existant. Pour créer un circuit à partir de sa table de vérité, une méthode générale donne un circuit en deux couches, la première ayant au plus 2^m portes-*non-et*, chacune à m entrées, et la deuxième une seule porte *non-et* avec autant d'entrées que de portes dans la première couche. Dans beaucoup de cas, un circuit plus simple peut être obtenu en tirant parti du fait que certaines entrées ne sont pas spécifiées ou en utilisant les même portes dans la réalisation des diverses fonctions.

EXERCICES

Exercice 3.1. Donner un circuit correspondant à la table de vérité suivante :

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

Exercice 3.2. Donner la table de vérité correspondant au circuit ci-dessous :



Exercice 3.3. Donner un circuit correspondant à la table de vérité suivante :

x	y	z	a	b
0	0	0	1	1
0	0	1	1	0
0	1	0	-	0
0	1	1	-	-
1	0	0	0	0
1	0	1	0	0
1	1	0	0	-
1	1	1	0	1

Chapitre **4**

Expressions logiques

La spécification d'un circuit combinatoire (voir chapitre 3) peut être faite grâce à une *expression logique*, ou simplement une *expression* qui utilise les constantes 0 et 1, des variables comme x, y et z (parfois indicées) pour les noms des entrées et enfin les opérateurs « $+$ » (pour *ou*), « $\cdot$ » (pour *et*) qui sera souvent omis et remplacé par la juxtaposition et « $\overline{\ }$ » placé au-dessus d'une expression (pour *non*). Comme souvent, la multiplication (ici le *et*) est prioritaire par rapport à l'addition. Des parenthèses peuvent être employées pour modifier la priorité.

Exemples :

$$(\overline{x} + y)\overline{(y + x\,\overline{z})} + x\,\overline{y\,z}$$

$$x_1\,\overline{x_2}\,x_3 + \overline{x_1}\,\overline{x_2}\,x_3 + \overline{x_1}\,x_2\,\overline{x_3}$$

$$(x + 1)(y + 0\,z)$$

4.1 CORRESPONDANCE ENTRE EXPRESSION ET CIRCUIT

À chaque expression correspond directement un circuit combinatoire. Par exemple, aux expressions ci-dessus correspondent les circuits suivants :

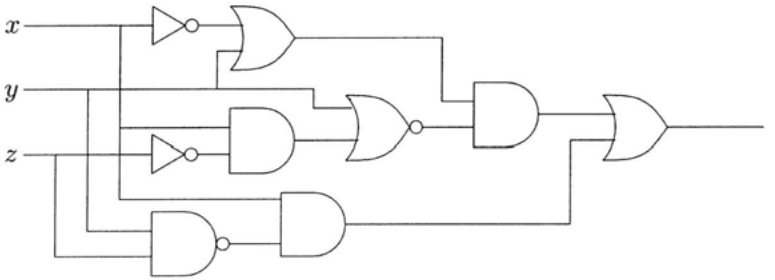


Figure 4.1 : Circuit pour la première expression

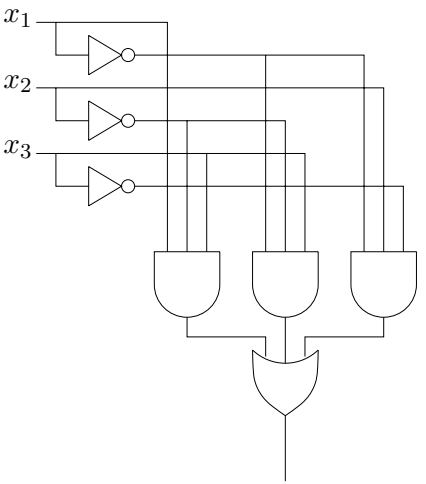


Figure 4.2 : Circuit pour la deuxième expression

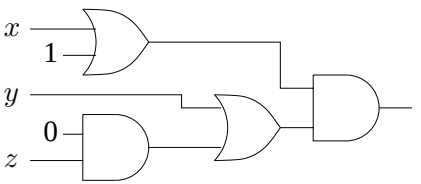


Figure 4.3 : Circuit pour la troisième expression

4.2 PUISSANCE DES EXPRESSIONS

Les expressions logiques sont-elles capables de décrire n'importe quel circuit combinatoire ? La réponse est oui ; en voici la justification : il est trivial de convertir la table de vérité (voir chapitre 3.1) de n'importe quel circuit en une expression logique. L'expression qui en résulte aura la forme d'une somme de produits des variables d'entrées ou de leur inverse. Chaque ligne de la table dont la sortie vaut 1 correspond à un terme de la somme. Dans un tel terme, une variable valant 1 dans la table de vérité figure sans inversion, tandis qu'une variable valant 0 figure inversée.

Prenons la table de vérité suivante :

x	y	z	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

L'expression correspondante est : $\overline{x}\,y\,\overline{z} + x\,\overline{y}\,\overline{z} + x\,y\,z$.

Puisqu'il est possible de décrire n'importe quel circuit combinatoire avec une table de vérité et de décrire n'importe quelle table de vérité par une expression, on peut donc décrire n'importe quel circuit par une expression.

Alors qu'il est possible de *décrire* un circuit combinatoire existant avec une expression logique, une telle expression est moins pratique pour *spécifier* un circuit à construire. La raison est que l'expression ne permet pas d'exprimer des combinaisons d'entrées sans importance. Une expression logique a une valeur précise pour chacune des combinaisons.

4.3 SIMPLICITÉ DES EXPRESSIONS LOGIQUES

Il y a plusieurs expressions logiques (et donc plusieurs circuits) correspondant à une table de vérité donnée, et donc à une certaine fonction calculée. Par exemple, les deux expressions suivantes calculent

la même fonction :

$$x\,(y+z)$$
$$(x\,y+x\,z)$$

La première nécessite deux portes, une porte-*et* et une porte-*ou*. La deuxième nécessite deux portes-*et* et une porte-*ou*. Il semblerait évident que la première est préférable à la deuxième. Or, cela n'est pas forcément le cas. Le nombre de portes logiques n'est pas le seul, ni forcément le meilleur critère raisonnable de simplicité.

Nous avons, par exemple, implicitement supposé que les portes logiques sont idéales, c'est-à-dire que les valeurs des sorties sont calculées de manière instantanée. En réalité un signal met un certain temps pour traverser une porte ; cette durée s'appelle le *retard* de la porte. On peut considérer qu'un circuit est plus simple si son temps de retard est plus court. Dans ce cas, il est intéressant de réaliser des circuits en minimisant le nombre de portes à traverser entre les entrées et les sorties. Les circuits obtenus avec un tel critère ne sont pas forcément les mêmes que ceux obtenus en minimisant le nombre de total de portes logiques.

4.4 RÉCAPITULATIF

Une expression logique peut parfaitement décrire le fonctionnement d'un circuit combinatoire. En utilisant la structure de l'expression, il est même possible de déduire la structure du circuit correspondant, mais cela donne souvent un circuit non optimal. Une expression est moins pratique pour établir la spécification d'un circuit à réaliser, car incapable d'exprimer des combinaisons d'entrées sans importance.

EXERCICES

Exercice 4.1. Soit l'opération binaire xor réalisée par la porte-*ou-exclusif*.

(1) Montrer l'associativité de l'opération xor.

(2) Quelle relation y a-t-il entre $\overline{x}\ xor\ y$, $x\ xor\ \overline{y}$ et $\overline{x\ xor\ y}$?

(3) Donner une interprétation de $x\ xor\ y\ xor\ c$, puis une interprétation de $x_1\ xor\ x_2\ xor\ \ldots xor\ x_n$.

Exercice 4.2. On considère les fonctions f et g définies par les tables de vérité suivantes :

x	y	z	f
0	0	0	1
0	1	1	1
1	1	1	1
1	0	0	1
1	0	1	1

x	y	z	t	g
0	0	0	0	1
0	0	1	0	1
0	1	0	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	1	1

(Seuls les arguments pour lesquels la fonction vaut 1 sont précisés.)

Exprimer chacune de ces fonctions sous forme d'expression logique en utilisant les ensembles d'opérateurs suivants :

(1) $\mathcal{C}_1 = \{and, or, not\}$;

(2) $\mathcal{C}_2 = \{and, xor, true\}$.

Chapitre 5

Circuits combinatoires classiques

5.1 MULTIPLEXEUR

Un *multiplexeur* est un circuit combinatoire (voir chapitre 3) à n entrées d'adresse et 2^n entrées de données. Les n entrées d'adresse sont interprétées comme un nombre binaire utilisé pour sélectionner une des entrées de données. Le multiplexeur a une seule sortie, ayant la même valeur que l'entrée sélectionnée.

La taille de la table de vérité d'un multiplexeur est très grande sauf pour de très petites valeurs de n. C'est pourquoi on utilise une abréviation de la table de vérité dans laquelle certaines entrées sont remplacées par « - » pour indiquer que la valeur de l'entrée n'a pas d'importance.

La figure 5.1 montre une telle table pour un multiplexeur avec $n = 3$. La table complète aurait $2^{(3+2^3)} = 2\,048$ lignes.

On peut encore diminuer la taille de la table en utilisant des lettres pour indiquer certaines valeurs d'entrée, comme présenté figure 5.2.

De la même manière que la table de vérité du multiplexeur, le circuit peut aussi être simplifié. Notre méthode générale de fabrication de circuits combinatoires donnerait un circuit avec un grand nombre de portes. Le circuit simplifié est présenté figure 5.3.

a_2	a_1	a_0	d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0	x
0	0	0	-	-	-	-	-	-	-	0	0
0	0	0	-	-	-	-	-	-	-	1	1
0	0	1	-	-	-	-	-	-	0	-	0
0	0	1	-	-	-	-	-	-	1	-	1
0	1	0	-	-	-	-	-	0	-	-	0
0	1	0	-	-	-	-	-	1	-	-	1
0	1	1	-	-	-	-	0	-	-	-	0
0	1	1	-	-	-	-	1	-	-	-	1
1	0	0	-	-	-	0	-	-	-	-	0
1	0	0	-	-	-	1	-	-	-	-	1
1	0	1	-	-	0	-	-	-	-	-	0
1	0	1	-	-	1	-	-	-	-	-	1
1	1	0	-	0	-	-	-	-	-	-	0
1	1	0	-	1	-	-	-	-	-	-	1
1	1	1	0	-	-	-	-	-	-	-	0
1	1	1	1	-	-	-	-	-	-	-	1

Figure 5.1 : Table de vérité du multiplexeur

a_2	a_1	a_0	d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0	x
0	0	0	-	-	-	-	-	-	-	c	c
0	0	1	-	-	-	-	-	-	c	-	c
0	1	0	-	-	-	-	-	c	-	-	c
0	1	1	-	-	-	-	c	-	-	-	c
1	0	0	-	-	-	c	-	-	-	-	c
1	0	1	-	-	c	-	-	-	-	-	c
1	1	0	-	c	-	-	-	-	-	-	c
1	1	1	c	-	-	-	-	-	-	-	c

Figure 5.2 : Table simplifiée

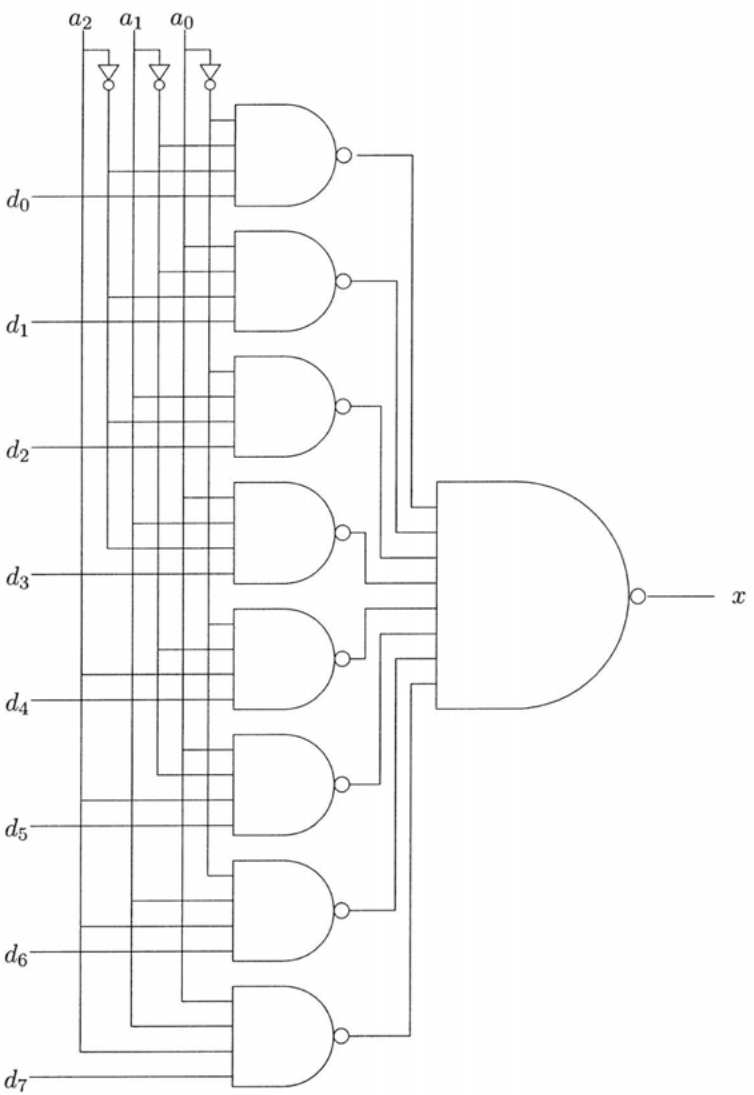


Figure 5.3 : Multiplexeur

5.2 DÉMULTIPLEXEUR

Le démultiplexeur est l'inverse du multiplexeur. Il a une seule entrée de donnée et n entrées d'adresse. Il a 2^n sorties. Les entrées d'adresse déterminent parmi les 2^n sorties celle qui a la même valeur que l'entrée de donnée. Les autres sorties ont la valeur 0.

La table de vérité d'un démultiplexeur avec $n = 3$ est illustrée par la figure 5.4. Nous aurions pu donner la table complète, car elle n'a que 16 lignes, mais nous utilisons la même convention que pour le multiplexeur (voir section 5.1) où une abréviation est utilisée pour les entrées.

a_2	a_1	a_0	d	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
0	0	0	c	0	0	0	0	0	0	0	c
0	0	1	c	0	0	0	0	0	0	c	0
0	1	0	c	0	0	0	0	0	c	0	0
0	1	1	c	0	0	0	0	c	0	0	0
1	0	0	c	0	0	0	c	0	0	0	0
1	0	1	c	0	0	c	0	0	0	0	0
1	1	0	c	0	c	0	0	0	0	0	0
1	1	1	c	c	0	0	0	0	0	0	0

Figure 5.4 : Table de vérité du démultiplexeur

Un diagramme de circuit possible pour le démultiplexeur est illustré par la figure 5.5.

5.3 DÉCODEUR

Le multiplexeur et le démultiplexeur contiennent une partie dont le but est de *décoder* l'adresse en entrée, à savoir d'associer un nombre binaire de n chiffres à une sortie parmi 2^n, laquelle vaudra 1 tandis que toutes les autres vaudront 0.

Il est parfois avantageux de séparer cette fonction du reste du circuit, car elle est utile dans d'autres applications. Nous obtenons donc un autre circuit combinatoire que nous appelons *décodeur*. Sa table de vérité (pour $n = 3$) est illustrée par la figure 5.6.

Un circuit réalisant cette table est présenté figure 5.7.

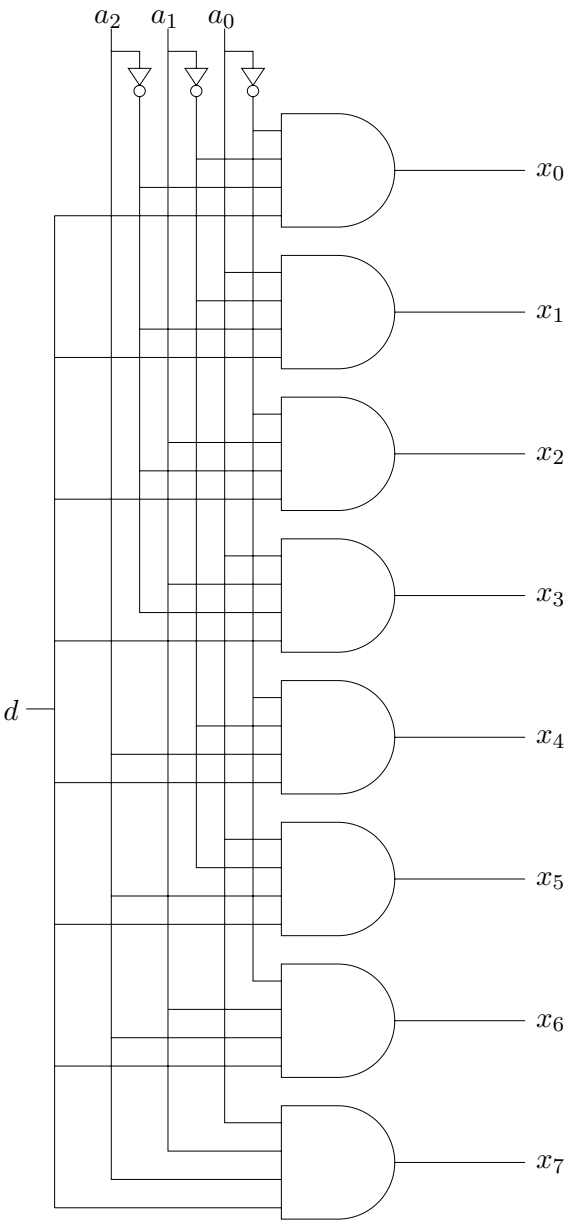


Figure 5.5 : Démultiplexeur

a_2	a_1	a_0	d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Figure 5.6 : Table de vérité du décodeur

5.4 RÉCAPITULATIF

Pour un certain nombre de circuits combinatoires classiques, la méthode générale de réalisation est inadaptée, car elle donne souvent des circuits d'une taille beaucoup trop grande. Pour ces circuits, nous utilisons des solutions optimisées une fois pour toutes.

EXERCICES

Exercice 5.1. Écrire la table de vérité d'un multiplexeur 4-bits (avec 4 entrées de données). Le réaliser avec des portes logiques *non, et* et *ou*.

Exercice 5.2. Réaliser un multiplexeur 16-bits qui donne en sortie la valeur d'une des entrées $x_{15}, x_{14} \dots x_0$ en fonction de l'entrée a_3, a_2, a_1, a_0. Combien de lignes faudrait-il dans la table de vérité ? Le réaliser en utilisant des multiplexeurs 4-bits.

Exercice 5.3. Réaliser un circuit comparateur de deux nombres binaires à n bits (voir chapitre 6 pour la notion de nombre binaire).

Entrées : $a_{n-1}, \dots, a_0$ et $b_{n-1}, \dots b_0$.

Sorties :

- $y_1, y_2 = 0, 0$ si $a = b$
- $y_1, y_2 = 1, 0$ si $a > b$
- $y_1, y_2 = 0, 1$ si $a < b$

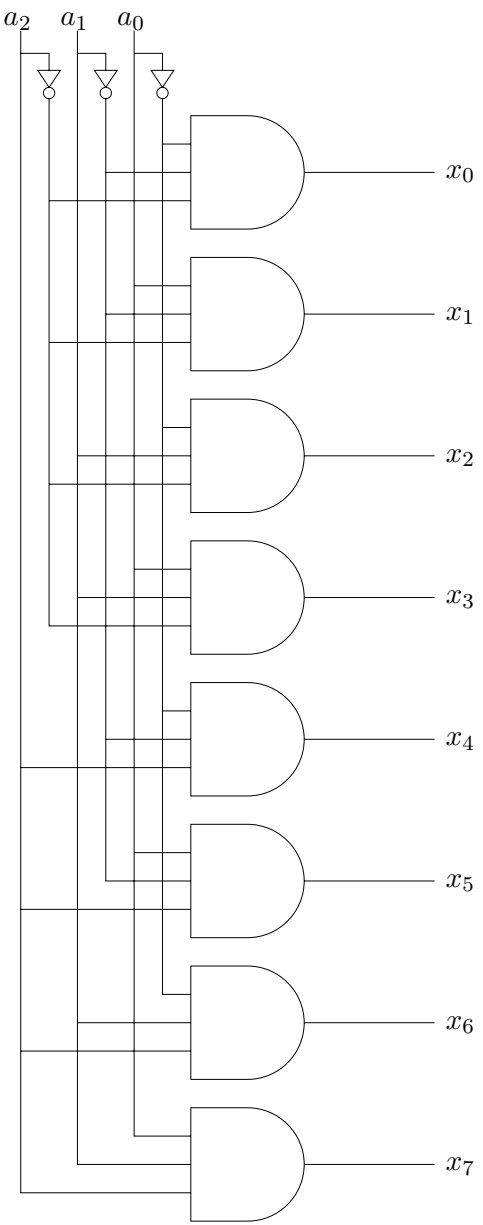


Figure 5.7 : Décodeur

Chapitre 6

Arithmétique binaire

Les circuits logiques sont notamment utilisés pour le calcul d'opérations mathématiques diverses comme l'addition, la multiplication, les opérations trigonométriques, etc. Il est donc nécessaire de disposer d'une représentation des nombres en tant que données binaires.

6.1 ENTIERS NATURELS

Les nombres les plus simples à représenter sont les nombres entiers naturels. Pour cela, il suffit de rappeler qu'un nombre qui s'écrit 2 034 en base 10 est interprété comme :

$$2 * 10^3 + 0 * 10^2 + 3 * 10^1 + 4 * 10^0$$

Nous pouvons également utiliser la base 2 dans laquelle un chiffre vaut soit 0 soit 1, ce que nous pouvons respectivement représenter par *faux* et *vrai*. En fait, nous avons déjà indiqué cette possibilité en utilisant 0 à la place de *faux*, et 1 à la place de *vrai*.

Tous les algorithmes applicables en base 10 s'adaptent à l'arithmétique binaire, cette dernière étant souvent plus simple.

Pour additionner deux nombres, il suffit de remarquer qu'une retenue de 1 est produite chaque fois que deux ou trois 1 sont additionnés :

$$\begin{array}{cccccc} & 1 & & 1 & 1 & \\ & - & & - & - & \\ & & 1 & 0 & 1 & 1 \\ + & & 1 & 0 & 0 & 1 \\ \hline & 1 & 0 & 1 & 0 & 0 \end{array}$$

La soustraction n'est pas plus difficile :

$$\begin{array}{cccccc} & & 10 & 10 & & \\ & & - & - & & \\ & 1 & 0 & 0 & 1 & \\ - & & 1 & 1 & 0 & \\ \hline & 0 & 0 & 1 & 1 & \end{array}$$

L'algorithme de multiplication est beaucoup plus simple que sa version décimale, car il suffit de pouvoir multiplier par 0 (ce qui ne donne que des zéros), ou par 1 (ce qui donne le nombre d'origine) :

$$\begin{array}{ccccccc} & & & 1 & 1 & 0 & 1 \\ & * & & & 1 & 0 & 1 \\ \hline & & & 1 & 1 & 0 & 1 \\ & & 0 & 0 & 0 & 0 & \\ & 1 & 1 & 0 & 1 & & \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$$

Finalement, la division est une simple succession de soustractions comme en arithmétique en base 10 :

$$\begin{array}{cccc|cc} 1 & 1 & 0 & 1 & 1 & 0 \\ \cline{5-6} 1 & 0 & & & 1 & 1 \quad 0 \\ \cline{1-2} 0 & 1 & 0 & & & \\ & 1 & 0 & & & \\ \cline{2-3} & 0 & 0 & 1 & & \end{array}$$

6.2 ENTIERS RELATIFS

Les choses se compliquent si nous souhaitons représenter des entiers négatifs.

On pourrait utiliser la technique qui consiste à représenter séparément le signe ($+$ ou $-$) de la valeur absolue. En représentation binaire, il suffirait alors de consacrer un bit supplémentaire au signe.

Pour additionner ou soustraire deux entiers, il faudrait alors deux circuits, un pour l'addition et un pour la soustraction d'entiers naturels. La combinaison des signes des deux opérandes déterminerait le circuit à employer.

Bien que cette méthode soit valide, il en existe une autre beaucoup plus adaptée aux circuits électroniques. Il s'agit de la méthode du *complément à deux* dans laquelle *un seul et même* circuit sert à la fois à l'addition et à la soustraction.

Pour comprendre le fonctionnement de la méthode du complément à deux, examinons d'abord comment cela fonctionne en base 10, avec une taille infinie. Cette méthode s'applique à n'importe qu'elle base avec peu de modifications ; elle s'applique aussi au cas fini.

6.2.1 Complément à dix avec précision infinie

Imaginons l'odomètre mécanique (compteur) d'une voiture. Il contient un certain nombre de roues, de dix chiffres chacune. Lorsqu'une roue passe de 9 à 0, celle qui est située immédiatement à sa gauche avance d'une position. Si *cette* roue est déjà en position 9, elle passe à 0 et la roue à sa gauche avance, etc. Supposons que la voiture recule. Le contraire se produit, à savoir que lorsqu'une roue passe de 0 à 9, celle qui est située à sa gauche recule d'une position.

Maintenant, supposons que nous ayons un odomètre avec une infinité de roues. Nous l'utilisons pour représenter l'ensemble des entiers relatifs.

Lorsque toutes les roues sont en position 0, nous interprétons la valeur de l'odomètre comme l'entier 0.

Un entier positif n est représenté par la position de l'odomètre obtenue en avançant l'odomètre de n positions à partir de 0. On remarque que pour chaque nombre de ce type, il y aura un nombre infini de roues à gauche en position 0.

Un entier négatif $-n$ est représenté par la position de l'odomètre obtenue en reculant de n positions à partir de 0. On remarque que, pour chaque entier négatif, il y a un nombre infini de roues à gauche en position 9.

En fait, nous n'avons pas besoin d'une infinité de roues. Étant donné un entier, le nombre de roues nécessaire pour le représenter

est fini. Il suffit d'adopter la convention selon laquelle la roue la plus à gauche doit être dupliquée une infinité de fois pour retrouver la vraie position de l'odomètre. Cette représentation est utilisée par un certain nombre de langages de programmation capables de manipuler des entiers de taille arbitraire.

Bien que nous n'ayons besoin que d'un nombre fini de roues pour représenter un nombre quelconque, nous avons toujours besoin d'un nombre de roues *non borné*. Il est impossible de borner *a priori* le nombre de roues et de toujours être capable de représenter tous les nombres possibles. La différence entre fini et borné est subtile mais importante. Si nous avons besoin d'un nombre infini de roues pour représenter les entiers, il n'y a aucun espoir de les représenter dans la mémoire d'un ordinateur, car cela nécessiterait une quantité infinie de mémoire. Si nous n'avons besoin que d'un nombre non borné de roues, il est possible que la quantité de mémoire ne soit pas suffisante pour une application donnée, mais nous pouvons quand même stocker plusieurs entiers, chacun de taille finie. Puisque chaque programme a un temps d'exécution fini, il n'utilise qu'une quantité finie de mémoire ; si cette dernière est suffisamment grande, il se peut que le programme s'exécute correctement.

Maintenant, supposons que nous ayons un circuit d'addition, capable d'additionner deux entiers ayant potentiellement un nombre infini de chiffres significatifs. Par exemple, étant donné un entier commençant par une infinité de chiffres 9, le circuit l'interprète comme un nombre positif, alors qu'avec notre représentation, il l'interprêtera comme un nombre négatif. Prenons l'exemple des deux nombres . . .9998 (interprété comme -2) et . . .0005 (interprété comme 5). Le circuit commence par additionner 8 et 5 pour un résultat de 3 et une retenue de 1. Puis il additionne 9, 0 et la retenue 1 pour un résultat de 0 avec une retenue de 1. Pour toutes les autres positions (une infinité), le résultat sera 0 avec une retenue de 1. Le résultat final sera donc . . .0003. Celui-ci est correct même avec notre interprétation des nombres négatifs.

Certaines implémentations de quelques langages de programmation qui manipulent des entiers de taille arbitraire (Lisp par exemple) utilisent exactement cette représentation pour les nombre négatifs.

Terminons cette section par une méthode permettant de calculer la valeur absolue d'un entier négatif de notre représentation. Il suffit de remplacer chaque chiffre individuel par 9 moins la valeur d'origine,

puis d'additionner 1 au nombre ainsi obtenu. Par exemple le nombre . . .9998 devient 1 plus . . .0001, soit . . .0002. Cette méthode fonctionne dans les deux sens, à savoir qu'elle permet aussi de calculer l'opposé d'un nombre positif.

6.2.2 Complément à dix fini

La méthode de la section précédente s'applique presque aussi bien lorsque le nombre de roues de l'odomètre est fini. Le seul problème supplémentaire est qu'il existe un risque de débordement (en anglais *overflow* ou *underflow*).

Supposons que nous n'ayons qu'un nombre fixe de roues, disons 3. Dans ce cas, nous allons utiliser la convention suivante : lorsque la roue la plus à gauche montre un chiffre entre 0 et 4 inclus, alors nous avons un nombre positif égal à sa représentation. Si, par contre, la roue la plus à gauche montre un chiffre entre 5 et 9, nous avons un nombre négatif dont la valeur absolue peut être calculée en utilisant la méthode de la section précédente, sauf pour -500 (représenté par 500) car 500 n'est pas représentable avec trois chiffres).

Nous allons voir qu'un circuit capable d'additionner deux entiers positifs sur trois chiffres peut être utilisé pour additionner des entiers négatifs de notre représentation.

Supposons encore que nous souhaitions additionner -2 (représenté par 998) et $+5$ (représenté par 005). Notre circuit additionne les deux entiers positifs 998 et 005, ce qui donne 1 003. Mais puisque le circuit ne contient que trois chiffres, il va tronquer le résultat à 003, ce qui est bien la représentation de 3.

Dans quelles situations le circuit fini donne-t-il un résultat incorrect ? La réponse est relativement simple, sa preuve un peu plus compliquée. En fait, il y a débordement lorsqu'une tentative d'addition de deux entiers positifs donne la représentation d'un entier négatif, et lorsque celle visant à additionner deux entiers négatifs donne la représentation d'un entier positif.

Nous avons donc un circuit pour l'addition des entiers relatifs représentés en complément à dix : il suffit d'utiliser un circuit pour l'addition des entiers naturels, plus des circuits supplémentaires, pour vérifier :

– si les deux entiers sont positifs et le résultat est négatif, alors signaler un débordement (en anglais *overflow*),

– si les deux entiers sont négatifs et le résultat est positif, alors signaler un débordement (en anglais *underflow*).

6.2.3 Entiers en complément à deux et de taille finie

Dans les sections précédentes, nous avons utilisé la méthode du complément à dix pour représenter des entiers relatifs. Dans un ordinateur, on utilise la base 2 plutôt que la base 10. Heureusement, la même méthode décrite ci-dessus s'applique également à la base 2. Pour un additionneur n bits (n étant souvent 32 ou 64), nous pouvons représenter les entiers positifs avec le chiffre le plus significatif égal à 0, ce qui donne des valeurs entre 0 et $2^{(n-1)} - 1$, et les entiers négatifs avec le chiffre le plus significatif égal à 1, ce qui donne des valeurs entre $-2^{(n-1)}$ et -1.

Les règles de détection de débordement sont exactement les mêmes qu'en complément à dix. Par conséquent, lorsque deux entiers positifs sont additionnés et que le résultat commence par le chiffre 1, alors nous avons un débordement. De même, lorsque deux entiers négatifs sont additionnés et que le résultat commence par le chiffre 0, nous avons un débordement.

6.3 ENTIERS DE TAILLE ARBITRAIRE

Un inconvénient de la représentation de taille fixe est le risque de débordement. Pour éviter cela, on peut envisager une représentation des entiers avec une taille finie mais non bornée.

Chaque entier relatif peut être représenté avec une taille finie. Sa représentation sous la forme de suite de bits infinie commence soit par un nombre infini de chiffres 0 (nombre positif), soit par un nombre infini de chiffres 1 (nombre négatif). La partie significative qui contient à la fois des chiffres 0 et 1 est toujours finie. Pour représenter l'entier, il suffit alors de conserver le dernier chiffre de la suite infinie de 0 ou de 1 ainsi que la partie significative.

Alors que chaque nombre contient un nombre fini de bits, il est bien sûr impossible de borner *a priori* le nombre de bits nécessaire.

Peu d'architectures supportent les entiers de taille arbitraire. C'est par contre le cas de certaines implémentations de quelques langages de programmation (Lisp en particulier). Ces langages utilisent l'arithmétique de taille fixe du processeur pour obtenir des entiers de taille arbitraire.

6.4 NOMBRES RATIONNELS

Les entiers sont indispensables, mais nous avons parfois besoin de calculer en utilisant des nombres non entiers.

Une idée intéressante est d'utiliser des nombres rationnels. Plusieurs algorithmes importants, tels que l'*algorithme du simplexe* pour la programmation linéaire, donnent un résultat rationnel pour des entrées rationnelles.

La représentation des nombres rationnels dans un ordinateur ne pose pas de problème particulier. Il suffit de maintenir une paire d'entiers, un pour le numérateur et l'autre pour le dénominateur.

Cependant certaines restrictions supplémentaires s'imposent, par exemple :

– un nombre rationnel positif est toujours représenté sous la forme d'une paire d'entiers positifs (l'autre possibilité étant deux entiers négatifs),
– un nombre rationnel négatif est toujours représenté avec un numérateur négatif et un dénominateur positif (l'autre possibilité étant un numérateur positif et un dénominateur négatif),
– le numérateur et le dénominateur sont toujours premiers entre eux (il n'ont pas de facteurs communs).

Ces règles donnent une représentation *canonique*, à savoir que la représentation d'une nombre rationnel est unique même si, *a priori*, plusieurs représentations sont possibles.

Les circuits de l'arithmétique rationnelle doivent respecter ces règles. En particulier, la dernière règle implique la nécessité de diviser les deux entiers par leur plus grand facteur commun après chaque opération arithmétique afin d'obtenir la représentation canonique.

Comme pour les entiers de taille arbitraire, peu de processeurs offrent un support des nombres rationnels. La raison est que les deux composantes de ces nombres sont souvent très grandes, même si la valeur du nombre rationnel même ne l'est pas. Il est donc obligatoire de représenter les composants sous la forme d'entiers de taille arbitraire. Ces entiers n'ayant pas de support, les rationnels ne l'ont pas non plus. Par contre, c'est le cas de certaines implémentations de quelques langages de programmation (notamment Lisp).

6.5 NOMBRES FLOTTANTS

Au lieu d'employer la représentation des nombres rationnels présentée dans la section précédente, la plupart des ordinateurs représentent un sous-ensemble des nombres rationnels. Il s'agit des *nombres en virgule flottante* ou *nombres flottants*, ou simplement *flottants*.

Les opérations arithmétiques sur les flottants sont inexactes. En contrepartie, leur représentation est de taille fixe. Pour un grand nombre de calculs comme dans le domaine du calcul scientifique, (comme si d'autres calculs ne l'étaient pas), une telle représentation donne le plus souvent une précision adéquate.

Cependant, il y a des cas (parfois spectaculaires) où la précision se dégrade tant que les résultats sont erronés. Pour pallier ce problème s'est développée l'analyse numérique, discipline à part entière du domaine des mathématiques appliquées, dont le but est d'expliquer le comportement d'algorithmes numériques du point de vue de la précision numérique et d'inventer des algorithmes qui réduisent la dégradation.

L'idée principale de la représentation en virgule flottante est de représenter le nombre sous la forme d'une *mantisse* et d'un *exposant*, tous deux avec un nombre fixe de bits. Si on utilise la notation m pour la mantisse et e pour l'exposant, le nombre représenté sera $m \cdot 2^e$.

Comme dans le cas des rationnels, plusieurs couples de valeurs (mantisse, exposant) pourraient représenter le même nombre. Nous obtenons une représentation canonique en imposant que la mantisse m vérifie la condition $1 \leqslant m < 2$. La suite de chiffres binaires de la mantisse commence donc toujours par 1. Le chiffre initial étant toujours 1, on peut l'omettre de la représentation.

La raison du succès relatif de la représentation flottante pour le « calcul scientifique » est que ce type de calcul effectue en général plus de multiplications et divisions que d'additions et soustractions. La multiplication de deux nombres flottants est relativement simple. Il suffit de multiplier les mantisses et d'additionner les exposants. Le résultat peut être une mantisse dont la valeur est supérieure à 2. Afin d'obtenir une représentation canonique, la mantisse doit être divisée par 2 (décalée à droite) et l'exposant doit être incrémenté. La division est similaire.

L'imprécision du résultat d'une multiplication ou d'une division est due uniquement à l'imprécision des deux opérandes. Aucune erreur supplémentaire n'est introduite par l'opération elle-même (sauf peut-être d'une demi-unité dans le chiffre le moins significatif). Les opérations d'addition et de soustraction n'ont pas cette caractéristique.

Afin d'additionner deux nombres flottants, la mantisse de celui qui a l'exposant le plus petit doit d'abord être décalée à droite de n bits, où n est la différence de valeur des deux exposants. Si n est plus grand que le nombre de bits de la représentation de la mantisse, le second nombre sera traité comme un 0 en ce qui concerne l'opération d'addition. La situation est encore pire pour la soustraction (et pour l'addition d'un nombre positif et d'un nombre négatif). Si les deux nombres ont une valeur absolue de même ordre de grandeur, le résultat de l'opération est environ zéro et peut avoir très peu de chiffres significatifs.

La représentation en complément à deux présentée dans la section 6.1 n'est utile que pour l'addition et la soustraction. Elle complique la situation pour la multiplication et la division. Pour ces deux dernières opérations, il est avantageux d'utiliser une représentation sous la forme de signe plus valeur absolue. Puisque ces deux opérations sont les plus fréquentes pour les flottants et qu'elles nécessitent la multiplication ou la division de deux mantisses, il est avantageux de représenter la mantisse avec signe et valeur absolue. Pour les exposants, c'est l'addition ou la soustraction qui sont les opérations les plus utilisées. Pour cette raison, on utilise une représentation ayant les même propriétés que celle du complément à deux.

Les ordinateurs manipulent souvent des données codées par *mots* de 32 ou de 64 bits. Pour cette raison, il est commode de découper un tel mot de sorte que le nombre de bits permettant de représenter le signe, la mantisse et l'exposant soit exactement celui d'un mot. Bien qu'il y ait plusieurs façons de découper le mot, en pratique c'est la norme IEEE-754 qui est presque toujours utilisée.

6.6 LA NORME IEEE-754

Cette norme définit trois formats différents, dont deux formats externes, simple précision (32 bits) et double précision (64 bits), et un

format interne étendu (80 bits). Le format interne est utilisé à l'intérieur de l'unité flottante du processeur afin de maintenir un maximum de précision.

Pour la simple précision, la mantisse contient 23 bits, et l'exposant 8 bits (1 bit étant toujours réservé au signe). Pour la double précision, la mantisse contient 52 bits et l'exposant 11 bits.

La norme IEEE-754 définit deux représentations différentes pour chaque taille de mot : la *représentation normalisée* et la *représentation non normalisée*.

6.6.1 Représentation normalisée

La représentation de l'exposant s'appelle « excédant 127 » pour la simple précision et « excédant 1 023 » pour la double précision. C'est une représentation apparentée au complément à deux. Pour représenter un exposant dont la valeur est e, on y ajoute d'abord 127 ou 1 023 selon qu'on est en simple ou en double précision. Le nombre qui en résulte est stocké en représentation binaire. Dans la représentation normalisée, un champ de l'exposant ne contenant que des chiffres 1 ou que des chiffres 0 est exclu. Par conséquent, en représentation normalisée avec simple précision, le plus petit nombre représentable est $1 \cdot 2^{1-127} = 2^{-126}$. Le plus grand nombre est $1{,}11111111111111111111111 \cdot 2^{254-127} \approx 2^{128}$. Pour la double précision, les valeurs sont 2^{-1022} et 2^{1024}.

6.6.2 Représentation non normalisée

La norme définit quatre types de représentations non normalisées, la représentation dénormalisée, la représentation de zéro, la représentation de l'infini, et NaN (*Not a Number*) indiquant une situation d'erreur.

Le champ de l'exposant de la représentation dénormalisée ne contient que des chiffres 0 dont la signification est -127 en simple précision et $-1\,023$ en double précision. La valeur de la mantisse respecte la condition $0 < m < 1$ et tous les chiffres de la mantisse sont représentés. Le plus grand nombre dénormalisé en simple précision est donc $0{,}11111111111111111111111 \cdot 2^{-127}$, soit presque 2^{-127}. Le plus petit nombre dénormalisé en simple précision est $0{,}00000000000000000000001 \cdot 2^{-127}$, soit 2^{-150}.

Pour la représentation de zéro, le champ de la mantisse et celui de l'exposant ne contiennent que des 0. C'est l'extension logique de la représentation dénormalisée. Attention, il y a deux représentations du nombre 0, une positive et une négative selon la valeur du bit de signe.

Pour la représentation de l'infini, le champ de l'exposant ne contient que des 1, et celui de mantisse ne contient que des 0. Cette valeur est produite par certaines opérations arithmétiques lorsque le résultat de l'opération dépasse ce qui est représentable de façon normalisée. Cette valeur est aussi acceptable en tant qu'opérande pour les opérations arithmétiques.

Certaines opérations, la division de l'infini par l'infini, par exemple, donnent un résultat indéfini, dont la représentation a un champ d'exposant ne contenant que des chiffres 1 et un champ de mantisse contenant n'importe quelle configuration de chiffres autre que 000. . .

6.7 RÉCAPITULATIF

Pour des raisons d'adéquation avec les circuits binaires, un ordinateur moderne utilise le plus souvent l'arithmétique binaire. De plus, l'arithmétique binaire est beaucoup plus simple que l'arithmétique décimale.

Afin d'éviter un circuit pour l'addition et un autre circuit similaire pour la soustraction, on utilise une représentation de nombre négatifs nommée « complément à deux ».

La plupart des processeurs utilisés avec les ordinateurs personnels et les gros calculateurs contiennent des circuits spécialisés pour l'arithmétique sur les flottants. Le plus souvent, cette arithmétique est conforme à la norme IEEE-754. Ces circuits permettent aujourd'hui d'effectuer une opération flottante en quelques cycles d'horloge seulement.

EXERCICES

Exercice 6.1. Le registre d'état d'un microprocesseur 8 bits comporte un bit C(arry) de retenue, et un bit (o)V(erflow) de débordement.
Donner une expression logique de C et V en fonction des bits de poids fort, des opérandes et du résultat de l'addition.

Exercice 6.2. On note $(x)_{c2}$ la représentation en *complément à deux* sur n bits de l'entier x. On rappelle que si $(x)_{c2} = x_{n-1} \cdots x_1 x_0$, alors

$$x = -x_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i \ .$$

Donner $(51)_{c2}$, $(128)_{c2}$, $(-1)_{c2}$, $(-51)_{c2}$, et $(-128)_{c2}$ pour $n = 8$ et $n = 16$.

Exercice 6.3. On rappelle qu'en flottant simple précision, un mot de 32 bits représente le nombre

$$(-1)^s 2^{e-127}(1+m)$$

où le signe s est donné par le bit fort, l'exposant e est codé sur les 8 bits suivants, et la mantisse m est codée sur les 23 bits restants par

$$m = \sum_{i=22}^{i=0} \frac{1}{2^{23-i}} b_i$$

Indiquer la représentation binaire en flottant simple précision des nombres suivants :
30,4, $-0{,}0625$, $\frac{1}{3}$.

Exercice 6.4. Déterminer le nombre représenté en flottant simple précision par 1000 1111 1110 1111 1100 0000 0000 0000.

Exercice 6.5. L'addition flottante est-elle une opération commutative ? Justifier la réponse.

Chapitre 7

Circuits pour l'arithmétique binaire

L'arithmétique binaire est un problème combinatoire. Il semble trivial d'utiliser les méthodes déjà décrites pour la conception d'un circuit combinatoire (voir chapitre 3) afin d'obtenir des circuits pour l'arithmétique binaire.

Malheureusement, les circuits combinatoires obtenus pour ce type de problèmes avec les méthodes habituelles contiennent beaucoup trop de portes logiques pour être utilisés en pratique. Il faut trouver une autre solution.

7.1 ADDITION ENTIÈRE BINAIRE

Pour l'addition entière binaire, une solution consiste à sacrifier la contrainte concernant la *profondeur du circuit* afin d'obtenir un circuit avec un nombre moins important de portes logiques. Le circuit qui en résulte est d'un type que nous appelons *circuit combinatoire itératif*, car il contient plusieurs exemplaires d'un même type d'élément. Pour l'addition binaire, l'élément de base est l'*additionneur 1-bit*.

Un tel additionneur est un circuit combinatoire à trois entrées et deux sorties. Son but est d'additionner deux chiffres binaires plus

la retenue de la position précédente et de donner un résultat sur deux bits : la sortie normale et la retenue pour la position suivante. La table de vérité de l'additionneur 1-bit est illustrée par la figure 7.1.

x	y	c_{in}	c_{out}	s
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

Figure 7.1 : Table de vérité pour l'additionneur 1-bit

Ici, nous avons utilisé les noms de variables x et y pour les entrées, c_{in} pour la retenue en entrée, s pour la somme et c_{out} pour la retenue en sortie.

Un additionneur 1-bit peut être trivialement construit en utilisant nos méthodes habituelles. Le circuit simplifié qui en résulte est donné par la figure 7.3.

L'étape suivante combine une série de tels additionneurs afin d'obtenir un circuit capable d'additionner, par exemple, des nombres positifs sur 8 bits. Pour cela, il suffit de brancher la retenue en sortie (c_{out}) d'un additionneur avec la retenue en entrée (c_{in}) de l'additionneur à sa gauche. Le signal c_{in} de l'additionneur le plus à droite est branché à 0. Le circuit d'un additionneur 8-bits est présenté figure 7.2. Ici, nous avons utilisé l'indice i pour la i-ième position binaire.

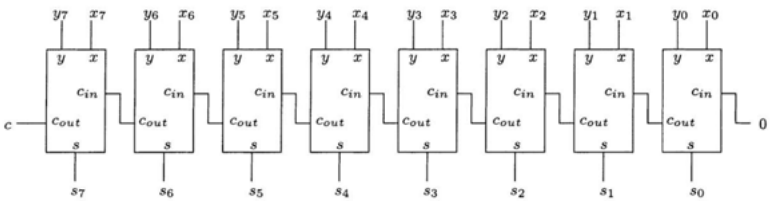


Figure 7.2 : Additionneur 8-bits

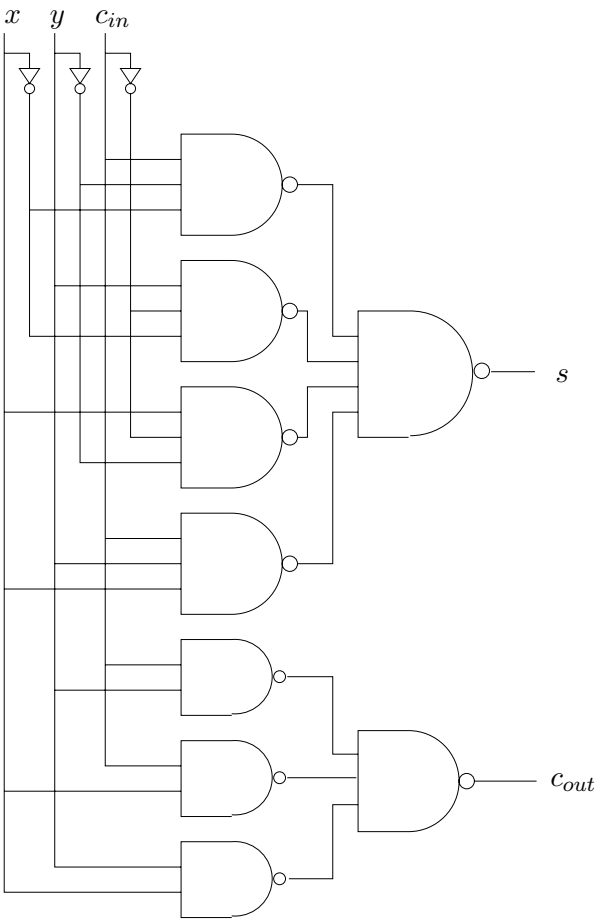


Figure 7.3 : Additionneur 1-bit

On constate que la profondeur de ce circuit est bien supérieure à 2. En fait, la sortie et la retenue de la position 7 sont déterminées partiellement par les entrées de la position 0. Le signal doit traverser l'ensemble des additionneurs accumulant les retards.

Il existe des solutions intermédiaires entre les deux solutions extrêmes que nous avons considérées pour l'instant, à savoir soit un circuit combinatoire à deux niveaux pour l'additionneur complet sur

32 bits, par exemple, soit un circuit combinatoire itératif dont les éléments sont des additionneurs 1-bit construits comme des circuits combinatoires ordinaires.

Une solution intermédiaire envisageable est d'utiliser comme éléments du circuit combinatoire itératif non pas des additionneurs à un seul bit, mais des additionneurs légèrement plus larges, par exemple à 4 bits. Cette solution est donnée par la figure 7.4. Un additionneur capable d'additionner deux nombres sur quatre bits nécessite au plus quelques centaines de portes logiques, même avec une profondeur limitée à deux. La profondeur de l'additionneur complet (sur 32 bits par exemple) est donc instantanément divisée par 4.

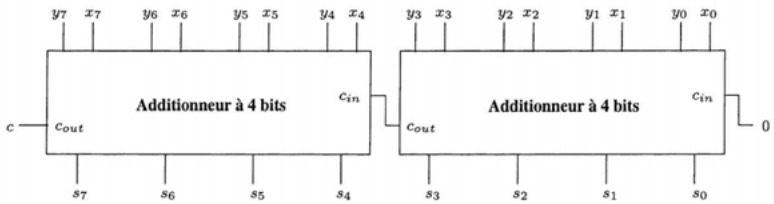


Figure 7.4 : Additionneur 2 fois 4-bits

Une autre solution est de construire un circuit supplémentaire que nous appelons circuit *accélérateur de retenue*. L'idée est de commencer par le circuit itératif et de le couper en deux parties. La partie gauche, au lieu d'obtenir son entrée c_{in} de la sortie c_{out} pour la partie droite, l'obtient par la sortie de ce nouveau circuit. Ce circuit a les même entrées que la partie droite, mais une seule sortie, c_{out}. Puisque le nombre de sorties est plus faible que celui d'un additionneur, on peut espérer que la complexité d'un tel circuit devienne raisonnable, même sous la forme de circuit combinatoire à deux niveaux. Cette solution est illustrée par la figure 7.5.

Une troisième solution consiste également à commencer par le circuit itératif coupé en deux parties, mais au lieu de tenter de calculer la retenue de la partie droite plus rapidement qu'avant, on calcule l'addition de la partie gauche *simultanément* avec la retenue égale

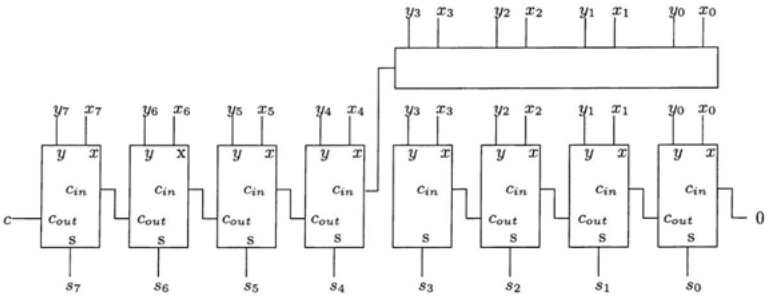


Figure 7.5 : Additionneur avec accélérateur de retenue

à 0 et la retenue égale à 1. La retenue de la partie droite est utilisée pour sélectionner lequel des deux calculs servira pour le résultat final. Cette solution (appelée *addition spéculative*) est illusrée par la figure 7.6. Les trois additionneurs à 4 bits calculent leur résultat respectif simultanément, car aucun des trois ne dépend du résultat de l'autre. La profondeur totale du circuit est donc la somme de celle d'un additionneur plus 2 (la profondeur du multiplexeur). Le prix à payer est bien sûr la multiplication par deux du nombre de circuits de la partie gauche.

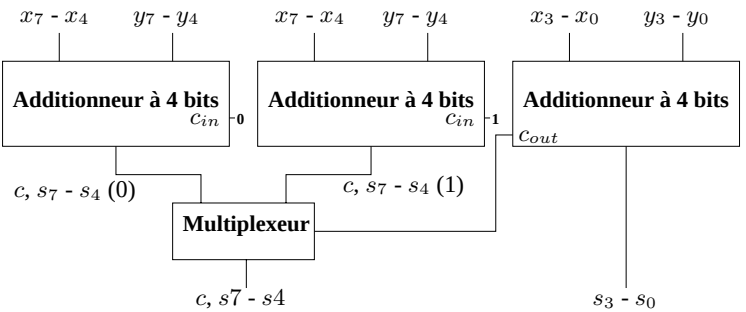


Figure 7.6 : Addition spéculative

7.2 SOUSTRACTION BINAIRE

Notre additionneur est déjà capable de traiter des nombres négatifs représentés en *complément à deux* comme indiqué dans le chapitre 6. Il nous reste à déterminer comment traiter la soustraction.

On remarque que le calcul de l'expression $x - y$ équivaut au calcul de l'expression $x + (-y)$. Au chapitre 6, nous avons vu comment calculer l'opposé d'un nombre en inversant chaque position, puis en additionnant 1 au résultat. Il reste donc à calculer l'expression $x + \overline{y} + 1$. Pour calculer $\overline{y}$, il suffit d'inverser chaque position de y avant qu'elles n'arrivent à l'additionneur, mais comment additionner 1 ensuite ? Cette opération semble nécessiter encore un additionneur. Heureusement, nous avons une entrée c_{in} de la position 0 qui n'est pas actuellement utilisée. Or justement mettre la valeur 1 sur cette entrée additionne 1 au résultat entier. Le circuit complet pour l'addition et la soustraction est présenté figure 7.7.

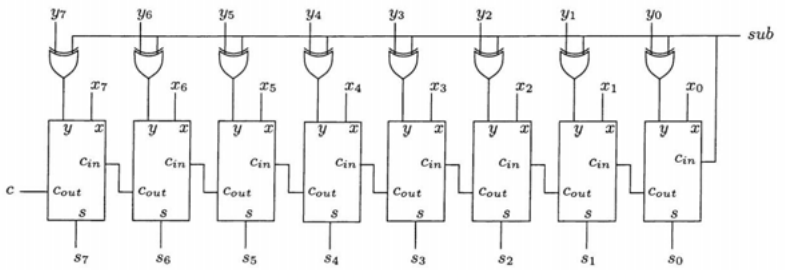


Figure 7.7 : Circuit pour l'addition et la soustraction

7.3 MULTIPLICATION ET DIVISION BINAIRES

La multiplication et la division entière sont encore plus difficiles à implémenter que l'addition. Il n'existe pas de bon circuit combinatoire itératif. La solution consiste à utiliser un circuit séquentiel qui effectue une addition par front d'horloge. Ce sera le sujet d'un section ultérieure, car nous n'avons pas encore introduit la notion de circuit séquentiel.

7.4 RÉCAPITULATIF

Il n'est pas pratique d'utiliser la méthode générale de construction de circuits combinatoires pour la construction de circuits d'arithmétique binaire. Pour ces circuits, il est nécessaire d'accepter une profondeur des circuits supérieure à celle donnée par la méthode générale.

Pour la multiplication binaire, on préférera un circuit séquentiel à un circuit combinatoire de taille trop importante.

EXERCICES

Exercice 7.1. Combien faudrait-il de portes *non-et* pour réaliser un additionneur n-bits en utilisant un circuit combinatoire de profondeur 2 construit suivant la méthode générale présentée au chapitre 3 ?

Exercice 7.2. Combien utilise-t'on de portes *non-et* en construisant un additionneur n-bits à l'aide de n additionneurs 1-bit comme expliqué dans la section 7.1.

Chapitre **8**

Bascules et bistables

De la même manière que les portes logiques (voir chapitre 2) sont les briques de base des circuits combinatoires (voir chapitre 3), les *bascules* et les *bistables* sont les briques de base des *circuits séquentiels*.

Alors que les portes logiques sont construites directement avec des transistors, les bascules sont construites à partir de portes et les bistables à partir de bascules.

Dans les bascules et les bistables, les valeurs des sorties dépendent non seulement des valeurs des entrées, mais aussi des anciennes valeurs des sorties. La différence principale entre bascule et bistable est que la première n'a pas de signal d'*horloge*, alors que le second en a toujours un.

8.1 BASCULES

Comment créer, à partir de portes logiques, un circuit qui ne soit pas combinatoire ? La réponse est : en utilisant la notion de *retour* (en anglais : *feed-back*), à savoir l'introduction de *boucles* dans le diagramme du circuit. Avec une telle boucle, la sortie dépend indirectement d'elle-même. Si le retour est *positif* (nombre pair d'inverseurs), le circuit aura tendance à avoir des états stables, alors qu'il oscillera généralement avec un retour *négatif*.

Les bascules doivent avoir un retour positif. Un exemple de bascule simple est illustré par la figure 8.1.

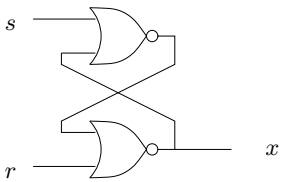


Figure 8.1 : bascule SR

Il s'agit d'une *bascule* SR (pour *set-reset*).

Les méthodes utilisées pour les circuits combinatoires ne sont pas adaptées à la description des bascules. Au chapitre 9.1, nous présenterons une méthode de description de bistables et de circuits séquentiels avec horloge, similaire à celle utilisée avec les circuits combinatoires. Pour l'instant, la méthode d'analyse de la bascule sera plutôt intuitive.

On convient que les deux entrées de la bascule SR ne peuvent valoir 1 simultanément (nous allons nous servir de la bascule uniquement dans des situations où cette condition est garantie). Lorsque les deux entrées valent 0, la bascule a deux états stables possibles. Soit x vaut 0, ce qui donne les signaux de la figure 8.2, soit x vaut 1, ce qui donne les signaux de la figure 8.3.

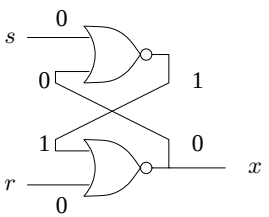


Figure 8.2 : Analyse de la bascule SR (1)

La valeur réelle obtenue pour x dépend de l'*historique* des valeurs d'entrées, comme nous allons maintenant le montrer.

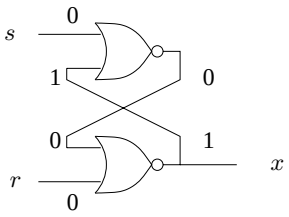


Figure 8.3 : Analyse de la bascule SR (2)

On suppose que s vaut 1, et par conséquent r vaut 0 puisque nous avons fait l'hypothèse qu'au plus l'une des deux valeurs en entrées vaut 1. On obtient les signaux de la figure 8.4.

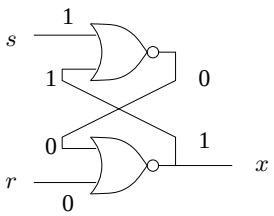


Figure 8.4 : Analyse de la bascule SR (3)

La valeur 1 sur l'entrée s force la sortie de la porte-*non-ou* du haut à 0, et les deux 0 sur les entrées de la porte-*non-ou* du bas forcent la sortie x à 1.

Maintenant, on suppose que la valeur de s passe de 1 à 0 alors que l'entrée r reste à 0. La deuxième entrée de la porte-*non-ou* du haut vaut 1. Par conséquent la transition de s ne change pas la valeur de la sortie de cette porte. La sortie x reste à 1. Dans ce cas, lorsque les deux entrées s et r valent 0, il n'y a qu'un seul état stable possible, celui avec $x = 1$.

De la même manière, on suppose que r vaut 1 (et donc $s = 0$ par hypothèse). On obtient les signaux de la figure 8.5.

La valeur 1 sur l'entrée r force x à 0 et les deux 0 sur les entrées de la porte-*non-ou* du haut forcent la sortie de celle-ci à 1.

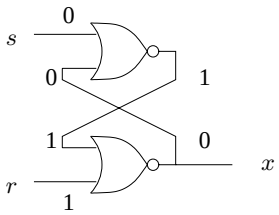


Figure 8.5 : Analyse de la bascule SR **(4)**

Maintenant, on suppose que la valeur de r passe de 1 à 0 alors que l'entrée s reste à 0. La deuxième entrée de la porte-*non-ou* du bas vaut 1. Par conséquent la transition de r ne change pas la valeur de la sortie de cette porte. La sortie x reste à 0. Dans ce cas, lorsque les deux entrées s et r valent 0, il n'y a qu'un seul état stable possible, celui avec $x = 0$.

Par la description ci-dessus, on constate que la bascule SR est capable de mémoriser la valeur précédente de ses entrées, dans le sens où elle se souvient de laquelle des deux entrées valait 1 en dernier. La mise à 1 de l'entrée s positionnera la sortie à 1 (Set) tandis que la mise à 1 de l'entrée r positionnera la sortie à 0 (Reset). Les entrées à 0 permettent de conserver l'ancienne valeur de la sortie.

Lorsque nous voulons dessiner une bascule SR, nous utilisons le symbole de la figure 8.6.

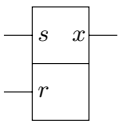


Figure 8.6 : Symbole de la bascule SR

8.2 HORLOGE

Les bascules sont *asynchrones*, à savoir que leur sortie change rapidement après un changement des entrées. De nos jours, cependant, les ordinateurs sont *synchrones*, c'est-à-dire que les sorties de tous les circuits changent simultanément suivant le rythme d'un signal périodique global appelé *horloge*.

Une *horloge* est un signal périodique. Sa fréquence est l'inverse de sa *période* (ou *temps de cycle*). La période de l'horloge se divise en deux parties : une période haute et une période basse. Un signal d'horloge typique est illustré par la figure 8.7.

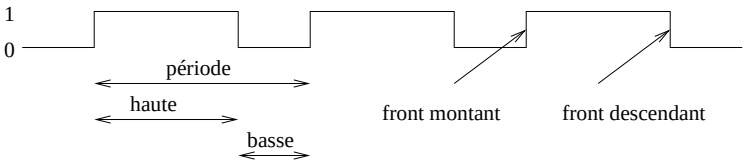


Figure 8.7 : Signal d'horloge

Il existe deux types de synchronisation. Dans la *synchronisation par niveau*, tous les changements d'état ont lieu pendant que l'horloge est à un niveau fixé (haut ou bas).

Dans la *synchronisation par front d'impulsion*, tous les changements d'état se font sur un front d'impulsion fixé (montant ou descendant) d'horloge. L'avantage de cette dernière est que l'on peut lire et écrire un élément d'état lors d'un seul et même cycle d'horloge. Cependant, elle nécessite la contrainte supplémentaire que tous les signaux à écrire soient stables lorsque le front d'impulsion actif survient. L'horloge doit avoir une période suffisamment longue pour que les signaux aient le temps de se stabiliser à l'intérieur des blocs de circuits combinatoires comme indiqué par la figure 8.8.

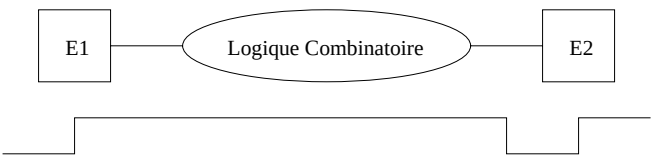


Figure 8.8 : Limite inférieure de la période d'horloge

Dans tout ce qui suit nous nous plaçons dans le cadre d'une synchronisation par front d'impulsion descendant.

8.3 BISTABLES

Un bistable est la version synchrone d'une bascule. Bien qu'il existe plusieurs réalisations possibles pour un bistable, nous n'en considérons ici qu'un seul type, à savoir *maître-esclave*.

De plus, il y a diverses variations mineures concernant le nombre d'entrées et la façon dont les entrées pilotent le bistable. Ici, nous allons considérer un seul cas : le bistable D (en anglais : D *flip-flop*). Un bistable D de type maître-esclave peut être construit à partir de deux bascules SR et quelques portes logiques. Le circuit correspondant est donné par la figure 8.9.

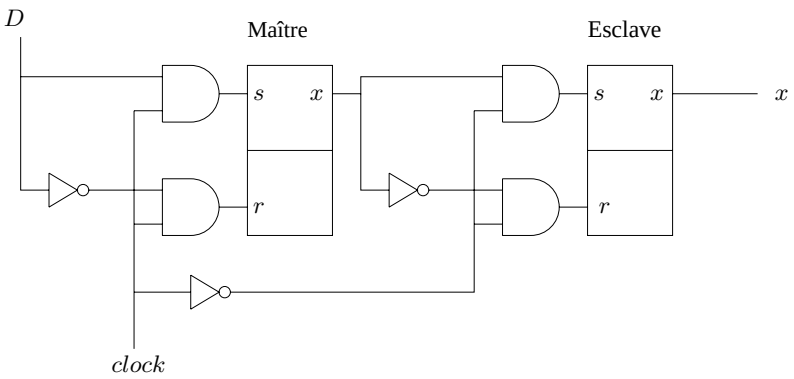


Figure 8.9 : Bistable D

La bascule de gauche est le *maître* et celle de droite l'*esclave*. Regardons d'abord ce qui se passe lorsque le signal d'horloge vaut 1. Dans ce cas, les deux portes-*et* devant les entrées du maître sont *ouvertes*, à savoir qu'elles permettent à la valeur de l'entrée D de passer directement sur l'entrée s de la bascule de gauche et de son inverse de passer sur l'entrée r. Par conséquent, la valeur de l'entrée D va passer directement sur la sortie x du maître. Par contre, les deux portes-*et* qui se trouvent devant l'esclave sont *fermées*, à savoir que leur sortie vaut 0. Par conséquent, l'esclave maintient son ancienne valeur.

Au contraire, lorsque l'horloge vaut 0, c'est la situation inverse, à savoir que les portes-*et* du maître sont fermées et celles de l'esclave sont ouvertes. Dans ce cas, le bistable est complètement insensible aux variations du signal D.

Maintenant, considérons ce qui se passe lorsque l'horloge passe de 1 à 0. Pour que cela fonctionne, il va falloir supposer que l'entrée reste stable brièvement pendant ce passage. La première chose qui se produit est la fermeture des portes-*et* du maître, à savoir qu'elles deviennent insensibles aux changements futurs du signal D. La valeur de la sortie x du maître sera donc égale à la valeur du signal D juste avant que l'horloge passe de 1 à 0. Un moment bref plus tard, le signal de l'horloge aura traversé l'inverseur et arrivera aux portes-*et* de l'esclave. Ces portes *s'ouvrent* et laissent passer la valeur x du maître sur la sortie x de l'esclave. La valeur de la sortie x de l'esclave (et donc celle du bistable entier) est alors celle du signal D juste avant le passage de l'horloge de 1 à 0. Il est donc juste de dire que le front d'horloge provoque la copie de l'entrée vers la sortie du bistable. Par contre, il n'y a jamais de passage direct de l'entrée vers la sortie du bistable. La sortie change de manière synchrone avec le passage de l'horloge de 1 à 0 (le front descendant d'horloge).

Finalement, regardons ce qui se passe lorsque l'horloge passe de 0 à 1. D'abord, les portes-*et* du maître s'ouvrent, laissant passer la valeur du signal D. Avant que le signal D n'arrive au maître, le signal d'horloge inversé arrive aux portes-*et* de l'esclave qui se ferment avant que la sortie (potentiellement modifiée) du maître n'arrive à l'esclave. L'esclave maintient son ancienne valeur. Vu de l'extérieur, il ne se passe rien. Hors, désormais, le maître est sensible aux changements du signal D.

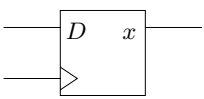


Figure 8.10 : Symbole du bistable D

Le symbole du bistable D est illustré par la figure 8.10. Le petit triangle sur l'entrée de l'horloge indique que cette entrée est sensible uniquement aux *transitions* et non aux *niveaux* du signal comme nous l'avons expliqué ci-dessus. Souvent, nous allons omettre le signal de l'horloge, car il est supposé toujours présent. Toutes les entrées d'horloge de tous les circuits d'un diagramme sont toujours connectées ensemble. Les dessiner n'améliore donc pas la compréhension du circuit.

8.4 RÉCAPITULATIF

Nous avons montré comment construire un bistable D. Celui-ci copie son entrée vers sa sortie lorsque le signal d'*horloge* passe de 1 à 0. La valeur copiée est celle de l'entrée *immédiatement avant la transition de l'horloge*. Pendant le reste de la période de l'horloge, le bistable est insensible à tout changement de son entrée.

Cette caractéristique du bistable D est utile pour construire des *circuits séquentiels synchrones*.

EXERCICES

Exercice 8.1. Réaliser un bistable D en n'utilisant que des portes-*non-et* et des inverseurs. Indication : remplacer les portes-*non-ou* des deux bascules SR par des portes-*et* et des inverseurs, puis simplifier autant que possible.

Exercice 8.2.

(1) Analyser le comportement du circuit de la figure 8.11, lorsque l'entrée d est à 0 puis lorsqu'elle est à 1.

(2) Que se passe-t-il si on remplace les portes *non-ou* par des portes *non-et* ?

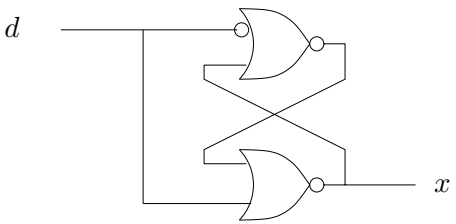


Figure 8.11 : Circuit pour l'exercice 8.2

Exercice 8.3.

(1) Analyser le comportement du circuit de la figure 8.12.

(2) Le circuit est en fait un enchaînement de trois éléments identiques. De manière générale, que ferait un circuit similaire enchaînant n (au lieu de trois) de ces éléments ?

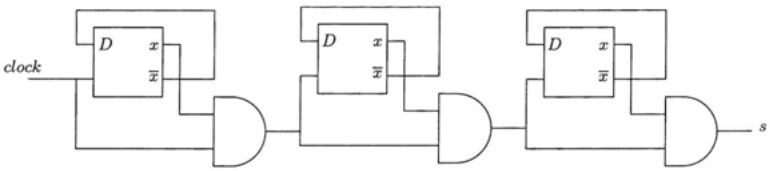


Figure 8.12 : Circuit pour l'exercice 8.3

Chapitre 9

Circuits séquentiels

De la même façon que les circuits combinatoires (voir chapitre 3) sont construits à partir de portes logiques (voir chapitre 2), les circuits séquentiels sont contruits à partir de bistables (voir chapitre 8).

En général, un *circuit séquentiel synchrone*, ou simplement *circuit séquentiel* est un circuit à m entrées et n sorties plus une entrée distinguée : l'*horloge*. La description du circuit se fait à l'aide d'une *table d'états*.

Nous avons simplifié volontairement notre définition de circuit séquentiel. Avec notre définition, le nombre d'états possibles du circuit est entièrement déterminé par le nombre de ses sorties. Une définition plus générale séparerait la notion de sortie de celle d'état. Cette généralité supplémentaire n'étant pas utile pour notre objectif, nous en resterons à la définition simplifiée.

Les critères d'optimalité qui s'appliquent à la conception d'un circuit combinatoire s'appliquent aussi à la conception d'un circuit séquentiel, à savoir, nombre de transistors, vitesse, consommation d'électricité, etc. Comme pour les circuits combinatoires, nous ne considérerons pas des méthodes de conception optimisant ces critères, mais seulement une méthode très générale qui, dans le pire des cas, gaspillera un grand nombre de transistors. Nous rappelons que l'objectif de ce livre n'est pas de former des concepteurs de circuits

séquentiels, mais simplement de donner une idée de la manière dont ce type de circuit fonctionne.

9.1 TABLE D'ÉTATS

Comme une *table de vérité* pour un circuit combinatoire, une *table d'états* est utilisée pour décrire un circuit séquentiel.

Une table d'états ressemble à une table de vérité, sauf qu'à sa gauche figurent non seulement les entrées du circuit, mais aussi les *sorties*. La partie droite de la table ne contient pas les *valeurs des sorties*, mais les *valeurs des sorties après le prochain front d'horloge*. Afin de distinguer les valeurs des sorties avant et après le prochain front d'horloge, nous les appelons x, y, etc, avant le front d'horloge et x', y', etc, après le front d'horloge.

Un exemple de table d'états est illustré par la figure 9.1.

x	a	b	c	a'	b'	c'
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	1	0	0
0	1	0	1	1	0	1
0	1	1	0	1	1	0
0	1	1	1	1	1	1
1	0	0	0	0	0	1
1	0	0	1	0	1	0
1	0	1	0	0	1	1
1	0	1	1	1	0	0
1	1	0	0	1	0	1
1	1	0	1	1	1	0
1	1	1	0	1	1	1
1	1	1	1	0	0	0

Figure 9.1 : Table d'états

Comme on peut le constater dans cet exemple, lorsque l'entrée x vaut 0, les valeurs des sorties ne changent pas après le prochain front

d'horloge. En revanche, lorsque l'entrée x vaut 1, la valeur des sorties (interprétée comme un nombre binaire) est l'ancienne valeur (aussi interprétée comme un nombre binaire) plus 1. Cette table d'états décrit donc un *compteur* avec une entrée supplémentaire permettant d'indiquer si le compteur compte ou non.

9.2 CONCEPTION D'UN CIRCUIT SÉQUENTIEL

Pour la conception d'un circuit séquentiel à m entrées et n sorties, la méthode générale de conception utilise n bistables D (un pour chaque sortie), et un circuit combinatoire à $m+n$ entrées et n sorties. La structure générale du circuit est illustrée par la figure 9.2.

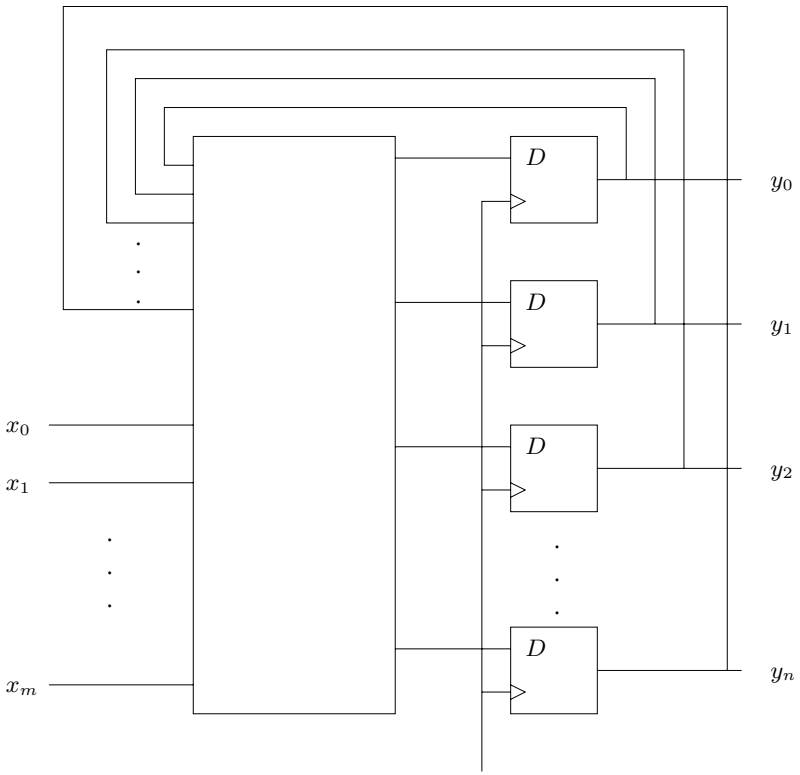


Figure 9.2 : Circuit séquentiel général

Puisque nous utilisons des bistables D, la sortie du circuit après le prochain front d'horloge sera exactement la même que celle du circuit combinatoire. Il sera donc possible de se servir de la méthode générale pour la conception de circuits combinatoires, mais cette fois-ci, elle sera appliquée à la *table d'états* du circuit séquentiel.

Pour illustrer cette idée, construisons un compteur de 2 bits de large avec une entrée (u/d) indiquant si le compteur doit compter de manière ascendante ($u/d = 1$) ou descendante ($u/d = 0$). Le compteur doit s'arrêter à 00 lorsqu'il descend et à 11 lorsqu'il monte. La table d'états d'un tel compteur est illustrée par la figure 9.3.

u/d	y_1	y_0	y_1'	y_0'
0	0	0	0	0
0	0	1	0	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	1
1	1	1	1	1

Figure 9.3 : Table d'états du compteur

Avec notre méthode générale, il suffit de construire un circuit combinatoire à partir de la table de vérité correspondant à cette table d'états. Le circuit qui en résulte est représenté à la figure 9.4.

9.3 RÉCAPITULATIF

La conception d'un circuit séquentiel est possible grâce à une méthode générale similaire à celle utilisée pour les circuits combinatoires. Cette méthode utilise une table d'états similaire à une table de vérité. En utilisant des bistables D, le problème de la conception d'un tel circuit peut être réduit à celui de la conception d'un circuit combinatoire.

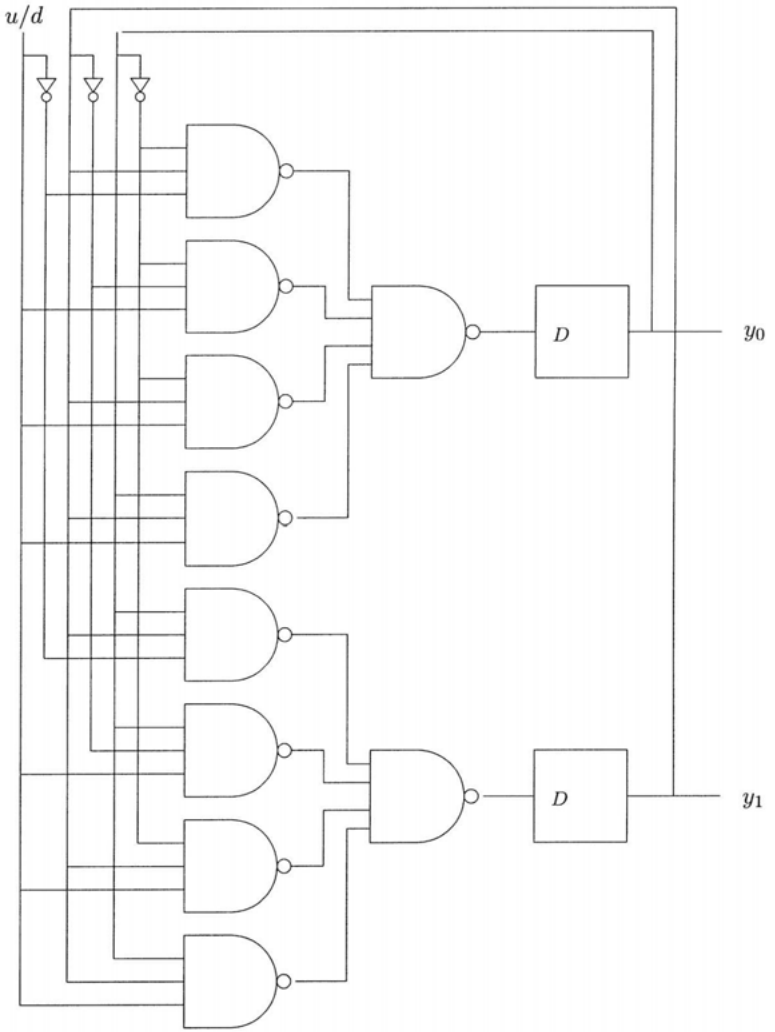


Figure 9.4 : Circuit séquentiel du compteur

EXERCICES

Exercice 9.1. Soit la table d'états de la figure 9.5. Dessiner le circuit décrit par celle-ci.

x	a	y	a'	y'
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
0	1	1	1	1
1	0	0	0	1
1	0	1	0	1
1	1	0	1	0
1	1	1	1	0

Figure 9.5 : Table d'états

Exercice 9.2. Soit le circuit séquentiel de la figure 9.6. Ce circuit est réalisé avec trois bistables D ; il a une entrée x et trois sorties y_2, y_1 et y_0.

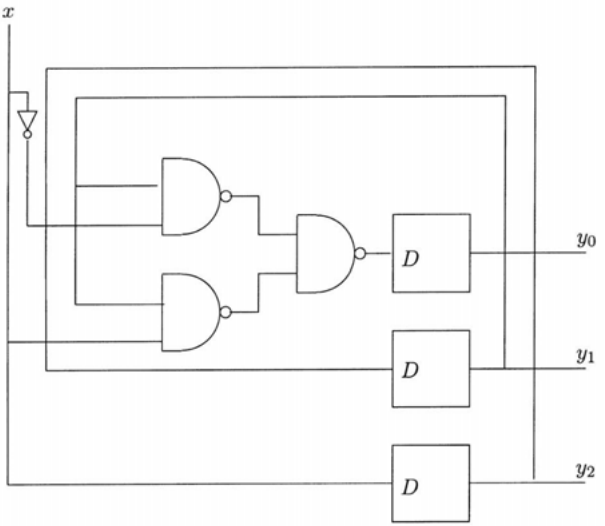


Figure 9.6 : Circuit séquentiel pour l'exercice 9.2

(1) Soient y_2, y_1 et y_0 les valeurs des sorties au cours d'un cycle. Exprimer y_2', y_1' et y_0', les valeurs des sorties au cycle suivant en fonction de x, y_2, y_1 et y_0.
(2) Donner la table d'états correspondant à ce circuit.
(3) Expliquer en une phrase ce que fait le circuit.
(4) Comment pourrait-on simplifier ce circuit ?

Chapitre **10**

Circuits séquentiels classiques

Les circuits séquentiels les plus courants sont les registres et les compteurs. Les circuits séquentiels sont aussi utilisés pour réaliser des fonctions logiques qui nécessiteraient un circuit combinatoire trop complexe, comme la multiplication.

10.1 REGISTRES

Un *registre* est un circuit séquentiel (voir chapitre 9) à $n + 1$ entrées (l'horloge ne compte pas) et n sorties. À chaque sortie correspond une entrée. Les n premières entrées sont appelées $x_0, x_1, \ldots, x_{n-1}$ et la dernière est appelée ld (pour *load*). Les sorties sont appelées $y_0, y_1, \ldots, y_{n-1}$.

Lorsque ld vaut 0, les sorties ne sont pas affectées par un front d'horloge. Elles préservent leurs anciennes valeurs. Par contre, lorsque ld vaut 1, les entrées sont copiées vers les sorties après le prochain front d'horloge.

On peut expliquer ce comportement plus formellement avec une table d'états (voir section 9.1). Par exemple, prenons un registre avec

$n = 4$. La partie gauche de la table contient donc 9 colonnes étiquetées ld, x_0, x_1, x_2, x_3, y_0, y_1, y_2 et y_3. La table d'états complète a donc 512 lignes ! Mais nous utilisons l'abréviation de celle-ci présentée dans la figure 10.1.

ld	x_3	x_2	x_1	x_0	y_3	y_2	y_1	y_0	y'_3	y'_2	y'_1	y'_0
0	–	–	–	–	c_3	c_2	c_1	c_0	c_3	c_2	c_1	c_0
1	c_3	c_2	c_1	c_0	–	–	–	–	c_3	c_2	c_1	c_0

Figure 10.1 : Table d'états du registre

Comme on peut le constater, lorsque ld vaut 0 (partie haute de la table), la partie droite de la table est une copie des anciennes valeurs des sorties, indépendamment des entrées. Par contre, lorsque ld vaut 1, la partie droite de la table est plutôt une copie des valeurs des entrées, indépendamment des anciennes valeurs des sorties.

Les registres jouent un rôle très important dans les ordinateurs. Certains sont visibles par le programmeur et sont utilisés pour stocker des valeurs en vue d'une utilisation ultérieure. D'autres sont invisibles de l'utilisateur et sont utilisés exclusivement pour conserver des valeurs temporaires internes au processeur.

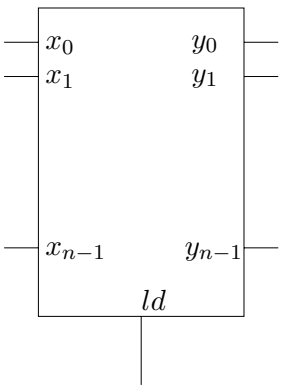


Figure 10.2 : Registre

Nous utilisons le symbole de la figure 10.2 pour représenter un registre. Le signal de l'horloge est omis, car toujours présent.

10.2 COMPTEURS

Un *compteur* est un circuit séquentiel (voir chapitre 9) avec 0 entrée et n sorties. Par conséquent, la valeur après le prochain front d'horloge dépend uniquement de l'ancienne valeur des sorties.

Pour un compteur, les valeurs des sorties sont interprétées comme une suite de chiffres binaires (voir chapitre 6). Les sorties sont notées $o_{n-1}, o_{n-2}, \ldots, o_0$. Les valeurs des sorties après le front d'horloge interprétées de cette manière correspondent à l'ancienne valeur *plus 1*.

Avec une table d'états (voir section 9.1) nous pouvons expliquer plus formellement ce comportement. Prenons par exemple un compteur avec $n = 4$. La partie gauche de la table contient 4 colonnes étiquetées o_3, o_2, o_1 et o_0. La table contient donc 16 lignes. Elle est illustrée par la figure 10.3.

o_3	o_2	o_1	o_0	o_3'	o_2'	o_1'	o_0'
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	1	0	1	0
1	0	1	0	1	0	1	1
1	0	1	1	1	1	0	0
1	1	0	0	1	1	0	1
1	1	0	1	1	1	1	0
1	1	1	0	1	1	1	1
1	1	1	1	0	0	0	0

Figure 10.3 : Table d'états du compteur

Comme on peut le constater, la partie droite est toujours égale à la partie gauche *plus 1*, sauf dans la dernière ligne où on a la valeur 0 pour toutes les sorties. C'est donc un compteur *modulo 16*.

Les compteurs (avec diverses variations) jouent un rôle important dans les ordinateurs. Certains sont visibles par le programmeur, comme le compteur ordinal (en anglais : *program counter, PC*). D'autres sont invisibles par le programmeur et sont utilisés principalement pour maintenir des valeurs internes de l'unité centrale.

Voici quelques variations importantes :

– la possibilité de compter avec un pas de $+1$ ou -1 selon la valeur d'une entrée supplémentaire,
– la possibilité de compter ou non selon la valeur d'une entrée supplémentaire,
– la possibilité de mettre le contenu à zéro si une entrée supplémentaire vaut 1,
– la possibilité de fonctionner comme un registre aussi, pour qu'une valeur externe puisse être chargée si une entrée supplémentaire vaut 1,
– la possibilité d'utiliser un codage différent pour les nombres (comme le code Gray, le code à 7 segments, etc),
– la possibilité d'utiliser des pas autres que $+1$ et -1.

10.3 MULTIPLICATION BINAIRE

Comme indiqué dans le chapitre 7, la multiplication est trop complexe pour être réalisée par un circuit combinatoire.

La solution du problème sera l'utilisation d'un circuit séquentiel qui divise le travail en étapes, une par cycle d'horloge.

L'algorithme que nous utilisons est le même que celui servant pour la multiplication binaire ordinaire, sauf que l'ordre entre les étapes est légèrement modifié. Avec la multiplication ordinaire, on calcule d'abord tous les résultats intermédiaires obtenus par produit du premier opérande avec chaque chiffre du deuxième opérande. Finalement, on additionne les résultats intermédiaires pour obtenir le résultat final. Dans la version modifiée, nous maintenons un *accumulateur* contenant la somme de tous les résultats intermédiaires

déjà produits. Lorsque le dernier résultat intermédiaire est calculé, on obtient le résultat final.

Pour comprendre la faisabilité, on remarque d'abord que le résultat de la multiplication de deux nombres positifs à n chiffres peut contenir jusqu'à $2n$ chiffres.

Notons le premier opérande x, le deuxième y et le résultat r. D'abord nous écrivons y sous la forme :

$$y_{n-1}2^{n-1} + y_{n-2}2^{n-2} + \ldots + y_0 2^0.$$

À tout instant i, le résultat contient x multiplié par

$$y_{i-1}2^{n-1} + y_{i-2}2^{n-2} + \ldots + y_0 2^{n-i}.$$

Il est clair que lorsque $i = n$, nous disposons du résultat final. Afin de passer de l'étape i à l'étape $i + 1$, il faut diviser le résultat par 2 et additionner x multiplié par $y_i\ 2^{n-1}$. Or il revient au même de commencer par additionner x multiplié par $y_i\ 2^n$, puis de diviser le résultat par 2. Ce processus doit être répété n fois.

Notre première tentative de circuit pour la multiplication est présentée figure 10.4.

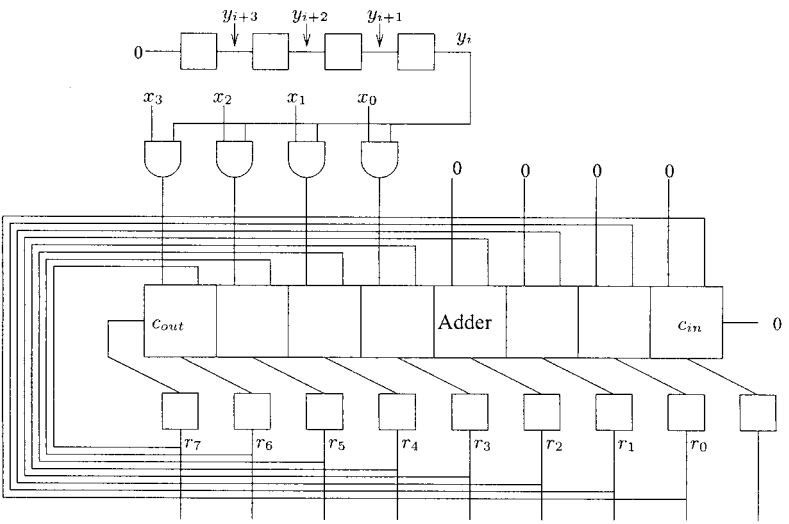


Figure 10.4 : Multiplication : circuit 0

Ce circuit fonctionne très bien, mais il peut être simplifié. D'abord on remarque que la partie la moins significative de l'additionneur additionne toujours r_3, r_2, r_1 et r_0 avec 0. Utiliser un additionneur pour cela n'est pas nécessaire, et le circuit peut ainsi être simplifié comme présenté par la figure 10.5.

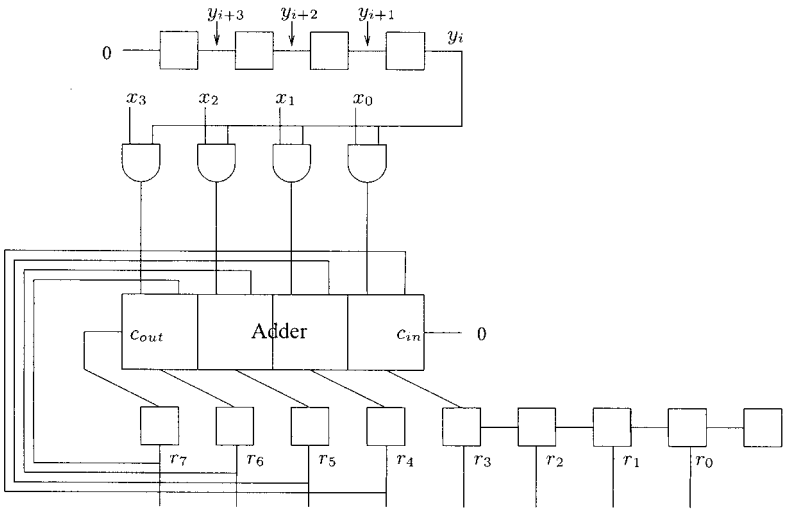


Figure 10.5 : Multiplication : circuit 1

Ensuite, puisque r_3 (ou r_{n-1} en général) contient initialement 0, le bistable D le plus à droite aussi contient toujours 0, ce qui est correct car il n'est jamais utilisé. On obtient le circuit de la figure 10.6.

Pour la dernière simplification, on remarque que les bistables contenant r_{n-1} à r_0 et les n bistables supérieurs ne contiennent jamais simultanément des informations. Initialement, tous les bistables supérieurs contiennent des informations utiles, et aucun des n bistables inférieurs les plus à droite n'en contient. Après la première étape, r_{n-1} contient une information utile, mais le bistable supérieur le plus à gauche n'en contient plus. Il est donc possible d'utiliser un seul jeu de n bistables pour contenir à la fois y et les n bits inférieurs du résultat. Le circuit final est présenté par la figure 10.7.

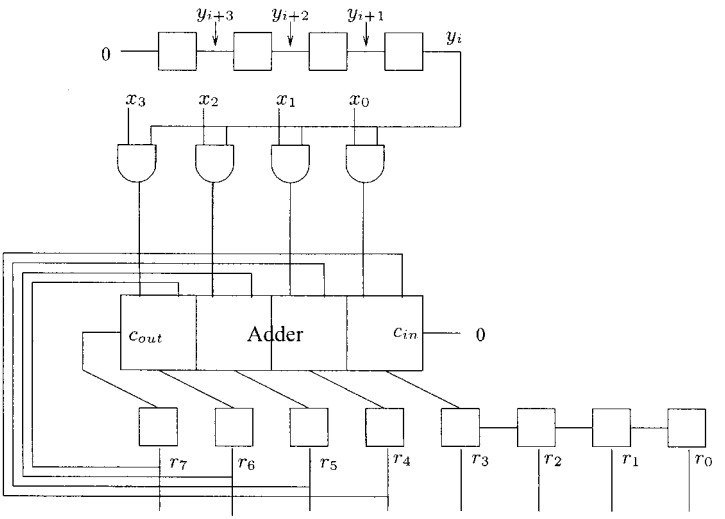


Figure 10.6 : Multiplication : circuit 2

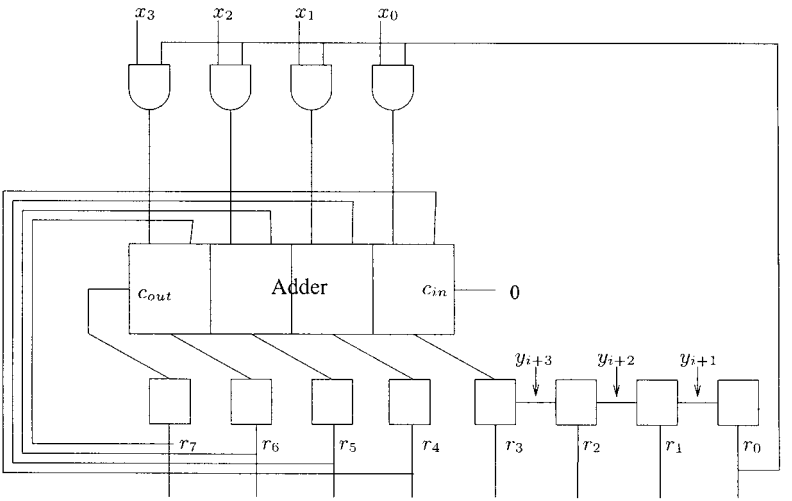


Figure 10.7 : Multiplication : circuit final

10.4 RÉCAPITULATIF

Comme avec les circuits combinatoires, certains circuits séquentiels se prêtent mal à la méthode générale de conception. L'utilisation de cette méthode donnerait des circuits de taille trop importante.

Ces circuits sont à la fois très réguliers et très fréquemment utilisés. Il est donc justifié de concevoir pour ceux-ci une version optimisée une fois pour toutes.

EXERCICES

Exercice 10.1.

(1) Réaliser un compteur modulo 3, c'est-à-dire un circuit séquentiel donnant la séquence 0, 1, 2, 0, 1, . . .

(2) Comment se comporte-t-il si son état initial est 3 ?

(3) Comment peut-on compléter ce circuit pour permettre de l'initialiser à une valeur arbitraire ?

Exercice 10.2.

(1) Réaliser un circuit séquentiel donnant la séquence

$$0, 1, 2, 3, 4, 5, 4, 3, 2, 1, 0, \ldots$$

(2) Combien faut-il de bistables pour réaliser un circuit donnant la séquence indiquée ci-dessous ?

$$0, 1, 2, \ldots, n-1, n, n-1, \ldots, 2, 1, 0, \ldots$$

Exercice 10.3. Réaliser un circuit séquentiel qui, après n cycles d'horloge, donne la représentation binaire de $11*n$ modulo 16.

Chapitre **11**

Logique à trois états

11.1 LOGIQUE À TROIS ÉTATS

Les circuits combinatoires (voir chapitre 3) et les circuits séquentiels (voir chapitre 9) sont relativement faciles à comprendre puisqu'il suffit de se familiariser avec le fonctionnement des portes logiques et avec leur processus d'interconnexion. Avec les circuits de *logique à trois états*, ceci n'est plus vrai. Comme leur nom l'indique, ils manipulent des signaux pouvant être dans l'un des *trois* états possibles, et non plus seulement 0 et 1. Bien que cela puisse paraître étrange à première vue, l'idée est relativement simple.

Considérons le cas relativement fréquent d'un certain nombre de circuits $S_1, S_2, \ldots$, dans différentes parties d'une puce de silicium (c'est-à-dire que les circuits ne sont pas vraiment géographiquement proches). À un instant donné, exactement *un* des circuits source produit un signal binaire devant être distribué à un ensemble de circuits $D_1, D_2, \ldots$, eux aussi dans différentes parties de la puce. Afin de garantir qu'au plus un circuit produit un signal à un instant donné, il faut un signal pour sélectionner celui des circuits source qui doit produire le signal. Supposons que nous ayons des signaux $s_1, s_2, \ldots$ pour cela. Une solution du problème est indiquée dans la figure 11.1.

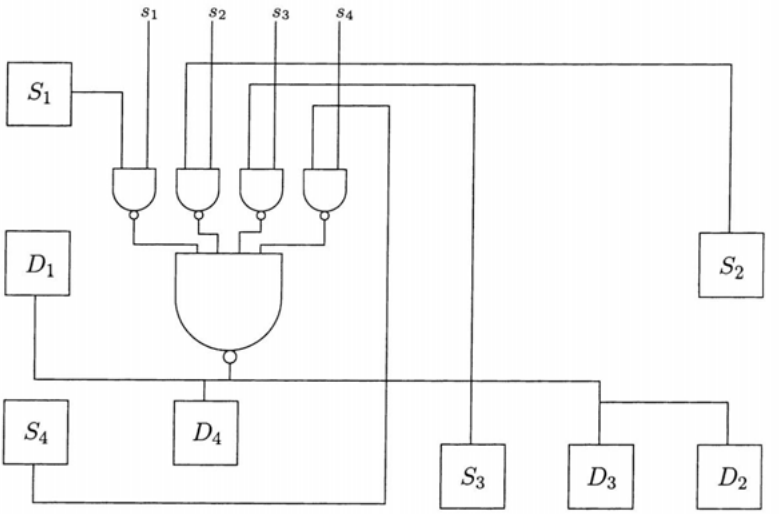


Figure 11.1 : Logique à trois états 1

Comme on peut le constater, cette solution nécessite que toutes les sorties soit routées vers un endroit central. Souvent ce type de solution est coûteux. Puisque au plus une source à la fois n'est active, il devrait être possible de trouver une solution similaire à celle présentée figure 11.2.

Mais connecter les sorties de deux ou plusieurs circuits ensemble peut détruire ceux-ci. Pour résoudre le problème, il faut utiliser des circuits à *trois états*.

11.2 RETOUR VERS LES TRANSISTORS

Un circuit à trois états (combinatoire ou séquentiel) est à la base un circuit ordinaire, mais avec une entrée supplémentaire appelée *enable*. Lorsque *enable* vaut 1, le circuit se comporte comme le circuit ordinaire correspondant. Par contre, lorsque *enable* vaut 0, *les sorties sont complètement déconnectées du reste du circuit*. C'est comme si nous avions pris un circuit ordinaire, puis ajouté à chaque sortie un interrupteur fermé lorsque *enable* vaut 1, et ouvert lorsque *enable*

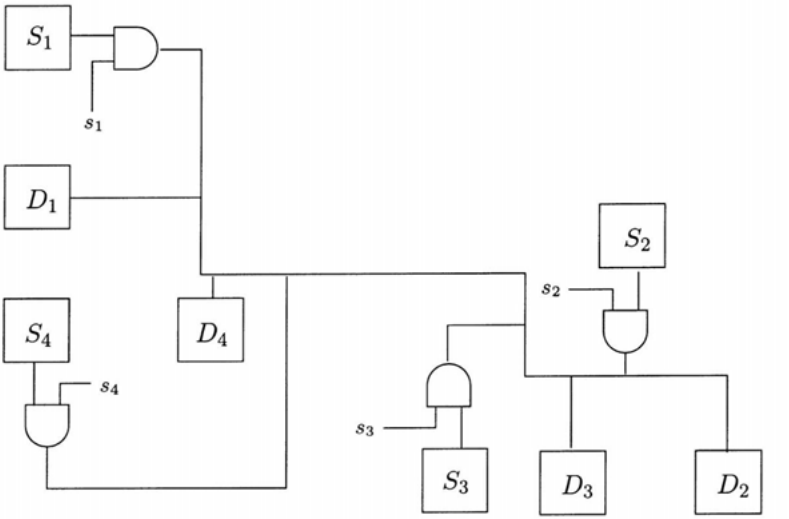


Figure 11.2 : Logique à trois états 2

vaut 0, comme indiqué figure 11.3. Ceci est en fait très proche de la vérité. En effet, l'interrupteur est un autre transistor qui peut être ajouté à un coût très faible.

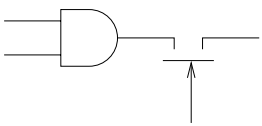


Figure 11.3 : Interrupteurs

N'importe quel circuit peut exister en version à trois états. Mais il est également possible de convertir un circuit ordinaire en un circuit à trois états. Pour cela, il faut utiliser un circuit à trois états combinatoire spécial qui copie simplement ses entrées sur ses sorties, mais qui a une entrée *enable*. Un tel circuit est appelé un *pilote de bus* (en anglais : *bus driver*). Un pilote de bus à une seule entrée est représenté par le pictogramme de la figure 11.4. La figure 11.5 en montre une version pour des signaux bidirectionnels.

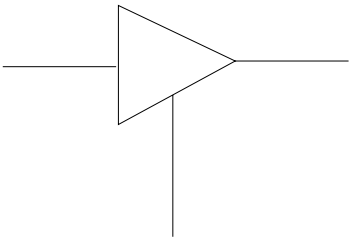


Figure 11.4 : Pilote de bus unidirectionnel

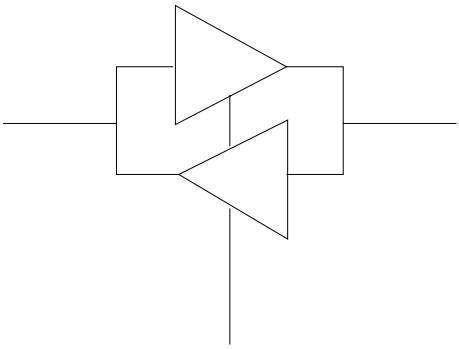


Figure 11.5 : Pilote de bus bidirectionnel

11.3 BUS

Avec la logique à trois états (voir section 11.1), on peut utiliser un seul fil pour connecter toutes les sources et toutes les destinations d'un signal commun. Ce fil peut faire gagner beaucoup d'espace sur la puce et aussi diminuer considérablement la distance parcourue par un signal.

Un tel fil, ou une collection de fils pour un signal multi-bits, s'appelle un *bus*.

11.4 RÉCAPITULATIF

La notion de bus permet de diminuer le nombre de fils utilisés pour connecter les différents éléments d'un circuit compliqué, la raison

essentielle étant qu'un bus est *bidirectionnel*. Les circuits habituels ne permettent qu'une seule direction des signaux. Pour réaliser un bus, il nous faut un nouveau type de circuit, à trois états. Le troisième état représente le fait que les sorties du circuit sont totalement isolées de leur environnement.

EXERCICES

Pour comparer le coût d'un circuit réalisé avec des portes logiques à celui d'un circuit équivalent réalisé avec des portes *trois états*, il est utile de connaître le nombre de transistors nécessaires pour réaliser chacun des éléments dans la technologie CMOS.

porte	nombre de transistors
inverseur (NOT)	2
portes NAND/NOR avec 2 entrées	4
portes NAND/NOR avec 3 entrées	6
portes NAND/NOR avec 4 entrées	8
porte logique *trois états*	2

Pour des raisons techniques, les portes NAND / NOR avec plus de 4 entrées ne sont pas implémentées directement mais doivent être réalisées en combinant des portes NOT et NAND / NOR avec au plus 4 entrées.

Exercice 11.1. Implémenter un multiplexeur 2:1 avec des portes *trois états* et des inverseurs.
Comparer le coût (nombre de transistors) avec une implémentation utilisant les portes NAND / NOR / NOT.

Exercice 11.2. Implémenter un démultiplexeur 1:2 avec des portes *trois états* et des inverseurs.

Exercice 11.3. En utilisant des multiplexeurs, implémenter un bus permettant d'échanger des données entre 4 registres R_0, R_1, R_2, R_3 de 8 bits chacun. Le bus doit permettre de réaliser une opération de type $R_i \leftrightarrow R_j$, $i \neq j$, c'est-à-dire l'échange des contenus de R_i et R_j en un seul cycle d'horloge.

Chapitre **12**

Mémoires

Une mémoire n'est ni un circuit séquentiel (voir chapitre 9), car notre définition de circuit séquentiel exige que le circuit soit synchrone avec un signal d'horloge, ni un circuit combinatoire (voir chapitre 3), car les valeurs des sorties dépendent des anciennes valeurs des sorties.

12.1 MÉMOIRES À LECTURE/ÉCRITURE

En général, une mémoire a m entrées appelées *entrées d'adresse* qui sont utilisées pour sélectionner exactement 1 parmi 2^m *mots*, chacun contenant n bits.

De plus, elle a n fils *bidirectionnels* appelés *lignes de données*. Ces lignes de données servent à la fois d'entrée pour stocker des informations dans un mot sélectionné par les entrées d'adresse, et de sortie pour récupérer un mot préalablement stocké. Une telle solution réduit le nombre de connecteurs requis d'un facteur deux.

Finalement, la mémoire a une entrée appelée $enable$ (voir chapitre 11) qui détermine si les lignes de données sont dans un état stable ou non, ainsi qu'une entrée appelée r/w qui détermine la direction des lignes de données.

Une mémoire avec des valeurs arbitraires pour m et n peut être construite à partir de mémoires plus petites. Pour illustrer cette construction, nous commençons par réaliser une mémoire d'un seul bit (avec $m = 0$ et $n = 1$). Le circuit correspondant est illustré par la figure 12.1.

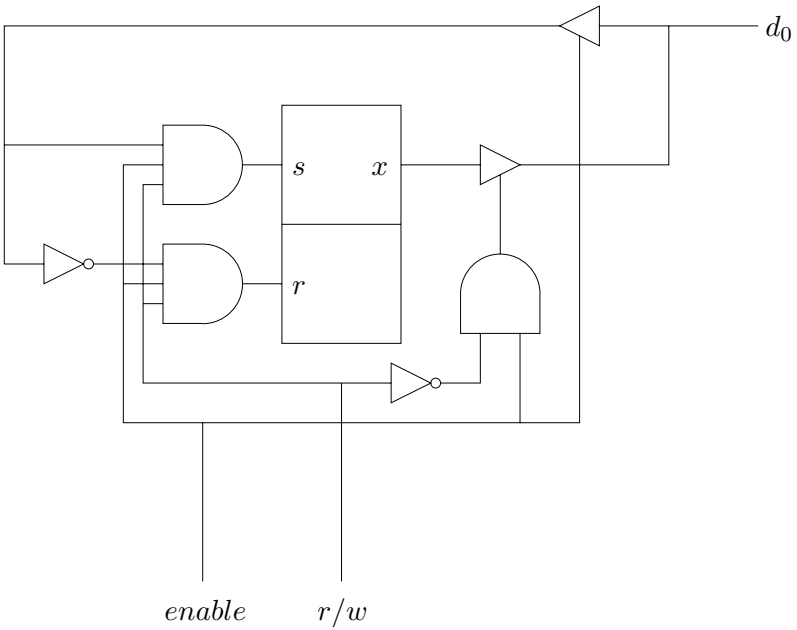


Figure 12.1 : Mémoire de 1 bit

La partie centrale du circuit est une bascule SR capable de maintenir un bit d'information. Lorsque $enable$ vaut 0, la sortie d_0 est isolée tant des entrées que de la sortie de la bascule. L'information est transmise de d_0 vers les entrées de la bascule lorsque $enable$ vaut 1 et r/w vaut 1 (indiquant une écriture). L'information est transmise de la sortie x de la bascule vers d_0 lorsque $enable$ vaut 1 et r/w vaut 0 (indiquant une lecture).

Nous allons maintenant combiner des mémoires 1-bit pour obtenir des mémoire plus grandes. D'abord, supposons que nous ayons n mémoires de 2^m mots, chacun d'un seul bit. Nous pouvons facilement obtenir une mémoire de 2^m mots, chacun de n bits. Le circuit est présenté figure 12.2.

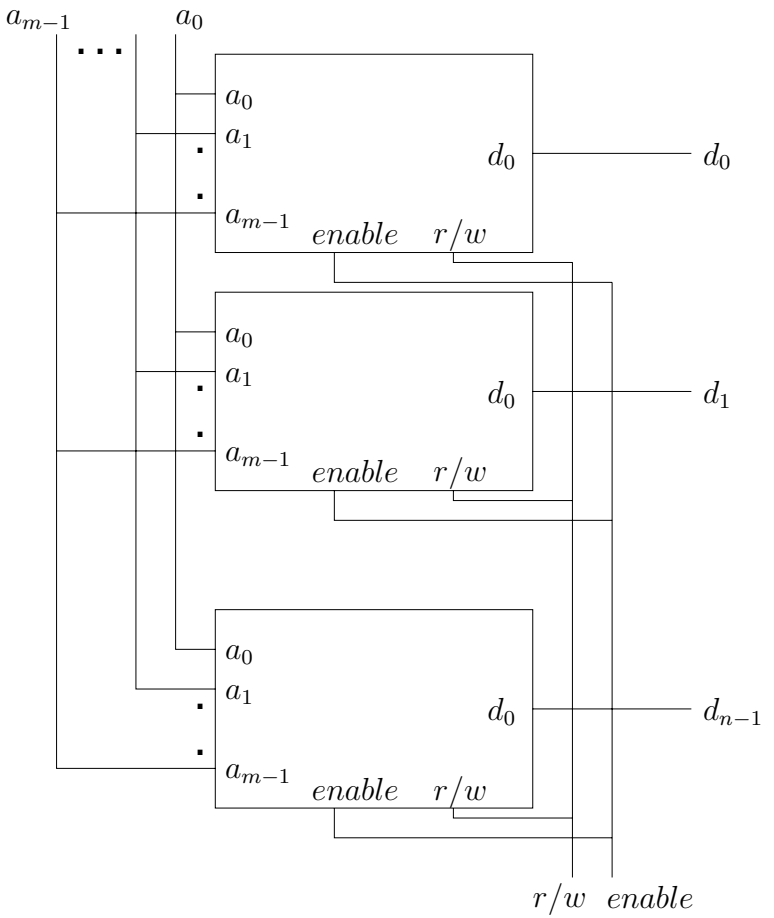


Figure 12.2 : Mémoire de 2^m mots de *n* bits

Nous avons simplement connecté toutes les entrées d'adresse ensemble, de même que tous les $enable$ et tous les r/w. Chaque mémoire d'un seul bit fournit un bit des n bits d'un mot.

Nous allons maintenant fabriquer des mémoires avec plusieurs mots. Pour cela, nous supposons avoir à notre disposition deux mémoires, chacune avec m entrées d'adresse et n lignes de données. Nous allons montrer comment connecter ces deux mémoires pour en obtenir une seule de $m + 1$ entrées d'adresse et n lignes de données. Le circuit est illustré par la figure 12.3.

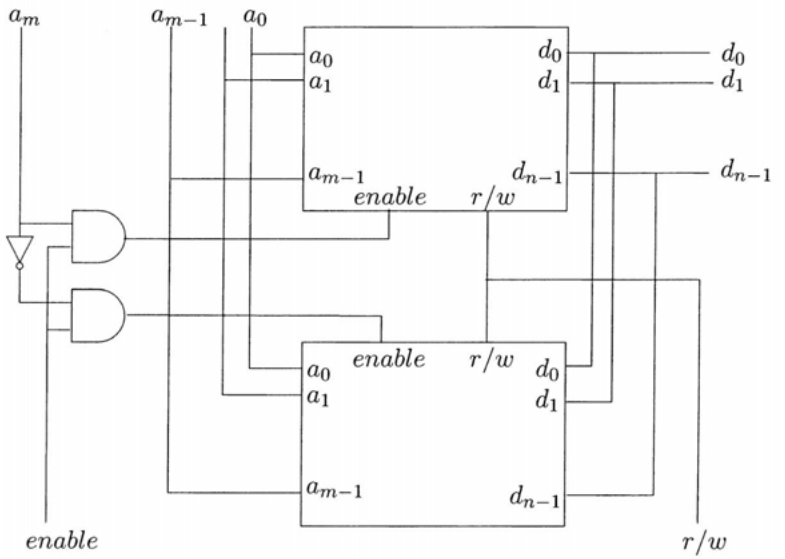


Figure 12.3 : Mémoire de 2^{m+1} mots de *n* bits

Comme on peut le constater, l'entrée d'adresse supplémentaire est combinée avec l'entrée $enable$ pour sélectionner l'une des deux petites mémoires. Au plus l'une des deux sera connectée sur les lignes de données grâce au fonctionnement de la logique à trois états.

12.2 MÉMOIRE À LECTURE SEULE

Une *mémoire à lecture seule* (en anglais : *read-only memory* ou *ROM*) est similaire à une mémoire ordinaire (voir section 12.1), sauf qu'elle n'est pas réinscriptible : son contenu est fixé à l'usine.

Puisque le contenu ne peut être modifié, il n'y a pas de signal r/w. Mis à part le signal $enable$, une mémoire à lecture seule est donc comme un circuit combinatoire ordinaire à m entrées et n sorties.

Si une mémoire à lecture seule est apparentée à un circuit combinatoire, alors pourquoi la traiter séparément ? La réponse est que ces mémoires sont parfois *programmables*. Elles sont souvent vendues

avec un contenu de 0 ou de 1 partout. L'utilisateur peut la mettre dans une machine spéciale pour la remplir avec le contenu qu'il souhaite ; la mémoire peut donc être *programmée*. Dans ce cas, nous parlons de PROM (pour *programmable ROM*).

Certains types de PROM peuvent être effacés et reprogrammés. Ces mémoires sont par exemple effaçables à l'aide de rayons ultraviolets. Une PROM effaçable s'appelle parfois EPROM (pour *erasable PROM*).

12.3 RÉCAPITULATIF

Les mémoires forment un type de circuit intermédiaire entre le circuit combinatoire et le circuit séquentiel. Elles ne sont pas combinatoires car les sorties prennent des valeurs qui dépendent du passé, mais elles ne sont pas séquentielles non plus car notre définition de circuit séquentiel impose l'existence d'une horloge, ce qui n'est pas le cas pour les mémoires.

EXERCICES

Exercice 12.1. Nous avons à notre disposition 4 puces de mémoire de largeur 8 bits et hauteur $2^{13} = 8\,192$. Les connecter pour obtenir une mémoire de largeur 16 bits et hauteur $2^{14} = 16\,384$.

Exercice 12.2. Déterminer le nombre maximum de portes que le signal *enable* doit traverser (la profondeur du circuit) afin d'arriver à la cellule élémentaire, si la méthode de ce chapitre a été utilisée, pour construire une mémoire de 16 mots de n bits (la valeur de n n'influence pas la réponse).

Exercice 12.3. En utilisant un démultiplexeur avec 4 entrées d'adresse, diminuer la profondeur de la mémoire de l'exercice précédent à celle du multiplexeur.

Exercice 12.4. Plus généralement, déterminer le nombre maximum de portes que le signal *enable* doit traverser afin d'arriver à la cellule élémentaire, si la méthode de ce chapitre a été utilisée, pour construire une mémoire de 2^m mots de n bits.

PARTIE **2**

EXEMPLE D'ARCHITECTURE

Chapitre **13**

Éléments de base

Nous présentons ici les éléments de base nécessaires à la construction de tout ordinateur. L'étape finale se fait principalement en connectant les différents éléments par des fils conducteurs.

13.1 COMPTEUR AVEC MISE À ZÉRO

Un *compteur avec mise à zéro* est un circuit séquentiel (voir chapitre 9) à une entrée et n sorties. Il diffère d'un compteur ordinaire (voir section 10.2) par le signal d'entrée cl (pour *clear*) dont le but est de remettre son contenu à zéro.

La table d'états (voir section 9.1) d'un compteur de 4 bits de large avec mise à zéro est illustrée par la figure 13.1.

On peut constater que lorsque le signal cl vaut 0, ce compteur se comporte comme un compteur ordinaire ; par contre, lorsque le signal cl vaut 1 son contenu est remis à zéro après le prochain front d'horloge.

cl	o_3	o_2	o_1	o_0	o'_3	o'_2	o'_1	o'_0
0	0	0	0	0	0	0	0	1
0	0	0	0	1	0	0	1	0
0	0	0	1	0	0	0	1	1
0	0	0	1	1	0	1	0	0
0	0	1	0	0	0	1	0	1
0	0	1	0	1	0	1	1	0
0	0	1	1	0	0	1	1	1
0	0	1	1	1	1	0	0	0
0	1	0	0	0	1	0	0	1
0	1	0	0	1	1	0	1	0
0	1	0	1	0	1	0	1	1
0	1	0	1	1	1	1	0	0
0	1	1	0	0	1	1	0	1
0	1	1	0	1	1	1	1	0
0	1	1	1	0	1	1	1	1
0	1	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0
1	0	0	1	0	0	0	0	0
1	0	0	1	1	0	0	0	0
1	0	1	0	0	0	0	0	0
1	0	1	0	1	0	0	0	0
1	0	1	1	0	0	0	0	0
1	0	1	1	1	0	0	0	0
1	1	0	0	0	0	0	0	0
1	1	0	0	1	0	0	0	0
1	1	0	1	0	0	0	0	0
1	1	0	1	1	0	0	0	0
1	1	1	0	0	0	0	0	0
1	1	1	0	1	0	0	0	0
1	1	1	1	0	0	0	0	0
1	1	1	1	1	0	0	0	0

Figure 13.1 : Table d'états du compteur avec remise à 0

13.2 REGISTRE COMPTEUR

Un *registre compteur* est un circuit séquentiel (voir chapitre 9) à $n + 1$ entrées et n sorties. Il diffère d'un compteur ordinaire (voir section 10.2) car il peut aussi fonctionner comme un registre (voir section 10.1). Il diffère d'un registre car, lorsque aucune valeur n'est chargée, il fonctionne comme un compteur au lieu de maintenir son ancienne valeur.

La table d'états (voir section 9.1) d'un registre compteur de 4 bits de large est illustrée par la figure 13.2.

ld	i_3	i_2	i_1	i_0	o_3	o_2	o_1	o_0	o'_3	o'_2	o'_1	o'_0
0	-	-	-	-	0	0	0	0	0	0	0	1
0	-	-	-	-	0	0	0	1	0	0	1	0
0	-	-	-	-	0	0	1	0	0	0	1	1
0	-	-	-	-	0	0	1	1	0	1	0	0
0	-	-	-	-	0	1	0	0	0	1	0	1
0	-	-	-	-	0	1	0	1	0	1	1	0
0	-	-	-	-	0	1	1	0	0	1	1	1
0	-	-	-	-	0	1	1	1	1	0	0	0
0	-	-	-	-	1	0	0	0	1	0	0	1
0	-	-	-	-	1	0	0	1	1	0	1	0
0	-	-	-	-	1	0	1	0	1	0	1	1
0	-	-	-	-	1	0	1	1	1	1	0	0
0	-	-	-	-	1	1	0	0	1	1	0	1
0	-	-	-	-	1	1	0	1	1	1	1	0
0	-	-	-	-	1	1	1	0	1	1	1	1
0	-	-	-	-	1	1	1	1	0	0	0	0
1	c_3	c_2	c_1	c_0	-	-	-	-	c_3	c_2	c_1	c_0

Figure 13.2 : Table d'états du compteur registre

Comme on peut le constater, le registre compteur se comporte comme un compteur ordinaire lorsque le signal ld vaut 0, et comme un registre lorsque le signal ld vaut 1.

13.3 REGISTRE COMPTEUR AVEC MISE À ZÉRO

Un *registre compteur avec mise à zéro* est un circuit séquentiel (voir chapitre 9) à $n + 2$ entrées et n sorties. Il diffère d'un registre compteur ordinaire (voir section 13.2) par le signal d'entrée cl *(clear)* pour la mise à zéro du registre.

Ce qui se passe lorsque les entrées ld et cl valent 1 simultanément est une question de convention ou de préférence. Pour notre application, nous souhaitons que le signal cl soit prioritaire par rapport au signal ld.

La table d'états (voir section 9.1) d'un registre compteur de 4 bits de large avec mise à zéro est illustrée par la figure 13.3.

cl	ld	i_3	i_2	i_1	i_0	o_3	o_2	o_1	o_0	o_3'	o'_2	o'_1	o'_0
0	0	-	-	-	-	0	0	0	0	0	0	0	1
0	0	-	-	-	-	0	0	0	1	0	0	1	0
0	0	-	-	-	-	0	0	1	0	0	0	1	1
0	0	-	-	-	-	0	0	1	1	0	1	0	0
0	0	-	-	-	-	0	1	0	0	0	1	0	1
0	0	-	-	-	-	0	1	0	1	0	1	1	0
0	0	-	-	-	-	0	1	1	0	0	1	1	1
0	0	-	-	-	-	0	1	1	1	1	0	0	0
0	0	-	-	-	-	1	0	0	0	1	0	0	1
0	0	-	-	-	-	1	0	0	1	1	0	1	0
0	0	-	-	-	-	1	0	1	0	1	0	1	1
0	0	-	-	-	-	1	0	1	1	1	1	0	0
0	0	-	-	-	-	1	1	0	0	1	1	0	1
0	0	-	-	-	-	1	1	0	1	1	1	1	0
0	0	-	-	-	-	1	1	1	0	1	1	1	1
0	0	-	-	-	-	1	1	1	1	0	0	0	0
0	1	c_3	c_2	c_1	c_0	-	-	-	-	c_3	c_2	c_1	c_0
1	-	-	-	-	-	-	-	-	-	0	0	0	0

Figure 13.3 : Table d'états du registre compteur avec mise à 0

C'est exactement le circuit dont nous avons besoin pour le compteur ordinal du micro-programme (micro-PC) (en anglais : *micro-Program Counter*) de notre exemple d'ordinateur, mis à part que pour notre ordinateur, micro-PC aura 6 bits de large au lieu de 4.

13.4 REGISTRE COMPTEUR AVEC MISE À ZÉRO ET INCRÉMENTATION

Un *registre compteur avec mise à zéro et incrémentation* est un circuit séquentiel (voir chapitre 9) à $n + 3$ entrées et n sorties. Il diffère

d'un registre compteur avec remise à zéro (voir section 13.3), car il a en plus un signal d'entrée $incr$ pour un comptage explicite. Lorsque $incr$ vaut 0, la sortie ne change pas après le front d'horloge et lorsque $incr$ vaut 1, le circuit fonctionne comme un compteur.

Pour ce circuit, le signal d'entrée ld est prioritaire par rapport au signal $incr$. Lorsque les deux valent 1, le registre est chargé à partir des n entrées. Le signal cl est le plus prioritaire.

cl	ld	$incr$	i_3	i_2	i_1	i_0	o_3	o_2	o_1	o_0	o'_3	o'_2	o'_1	o'_0
0	0	0	-	-	-	-	c_3	c_2	c_1	c_0	c_3	c_2	c_1	c_0
0	0	1	-	-	-	-	0	0	0	0	0	0	0	1
0	0	1	-	-	-	-	0	0	0	1	0	0	1	0
0	0	1	-	-	-	-	0	0	1	0	0	0	1	1
0	0	1	-	-	-	-	0	0	1	1	0	1	0	0
0	0	1	-	-	-	-	0	1	0	0	0	1	0	1
0	0	1	-	-	-	-	0	1	0	1	0	1	1	0
0	0	1	-	-	-	-	0	1	1	0	0	1	1	1
0	0	1	-	-	-	-	0	1	1	1	1	0	0	0
0	0	1	-	-	-	-	1	0	0	0	1	0	0	1
0	0	1	-	-	-	-	1	0	0	1	1	0	1	0
0	0	1	-	-	-	-	1	0	1	0	1	0	1	1
0	0	1	-	-	-	-	1	0	1	1	1	1	0	0
0	0	1	-	-	-	-	1	1	0	0	1	1	0	1
0	0	1	-	-	-	-	1	1	0	1	1	1	1	0
0	0	1	-	-	-	-	1	1	1	0	1	1	1	1
0	0	1	-	-	-	-	1	1	1	1	0	0	0	0
0	1	-	c_3	c_2	c_1	c_0	-	-	-	-	c_3	c_2	c_1	c_0
1	-	-	-	-	-	-	-	-	-	-	0	0	0	0

Figure 13.4 : Table d'états du registre compteur avec mise à 0 et incrémentation

La table d'états (voir section 9.1) d'un registre compteur de 4 bits de large avec mise à zéro et incrémentation est illustrée par la figure 13.4.

C'est exactement le circuit que nous allons utiliser pour le compteur ordinal ou PC (en anglais : *Program Counter*) de notre exemple d'ordinateur, sauf que le PC de notre ordinateur aura 8 bits de large.

13.5 MICRO-MÉMOIRE

La micro-mémoire d'un processeur est un circuit combinatoire (voir chapitre 3) réalisé parfois sous forme de mémoire à lecture (voir section 12.2), parfois sous forme de circuit combinatoire. Il fut un temps où il était relativement fréquent de vouloir modifier le contenu de la micro-mémoire afin de créer des instructions spécialisées pour des applications particulières. La micro-mémoire était alors similaire à une mémoire à lecture et écriture ordinaire (voir section 12.1). Un tel processeur est dit *micro-programmable*.

Le nombre d'entrées d'adresse nécessaire dépend du nombre d'instructions du processeur et de leur complexité. Dans un processeur CISC *(Complex Instruction Set Computer)* traditionnel tel que le VAX, la micro-mémoire peut avoir une taille très importante et être structurée avec des sous-routines et d'autres structures de contrôle habituellement utilisées au niveau des instructions. Dans un processeur RISC *(Reduced Instruction Set Computer)*, la micro-mémoire peut être si simple qu'un circuit combinatoire ordinaire est suffisant pour la réaliser. Dans notre ordinateur, la micro-mémoire aura 6 entrées d'adresse pour un maximum de $2^6 = 64$ adresses différentes.

Le nombre de lignes de données est le même que le nombre de micro-opérations de la machine. Dans notre d'ordinateur, nous avons initialement besoin de 15 micro-opérations, mais ce nombre doit pouvoir être étendu afin de permettre l'ajout de nouvelles fonctionnalités comme les sauts conditionnels, la gestion d'un pointeur de pile, etc.

13.6 DÉCODEUR D'INSTRUCTIONS

Le *décodeur d'instructions* d'un processeur est un circuit combinatoire (voir chapitre 3) réalisé parfois sous forme de mémoire à lecture (voir section 12.2), parfois sous forme de circuit combinatoire ordinaire. Son but est de traduire le code d'une instruction en l'adresse de la micro-mémoire où commence le micro-code la concernant (voir section 13.5).

Dans notre exemple d'ordinateur, l'entrée du décodeur d'instructions a 5 bits de large pour un total de 32 instructions possibles. La sortie est de 6 bits correspondant à la largeur du micro-PC (voir section 13.3) et de la micro-mémoire (voir section 13.5).

13.7 UNITÉ ARITHMÉTIQUE ET LOGIQUE (ALU)

Une *unité arithmétique et logique* (ALU) d'un processeur est un circuit combinatoire (voir chapitre 3) capable d'effectuer les opérations de base comme l'addition, la soustraction, le *ou* et le *et* bit-à-bit, etc.

La multiplication est exclue, car elle n'est pas implémentée sous forme de circuit combinatoire.

Une implémentation possible de l'ALU est de réaliser toutes les opérations en parallèle puis, à l'aide d'un multiplexeur (voir section 5.1), de sélectionner l'une des sorties selon l'opération souhaitée. La figure 13.5 schématise l'ALU.

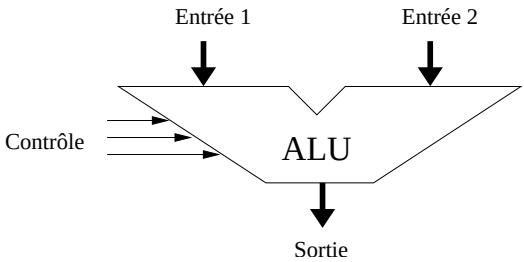


Figure 13.5 : Schéma de l'ALU

Dans notre exemple d'ordinateur, l'ALU a deux entrées de 8 bits chacune, une sortie de 8 bits (qui sera étendue à 12 plus tard pour la gestion des débordement et de la retenue) et trois fils de contrôle. Les trois fils de contrôle sélectionnent l'opération à effectuer. Dans notre exemple d'ordinateur, pour ces trois fils, nous utilisons les significations indiquées par la figure 13.6.

Combinaison	Nom	Valeur de la sortie
`000`	`copy`	la première entrée
`001`	`shl`	la 2^{e} entrée décalée à gauche d'une position
`010`	`shr`	la 2^{e} entrée décalée à droite d'une position
`011`	`add`	la somme des deux entrées
`100`	`sub`	la différence des deux entrées
`101`	`and`	le *et* logique entre les deux entrées
`110`	`or`	le *ou* logique entre les deux entrées
`111`	`not`	le *non* logique de la deuxième entrée

Figure 13.6 : Contrôle de l'ALU

On utilise la deuxième entrée au lieu de la première pour les opérations de décalage car il est souvent nécessaire de décaler de plusieurs positions. Dans le cas de notre solution, cela peut se faire avec une suite d'instructions de décalage. Si nous avions choisi la première entrée, il serait nécessaire de copier le contenu du registre R1 (branché sur la deuxième entrée de l'ALU, voir chapitre 14) dans le registre R0 (branché sur la première entrée) entre chaque décalage.

13.8 RÉCAPITULATIF

Un ordinateur nécessite un certain nombre d'éléments de base qui sont des variations de circuits combinatoires, de circuits séquentiels et de mémoires déjà présentés.

Chapitre **14**

Le premier ordinateur

14.1 ARCHITECTURE DU PREMIER ORDINATEUR

Le diagramme de la première version de notre exemple d'ordinateur est présenté à la figure 14.1.

À gauche du diagramme, on trouve la micro-mémoire (voir section 13.5) adressable par 64 adresses différentes, chacune désignant un mot de 15 bits (ces bits étant numérotés de 1 à 15) qui correspond à une *micro-opération* ou *MOP*. Les MOP pilotent chacune des aspects de l'ordinateur.

À gauche, on trouve aussi le *compteur ordinal* pour le micro-programme (micro-PC). Il s'agit d'un registre compteur avec mise à zéro (voir section 13.3). Micro-PC a 6 bits de large pour un nombre total de 64 valeurs différentes. Micro-PC est piloté par les MOP 1 et 2. Lorsque MOP1 vaut 1, micro-PC sera mis à zéro après le prochain front d'horloge. De même, lorsque MOP2 vaut 1, une nouvelle valeur de micro-PC sera chargée à partir du décodeur d'instructions (voir section 13.6). Micro-PC peut être mis à zéro manuellement par le bouton *reset*. Puisque l'entrée clr de micro-PC est prioritaire par rapport à l'entrée ld, celui-ci sera toujours mis à zéro par le bouton *reset*, indépendamment de la valeur de MOP2.

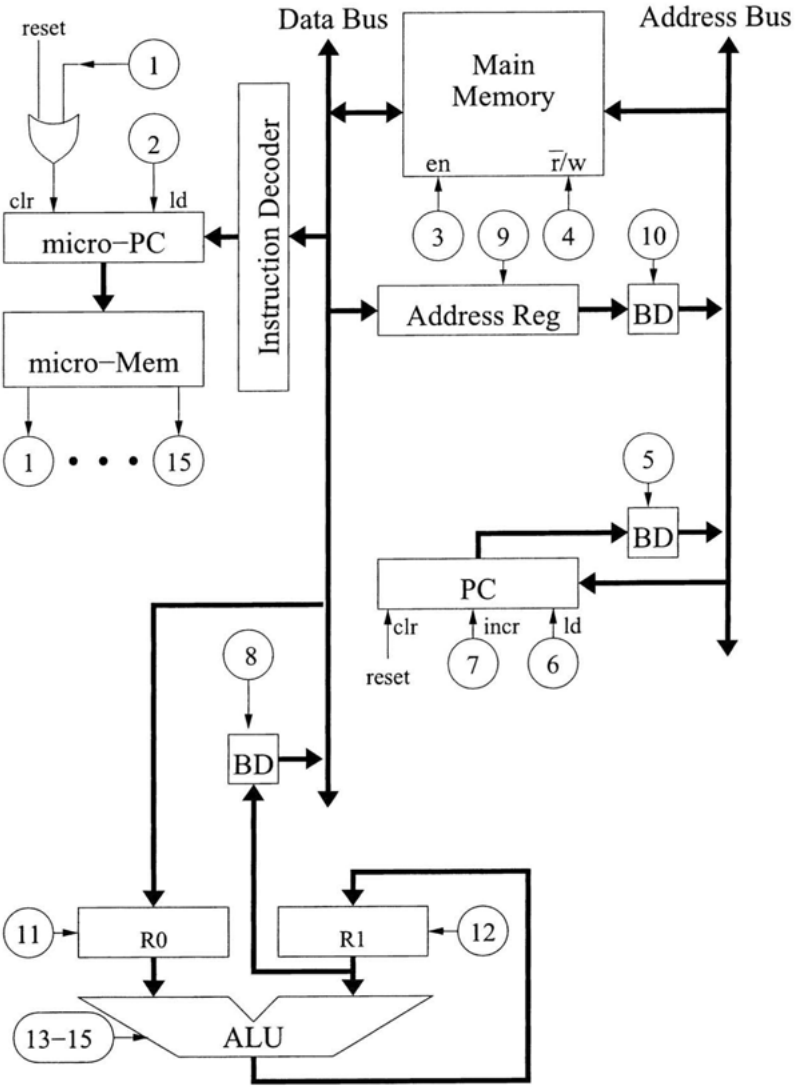


Figure 14.1 : Première version de l'ordinateur

Le dernier circuit, à gauche du bus de données (Data Bus) est le décodeur d'instructions (voir section 13.6). Son rôle est de traduire le code d'une instruction en une adresse de la micro-mémoire, celle où commence le micro-programme qui réalise l'instruction. L'entrée du décodeur d'instructions correspond aux 5 bits les moins significatifs du bus de données ce qui donne la possibilité de définir 32 instructions différentes (32 codes d'instruction différents). La sortie contient 6 bits ; ces derniers sont utilisés pour adresser la micro-mémoire.

En bas du diagramme se trouve l'unité arithmétique et logique (ou ALU) avec deux registres ordinaires (voir section 10.1), R0 et R1. MOP11 est utilisée pour déterminer si R0 doit être chargé à partir du bus de données ou rester inchangé. MOP12 est utilisée pour déterminer si R1 doit être chargé à partir de la sortie de l'unité arithmétique et logique. L'unité arithmétique et logique utilise les MOP 13, 14 et 15 pour sélectionner l'opération à effectuer. Les entrées de l'unité arithmétique et logique contiennent chacune 8 bits en provenance des registres R0 et R1 et la sortie (aussi de 8 bits) est connectée à l'entrée de R1. La sortie de R1 peut être connectée au bus de données grâce à MOP8 branchée sur le pilote de bus (en anglais : *Bus Driver*).

La partie centrale en haut du diagramme contient la mémoire centrale (voir section 12.1). L'entrée $enable$ est pilotée par MOP3 et l'entrée r/w par MOP4. Lorsque MOP4 vaut 1, une écriture en mémoire est effectuée. L'adresse à utiliser pour les opérations sur la mémoire (lecture ou écriture) est prise sur le bus d'adresse. En cas de lecture, les données lues dans la mémoire à l'adresse indiquée par le bus d'adresse seront disponibles sur le bus de données. Dans le cas d'une écriture, les données présentes sur le bus de données seront écrites en mémoire à l'adresse indiquée par le bus d'adresse. Les lignes de données de la mémoire sont donc bidirectionnelles.

Au milieu du diagramme, on trouve un autre registre (voir section 10.1), le *registre d'adresse*. Il est utilisé pour communiquer des données du bus de données au bus d'adresse. Ce registre est chargé lorsque MOP9 vaut 1. Le contenu du registre d'adresse peut être transmis sur le bus d'adresse à l'aide de MOP10. De cette manière, une donnée de la mémoire centrale ou de R1 peut être utilisée comme une adresse de la mémoire centrale. La différence principale entre R0, R1 d'une part et le registre d'adresse d'autre part, réside dans le fait que ce dernier n'est pas visible par le programmeur de l'ordinateur ; R0 et R1 font partie du *modèle de programmation* (voir annexe A) de l'ordinateur, mais ce n'est pas le cas du le registre

d'adresse. Le manuel d'utilisation contenant la description de chaque instruction de l'ordinateur mentionne R0 et R1, car plusieurs de ses instructions utilisent leur valeur ou la modifient. Par contre, le registre d'adresse n'est pas directement manipulable par le programmeur. Il est utilisé en interne pour la réalisation de certaines instructions mais, dans le manuel, l'explication du fonctionnement des instructions ne le mentionne pas.

À droite du diagramme, on trouve le *compteur ordinal* (*PC* pour Progam Counter) qui est un registre avec mise à zéro et incrémentation explicite (voir section 13.4). Pour que le bouton *reset* fonctionne toujours, le signal clr est prioritaire. Si le signal clr vaut 0, mais que le signal ld (MOP6) vaut 1, alors PC est chargé à partir du bus d'adresse. Si clr et ld valent 0, mais que $incr$ (MOP7) vaut 1, alors PC sera incrémenté après le front d'horloge suivant. Finalement, si clr, ld et $incr$ valent 0, alors la valeur de PC reste intacte après le prochain front d'horloge. La valeur de PC peut être transmise au bus d'adresse par le biais de MOP5. C'est de cette manière que les instructions à exécuter sont récupérées en mémoire centrale.

14.2 CONTENU DE LA MICRO-MÉMOIRE

Nous allons maintenant étudier le contenu de la micro-mémoire. La largeur de cette mémoire est de 15 bits, un bit par MOP. Nous allons organiser le contenu de la micro-mémoire de telle sorte que l'adresse numéro 0 contienne toujours le début d'un micro-programme dont le but est de récupérer le code d'une instruction en mémoire centrale à l'adresse indiquée par PC. Il faut donc être sûr que PC contienne l'adresse de l'instruction suivante à exécuter lorsque ce micro-programme démarre.

14.2.1 Chargement et décodage d'une instruction

Une instruction en mémoire centrale commence toujours par un *code* d'instruction sur un octet suivi éventuellement d'arguments. Le chargement et le décodage du code d'une instruction est une opération relativement simple qui ne nécessite qu'un seul cycle d'horloge. Il faut mettre le contenu de PC sur le bus d'adresse en utilisant MOP5. La mémoire centrale doit être lue à l'aide de MOP3. Le contenu

de la mémoire centrale à l'adresse indiquée par PC sera automatiquement traduit par le décodeur d'instructions en une adresse qui indique le début du micro-programme réalisant l'instruction. Cette adresse est chargée dans micro-PC à l'aide de MOP2. Finalement, par convention, nous allons incrémenter PC pour que l'instruction puisse charger ses *arguments* (qui suivent le code de l'instruction en mémoire centrale). Si l'instruction n'a pas d'argument, alors PC contiendra simplement l'adresse du code de l'instruction suivante. Nous allons veiller à ce que cette convention soit toujours respectée.

Nous avons maintenant le micro-programme entier pour le chargement et le décodage d'une instruction :

Adresse	Contenu	Opération
`000000`	`011010100000000`	fetch

Comme on peut le constater, les MOP 2, 3, 5 et 7 valent 1 et toutes les autres valent 0.

14.2.2 Micro-programme pour l'instruction NOP

Nous sommes maintenant prêts à écrire le micro-programme correspondant à chaque instruction individuelle. La première instruction que nous allons traiter est NOP (en anglais : *No OPeration*) qui ne fait absolument rien. Une telle instruction est parfois utilisée pour des raisons d'alignement, ou pour une modification temporaire d'un programme. Nous allons utiliser le code 0 pour cette instruction. Le micro-programme de l'instruction est écrit à partir de la première adresse libre en micro-mémoire, à savoir 1 (car l'adresse 0 contient le micro-programme pour le chargement d'une instruction. Le contenu du décodeur d'instructions à l'adresse 0 sera donc le suivant :

Adresse	Contenu	Opération
`00000`	`000001`	NOP

Le micro-programme de l'instruction NOP ne fait rien. Par conséquent, il faut juste faire en sorte que l'adresse du micro-PC après son exécution vaille 0 pour que l'instruction suivante soit chargée. En fait, le micro-programme de chaque instruction sera terminé par une valeur de 1 pour MOP1 afin de garantir le bon chargement de l'instruction suivante. Avec le micro-programme de l'instruction NOP, voici le contenu actuel de la micro-mémoire :

Adresse	Contenu	Opération
000000	0110101000000000	fetch
000001	1000000000000000	NOP

14.2.3 Micro-programme pour l'instruction LDIMM

L'instruction suivante à réaliser s'appelle LDIMM (en anglais : *load immediate*). C'est une instruction à un seul argument, d'une longueur totale de 2 octets en comptant celui du code de l'instruction. Le premier octet contient comme d'habitude le code de l'instruction et le second contient la valeur à charger dans R0. Une telle instruction est utile pour le chargement de *constantes* (à savoir des valeurs connues à la compilation) dans des registres.

Le code de l'instruction sera 1, premier code libre, et l'adresse en micro-mémoire du début du micro-programme de sa réalisation sera 2, la première adresse libre. Le décodeur d'instructions contient maintenant ceci :

Adresse	Contenu	Opération
00000	000001	NOP
00001	000010	LDIMM

Pour implémenter l'instruction LDIMM, il faut adresser la mémoire en utilisant le contenu de PC qui contient déjà l'adresse de l'octet en mémoire centrale stockant l'argument, car le micro-programme de chargement d'instruction a déjà incrémenté PC. Pour adresser la mémoire à partir de PC, nous utilisons MOP5. De plus, il nous faut MOP3 pour la lecture de la mémoire centrale, ainsi que MOP11 pour stocker la valeur lue dans R0. Par convention, nous allons aussi incrémenter PC pour qu'il contienne le code de l'instruction suivante. Tout ceci peut être fait en un seul cycle d'horloge. Afin de pouvoir charger l'instruction suivante, nous mettons MOP1 à 1 comme dans le cas de l'instruction NOP. Voici donc le contenu de la micro-mémoire :

Adresse	Contenu	Opération
000000	0110101000000000	fetch
000001	1000000000000000	NOP
000010	1010101000010000	LDIMM

14.2.4 Micro-programme pour l'instruction LD

Le chargement de constantes connues à la compilation est utile, mais il est aussi nécessaire de pouvoir charger le contenu de variables dont les valeurs ne sont pas connues à la compilation. De telles variables correspondent à des cellules en mémoire dont les adresses sont, quant à elles, connues à la compilation. Nous avons donc besoin d'une instruction de chargement dont l'argument indique une *adresse* en mémoire centrale et qui charge le contenu de cette adresse dans R0.

Le premier code d'instruction disponible est 2 et la première adresse en micro-mémoire disponible est 3, ce qui donne le contenu du décodeur d'instructions suivant :

Adresse	Contenu	Opération
00000	000001	NOP
00001	000010	LDIMM
00010	000011	LD

Le plan de l'implémentation de LD est le suivant : d'abord utiliser l'adresse contenue dans PC pour adresser la mémoire (MOP 5 et 3) et charger la valeur se trouvant à cette adresse de la mémoire dans le registre d'adresse à l'aide de MOP9. Simultanément, incrémenter PC (MOP7). Puis, pendant le cycle suivant, utiliser le contenu du registre d'adresse pour adresser la mémoire centrale (MOP 10 et 3) et charger le contenu dans R0 (MOP11). Comme on peut le constater, cette instruction nécessite deux cycles d'horloge, simplement parce que deux adresses différentes en mémoire centrale doivent être utilisées, et parce que nous n'avons qu'un seul bus de données et un seul bus d'adresse. Par convention, nous incrémentons PC le plus vite possible, à savoir après le premier cycle d'horloge. Voici le contenu actuel de la micro-mémoire :

Adresse	Contenu	Opération
000000	011010100000000	fetch
000001	100000000000000	NOP
000010	101010100010000	LDIMM
000011	001010101000000	LD
000100	101000000110000	LD

14.2.5 Micro-programme pour l'instruction ST

De la même façon qu'il est nécessaire de pouvoir charger la valeur d'une variable, il faut aussi pouvoir stocker une nouvelle valeur dans une variable. Pour cela, il nous faut une instruction capable de prendre le contenu d'un registre (ici R1) et de le stocker dans la mémoire centrale à l'adresse indiquée par l'argument de l'instruction. Nous appelons cette instruction ST (pour store).

Comme toujours, nous prenons le premier code d'instruction disponible et la première adresse libre de la micro-mémoire, ce qui nous donne le contenu du décodeur d'instructions ci-contre :

Adresse	Contenu	Opération
00000	000001	NOP
00001	000010	LDIMM
00010	000011	LD
00011	000101	ST

Pour implémenter l'instruction ST, il faut prendre des précautions qui jusque là n'étaient pas nécessaires. En effet, la mémoire n'est pas synchrone. Il faut donc s'assurer que les valeurs sur le bus d'adresse et le bus de données sont stables avant de commencer l'écriture en mémoire. La seule manière d'en être sûr est de séparer la production de l'adresse et des données d'une part et l'écriture d'autre part en deux cycles d'horloge différents, et de maintenir l'adresse et la donnée un cycle de plus après avoir activé l'écriture.

Voici le plan de réalisation de l'instruction ST : pendant le premier cycle d'horloge, charger l'argument de l'instruction (indiquant l'adresse en mémoire à utiliser pour le stockage) dans le registre d'adresse, comme pour l'instruction LD. Pendant le deuxième cycle, activer les MOP 8 et 10 pour que les bus soient stables. Pendant le cycle suivant, ajouter le signal d'écriture (MOP4) ainsi que le signal enable (MOP3). L'écriture en mémoire sera alors effectuée. Dans le cycle final, supprimer les MOP 3 et 4 tout en maintenant les MOP 8 et 10. Le contenu de la micro-mémoire est présenté figure 14.2.

14.2.6 Micro-programmes pour les instructions arithmétiques

Charger et stocker des valeurs en mémoire est utile, mais il faut également pouvoir effectuer des opérations arithmétiques et logiques sur ces valeurs. Pour cela, nous définissons huit instructions différentes, une pour chaque combinaison des MOP 13, 14 et 15. Ces

Adresse	Contenu	Opération
000000	0110101000000000	fetch
000001	1000000000000000	NOP
000010	1010101000010000	LDIMM
000011	0010101010000000	LD
000100	1010000001100000	LD
000101	0010101010000000	ST
000110	0000000101000000	ST
000111	0011000101000000	ST
001000	1000000101000000	ST

Figure 14.2 : Contenu de la micro-mémoire

MOP instruisent simplement l'unité arithmétique et logique de combinaison du contenu de R0 et de R1 d'une manière qui dépend des valeurs de ces MOP (voir section 13.7 pour la définition précise de ces combinaisons). Dans un souci de cohérence, nous allons utiliser le même ordre ici. Chacune de ces instructions va prendre un seul cycle d'horloge. Le contenu du décodeur d'instructions est présenté figure 14.3.

Adresse	Contenu	Opération
00000	000001	NOP
00001	000010	LDIMM
00010	000011	LD
00011	000101	ST
00100	001001	COPY
00101	001010	SHL
00110	001011	SHR
00111	001100	ADD
01000	001101	SUB
01001	001110	AND
01010	001111	OR
01011	010000	NOT

Figure 14.3 : Contenu du décodeur d'instructions

L'implémentation de chacune de ces instructions consiste simplement à donner la bonne combinaison des MOP 13, 14 et 15 et à mettre MOP12 à 1 afin que le résultat soit stocké dans R1. Le contenu de la micro-mémoire est présenté figure 14.4.

Adresse	Contenu	Opération
000000	0110101000000000	fetch
000001	1000000000000000	NOP
000010	1010101000010000	LDIMM
000011	0010101010000000	LD
000100	1010000001100000	LD
000101	0010101010000000	ST
000110	0000000101000000	ST
000111	0011000101000000	ST
001000	1000000101000000	ST
001001	1000000000010000	COPY
001010	1000000000010001	SHL
001011	1000000000010010	SHR
001100	1000000000010011	ADD
001101	1000000000011100	SUB
001110	1000000000011101	AND
001111	1000000000011110	OR
010000	1000000000011111	NOT

Figure 14.4 : Contenu de la micro-mémoire

14.2.7 Micro-programme pour l'instruction JAL

Avec les instructions définies pour l'instant, il est impossible de modifier le cours de l'exécution du programme. Seuls des programmes purement séquentiels sont possibles. Afin de modifier le cours de l'exécution, nous avons besoin d'une instruction de *saut*. Nous allons appeler cette instruction JAL (pour *jump always*). Il s'agit d'une instruction de saut *non conditionnel*, car notre architecture ne dispose pas encore de circuit permettant de tester des conditions.

Comme toujours nous prenons le premier code et la première adresse en micro-mémoire disponibles, ce qui donne le contenu du décodeur d'instructions de la figure 14.5.

L'instruction JAL a un seul argument : l'adresse où l'exécution du programme va se poursuivre. Le plan pour l'implémentation de l'instruction JAL est le suivant : pendant le premier cycle d'horloge, charger l'argument dans le registre d'adresse comme pour les instructions LD et ST. Pendant le deuxième cycle, utiliser MOP 10 et 6 pour stocker cette valeur dans PC. Alors qu'il n'est pas nécessaire d'incrémenter PC pendant le premier cycle (car il sera modifié pendant

Adresse	Contenu	Opération
00000	000001	NOP
00001	000010	LDIMM
00010	000011	LD
00011	000101	ST
00100	001001	COPY
00101	001010	SHL
00110	001011	SHR
00111	001100	ADD
01000	001101	SUB
01001	001110	AND
01010	001111	OR
01011	010000	NOT
01100	010001	JAL

Figure 14.5 : Contenu du décodeur d'instructions

le deuxième cycle de toute façon), nous allons le faire quand même afin de respecter notre convention. Le contenu de la micro-mémoire est illustré par la figure 14.6.

Adresse	Contenu	Opération
000000	011010100000000	fetch
000001	100000000000000	NOP
000010	101010100010000	LDIMM
000011	001010101000000	LD
000100	101000000110000	LD
000101	001010101000000	ST
000110	000000010100000	ST
000111	001100010100000	ST
001000	100000010100000	ST
001001	100000000001000	COPY
001010	100000000001001	SHL
001011	100000000001010	SHR
001100	100000000001011	ADD
001101	100000000001100	SUB
001110	100000000001101	AND
001111	100000000001110	OR
010000	100000000001111	NOT
010001	001010101000000	JAL
010010	100001000100000	JAL

Figure 14.6 : Contenu de la micro-mémoire

14.3 RÉCAPITULATIF

Notre premier ordinateur est capable d'exécuter un programme simple, contenant des opérations de transfert de données entre la mémoire et les registres, des opérations arithmétiques et une opération pour modifier le contenu du compteur ordinal, permettant ainsi des boucles.

Chaque instruction est réalisée grâce à un micro-programme, invisible au programmeur de l'ordinateur. Ce micro-programme est stocké dans la micro-mémoire et exécuté au rythme de son propre compteur ordinal. Chaque instruction est ainsi exécutée en deux phases : le chargement et décodage de l'instruction, puis son exécution proprement dite par le micro-programme correspondant.

EXERCICES

Exercice 14.1. Sur le schéma du *premier ordinateur*, écrire le nombre de fils sur chaque groupe de fils.

Exercice 14.2. Quels éléments du *premier ordinateur* sont des circuits séquentiels ? Des circuits combinatoires ? Des circuits logiques à 3 états ?

Exercice 14.3.

(1) Quelles MOP ne doivent jamais valoir 1 simultanément, pour que soit respectée la règle *un bus de doit pas être connecté à 2 sorties convergentes, définies, contradictoires* ?

(2) La ligne 001000 de la micro-mémoire viole-t-elle ce principe ?

Exercice 14.4.

(1) Que fait la micro-instruction 000000 ?

(2) Que fait la micro-instruction 000001 ?

(3) Que fait la micro-instruction 000010 ?

Exercice 14.5. Écrire un micro-programme qui charge dans R0 le contenu de M[R1].

Chapitre **15**

Extensions du premier ordinateur

15.1 SAUTS CONDITIONNELS

Notre premier ordinateur (voir chapitre 14) est incapable d'effectuer des sauts conditionnels. Il ne dispose que d'une instruction JAL de saut non conditionnel. De fait, ce premier ordinateur n'a même pas les circuits nécessaires pour l'implémentation d'instructions de sauts conditionnels.

Tout d'abord, il faut un registre supplémentaire pour mémoriser des informations concernant la dernière opération effectuée par l'unité arithmétique et logique. Ce registre s'appelle *codes de conditions* (en anglais : *condition codes*, ou `CC`). Il contient 4 bits qui indiquent respectivement si la dernière opération a donné un résultat négatif `N`, un résultat nul `Z`, un résultat avec retenue `C` ou un autre avec débordement `V`. Le registre `CC` est piloté par une MOP qui, lorsqu'elle vaut 1, charge le registre avec les informations produites par l'unité arithmétique et logique. Nous aurions pu utiliser la même MOP que pour le chargement de R1, car ces opérations sont pour le moment utilisées simultanément. Plus tard, par contre, on pourra les piloter séparément afin, par exemple, de réaliser des opérations de comparaison. Un telle instruction est identique à une instruction de soustraction, mis à part que le résultat de la soustraction n'est pas conservé : seuls les codes de conditions le sont.

Il est nécessaire de piloter le signal ld du PC plus finement. Actuellement, ce signal vaut 1 si et seulement si la MOP produite par le saut non conditionnel vaut 1. Ce signal sera désormais produit par un circuit plus compliqué qui donne aussi la valeur 1 lorsque la micro-mémoire a produit une MOP pour une instruction *saut si zéro* et que le bit Z du registre CC vaut 1, etc. La nouvelle version de l'ordinateur est présentée figure 15.2.

Voici la table d'états du registre CC. Les signaux an, az, ac et av sont les sorties de l'unité arithmétique et logique :

ld	an	az	ac	av	N	Z	C	V	N'	Z'	C'	V'
0	-	-	-	-	a	b	c	d	a	b	c	d
1	a	b	c	d	-	-	-	-	a	b	c	d

Comme on peut le constater, c'est un registre ordinaire. Détaillons maintenant le contenu de la boîte chargée de calculer le signal ld de PC. C'est un circuit combinatoire à 9 entrées, ce qui donne une table de vérité assez grande. Il est plus simple de donner directement le circuit (figure 15.1).

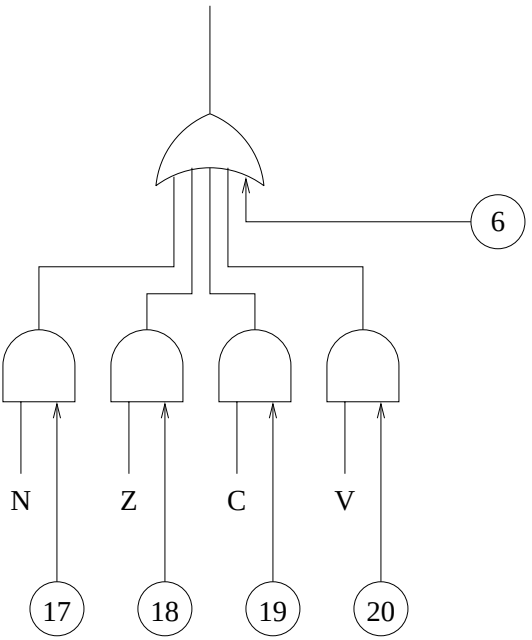


Figure 15.1 : Circuit calculant les codes conditions

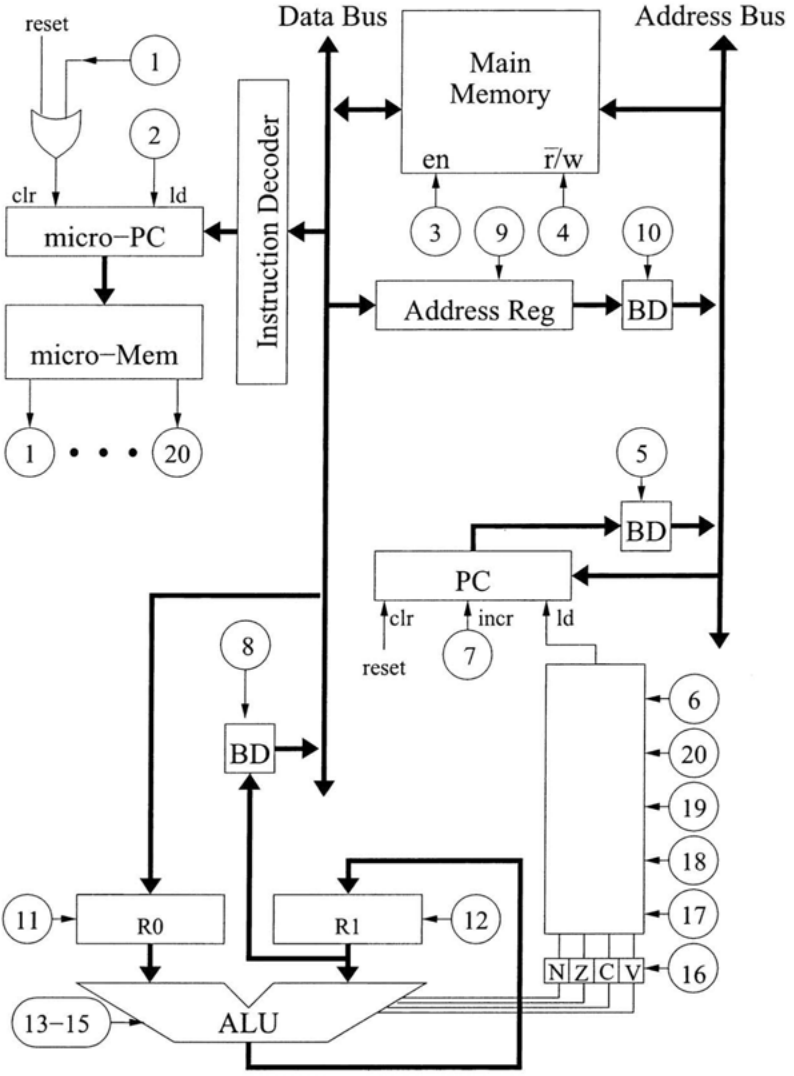


Figure 15.2 : Deuxième version de l'ordinateur avec prise en compte des sauts conditionnels

La sortie (le signal ld de PC) vaut 1 si MOP6 vaut 1, ou si simultanément le bit N et MOP17 valent 1, ou si simultanément le bit Z et MOP18 valent 1, ou si simultanément le bit C et MOP19 valent 1, ou encore si simultanément le bit V et MOP20 valent 1.

Comme MOP6 indique un saut non conditionnel, la génération de cette MOP donne automatiquement une valeur 1 au signal ld de PC, qui sera chargé avec la nouvelle adresse comme auparavant.

Pour exécuter une instruction comme JN *(jump if negative)*, il faut émettre MOP17. Si en même temps le bit N vaut 1 alors PC sera chargé avec la nouvelle valeur. Si par contre le bit N vaut 0, PC maintiendra son ancienne valeur, à savoir l'adresse du début de l'instruction située immédiatement après l'instruction JN. Le cours de l'exécution ne sera donc pas modifié. Le même raisonnement est valable pour les autres bits de CC.

Adresse	Contenu	Opération
000000	01101010000000000000	fetch
000001	10000000000000000000	NOP
000010	10101010001000000000	LDIMM
000011	00101010100000000000	LD
000100	10100000011000000000	LD
000101	00101010100000000000	ST
000110	00000001010000000000	ST
000111	00110001010000000000	ST
001000	10000001010000000000	ST
001001	10000000000100010000	COPY
001010	10000000000100110000	SHL
001011	10000000000101010000	SHR
001100	10000000000101110000	ADD
001101	10000000000110010000	SUB
001110	10000000000110110000	AND
001111	10000000000111010000	OR
010000	10000000000111110000	NOT
010001	00101010100000000000	JAL
010010	10000100010000000000	JAL

Figure 15.3 : Micro-mémoire 2

Maintenant, il faut déterminer le contenu de la micro-mémoire. D'abord, puisque nous avons fait passer le nombre de MOP de 15 à 20, la micro-mémoire doit être plus large. Des bits supplémentaires

sont donc ajoutés aux instructions existantes. Il faut simplement veiller à ce que MOP16 soit émise en même temps que MOP12 pour que le registre CC soit chargé avec R1. La micro-mémoire est présentée figure 15.3.

Maintenant, l'implémentation des différentes instructions de sauts conditionnels est identique à celle de l'instruction JAL à l'exception de MOP6 qui sera remplacée par MOP17 pour JN, 18 pour JN, 19 pour JV et 20 pour JC. Le nouveau contenu de la micro-mémoire est illustré par la figure 15.4.

Adresse	Contenu	Opération
000000	01101010000000000000	fetch
000001	10000000000000000000	NOP
000010	10101010001000000000	LDIMM
000011	00101010100000000000	LD
000100	10100000011000000000	LD
000101	00101010100000000000	ST
000110	00000001010000000000	ST
000111	00110001010000000000	ST
001000	10000001010000000000	ST
001001	10000000000100010000	COPY
001010	10000000000100110000	SHL
001011	10000000000101010000	SHR
001100	10000000000101110000	ADD
001101	10000000000110010000	SUB
001110	10000000000110110000	AND
001111	10000000000111010000	OR
010000	10000000000111110000	NOT
010001	00101010100000000000	JAL
010010	10000100010000000000	JAL
010011	00101010100000000000	JN
010100	10000000010000001000	JN
010101	00101010100000000000	JZ
010110	10000000010000000100	JZ
010111	00101010100000000000	JV
011000	10000000010000000010	JV
011001	00101010100000000000	JC
011010	10000000010000000001	JC

Figure 15.4 : Micro-mémoire 3

15.2 SOUS-PROGRAMMES

15.2.1 Appel d'un sous-programme

Pour l'instant, notre architecture n'est capable d'exécuter que des programmes simples avec des sauts conditionnels, ce qui correspond aux boucles des langages de programmation de haut niveau. Cependant ces langages ont beaucoup d'autres fonctionnalités. En particulier, ils sont capables de manipuler des *sous-programmes*, à savoir des fonctions ou des procédures.

Par exemple, on peut imaginer le sous-programme (C) suivant :

```
h()
{
  ...
  return ;
}
```

puis deux ou plusieurs appelants comme ceci :

```
f()
{
  h() ;
}
...
g()
{
  h() ;
}
```

Il n'est pas possible d'implémenter *l'appel* au sous-programme avec l'instruction JAL, car l'*adresse de retour* n'est pas mémorisée. Le sous-programme ne peut pas exécuter lui-même une autre instruction JAL à la fin de son exécution, car il ne sait pas où il doit poursuivre : à l'intérieur de la fonction `f` ou de la fonction `g`.

Des versions initiales de l'implémentation du langage Fortran tentaient de résoudre ce problème en stockant l'adresse de retour dans la cellule de mémoire qui précède le sous-programme même et en utilisant une instruction JIN (pour *jump indirect*) afin de retourner dans l'appelant. L'instruction JIN est similaire à JAL dans le sens où le saut effectué est non conditionnel. Cependant, l'argument de l'instruction JAL est l'adresse où poursuivre l'exécution, tandis que celui

de l'instruction JIN est l'adresse d'une cellule en mémoire contenant cette adresse. Avec une telle instruction, la traduction du sous-programme `h` est aisée :

```
h-1 :   0
h :     ...
        jin h-1
```

Ici, nous avons deux adresses, respectivement indiquées symboliquement par `h-1` et `h`. L'adresse `h-1` est simplement l'adresse `h` moins un. Par conséquent, les deux sont connues à la compilation. L'adresse `h-1` est utilisée pour stocker l'adresse de retour, alors que `h` est le point d'entrée du sous-programme. On remarque que le sous-programme, après avoir terminé son calcul, saute indirectement à l'adresse stockée dans la cellule dont l'adresse est `h-1`. Les appelants peuvent maintenant être implémentés comme indiqué figure 15.5.

```
f :     ...
        ldimm fhret
        copy
        st h-1
        ...
        jal h
fhret : ...
g :     ...
        ldimm ghret
        copy
        st h-1
        ...
        jal h
ghret : ...
```

Figure 15.5 : Implémentation des sous-programmes

Ici, nous avons introduit dans chaque appelant une adresse à laquelle le sous-programme appelé doit retourner après avoir terminé son calcul (ce sont les adresses `fhret` et `ghret`). Une telle adresse est utilisée comme argument de l'instruction LDIMM. L'adresse est ensuite stockée dans la cellule réservée à cet effet au sein du sous-programme appelé. Finalement, à l'aide de l'instruction JAL, un saut est effectué au début du sous-programme.

Nous avons maintenant un *protocole d'appel de sous-programme* rudimentaire, à savoir un ensemble de règles pour déterminer exactement la responsabilité d'un appelant et d'un appelé afin que le mécanisme des sous-programmes fonctionne.

Notre protocole dépend de l'existence de l'instruction JIN, mais celle-ci n'existe pas encore dans notre architecture. Or, elle est relativement facile à implémenter sans aucune modification de notre l'architecture actuelle. Le plan pour son implémentation est le suivant : d'abord émettre les MOP 5, 3 et 9 pour adresser la mémoire centrale, stocker l'argument de l'instruction dans le registre d'adresse et incrémenter PC. Ensuite, au cycle suivant, utiliser le contenu du registre d'adresse pour adresser la mémoire (MOP 10 et 3), obtenir l'adresse où poursuivre l'exécution et la stocker dans le registre d'adresse (MOP9 encore). Pendant le cycle final, stocker le contenu du registre d'adresse dans PC (MOP 10 et 6). Voici le micro-programme complet :

...	`00101010100000000000`	(JIN)
...	`00100000110000000000`	(JIN)
...	`10000100010000000000`	(JIN)

Nous avons omis les adresses exactes dans la micro-mémoire de ce programme, ainsi que le code de l'instruction, car ils sont déterminés comme d'habitude (les premiers disponibles).

La solution du problème des sous-programmes présentée ci-dessus comporte un inconvénient majeur. L'appelant doit utiliser les registres R0 et R1 afin de charger et stocker l'adresse de retour. On peut faire mieux si on a une instruction spéciale pour l'appel à un sous-programme. La plupart des architectures proposent une telle instruction (souvent appelée JSR pour *jump to subroutine*). Celle-ci s'occupe à la fois de stocker l'adresse de retour (le contenu courant de PC) et de sauter de façon non conditionnelle au point d'entrée du sous-programme. Avec une telle instruction, nos sous-programmes peuvent être implémentés comme indiqué figure 15.6.

On ne peut pas implémenter l'instruction JSR dans l'architecture actuelle car il est nécessaire de stocker le contenu de PC dans une cellule en mémoire. Malheureusement, il n'y a pas de chemin de données entre PC et le bus de données. Afin d'implémenter JSR, il nous faut un tel chemin.

```
h-1 :   0
h :     ...
        jin h-1

f :     ...
        jsr h-1
        ...

g :     ...
        jsr h-1
        ...
```

Figure 15.6 : Implémentation des sous-programmes

Voici une description complète de ce que doit faire JSR :

– stocker la valeur de PC en mémoire à l'adresse donnée en argument à l'instruction JSR (`h-1` dans l'exemple ci-dessus),

– charger l'adresse donnée *plus un* dans PC (`h` dans notre exemple).

Afin d'implémenter JSR, nous ajoutons un registre que nous appelons le *registre de données* (*data register*, pour être homogène avec le registre d'adresse). L'architecture modifiée est présentée figure 15.7.

On remarque les nouvelles MOP 21 et 22 pour charger le nouveau registre et transmettre son contenu sur le bus d'adresse. Voici le plan pour l'implémentation de JSR :

– pendant le premier cycle d'horloge, charger l'argument de l'instruction JSR dans le registre d'adresse et incrémenter PC,

– stocker le contenu de PC dans le registre de données,

– pendant les cycles 3 à 5, écrire le contenu du registre de données en mémoire en utilisant l'adresse contenue dans le registre d'adresse,

– puis, stocker le contenu du registre d'adresse dans PC,

– finalement, incrémenter PC pour obtenir l'argument *plus un*.

Voici le micro-programme :

...	00101010100000000000	(JSR)
...	00001000000000000010	(JSR)
...	00000000010000000001	(JSR)
...	00110000010000000001	(JSR)
...	00000100010000000001	(JSR)
...	10000010000000000000	(JSR)

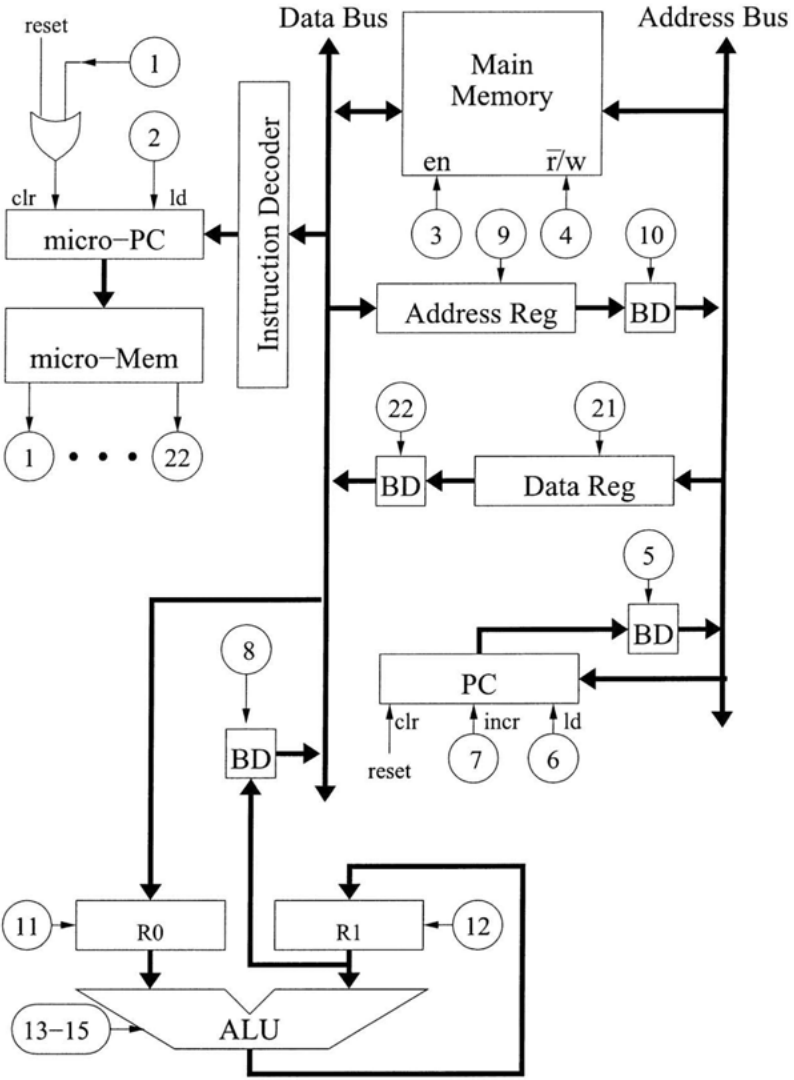


Figure 15.7 : Nouvelle version avec prise en compte de l'instruction JSR

15.2.2 Récursivité

Le protocole d'appel précédent ne permet pas de sous-programmes récursifs, à savoir des sous-programmes qui s'appellent directement ou indirectement. La raison est simple : la deuxième fois qu'un sous-programme est appelé, l'adresse de retour stockée est écrasée et perdue pour toujours. Pour un langage comme Fortran, cela ne pose pas de problème, car la récursivité n'est pas autorisée par le langage. Cependant la plupart de langages modernes autorisent la récursivité. Comment peut-on corriger l'architecture actuelle pour autoriser la récursivité ? Nous avons besoin d'une *pile* d'exécution.

Une pile est une suite de cellules en mémoire dont l'adresse d'une extrémité est stockée dans un registre appelé *pointeur de pile* (en anglais : *stack pointer*). Ce côté-là est appelé le *sommet* (en anglais : *top*) de la pile. Quatre conventions différentes mais équivalentes sont utilisées selon que le sommet de la pile se trouve à l'adresse la plus haute ou la plus basse en mémoire, et selon que le pointeur de pile indique la dernière adresse utilisée ou la première adresse libre. Ici, nous utilisons la convention utilisée dans la plupart des systèmes Unix : le sommet est à l'adresse la plus basse et le pointeur de pile indique la dernière adresse utilisée.

Les modifications à apporter à notre architecture pour créer une pile sont présentées figure 15.9.

Le pointeur de pile peut être mis à zéro, incrémenté et décrémenté. Remettre le pointeur de pile à zéro par le bouton reset est justifié, car on peut voir la mémoire comme étant circulaire. L'adresse 0 de la mémoire peut être considérée comme étant positionnée au-delà de la fin de la mémoire. Trois nouvelles MOP sont nécessaires : 23 pour l'incrémentation, 24 pour la décrémentation et 25 pour transmettre le contenu du pointeur de pile au bus d'adresse.

Avec cette nouvelle architecture, il est possible de donner une nouvelle définition de l'instruction JSR et d'ajouter une nouvelle instruction RET (pour *return*). L'implémentation de nos sous-programmes avec ces nouvelles instructions est donnée par la figure 15.8.

Voici la nouvelle description de l'instruction JSR modifiée :

– empiler PC sur la pile,
– charger l'adresse donnée en argument de l'instruction JSR dans PC.

```
h :   ...
      ret

f :   ...
      jsr h
      ...

g :   ...
      jsr h
      ...
```

Figure 15.8 : Implémentation des sous-programmes

Et voici celle de l'instruction RET :

- charger le sommet de la pile dans PC,
- dépiler la pile.

Voici le plan pour l'implémentation de l'instruction JSR :

- pendant le premier cycle, charger l'argument de l'instruction dans le registre d'adresse et incrémenter PC,
- puis charger le contenu de PC dans le registre de données. Décrémenter simultanément le pointeur de pile,
- pendant les cycles 3 à 5, écrire en mémoire le contenu du registre de données à l'adresse indiquée par le pointeur de pile,
- finalement, charger le contenu du registre d'adresse dans PC.

Voici le micro-programme :

...	00101010100000000000000000	(JSR)
...	00001000000000000000010010	(JSR)
...	00000000000000000000001001	(JSR)
...	00110000000000000000001001	(JSR)
...	10000100010000000000001001	(JSR)

Voici le plan pour l'implémentation de l'instruction RET :

- pendant le premier cycle, utiliser le pointeur de pile pour adresser la mémoire et stocker le sommet de la pile dans le registre d'adresse,
- pendant le deuxième et dernier cycles, charger le contenu du registre d'adresse dans PC et incrémenter le pointeur de pile.

Voici le micro-programme pour RET :

...	00100000100000000000000001	(RET)
...	10000100010000000000000100	(RET)

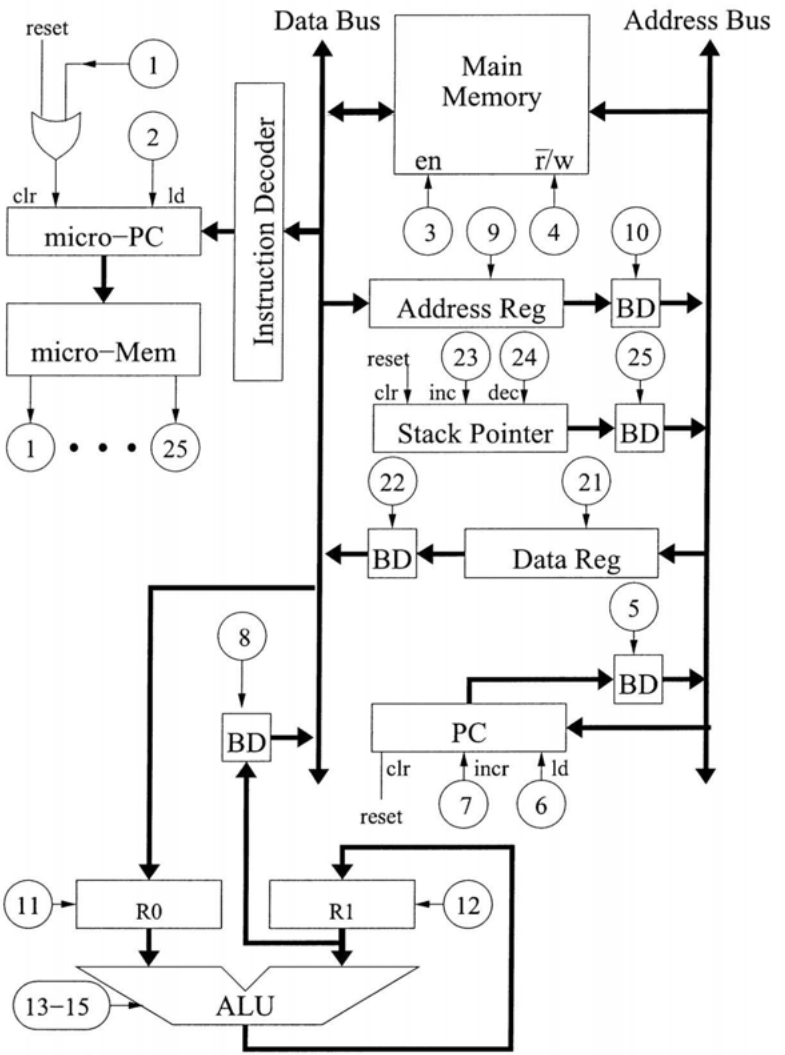


Figure 15.9 : Version avec pile

15.2.3 Passage de paramètres

Dans la section 15.2.2, nous avons vu comment utiliser une pile pour stocker l'adresse de retour d'une procédure ou d'une fonction. Dans cette section, nous allons étendre l'utilisation de la pile pour le *passage de paramètres* aux sous-programmes.

Pour cela, nous avons besoin de deux instructions supplémentaires : PUSH et POP. L'instruction PUSH empile la valeur de R1 au sommet de la pile ; l'instruction POP dépile ce dernier et le met dans R0. Ces deux instructions peuvent être implémentées sans modification de notre architecture. Voici le micro-programme pour PUSH :

...	`00000000000000000000000010`	(PUSH)
...	`00000001000000000000000001`	(PUSH)
...	`00110001000000000000000001`	(PUSH)
...	`10000001000000000000000001`	(PUSH)

Voici celui de l'instruction POP :

...	`10100000001000000000000101`	(POP)

Avec ces deux instructions, il est possible de créer un protocole d'appel plus complet contenant aussi des règles de passage de paramètres. Voici un exemple d'un tel protocole. Il empile les arguments dans l'ordre inverse (argument n en premier, argument 1 en dernier). C'est souvent le cas pour le langage C, car cela permet à un sous-programme de toujours trouver le premier argument sans connaître le nombre d'arguments passés.

Appelant	Appelé
calculer l'argument n dans R1 empiler R1 ... calculer l'argument 1 dans R1 empiler R1 jsr Appelé	
	calculer la valeur de retour dans R1 ret
dépiler n arguments de la pile	

a) Accès aux arguments

Bien que cette méthode de passage de paramètres fonctionne, il n'est pas très pratique pour l'appelé d'accéder à ces paramètres. La seule possibilité actuelle est d'utiliser l'instruction POP. Il serait plus facile d'avoir une instruction capable d'utiliser une adresse composée du contenu du pointeur de pile plus une petite constante. Avec une telle instruction, il serait possible d'accéder à l'argument numéro i en utilisant l'adresse SP+i, où SP est le contenu du pointeur de pile.

Il faut donc un moyen d'additionner une constante au pointeur de pile. La constante sera un argument de la nouvelle instruction. Afin d'obtenir le résultat souhaité, nous pouvons stocker la constante dans le registre d'adresse et rajouter des circuits pour additionner le contenu du registre d'adresse et le pointeur de pile pour former une adresse destinée à la mémoire centrale. La modification de la figure 15.10 illustre une solution.

On remarque la nouvelle MOP numéro 26 pour transmettre la somme au bus d'adresse.

Maintenant, il est possible d'écrire une nouvelle instruction LDS *(load from stack)* qui fait le nécessaire et dont voici le micro-programme :

...	00101010100000000000000000	(LDS)
...	10100000001000000000000001	(LDS)

b) Empilement des arguments

De même que l'instruction LDS facilite l'accès aux paramètres pour l'appelé, il serait commode pour l'appelant de pouvoir placer en une seule instruction le paramètre numéro i à l'adresse SP+i. Nous appelons cette instruction STS *(STore to Stack)*. Aucune modification autre que celles apportées par l'implémentation de LDS n'est nécessaire pour implémenter STS. Comme pour LDS, nous utilisons le registre d'adresse afin de stocker la constante i. Voici le micro-programme correspondant :

...	00101010100000000000000000	(STS)
...	00000001000000000000000001	(STS)
...	00110001000000000000000001	(STS)
...	10000001000000000000000001	(STS)

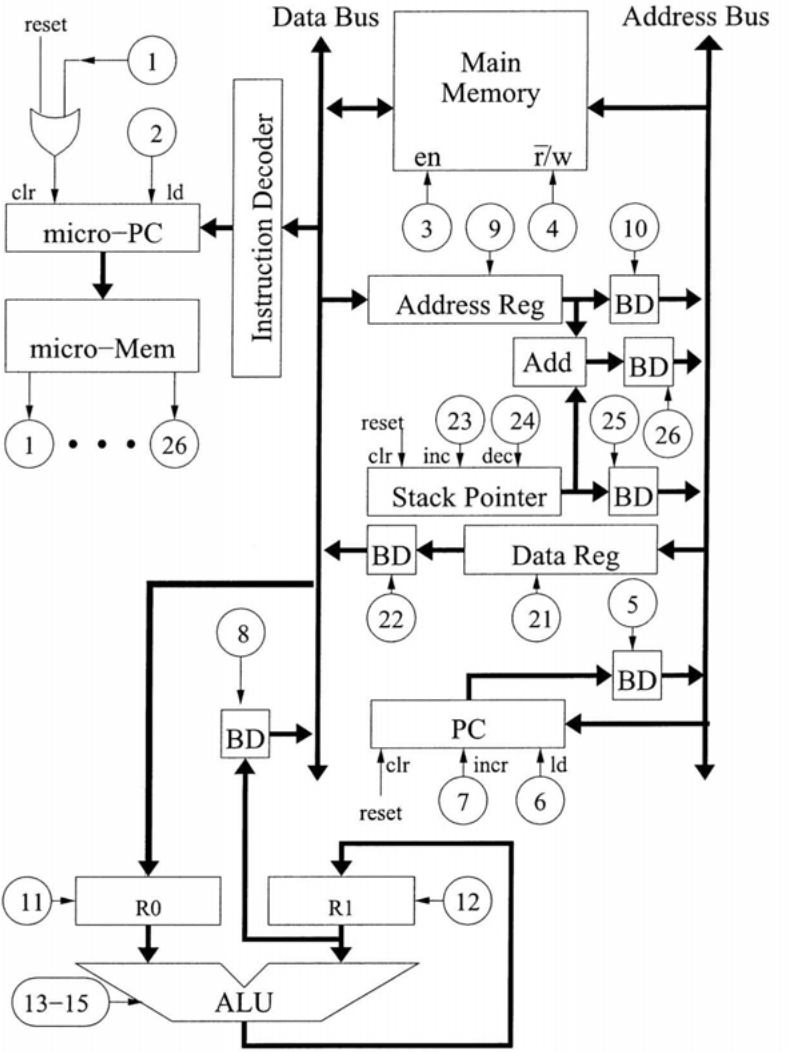


Figure 15.10 : Accès aux arguments

c) Dépilement des arguments

Après le retour de l'appelé, l'appelant doit dépiler les arguments qu'il a empilés avant l'appel. Actuellement, la seule possibilité est d'émettre autant d'instructions POP que d'arguments empilés. On peut améliorer avec la possibilité d'additionner une constante au pointeur de pile. Pour cela, nous pouvons nous servir de l'additionneur existant, mais il va falloir ajouter des circuits pour stocker le résultat de l'addition dans le pointeur de pile. Cela nécessite des lignes supplémentaires, ainsi qu'une micro-opération permettant au pointeur de pile d'être chargé depuis des lignes extérieures. Voir figure 15.11 pour ces modifications.

On remarque MOP numéro 27 pour charger le pointeur de pile. Il est maintenant possible d'écrire une instruction ADDSP pour additionner une constante à la valeur contenue dans le pointeur de pile. Voici son implémentation :

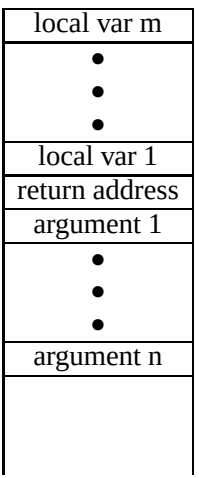
...	00101010100000000000000000	(ADDSP)
...	10000000000000000000000001	(ADDSP)

15.2.4 Variables locales

Dans la sous-section 15.2.3 sur le passage de paramètres, nous avons vu comment l'appelant transmet des paramètres à l'appelé, et comment l'appelé accède à ces paramètres durant son exécution.

Afin que notre protocole d'appel soit complet, il faut aussi montrer comment allouer des variables locales à un sous-programme particulier. Ces variables sont allouées sur la pile. Heureusement, nous avons déjà le support nécessaire dans notre architecture pour les variables locales.

Nous utilisons la disposition de pile ci-contre :



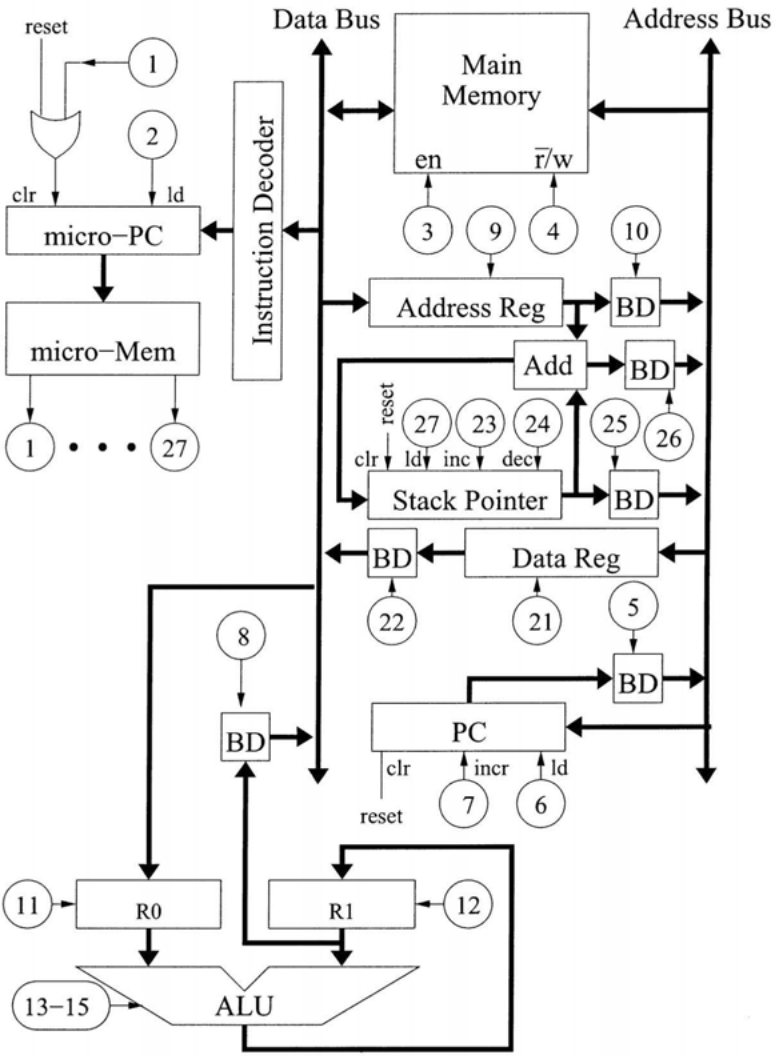


Figure 15.11 : Dépiler les arguments

L'allocation de variables locales doit maintenant être intégrée dans le protocole d'appel de la manière suivante :

Appelant	Appelé
calculer l'argument n dans R1 empiler R1 ... calculer l'argument 1 dans R1 empiler R1 jsr Appelé	
	addsp -m (pour allouer m variables) calculer la valeur de retour dans R1 addsp m (pour désallouer m variables) ret
addsp n (pour dépiler les arguments)	

15.3 RÉCAPITULATIF

Afin d'être réellement utilisable, notre ordinateur doit être doté d'instructions supplémentaires, en particulier pour réaliser des sauts conditionnels, l'appel à des sous-programmes, etc. La structure de base de l'ordinateur permet d'ajouter facilement les circuits nécessaires pour ces extensions.

EXERCICES

Exercice 15.1. Voici un sous-programme en code *assembleur* ; pour plus de lisibilité, on a utilisé les noms des instructions (plutôt que leur codes) et des noms symboliques pour les adresses (plutôt que des valeurs numériques).

On suppose l'existence l'un sous-programme à l'adresse (symbolique) `mult` qui, étant donnés deux arguments sur la pile, calcule leur produit dans R1.

En supposant que la première instruction du sous-programme `mystere` se trouve à l'adresse 40 (en base 10), et celle du sous-programme `mult` à l'adresse 150 (en base 10), quel est le contenu

de la mémoire (sous la forme de suite d'octets) entre les adresses 40 et 70 ? Les valeurs numériques des codes de toutes les instructions sont indiquées dans l'annexe A.2.

```
mystere: lds 2
         copy
         jz out
         ldimm -1
         add
         push
         lds 2
         copy
         push
         jsr mystere
         addsp 2
         push
         lds 2
         copy
         push
         jsr mult
         addsp 2
         ret
out:     ldimm 1
         copy
         ret
```

Exercice 15.2. Certains langages de programmation manipulent des données typées, à savoir qu'il est possible de déterminer en regardant une donnée s'il s'agit d'un nombre, d'un caractère, d'un pointeur ou d'autre chose. On parle de langage dynamiquement typé. Pour aider le concepteur du compilateur d'un tel langage, on souhaite lui proposer un ensemble d'instructions supplémentaires permettant des opérations arithmétiques entre deux opérandes ainsi typés. Pour cela, on représente un petit entier i traditionnellement à l'aide de $2i$. L'entier 13 sera donc représenté par 00011010, à savoir la représentation habituelle de 26. De cette manière, chaque petit entier a toujours le bit le moins significatif égal à 0. Si le bit le moins significatif n'est pas égal à 0, il s'agit d'un autre type de donnée et toute tentative d'effectuer une opération arithmétique là-dessus doit déclencher une erreur. On remarque que les opérations arithmétiques habituelles (addition et

soustraction) fonctionnent également sur ces données typées. Aucune modification de l'unité arithmétique et logique n'est donc nécessaire.

(1) Modifier l'architecture illustrée par la figure 15.2 pour permettre une telle famille d'instructions. Ces instructions diffèrent des opérations arithmétiques habituelles par le fait qu'elles acceptent un argument correspondant à une adresse dans le programme. Si les arguments sont du bon type, l'instruction effectue l'opération sans se servir de l'adresse. Si par contre une erreur de typage est détectée, l'instruction fonctionne comme une instruction de saut conditionnel dans le sens où elle provoque un saut à l'adresse donnée en argument (l'opération arithmétique est éventuellement effectuée quand même).

(2) Donner le micro-programme des deux instructions ADDT (addition typée) et SUBT (soustraction typée).

(3) En supposant qu'une fonction `error` existe déjà (et qu'elle peut donc être appelée comme une fonction normale), donner le code généré par un compilateur pour un langage dynamiquement typé à partir d'une expression comme `z = x + y`.

Chapitre **16**

Entrées/sorties et interruptions

16.1 ENTRÉES ET SORTIES

L'ordinateur présenté dans les chapitres précédents est capable d'effectuer des calculs arithmétiques ; il permet la structuration d'un programme en plusieurs sous-programmes, en boucles, etc. Par contre, il est incapable de communiquer avec l'extérieur.

Le monde extérieur se présente sous la forme d'un certain nombre de périphériques. Il peut s'agir de claviers, de disques, d'écrans, d'imprimantes, etc. Mais comment connecter ces périphériques à l'ordinateur ? La solution la plus simple est de prétendre que les périphériques forment une partie de la mémoire centrale.

Comme la mémoire centrale, les périphériques vont correspondre à un certain nombre d'adresses en mémoire, et avoir des cellules données à lire ou à écrire. Mais au lieu de stocker seulement la valeur, le périphérique va aussi effectuer une action. De même, une lecture à une telle adresse ne récupère pas simplement la dernière valeur écrite, mais permet à une nouvelle valeur d'être transmise de l'extérieur vers l'ordinateur (il peur s'agir de la touche sur laquelle l'utilisateur a appuyé).

Notre première tâche est de construire un circuit qui se comporte comme une mémoire vis-à-vis de l'ordinateur et qui puisse être utilisé pour communiquer avec l'extérieur.

Commençons par la sortie d'une valeur vers l'extérieur. Pour cela, il suffit de commencer avec une cellule de mémoire (un mot d'un seul bit) et de supprimer la partie permettant la lecture.

On se reportera à la figure 12.1 de la page 92 pour la structure générale d'une telle cellule.

Après avoir enlevé les circuits spécifiques à la lecture, nous obtenons le schéma indiqué dans la figure 16.1.

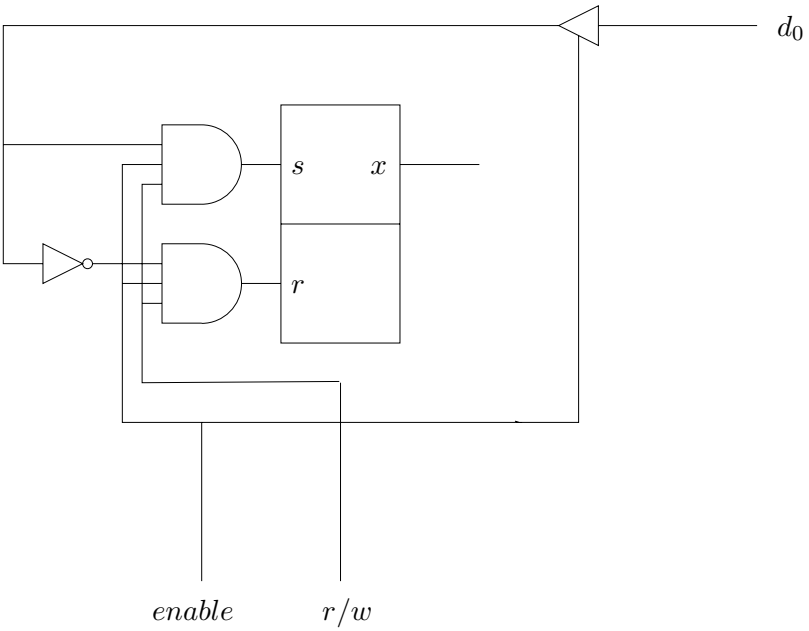


Figure 16.1 : Circuit de sortie

Une tentative de lecture à une adresse contenant ce type de cellule ne donne aucune valeur utile sur le bus de données. Une écriture, par contre, stocke la valeur écrite dans la bascule SR. La sortie de cette bascule (indiquée par x dans la figure) reflète la valeur écrite et pourra servir à piloter des périphériques.

La même technique que celle utilisée pour la mémoire peut être employée afin d'augmenter le nombre de bits de sortie. On peut, par exemple, mettre huit cellules de ce type en parallèle afin d'obtenir un octet de sortie comme l'indique la figure 16.2.

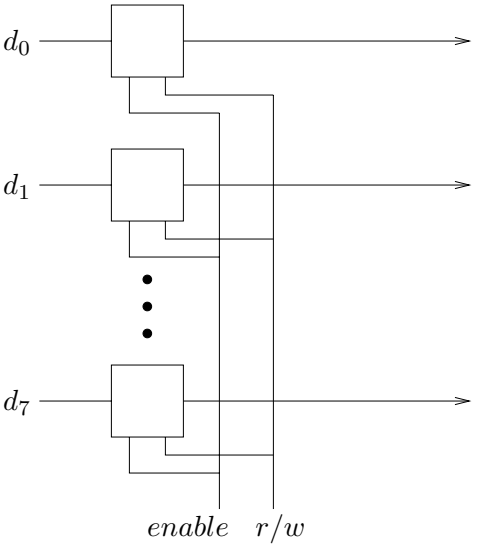


Figure 16.2 : Un octet de sortie

Pour qu'on puisse brancher le circuit sur les bus, il faut lui choisir une adresse, puis fabriquer un *décodeur d'adresses* comme indiqué par la figure 16.3. Ce décodeur est branché sur le bus d'adresse et détecte l'adresse unique qui lui est attribuée, dans notre cas `10011011`.

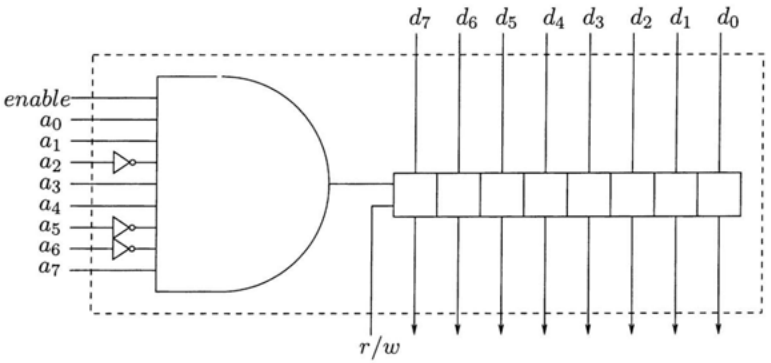


Figure 16.3 : Décodeur d'adresses

Un exemple d'utilisation de ce type de périphérique simple serait un tableau d'affichage de chiffres comme présenté figure 16.4.

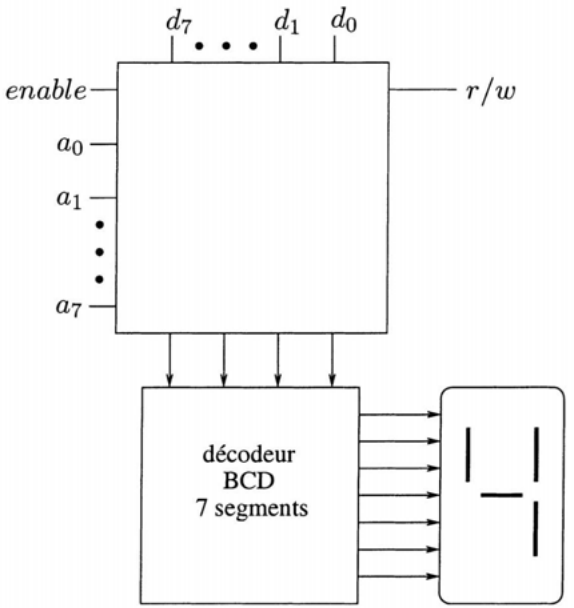


Figure 16.4 : Exemple de périphérique de sortie

L'entrée n'est pas plus compliquée. La figure 16.5 montre une cellule de mémoire d'un mot d'un seul bit, privée des circuits permettant l'écriture.

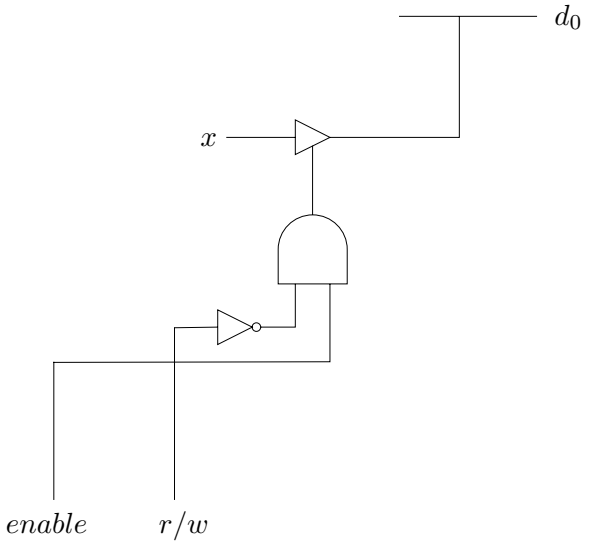


Figure 16.5 : Circuit d'entrée

De même que pour la sortie, plusieurs cellules de ce type peuvent être connectées en parallèle afin de permettre l'entrée d'un octet à la fois. La figure 16.6 illustre l'utilisation de ce type de circuit. Il s'agit d'un ensemble de boutons d'un magnétoscope, ou appareil similaire, permettant à l'ordinateur de détecter une action de la part de l'utilisateur.

16.2 INTERRUPTIONS

Les circuits présentés dans la section précédente permettent à un programme de sortir une valeur vers l'extérieur relativement facilement. L'entrée est en fait plus compliquée : le programme doit constamment vérifier si une entrée est disponible. Ce style d'entrée s'appelle *polling* et il est relativement inefficace, surtout si la plupart du temps aucune entrée n'est effectivement disponible.

Une meilleure technique est de permettre aux périphériques d'*interrompre* l'exécution actuelle du programme lorsqu'une entrée

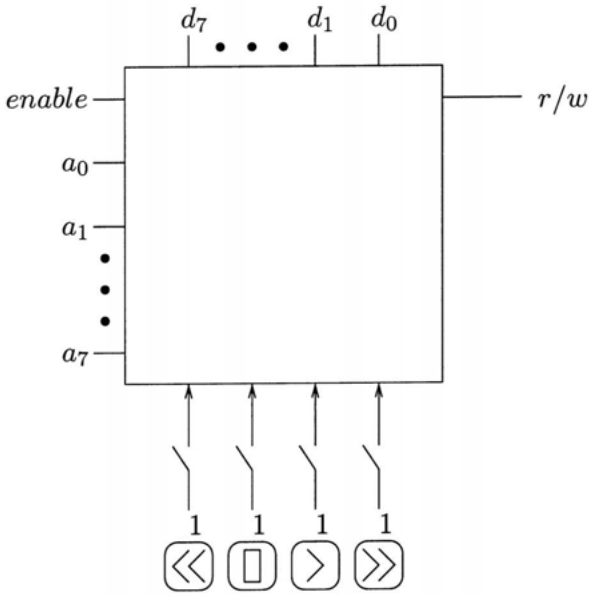


Figure 16.6 : Exemple de périphérique d'entrée

est présente. Le programme s'exécute normalement et, lorsque une entrée est présente, un appel *forcé* est fait vers un sous-programme que l'on appelle habituellement *traitant d'interruptions*.

Le traitant doit trouver la raison de l'interruption (clavier, disque, etc) et traiter la valeur devenue disponible. Une fois ce travail terminé, le traitant exécute une instruction RET afin de revenir à l'exécution du programme normal.

La figure 16.7 montre les modifications à apporter à notre architecture pour la prise en compte des interruptions.

On remarque les deux entrées supplémentaires de micro-PC. Ce circuit se comporte désormais de la manière suivante :

clr_1	clr_2	int	ld	$Action$
1	-	-	-	remise à zéro
0	0	0	0	incrémentation
0	0	0	1	chargement
0	0	1	0	incrémentation
0	0	1	1	chargement
0	1	0	0	remise à zéro
0	1	0	1	ne peut pas arriver
0	1	1	0	chargement d'une constante c
0	1	1	1	ne peut pas arriver

L'entrée clr_1 provoque une remise à zéro sans condition. C'est le signal utilisé par *reset*. La nouvelle entrée clr_2 provoque une remise à zéro uniquement si le nouveau signal int (pour interruption) vaut 0. Le signal clr_2 est désormais utilisé par les micro-programmes pour commencer le chargement de l'instruction suivante.

Lorsqu'une interruption a eu lieu pendant l'exécution du micro-programme d'une instruction, la sortie de la bascule SR vaut 1. Ceci n'a aucun effet sur micro-PC tant que la valeur du signal clr_2 vaut 0. À la fin du micro-programme, lorsque MOP1 vaut 1, une valeur de 1 du signal int provoque le chargement d'une constante dans micro-PC. Cette constante correspond à l'adresse du micro-programme d'une instruction spéciale permettant d'interrompre l'exécution actuelle et de provoquer un saut à un sous-programme indiqué par l'adresse A à l'entrée du pilote de bus contrôlé par MOP28.

Reste à écrire le micro-programme permettant de sauvegarder le contenu de PC sur la pile, puis de sauter à l'adresse A, comme si une

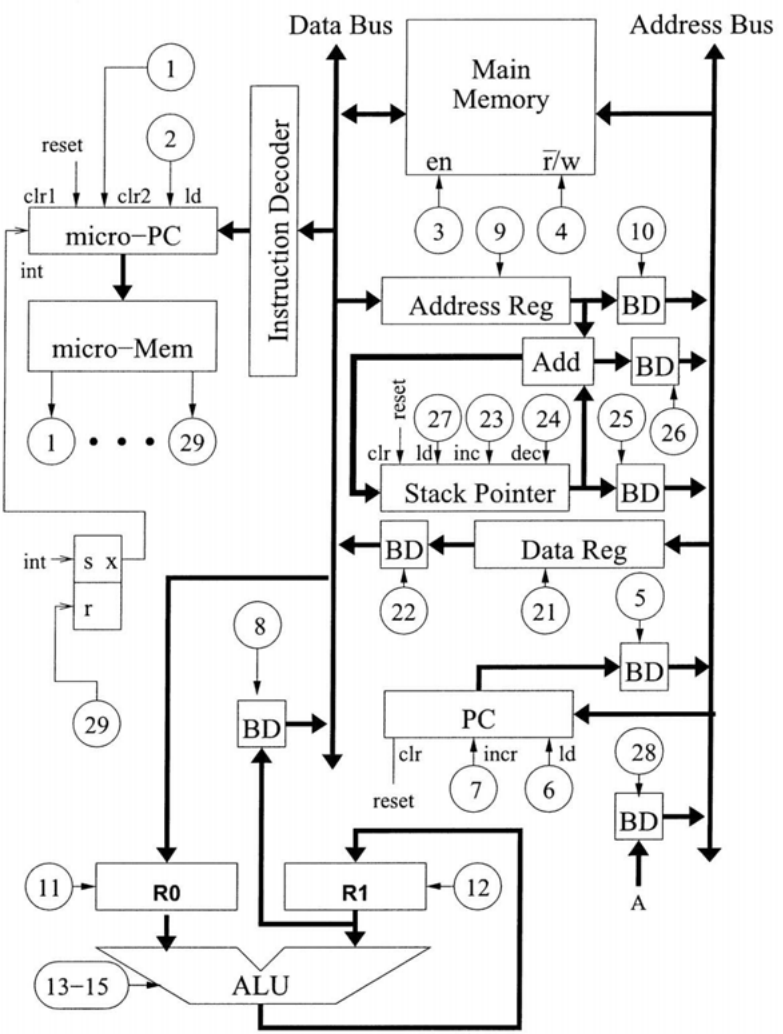


Figure 16.7 : Modifications pour les interruptions

instruction JSR A avait été exécutée. Voici le micro-programme (il commence à l'adresse C chargée dans micro-PC) :

...	00001000000000000000100100001	(INT)
...	00010000000000000000010010000	(INT)
...	00110000000000000000010010000	(INT)
...	00010000000000000000010010000	(INT)
...	10000100000000000000000000010	(INT)

16.3 RÉCAPITULATIF

L'ordinateur doit pouvoir exécuter des programmes, mais il doit aussi être capable de communiquer avec le monde extérieur. La façon la plus simple de permettre cette communication est probablement de construire des unités périphériques dont le comportement vu par l'ordinateur est identique à celui d'une mémoire. De cette manière, les instructions de transfert de données entre les registres et la mémoire peuvent également être utilisées pour le transfert de données entre le monde extérieur et les registres.

Deux méthodes permettent de transmettre des données externes vers le processeur. Ce dernier peut régulièrement exécuter des instructions pour tester si une donnée est prête, mais ceci est assez inefficace. La solution la plus courante dans les processeurs modernes consiste à permettre au monde extérieur d'interrompre l'exécution normale du processeur afin de signaler la présence d'une donnée.

EXERCICES

Exercice 16.1. Le circuit de la figure 16.1 n'étant plus bidirectionnel (car la partie entrée a été enlevée), il n'est pas nécessaire d'employer un circuit à trois états. Construire une nouvelle version sans circuit à trois états.

Exercice 16.2. La modification pour les interruptions introduite dans ce chapitre laisse le signal d'interruption en suspend en attendant la fin de l'exécution du micro-programme en cours d'exécution. Pourquoi ?

PARTIE 3

SUJETS AVANCÉS

Chapitre **17**

Mémoire Cache

On peut construire une mémoire centrale accessible en un cycle d'horloge, mais celle-ci est trop coûteuse lorsqu'elle est de taille raisonnable.

Il s'agit là d'un principe très général : le coût d'une technologie augmente avec sa qualité (ici sa vitesse).

À taille égale, une mémoire plus lente est donc (souvent considérablement) moins coûteuse. Le principe de la *mémoire cache* ou simplement *cache* permet d'obtenir la plupart des avantages des deux technologies : rapidité et faible coût.

Ce principe est simple : pour des raisons de coût, la mémoire centrale est relativement lente. Une mémoire plus rapide, mais beaucoup plus petite (donc de coût raisonnable) est utilisée pour stocker les données utilisées le plus fréquemment. De cette manière, la plupart du temps, la donnée recherchée se trouve dans la mémoire rapide. Lorsqu'une donnée doit être recherchée dans la mémoire lente, elle est écrite dans la mémoire rapide pour des références ultérieures. On obtient alors une rapidité proche de celle de la mémoire rapide pour un coût généralement associé à la mémoire lente.

Pour que le scénario indiqué ci-dessus fonctionne, l'accès aux données ne doit pas être aléatoire. Heureusement, on sait par expérience

qu'un programme accède à ses données (et à son code) d'une manière relativement prévisible. La probabilité qu'un accès à une adresse a soit suivi d'un accès à une adresse proche ou égale à a est très élevée. C'est le principe de *localité*, qui rend possible la technologie de la mémoire cache. Ceci s'explique principalement par le fait qu'un programme est linéaire avec des boucles et que les données sont le plus souvent organisées en tableaux et structures.

En général, un accès à la mémoire centrale prend entre 5 et 10 cycles d'horloge. Afin d'être rentable, le cache doit fournir une donnée en un cycle d'horloge.

Il est également possible d'avoir plusieurs niveaux de cache, chacun avec un coût et une rapidité différents. Ici, nous allons nous restreindre à deux niveaux : ceux du cache et de la mémoire centrale.

La figure 17.1 montre la position conceptuelle du cache entre le processeur et la mémoire centrale.

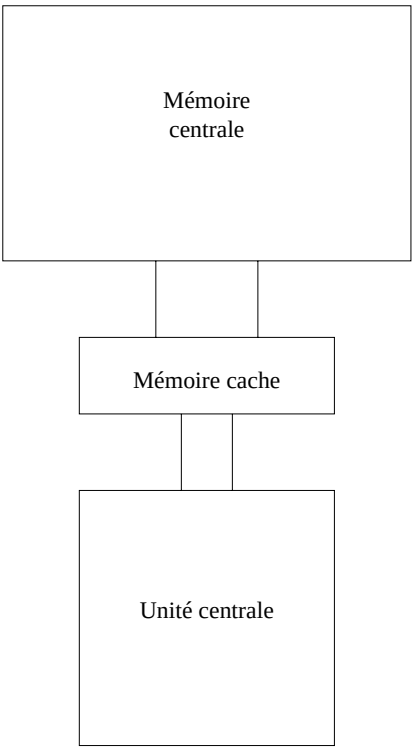


Figure 17.1 :
Mémoire cache

Le cache peut être organisé de plusieurs façons. Le cache *associatif*, qui permet de stocker une donnée n'importe où dans le cache, est le plus général et le plus simple à comprendre, mais aussi le plus coûteux à mettre en œuvre. Le cache *direct* (ou *à correspondance directe*), dans lequel chaque donnée doit être stockée à un endroit précis, représente l'autre bout du spectre. Des solutions intermédiaires sont possibles.

17.1 CACHE ASSOCIATIF

La mémoire centrale est divisée en *blocs* dont la taille est une puissance de 2, souvent 16 octets soit 128 bits. Les données sont transmises entre la mémoire centrale et le cache par blocs. Ceci augmente le *débit*, à savoir la quantité de données accessible par unité de temps. Entre le cache et le processeur, l'accès se fait comme d'habitude mot par mot, souvent de 4 octets. Les blocs ne se chevauchent pas, chacun d'entre eux commence donc à une adresse multiple de sa taille.

Une adresse arbitraire (par exemple sur 32 bits) est donc abstraitement divisée en deux parties : la première indiquant le numéro de bloc et la seconde indiquant le numéro d'octet à l'intérieur du bloc. Pour une taille de bloc de 16 octets et une adresse de 32 bits, la première partie de l'adresse contient donc 28 bits et la deuxième 4 bits.

Un cache associatif est divisé en *lignes*, chacune d'entre elles contenant une copie d'un bloc de la mémoire centrale. La figure 17.2 montre l'organisation conceptuelle du cache associatif. Comme on peut le constater, les lignes du cache ne sont pas dans un ordre particulier. Par conséquent, chaque ligne doit être marquée par le bloc qu'elle stocke.

Lorsque le processeur souhaite accéder à une donnée, le numéro de bloc de l'adresse utilisée pour l'accès est comparé *simultanément* à l'ensemble des numéros de blocs stockés dans les lignes de cache. Un exemple de scénario est indiqué dans la figure 17.3.

Ici, le processeur souhaite accéder à l'octet numéro 7 du bloc numéro 1 534. Le numéro de bloc est comparé simultanément à tous ceux stockés dans les lignes du cache. Dans le cas présent, le bloc recherché se trouve dans le cache et l'octet numéro 7 de ce bloc est envoyé à l'unité centrale.

Figure 17.2 :
Cache associatif
(ci-contre)

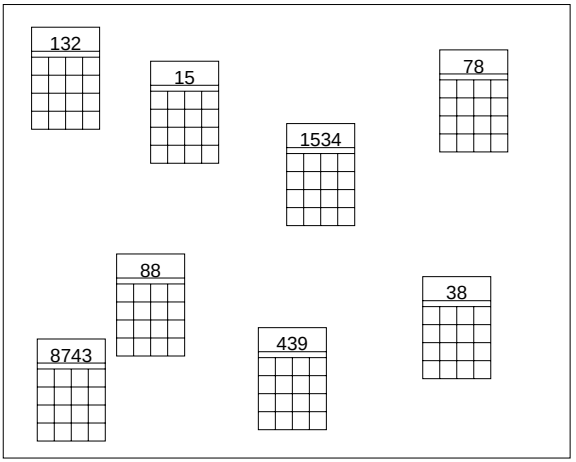
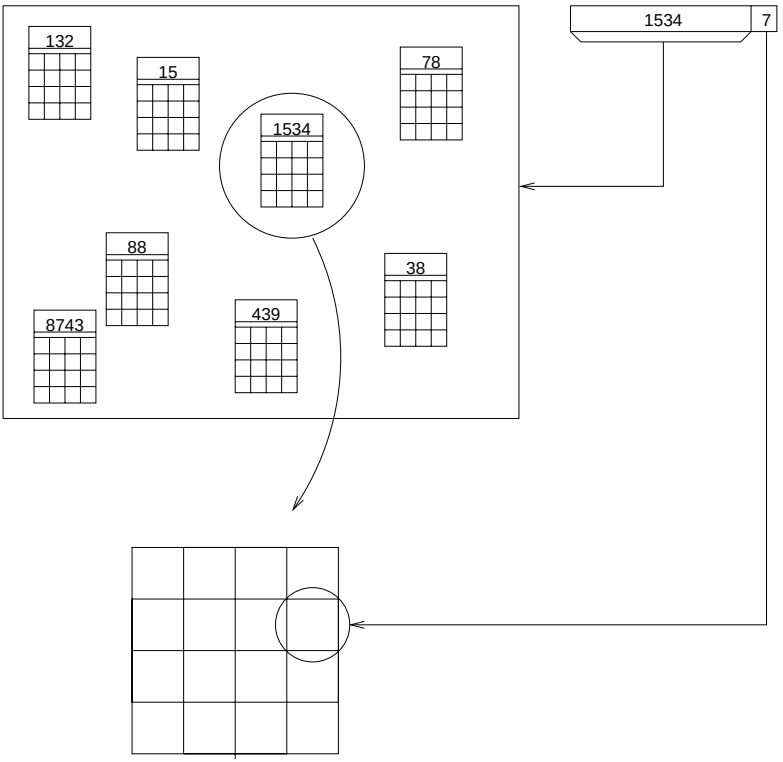


Figure 17.3 :
Accès au cache
associatif
(ci-dessous)



Lorsque le processeur tente d'accéder à un bloc qui n'est pas présent dans l'une des lignes du cache, la mémoire centrale doit être adressée pour que le bloc soit transmis au cache, puis au processeur. Cette situation est appelée *défaut de cache*.

Après un certain temps d'exécution (assez court), le cache est complètement rempli de blocs. Si un défaut de cache se produit alors que le cache est plein, on doit d'abord supprimer un des blocs qu'il contient ; on préférerait supprimer un bloc auquel on n'a pas accédé depuis longtemps. En pratique, un tel schéma nécessiterait trop de circuits pour sa réalisation. Une méthode plus simple est donc généralement utilisée, par exemple un choix aléatoire.

La réalisation d'un cache associatif est assez coûteuse. Le coût principal vient de la complexité des circuits pour la comparaison en parallèle des numéros de blocs. Pour cette raison, on trouve souvent d'autres types d'organisation de cache.

17.2 CACHE DIRECT

De la même manière que pour le cache associatif, la mémoire est divisée en blocs, et le cache en lignes. Contrairement au cache associatif, les lignes du cache direct sont organisées séquentiellement. Si le cache contient, par exemple, 512 kilo-octets, il a donc 32 kilo-blocs, et l'adresse du bloc nécessite 15 bits. Lorsqu'il est copié de la mémoire centrale vers le cache, un bloc aura un emplacement fixe dans le cache, déterminé par les derniers bits (ici les 15 derniers) du numéro de bloc.

Pour cela, l'adresse émise par le processeur est divisée en *trois* parties au lieu des deux du cas associatif. La dernière partie est comme pour le cache associatif : le numéro d'octet dans le bloc. Le numéro de bloc est divisé en deux parties : la *zone* et la *ligne*. Dans notre exemple, la zone sera indiquée par les 13 premiers bits de l'adresse, la ligne par les 15 bits suivants et l'octet par les 4 derniers bits (voir figure 17.4).

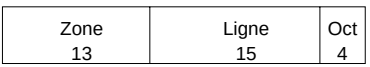


Figure 17.4 : Zone, ligne, octet

Le cache direct s'adresse de la même façon qu'une mémoire normale. La partie *ligne* de l'adresse émise par le processeur est utilisée pour déterminer la ligne du cache dans laquelle doit être stocké le bloc cherché. Plusieurs blocs (ici 2^{13}) sont associés à la même ligne de cache. Mais une ligne peut contenir au plus un bloc à la fois. Il est donc nécessaire de stocker la zone de provenance du bloc avec les données du bloc dans le cache.

La figure 17.5 montre l'adressage d'un octet dans un cache direct.

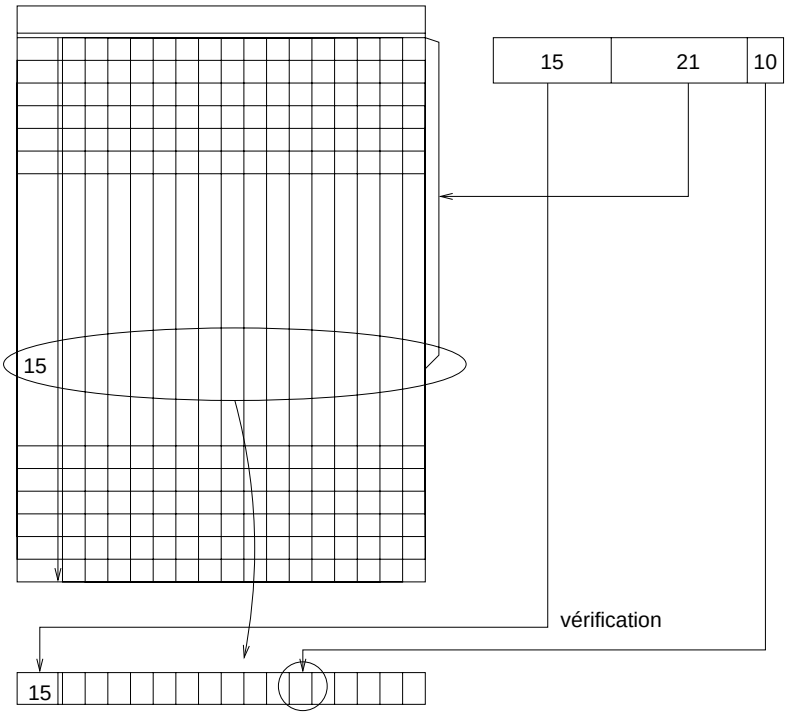


Figure 17.5 : Adressage dans un cache direct

Ici la ligne numéro 21 est inspectée pour vérifier que le bloc qui y est stocké vient bien de la zone 14. Après vérification, l'octet numéro 10 du bloc est transmis au processeur.

Le cache direct est beaucoup moins coûteux à implémenter. Par contre, il a un inconvénient que n'a pas le cache associatif : la probabilité que deux blocs nécessaires simultanément soient affectés à la même ligne est non négligeable. Dans une telle situation, un bloc est inséré dans une ligne pour être presque immédiatement supprimé afin de laisser sa place à un autre bloc affecté à la même ligne.

17.3 SOLUTIONS MIXTES

Afin d'éviter le problème du cache direct sans toutefois être pénalisé par le coût élevé du cache associatif, on peut imaginer des solutions mixtes.

La mémoire est toujours divisée en blocs et le cache en lignes. Les lignes du cache sont organisées de manière séquentielle comme pour le cache direct. La différence par rapport au cache direct est que chaque ligne peut contenir un nombre de blocs supérieur à 1. En fait, chaque ligne du cache est un mini-cache associatif. Le nombre de blocs de chaque ligne est appelé l'*associativité* du cache.

La figure 17.6 montre l'organisation d'un cache d'associativité 2.

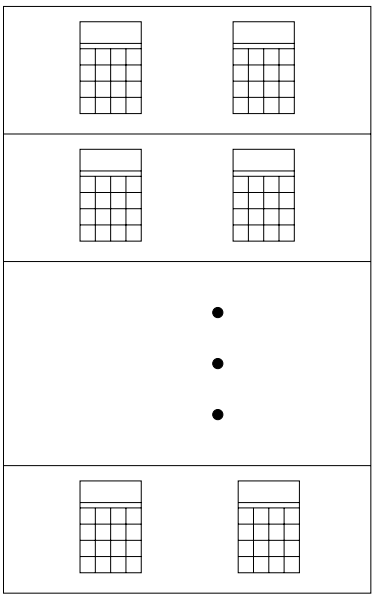


Figure 17.6 : Cache d'associativité 2

Comme avec le cache direct, l'adresse est divisée en *zone*, *ligne* et *octet*. La partie *ligne* est utilisée pour déterminer la ligne du cache pour le bloc. La partie *zone* est stockée avec le bloc dans le cache, et l'ensemble des blocs d'une ligne est recherché en parallèle comme dans le cas d'un cache associatif.

La figure 17.7 montre l'adressage d'un octet dans un cache d'associativité 2.

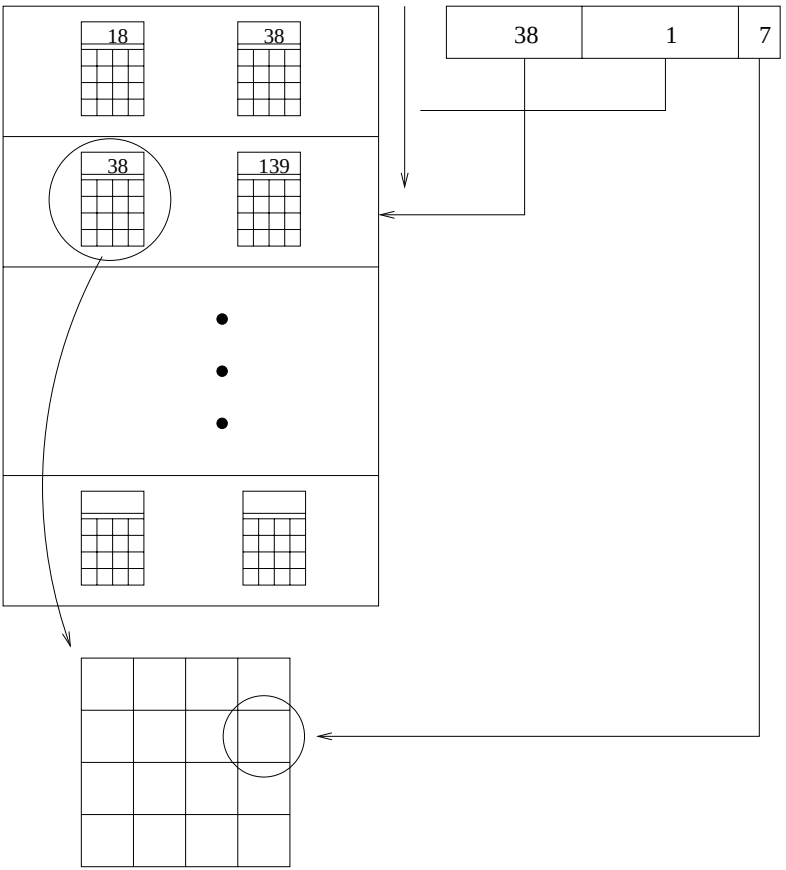


Figure 17.7 : Adressage dans un cache d'associativité 2

On peut voir le cache associatif et le cache direct comme des cas spéciaux de la solution mixte avec une associativité égale au nombre de lignes et à 1 respectivement.

La solution mixte est un bon compromis entre la performance du cache associatif et le coût du cache direct. En pratique une associativité de 4 suffit pour éliminer le problème du cache direct.

17.4 CHOIX DE LA TAILLE DES BLOCS

Dans les exemples des sections précédentes, nous avons supposé que la taille du bloc est de 16 octets. La question concernant la taille optimale d'un bloc se pose alors naturellement. Comme souvent, la réponse est un compromis.

Afin de transmettre un maximum de données simultanément de la mémoire centrale vers le cache, on doit choisir une taille de bloc assez grande. Malheureusement, avoir des blocs de grande taille augmente la probabilité qu'une donnée inutile soit contenue dans le bloc. En pratique, une taille de bloc de 4 mots parait raisonnable.

17.5 RÉCAPITULATIF

Le cache permet à une petite mémoire rapide et chère en combinaison avec une grande mémoire relativement lente, mais peu chère, de simuler une grande mémoire rapide et bon marché.

L'organisation du cache est un compromis entre le coût de fabrication et la probabilité qu'un programme ait un comportement rendant le cache essentiellement inutile.

EXERCICES

Exercice 17.1. On considère des blocs de 2 mots (8 octets). Soit un cache direct de 16 blocs.

(1) Quelle est la taille du cache ?

(2) En supposant que la mémoire à une taille de 2^{32} octets, schématiser la décomposition d'une adresse mémoire en zone, ligne, octet.

(3) Quelle est la taille totale occupée par le cache ?

(4) Voici une suite de références à des adresses de mots (4 octets) :
1, 6, 8, 32, 7, 33, 1, 39, 27, 59, 58, 64, 97, 1.
On suppose que le cache est initialement vide. Déterminer pour chaque référence de la suite si elle conduit à un succès ou un défaut. Donner le contenu final du cache.

Exercice 17.2. Soit un cache associatif de 16 blocs. Répondre aux questions de l'exercice précédent.

Exercice 17.3. Soit un cache d'associativité 2 de 16 blocs. Répondre aux questions de l'exercice précédent.

Chapitre **18**

Multiprogrammation

Pour un grand nombre d'applications, le processeur reste inutilisé pendant un temps considérable. Une application de type traitement de texte passe la plupart de son temps en attente de l'utilisateur, une base de données doit souvent attendre le positionnement de la tête de lecture du disque, etc.

Pour tirer le meilleur parti du processeur, on peut permettre à *plusieurs programmes différents* d'être présents en mémoire simultanément. Un programme en attente peut alors céder le processeur à un autre programme en mémoire. Cette possibilité est appelée la *multiprogrammation*.

Il y a quelques problèmes à résoudre afin de pouvoir autoriser plusieurs programmes (ainsi que leur données respectives) à résider simultanément en mémoire. Dans ce chapitre, nous allons voir plusieurs solutions possibles.

On rencontre deux problèmes similaires, pour lesquels les solutions peuvent être assez différentes. Le premier concerne les instructions de transfert de contrôle (sauts) qui doivent continuer à fonctionner même si le programme est déplacé dans la mémoire. Le deuxième est relatif à la manière dont un programme trouve ses données.

18.1 ADRESSAGE RELATIF AU COMPTEUR ORDINAL

Une façon simple de résoudre le problème des instructions de transfert de contrôle est d'introduire un mode d'*adressage relatif* au compteur ordinal. Au lieu de stocker une adresse absolue dans l'argument d'une telle instruction, on stocke un *décalage* par rapport à l'adresse actuelle du compteur ordinal. On parle alors d'*adresse relative*.

Afin d'effectuer un transfert, l'unité centrale doit donc additionner le contenu du compteur ordinal avec l'argument de l'instruction. Cela donne lieu à une nouvelle famille d'instructions de transfert de contrôle que nous allons appeler des instructions de *branchement*. À chaque instruction de saut (JAL, JZ, JN, etc) correspondra donc une instruction de branchement (BAL, BZ, BN, etc).

La réalisation d'une telle famille d'instructions nécessite de modifier l'architecture. Ces modifications sont illustrées figure 18.1, laquelle doit être comparée à la figure 15.2 (il faudrait aussi rajouter la micro-opération nécessaire au nouveau pilote de bus).

Comme on le voit dans cette figure, le contenu du registre d'adresse est additionné (grâce à l'additionneur supplémentaire) à celui du compteur ordinal, et le résultat peut être transmis sur le bus d'adresses, d'où il peut être chargé dans le compteur ordinal grâce à l'entrée ld de celui-ci.

La seule différence entre le micro-programme d'une instruction de saut et celui de l'instruction de branchement correspondante est que l'utilisation de MOP10 dans le micro-programme de l'instruction de saut doit être remplacée par une nouvelle micro-instruction (non illustrée) permettant de sortir la somme du registre d'adresse et du compteur ordinal sur le bus d'adresse.

Un programme qui n'utilise que des instructions de branchement pour le transfert de contrôle continue à fonctionner même après avoir été déplacé dans la mémoire, alors que ce n'est évidemment pas le cas d'un programme contenant des instructions de saut, dont l'adresse cible est absolue.

Un code ne contenant aucune instruction de saut (mais pouvant contenir des instructions de branchement) est dit *translatable* (en anglais : *position-independent code*), c'est-à-dire code indépendant

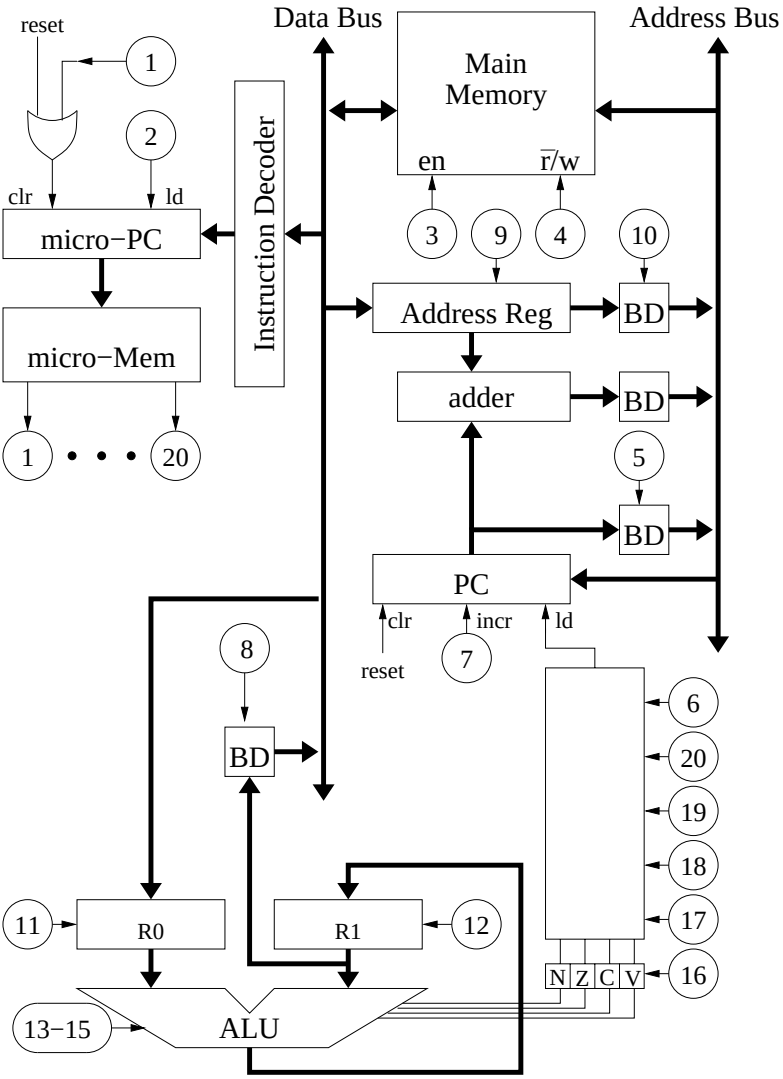


Figure 18.1 : Ordinateur avec instructions de branchement

de la position), ou PIC. Cette technique est toujours utilisée dans les systèmes d'exploitation modernes, mais dans un contexte un peu différent. En fait, elle est utilisée pour réaliser ce que l'on appelle des *bibliothèques partagées* (en anglais : *shared libraries*). Le code d'une

telle bibliothèque peut être incorporé dans plusieurs programmes différents et à des adresses diverses ce qui nécessite justement cette technique.

C'est pour cette raison que, lorsqu'on utilise GCC (le compilateur C du projet GNU) pour produire une bibliothèque partagée, on doit obligatoirement lui donner l'option de compilation `-PIC`.

On peut également imaginer l'utilisation de l'adressage relatif au compteur ordinal pour les données d'un programme, à condition, bien sûr, que ces données restent à une distance fixe des instructions de celui-ci. Dans les sections à suivre, nous allons examiner de meilleures solutions pour positionner le code et les données des programmes.

18.2 REGISTRES BASE ET LIMITE

Une autre solution possible pour rendre un programme et ses données indépendants de la position exacte dans la mémoire est d'introduire ce que l'on appelle un *registre de base*. Ce registre ne doit pas pouvoir être modifié par un programme normal. Toute adresse émise par le programme sera additionnée au contenu de ce registre avant d'être utilisée pour adresser la mémoire. Cette idée est illustrée figure 18.2.

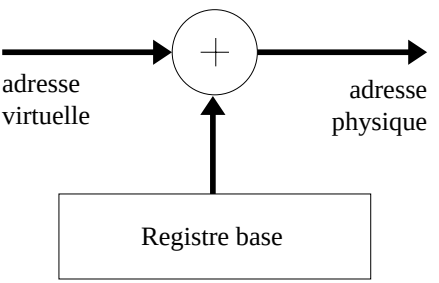


Figure 18.2 : Registre base

Il faut désormais distinguer entre une adresse manipulée par le programme, relative au registre base, et une adresse émise vers la mémoire centrale. Nous allons appeler une adresse du premier type *adresse virtuelle*, et une adresse du deuxième type *adresse physique*.

Le compilateur travaille uniquement avec des adresses virtuelles. Autrement dit, le compilateur fait *comme si* le programme était chargé à l'adresse zéro de la mémoire.

Ce mécanisme permet aux adresses physiques d'avoir plus de bits que les adresses virtuelles. On peut par exemple imaginer des adresses virtuelles de 16 bits et des adresses physiques de 20 bits. À une l'époque où le coût de la mémoire centrale était considérable et où il était par conséquent important que les programmes soient les plus petits possible, il était très intéressant de permettre à la fois des adresses de petite taille dans le programme et de pouvoir munir l'unité centrale de plus de mémoire si nécessaire.

La multiprogrammation introduit un problème supplémentaire qui n'existe pas lorsqu'un seul programme à la fois peut être présent en mémoire centrale, à savoir la *protection* entre deux programmes. En effet, rien n'empêcherait un programme de lire, voire d'écrire des données et même des instructions d'un autre programme. À l'époque de l'invention de la multiprogrammation le problème actuel des virus n'existait pas, mais il fallait quand même éviter qu'un programme défectueux puisse endommager les données ou les instructions d'un programme *a priori* correct.

Un *registre limite* représente une solution simple à ce problème. Ce registre contient la *taille* du programme concaténé à ses données. L'espace physique en mémoire centrale disponible pour un programme est désormais défini à la fois par le registre de base et par le registre limite. Cette idée est illustrée figure 18.3.

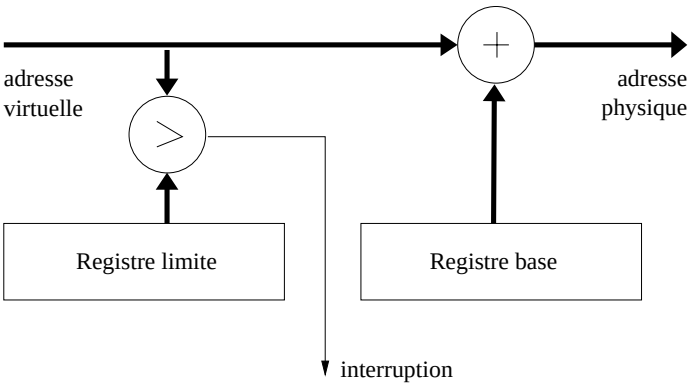


Figure 18.3 : Registre limite

Avant la conversion d'une adresse virtuelle en adresse physique, celle-ci est comparée au contenu du registre limite. Si elle est supérieure à celui-ci, alors une interruption de l'unité centrale se produit. Constatant que le programme est défectueux ou malintentionné, l'unité centrale termine alors l'exécution de ce programme.

Chaque registre base et limite existe en un seul exemplaire dans l'unité centrale, mais les valeurs qu'il contient sont propres à chaque exemplaire de programme présent en mémoire simultanément. Lorsque l'on décide d'arrêter temporairement l'exécution d'un programme afin de continuer celle d'un autre, il faut donc remplacer les valeurs de ces registres par celles appartenant au nouveau programme. Par conséquent, il faut maintenir quelque part l'ensemble des valeurs de ces registres afin de les rendre disponibles au programme chargé d'effectuer ce remplacement.

18.3 SEGMENTATION

La solution avec les registres base et limite fonctionne bien, mais la multiprogrammation fait aussi des optimisations de place qui ne sont pas exploitables par le biais de cette technique simple.

Afin de voir où des optimisations seraient possibles, on doit se rendre compte qu'il arrive souvent que *plusieurs exemplaires* du même programme, mais avec des données différentes, soient présents simultanément en mémoire. C'est le cas chaque fois qu'un utilisateur exécute plusieurs fois un programme comme un éditeur de texte ou un compilateur, ou si deux personnes différentes exécutent le même programme.

Avec la solution des registres base et limite, il est obligatoire de dupliquer non seulement les données du programme pour chaque exemplaire présent, mais également les instructions, même si celles-ci sont identiques pour chaque exemplaire.

Afin d'éviter cette duplication, on peut imaginer l'introduction d'un couple de registres (base et limite) pour les instructions et un autre couple pour les données. En fait, il est courant de séparer non seulement les données des instructions, mais également la pile des instructions des autres données. Enfin, on peut également imaginer plusieurs zones de données dont certaines peuvent être partagées avec d'autres programmes et d'autres réservées à un seul exemplaire du

programme. Une telle zone est appelée un *segment*, et le mécanisme pour exploiter cette idée la *segmentation*.

Une architecture donnée contiendra un nombre fixe de segments utilisables à un instant donné par un programme. Ce nombre est souvent une petite puissance de 2, par exemple 4 ou 8.

Afin d'exploiter la segmentation, on considère que l'adresse virtuelle est composée de deux parties. La partie la plus significative contient 2 (pour 4 segments) ou 3 bits (pour 8 segments) qui donnent le *numéro de segment* à utiliser, et la partie la moins significative donne le *décalage par rapport au début du segment*. Cette idée est illustrée figure 18.4.

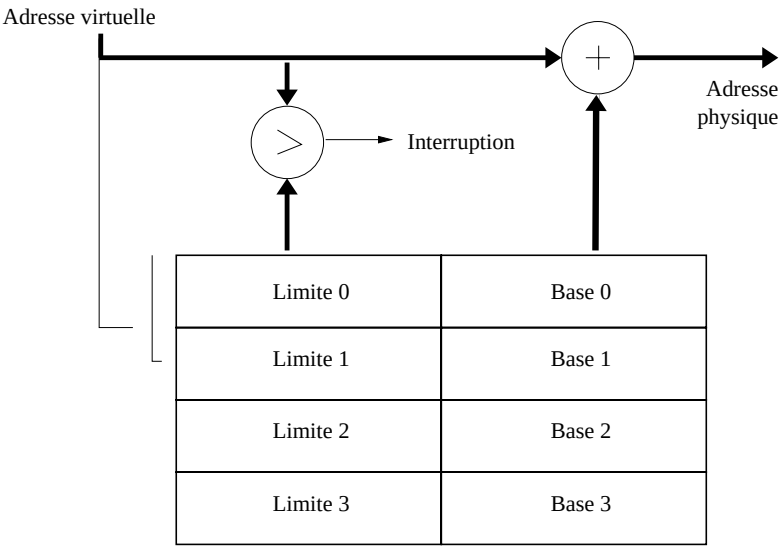


Figure 18.4 : Segmentation

Comme le mécanisme des registres base et limite, la segmentation permet à l'adresse physique d'avoir davantage de bits que les adresses virtuelles, avec les mêmes avantages.

Comme avec les registres base et limite, la table illustrée par la figure 18.4 existe en un seul exemplaire dans l'unité centrale, mais les valeurs qu'elle contient sont propres à chaque exemplaire de programme présent en mémoire simultanément.

18.4 PAGINATION

Le mécanisme de segmentation décrit dans la section précédente a un inconvénient majeur : il ne permet pas facilement à un segment de grandir. Pourtant, un programme a souvent besoin d'allouer de la mémoire supplémentaire. C'est le rôle de la fonction `malloc` du langage C.

Avec la segmentation, afin de permettre à un segment de grandir, il faut déplacer tous les autres segments qui lui succèdent en mémoire. Pour cette raison, la segmentation n'est pas utilisée dans les ordinateurs modernes. Dans cette section, nous décrivons un autre mécanisme appelé la *pagination* qui a totalement remplacé celui de la segmentation depuis longtemps.

Avec la pagination, la mémoire est divisée en *pages*. La taille habituelle d'une page pour un processeur avec des adresses de 32 bits est de 4 Ko. L'adresse est donc divisée en deux parties : le numéro de page et le numéro d'octet à l'intérieur de la page.

Les adresses soumises à la mémoire sont d'abord traduites par une unité que l'on appelle MMU (pour *memory management unit*, unité de gestion de la mémoire). Une grande partie de la MMU comprend une *table des pages*. Cette table contient autant d'entrées que de pages adressables par un programme, soit 2^{20} pour un processeur de 32 bits avec des pages de 4 Ko. Ceci est illustré par la figure 18.5.

Chaque programme possède sa propre table des pages. Lorsqu'un programme n'a plus besoin du processeur, ce dernier passe à l'exécution d'un autre programme. Il faut alors que la table des pages du nouveau programme soit employée par la MMU. Une fois ce changement effectué, c'est comme si le contenu de la mémoire entière avait changé. Les deux programmes ne voient pas le même contenu de la mémoire à la même adresse.

L'espace occupé par la table des pages est considérable. Comme cela a été indiqué ci-dessus, pour un processeur de 32 bits avec des pages de 4 Ko, la table contient environ un million d'entrées. La table des pages doit donc être stockée elle-même dans la mémoire centrale. Pour que le processeur puisse accéder à une instruction ou à une donnée à une adresse quelconque, il faut d'abord accéder à la mémoire afin de récupérer l'entrée dans la table des pages correspondant à cette adresse, puis à nouveau accéder à la mémoire et récupérer l'instruction ou la donnée en question. Il faudrait donc

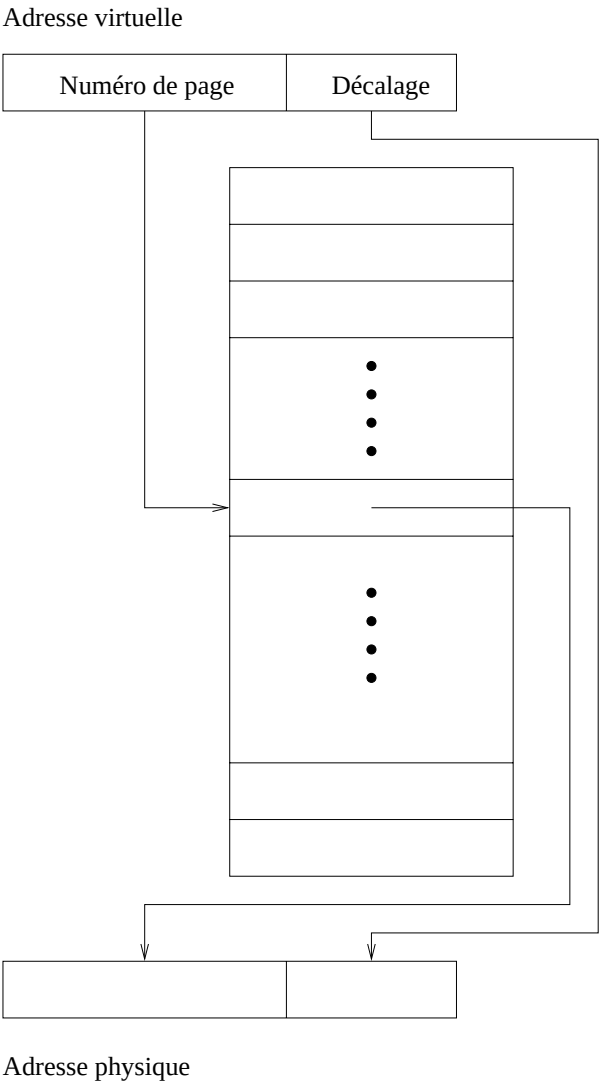


Figure 18.5 : Table des pages

accéder à la mémoire deux fois, ce qui est inacceptable du point de vue de la performance, surtout que la mémoire est déjà plus lente que le processeur.

Pour résoudre ce problème on introduit une petite mémoire cache associative nommée TLB (en anglais : *Translation Lookaside Buffer*). Cette mémoire ne contient pas les instructions ou les données du programmes, mais un sous-ensemble des entrées de la table des pages. Lorsque le processeur tente d'accéder à une page, le TLB est d'abord consulté pour déterminer si l'adresse de la page physique correspondante y figure. Dans ce cas, la traduction de l'adresse virtuelle et l'adresse physique est immédiate. Dans le cas contraire, une interruption est provoquée, et on passe la main au traitant d'interruptions qui doit alors accéder à la mémoire centrale pour trouver l'entrée de la table des pages souhaitée, et stocker cette entrée dans le TLB.

Fort heureusement, le TLB n'a pas besoin de beaucoup d'entrées. Un programme accède souvent plusieurs fois à un petit sous-ensemble de ses pages avant de passer à d'autres pages. De plus, la pénalité lorsque l'entrée doit être cherchée en mémoire centrale est relativement modeste.

Lors du passage d'un programme à un autre, il faut par contre vider le TLB, car la traduction entre adresses virtuelles et adresses physiques est spécifique à chaque programme. Le processeur possède alors une instruction spécialisée pour cet effet.

Comme avec la segmentation, la pagination permet à la taille de la mémoire physique de dépasser celle de la mémoire disponible pour un programme donné. On distingue alors l'*espace adressable* déterminé par le nombre de bits d'une adresse et la *taille de la mémoire*. Rien ne nous empêche comme dans la figure 18.5 d'avoir des entrées dans la table avec un nombre de bits supérieur au nombre de bits utilisé pour indexer la table. Ce type de technique est parfois utilisé lorsque l'espace adressable est assez petit (par exemple 2^{16}), mais le degré de multiprogrammation – le nombre de programmes présents simultanément en mémoire – important. Dans ce cas, l'espace adressable par chaque programme est relativement petit mais, du fait d'un nombre important de programmes, la taille de la mémoire physique peut être considérable.

La pagination permet à l'espace adressable d'avoir des *trous*, à savoir des adresses sans mémoire physique allouée. La table des pages contiendra un bit pour déterminer si l'entrée est *valide*, à savoir si la valeur qu'elle contient correspond à une page réelle. Lors du charge-

ment du TLB, ce bit est vérifié. S'il indique que la page correspondante est absente, une nouvelle page est allouée, puis l'entrée de la table des pages est mise à jour.

On utilise souvent l'existence de ces trous pour organiser l'espace adressable d'un programme d'une manière identique, indépendamment de la taille réelle de celui-ci. Cette idée est illustrée par la figure 18.6 qui montre la disposition des différentes partie d'un programme utilisé par le système Unix.

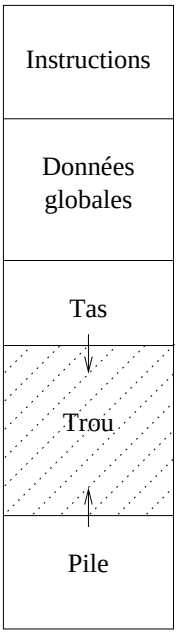


Figure 18.6 : Disposition de l'espace adressable

À l'adresse 0 de l'espace, on trouve les instructions du programme, suivies des données globales. Le *tas* contient les données allouées dynamiquement par la fonction `malloc`. Le trou entre le tas et la pile permet à ces deux espaces de grandir lorsque le programme en a besoin.

La plupart des programmes occupent relativement peu de place en mémoire par rapport à l'espace adressable. Dans ce cas, l'espace mémoire occupé par la table des pages devient considérable par rapport à la taille du programme. Par exemple, un programme dont la taille des instructions et des données est de 4 Mo occupe 1 000 pages. Si l'on considère que chaque entrée de la table des pages nécessite 4 octets

et que la table des pages contient environ un million d'entrées, alors la table des pages du programme nécessite autant de mémoire que les instructions et les données de celui-ci.

Une solution à ce problème est d'organiser la table des pages en plusieurs niveaux. Cette idée est illustrée par la figure 18.7.

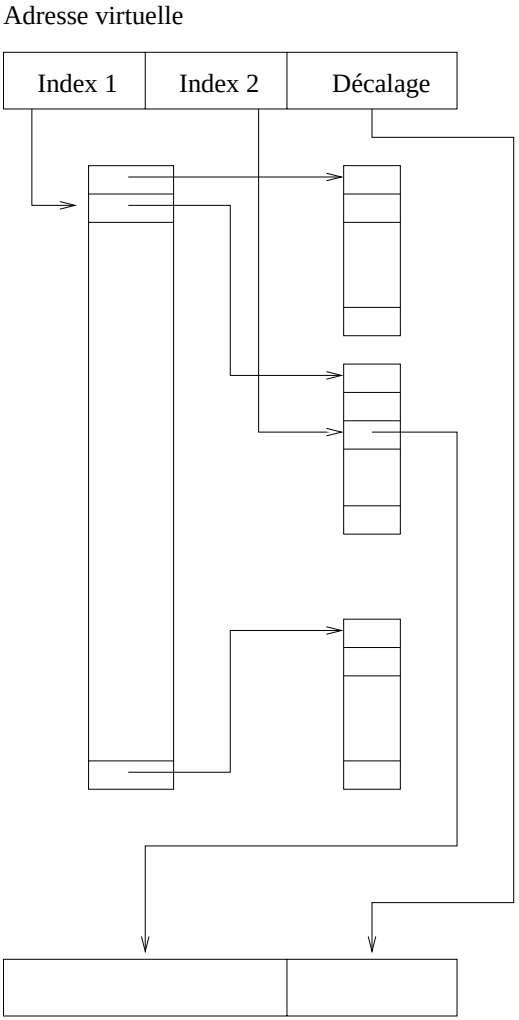


Figure 18.7 : Table des pages à deux niveaux

L'adresse virtuelle est divisée en trois parties au lieu de deux. La première constitue un index dans la table du premier niveau. Cette table contient des pointeurs sur les véritables tables des pages du deuxième niveau. La deuxième partie de l'adresse virtuelle est utilisée pour indexer l'une des tables du deuxième niveau qui contient le numéro de la page physique.

En supposant que les tables du deuxième niveau aient la taille d'une page (4 Ko), une telle table aura alors 2^{10} entrées si l'on réserve 4 octets pour chaque entrée. La deuxième partie de l'adresse virtuelle est donc constituée de 10 bits, ce qui laisse 10 bits pour la première partie si la taille de l'adresse virtuelle est de 32 bits au total. La table du premier niveau contient donc également 2^{10} entrées de 4 octets chacune.

Afin de voir l'économie considérable obtenue grâce à ce système de table des pages à plusieurs niveaux, reprenons notre exemple ci-dessus avec un programme ayant besoin de 1 000 pages, disons la moitié au début de la mémoire (instructions et données) et l'autre moitié à la fin (pile). Seules deux tables de deuxième niveau sont nécessaires, une pour le début de la mémoire et une autre pour la fin. Chacune des deux contiendra 500 entrées réellement utilisées. Avec la table du premier niveau, l'espace mémoire occupé par les tables est donc 12 Ko ce qui doit être comparé avec 4 Mo pour la table à un seul niveau.

Lorsque le TLB ne contient pas de numéro de page, et que le système doit accéder à la table des pages en mémoire, avec une table des pages à deux niveaux, cela nécessite maintenant deux accès au lieu d'un seul pour remplir l'entrée du TLB. Cela reste fort heureusement négligeable, car le coût de la mise à jour du TLB est dominé par l'interruption et l'exécution du traitant.

18.5 PROCESSUS

Dans la section précédente, nous avons vu que la table des pages est spécifique à chaque programme présent en mémoire simultanément.

Désormais, nous appelons *processus* l'exécution d'un programme ou encore un programme en cours d'exécution.

D'autres informations doivent être stockées pour qu'un processus suspendu puisse reprendre son exécution ultérieurement :

– la valeur de PC,
– la valeur de CC (Code Conditions),
– la valeur de SP,
– le contenu des registres habituels.

Ces informations constituent le *contexte* du processus. Notons que, dans notre exemple d'architecture, il n'est pas nécessaire de stocker le registre d'adresse et celui de données, car ces derniers sont utilisés de façon strictement temporaire par les instructions. Aucune instruction ne dépend du contenu préalable de ces registres.

Le système d'exploitation de la machine doit stocker toutes ces informations quelque part en mémoire. Il est pratique de les regrouper dans une structure (au sens du langage C) avec suffisamment de champs pour stocker toutes les informations nécessaires à la suspension d'un processus et à la reprise de son exécution plus tard.

18.6 CHANGEMENT DE CONTEXTE

Le fait de suspendre le processus et de passer à l'exécution d'un autre processus suspendu est appelé *changement de contexte*. C'est la partie centrale du système d'exploitation qui s'occupe des changements de contexte.

Un changement de contexte est provoqué soit par une interruption (voir section 16.2), soit par l'exécution d'une instruction qui se comporte comme si une interruption avait eu lieu : on parle de pièges (en anglais : *traps*), ou d'interruptions logicielles (en anglais : *software interrupts*).

Le traitant d'interruptions est chargé (directement ou indirectement) de détecter si celle-ci doit entraîner un changement de contexte.

La seule information stockée dans la pile lors d'une interruption dans notre exemple d'architecture est la valeur de PC. Si le traitant d'interruption détermine qu'un changement de contexte est nécessaire, il doit alors sauvegarder l'ensemble des registres indiqués à la section précédente dans le contexte appartenant au processus en exécution, puis restaurer ces registres à partir du contexte d'un autre processus. Puisque le pointeur de pile change pendant ce changement de contexte, lorsque le traitant exécute l'instruction RET, le retour aura comme effet de restaurer le compteur ordinal du nouveau processus.

18.7 INSTRUCTIONS PRIVILÉGIÉES

Afin d'éviter qu'un processus puisse modifier le contenu de la mémoire appartenant à un autre processus, il est important d'empêcher ce premier de modifier son espace adressable. Sinon, il serait toujours possible qu'un processus change le contenu de sa table des pages, y mette la référence d'une page appartenant à un autre processus, puis modifie cette page.

La solution à ce problème est de réserver certaines instructions au système d'exploitation. Ce sont des *instructions privilégiées*.

Pour permettre à certaines instructions d'être privilégiées, un registre supplémentaire de taille un bit est ajouté à l'architecture. Ce registre indique le *mode* d'exécution, où 0 signifie *mode utilisateur* et 1 signifie *mode superviseur*. Le micro-programme pour les instructions privilégiées vérifie d'abord que ce registre vaut 1 et refuse l'exécution si tel n'est pas le cas.

Mais comment passer du mode utilisateur au mode superviseur ? Une solution possible serait d'ajouter des instructions supplémentaires pour ce passage. Mais ceci permettrait à un processus d'effectuer ce passage, ce qui n'est pas souhaité. Une meilleure solution est de passer du mode utilisateur au mode superviseur lorsque le processeur est interrompu. Le micro-programme pour les interruptions est donc chargé de ce passage. Une nouvelle instruction RTI est responsable du passage inverse. Elle fonctionne comme l'instruction RET.

Le fait d'empêcher un processus ordinaire de modifier son espace adressable, permet aussi au système d'exploitation de protéger les unités périphériques. Il suffit alors de ne pas inclure les adresses physiques correspondant à ces unités dans l'espace adressable d'un processus ordinaire. Puisque ce dernier est incapable de modifier son espace adressable, il est aussi incapable d'accéder directement aux unités périphériques.

18.8 PROTECTION

La multiprogrammation permet à plusieurs processus de partager des instructions. Puisqu'il existe souvent plusieurs processus simultanés exécutant le même programme (comme un éditeur de texte ou un

navigateur Web), cela permet d'éviter la duplication des pages contenant les instructions et donc d'économiser l'espace mémoire global de l'ordinateur.

Par contre, ce partage pose un problème de sécurité. Dans le cas général, deux processus exécutant le même programme n'appartiennent pas au même utilisateur. Rien n'empêche au programme de modifier ses propres instructions. Si ces instructions sont partagées, un utilisateur peut alors influencer le fonctionnement du programme d'un autre utilisateur, peut-être avec un résultat désastreux.

Afin d'éviter cela, tout en gardant la possibilité que plusieurs processus puissent partager les instructions d'un même programme, on a recours à la notion de *protection*. On introduit alors dans chaque entrée du TLB (voir section 18.4) quelques bits supplémentaires (souvent 3) indiquant, pour chaque page, si le processus a le droit de la lire, l'écrire ou de l'exécuter (en anglais : *read, write, execute*). Le processeur vérifie à chaque accès à la mémoire (et selon la nature de l'instruction en exécution) que les valeurs des bits de protection permettent l'accès. Dans le cas contraire, une interruption est provoquée. C'est alors encore une fois le système d'exploitation qui doit déterminer l'action à entreprendre, souvent celle de terminer l'exécution du processus.

18.9 RÉCAPITULATIF

Un ordinateur personnel moderne permet à plusieurs programmes d'être chargés simultanément en mémoire. Pour faire croire que les programmes s'exécutent simultanément, le système d'exploitation change régulièrement le programme en cours d'exécution. Pour que les instructions des programmes puissent être facilement déplacées, elles ne doivent pas dépendre de l'endroit exact où se trouvent le programme et ses données.

Une solution possible de ce problème est de faire croire à chaque programme que tout l'espace adressable de l'ordinateur est à sa disposition. Cela nécessite que le changement du programme en exécution modifie la correspondance entre adresses virtuelles et adresses physiques.

EXERCICES

Aujourd'hui les processeurs à 64 bits commencent à être couramment utilisés. Un tel processeur a un espace adressable de 2^{64} octets.

Exercice 18.1. Si l'on estime le prix d'achat de la mémoire RAM à 10 centimes par Mo, combien faut-il payer pour remplir un tel processeur de mémoire ?

Exercice 18.2. Avec une taille de page de 4 Ko, quelle serait la taille d'une table des pages à un seul niveau pour un processeur de 64 bits ?

Exercice 18.3. Quel serait l'espace mémoire minimum occupé par une table des pages à deux niveaux pour un processeur 64 bits ? Pour cela, il faut envisager différentes manières de découper la partie la plus significative d'une adresse virtuelle de 64 bits.

Exercice 18.4. En combien de niveaux faut-il découper la table des pages d'un processeur 64 bits pour que l'espace mémoire occupé par cette table reste négligeable pour un programme ayant besoin de 4 Mo de mémoire (environ 1 000 pages) ?

Chapitre **19**

Mémoire virtuelle

La mémoire cache (voir chapitre 17) permet à la combinaison d'une petite mémoire rapide et d'une grande mémoire relativement lente, et moins onéreuse, de se comporter comme une mémoire relativement grande et rapide. Une technique similaire peut être utilisée entre la mémoire centrale et le disque. On parle alors de *mémoire virtuelle*

19.1 PERFORMANCE

La mémoire centrale sert de cache pour le disque au même titre que le cache pour la mémoire centrale. Mais alors que la différence de performance entre le cache et la mémoire centrale est environ d'un facteur 10, la différence de performance entre la mémoire centrale et le disque est d'environ un facteur 100 000.

Cette différence en performance fait que la mémoire virtuelle peut être implémentée en grande partie sous forme logicielle (dans le système d'exploitation) alors que le cache est entièrement implémenté par du matériel.

Le grand problème avec l'écart de performance entre la mémoire centrale et celle du disque (par rapport à la différence entre la performance de la mémoire cache et celle de la mémoire centrale dans le cas du cache) est que l'illusion d'avoir une mémoire de la taille

du disque et de la rapidité de la mémoire centrale est plus difficile à obtenir que l'illusion similaire dans le cas du cache.

En fait, afin d'obtenir cette apparence, le programme doit respecter le principe de localité (voir chapitre 17) encore plus que dans le cas du cache. La fréquence de transfert de pages entre le disque et la mémoire centrale doit être beaucoup plus faible que dans le cas du cache pour compenser la lenteur du disque. Si cette fréquence est trop élevée, l'ordinateur passe une grande partie de son temps à attendre le déplacement du bras du disque. C'est un phénomène que l'on peut observer sur un ordinateur personnel lorsque l'on démarre une nouvelle application de taille considérable, ou alors si une application démarrée est utilisée après un certain temps d'inactivité.

Le principe de localité est naturellement respecté dans le cas d'un ordinateur personnel avec plusieurs applications potentiellement en exécution, mais dont la taille de chacune ne dépasse pas celle de la mémoire centrale. La raison est que l'utilisateur a tendance à travailler pendant un certain temps avec une seule application avant de l'abandonner temporairement en faveur d'une autre.

19.2 SUPPORT MATÉRIEL

La mémoire virtuelle nécessite relativement peu de support matériel supplémentaire par rapport à la table des pages (voir section 18.4). Un bit d'information est ajouté à chaque entrée de la table des pages. Le bit indique si la page est présente en mémoire centrale. Si c'est le cas, le reste de l'entrée indique le numéro de page physique comme auparavant. Si la page n'est pas présente, alors l'entrée indique l'adresse sur disque où est stockée la page.

L'entrée dans la table des pages est vérifiée par le matériel à chaque accès à la mémoire. Si la page n'est pas présente, alors un piège (en anglais : *trap*) (voir section 18.6) est provoqué. Le processus est alors suspendu et une lecture sur disque commence afin de charger la page en mémoire centrale. Le processus est réveillé lorsque la page est chargée. Entre-temps, à cause de la lenteur relative du disque, un autre processus peut exécuter plusieurs millions d'instructions.

19.3 RÉCAPITULATIF

La mémoire virtuelle est une technique puissante permettant dans certains cas de créer l'illusion que l'ordinateur est équipé d'une mémoire de la rapidité de la mémoire centrale et de la taille du disque.

Dû au coût relativement faible du disque par rapport à celui de la mémoire centrale, cette technique est importante, car elle permet d'exécuter des applications dont la taille combinée dépasse celle de la mémoire centrale.

EXERCICE

Exercice 19.1. Si l'on estime le temps d'accès à la mémoire centrale à environ 10 ns et celui d'accès au disque à environ 1 ms, déterminer le *taux des défauts de page* maximum afin que la performance soit dégradée d'au plus un facteur 2 par rapport à la performance sans défaut de page. Le taux de défauts de page est défini comme étant le nombre d'accès à la mémoire nécessitant un transfert du disque vers la mémoire par rapport au nombre total d'accès.

PARTIE 4

ANNEXES

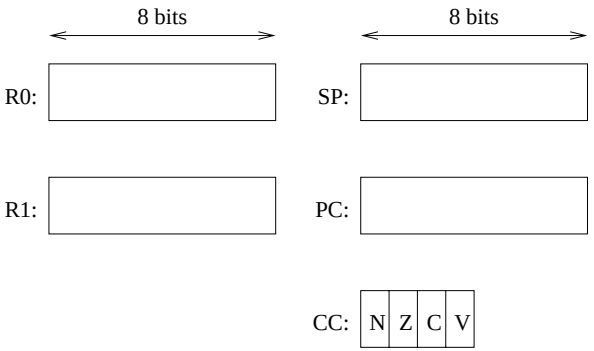
Annexe **A**

Le modèle de programmation

Toutes les parties d'un ordinateur ne sont pas visibles par un programmeur. Cela permet au fabricant de sortir un nouveau modèle avec une implémentation différente tout en gardant le même *modèle de programmation*. Cela implique qu'un ancien programme peut continuer à tourner sur le nouveau modèle sans modification.

A.1 REGISTRES

Le modèle de programmation contient quatre registres : R0 et R1 pour l'arithmétique, le pointeur de pile (SP), le compteur ordinal (PC) et le registre des conditions (CC) :



Le pointeur de pile SP pointe sur le sommet de la pile, à savoir la plus petite adresse dans la pile dont le contenu est défini. Le fait d'empiler une donnée, décrémente d'abord la valeur de SP, puis stocke la donnée à la nouvelle adresse indiquée par SP. Le fait de dépiler une donnée récupère d'abord le contenu de la mémoire indiqué par SP, puis incrémente SP.

Le registre CC contient quatre bits :

N égal à `1` si et seulement si le résultat de la dernière opération arithmétique est négatif,

Z égal à `1` si et seulement si le résultat de la dernière opération arithmétique est zéro,

C égal à `1` si et seulement si le résultat de la dernière opération arithmétique donne une retenue,

V égal à `1` si et seulement si le résultat de la dernière opération arithmétique donne un débordement.

A.2 INSTRUCTIONS

Lorsque l'exécution d'une instruction commence, PC pointe déjà sur l'octet immédiatement après le code de l'instruction. Le tableau décrivant toutes les instructions de notre modèle est le suivant :

Nom	Taille octets	Code	Description française	Description formelle
NOP	1	0	aucun effet	–
LDIMM	2	1	chargement immédiat	$R0 = M[PC++]$
LD	2	2	chargement	$R0 = M[M[PC++]]$
ST	2	3	stockage	$M[M[PC++]] = R1$
COPY	1	4	copie de R0 dans R1	$R1 = R0$
SHL	1	5	décalage de R1 à gauche	$R1 <<= R1$
SHR	1	6	décalage de R1 à droite	$R1 >>= R1$
ADD	1	7	R1 reçoit la somme de R0 et R1	$R1+ = R0$
SUB	1	8	R1 reçoit la différence de R0 et R1	$R1- = R0$
AND	1	9	R1 reçoit la *et* logique de R0 et R1	$R1\& = R0$
OR	1	10	R1 reçoit la *ou* logique de R0 et R1	$R1\| = R0$
NOT	1	11	R1 reçoit la négation logique de R1	$R1 = \bar{R1}$
JAL	2	12	saut non conditionnel	$PC = M[PC]$
JN	2	13	saut si résultat négatif	$PC = N == 1?$ $M[PC] : PC + 1$
JZ	2	14	saut si résultat nul	$PC = Z == 1$ $M[PC] : PC + 1$
JV	2	15	saut si débordement	$PC = V == 1$ $M[PC] : PC + 1$
JC	2	16	saut si retenue	$PC = C == 1$ $M[PC] : PC + 1$
JSR	2	17	appel à sous-programme	$M[--SP] = PC + 1$ $PC = M[PC]$
RET	1	18	retour de sous-programme	$PC = M[SP++]$
PUSH	1	19	empilement de R1	$M[--SP] = R1$
POP	1	20	dépilement dans R0	$R0 = M[SP++]$
LDS	2	21	chargement d'une valeur de la pile	$R0 = M[SP + M[PC++]]$
STS	2	22	stockage dans la pile	$M[SP + M[PC++]] = R1$
ADDSP	2	23	addition d'une constante à SP	$SP+ = M[PC++]$

Annexe **B**

Solutions des exercices

Exercice 1.1 3,52 Go.

Exercice 1.2

(1) 30,3 Mo.

(2) 2,8 Mo.

(3) Le fichier compressé est 10,8 fois plus petit.

(4) 57 minutes.

Exercice 2.1 Utiliser deux portes-*ou* et trois inverseurs.

Exercice 2.2 Remarquer qu'une porte-*non-ou* recevant la même variable x sur ses deux entrées est un inverseur. On utilisera six portes-*non-ou* à deux entrées dont quatre pour inverser les entrées des deux autres portes-*non-ou* de manière à les transformer et portes-*et*.

Exercice 2.3 Non. Les fonctions réalisées par les portes *ou* et *et* sont toutes deux croissantes (au sens où si $x \leqslant y$ and $x' \leqslant y'$ alors $x\ ou\ y \leqslant x'\ ou\ y'$ et $x\ et\ y \leqslant x'\ et\ y'$). La composition de ces fonctions donnera toujours une fonction croissante et donc on ne pourra pas réaliser de fonction non croissante comme celle réalisée par la porte-*non-et*.

Exercice 3.1 Le circuit est présenté figure B.1.

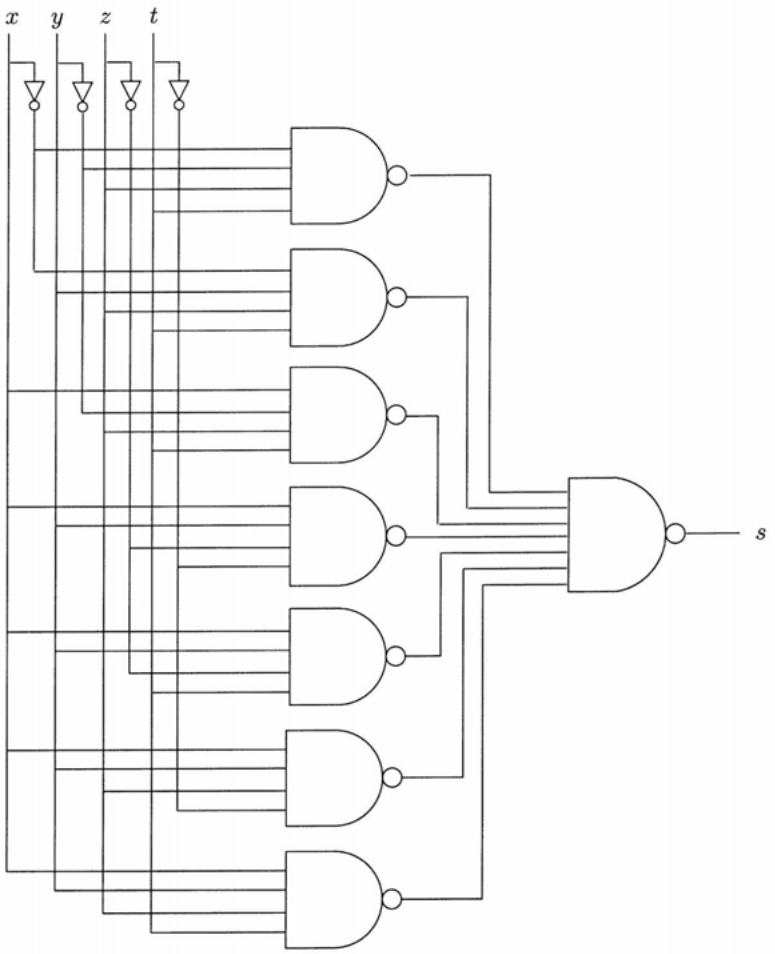


Figure B.1 : Solution de l'exercice 3.1

Exercice 3.2 La table est présentée figure B.2.

Figure B.2 : Solution de l'exercice 3.2

x	y	s_1	s_2
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	1

Exercice 3.3 Le circuit est présenté figure B.3.

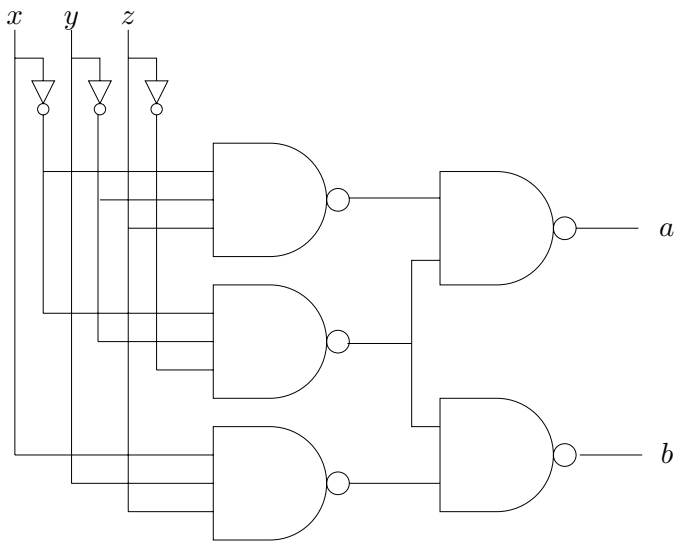


Figure B.3 : Solution de l'exercice 3.3

Exercice 4.1

(1) Développer $x\ xor\ (y\ xor\ z)$ en une somme de 4 produits de trois variables. Faire de même pour $(x\ xor\ y)\ xor\ z$.

(2) $x\ xor\ \overline{y} = \overline{x}\ xor\ y = \overline{x\ xor\ y}$.

(3) – $x \oplus y \oplus z$ vaut 1 si et seulement si x, y et z valent toutes 1 ou alors une seule des trois vaut 1.

– $x_1\ xor\ x_2\ xor\ \ldots\ xor\ x_n$ vaut 1 si et seulement si le nombre des x_i valant 1 est pair.

Exercice 4.2

(1) – $f = \overline{yz} + yz + x\overline{y}$ ou $f = \overline{yz} + yz + xz$

– $g = x\overline{y} + \overline{y}\overline{t} + y\overline{z}t$

(2) Les formes suivantes sont obtenues à partir de la table, par simplifications successives, puis remplacement des négations grâce au fait que $\overline{x} = 1\ xor\ x$.

– $f = 1\ xor\ y\ xor\ z\ xor\ xz\ xor\ xyz$

– $g = 1\ xor\ y\ xor\ t\ xor\ xt\ xor\ xyt\ xor\ yzt$

Exercice 5.1

a_1	a_0	x_3	x_2	x_1	x_0	y
0	0	-	-	-	0	0
0	0	-	-	-	1	1
0	1	-	-	0	-	0
0	1	-	-	1	-	1
1	0	-	0	-	-	0
1	0	-	1	-	-	1
1	1	0	-	-	-	0
1	1	1	-	-	-	1

$$y = \overline{a_1 a_0} x_0 + \overline{a_1} a_0 x_1 + a_1 \overline{a_0} x_2 + a_1 a_0 x_3$$

Le circuit s'obtient immédiatement à partir de la table suivant la méthode générale.

Exercice 5.2 Les 4 entrées a_i plus les 16 entrées x_j donneraient une table de vérité avec 2^{20} lignes. Le multiplexeur est présenté figure B.4.

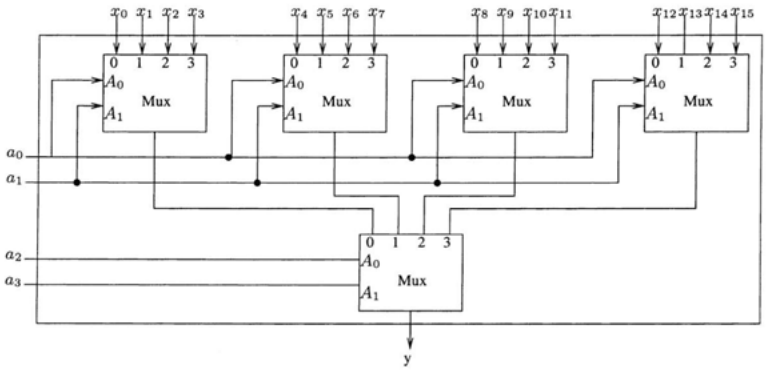


Figure B.4 : Solution de l'exercice 5.2

Exercice 5.3 On utilise une méthode récursive. On construit d'abord la table de vérité pour comparer deux mots de 1 bit chacun (donnée figure B.5).

a	b	y_1	y_2
0	0	0	0
0	1	0	1
1	1	0	0
1	0	1	0

Figure B.5 : Table pour l'exercice 5.3

a_i	b_i	y_1^{i-1}	y_2^{i-1}	y_1^i	y_2^i
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	-	-
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	0	1
0	1	1	1	-	-
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1	-	-
1	1	0	0	0	0
1	1	0	1	0	1
1	1	1	0	1	0
1	1	1	1	-	-

Figure B.6 : Table pour l'exercice 5.3

En utilisant la méthode générale, on construit facilement un circuit comparateur pour deux mots de 1 bit. Puis on construit la table de vérité pour comparer deux mots de i bits (donnée figure B.6) connaissant au préalable la comparaison des $i-1$ premiers bits.

Le circuit final (donné figure B.7) est un assemblage de comparateurs 1-bits.

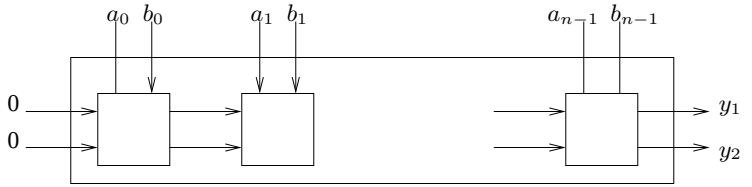


Figure B.7 : Circuit pour l'exercice 5.3

Exercice 6.1 Il ne peut jamais y avoir de débordement avec des opérandes de signe différents.

– Carry (C) : retenue sur le dernier bit.

– Débordement (V) : quand les opérandes sont de même signe et que le résultat est de signe différent.

x	y	c	s	V	C
0	0	0	0	0	0
0	0	1	1	1	0
0	1	0	1	0	0
0	1	1	0	0	1
1	0	0	1	0	0
1	0	1	0	0	1
1	1	0	0	1	1
1	1	1	1	0	1

$$C = \overline{x}y\overline{s} + x\overline{ys} + xy$$
$$V = \overline{xy}s + xy\overline{s}$$

Exercice 6.2

- $n = 8$

51	0011 0011
128	impossible !
-1	1111 1111
-51	1100 1101
-128	1000 0000

- $n = 16$

51	0000 0000 0011 0011
128	0000 0000 1000 0000
-1	1111 1111 1111 1111
-51	1111 1111 1100 1101
-128	1111 1111 1000 0000

Exercice 6.3

30,4	01000001111100110011001100110011
$-0{,}625$	10111101100000000000000000000000
1/3	00111110101010101010101010101010

Exercice 6.4 $-959 \cdot 2^{-105}$.

Exercice 6.5 L'addition flottante est commutative.

Exercice 7.1 Un additionneur n-bits a $2n$ entrées $n+1$ sorties. La seule configuration de sortie que ne contienne aucun 1 est celle de la première ligne correspondant à $0+0=0$. On aura donc $2^{2n}-1$ portes-*non-et* au premier niveau et $n+1$ portes-*non-et* au deuxième niveau pour les $n+1$ sorties ce qui donne $2^{2n}+n$.

Exercice 7.2 $9n$.

Exercice 8.1 Une solution est donnée par la figure B.8.

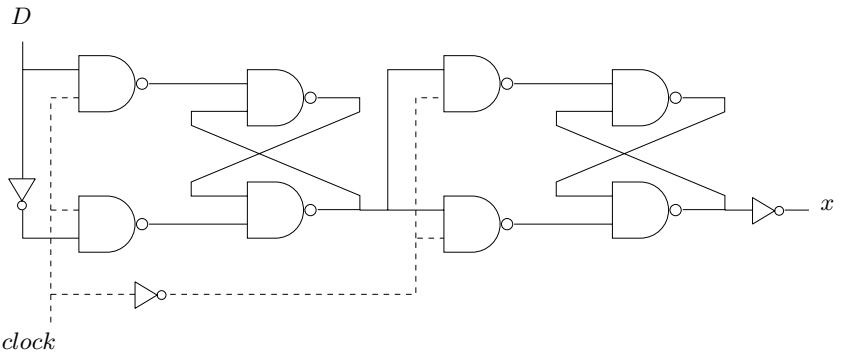


Figure B.8 : Solution de l'exercice 8.1

Exercice 8.2

(1) Le comportement du circuit est présenté par la figure ci-dessous.

(2) Le circuit a exactement le même comportement que le précédent.

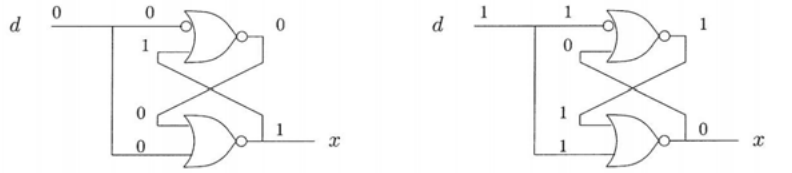


Figure B.9 : Solution de l'exercice 8.2

Exercice 8.3 Ce circuit est un démultiplicateur de fréquence. Chaque élément divise la fréquence du signal en entrée par 2. Si on enchaîne n éléments, la fréquence en sortie est la fréquence en entrée divisée par 2^n.

Exercice 9.1 En utilisant le méthode générale, on obtient un circuit très similaire à celui de la figure 9.4 de la page 75.

Exercice 9.2

La table d'états est donnée par la figure B.10.

Ce circuit décale les valeurs de y_i vers y_{i-1} pour $i \in \{1, 2\}$ et place la valeur de x dans y_2. Le circuit peut être simplifié en supprimant les portes-*non-et*.

x	y_2	y_1	y_0	y_2'	y_1'	y_0'
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	0	1	0
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	0	1	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	1	0	1
1	0	1	1	1	0	1
1	1	0	0	1	1	0
1	1	0	1	1	1	0
1	1	1	0	1	1	1
1	1	1	1	1	1	1

Figure B.10 : Table d'états pour l'exercice 9.2

Exercice 10.1 Il y a 3 états possibles : 2 bistables sont nécessaires. La table d'états est donnée par la figure B.11.

Figure B.11 : Table d'états pour l'exercice 10.1

y_1	y_0	y_1'	y_0'
0	0	0	1
0	1	1	0
1	0	0	0
1	1	-	-

En choisissant la valeur 0 pour les 2 sorties non spécifiées, le circuit passera de l'état (11) à l'état (00). Pour permettre l'initialisation avec une valeur arbitraire, utiliser des multiplexeurs.

Exercice 10.2 Pas d'entrée sauf l'horloge. On pourrait rajouter une entrée pour réinitialiser mais ce n'est pas spécifié. 4 bistables y_3, y_2, y_1, y_0 pour coder 10 états. On numérote les états de manière à ce que y_2, y_1, y_0 correspondent aux valeurs des 3 sorties S_2, S_1, S_0 qui permettent de coder les valeurs de 0 à 5. La table d'états est donnée par la figure B.12.

Figure B.12 : Table d'états pour l'exercice 10.2

y_3	y_2	y_1	y_0	y_3'	y_2'	y_1'	y_0'
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	1	1	0	1
0	1	0	1	-	-	-	-
0	1	1	0	-	-	-	-
0	1	1	1	-	-	-	-
1	0	0	0	-	-	-	-
1	0	0	1	0	0	0	0
1	0	1	0	1	0	0	1
1	0	1	1	1	0	1	0
1	1	0	0	1	0	1	1
1	1	0	1	1	1	0	0
1	1	1	0	-	-	-	-
1	1	1	1	-	-	-	-

Pour aller jusqu'à n, il faut $2n$ états, donc $\lceil \log_2(2n) \rceil$ bistables.

Exercice 10.3 Pas d'entrées sauf l'horloge. 4 bistables y_3, y_2, y_1, y_0 pour coder des valeurs de 0 à 15. La table d'états est donnée par la figure B.13.

n		y_3	y_2	y_1	y_0	y_3'	y_2'	y_1'	y_0'
0	0	0	0	0	0	1	0	1	1
1	11	1	0	1	1	0	1	1	0
2	6	0	1	1	0	0	0	0	1
3	1	0	0	0	1	1	1	0	0
4	12	1	1	0	0	0	1	1	1
5	7	0	1	1	1	0	0	1	0
6	2	0	0	1	0	1	1	0	1
7	13	1	1	0	1	1	0	0	0
8	8	1	0	0	0	0	0	1	1
9	3	0	0	1	1	1	1	1	0
10	14	1	1	1	0	1	0	0	1
11	9	1	0	0	1	0	1	0	0
12	4	0	1	0	0	1	1	1	1
13	15	1	1	1	1	1	0	1	0
14	10	1	0	1	0	0	1	0	1
15	5	0	1	0	1	0	0	0	0

Figure B.13 : Table d'états pour l'exercice **??**

Exercice 11.1 Le multiplexeur est présenté figure B.14 . Il utilise 6 transistors tandis que un multiplexeur réalisé avec des portes logiques utiliserait 14 transistors.

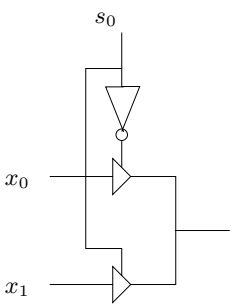


Figure B.14 : Circuit pour l'exercice 11.1

Exercice 11.2 En inversant les flèches dans la figure B.14, nous obtenons un circuit qui ressemble à un multiplexeur mais qui n'en est pas tout à faut un : la sortie qui n'est pas sélectionnée est bloquée or elle devrait sortir 0. En corrigeant ce défaut on obtient le démultiplexeur donné figure B.15.

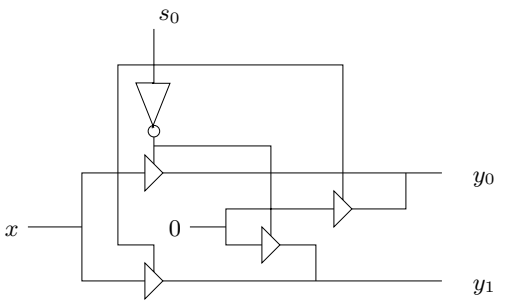


Figure B.15 : Circuit pour l'exercice 11.2

Exercice 11.3 La solution est présentée sur la figure B.16. Pour échanger les contenus de R_i et R_j, on positionne : $ld_i \leftarrow 1$, $ld_j \leftarrow 1$, on positionne les autres ld_k à 0 et finalement $s_i \leftarrow j$ et $s_j \leftarrow i$.

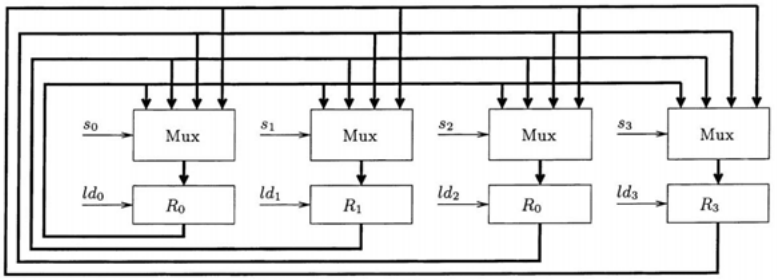


Figure B.16 : Le bus de l'exercice 11.3

Exercice 12.1 Voir la figure B.17.

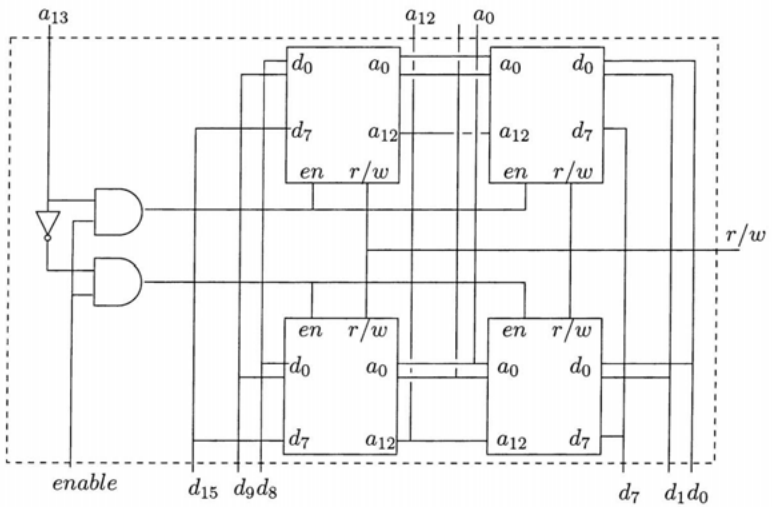


Figure B.17 : Solution de l'exercice 12.1

Exercice 12.2 4 portes.

Exercice 12.3 Voir la figure B.18.

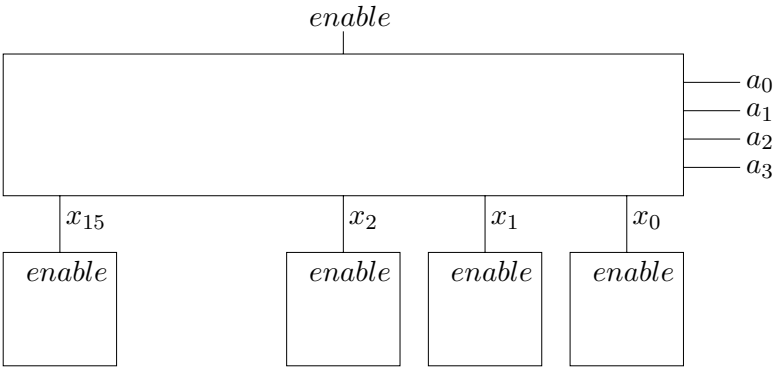


Figure B.18 : Solution de l'exercice 12.3

Exercice 12.4 m portes.

Exercice 14.1 Tous les fils sont de largeur 8 sauf entrée de Instruction-Decoder(5), sortie de Instruction-Decoder (6), entrée et sortie de micro-PC (6).

Exercice 14.2 Circuits combinatoires purs : Instruction-Decoder, ALU, micro-Mem. Circuits contenant des éléments séquentiels : PC, micro-PC, Address-Reg, R0, R1. Circuits contenant des éléments séquentiels et de logique à trois états : Main-Memory. Circuits contenant uniquement des éléments à trois états : Bus Drivers (BD).

Exercice 14.3 On doit éviter [5, 10] (Address Bus) et [8, 3, $\overline{4}$] (Data Bus). La ligne 001000 de la micro-mémoire ne pose pas de problème.

Exercice 14.4

(1) MOP2, 3, 5, 7 :
3, $\overline{4}$: MM est lue ...
5 : ... l'adresse de l'octet lu est celle de PC
2 : ... la donnée lue est décodée puis chargée dans micro-PC
7 : PC est incrémenté

(2) MOP1
micro-PC est mis à 0 (*fetch* en préparation)
PC est inchangé (pour l'instant...).

(3) MOP1, 3, 5, 7, 11
3, $\overline{4}$: MM est lue...
5 : ... l'adresse de l'octet lu est donnée par PC
11 : ... la donnée lue est stockée dans R0
7 : PC est incrémenté
1 : micro-PC mis à 0 (*fetch*...)

Exercice 14.5 2 cycles sont nécessaires :

(1) I1 : 000000011000000 (MOP8, 9)

(2) I2 : 100001000100000 (MOP10, 3, 11)

Exercice 15.1 La solution est donnée figure B.19.

Adresse	Contenu (en base 10)
40	21
41	2
42	4
43	14
44	67
45	1
46	-1 (ou 255)
47	7
48	19
49	21
50	2
51	4
52	19
53	17
54	40
55	23
56	2
57	19
58	21
59	2
60	4
61	19
62	17
63	150
64	23
65	2
66	18
67	1
68	1
69	4
70	18

Figure B.19 : Solution de l'exercice 15.1

Exercice 15.2

(1) Il s'agit de rajouter un bit supplémentaire dans le registre des codes de conditions CC. Ce bit, que nous allons appeler T pour *type* sera mis à 1 si le bit le moins significatif de R0 est égal à 1 *ou* le bit le moins significatif de R1 est égal à 1. Dans la figure 15.1, il faut rajouter une MOP supplémentaire (disons également T) qui sera utilisée par la nouvelle famille d'instructions.

(2) Le premier cycle est identique à celui des instructions de sauts conditionnels habituelles, avec les MOP 3, 5, 7 et 9 pour charger l'argument de l'instruction dans le registre d'adresses. Le deuxième cycle est chargé d'effectuer l'opération arithmétique, donc 12, 14 et 15 pour ADDT et 12 et 13 pour SUBT. Le dernier cycle est chargé d'effectuer le saut si le bit T vaut 1, il est donc similaire au dernier cycle d'une instruction de saut conditionnelle avec les MOP 1, 10 et T.

(3) On peut par exemple supposer que `x`, `y`, et `z` sont des variables locales stockées dans la pile aux adresses 0, 1 et 2 par rapport au pointeur de pile respectivement. On obtient alors le code suivant :

```
lds 0
copy
lds 1
addt error
sts 2
```

Exercice 16.1

Il suffit de remplacer le circuit à trois états par un fil comme c'est indiqué dans la figure B.20.

Exercice 16.2 Il est important qu'une instruction soit *atomique*. Si une instruction peut être interrompue au cours de son exécution, on ne peut pas savoir si l'effet souhaité a été effectué ou non. Dans la cas d'une opération arithmétique, par exemple, on ne peut pas savoir si le registre R1 contient le résultat de l'opération, ou alors s'il faut réexécuter l'instruction après traitement de l'interruption.

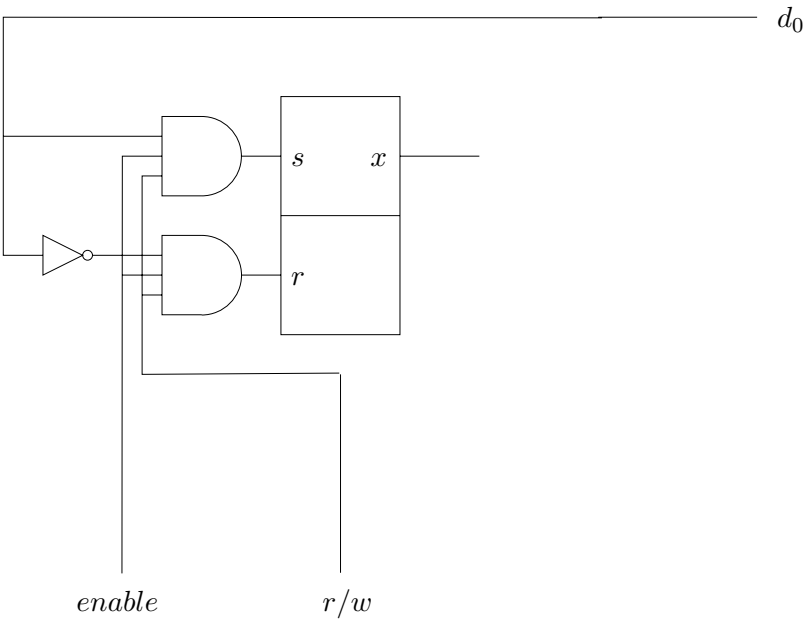


Figure B.20 : Solution de l'exercice 16.1

Exercice 17.1

(1) 2^7 octets.

(2) zone (25), ligne (4), octet (3).

(3) $8 * 2^7 + 25 * 2^4$ bits.

(4) S : succès, D : Défaut
$1D$, $6D$, $8D$, $32D$,$7S$, $33S$, $1D$, $39D$, $27D$, $59D$, $58S$, $64D$, $97D$, $1D$.

Exercice 17.2

(1) 2^7 octets.

(2) bloc (29), octet (3).

(3) $8 * 2^7 + 29 * 2^4$.

(4) S : succès, D : Défaut
$1D$, $6D$, $8D$, $32D$,$7S$, $33S$, $1S$, $39D$, $27D$, $59D$, $58S$, $64D$, $97D$, $1S$.

Exercice 17.3

(1) 2^7 octets.

(2) zone (26), ligne (3), octet (3).

(3) $8 * 2^7 + 26 * 2^4$ bits.

(4) S : succès, D : Défaut
$1D$, $6D$, $8D$, $32D$,$7S$, $33S$, $1S$, $39D$, $27D$, $59D$, $58S$, $64D$, $97D$, $1D$.

Exercice 18.1 1 M $= 2^{20}$ ce qui fait que $2^{64} = 2^{44}$ M. Puisque $2^{10} \approx 10^3$, on a que $2^{44} \approx 16 \cdot 10^{12}$ pour un coût total d'environ $1{,}6 \cdot 10^{12}$ euros. Pour apprécier une telle valeur, il convient parfois de diviser par un grand nombre, tel que la population de la terre, soit environ 10^{10}. Pour remplir un seul ordinateur de ce type de RAM, il faut donc 160 euros pour chaque personne de la planète.

Exercice 18.2 4 K $= 2^{12}$ ce qui donne une table à 2^{52} entrées. Si chaque entrée nécessite $8 = 2^3$ octets, une telle table des page nécessiterait 2^{55} octets, ce qui est totalement inpensable.

Exercice 18.3 Pour une taille de page de 4 Ko, les 12 derniers bits indiquent le numéro d'octet à l'intérieur de la page. Il reste donc 54 bits à répartir dans les deux niveaux. Soit n le nombre de bits consacrés au premier niveau (et donc $54 - n$ le nombre de bits consacrés au deuxième niveau). Le bloc du premier niveau contient donc 2^n entrées. Il faut au moins un bloc du deuxième niveau (on estime que le programme a besoin d'au moins une page de code et de données). Le bloc du deuxième niveau contient 2^{54-n} entrées et le nombre d'entrées est donc la somme des deux, soit $2^n + 2^{54-n}$. Pour trouver la valeur de n qui minimise cette expression, le plus simple est de tester toutes les valeurs possibles, ce qui donne $n = 27$ pour une table d'environ 250 millions d'entrées, soit environ 1 Go ce qui est toujours trop élevé.

Exercice 18.4 Indication : suivre la même démarche que dans l'exercice précédent pour un nombre de niveaux de 3, 4, etc.

Exercice 19.1 Soit t le taux de défauts de page. On obtient l'équation :

$$(1-t)\cdot 10^{-8}s + t\cdot 10^{-3}s = 2\cdot 10^{-8}s$$

ce qui donne $t \approx 10^{-5}$. L'interprétation de cette valeur est que au plus un accès sur 100 000 peut provoquer un défaut de page.

SCIENCES SUP

Robert Strandh, Irène Durand

ARCHITECTURE DE L'ORDINATEUR

Portes logiques, circuits combinatoires, arithmétique binaire, circuits séquentiels et mémoires. Exemple d'architecture.

Cet ouvrage s'adresse aux futurs informaticiens.

Son objectif n'est pas de rentrer dans les subtilités de l'architecture de tel ou tel type de processeur, mais de donner les connaissances sur le fonctionnement d'un ordinateur qui permettront à l'informaticien d'optimiser l'efficacité d'un programme et d'anticiper l'impact d'une modification sur la performance de ce programme.

La première partie explique les circuits combinatoires, les circuits séquentiels et les mémoires.

La seconde partie décrit un exemple d'architecture simple mais complète.

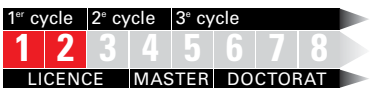
La troisième partie introduit des notions comme la mémoire cache, l'adressage virtuel et la multiprogrammation.

Ce cours synthétique est accompagné de 50 exercices corrigés.

ROBERT STRANDH
est professeur
à l'université Bordeaux 1.

IRÈNE DURAND
est maître de conférences
à l'université Bordeaux 1.

MATHÉMATIQUES
PHYSIQUE
CHIMIE
SCIENCES DE L'INGÉNIEUR
INFORMATIQUE
SCIENCES DE LA VIE
SCIENCES DE LA TERRE



ISBN 2 10 049214 4

www.dunod.com

DUNOD