UNIVERSITÉ DE FRANCHE-COMTÉ

ct u

Licence 3

Informatique

Architecture des Systèmes Informatiques

Didier Teifreto

didier.teifreto@univ-fcomte.fr

25 juin 2014 à 08:50

UFC centre de télé-enseignement universitaire

Cette brochure a été réalisée en LaTeX 2_ε.

Table des matières

I Codage et manipulation de l'information **1**

1 Introduction **5**

2 Codage et manipulation des nombres entiers non signés ($\in \mathbb{N}$) **9**

3 Codage des nombres entiers signés ($\in \mathbb{Z}$) **63**

4 Codage des nombres réels ($\in \mathbb{R}$) **79**

II Programmation et Architecture des microprocesseurs MIPS **103**

5 Introduction **107**

6 La programmation en langage machine du MIPS **109**

7 Architecture des processeurs MIPS **143**

8 Système d'interruptions et d'entrées/sorties **167**

III Accélération des processeurs **183**

9 La hiérarchie mémoire **187**

10 Le pipeline d'instructions **205**

11 Additions rapides **219**

IV Annexe **229**

Table des figures **231**

Liste des algorithmes **235**

Listings **238**

Première partie

Codage et manipulation de l'information

Bibliographie :

1. **Architecture des Ordinateurs EdiSciences Nicholas P. Carters** pages 21 à 55
2. **Architecture logicielles et matérielles : Cours, études de cas et exercices corrigés DUNOD Ouvrage collectif** Chapitre 3 pages 47 à 72
3. **Organisation et conception des ordinateurs : L'interface matériel/logiciel DUNOD D Patterson J Hennessy** Chapitre 4.1 à 4.4 et 4.8
4. **Architecture des Ordinateur : Une approche Quantitative ITP J Hennessy A Patterson** Annexe A.2 et A.3
5. **Architecture des Ordinateur EdiSciences Nicholas P. Carters** pages 21 à 55

Webographie :

1. Système binaire `://fr.wikipedia.org/wiki/Système_binaire`
2. Convertisseur de nombres entiers non signés `http://calc.50x.eu/`
3. Codage des nombres réels au format IEEE 754 `http://fr.wikipedia.org/wiki/IEEE_754`
4. Convertisseur de nombres réels au format IEEE 754 `http://www.h-schmidt.net/FloatApplet/IEEE754.html`
5. Autres convertisseurs `http://www.binaryconvert.com/convert_signed_short.html`
6. En faire plus : Utilisation avancée des opérateurs bit à bit `http://graphics.stanford.edu/~seander/bithacks.html`

Chapitre 1

Introduction

Dans cette partie, nous allons étudier la représentation et le codage des nombres dans un système informatique. Nous allons commencer avec les nombres entiers non signés (positifs), puis les nombres entiers signés (positifs ou négatifs) et pour terminer nous étudierons le codage des nombres réels. Ceci est nécessaire avant d'aborder la conception logicielle et matérielle de l'ordinateur. Le matériel et le logiciel ne sont que des supports de traitement de l'information.

1.1 Principe

Dans un ordinateur, l'information manipulée est codée en binaire. Une variable binaire peut prendre deux états : **VRAI** (ou 1) et **FAUX** (ou 0) . Elle est aussi nommée **bit** contraction de **contraction de BInary digiT**. Un digit (acronyme de "DIGital unIT") est un élément d'information numérique.

Les informations plus complexes sont codées sous la forme d'un ensemble de bits. Le codage des informations sous forme de bits est très fiable. Chaque bit est codé sous forme de deux tensions électriques

La valeur 0 est codée avec une tension comprise entre 0 et 0.4 Volts
La valeur 1 est codée avec une tension comprise entre 1.4 et 1.8V

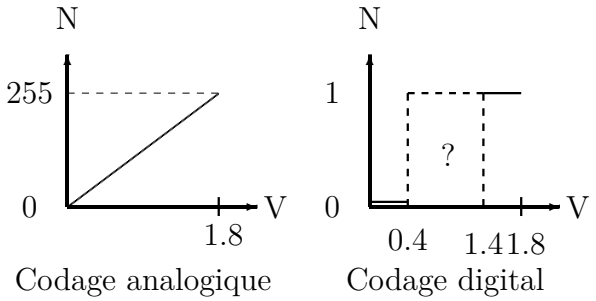


FIGURE 1.1 – Codage analogique et digital

La figure FIG. 1.1 illustre les deux méthodes de codage. Le codage analogique d'une valeur est soumis à une grande incertitude. En effet, une petite variation de la valeur physique (ici V) à une incidence sur la valeur représentée (ici N). Le codage digital ou numérique est moins sensible aux variation de la valeur physique. Un ensemble de valeurs physique représente une valeur unique 0 ou 1.

1.2 Groupement de bits

Dans toute la suite de cette partie du cours, nous allons nous intéresser au regroupement des bits qui permettront de représenter des informations complexes (caractères, entiers, réels) Quelle que soit la nature de la donnée représentée (entier signé ou non signé, réel ...), le codage est toujours le même ; **c'est seulement l'interprétation qui est différente** Les bits sont regroupés dans les ordinateurs par ensemble de bits. La figure FIG. 1.2 spécifie les noms donnés aux ensembles de bits .

Taille de l'ensemble	*Nom de l'ensemble*
4 bits	Digit hexadécimal
8 bits	Octet ou Byte
16 bits	Demi-Mot ou HalfWord
32 bits	Mot ou Word
64 bits	Double Mot ou un Double Word
128 bits	Quadruple mot ou Quad Word

FIGURE 1.2 – Noms des ensembles de bits

Exemple 1 :

Un ordinateur ayant une mémoire de 4 Giga Octets de mémoire contient :

$$\begin{aligned} 4 \times 8 &= 32 \text{ Giga bits de mémoire} \\ \frac{4}{4} &= 1 \text{ Giga mots de mémoire} \\ \frac{4}{16} &= 0,25 \text{ Giga quadruple mots de mémoire} \end{aligned}$$

1.3 Portes logiques

Pour manipuler les bits d'une donnée nous utiliserons de portes logiques. Nous disposons de celles représentées sur la figure FIG 1.3 .

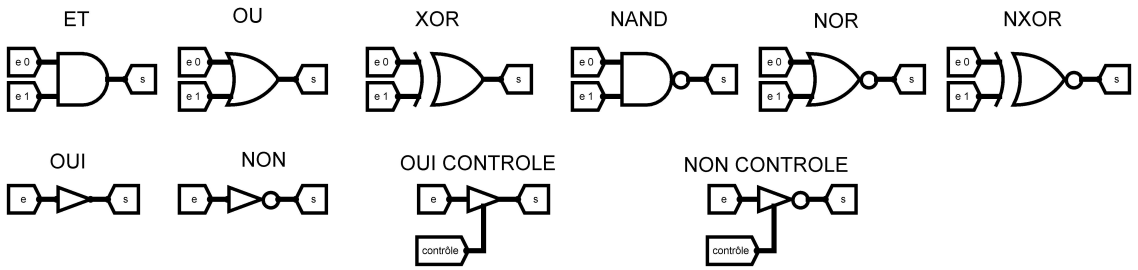


FIGURE 1.3 – Portes logiques

1.3.1 Fonction booléenne à une variable

La seule fonction intéressante est l'inversion logique de la variable **a** notée $\overline{\mathbf{a}}$ (notée aussi **a** /) appelée fonction NON. La table de vérité de cette fonction est présentée sur la figure FIG 1.4 .

a	$\overline{\mathbf{a}}$
0	1
1	0

FIGURE 1.4 – Fonction Non

Remarque 1 :
Il existe aussi un fonction nommée OUI donc l'état de la sortie est celui de l'entrée. Nous utiliserons la fonction contrôlée.

1.3.2 Fonctions à deux variables

Sur les 16 fonctions possibles avec 2 variables **a** et **b**, seules 6 sont intéressantes. Les tables de vérité de ces fonctions sont présentées sur la figure FIG 1.5 .

a	**b**	**a** + **b**	$\overline{\mathbf{a} + \mathbf{b}}$	**a** . **b**	$\overline{\mathbf{a} . \mathbf{b}}$	**a** $\oplus$ **b**	$\overline{\mathbf{a} \oplus \mathbf{b}}$
		OR	NOR	AND	NAND	XOR	NXOR
0	0	0	1	0	1	0	1
0	1	1	0	0	1	1	0
1	0	1	0	0	1	1	0
1	1	1	0	1	0	0	1

FIGURE 1.5 – Fonctions logiques

En pratique, les circuits des ordinateurs sont réalisées avec des portes identiques. Ce type de porte permet de réaliser touts les autres types. Il doit être inverseur et avoir plusieurs entrées. Si nous choisissons le type NAND nous pouvons créer les autres types de la façon suivante :

$$\begin{aligned} \overline{a} &= \overline{a.a} = \overline{a.1} \\ a.b &= \overline{\overline{a.b}.1} \\ a+b &= \overline{\overline{a+b}} = \overline{\overline{a.1}.\overline{b.1}} \end{aligned}$$

- Porte NON : une porte NAND
- Porte ET : deux portes NAND
- Porte OU deux entrées avec trois portes NAND.

La figure FIG 1.6 illustre la réalisation d'un système complet utilisant des portes logiques NAND.

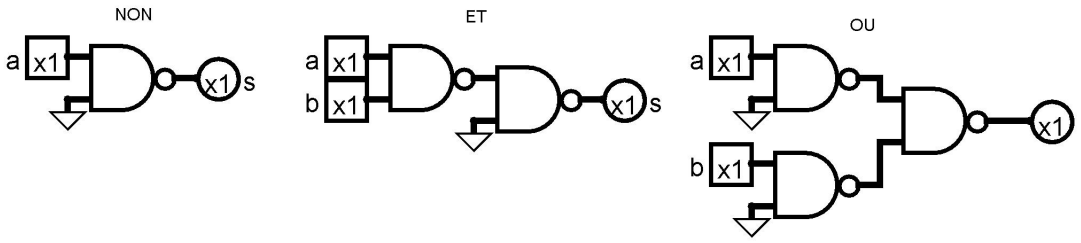


FIGURE 1.6 – Portes logiques réalisées avec des NAND

Remarque 2 :
Le symbole formé d'un triangle représente le niveau logique 1.

1.3.3 Portes logiques contrôlées

Nous remarquons que certaine portes logiques ont une entrée supplémentaire nommé contrôle. Lorsque celle ci est à 0, la sortie est dans un état indéfini noté X (la sortie n'est ni à 0, ni à 1). Ces portes s'appellent 3 états.
Ceci permet de connecter entre elles des sorties, si et seulement si une entrée de contrôle est à 1. Ceci n'est pas possible avec des portes non contrôlées.

Exemple 2 :

La figure Fig 1.7 présente deux portes OUI contrôlées donc les sorties sont reliées ensemble.

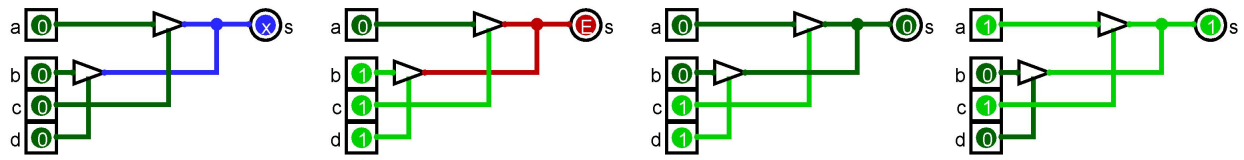


Figure 1.7 – Fonctions logiques

— Si les deux entrées **c** et **d** sont à 0, la sortie n'est dans aucun état, noté X
— Si les deux entrées **c** et **d** sont à 1 et si les entrées **a** et **b** sont différentes alors nous avons un erreur notée E,
— Si les deux entrées **c** et **d** sont à 1 et si les entrées **a** et **b** sont identiques alors nous n'avons pas d'erreur.
— Si les deux entrées **c** =1 et **d** =0 l'état de **a** et placée sur la sortie.

Chapitre 2

Codage et manipulation des nombres entiers non signés ($\in \mathbb{N}$)

2.1 Base

Un nombre entier non signé, codé dans la base b, est un ensemble de digits compris entre 0 et $b-1$; en base 10 (décimale) un nombre est un ensemble de chiffres compris entre 0 et 9 ; en base 2 (binaire) un nombre est un ensemble de bits valant 0 ou 1.
Un nombre non signé en base b, codé sur n digits est représenté par une suite de digits notés d_i :

$$d_{n-1}d_{n-2}....d_i...d_1d_0 \qquad \text{avec } 0 \leq d_i \leq b-1 \tag{2.1}$$

Remarque 3 :

- Dans le cas où une donnée est représentée en décimale, la base ne sera pas mentionnée dans sa représentation.
- Dans le cas du binaire (base 2), la représentation sera préfixée par **0b**. Les bits sont groupés par ensemble de 4.
- Dans le cas de l'octal (base 8), la représentation sera préfixée par le **0**.
- Dans le cas de hexadécimale (base 16), le représentation sera préfixée par **0x**. Les digits sont groupés par ensemble de 4.

Exemple 3 :

0b1000 0100	est une représentation binaire
01234567	est une représentation octale
0x2345 67AB	est une représentation hexadécimale
1234567	est une représentation décimale

2.2 Conversion en une valeur décimale

Un nombre non signé représenté en base b, aura en décimale la valeur donnée par l'évaluation du polynôme suivant :

$$\text{N} = d_{n-1} \times b^{n-1} + d_{n-2} \times b^{n-2} + ...d_i \times b^i + d_1 \times b^1 + d_0 \times b^0 \quad (2.2)$$

avec $i \subset [0, n-1]$ nommé le rang du chiffre
et b^i nommé poids du chiffre de rang i.

Remarque 4 :

— Le digit le plus à gauche est appelé le **digit de poids fort** et celui le plus à droite le **digit de poids faible**. En binaire, le bit de gauche est nommé le **MSB** (most significant bit) et celui de droite est nommé **LSB** (less significant bit) .

— Les poids décimaux des bits de droite à gauche sont : $2^0 = 1, 2^1 = 2, 2^2 = 4, 2^3 = 8, 2^4 = 16, 2^5 = 32, 2^6 = 64, 2^7 = 128, 2^8 = 256, 2^9 = 512, 2^{10} = 1024, 2^{11} = 2048, 2^{12} = 4096, 2^{13} = 8192, 2^{14} = 16384, 2^{15} = 32768, 2^{16} = 65536$ ....

— Connaître ces valeurs est indispensable pour la suite du cours. En effet nous manipulerons beaucoup les puissances de deux.

Exemple 4 :

$$\begin{aligned} \textbf{0b}1011 &= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 8 + 2 + 1 = 11 \end{aligned}$$

2.3 Étendue des valeurs des nombres non signés

1. La **Valeur maximale** est obtenue dans le cas ou les n digits ont pour valeur $(b-1)$.

$$\begin{aligned} \text{Nmax} &= (b-1) \times b^{n-1} + (b-1) \times b^{n-2} + ...(b-1) \times b^i ... + (b-1) \times b^1 + (b-1) \times b^0 \\ &= (b-1) \times (b^{n-1} + b^{n-2} + ...b^i ... + b^1 + 1) \\ &= b \times (b^{n-1} + b^{n-2} + ...b^i ... + b^1 + 1) - 1 \times (b^{n-1} + b^{n-2} + ...b^i ... + b^1 + 1) \\ &= b^n + b^{n-1} + ...b^i ... + b^1 - 1 \times (b^{n-1} + b^{n-2} + ...b^i ... + b^1 + 1) \\ &= b^n - 1 \quad (2.3) \end{aligned}$$

2. La **Valeur minimale** est obtenue dans le cas ou les n chiffres ont pour valeur 0.

$$\begin{aligned} \text{Nmin} &= 0 \times b^{n-1} + 0 \times b^{n-2} + ...0 \times b^i ... + 0) \times b^1 + 0 \times b^0 \\ &= 0 \quad (2.4) \end{aligned}$$

3. Le **nombre de valeurs non signées** qu'un entier non signé codé en base b sur n digits permet de représenter est :

$$\begin{aligned} \text{Nval} &= (b^n - 1) + 1 \text{ pour la valeur } 0 \\ &= b^n \quad (2.5) \end{aligned}$$

Remarque 5 :

— Dans un ordinateur les nombres sont codés 8, 16 ou 32 bits. La figure FIG. 2.1 présente la valeur maximale représentée pour chaque cas.

Nombre de bits	Valeur minimale	Valeur maximale
8	0	$2^8 - 1 = 255$
16	0	$2^{16} - 1 = 65535$
32	0	$2^{32} - 1 = 4294967295$

FIGURE 2.1 – Valeurs maximales des nombres non signés

— L'informatique utilise les notions de Kilo, Méga, Giga et Téra . Dans le monde décimal un Kilo vaut 1000 (10^3) mais en binaire, 1000 n'est pas une valeur maximale associée à un nombre de bits. La valeur immédiatement supérieure sera utilisée.

Pour le kilo $2^9 = 512$ et $2^{10} = 1024$. Il faut donc 10 bits pour coder un Kilo (de 0 à 1023)

Pour le Méga $2^{20} = 1024 \times 1024 = 1\,048\,576$. Il faut donc 20 bits pour coder un Méga (de 0 à 1048575)

Pour le Giga $2^{30} = 1\,048\,576 \times 1024 = 1\,073\,741\,824$. Il faut donc 30 bits pour coder un Giga (de 0 à 1 073 741 823)

— Si nous raisonnons en Kilo, une information codée sur 16 bits permet de représenter :

$$\begin{aligned} 2^{16} &= 2^{10+6} \\ &= 2^6 \times 2^{10} \\ &= 64 \text{ Kilo} \end{aligned} \tag{2.6}$$

— Si nous raisonnons en Giga, une information codée sur 30 bits permet de représenter :

$$\begin{aligned} 2^{30} &= 2^{10} \times 2^{20} \\ &= 1024 \times 2^{20} \\ &= 1024 \text{ Méga} \end{aligned} \tag{2.7}$$

— Si nous raisonnons en Giga, une information codée sur 32 bits permet de représenter :

$$\begin{aligned} 2^{32} &= 2^{30+2} \\ &= 2^2 \times 2^{30} \\ &= 4 \text{ Giga} \end{aligned} \tag{2.8}$$

2.4 Codage Hexadécimal

Pour améliorer la lisibilité des informations, l'informatique utilise la base 16, appelée base hexadécimale. Les digits que nous utiliserons seront compris entre 0 et 9, puis A pour 10, B pour 11, C pour 12, D pour 13, E pour 14 et F pour 15. Ceci permet d'obtenir les 16 symboles hexadécimaux.
Dans un ordinateur les nombres sont codés sur 2, 4 ou 8 digits hexadécimaux. La figure FIG. 2.2 présente les valeurs maximales pour chaque cas.

Nombre de digits	Valeur minimale	Valeur maximale
2	0	$2^8 - 1 = 255$
4	0	$2^{16} - 1 = 65535 = 64K - 1$
8	0	$2^{32} - 1 = 4294967295 = 4G - 1$

FIGURE 2.2 – Valeurs maximales des nombres hexadécimaux non signés

Les valeurs maximales sont les mêmes que pour 8, 16 et 32 bits. Un digit représente donc l'équivalent de 4 bits (en effet $16 = 2^4$).

$$\begin{aligned} N &= d_{n-1} \times 16^{n-1} + d_i \times 16^i ... + d_1 \times 16 + d_0 \\ &= d_{n-1} \times 2^{4\times(n-1)} + d_i \times 2^{4*i} ... + d_1 \times 2^4 + d_0 \end{aligned} \tag{2.9}$$

La figure FIG. 2.3 représente sur un octet, les poids binaire et hexadécimal.

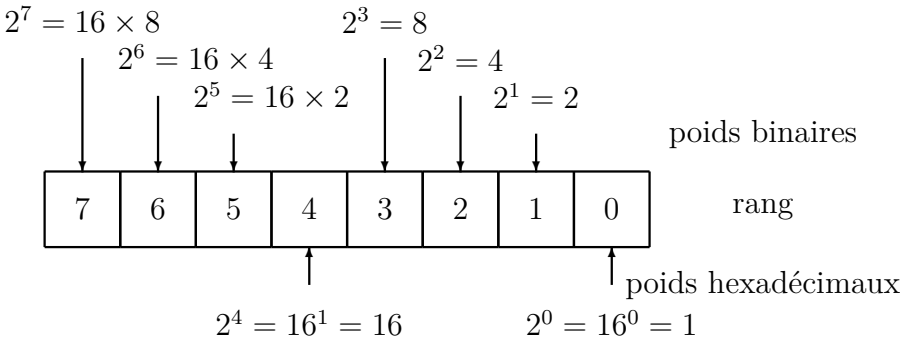


FIGURE 2.3 – Poids binaires et hexadécimaux

2.5 Conversion d'un nombre représenté en décimal en un nombre représenté en base b

L'algorithme des divisions successives présenté avec l'ALGORITHME. 1 permet de convertir un nombre codé en décimal en un nombre codé en base b. Cet algorithme utilise le quotient (noté /) et le reste (note mod) nommé aussi modulo) de la division entière . L'algorithme calcule le digit de poids faible en premier et le digit de poids fort en dernier. Ainsi, il sera possible d'afficher le nombre codé en base b, en inversant l'ordre des chiffres obtenus.

Algorithme 1 Divisions successives

```
1: Variables
2:    nombre, base : entier
3: Début
4: Lire (nombre, nase)
5: Répéter
6:    Afficher (nombre mod base)
7:    nombre ← nombre / base
8: Jusqu'à Nombre = 0
9: Fin
```

Remarque 6 :

- Pour convertir un nombre représenté dans une base $b1$ vers une base $b2$, il suffit de convertir en décimal (évaluation du polynôme) puis de le convertir dans sa représentation dans une base $b2$ en utilisant l'algorithme des divisions successives.
- Pour passer d'une valeur binaire à une valeur hexadécimale, il suffit de grouper les bits par paquets de 4, puis de trouver la valeur associée correspondante. Par exemple si une donnée d'un programme se trouve à l'adresse **0x**5ABCD, la valeur binaire correspondante est **0b**0101 1010 1011 1100 1101

— Les données sont **toujours stockées en binaire** dans l'ordinateur, l'affichage ou la saisie des données sera effectuée dans une autre base (décimale ou hexadécimale). Des fonctions étudiées en exercices se chargeront de ces conversions.

Exemple 5 :

Conversion du nombre 11 en binaire
La figure FIG. 2.4 illustre la conversion de ce nombre de la base décimale dans la base binaire.

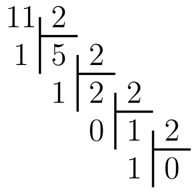


FIGURE 2.4 – Divisions successives en binaire

$$\begin{aligned}
\frac{11}{2} &= 5 \text{ reste } 1 \text{ - le poids de ce bit vaut } 1\\
\frac{5}{2} &= 2 \text{ reste } 1 \text{ - le poids de ce bit vaut } 2\\
\frac{2}{2} &= 1 \text{ reste } 0 \text{ - le poids de ce bit vaut } 4\\
\frac{1}{2} &= 0 \text{ reste } 1 \text{ - le poids de ce bit vaut } 8\\
11 &= \mathbf{0b}1011\\
\mathbf{0b}1011 &= 1\times 2^3 + 0\times 2^2 + 1\times 2^1 + 1\times 2^0\\
\mathbf{0b}1011 &= 11 \text{ (pour vérification)}
\end{aligned}$$

Exemple 6 :

La conversion des nombres compris entre 0 et 15 donne la table présentée sur la figure FIG 2.5 .

Nombre décimal	*Nombre hexadécimal*	b_3	$b2$	b_1	b_0
0	0	0	0	0	0
1	1	0	0	0	1
2	2	0	0	1	0
3	3	0	0	1	1
4	4	0	1	0	0
5	5	0	1	0	1
6	6	0	1	1	0
7	7	0	1	1	1
8	8	1	0	0	0
9	9	1	0	0	1
10	A	1	0	1	0
11	B	1	0	1	1
12	C	1	1	0	0
13	D	1	1	0	1
14	E	1	1	1	0
15	F	1	1	1	1

FIGURE 2.5 – Table de conversion hexadécimale en binaire

Exemple 7 :

Conversion du nombre 1020 en hexadécimal et en binaire :

$$\begin{aligned}
\frac{1020}{16} &= 63 \text{ reste } 12 \text{ soit } \mathbf{0x}\text{C en hexadécimal (le poids de ce digit est 1)}\\
\frac{0063}{16} &= 03 \text{ reste } 15 \text{ soit } \mathbf{0x}\text{F en hexadécimal(le poids de ce digit est 16)}\\
\frac{0003}{16} &= 00 \text{ reste } 03 \text{ (le poids de ce digit est 256)}\\
\mathbf{0x}3\text{FC} &= 3\times 16^2 + 15\times 16^1 + 12\times 16^0 = 1020 \text{ pour vérification}\\
\mathbf{0x}3\text{FC} &= \mathbf{0b}0011\,1111\,1100
\end{aligned}$$

2.6 Opérations binaires sur les nombres non signés

Les nombres étant stocké en binaire, les calculs sur les nombres non signés sont effectués exclusivement en binaire.

2.6.1 Addition

Addition de deux bits L'addition la plus simple est celle de deux bits, présentée sur la figure Fig 2.6 (a). Nous remarquons que l'addition de deux bits produit un résultat sur deux bits.

Addition de trois bits L'addition de 3 bits présentée sur la figure Fig 2.6 (b) permet l'addition deux bits et d'un bit de retenue. Nous remarquons que l'addition de trois bits produit un résultat sur deux bits. Le bit représenté de gauche est nommé **retenue non signée** de l'addition , celui de droite est nommé résultat.

+	**0**	**1**
0	**0**0	**0**1
1	**0**1	**1**0

(a) Addition de 2 bits

+	$0+0=\mathbf{0}0$	$0+1=1+0=\mathbf{0}1$	$1+1=\mathbf{1}0$
0	**0**0	**0**1	**1**0
1	**0**1	**1**0	**1**1

(b) Addition de 3 bits

Figure 2.6 – Addition de deux et de trois bits

Remarque 7 :

$$1+1 = 10_2 = 1\times 2^1 + 0\times 2^0 = 2 \tag{2.10}$$

$$1+1+1 = 11_2 = 1\times 2^1 + 1\times 2^0 = 3 \tag{2.11}$$

Addition de $2\times n$ bits Lors de l'addition de deux informations codées sur n bits, il y a propagation de la retenue générée par l'addition des deux bits et de la retenue venant du rang précédent, depuis le **LSB** jusqu'au **MSB**. La retenue initiale placée sur le **LSB** est fixée à 0. La figure Fig. 2.7 illustre ce principe.

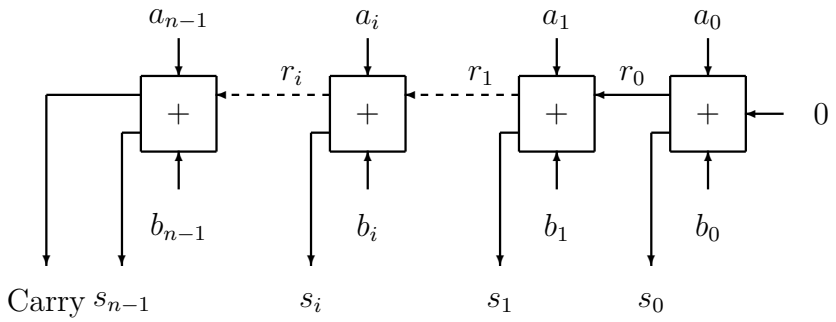


FIGURE 2.7 – Principe de l'addition à propagation de retenue

Exemple 8 :

La figure FIG. 2.8 présente deux additions binaires et la propagation de la retenue.

	0	1	0	1	1	1	0	0	92		1	1	1	1	1	1	1	1	255
+	0	1	1	0	1	0	1	0	+106	+	1	1	1	1	1	1	1	1	+255
	1	1	1	1	0	0	0	0	(retenue)		1	1	1	1	1	1	1	0	(retenue)
0	1	1	0	0	0	1	1	0	198	1	1	1	1	1	1	1	1	0	510

(a) sans retenue (b) avec retenue

FIGURE 2.8 – Addition binaire de deux nombres non signés

L'addition de deux nombres codés sur n bits génère quelquefois un résultat sur $n+1$ bits comme l'illustre la figure FIG. 2.8 (b). Le bit supplémentaire (ici en **gras**) est nommé **retenue non signée** et se trouve dans tous les microprocesseurs, souvent sous le nom générique de **CARRY**.

Exemple 9 :

La figure FIG. 2.9 présente l'addition de deux nombres hexadécimaux sans et avec retenue (CARRY ou ici c_{out}).

	5	C	92		F	F	255
+	6	A	+106	+	F	F	+255
	1	0	(retenue)		1	0	(retenue)
0	C	6	198	1	F	E	510

(a) sans retenue (b) avec retenue

FIGURE 2.9 – Addition hexadécimale de deux nombres non signés

Remarque 8 :

La retenue passant entre les deux digits est nommée **demi-retenue non signée** (Auxiliary Carry). Elle est utilisée dans la technique du DCB (décimal codé binaire) . Dans ce cas, le nombre est représenté directement en décimal au niveau des bits. Cette technique n'est plus utilisée de nos jours.

Circuit d'addition Pour concevoir un circuit, nous devons le modéliser. Pour cela nous utiliserons une table de vérité. Nous obtiendrons alors une équation et nous pourrons alors tracer le schéma électrique du circuit.

1. Demi-additionneur (Addition de deux bits)

   La table de vérité de la figure FIG 2.10 présente toutes les combinaisons possibles des 2 entrées (soit 2^2 lignes) et l'état des sorties correspondantes. Une fois la table de vérité écrite, nous devons la transformer en une équation logique. C'est une somme (fonction ou) de produits (et).

   Lorsque nous additionnons deux informations codées sur un bit a et b, nous générons un résultat nommé r et une retenue sortante nommée c_{out}.

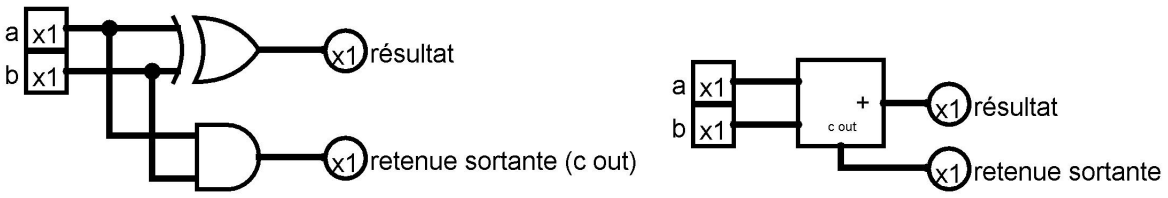
a	b	c_{out}	r
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

FIGURE 2.10 – Table de vérité de l'addition 2 bits

$$c_{out} = a.b \tag{2.12}$$

$$r = \overline{a}b + a\overline{b} = a \oplus b \tag{2.13}$$

La figure FIG 2.11 représente le schéma électrique du demi-additionneur. Dans le schéma le carré contenant x1 indique que les données a et b sont des entrées un bit et le rond x1 indique que c_{out} et r sont des sorties un bit.



(a) Vue interne (b) Vue externe

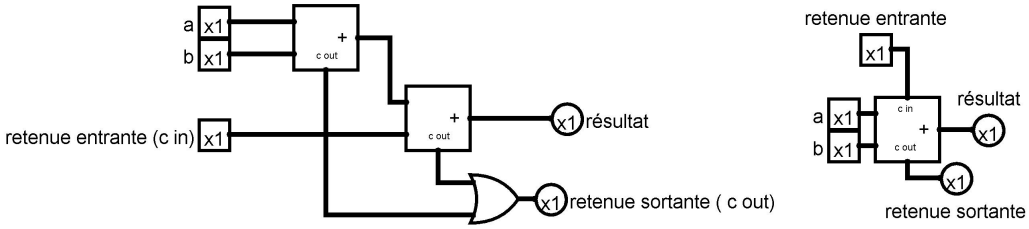
FIGURE 2.11 – Schéma du demi-additionneur

Remarque 9 :

— La vue interne représente les portes formant le demi-additionneur,
— La vue externe représente une abstraction du demi-additionneur. Nous réutiliserons cette vue dans le circuit suivant.

2. Additionneur complet (addition 3 bits)

   L'additionneur complet peut être réalisée à partir de deux demi-additionneurs comme le représente la FIG 2.12 .



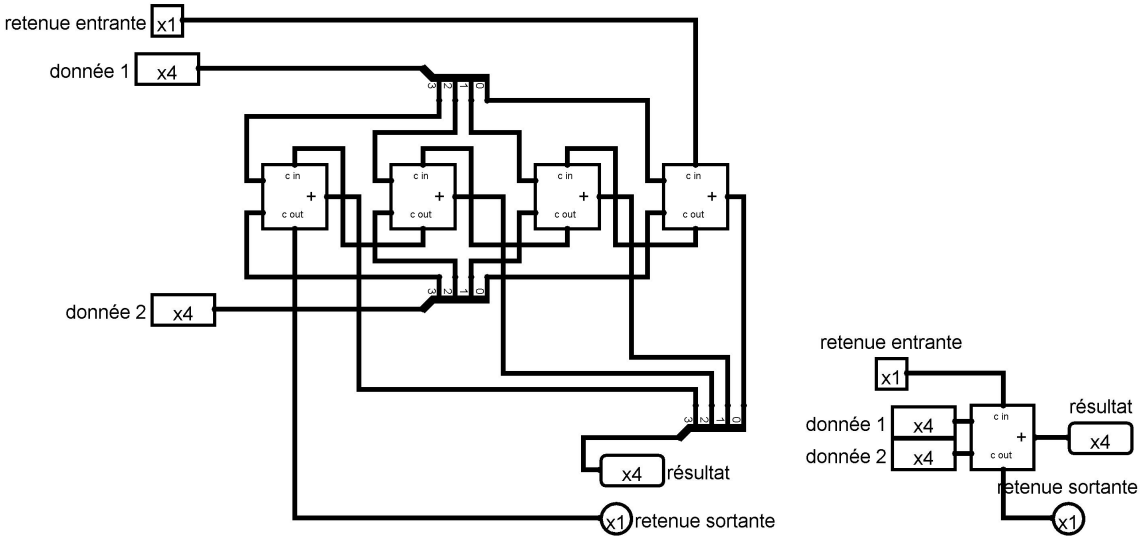
(a) Vue interne (b) Vue externe

FIGURE 2.12 – Schéma de l'additionneur complet

Remarque 10 :

La retenue c_{out} est générée par l'additionneur complet si elle l'est par un des deux demi-additionneur, le résultat est la somme de a et b du premier demi-additionneur et de c_{out}.

3. Addition de 2×4 bits
   L'addition de 2×4 bits peut être réalisée à partir de quatre additionneurs complets comme le représente la FIG 2.13 .



(a) Vue interne (b) Vue externe

FIGURE 2.13 – Schéma de l'additionneur 2×4 bits

Remarque 11 :

Les splitter permette de décomposer la donnée quatre bits en quatre données un bit, qui seront utilisé par les additionneurs complets.

2.6.2 Opérateur non bit à bit, noté $\sim$

Cette opération permet d'inverser (complément à 1) les bits d'une donnée.Les bits représentés en rouge sur la figure FIG 2.14 changent.

253	1	1	1	1	1	1	0	1
$\sim$ (253)=2	0	0	0	0	0	0	1	0

0xAF	1	0	1	0	1	1	1	1
$\sim$ **0x**AF =**0x**50	0	1	0	1	0	0	0	0

FIGURE 2.14 – Opérateur non bit à bit

Lors de cette opération, les bits à 0 se trouvant à gauche de la donnée sont aussi inversés.

Exemple 10 :

$$\sim \mathbf{0x}\text{AF} = \mathbf{0x}50 \text{ au format 8 bits} \tag{2.14}$$

$$\sim \mathbf{0x}00\text{AF} = \mathbf{0x}\text{FF}50 \text{ au format 16 bits} \tag{2.15}$$

$$\sim \mathbf{0x}0000\,00\text{AF} = \mathbf{0x}\text{FFFF}\,\text{FF}50 \text{ au format 32 bits} \tag{2.16}$$

2.6.3 Opérateur et bit à bit, noté &

Pour tester un bit (ou un ensemble de bits) d'une donnée, la technique utilisée est appelée **masquage**. Nous appliquons une fonction et bit à bit à la donnée et au masque. Il faut que les deux bits soient simultanément à 1 pour obtenir un bit résultat à 1.

Exemple 11 :

L'opération de la figure Fig. 2.15 permet d'isoler le bit 0 (c'est à dire de poids 1) de la donnée en utilisant l'opération : Donnée & Masque.

0xAF	1	0	1	0	1	1	1	1
0x2	0	0	0	0	0	0	1	0
0xAF & **0x**2	0	0	0	0	0	0	1	0

(a) bit 1 à 1

0xAD	1	0	1	0	1	1	0	1
0x2	0	0	0	0	0	0	1	0
0xAD & **0x**2	0	0	0	0	0	0	0	0

(b) bit 1 à 0

Figure 2.15 – Valeur du bit 1

Seul le bit à 1 dans le masque sera conservé dans le résultat (en vert). L'Algorithme 2 présente un exemple d'utilisation du masquage.

Algorithme 2 Tester la valeur du bit 1

```
1: Variables
2:    i : entier
3: Début
4: Lire (i)
5: Si i & 2 = 2 Alors
6:    Afficher ("Le bit 1 est à 1")
7: Sinon
8:    Afficher ("Le bit 1 est à 0")
9: FinSi
10: Fin
```

2.6.4 Opérateur ou bit à bit, noté |

Pour positionner un bit particulier d'une donnée à 1 , la technique utilisée est appelée **forçage** . Nous appliquons une fonction ou bit à bit à la donnée et au masque. Il faut qu'une des deux données soit à 1 pour obtenir un résultat à 1.

Exemple 12 :

L'opération de la figure Fig. 2.16 permet de positionner le bit 0 à 1 (c'est-à-dire de poids 1) de la donnée en utilisant l'opération : Donnée | Masque.

0xAF		1	0	1	0	1	1	1	1	**0x**AD		1	0	1	0	1	1	0	1
0x2		0	0	0	0	1	0	1	0	**0x**2		0	0	0	0	0	0	1	0
0xAF	| **0x**2	1	0	1	0	1	1	1	1	**0x**AD	| **0x**2	1	0	1	0	1	1	1	1

(a) Non forcé (b) Forcé

FIGURE 2.16 – Forcer bit 1 à 1

L'ALGORITHME 3 présente un exemple d'utilisation du forçage.

Algorithme 3 Positionner le bit 1 à 1

```
1: Variables
2:    i : entier
3: Début
4: Lire (i)
5: i ← i | 2
6: Afficher (i) //bit 1 mis à 1
7: Fin
```

2.6.5 Opérateur ou exclusif bit à bit, noté ^

L'opérateur ou exclusif permet d'inverser la valeur de bit en fonction d'un masque.
La table de vérité de l'opérateur ou exclusif est représentée sur la figure FIG. 2.17 .

a	b	$a \hat{} b$
0	0	0
0	1	1
1	0	1
1	1	0

FIGURE 2.17 – Ou exclusif bit à bit

$$
\begin{aligned}
a \hat{} b &= (\sim a \,\&\, b) \mid (\sim b \,\&\, a) && (2.17)\\
&= (1 \,\&\, b) \mid (\sim b \,\&\, 0) \text{ si } a = 0\\
&= b \text{ si } a = 0 && (2.18)\\
&= (0 \,\&\, b) \mid (\sim b \,\&\, 1) \text{ si } a = 1\\
&= \sim b \text{ si } a = 1 && (2.19)
\end{aligned}
$$

Les bits devant être inversés seront positionnés à 1 dans le masque. Ceux qui conserveront leur valeur seront positionnées à 0 dans le masque.

Exemple 13 :

Pour inverser les bits numérotés de 1 à 4 d'une donnée, nous utiliserons un masque ayant pour valeur **0b**0001 1110=30=**0x**1E. La figure FIG. 2.18 illustre ce principe.

0x55		0	1	0	1	0	1	0	1
0x1E		0	0	0	1	1	1	1	0
0x55	^ **0x**1E	0	1	0	0	1	0	1	1

FIGURE 2.18 – Opérateur ou exclusif bit à bit

Exemple 14 :

Algorithme 4 Valeurs minimale et maximale en utilisant $\hat{}$

```
1: Variables
2:    i, j, k : entier
3: Début
4: Lire (i)
5: j ← i ^ i
6: k ← i ^ ~ (i)
7: Afficher (j, k) //Affichera la valeur 0 et le plus grand nombre représentable
8: Fin
```

2.6.6 Opérateurs de décalage logique notés << et >>>

Les opérations de décalage sur les nombres non signés sont couramment appelées opérations de **décalage logique à gauche ou à droite**. Elles sont présentés sur la FIG. 2.19 et permettent la translation des bits d'une donnée vers le **MSB** (à gauche) ou vers le **LSB** (à droite) d'un nombre de rang donné. Les bits sortant sont perdus et ceux entrant sont à 0 (en rouge)

0x2F	0	0	1	0	1	1	1	1
0x2F >>> 1 = **0x**17	0	0	0	1	0	1	1	1
0x2F << 1 = **0x**5E	0	1	0	1	1	1	1	0

FIGURE 2.19 – Décalage logique d'un rang à gauche et à droite

Remarque 12 :

— Le décalage logique à droite d'un rang correspond à une division non signée par deux de la valeur de la donnée. Le décalage à gauche d'un rang correspond à une multiplication non signée par deux de la valeur de la donnée.
— Le décalage logique interprète la donnée comme étant non signée.

2.6.7 Circuits bit à bit

La figure FIG 2.20 présente la réalisation d'une porte logique ET utilisant des données représentées sur 32 bits (vue interne et vue externe)

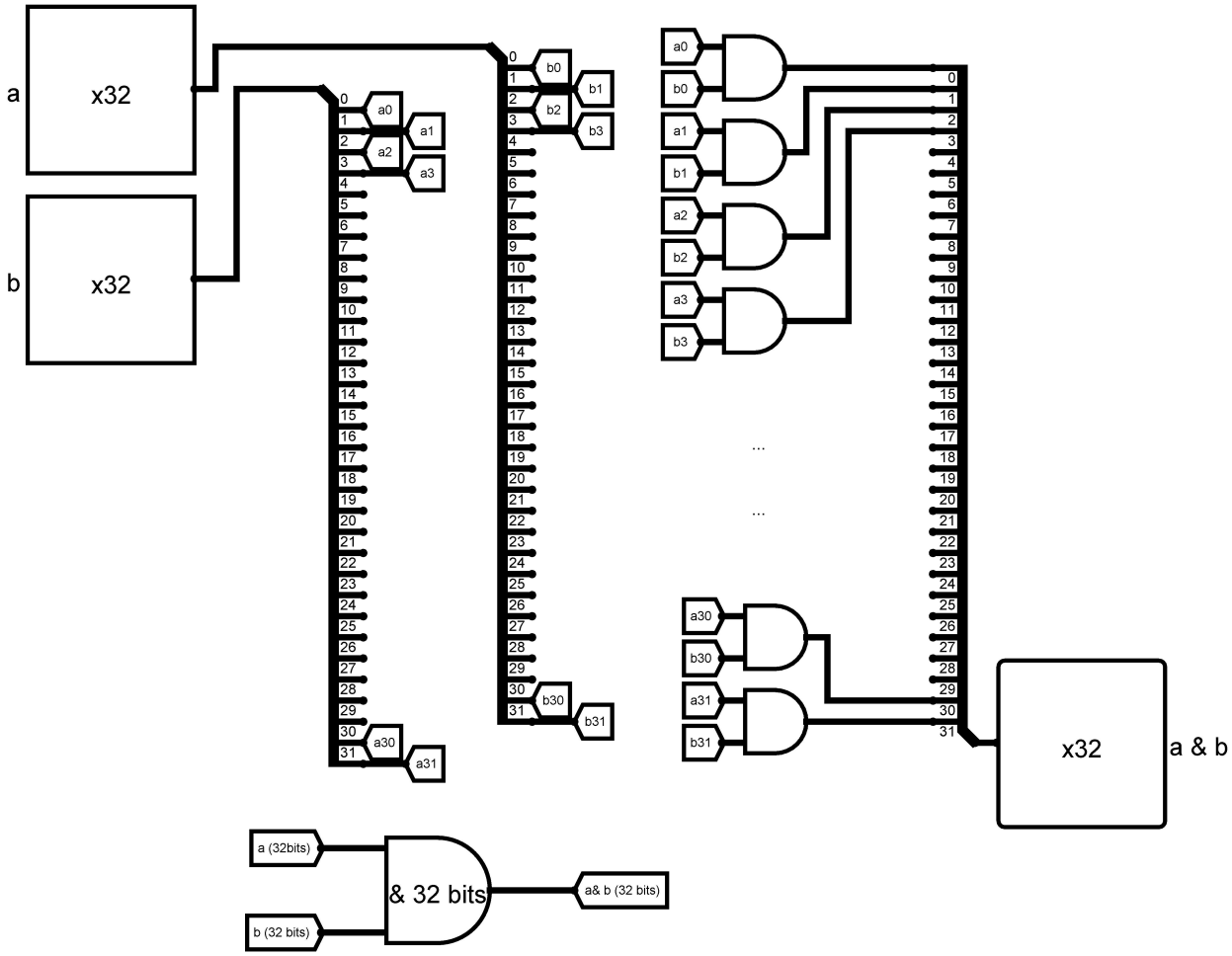


FIGURE 2.20 – Circuit de l'opérateur et bit à bit

— Identique pour l'opérateur | et ^

La figure FIG 2.21 présente la réalisation de décalage logique d'un rang. (vue interne et vue externe)

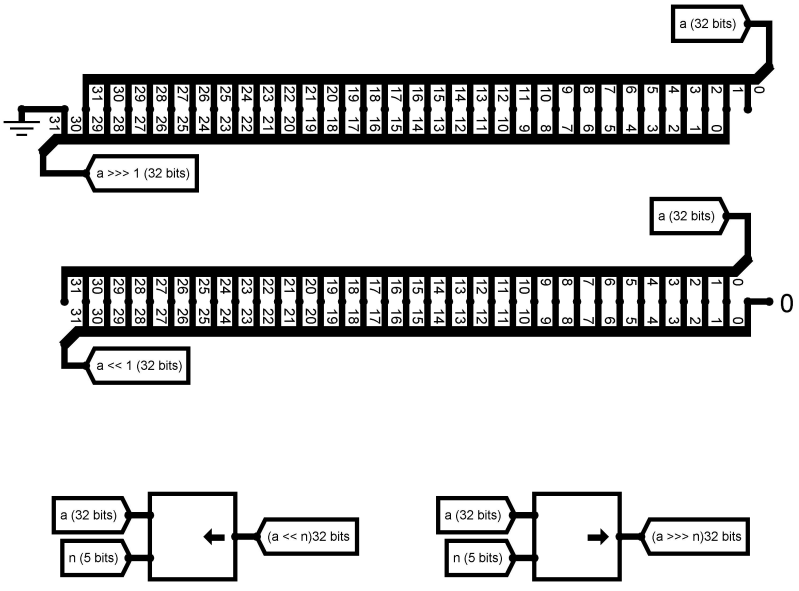


FIGURE 2.21 – Circuit décalage logique de 1 rang

Remarque 13 :
La valeur 0 peut être représentée avec un constante 1 bit valant 0, ou le symbole de la terre.

2.6.8 Exemples de calculs itératifs utilisant les opérateurs bit à bit

1. **Comptage du nombre de bit à 1 d'une donnée**
   Pour compter le nombre de bits à 1 d'une donnée (125) nous pouvons réaliser la séquence d'opérations suivantes :

125	0	1	1	1	1	1	0	1
125 & 1	0	0	0	0	0	0	0	1
125 >>> 1=62	0	0	1	1	1	1	1	0
62 & 1	0	0	0	0	0	0	0	0
62 >>> 1=31	0	0	0	1	1	1	1	1
31 & 1	0	0	0	0	0	0	0	1
31 >>> 1=15	0	0	0	0	1	1	1	1
15 & 1	0	0	0	0	0	0	0	1
15 >>> 1=7	0	0	0	0	0	1	1	1
7 & 1	0	0	0	0	0	0	0	1
7 >>> 1=3	0	0	0	0	0	0	1	1
3 & 1	0	0	0	0	0	0	0	1
3 >>> 1=1	0	0	0	0	0	0	0	1
15 & 1	0	0	0	0	0	0	0	1
1 >>> 1=0	0	0	0	0	0	0	0	0

FIGURE 2.22 – Comptage du nombre de donnée à 1

— Tous les bits de la donnée passent séquentiellement dans le bit de poids faible.
— Il suffit d'additionner les bits de poids faible (0 ou 1 en rouge) pour obtenir le nombre de bit à 1 de la donnée
— Le processus est lent car nous devons itérer sur tous les bits de la donnée

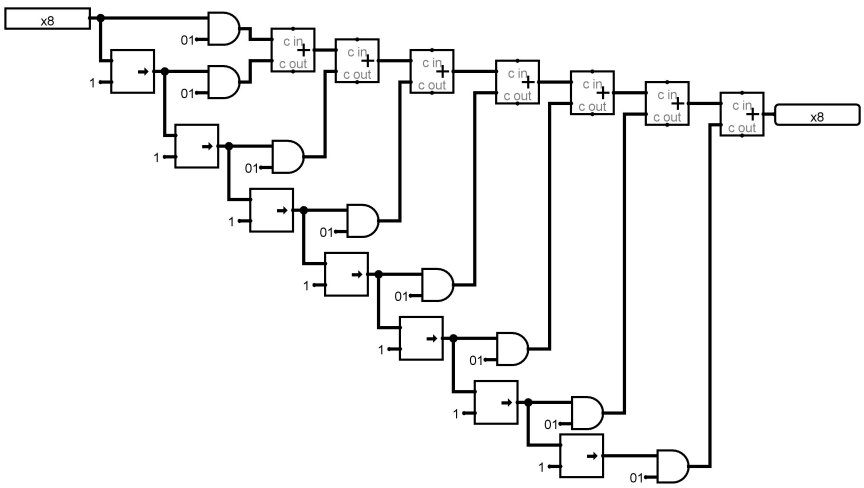


FIGURE 2.23 – Circuit de comptage du nombre de donnée à 1

Le circuit est complexe. En effet, pour chaque bit de la donnée nous avons besoin d'un additionneur 2 × 8 bits, d'un circuit de décalage logique à droite et de une porte ET. Dans ce cas, seul le bit 0 de l'additionneur est utilisé.

2. **Comptage rapide du nombre de bit à 1 d'une donnée**

   Afin d'utiliser les additionneurs de façon optimale, nous effectuerons des calculs en parallèle sur plusieurs des bits de la donnée :

125	0	1	1	1	1	1	0	1
125 & **0x**55	0	1	0	1	0	1	0	1
+ (125 >>> 1) & **0x**55	0	0	0	1	0	1	0	0
= 105	0	1	1	0	1	0	0	1
105 & **0x**33	0	0	1	0	0	0	0	1
+ (105 >>> 2) & **0x**33	0	0	0	1	0	0	1	0
= 51	0	0	1	1	0	0	1	1
51 & **0x**F	0	0	0	0	0	0	1	1
+ (51 >>> 4) & **0x**F	0	0	0	0	0	0	1	1
= 6	0	0	0	0	0	1	1	0

FIGURE 2.24 – Comptage rapide du nombre de donnée à 1

— Au début, nous addition de deux bits côte à côte. Pour cela, nous devons les aligner (décalage logique à droite des bits de rangs impairs) et masquer les bits de rangs pair. Nous additions ensuite les bits de rangs pairs et nous obtenons un résultat sur deux bits.
— Ensuite, nous additionnons les résultats précédents codés sur deux bits, en utilisant un décalage logique de deux rangs à droite et en masquant ces bits dans la donnée de départ.
— Pour terminer nous additionnons les deux résultats codés sur quatre bits en le additionnant.
— Plus la taille de la donnée est importante, plus le gain en performance important.

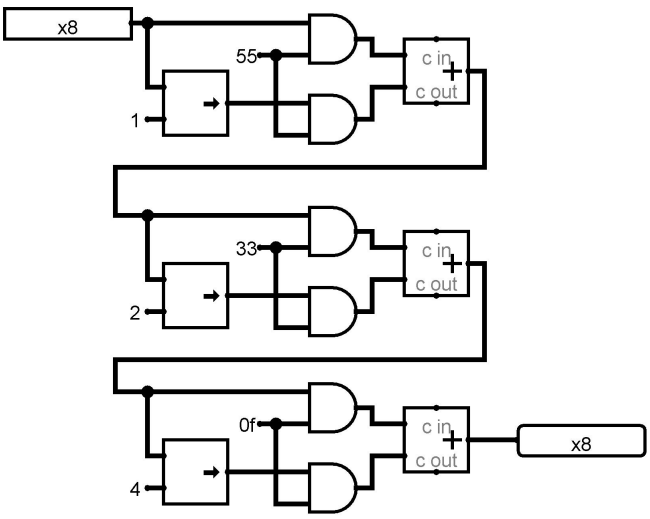


FIGURE 2.25 – Circuit de comptage rapide du nombre de donnée à 1

Nous remarquons que le circuit est beaucoup plus simple. Nous utilisons seulement trois additionneurs, 3 décalage logique à droite et 6 portes ET.

2.7 Utilisation des nombres non signés

2.7.1 Adresses mémoire

Ce codage est donc bien adapté pour représenter les adresses de la mémoire de l'ordinateur. En effet, elles débutent toujours à l'adresse 0 et se termine a l'adresse $2^n - 1$. L'arithmétique sur les nombres non signés permet de faire des opérations sur les pointeurs.

2.7.2 Représentation des caractères

La deuxième utilisation des nombres non signés est le codage des caractères. A chaque caractère est associé un nombre entier appelé **code ASCII** . Les codes ASCII standards sont codés sur 7 bits. Lorsque le huitième bit est à 1, les codes étendus sont activés (é,à,[etc ...). Pour pouvoir représenter tous les caractères régionaux le code est stocké sur 16 bits ; il est nommé UNICODE.
`http://fr.wikipedia.org/wiki/Unicode`
Les **codes ASCII** peuvent être traités comme d'autres données, c'est-à-dire que nous pouvons effectuer des opérations arithmétiques ou logiques bit à bit sur ceux-ci. La figure Fig 2.26 donne les valeurs des codes ascii pour les caractères hexadécimaux.

dec	hex	Char	dec	hex	char	dec	hex	char
48	**0x**30	'0'	65	**0x**41	'A'	97	**0x**61	'a'
49	**0x**31	'1'	66	**0x**42	'B'	98	**0x**62	'b'
50	**0x**32	'2'	67	**0x**43	'C'	99	**0x**63	'c'
...	...	...	...	...	...	...	...	...
53	**0x**35	'5'	70	**0x**46	'F'	102	**0x**66	'f'
...	...	...	...	...	...	...	...	...
59	**0x**39	'9'	74	**0x**4A	'J'	106	**0x**6A	'j'
...	...	...	...	...	...	...	...	...

Figure 2.26 – Table des codes ascii

Exemple 15 :

— L'opération 'A'+**0x**20 = **0x**61 calcule le code ascii de 'a'. L'Algorithme 5 présente un exemple d'utilisation de ce concept.

Algorithme 5 Conversion d'un caractère majuscule en minuscule

```
1: Variables
2:    c : caractère
3: Début
4: c ← 'F' //Lettre majuscule
5: c ←(caractère)((entier)c + 0x20)
6: Afficher (c) // Affichera le caractère f
7: Fin
```

Remarque 14 :

— La notation (entier)indique une conversion explicite de type ou transtypage (aussi nommé typecast). La variable sera alors considérée comme entière non signée.

— La notation (caractère) modifie le type et éventuellement la valeur, de la variable pour la traiter désormais comme un caractère.

— Les codes ASCII inférieur à **0x**20 sont des codes de commande. Ils n'affichent aucun caractère.
— Nous ne pouvont donc pas afficher directement un digit. La conversion d'un digit n compris entre 0 et **0x**F en son code ASCII est réalisée en effectuant l'opération
  — Si $n \subset [0, 9]$: $c \leftarrow n +' 0'$
  — Si $n \subset [\mathbf{0x}\text{A}, \mathbf{0x}\text{F}]$: $c \leftarrow n - 10 +' A'$
  L'Algorithme 6 présente un exemple d'utilisation de ce concept.

Algorithme 6 Conversion d'un nombre non signé en un caractère

```
1: Variables
2:    i : entier
3:    c : caractère
4: Début
5: i ← 9
6: c ← (caractère)(i+'0')
7: Afficher (c) // Afficher le caractère 9
8: i ← 10
9: c ← (caractère)(i − 10+ 'A')
10: Afficher (c) // Afficher le caractère A
11: Fin
```

2.8 Exercices

Exercice 1 : Théorique : Opérateurs et algorithmes bit à bit

Remarque 15 : Notation valeur indéterminée

La notation N signifie que le bit de la donnée est inchangé.

Question 1 : Changement de base Exprimez en décimal les nombres non signés suivants : **0x**C6, **0b**1 0011,**0**67.

Question 2 : Binaire et hexadécimal Exprimez en binaire et en hexadécimal les nombres : 1, 0 , 127, 128, 129. Vérifiez en évaluant le polynôme que les valeurs binaires sont correctes.

Question 3 : Évaluation Donnez les résultats des expressions suivantes :

$$\begin{aligned}
\mathbf{0x}\text{FF00 FF00} \,\&\, \mathbf{0x}\text{12FF 34FF} &= ? \\
\mathbf{0x}\text{FF00 FF00} \mid \mathbf{0x}\text{12FF 34FF} &= ? \\
\mathbf{0x}\text{FF00 FF00} \,\hat{}\, \mathbf{0x}\text{12FF 34FF} &= ? \\
\mathbf{0x}\text{FF00 FF00} << 4 &= ? \\
\mathbf{0x}\text{FF00 FF00} >>> 4 &= ?
\end{aligned}$$

Les questions suivantes sont des préparations d'un des exercices sur le codage des nombres réels.

Question 4 : Effacer le bit de poids fort b_{31} Donnez l'expression permettant de positionner à 0 le bit de rang 31 d'une donnée x de taille 32 bits sans modifier les autres bits de x. Vous donnerez deux versions : une avec un masque et l'autre sans masque.

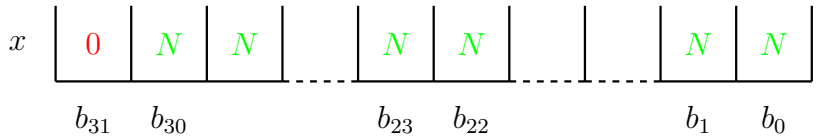


FIGURE 2.27 – Bit 31 de x à 0

Question 5 : Positionner le bit de poids fort b_{31} Donnez l'expression permettant de positionner à 1 le bit de rang 31 d'une donnée x de taille 32 bits, sans modifier les autres bits de x.

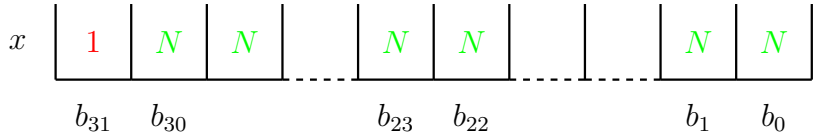


FIGURE 2.28 – Bit 31 de x à 1

Question 6 : Transférer le bit de poids fort b_{31} dans le bit poids faible Donnez l'expression permettant de positionner dans le bit de rang 0 d'une donnée x de taille 32 bits, le bit 31 d'une donnée y.

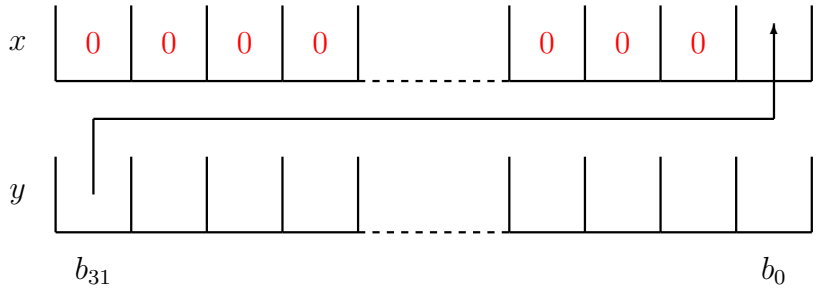


FIGURE 2.29 – Placer le bit 31 de y dans le bit 0 de x

Question 7 : Transférer le bit de poids faible dans le bit de poids fort b_{31} Donnez l'expression permettant de positionner dans le bit de rang 31 d'une donnée x de taille 32 bits, le bit 0 d'une donnée y. Les autres bits de x ne seront pas modifiés. Vous donnerez deux versions : une avec un masque et l'autre sans masque.

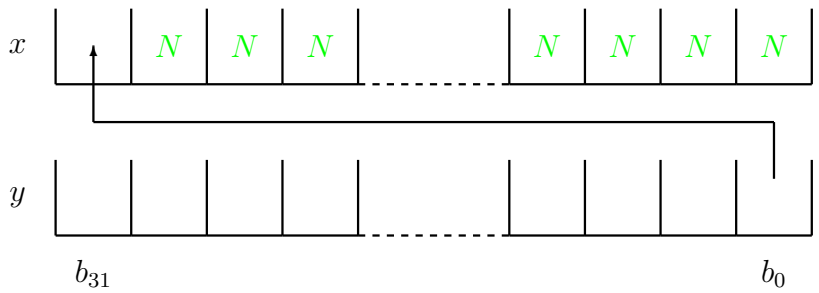


FIGURE 2.30 – Placer le bit 0 de y dans le bit 31 de x

Question 8 : Effacer les bits de poids intermédiaires (b_{30} à b_{23}) Donnez l'expression permettant de positionner à 0 les bits de rang 30 à 23 d'une donnée x de taille 32 bits sans modifier les autres bits de x.

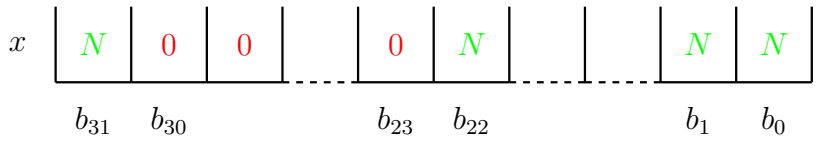


FIGURE 2.31 – Masquer bit 30 à 23 de x

Question 9 : Transférer l'ensemble des bits de poids intermédiaires (b_{30} à b_{23}) dans les bits de poids faibles Donnez l'expression permettant de positionner dans les bits de rang 7 à 0 d'une donnée x de taille 32 bits, les bits de rangs 30 à 23 d'une donnée y. Vous donnerez deux versions : une avec un masque et l'autre sans masque.

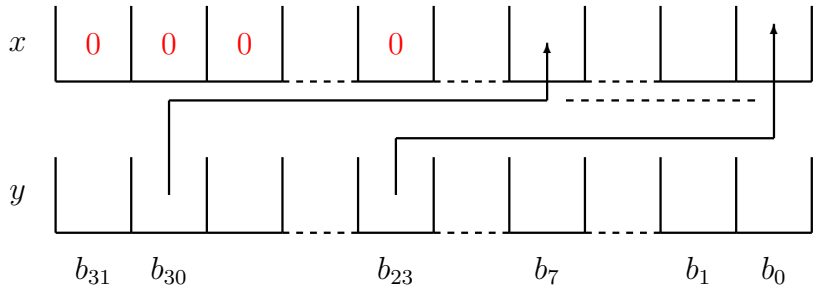


FIGURE 2.32 – Placer les bits 30 à 23 de y dans les bits 7 à 0 de x

Question 10 : Transférer les bits de poids faibles dans l'ensemble des bits de poids intermédiaires (b_{30} à b_{23}) Donnez l'expression permettant de positionner dans les bits de rang 30 à 23 d'une donnée x de taille 32 bits, les bits de rang 7 à 0 d'une donnée y. Les autres bits de x ne seront pas modifiés.

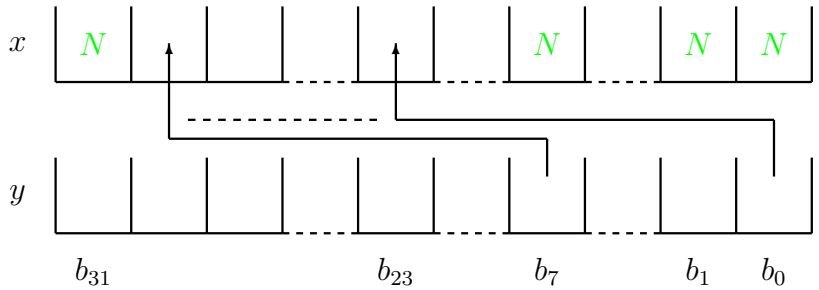


FIGURE 2.33 – Placer les bits 7 à 0 de y dans les bits 30 à 23 de x

Question 11 : Effacer l'ensemble des bits de poids faibles (b_{22} à b_0) Donnez l'expression permettant de positionner à 0 les bits de rang 22 à 0 d'une donnée x de taille 32 bits sans modifier les autres bits de x. Vous donnerez deux versions : une avec un masque et l'autre sans masque.

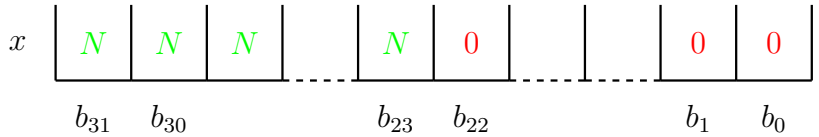


FIGURE 2.34 – Masquer bit 22 à 0 de x

Question 12 : Transférer l'ensemble des poids faibles dans l'ensemble des bits de poids faibles (b_{22} à b_0) Donnez l'expression permettant de positionner dans les bits de rang 22 à 0 d'une donnée x de taille 32 bits, les bits de rang 22 à 0 d'une donnée y. Vous donnerez deux versions : une avec un masque et l'autre sans masque.

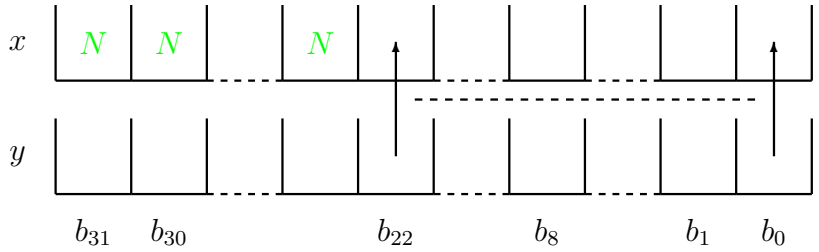


FIGURE 2.35 – Placer les bits 22 à 0 de y dans 22 à 0 de x

Question 13 : Transférer l'ensemble des bits de poids faibles (b_{22} à b_0) dans l'ensemble des poids faibles Donnez l'expression permettant de positionner dans les bits de rang 22 à 0 d'une donnée x de taille 32 bits, les bits de rang 22 à 0 d'une donnée y en positionnant à 1 le bit de rang 23.

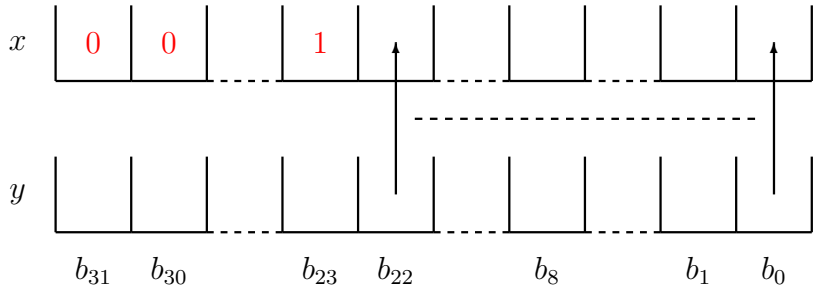


FIGURE 2.36 – Placer les bits 22 à 0 de y dans les bits 22 à 0 de x

Question 14 : Donnée taille inconnue Donnez l'expression permettant du positionner les bits de rangs 2 et 3 d'une donnée à 0 d'une donnée de taille inconnue.
Indication 1 : Nous ne pouvons pas spécifier le masque directement, vous utiliserez l'opérateur $\sim$.

Question 15 : Rotation de un rang à droite Donnez l'expression permettant de positionner dans x la donnée y de taille 32 bits après rotation de 1 rang à droite.

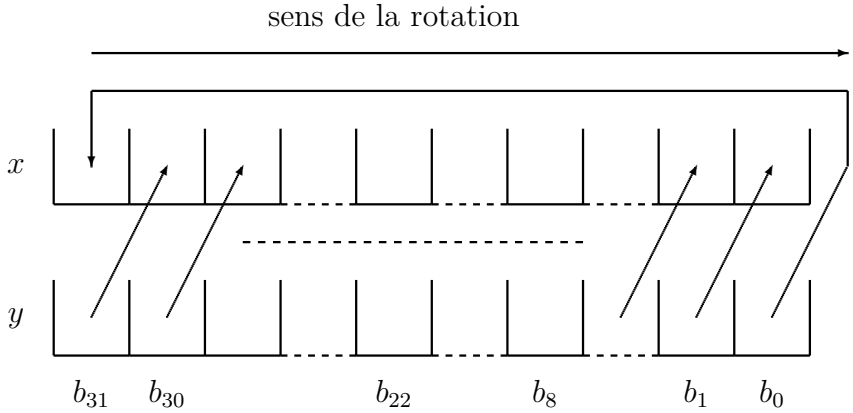


FIGURE 2.37 – Rotation d'un bit à droite

Exemple 16 :

$$\textit{rotateRight}(\mathbf{0x}8000\,0001) \quad = \quad \mathbf{0x}C000\,0000$$

Indication 1 : Vous décalerez la donnée dans le sens requis, vous placerez le bit sortant à sa nouvelle position et enfin vous combinerez ces deux expressions.
Indication 2 : Vous utilisez les opérateurs de décalage logique et l'opérateur | .

Question 16 : Rotation de un rang à gauche Donnez l'expression permettant de positionner dans x la donnée y de taille 32 bits après rotation de un rang à gauche.

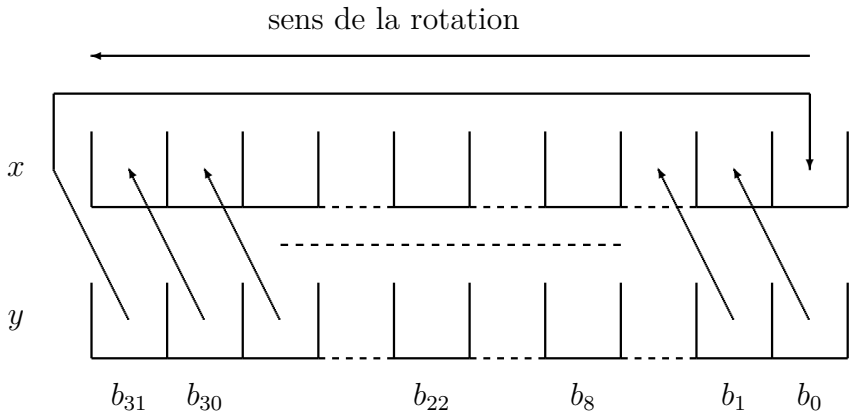


FIGURE 2.38 – Rotation d'un bit à gauche

Exemple 17 :

$$\textit{rotateLeft}(\mathbf{0x}8000\,0001) \quad = \quad \mathbf{0x}0000\,0003$$

Question 17 : Rotation de n rangs Modifier les deux expressions précédentes pour effectuer des rotations de n rangs ($0 \leq n \leq 31$) et non plus de un rang.
Exemple 18 :

$$\begin{aligned} \textit{rotateLeft}(\mathbf{0x}8000\,0001, 2) &= \mathbf{0x}0000\,0006 \\ \textit{rotateRight}(\mathbf{0x}8000\,0001, 2) &= \mathbf{0x}6000\,0000 \end{aligned}$$

Question 18 : Fusion Donnez l'expression permettant de fusionner deux valeurs (nommées a et b) à l'aide d'un masque (nommé m). Le résultat (nommé r) de la fusion contient les bits de la valeur a étant à 0 dans le masque m et les bits de la valeur b étant à 1 dans le masque m.
Exemple 19 :

$$\begin{aligned} a &= \mathbf{0b}1001\,0001 \\ b &= \mathbf{0b}0011\,1100 \\ m &= \mathbf{0b}1111\,0000 \\ \textit{fusion}(a, b, m) &= \mathbf{0b}0011\,0001 \end{aligned}$$

1. En utilisant les opérateurs bit à bit $\sim$, $|$ et $\&$ donnez l'expression permettant de déterminer la valeur de r.
2. Afin d'optimiser le nombre d'opérateurs logique, montrez que l'expression suivante est identique à celle de la question précédente.

$$r \quad = \quad a \hat{} ((a \hat{} b) \,\&\, m)$$

Question 19 : *bitCount* Complétez l'algorithme d'une fonction nommée *bitCount* permettant de compter le nombre de bits à 1 présent dans un entier de 32 bits.

Algorithme 7 Comptage du nombre de bits à 1 - énoncé

```
1: Fonction bitCount (nombre : entier) : entier
2: Variables
3:    résultat : entier
4: Début
5: résultat ← 0
6: Tant Que nombre... Faire
7:    résultat ← résultat + ...
8:    nombre ← nombre...
9: Fin Tant Que
10: Retourner résultat
11: Fin fonction
```

Indication 1 : Observerez si le bit de poids faible vaut 1, puis décalerez successivement les autres bits dans le bit de poids faible.
Indication 2 : Ajoutez la valeur du bit de poids faible au résultat.
Indication 3 : Les opérateurs $\&$ bit à bit ainsi que le décalage logique à droite $>>>$ et l'addition seront les seuls utilisés.

Question 20 : *numberOfTrailingZeros* Complétez l'algorithme de la fonction nommée *numberOfTrailingZeros* pour compter le nombre de bits à 0 à droite d'une donnée entière de 32 bits.

Algorithme 8 Nombre de zéros à droite - énoncé

```
1: Fonction numberOfTrailingZeros (nombre : entier) : entier
2: Variables
3:    résultat : entier
4: Début
5: Si nombre = 0 Alors
6:    Retourner ...
7: FinSi
8: résultat ← 0
9: Tant Que (nombre....) = 0 Faire
10:    résultat ← résultat...
11:    nombre ← nombre...
12: Fin Tant Que
13: Retourner résultat
14: Fin fonction
```

Exemple 20 :

$$\begin{aligned} \textit{numberOfTrailingZeros}(\mathbf{0x}\text{FFFF 0000}) &= 16 \\ \textit{numberOfTrailingZeros}(4) &= 2 \\ \textit{numberOfTrailingZeros}(0) &= 32 \end{aligned}$$

Indication 1 : Observez la valeur du bit de poids faible et décalez à droite le nombre tant que la valeur du bit de poids faible est à 0. Attention au cas ou le nombre vaut 0.

Question 21 : *numberOfLeadingZeros* Modifier l'algorithme de la fonction nommée *numberOfTrailingZeros* pour compter le nombre de bits à 0 à gauche d'une donnée entière codée sur 32 bits.

Fonction *numberOfLeadingZeros* (*nombre* : entier) : entier

Exemple 21 :

$$\begin{aligned} \textit{numberOfLeadingZeros}(\mathbf{0x}\text{FFFF}) &= 16 \\ \textit{numberOfLeadingZeros}(\mathbf{0x}\text{7FFF FFFF}) &= 1 \\ \textit{numberOfLeadingZeros}(\mathbf{0x}\text{3000 0000}) &= 2 \\ \textit{numberOfLeadingZeros}(0) &= 32 \end{aligned}$$

Indication 1 : Observez la valeur du bit de poids fort et décalez à gauche le nombre tant que la valeur du bit de poids fort est à 0. Attention au cas ou le nombre vaut 0.

Question 22 : *lowestOneBit* Complétez l'algorithme de la fonction nommée *lowestOneBit* pour conserver uniquement le bit à 1 le plus à droite d'une donnée entière codée sur 32 bits.

Algorithme 9 Poids du bit à 1 le plus à droite - énoncé

```
1:  Fonction lowestOneBit (nombre : entier) : entier
2:  Variables
3:     résultat : entier
4:  Début
5:  Si nombre = 0 Alors
6:     Retourner ...
7:  FinSi
8:  résultat ← ...
9:  Tant Que (nombre & ...) = 0 Faire
10:    résultat ← résultat...
11: Fin Tant Que
12: Retourner résultat
13: Fin fonction
```

Fonction *lowestOneBit* (*nombre* : entier) : entier

Exemple 22 :

$$\begin{aligned} \textit{lowestOneBit}(\mathbf{0x}\text{FFFF 0000}) &= \mathbf{0x}\text{1 0000} \\ \textit{lowestOneBit}(\mathbf{0x}\text{7F FFFF}) &= 1 \\ \textit{lowestOneBit}(0) &= 0 \end{aligned}$$

Indication 1 : Attention au cas où le nombre vaut 0. Vous décalerez un bit à 1 placé dans le résultat.

Question 23 : *hightestOneBit* Modifier l'algorithme de la fonction nommée *lowestOneBit* pour conserver uniquement le bit à 1 le plus à gauche d'une donnée entière codée sur 32 bits.

Fonction *hightestOneBit* (*nombre* : entier) : entier

Exemple 23 :

$$\begin{aligned} \textit{hightestOneBit}(\mathbf{0x}\text{FFFF}) &= \mathbf{0x}8000 \\ \textit{hightestOneBit}(\mathbf{0x}7\text{F FFFF}) &= \mathbf{0x}40\,0000 \\ \textit{hightestOneBit}(5) &= 4 \\ \textit{hightestOneBit}(0) &= 0 \end{aligned}$$

Question 24 : *reverse* Complétez l'algorithme d'une fonction nommée *reverse* inversant l'ordre des bits d'une donnée entière codée sur 32 bits. Le bit 0 est inversé avec le bit 31, le bit 1 est inversé avec le bit 30 ...

Algorithme 10 Inversion des bits - énoncé

```
1: Fonction reverse (nombre : entier) : typeentier
2: Variables
3:    i, résultat : entier
4: Début
5: résultat ← .
6: Pour i de 0 à 31 Faire
7:    résultat ← (résultat << ...) | (nombre & ....)
8:    nombre ← nombre...
9: Fin Pour
10: Retourner résultat
11: Fin fonction
```

Exemple 24 :

$$\textit{reverse}(\mathbf{0x}7\text{F00 0001}) = \mathbf{0x}8000\,00\text{FE}$$

Indication 1 : En parcourant les bits de la donnée depuis LSB jusq'au MSB , vous les placerez dans le LSB du résultat que vous décalerez à gauche d'un rang à chaque itération.

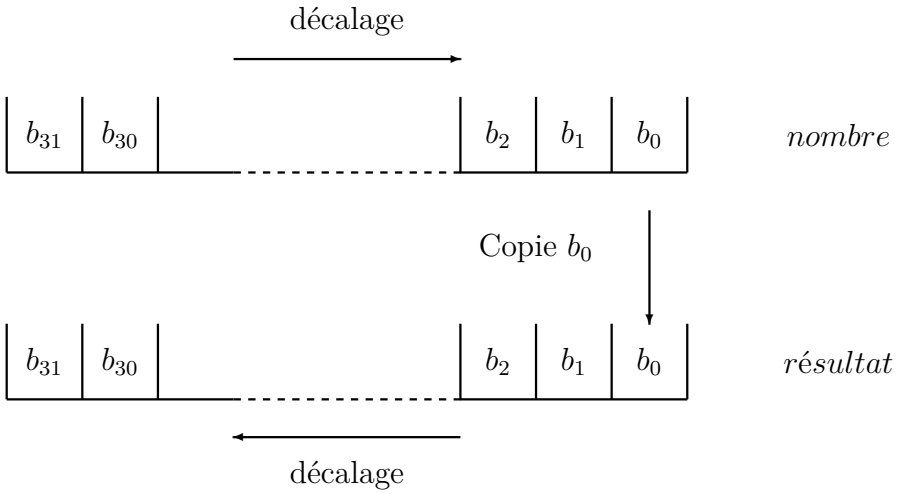


FIGURE 2.39 – Principe de l'inversion des bits d'une donnée

Question 25 : *reverseBytes* Donnez l'algorithme d'une fonction nommée *reverseBytes* inversant l'ordre des octets d'une donnée entière codée sur 32 bits.

Fonction *reverseBytes* (*nombre* : entier) : typeentier

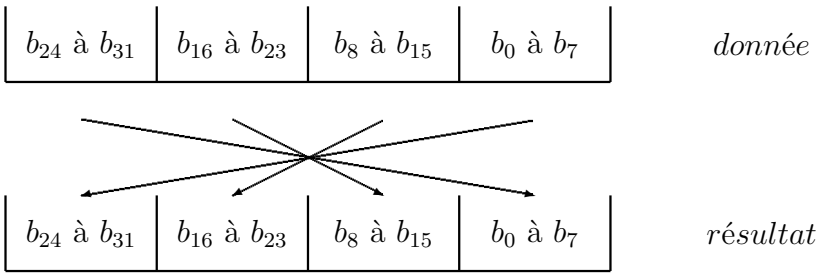


FIGURE 2.40 – Principe de l'inversion des octets d'une donnée

Exemple 25 :

$$\textit{reverseBytes}(\mathbf{0x}7\text{F}12\,3401) \quad = \quad \mathbf{0x}0134\,127\text{F}$$

Indication 1 : Vous n'utilisez pas d'itération et vous placerez les 4 octets directement dans le résultat.

Question 26 : *fastBitCount* Complétez l'algorithme d'une fonction nommée *fastBitCount* permettant de compter le nombre de bits à 1 d'une donnée entière codée sur 32 bits. La figure 2.41 illustre le principe sur une donnée entière codée sur 8 bits.

Algorithme 11 Comptage rapide du nombres de bit à 1 - énoncé

1: **Fonction** *fastBitCount* (*nombre* : entier) : entier
2: **Début**
3: $nombre \leftarrow nombre \,\&\, ... + ((nombre >>> ...) \,\&\, ...$
4: $nombre \leftarrow nombre \,\&\, ... + ((nombre >>> ...) \,\&\, ...$
5: $nombre \leftarrow nombre \,\&\, ... + ((nombre >>> ...) \,\&\, ...$
6: $nombre \leftarrow nombre \,\&\, ... + ((nombre >>> ...) \,\&\, ...$
7: $nombre \leftarrow nombre \,\&\, ... + ((nombre >>> ...) \,\&\, ...$
8: retourner(*nombre*)
9: **Fin Fonction**

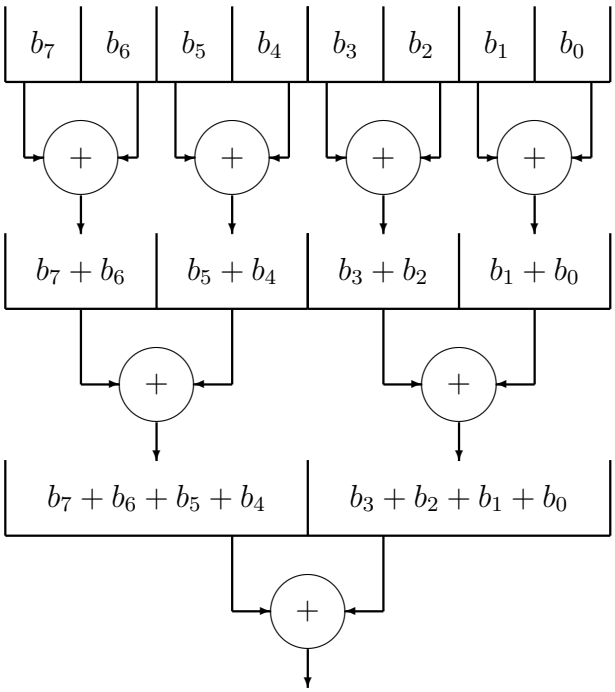


FIGURE 2.41 – Comptage rapide du nombre de bit à 1 d'un octet

Exemple 26 :

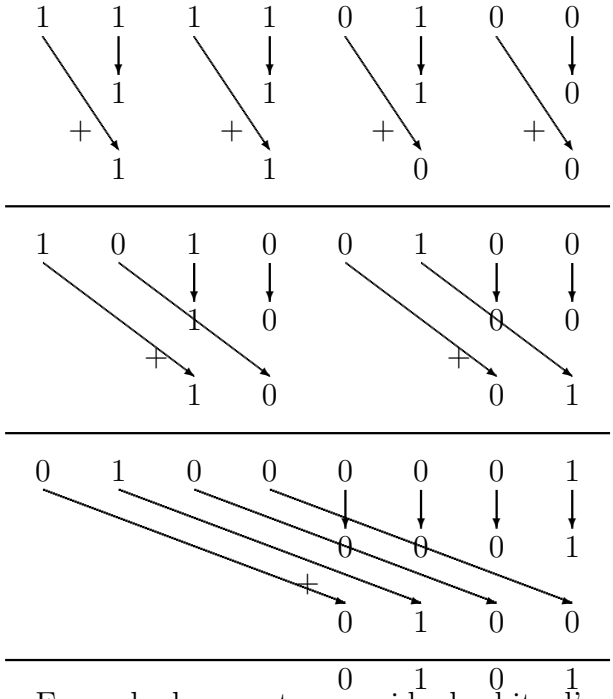


FIGURE 2.42 – Exemple de comptage rapide des bits d'une donnée

Indication 1 :

- Pour calculer $b_0 + b_1$ nous devons placer b_0 et b_1 à la même position (rang 0 de la donnée). Les résultats de b_0+b_1 sur deux bits seront placés aux bits de rang 0 et 1. Nous calculerons, en même temps, les autres résultats sur deux bits. Nous répéterons ensuite le processus sur 2×2 bits et 2×4 bits.
- Vous généraliserez l'algorithme pour un nombre entier codé sur 32 bits. Pour cela, vous répéterez le processus sur 2×8 bits et 2×16 bits.

Question 27 : *forDigit* Complétez l'algorithme d'une fonction nommée *forDigit* permettant de convertir un digit hexadécimal en un caractère. Le code ASCII d'un caractère sera représenté par la notation ” (par exemple 'A' représente le code ascii du caractère A). Les codes ASCII sont ordonnés de la façon suivante : '0' < '9' < 'A' < 'F' ou en hexadécimal **0x**30< **0x**39 < **0x**41 < **0x**46.

Algorithme 12 Conversion d'un digit en un caractère - énoncé

```
1: Fonction forDigit (nombre : entier) : caractère
2: Début
3: Si nombre > ... Alors
4:    Retourner (caractère)(nombre+ ...)
5: Sinon
6:    Retourner (caractère)(nombre+ ...)
7: FinSi
8: Fin fonction
```

Exemple 27 :

- *forDigit*(**0x**A) retourne le caractère de code ascii **0x**41, soit le caractère 'A',
- *forDigit*(1) retourne le caractère de code ascii **0x**31, soit le caractère '1'.

Indication 1 : Vous testerez les cas ou (nombre > 9) car les codes ascii '9' et 'A' ne sont pas consécutifs.

Question 28 : *digit* Complétez l'algorithme d'une fonction nommée *digit* permettant de convertir un caractère (supposé valide) en un digit.

Algorithme 13 Conversion d'un caractère en un digit - énoncé

```
1: Fonction digit (car : caractère) :entier
2: Début
3: Si (car) > ... Alors
4:    Retourner (entier)car...)
5: Sinon
6:    Retourner (entier)car...)
7: FinSi
8: Fin fonction
```

Exemple 28 :

- *digit*('A') retourne l'entier 10,
- *digit*('1') retourne l'entier 1.

Question 29 : *toString* Donnez l'algorithme d'une fonction nommée *toString* permettant de convertir un nombre non signé en un chaîne de caractères le représentant dans une base comprise entre 2 et 16. L'ajout d'un caractère dans une chaîne de caractères est effectué à l'aide de l'opérateur +.

Algorithme 14 Conversion d'un nombre non signé en une chaîne de caractères base donnée - énoncé

```
1: Fonction toString (nombre, base : entier) : chaîne de caractères
2: Variables
3:    chaîne : chaîne de caractères
4: Début
5: chaîne ← ""
6: Répéter
7:    chaîne ← forDigit(...) + ...
8:    nombre ← nombre...
9: Jusqu'à nombre = ...
10: Retourner chaîne
11: Fin fonction
```

Indication 1 : Vous utiliserez l'algorithme des divisions successive présenté sur l'ALGORITHME. 1. Vous utiliserez aussi la fonction *forDigit*.

Question 30 : *toHexString* Modifier l'algorithme de la fonction *toString* pour convertir un nombre non signé en une chaîne de caractères le représentant en hexadécimal.
Fonction *toHexString* (*nombre* : entier) : chaîne de caractères
Indication 1 : Les opérateurs *mod* et / de l'algorithme précédent seront remplacés par des opérateurs bits à bits. Les opérateurs & bit à bit ainsi que le décalage logique à droite >>> seront les seuls utilisés.

Question 31 : *toBinaryString* Modifier l'algorithme de la fonction *toString* pour convertir un nombre non signé en une chaîne de caractères le représentant en binaire.
Fonction *toBinaryString* (*nombre*, *base* : entier) : chaîne de caractères

Indication 1 : L'utilisation de la fonction *forDigit* n'est pas indispensable car seuls les caractères sont '0' ou '1' sont ajoutés dans la chaîne résultat.

Question 32 : *parseInt* Complétez l'algorithme d'une fonction nommée *parseInt* permettant de convertir une chaîne de caractères représentant un entier non signé dans une base donnée en un nombre non signé. La chaîne de caractères ne contiendra que des caractères valides, **nous ne gérerons pas les erreurs**.

Algorithme 15 Conversion d'une chaîne de caractère base quelconque en un nombre non signé - énoncé

```
1: Fonction parseInt (chaîne : chaîne de caractères,base : entier) : entier
2: Variables
3:    résultat, position : entier
4: Début
5: résultat ← 0
6: Pour position de 0 à Taille(chaîne) ... Faire
7:    résultat ← ... + digit(...)
8: Fin Pour
9: Retourner résultat
10: Fin fonction
```

Remarque 16 :

La taille de la chaîne de caractères ainsi qu'un caractère de la chaîne de caractères sont obtenus en utilisant les fonctions suivantes :

- *Taille*(*chaîne* : chaîne de caractères) : entier
- *Caractère*(*chaîne* : chaîne de caractères, *position* : entier) : caractèreavec $0 \leq position < Taille$(chaine)

Indication 1 :

- Vous évaluerez le polynôme en utilisant le schéma itératif de Horner ci-dessous :

$$\begin{aligned} \mathrm{N} &= d_{n-1} \times b^{n-1} + d_{n-2} \times b^{n-2} + ...d_i \times b^i + d_1 \times b + d_0 \\ &= ((((0 \times b + d_{n-1}) \times b + d_{n-2}) \times b + ...d_i) \times b + ...d_1) \times b + d_0 \end{aligned}$$

- Dans ce schéma nous avons répétition du calcul de $(a \times b) + c$.
- Vous utiliserez la fonction *digit*.

Question 33 : Additionneur complet avec des portes NAND Complétez la table de vérité et donnez les équations d'un additionneur complet (3 bits d'entrée noté a, b et c_{in} et deux sorties c_{out} et r.

c_{in}	a	b	c_{out}	r
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

FIGURE 2.43 – Table de vérité de l'additionneur complet, énoncé

Question 34 : Égalité n bits avec des portes NAND Complétez la table de vérité et donnez l'équation qui renvoie 1 si les deux entrées nommées a et b codées sur n bits sont identiques.

a_i	b_i	$Equal$
0	0	
0	1	
1	0	
1	1	

FIGURE 2.44 – Table de vérité égalité un bit, énoncé

Correction exercice 1 :

Question 1 : Changement de base

$$\begin{aligned} \mathbf{0x}\text{C6} &= 12 \times 16 + 6 = 198 \\ \mathbf{0b}1\,0011 &= 16 + 2 + 1 = 19 \\ \mathbf{0}67 &= 6 \times 8 + 7 = 55 \end{aligned}$$

Question 2 : Binaire et hexadécimal

$$\begin{aligned} 1 &= \mathbf{0b}1 = \mathbf{0x}1 \\ 127 &= \mathbf{0b}0111\,1111 = \mathbf{0x}7\text{F} = 7 \times 16 + 15 \\ 128 &= \mathbf{0b}1000\,0000 = \mathbf{0x}80 = 8 \times 16 \\ 129 &= \mathbf{0b}1000\,0001 = \mathbf{0x}81 = 8 \times 16 + 1 \end{aligned}$$

Question 3 : Évaluation

$$\begin{aligned} \mathbf{0x}\text{FF00 FF00} \,\&\, \mathbf{0x}\text{12FF 34FF} &= \mathbf{0x}\text{1200 3400} \\ \mathbf{0x}\text{FF00 FF00} \mid \mathbf{0x}\text{12FF 34FF} &= \mathbf{0x}\text{FFFF FFFF} \\ \mathbf{0x}\text{FF00 FF00} \,\hat{}\, \mathbf{0x}\text{12FF 34FF} &= \mathbf{0x}\text{EDFF CBFF} \\ \mathbf{0x}\text{FF00 FF00} << 4 &= \mathbf{0x}\text{F00F F000} \\ \mathbf{0x}\text{FF00 FF00} >>> 4 &= \mathbf{0x}\text{0FF0 0FF0} \end{aligned}$$

Question 4 : Effacer le bit de poids fort b_{31}

$$\begin{aligned} x &\leftarrow x \,\&\, \mathbf{0x}\text{7FFF FFFF} \\ x &\leftarrow (x << 1) >>> 1 \end{aligned}$$

Question 5 : Positionner le bit de poids fort b_{31}

$$x \leftarrow x \mid \mathbf{0x}\text{8000 0000}$$

Question 6 : Transférer le bit de poids fort b_{31} dans le bit de poids faible

$$x \leftarrow (y >>> 31)$$

Question 7 : Transférer le bit de poids faible dans le bit de poids fort b_{31}

$$\begin{aligned} x &\leftarrow (x \,\&\, \mathbf{0x}\text{7FFF FFFF}) \mid (y << 31) \\ x &\leftarrow ((x << 1) >>> 1) \mid (y << 31) \end{aligned}$$

Question 8 : Effacer les bits de poids intermédiaires (b_{30} à b_{23})

$$x \leftarrow x \,\&\, \mathbf{0x}\text{807F FFFF}$$

Question 9 : Transférer l'ensemble des bits de poids intermédiaires (b_{30} à b_{23}) dans les bits de poids faibles

$$\begin{aligned} x &\leftarrow (y >>> 23)\ \&\ \mathbf{0x}\text{FF}) \\ x &\leftarrow (y << 1) >>> 24 \end{aligned}$$

Question 10 : Transférer les bits de poids faibles dans l'ensemble des bits de poids intermédiaires (b_{30} à b_{23})

$$\begin{aligned} x &\leftarrow (x\ \&\ \mathbf{0x}807\text{F FFFF})\ |\ ((y\ \&\ \mathbf{0x}\text{FF}) << 23) \\ x &\leftarrow (x\ \&\ \mathbf{0x}807\text{F FFFF})\ |\ ((y << 24) >>> 1) \end{aligned}$$

Question 11 : Effacer l'ensemble des bits de poids faibles (b_{22} à b_0)

$$\begin{aligned} x &\leftarrow x\ \&\ \mathbf{0x}\text{FF80 0000} \\ x &\leftarrow (x >>> 23) << 23 \end{aligned}$$

Question 12 : Transférer l'ensemble des poids faibles dans l'ensemble des bits de poids faibles (b_{22} à b_0)

$$\begin{aligned} x &\leftarrow (x\ \&\ \mathbf{0x}\text{FF80 0000})\ |\ (y\ \&\ \mathbf{0x}7\text{F FFFF}) \\ x &\leftarrow ((x >>> 23) << 23)\ |\ ((y << 9) >>> 9) \end{aligned}$$

Question 13 : Transférer l'ensemble des bits de poids faibles (b_{22} à b_0) dans l'ensemble des poids faibles

$$\begin{aligned} x &\leftarrow \mathbf{0x}80\,0000\ |\ (y\ \&\ \mathbf{0x}7\text{F FFFF}) \\ x &\leftarrow \mathbf{0x}80\,0000\ |\ ((y << 9) >>> 9) \end{aligned}$$

Question 14 : Donnée de taille inconnue Pour mettre à zéro les bits 2 et 3, nous devons conserver les autres bits à leur valeur. Ceci est possible en masquant les bits 2 et 3 en utilisant la fonction logique et bit à bit.

$$\begin{aligned} x &\leftarrow x\ \&\ masque \\ x &\leftarrow x\ \&\ \mathbf{0b}11\ldots11\,0011 \\ x &\leftarrow x\ \&\ \mathbf{0x}\text{F}\ldots\text{FFF3} \end{aligned}$$

Ne connaissant pas la taille de la donnée, nous ne connaissons pas le nombre de bit à 1 ou de digit F à gauche du digit 3. Pour résoudre cette difficulté, nous savons que les bits de gauche à 0 ne sont pas représentés dans un nombre. Nous devons transformer les bits à 1 de gauche en des bits à 0 dans le masque, puis les transformer ensuite à l'aide de la fonction non bit à bit.

$$\begin{aligned} masque &\leftarrow \mathbf{0b}11\ldots11\,0011 \\ masque &\leftarrow\ \sim \mathbf{0b}1100 \end{aligned}$$

Nous en déduisons donc l'expression recherchée, indépendante de la taille de la taille de la donnée.

$$\begin{aligned} x &\leftarrow x\ \&\ (\ \sim \mathbf{0b}1100) \\ x &\leftarrow x\ \&\ (\ \sim \mathbf{0x}\text{C}) \end{aligned}$$

Question 15 : Rotation de un rang à droite

$$x \leftarrow (y >>> 1) \mid (y << 31)$$

Question 16 : Rotation de un rang à gauche

$$x \leftarrow (y << 1) \mid (y >>> 31)$$

Question 17 : Rotation de n rangs

$$\begin{aligned} x_{droite} &\leftarrow (y >>> n) \mid (y << (32 - n)) \\ x_{gauche} &\leftarrow (y << n) \mid (y >>> (32 - n)) \end{aligned}$$

Question 18 : Fusion

1.

$$r \leftarrow (b \,\&\, m) \mid (a \,\&\, \sim m)$$

2. Optimisation

$$\begin{aligned} r &\leftarrow a \hat{} ((a \hat{} b) \,\&\, m) \\ &\leftarrow (a \hat{} b) \text{ positionne à 1 les bits différents de } a \text{ et } b \\ &\leftarrow ((a \hat{} b) \,\&\, m) \text{ conserve les bits différents aux endroits où } b \text{ est résultat} \\ &\leftarrow a \hat{} ((a \hat{} b) \,\&\, m) \text{ inverse les bits de } a \text{ là ou ils sont différents dans } b \end{aligned}$$

Question 19 : *bitCount*

Algorithme 16 Comptage du nombre de bits à 1

```
1: Fonction bitCount (nombre : entier) : entier
2: Variables
3:    résultat : entier
4: Début
5: résultat ← 0
6: Tant Que nombre ≠ 0 Faire
7:    résultat ← résultat + (nombre & 1)
8:    nombre ← nombre >>> 1
9: Fin Tant Que
10: Retourner résultat
11: Fin fonction
```

Question 20 : *numberOfTrailingZeros*

Algorithme 17 Nombre de zéros à droite

```
1:  Fonction numberOfTrailingZeros (nombre : entier) : entier
2:  Variables
3:     résultat : entier
4:  Début
5:  Si nombre = 0 Alors
6:     Retourner 32
7:  FinSi
8:  résultat ← 0
9:  Tant Que (nombre & 1) = 0 Faire
10:    résultat ← résultat + 1
11:    nombre ← nombre >>> 1
12: Fin Tant Que
13: Retourner résultat
14: Fin fonction
```

Question 21 : *numberOfLeadingZeros*

Algorithme 18 Nombre de zéros à gauche

```
1:  Fonction numberOfLeadingZeros (nombre : entier) : entier
2:  Variables
3:     résultat : entier
4:  Début
5:  Si nombre = 0 Alors
6:     Retourner 32
7:  FinSi
8:  résultat ← 0
9:  Tant Que (nombre & 0x8000 0000) = 0 Faire
10:    résultat ← résultat + 1
11:    nombre ← nombre << 1
12: Fin Tant Que
13: Retourner résultat
14: Fin fonction
```

Question 22 : *lowestOneBit*

Algorithme 19 Poids du bit à 1 le plus à droite

```
1: Fonction lowestOneBit (nombre : entier) : entier
2: Variables
3:    résultat : entier
4: Début
5: Si nombre = 0 Alors
6:    Retourner 0
7: FinSi
8: résultat ← 1
9: Tant Que (nombre & resultat) = 0 Faire
10:   résultat ← résultat << 1
11: Fin Tant Que
12: Retourner résultat
13: Fin fonction
```

Question 23 : *hightestOneBit*

Algorithme 20 Poids du bit à 1 le plus à gauche

```
1: Fonction hightestOneBit (nombre : entier) : entier
2: Variables
3:    résultat : entier
4: Début
5: Si nombre = 0 Alors
6:    Retourner 0
7: FinSi
8: résultat ← 0x8000 0000
9: Tant Que (nombre & resultat) = 0 Faire
10:   résultat ← résultat >>> 1
11: Fin Tant Que
12: Retourner résultat
13: Fin fonction
```

Question 24 : *reverse*

Algorithme 21 Inversion des bits

```
1: Fonction reverse (nombre : entier) : typeentier
2: Variables
3:    i, résultat : entier
4: Début
5: résultat ← 0
6: Pour i de 0 à 31 Faire
7:    résultat ← (résultat << 1) | (nombre & 1)
8:    nombre ← nombre >>> 1
9: Fin Pour
10: Retourner résultat
11: Fin fonction
```

Question 25 : *reverseBytes*

Algorithme 22 Inversion des octets

```
1: Fonction reverseBytes (nombre : entier) : typeentier
2: Début
3: retourner ((nombre << < 24) | (nombre >> 24) | ((nombre & 0xFF00) << 8)
4:     | ((nombre & 0xFF 0000) >>> 8)).
5: Fin fonction
```

Question 26 : *fastBitCount*

Algorithme 23 Comptage rapide du nombres de bit à 1

```
1: Fonction fastBitCount (nombre : entier) : entier
2: Début
3: nombre ← nombre & 0x5555 5555 + ((nombre >>> 1) & 0x5555 5555)
4: nombre ← nombre & 0x3333 3333 + ((nombre >>> 2) & 0x3333 3333)
5: nombre ← nombre & 0x0F0F 0F0F + ((nombre >>> 4) & 0x0F0F 0F0F)
6: nombre ← nombre & 0x00FF 00FF + ((nombre >>> 8) & 0x00FF 00FF)
7: nombre ← nombre & 0x0000FFFF + ((nombre >>> 16) & 0x0000 FFFF)
8: retourner(nombre)
9: Fin Fonction
```

Question 27 : *forDigit*

Algorithme 24 Conversion d'un digit en un caractère

```
1: Fonction forDigit (nombre : entier) : caractère
2: Début
3: Si nombre > 9 Alors
4:    Retourner (caractère)('A'-10+nombre)
5: Sinon
6:    Retourner (caractère)('0'+nombre)
7: FinSi
8: Fin fonction
```

Question 28 : *digit*

Algorithme 25 Conversion d'un caractère en un digit

```
1: Fonction digit (car : caractère) :entier
2: Début
3: Si (car) >' 9' Alors
4:    Retourner (entier)car −' A' + 10)
5: Sinon
6:    Retourner (entier)car −' 0')
7: FinSi
8: Fin fonction
```

Question 29 : *toString*

Algorithme 26 Conversion d'un nombre non signé en une chaîne de caractères base donnée

```
1: Fonction toString (nombre, base : entier) : chaîne de caractères
2: Variables
3:    chaîne : chaîne de caractères
4: Début
5: chaîne ← ""
6: Répéter
7:    chaîne ← forDigit(nombre mod) + chaîne
8:    nombre ← nombre/base
9: Jusqu'à nombre = 0
10: Retourner chaîne
11: Fin fonction
```

Question 30 : *toHexString*

Algorithme 27 Conversion d'un nombre non signé en une chaîne de caractères hexadécimaux

```
1: Fonction toHexString (nombre : entier) : chaîne de caractères
2: Variables
3:    chaîne : chaîne de caractères
4: Début
5: chaîne ← ""
6: Répéter
7:    chaîne ← forDigit(nombre & 0xF) + chaîne
8:    nombre ← nombre >>> 4
9: Jusqu'à nombre = 0
10: Retourner chaîne
11: Fin Fonction
```

Question 31 : *toBinaryString*

Algorithme 28 Conversion d'un nombre non signé en une chaîne de caractères binaire

```
1: Fonction toBinaryString (nombre : entier) : chaîne de caractères
2: Variables
3:    chaîne : chaîne de caractères
4: Début
5: chaîne ← ""
6: Répéter
7:    chaîne ← (caractère)((nombre & 1) +' 0') + chaîne
8:    nombre ← nombre >>> 1
9: Jusqu'à nombre = 0
10: Retourner chaîne
11: Fin fonction
```

Question 32 : *parseInt*

Algorithme 29 Conversion d'une chaîne de caractère base quelconque en un nombre non signé

```
1: Fonction parseInt (chaîne : chaîne de caractères,base : entier) : entier
2: Variables
3:    résultat, position : entier
4: Début
5: résultat ← 0
6: Pour position de 0 à Taille(chaîne)-1 Faire
7:    résultat ← résultat × base + digit(Caractère(chaîne, position))
8: Fin Pour
9: Retourner résultat
10: Fin fonction
```

Question 33 : Additionneur complet avec des portes NAND La table de vérité présenté sur la figure FIG 2.45 donne l'état des sorties **co** et **r** pour l'additionneur complet.

c_{in}	a	b	c_{out}	r
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 2.45 – Table de vérité de l'additionneur complet, correction

$$
\begin{aligned}
c_{out} &= \overline{\overline{a.b + c_{in}.a + c_{in}.b}} \\
&= \overline{\overline{a.b}.\overline{c_{in}.a}.\overline{c_{in}.b}} \\
r &= \overline{c_{in}}.\overline{a}.b + \overline{c_{in}}.a.\overline{b} + c_{in}.\overline{a}.\overline{b} + c_{in}.a.b \\
&= \overline{\overline{\overline{c_{in}}.\overline{a}.b + \overline{c_{in}}.a.\overline{b} + c_{in}.\overline{a}.\overline{b} + c_{in}.a.b}} \\
&= \overline{\overline{\overline{c_{in}}.\overline{a}.b}.\overline{\overline{c_{in}}.a.\overline{b}}.\overline{c_{in}.\overline{a}.\overline{b}}.\overline{c_{in}.a.b}}
\end{aligned}
$$

Question 34 : Égalité n bits avec des portes NAND La table de vérité présenté sur la figure FIG 2.46 donne l'état de la sortie d'un comparateur 1 bit. Pour un comparateur n bits, nous devons vérifier que tous les bits sont identiques.

a_i	b_i	$Equal$
0	0	1
0	1	0
1	0	0
1	1	1

FIGURE 2.46 – Table de vérité égalité un bit, correction

$$
\begin{aligned}
(2.20) \qquad egal &= (\overline{a_{31}}.\overline{b_{31}} + a_{31}.b_{31}).(\overline{a_{30}}.\overline{b_{30}} + a_{30}.b_{30}).\ldots.(\overline{a_0}.\overline{b_0} + a_0.b_0) \\
(2.21) \qquad &= \overline{(a_{31} \oplus b_{31})}.\overline{(a_{30} \oplus b_{30})}.\ldots.\overline{(a_0 \oplus b_0)} \\
(2.22) \qquad &= \overline{\overline{(\overline{a_{31}}.\overline{b_{31}} + a_{31}.b_{31}).(\overline{a_{30}}.\overline{b_{30}} + a_{30}.b_{30}).\ldots.(\overline{a_0}.\overline{b_0} + a_0.b_0)}} \\
(2.23) \qquad &= \overline{\overline{(\overline{a_{31}}.\overline{b_{31}} + a_{31}.b_{31})} + \overline{(\overline{a_{30}}.\overline{b_{30}} + a_{30}.b_{30})} + \ldots + \overline{(\overline{a_0}.\overline{b_0} + a_0.b_0)}} \\
(2.24) \qquad &= \overline{\overline{(\overline{a_{31}}.\overline{b_{31}}}.\overline{a_{31}.b_{31})} + \overline{(\overline{a_{30}}.\overline{b_{30}}} + \overline{a_{30}.b_{30})} + \ldots + \overline{(\overline{a_0}.\overline{b_0}} + \overline{a_0.b_0)}} \\
(2.25) \qquad &= \overline{\overline{(\overline{a_{31}}.\overline{b_{31}}}.\overline{a_{31}.b_{31})}.\overline{(\overline{a_{30}}.\overline{b_{30}}} + \overline{a_{30}.b_{30})}.\ldots.\overline{(\overline{a_0}.\overline{b_0}} + \overline{a_0.b_0)}} \\
(2.26) \qquad &= \overline{\overline{(\overline{a_{31}}.\overline{b_{31}}}.\overline{a_{31}.b_{31})}.\overline{(\overline{a_{30}}.\overline{b_{30}}} + \overline{a_{30}.b_{30})}.\ldots.\overline{(\overline{a_0}.\overline{b_0}} + \overline{a_0.b_0)}}
\end{aligned}
$$

Exercice 2 : Pratique : Codage des nombres non signés

Les exercices de programmation de cette partie sont réalisées en utilisant le langage Java version 7 (ou version antérieure). SI ce n'est pas installé sur votre ordinateur, vous devez télécharger et installer le JDK 7 (Java Developpement Kit) SE (Standard Edition) `http://www.oracle.com/technetwork/java/javase/downloads/index.html`
Si vous ne souhaitez pas installer Java, vous pouvez utiliser un compilateur Java 7 en ligne `http://ideone.com/`. Dans ce cas, le nom de la classe devra obligatoirement être Main.
Pour tester l'installation, éditez le fichier source suivant et enregistrez-le dans le fichier nommé **obligatoirement** test.java

```
import java.*;
public class test {
   public static void main( String[] args ) {
   System.out.println("Hello World !");
   }
}
```

Compilez le programme avec la commande : **javac test.java**, le fichier test.class sera généré.
Lancez le programme avec la commande : **java test**
Les opérateurs et types Java sont donnés sur la figure Fig. 2.47 .

Type	Java
8 bits signé	byte
16 bits signé	short
16 bits non signé	char
32 bits signé	int
64 bits signé	long

(a) Types Java

Opérateur	Java
Division entière	/
Reste division entière	%
NON bit à bit	~
ET bit à bit	&
OU bit à bit	|
OU exclusif bit à bit	^
Décalage logique à droite	>>>
Décalage logique à gauche	<<

(b) Opérateurs Java

Figure 2.47 – Implémentation Java

Remarque 17 :

- Il n'y a pas de type non signé en JAVA. Seuls les types signés sont implémentés.
- L'opération de conversion de type est obtenue par l'utilisation de nom du type entre parenthèses devant l'expression à convertir.
- L'aide des fonctions de l'API JAVA est disponible sur `http://docs.oracle.com/javase/7/docs/api/java/lang/Character.html` et `http://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html`.

Question 1 : Expressions utilisant les opérateurs bit à bit Écrivez et testez les expressions étudiées dans l'exercice théorique. Pour afficher les résultats en hexadécimal et en binaire la *Aff* utilise la fonction *Integer.toHexString* et la fonction *ensemble*. Pour afficher un résultat en décimal vous utiliserez la fonction *System.out.println*. Lorsque nous affichons un nombre entier n, l'appel de la fonction *Integer.toString(n,10)* est effectuée automatiquement.
Pour cela complétez le code contenu dans le fichier nommé :
nombres_non_signes_expression_enonce.java

Question 2 : Algorithmes utilisant les opérateurs bits à bits Écrivez et testez les algorithmes étudiés dans l'exercice théorique.

Pour cela complétez le code contenu dans le fichier nommé :

nombres_non_signes_algo_enonce.java

Question 3 : Circuits arithmétique non signé Pour cet exercice nous utiliserons un simulateur de circuits logique nommé *logisim*. L'aide de ce simulateur est intégrée dans le menu *help*. Lisez et effectuez les exercices du tutoriel afin de comprendre la méthode de création et de simulation d'un circuit simple et de sous-circuit ou effectuez la manipulation suivante.

Pour apprendre à utiliser *logisim*, la première manipulation va permettre de créer un circuit demi-additionneur, puis des circuits d'addition complet puis d'addition 2 bits. Tout d'abord téléchargez et lancer le fichier *logisim.jar* avec la commande *java -jar logisim.jar*. Nous obtenons alors l'application présentée sur la figure Fig 2.48 qui identifie les différentes fenêtres. Vous renommerez le circuit main et changerez le nom affiché sur le composant.

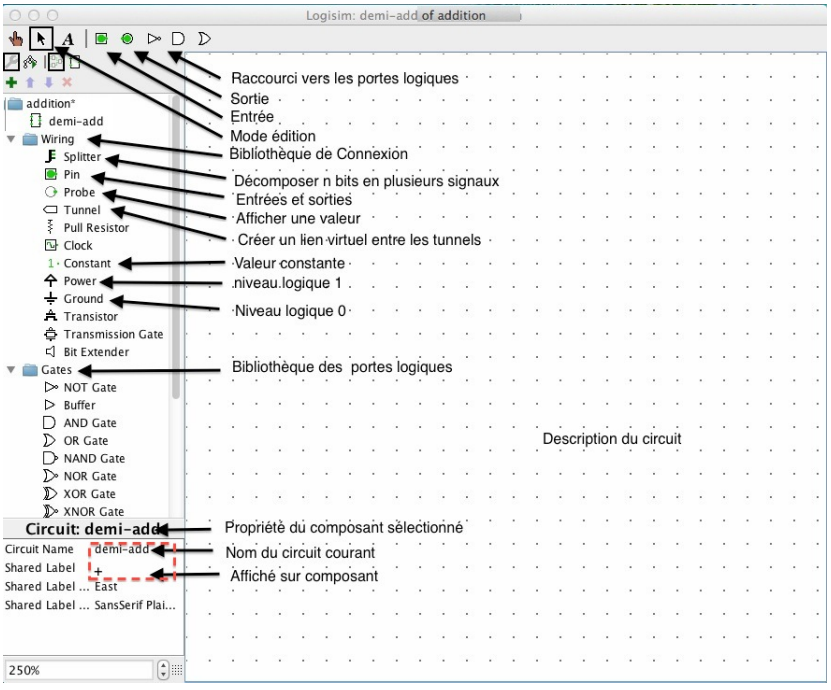


Figure 2.48 – Tutorial logisim 1

Pour créer un demi-additionneur vous placerez les entrées, les sorties les deux portes logiques et les connexions présentées sur la figure Fig 2.49 .

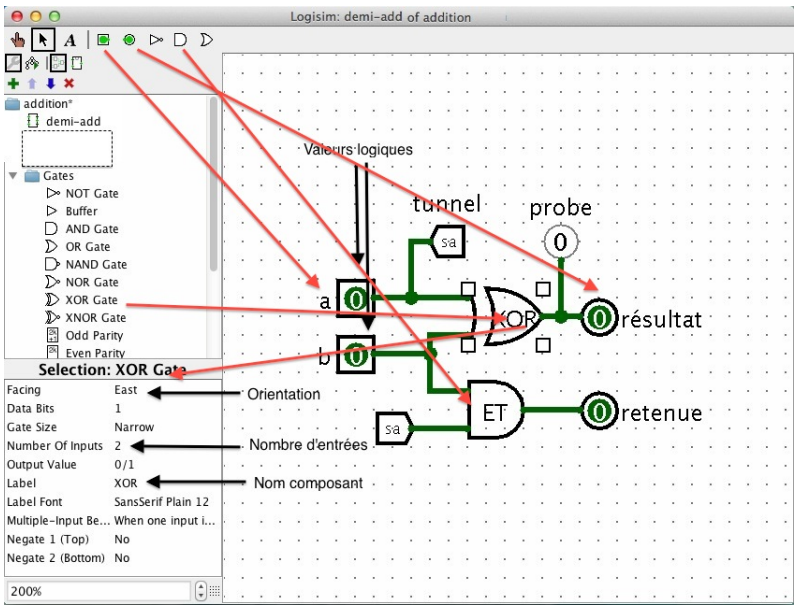


FIGURE 2.49 – Tutorial logisim 2

Bien qu'ils ne soient pas indispensables dans ce cas, vous placerez deux tunnel sur le signal nommé **a** et un probe pour afficher l'état d'une connexion présentés sur la figure FIG 2.50 .

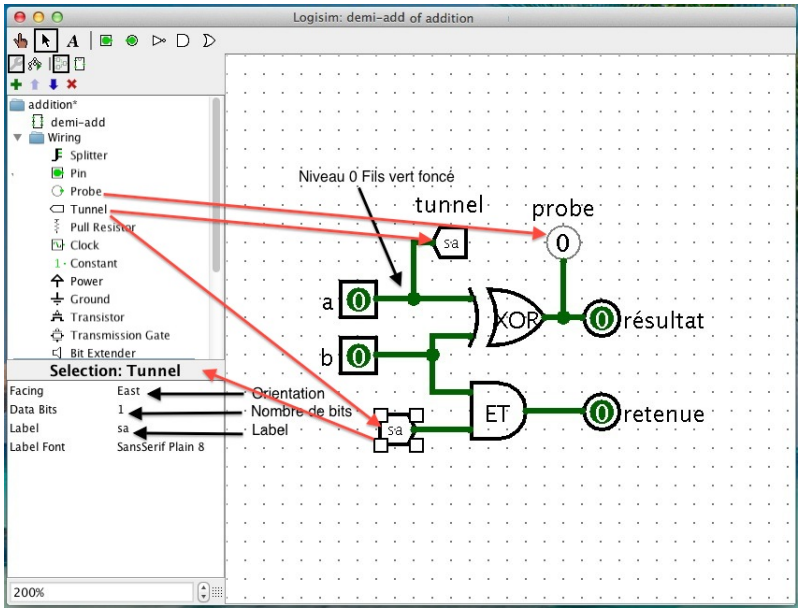


FIGURE 2.50 – Tutorial logisim 3

Pour tester le circuit vous passerez en mode simulation (le curseur est maintenant une main). En cliquant sur les entrées, vous changerez leur état logique comme le présente la figure FIG 2.51 . Si la simulation ne fonctionne pas, vous vérifierez que l'item *simulation enabled* est validé.

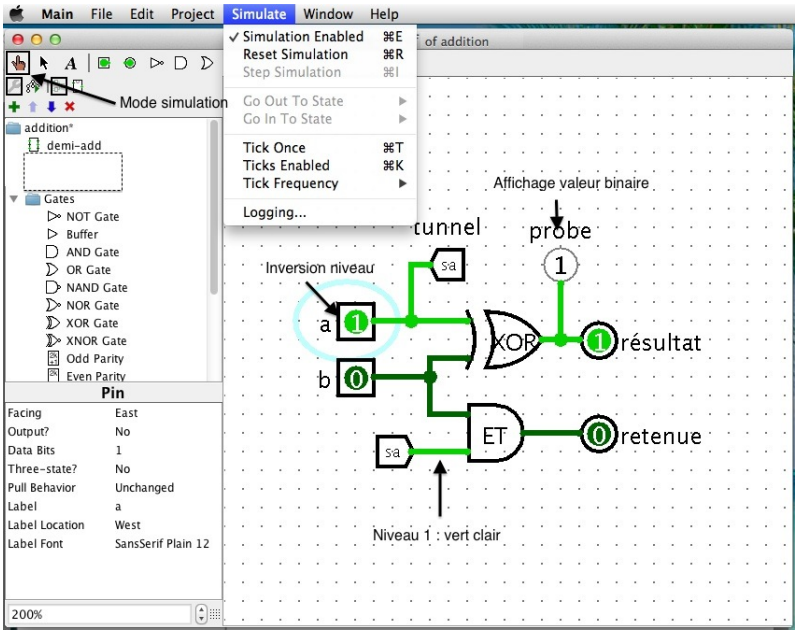


FIGURE 2.51 – Tutorial logisim 4

Nous souhaitons maintenant réutiliser le demi-additionneur pour créer un additionneur. Pour cela nous devons maintenant définir un sous-circuit demi-add. Nous passerons en mode apparence comme l'illustre la figure FIG 2.52 . Lorsque nous cliquons sur une entrée ou sortie, nous affichons un aperçu du schéma sur lequel elle est affichée. Nous pouvons déplacer les entrées et les sorties, mettre du texte et modifier la forme du circuit.

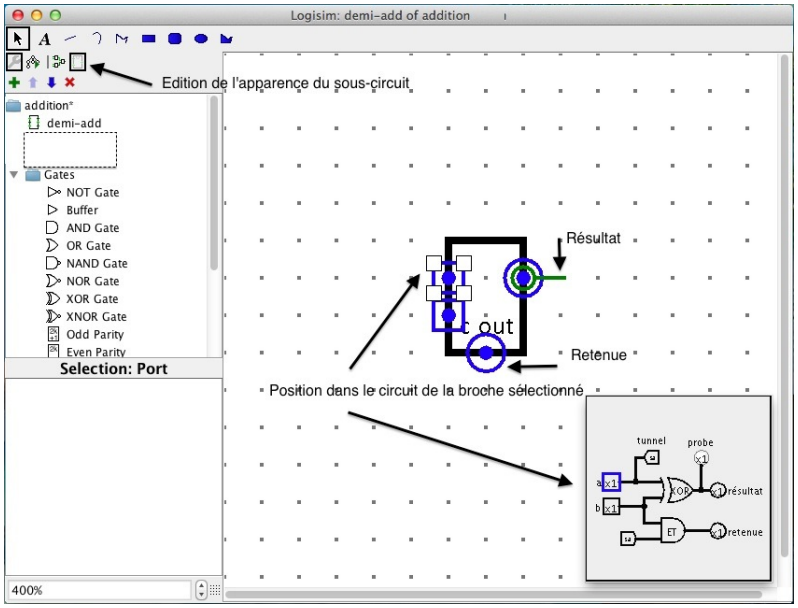


FIGURE 2.52 – Tutorial logisim 5

De la même façon, nous pouvons définir un sous-circuit additionneur avec une retenue entrante et une retenue sortante comme le présente la figure FIG 2.53 .

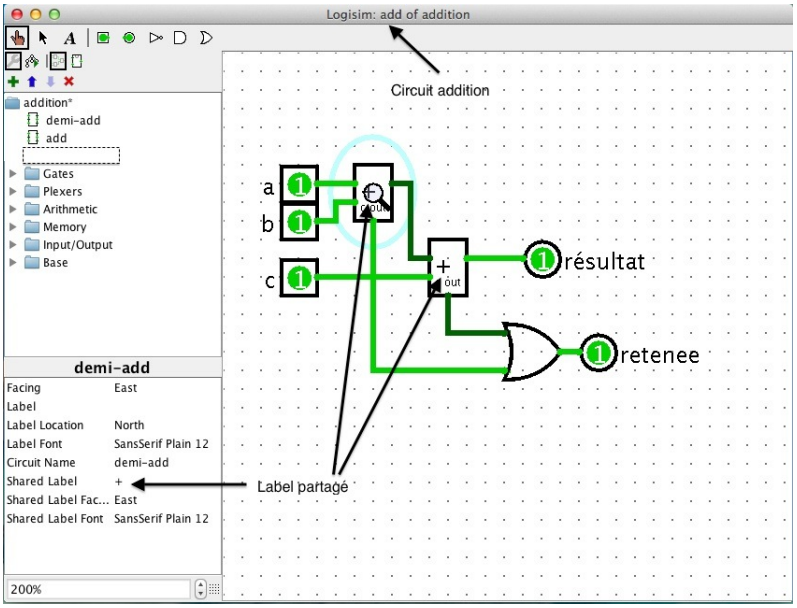
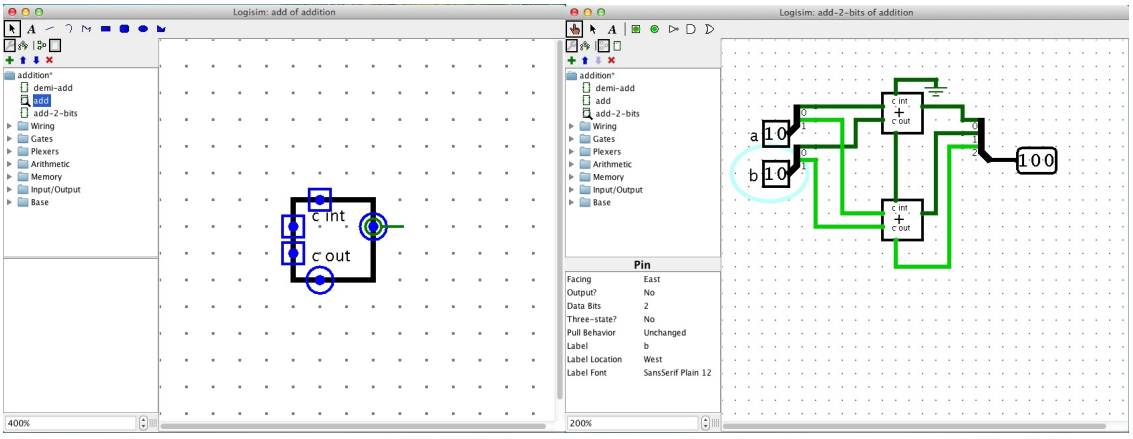


FIGURE 2.53 – Tutorial logisim 6

Pour créer un additionneur deux bits, nous utilisons deux circuits précédent et deux splitter deux bits et un splitter trois bits comme le suggère la figure FIG 2.54 .



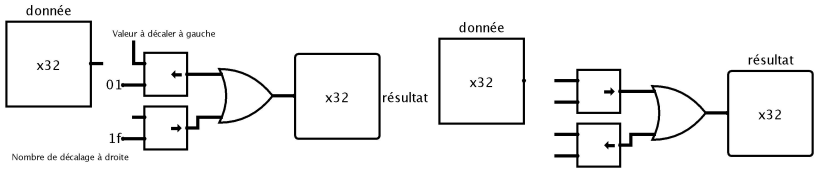
(a) Logisim 7 (b) Logisim 8

FIGURE 2.54 – Tutorial logisim 7 et 8

Implémentez les circuits permettant d'effectuer les actions suivantes. Pour cela complétez les circuits contenus dans le fichier nommé : *non_signes_enonce.circ*

1. **Rotation à 1 rang droite et à gauche** : Complétez les circuits présentés sur la figure FIG 2.55 . Vous utiliserez les équations théoriques suivantes :

$$x \leftarrow (y >>> 1) \mid (y << 31)$$
$$x \leftarrow (y << 1) \mid (y >>> 31)$$

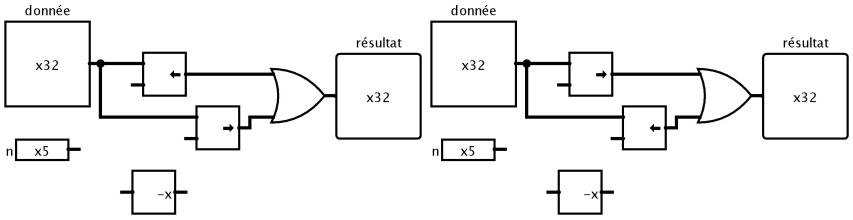


(a) Rotation 1 rang à gauche (b) Rotation 1 rang à droite

FIGURE 2.55 – Rotation un rang, énoncé

2. **Rotation de n rangs droite et à gauche (avec $0 < n < 32$)** : Complétez les circuits présentés sur la figure FIG 2.56 . Le nombre 32=**0b**10 0000 est codé sur 6 bits, et les entrées de décalages des *shifter* sur 5 bits. Nous devons supprimer le bit de poids fort est réaliser l'opération $0 - n = -n$. Vous utiliserez les équations théoriques suivantes :

$$\begin{aligned} x_{droite} &\leftarrow (y >>> n) \mid (y << (-n)) \\ x_{gauche} &\leftarrow (y << n) \mid (y >>> (-n)) \end{aligned}$$

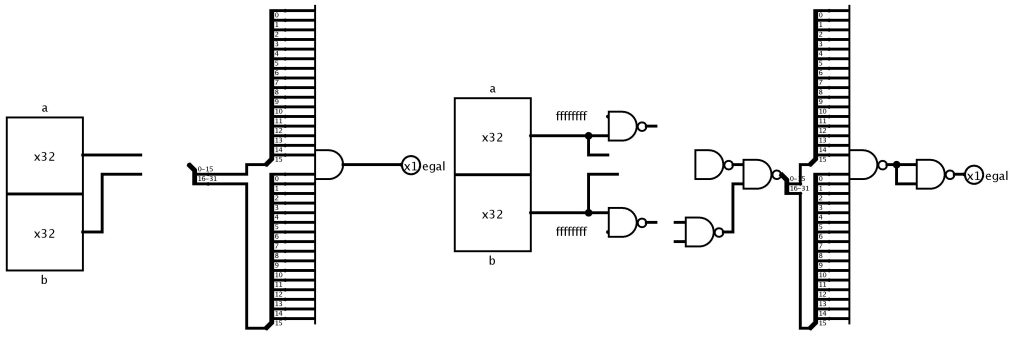


(a) Rotation n rangs à gauche (b) Rotation n rangs à droite

FIGURE 2.56 – Rotation n rangs, énoncé

3. **Comparer l'égalité de deux données 32 bits** (avec toutes les portes disponibles, puis avec des portes NAND uniquement). La figure FIG 2.64 présente le schéma permettant de réaliser l'opération ET entre les différents bits de la donnée. Complétez les figure en utiliseront les équations suivantes :

$$\begin{aligned} egal &= \overline{(a_{31} \oplus b_{31})}.\overline{(a_{30} \oplus b_{30})}.\ldots.\overline{(a_0 \oplus b_0)} \\ &= \overline{\overline{(\overline{a_{31}}.\overline{b_{31}}.a_{31}.b_{31})}.\overline{(\overline{a_{30}}.\overline{b_{30}} + a_{30}.b_{30})}.\ldots.\overline{(\overline{a_0}.\overline{b_0} + a_0.b_0)}} \end{aligned}$$

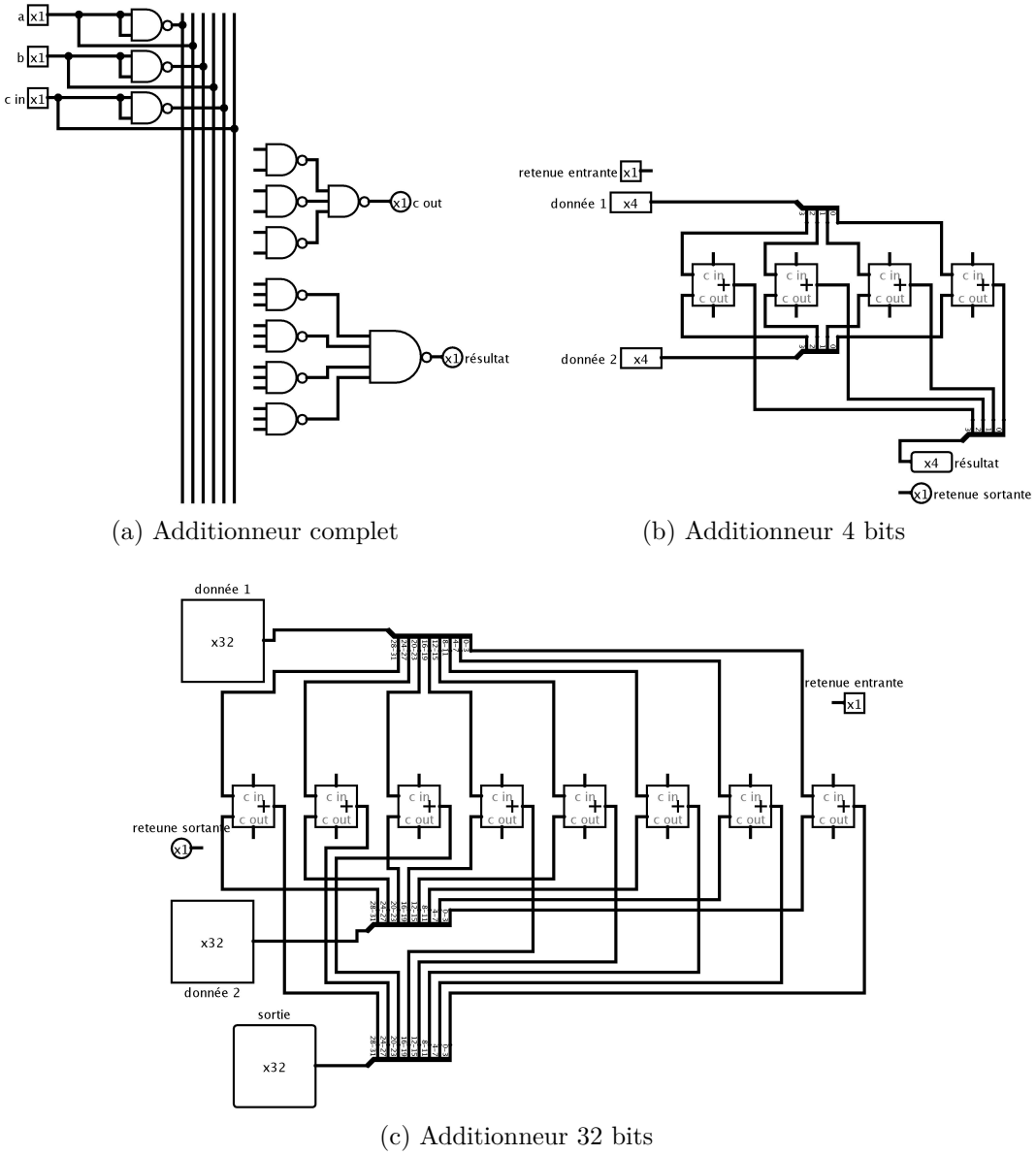


(a) Egalité (b) Egalité avec Nands

FIGURE 2.57 – Egalité de deux nombres, énoncé

4. **Additionner** trois bits avec des portes NAND, puis un additionner 4×2 bits en réutilisant le circuit précédent et additionner 32×2 bits et en utilisant les équations suivantes.

$$\begin{aligned} c_{out} &= \overline{\overline{a.b}.\overline{c_{in}.a}.\overline{c_{in}.b}} \\ r &= \overline{\overline{\overline{c_{in}}.\overline{a}.b}.\overline{\overline{c_{in}}.a.\overline{b}}.\overline{c_{in}.\overline{a}.\overline{b}}.\overline{c_{in}.a.b}} \end{aligned}$$



(a) Additionneur complet

(b) Additionneur 4 bits

(c) Additionneur 32 bits

FIGURE 2.58 – Egalité de deux nombres, énoncé

5. compter de façon rapide le nombre de bits à 1 d'une donnée 32 bits. Vous utiliserez le modèle du cours et l'algorithme des exercices théoriques.

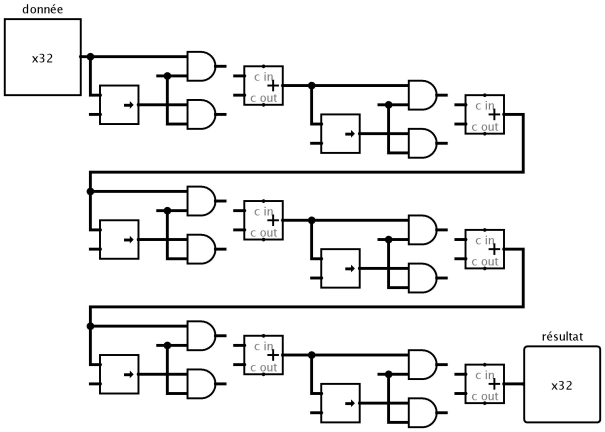


FIGURE 2.59 – Comptage rapide, énoncé

Correction exercice 2 :

Listing 2.1 – Expressions contenant des opérateurs des bit à bit

```
1  import java.*;
2  public class nombres_non_signes_expression {
3    static void ensemble(int n){
4      int i;
5      System.out.print("0b");
6      for(i=0;i<32;i++){
7        System.out.print(n >>>31);
8        if ((i==0)||(i==8))
9          System.out.print("␣");
10     n = n << 1;
11     }
12     System.out.println();
13   }
14   static void Aff(int n){
15     System.out.print("0x"+Integer.toHexString(n)+"␣␣");ensemble(n);
16   }
17   static void question(String s) {System.out.println(s);}
18   public static void main( String[] args ) {
19         int x = 0xFF00FF00  , y  = 0x12FF34FF;
20         int a = Integer.parseInt("10010001",2);int b = Integer.parseInt("111100",2);int m =
21         question("question␣1");
22   System.out.println(Integer.toString(0xC6,10));
23   System.out.println(Integer.toString(Integer.parseInt("10011", 2),10));
24   System.out.println(Integer.toString(067,10));
25   question("question␣2");
26   Aff(1); Aff(0); Aff(127); Aff(128); Aff(129);
27         question("question␣3");
28   Aff(x & y);Aff(x | y);Aff(x ^ y); Aff(x <<4 );Aff(x >>>4);
29         question("question␣4␣-␣Effacer␣MSB");
30   Aff(x | 0x80000000);
31         question("question␣5␣-␣Positionner␣MSB");
32   Aff(x & 0x7FFFFFFF);Aff((x <<1)>>>1);
33         question("question␣6␣-␣Transferer␣MSB␣->␣LSB");
34   Aff(y >>> 31);
35         question("question␣7␣-␣Transferer␣LSB␣->␣MSB");
36   Aff((x & 0x7FFFFFFF) | (y << 31));     Aff(((x << 1)>>>1) | (y << 31));
37         question("question␣8␣-␣Effacer␣b30␣a␣b23␣");
38   Aff(x&0x807FFFFF);
39         question("question␣9␣-␣Transferer␣b30␣a␣b23␣dans␣bits␣poids␣faibles");
40   Aff((y>>>23) & 0xFF); Aff((y<<1)>>>24);
41         question("question␣10␣-␣Transferer␣bits␣poids␣faibles␣dans␣b30␣a␣b23");
42   Aff((x&0x807FFFFF) | ((y & 0xFF)<<23)); Aff((x&0x807FFFFF) | ((y <<24)>>>1));
43         question("question␣11␣-␣Effacer␣b22␣a␣b0");
44   Aff(x&0xFF800000);Aff((x>>>23)<<23);
45         question("question␣12␣-␣Transferer␣b22␣a␣b0");
46   Aff((x & 0xFF800000) | (y & 0x7FFFFF)); Aff(((x>>>23)<<23) | ((y << 9)>>>9));
47         question("question␣13␣-␣Transferer␣b22␣a␣b0␣avec␣b23=1");
48   Aff((x & 0x800000)  | (y & 0x7FFFFF)); Aff((x & 0x800000)  | ((y <<9)>>>9));
49         question("question␣14␣-␣Donnee␣taille␣inconnue");
50   Aff(y & ~0xC);
51         question("question␣15␣-␣Rotation␣droite␣1␣rang");
52   Aff((y >>>1) | (y<<31));Aff(Integer.rotateRight(y,1));
53         question("question␣16␣-␣Rotation␣gauche␣1␣rang");
54   Aff((y <<1) | (y>>>31));Aff(Integer.rotateLeft(y,1));
55         question("question␣17␣-␣Rotation␣droite␣/gauche␣n␣rangs␣(n=3)");
56   Aff((y >>>3) | (y<<(32-3)));Aff(Integer.rotateRight(y,3));
```

```
57          Aff((y <<3 ) | (y>>>(32-3)));Aff(Integer.rotateLeft(y,3));
58          question("question 18 - Fusion");
59          Aff(( a & ~m) | (b & m));
60          Aff( a ^( a ^ b ) & m);
61      }
62  }
```

Listing 2.2 – Algorithmes utilisant les opérateurs bit à bit

```
1  import java.*;
2  public class nombres_non_signes_algo {
3    static int bitCount(int i){
4      int resultat=0;
5      while (i!=0){
6        resultat = resultat + (i&1);
7        i=i>>>1;
8      }
9      return resultat;
10   }
11   static int numberOfLeadingZeros(int i){
12     if (i==0) return Integer.SIZE;
13     int resultat=0;
14     while ((i&0x80000000)==0) {
15       resultat++;
16       i=i<<1;
17     }
18     return resultat;
19   }
20   static int numberOfTrailingZeros(int i){
21     if (i==0) return Integer.SIZE;
22     int resultat=0;
23     while ((i&1)==0) {
24       resultat++;
25       i=i>>>1;
26     }
27     return resultat;
28   }
29   static int lowestOneBit(int i){
30     int resultat=1;
31     if (i==0) return 0;
32     while((i & resultat)==0)
33     {
34       resultat = resultat <<1;
35     }
36     return resultat;
37   }
38   static int highestOneBit(int i){
39     int resultat=0x80000000;
40     if (i==0) return 0;
41     while((i & resultat)==0)
42     {
43       resultat = resultat >>>1;
44     }
45     return resultat;
46   }
47   static int reverse(int i){
48     int compteur,res=0;
49     for(compteur=0;compteur<Integer.SIZE;compteur++){
```

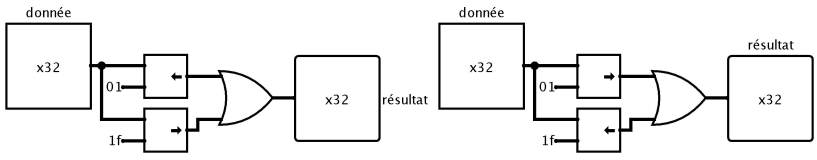
```
50        res = (res << 1) | (i & 1);
51        i = i >>> 1;
52        }
53      return res;
54    }
55    static int reverseBytes(int i) {
56              return ((i >>> 24) | ((i >>   8) &   0xFF00) |  ((i <<   8) & 0xFF0000) | (i <
57    }
58    public static int fastBitCount(int n){
59      n = (n & 0x55555555) + ((n>>>1)&0x55555555); // add 1 + 1 bits
60      n = (n & 0x33333333) + ((n>>>2)&0x33333333); // add 2 + 2 bits
61      n = (n & 0x0F0F0F0F) + ((n>>>4)&0x0F0F0F0F); // add 4 + 4 bits
62      n = (n & 0x00FF00FF) + ((n>>>8)&0x00FF00FF); // add 8 + 8 bits
63      n = (n & 0x0000FFFF) + ((n>>>16)&0x0000FFFF); // add 16 + 16 bits
64      return n;
65    }
66    static char forDigit(int nombre){
67      if (nombre > 9)
68        return (char)(nombre+'a'-0xA);
69      else
70        return (char)(nombre+'0');
71    }
72    static int digit(char caractere){
73      if ((caractere>'9') )
74        return (int)(caractere - 'A'+0xA);
75      else
76        return (int)(caractere - '0');
77    }
78    static String toString(int nombre,int base){
79      String chaine="";
80      do{
81        chaine=forDigit(nombre%base)+chaine;
82        nombre=nombre/base;
83      }while(nombre!=0);
84      return chaine;
85    }
86    static String toHexString(int nombre){
87      String chaine="";
88      do{
89        chaine=forDigit(nombre& 0xF)+chaine;
90        nombre=nombre>>>4;
91      }while(nombre!=0);
92      return chaine;
93    }
94    static String toBinaryString(int nombre){
95      String chaine="";
96      do{
97        chaine=(char)((nombre & 1)+'0')+chaine;
98        nombre=nombre>>>1;
99      }while(nombre!=0);
100     return chaine;
101   }
102   static int parseInt(String chaine,int base){
103     int position,resultat=0;
104     for(position=0;position<chaine.length();position++){
105       resultat = base * resultat + digit(chaine.charAt(position));
106     }
107     return resultat;
108   }
109   public static void main( String[] args ) {
```

```
110         System.out.println("Manipulation␣des␣nombres␣32␣bits:");
111         System.out.println("Integer.bitCount(0x7F000001):␣"+Integer.bitCount(0x7F000001));
112         System.out.println("bitCount(0x7F000001):␣"+bitCount(0x7F000001));
113         System.out.println("fastBitCount(0x7F000001)␣␣␣:␣"+fastBitCount(0x7F000001));
114         System.out.println("Integer.numberOfLeadingZeros(0x7F000001)␣␣:"+Integer.numberOfLeadi
115         System.out.println("numberOfLeadingZeros(0x7F000001)␣␣:"+numberOfLeadingZeros(0x7F0000
116         System.out.println("Integer.numberOfTrailingZeros(0x7F000001)␣:"+Integer.numberOfTrail
117         System.out.println("numberOfTrailingZeros(0x7F000001)␣:"+numberOfTrailingZeros(0x7F000
118         System.out.println("Integer.highestOneBit(0x7F000001)␣:␣"+toHexString(Integer.highestO
119         System.out.println("highestOneBit(0x7F000001)␣:␣"+toHexString(highestOneBit(0x7F000001
120         System.out.println("Integer.lowestOneBit(0x7F000001)␣␣␣:␣"+toHexString(Integer.lowestO
121         System.out.println("lowestOneBit(0x7F000001)␣␣␣:␣"+toHexString(lowestOneBit(0x7F000001
122         System.out.println("Integer.reverse(0x7F000001)␣␣␣␣␣␣␣:"+toBinaryString(Integer.revers
123         System.out.println("reverse(0x7F000001)␣␣␣␣␣␣␣:"+toBinaryString(reverse(0x7F000001)));
124         System.out.println("Integer.reverseBytes(0x7F000001)␣␣:"+toHexString(Integer.reverseBy
125         System.out.println("reverseBytes(0x7F000001)␣␣:"+toHexString(reverseBytes(0x7F000001))
126         System.out.println("Test␣des␣fonctions␣forDigit␣et␣digit␣:");
127         System.out.println("Character.forDigit(2)␣␣␣␣␣␣␣␣␣␣:"+Character.forDigit(2,16));
128         System.out.println("forDigit(2)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:"+forDigit(2));
129         System.out.println("Character.forDigit(11)␣␣␣␣␣␣␣:"+Character.forDigit(11,16));
130         System.out.println("forDigit(11)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:"+forDigit(11));
131         System.out.println("Character.digit('2')␣␣␣␣␣␣␣␣␣␣␣:"+Character.digit('2',16));
132         System.out.println("digit('2')␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:"+digit('2'));
133         System.out.println("Character.digit('B')␣␣␣␣␣␣␣␣␣␣␣:"+Character.digit('B',16));
134         System.out.println("digit('B')␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:"+digit('B'));
135         System.out.println("Test␣des␣fonctions␣to...String␣:");
136         System.out.println("Integer.toString(127,10)␣␣␣␣␣:"+Integer.toString(127,10));
137         System.out.println("toString(127,10)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:"+toString(127,10));
138         System.out.println("Integer.toString(127,16)␣␣␣␣␣:"+"0x"+Integer.toString(127,16));
139         System.out.println("toString(127,16)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:"+"0x"+toString(127,16));
140         System.out.println("Integer.toBinaryString(127)␣:"+"0b"+Integer.toBinaryString(127));
141         System.out.println("toBinaryString(127)␣␣␣␣␣␣␣␣␣␣␣␣:"+"0b"+toBinaryString(127));
142         System.out.println("Integer.toHexString(127)␣␣␣␣␣:"+"0x"+Integer.toHexString(127));
143         System.out.println("toHexString(127)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:"+"0x"+toHexString(127));
144         System.out.println("Test␣des␣fonction␣parseInt␣:");
145         System.out.println("Integer.parseInt(''1111111'',2):"+Integer.parseInt("1111111",2));
146         System.out.println("parseInt(''1111111'',2)␣␣␣␣␣␣␣:"+parseInt("1111111",2));
147         System.out.println("Integer.parseInt(''7F'',16)␣␣␣:"+Integer.parseInt("7F",16));
148         System.out.println("parseInt(''7F'',16)␣␣␣␣␣␣␣␣␣␣␣␣␣:"+parseInt("7F",16));
149     }
150 }
```

Question 3 : Circuits arithmétique non signé

1. rotation à 1 rang droite et à gauche,

   Nous devons placer la valeur à décaler et les constantes 1 et 31 (ou **0x**1F) sur circuits de décalages non signé comme le suggère le la figure Fig 2.60 .

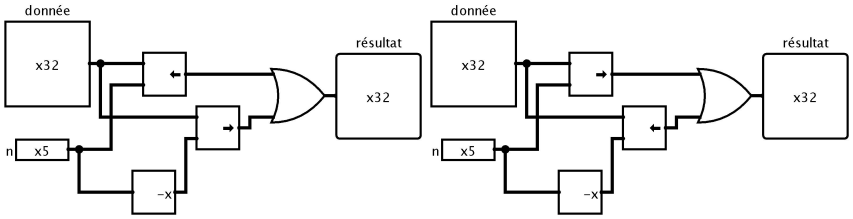


(a) Rotation 1 rang à gauche (b) Rotation 1 rang à droite

Figure 2.60 – Rotation un rang, correction

2. rotation de n rangs droite et à gauche (avec $0 < n < 32$)

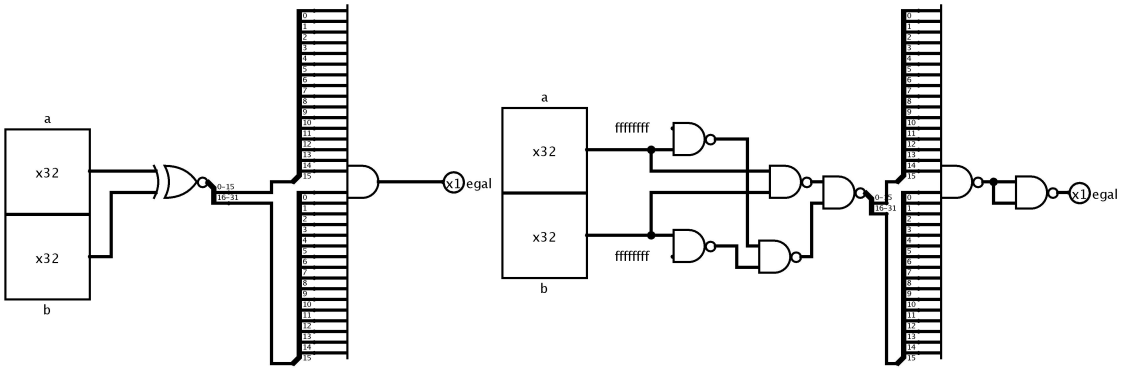
   Les circuits sont les mêmes que pour les rotations de 1 rang, sauf que nous plaçons la valeur $-n$ sur les circuits de décalage comme le suggère le la figure Fig 2.61 .



(a) Rotation n rangs à gauche (b) Rotation n rangs à droite

Figure 2.61 – Rotation n rangs, correction

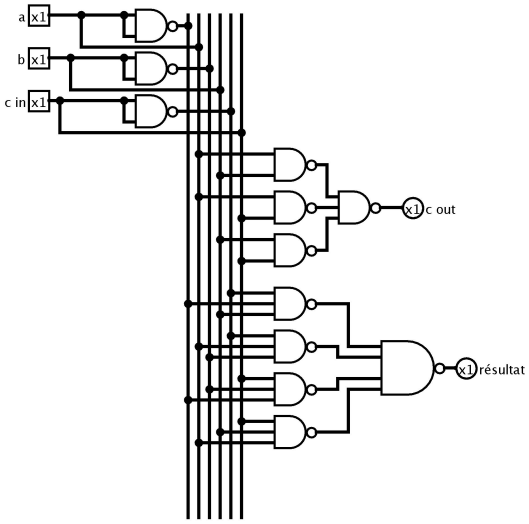
3. comparer l'égalité de deux données 32 bits (avec toutes les portes, puis avec des portes NAND uniquement)



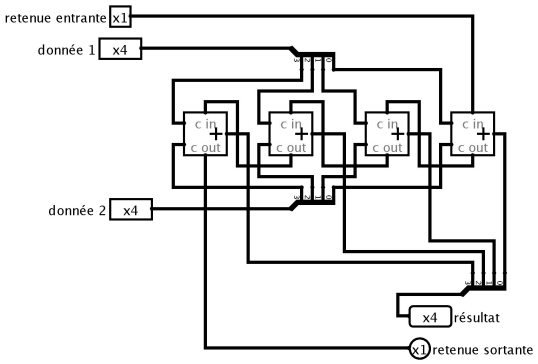
(a) Egalité (b) Egalité avec Nands

Figure 2.62 – Egalité de deux nombres, correction

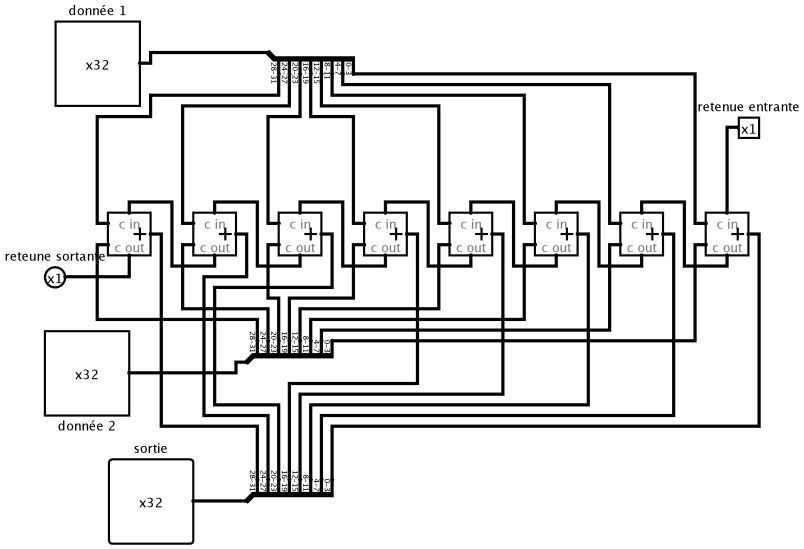
4. Additionner complet avec des portes NAND, puis un additionneur 4×2 bits réutilisant le circuit précédent et un additionneur 32×2 bits.



(a) Additionneur complet



(b) Additionneur 4 bits



(c) Additionneur 32 bits

FIGURE 2.63 – Egalité de deux nombres, correction

5. Compter de façon rapide le nombre de bits à 1 d'une donnée 32 bits.

   Nous devons placer les masques vu dans les exercices théoriques et le nombre de décalage correspondants.

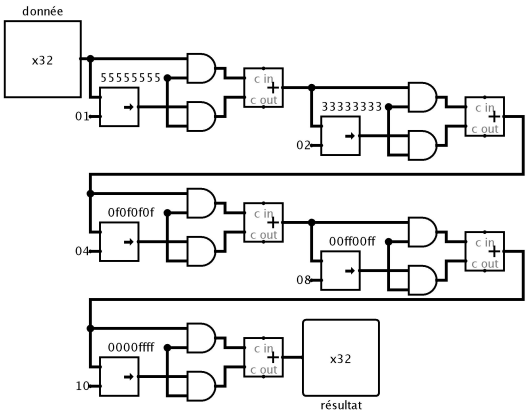


FIGURE 2.64 – Comptage rapide, correction

Chapitre 3

Codage des nombres entiers signés ($\in \mathbb{Z}$)

3.1 Principe de codage

Pour coder le signe d'un nombre entier, quatre méthodes sont possibles :

- **Valeur absolue et signe** : Le bit de poids fort représente le signe (négatif si **MSB**=1). Ce codage est très pratique pour multiplication et division
- **Complément à 1** : Le nombre est codé en inversant tous les bits. Ce codage génère une erreur de 1 lors des opérations d'addition et de soustraction et n'est JAMAIS utilisé.
- **Complément à 2** : Cette représentation corrige le défaut du complément à 1. Les opérations addition et soustraction sont simplifiées mais la gestion du signe de multiplication et la division reste complexe. Cette méthode est très utilisée
- **Codage avec excédent** : Dans ce cas nous ajoutons une valeur positive nommée *excédent* au nombre signé. De cette façon la valeur codée est toujours positive. Cette méthode est utilisée pour coder l'exposant des nombres réels.

Dans les ordinateurs la représentation en complément à 2 est couramment adoptée. Nous étudierons uniquement cette représentation dans ce chapitre du cours. Un nombre signé binaire, est un ensemble de taille **définie** de bits dont le bit de poids fort **MSB** aura le rôle de bit de signe. Son poids sera négatif.
Un nombre signé binaire codé sur n bits sera noté :

$$\pm d_{n-1}d_{n-2}....d_i...d_1d_0 \qquad \text{avec } 0 \leq d_i \leq 1 \tag{3.1}$$

3.2 Conversion en une valeur décimale

La valeur d'un nombre signé aura pour valeur décimale :

$$\begin{aligned} \text{N} &= -d_{n-1} \times 2^{n-1} + d_{n-2} \times 2^{n-2} + ...d_i \times 2^i ... + d_1 \times 2^1 + d_0 \times 2^0 \\ &\text{avec } i \subset [0, n-1] \text{ nommé le rang du chiffre} \\ &\text{et } 2^i \text{ nommé poids du chiffre de rang } i. \\ &d_{n-1} \text{ est le bit de signe de poids } -2^{n-1} \end{aligned} \tag{3.2}$$

Exemple 29 :

Soit un nombre signé codé sur 8 bits

$$\begin{aligned} \mathbf{0b}1111\,1011 &= -1\times 2^7+1\times 2^6+1\times 2^5+1\times 2^4+1\times 2^3+0\times 2^2+1\times 2^1+1\times 2^0 \\ &= -128+64+32+16+8+2+1 \\ &= -5 \end{aligned}$$

3.3 Calcul du complément à deux

Pour changer le signe d'un nombre, nous utilisons la méthode **du complément à deux**. Le complément à deux est la valeur obtenue en faisant le **complément à un** (inversion des bits de la donnée) à laquelle nous ajoutons la valeur 1.

$$\text{-N} = \sim (b_{n-1}b_{n-2}b_{n-3}...b_1b_0)+1$$

Exemple 30 :
Soit un nombre codé sur un octet (8 bits)

$$\begin{aligned} 5 &= \mathbf{0b}0000\,0101 \\ -5 &= \sim(\mathbf{0b}0000\,0101)+1 \\ &= \mathbf{0b}1111\,1010+1_2 = \mathbf{0b}1111\,1011 \end{aligned}$$

3.4 Intervalle d'un nombre signé

Avec n bits nous pouvons donc coder la valeur maximale si tous les bits ayant un poids positif sont à 1 :

$$\begin{aligned} \text{Nmax} &= -0\times 2^{n-1}+1\times 2^{n-2}+....1\times 2^i...+1\times 2^1+1\times 2^0 \\ &= 2^{n-2}+....2^i...+2^1+1 \\ &= (2-1)(2^{n-2}+....2^i...+2^1+1) \\ &= 2\times(2^{n-2}+....2^i...+2^1+1)-1\times(2^{n-2}+....2^i...+2^1+1) \\ &= 2^{n-1}+....2^i...+2^1-1\times(2^{n-2}+....2^i...+2^1+1) \\ &= 2^{n-1}-1 \end{aligned} \tag{3.3}$$

Avec n bits nous pouvons donc coder la valeur minimale si le bit ayant un poids négatif est à 1 :

$$\begin{aligned} \text{Nmin} &= -1\times 2^{n-1}+0\times 2^{n-2}+....0\times 2^i...+0\times 2^1+0\times 2^0 \\ &= -2^{n-1} \end{aligned} \tag{3.4}$$

Au total avec n bits, nous pouvons donc coder les nombres signés compris entre $-2^{n-1} \leq N_2 \leq 2^{n-1}-1$, soit au total b^n valeurs (ce qui est identique au cas non signé). Ces valeurs sont données sur la figure FIG. 3.1 pour les tailles courantes.

Taille donnée	Valeur maximale	Valeur minimale
8 bits	$2^7-1=127$	$-2^7=-128$
16 bits	$2^{15}-1=32767$	$-2^{15}=-32768$
32 bits	$2^{31}-1=2147483647$	$-2^{31}=-2147483648$

FIGURE 3.1 – Étendue des valeurs des nombres signés

3.5 Opérations sur les entiers signés

3.5.1 Addition

	1	0	0	1	1	1	0	1	-99
+	0	1	1	0	1	1	0	1	+ 109
1	0	0	0	0	1	0	1	0	10

FIGURE 3.2 – Addition de deux nombres signés

Nous remarquons sur la figure FIG. 3.2 que le résultat est juste, et que la retenue non signée est positionnée à 1. **La retenue non signée ne s'applique pas aux nombres signés**. Nous verrons lors d'un exercice comment cette **retenue signée** est obtenue.

3.5.2 Opérateurs de décalage arithmétique des nombres signés notés >> et <<

Les opérateur de décalage arithmétique permettent de conserver le signe des nombres. L'opérateur de décalage arithmétique à gauche est identique à l'opérateur logique. Par contre lors de décalage arithmétique à droite le bit de signe ne doit pas changer. Il doit être conservé (voir FIG. 3.3)

Donnée	0	0	0	0	0	0	1	0	2
Donnée >> 1	0	0	0	0	0	0	0	1	1
Donnée << 1	0	0	0	0	0	1	0	0	4

(a) Nombre positif

Donnée	1	1	1	1	1	1	1	0	−2
Donnée >> 1	1	1	1	1	1	1	1	1	−1
Donnée << 1	1	1	1	1	1	1	0	0	−4

(b) Nombre négatif

FIGURE 3.3 – Décalage arithmétique à droite

Remarque 18 :

- Les décalages *arithmétique* et *logique* à gauche sont identiques et utilisent la même notation.
- Les décalages *arithmétique* et *logique* à droite sont différents et utilisent deux notations. Le bit de signe est propagé lors du décalage à droite d'un nombre signé (en rouge).

3.6 Exercices

Exercice 3 : Théorique : Codage et algorithmes sur les nombres signés

Question 1 : Binaire et hexadécimal Exprimez en binaire et en hexadécimal (format 8 et 16 bits) les nombres signés suivants : 1, 0 , -1, +127 ,+128, -128,-129. Vérifiez en évaluant le polynôme que les valeurs binaires sont correctes.

Question 2 : Retenue signée

1. Donnez les résultats des opérations suivantes en binaire 8 bits signés. $126+1$;$126+2$;$(-127)+(-1)$;$(-127)+(-2)$. Identifiez celles qui sont fausses. Pour chacune donner la retenue entrante sur le bit 7 $R_{6->7}$ et la retenue non signée notée R_{ns}.
2. Donnez la condition sur les retenues pour obtenir le dépassement signé notée D_s : $D_s \leftarrow R_{6->7}\ Op\ R_{ns}$

Question 3 : *toString* Complétez l'algorithme de la fonction nommée *toString* permettant de convertir un nombre signé en un chaîne de caractères le représentant dans une base comprise entre 2 et 16.

Algorithme 30 Conversion d'un nombre signé en une chaîne de caractères, énoncé

```
1: Fonction toString (nombre, base : entier) : chaîne de caractères
2: Variables
3:    chaîne : chaîne de caractères
4:    négatif : booléen
5: Début
6: chaîne ← ""
7: négatif ← VRAI
8: Si (...)=0) Alors
9:    nombre ← −nombre
10:   négatif ← FAUX
11: FinSi
12: Répéter
13:   chaîne ← forDigit(...(nombre mod base), base) + chaîne
14:   nombre ← nombre/base
15: Jusqu'à nombre = 0
16: Si négatif = VRAI Alors
17:   chaîne ← ... + chaîne
18: Sinon
19:   chaîne ← ... + chaîne
20: FinSi
21: Retourner chaîne
22: Fin fonction
```

Exemple 31 :

$$\begin{aligned} \textit{toString}(127,16) &= \text{"}+7F\text{"} \\ \textit{toString}(127,2) &= \text{"}+1111111\text{"} \\ \textit{toString}(\mathbf{0x}\text{FFFF FFFF},2) &= \text{"}-1\text{"} \end{aligned}$$

Indication 1 : Le nombre -2^{31} ne peut pas être converti en un nombre positif sur 32 bit, par contre le nombre $2^{31}-1$ peut être converti en un nombre négatif. Vous manipulerez un nombre négatif dans l'itération de cette fonction.
Indication 2 : Les opérations mod et / seront appliqués à un nombre négatif.
Indication 3 : Vous traiterez le signe avant et après l'itération.

Question 4 : *parseInt* Complétez l'algorithme de la fonction nommée *parseInt* permettant de convertir une chaîne de caractères représentant un entier signé en un nombre signé. La chaîne de caractères est supposée correcte. Le premier caractère est '-' ou '+'.

Algorithme 31 Conversion d'une chaîne de caractères en un nombre signé, énoncé

```
1: Fonction parseInt (chaîne : chaîne de caractères,base : entier) : entier
2: Variables
3:    résultat, position : entier
4: Début
5: résultat ← 0
6: Pour position de 1 à Taille(chaîne)-1 Faire
7:    résultat ← résultat × base...digit(Caractère(chaîne, position), base)
8: Fin Pour
9: Si (Caractère(chaîne,0) = ...) Alors
10:   nombre ← −nombre
11: FinSi
12: Retourner résultat
13: Fin fonction
```

Indication 1 : Les opérations × et + seront appliqués à un nombre négatif et l'inversion de signe sera faite avant de retourner le résultat.

Question 5 : Décalage arithmétique à droite de 1 rang Donnez l'algorithme d'une fonction nommée *sra* permettant de décaler de 1 rang à droite une donnée signée 32 bits en utilisant uniquement les opérateurs bit à bit non signés.

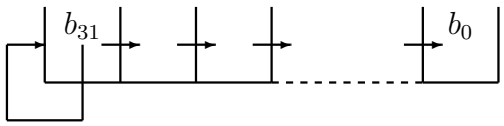


FIGURE 3.4 – Principe du décalage arithmétique à droite

Algorithme 32 Décalage arithmétique à droite de 1 rang, énoncé

```
1: Fonction sra (nombre : entier) : entier
2: Début
3: Retourner (nombre...) | (nombre...)
4: Fin fonction
```

Exemple 32 :

$$\begin{aligned} sra(\mathbf{0x}8000\,0001) &= \mathbf{0x}C000\,0000 \\ sra(\mathbf{0x}4000\,0001) &= \mathbf{0x}2000\,0000 \end{aligned}$$

Indication 1 : Vous devez conserver le bit de signe (MSB) lors du décalage à droite. Pour cela vous le conserverez et le placerez dans le résultat.
Indication 2 : Vous utilisez les opérateurs logique bit à bit ainsi que l'opérateur >>>.
Indication 3 : Vous n'utilisez pas d'alternative.

Question 6 : Décalage arithmétique à droite de n rangs Complétez l'algorithme d'une fonction nommée *sra* permettant de décaler de n rangs ($0 \leq n < 31$) à droite une donnée signée 32 bits en utilisant uniquement les opérateurs bit à bit non signés.
Fonction *sra* (*nombre* : entier, n : entier) : entier
Exemple 33 :

$$\begin{aligned} sra(\mathbf{0x}8000\,0001, 2) &= \mathbf{0x}\text{E}000\,0000 \\ sra(\mathbf{0x}4000\,0001, 2) &= \mathbf{0x}1000\,0000 \end{aligned}$$

Indication 1 : Vous utiliserez l'expression -(nombre >>> 31) pour obtenir une représentation du bit de signe.

Algorithme 33 Décalage arithmétique à droite de n rang, énoncé

```
1: Fonction sra (nombre : entier,n : entier) : entier
2: Début
3: Retourner (nombre >>> ...) | (−(nombre >>> 31)...(31 − n))
4: Fin fonction
```

Question 7 : *signum* Donnez l'algorithme d'une fonction nommée *signum* permettant de renvoyer le signe d'un nombre signé codé sur 32 bits.
Fonction *signum* (*nombre* : entier) : entier

$$signe(x) = \begin{cases} 1 & \text{si} \quad x > 0 \\ -1 & \text{si} \quad x < 0 \\ 0 & \text{si} \quad x = 0 \end{cases}$$

Indication 1 : Déterminez l'expression si le nombre est positif, s'il est négatif puis combinez les deux expressions.
Indication 2 : Si le nombre est positif nous devons retourner le bit de signe inversé.
Indication 3 : Si le nombre est négatif, utilisez l'opérateur signé >> .
Indication 4 : Vous utiliserez le nombre et le nombre de signe opposé. Vous n'utiliserez pas d'alternative, mais les opérateurs bit à bit non signés et signés.

Algorithme 34 Signe d'un nombre entier, énoncé

```
1: Fonction signum (nombre : entier) : entier
2: Début
3: Retourner (nombre...) | (−nombre...)
4: Fin fonction
```

Question 8 : *absolue* Donnez l'algorithme d'une fonction nommée *absolue* permettant de renvoyer la valeur absolue d'un nombre.
Fonction *absolue* (*nombre* : entier) : entier

$$abs(x) = \begin{cases} x & \text{si} \quad x > 0 \\ -x & \text{si} \quad x < 0 \\ 0 & \text{si} \quad x = 0 \end{cases}$$

Indication 1 : Vous n'utiliserez pas d'alternative, mais seulement les opérateurs bit à bit et la soustraction.
Indication 2 : Si le nombre est positif, retournez la valeur de l'argument, sinon soustraire deux fois l'argument.

Algorithme 35 Valeur absolue d'un nombre entier, énoncé

1: **Fonction** *absolue* (*nombre* : entier) : entier
2: **Début**
3: **Retourner** $nombre - ((nombre...)\ \&\ (nombre...))$
4: **Fin fonction**

Question 9 : *fastLowestOneBit* Donnez l'algorithme d'une fonction nommée *fastLowestOneBit* permettant de conserver uniquement le bit à 1 le plus à droite.
Fonction *fastLowestOneBit* (*nombre* : entier) : entier
Indication 1 : Vous utiliserez l'inversion de signe et l'opérateur & .
Exemple 34 :

$$\begin{aligned} 20 &= \mathbf{0b}0001\,0100 \\ -20 &= \mathbf{0b}1110\,1100 \text{ au format 8 bits} \end{aligned}$$

Algorithme 36 Poids du bit à 1 le plus à droite rapide, énoncé

1: **Fonction** *fastLowestOneBit* (*nombre* : entier) : entier
2: **Début**
3: **Retourner** $nombre...(-nombre)$
4: **Fin fonction**

Correction exercice 3 :

Question 1 : Binaire et hexadécimal

N	8 bits binaire	8 bits hexadécimal	16 bits binaire	16 bits hexadécimal
1	**0b**0000 0001	**0x**01	**0b**0000 0000 0000 0001	**0x**0001
0	**0b**0000 0000	**0x**00	**0b**0000 0000 0000 0000	**0x**0000
-1	**0b**1111 1111	**0x**FF	**0b**1111 1111 1111 1111	**0x**FFFF
127	**0b**0111 1111	**0x**7F	**0b**0000 0000 0111 1111	**0x**007F
128	impossible	impossible	**0b**0000 0000 1000 0000	**0x**0080
-128	**0b**1000 0000	**0x**80	**0b**1111 1111 1000 0000	**0x**FF80
-129	impossible	impossible	**0b**1111 1111 0111 1111	**0x**FF7F

FIGURE 3.5 – Codage de nombre signés

Question 2 : Retenue signée

126		0	1	1	1	1	1	1	0
1+									1
		0							
127	0	0	1	1	1	1	1	1	1

(a) Résultat juste et $R_{ns} = 0$ et $R_{6->7} = 0$

126		0	1	1	1	1	1	1	0
2+								1	0
			1						
-128	0	1	0	0	0	0	0	0	0

(b) Résultat faux et $R_{ns} = 0$ et $R_{6->7} = 1$

-127		1	0	0	0	0	0	0	1
+(-1)		1	1	1	1	1	1	1	1
		1							
-128	1	1	0	0	0	0	0	0	0

(c) Résultat juste et $R_{ns} = 1$ et $R_{6->7} = 1$

-127		1	0	0	0	0	0	0	1
+(-2)		1	1	1	1	1	1	1	0
		0							
+127	1	0	1	1	1	1	1	1	1

(d) Résultat faux et $R_{ns} = 1$ et $R_{6->7} = 0$

FIGURE 3.6 – Opérations sur des nombres signés

$$R_s \leftarrow R_{6->7} \neq R_{ns}$$

Question 3 : *toString*

Algorithme 37 Conversion d'un nombre signé en une chaîne de caractères

```
1: Fonction toString (nombre, base : entier) : chaîne de caractères
2: Variables
3:    chaîne : chaîne de caractères
4:    négatif : booléen
5: Début
6: chaîne ← ""
7: négatif ← VRAI
8: Si (nombre >>> 31)=0) Alors
9:    nombre ← −nombre
10:   négatif ← FAUX
11: FinSi
12: Répéter
13:   chaîne ← forDigit(−(nombre mod base), base) + chaîne
14:   nombre ← nombre/base
15: Jusqu'à nombre = 0
16: Si négatif = VRAI Alors
17:   chaîne ←' −' + chaîne
18: Sinon
19:   chaîne ←' +' + chaîne
20: FinSi
21: Retourner chaîne
22: Fin fonction
```

Question 4 : *parseInt*

Algorithme 38 Conversion d'une chaîne de caractères en un nombre signé

```
1: Fonction parseInt (chaîne : chaîne de caractères,base : entier) : entier
2: Variables
3:    résultat, position : entier
4: Début
5: résultat ← 0
6: Pour position de 1 à Taille(chaîne)-1 Faire
7:    résultat ← résultat × base − digit(Caractère(chaîne, position), base)
8: Fin Pour
9: Si (Caractère(chaîne,0) = '+') Alors
10:   nombre ← −nombre
11: FinSi
12: Retourner résultat
13: Fin fonction
```

Question 5 : Décalage arithmétique à droite de 1 rang

Algorithme 39 Décalage arithmétique à droite de 1 rang

```
1: Fonction sra (nombre : entier) : entier
2: Début
3: Retourner (nombre >>> 1) | (nombre & 0x8000 0000)
4: Fin fonction
```

Question 6 : Décalage arithmétique à droite de n rangs

Algorithme 40 Décalage arithmétique à droite de n rangs

1: **Fonction** *sra* ($nombre$: entier,n : entier) : entier
2: **Début**
3: **Retourner** $(nombre >>> n) \mid (-(nombre >>> 31) << (31 - n))$
4: **Fin fonction**

Question 7 : *signum*

Algorithme 41 Signe d'un nombre entier

1: **Fonction** *signum* ($nombre$: entier) : entier
2: **Début**
3: **Retourner** $(nombre >> 31) \mid (-nombre >>> 31)$
4: **Fin fonction**

Question 8 : *absolue*

Algorithme 42 Valeur absolue d'un nombre entier

1: **Fonction** *absolue* ($nombre$: entier) : entier
2: **Début**
3: **Retourner** $nombre - ((nombre << 1) \,\&\, (nombre >> 31))$
4: **Fin fonction**

Question 9 : *fastLowestOneBit*

Algorithme 43 Poids du bit à 1 le plus à droite rapide

1: **Fonction** *fastLowestOneBit* ($nombre$: entier) : entier
2: **Début**
3: **Retourner** $nombre \,\&\, (-nombre)$
4: **Fin fonction**

Exercice 4 : Programmation : Algorithmes sur les nombres signés

Opérateur	Java
Décalage arithmétique à droite	>>
Décalage arithmétique à gauche	<<

FIGURE 3.7 – Opérateurs Java signés

Question 1 : Algorithmes sur les nombres signés Écrivez et testez les algorithmes étudiés dans l'exercice théorique précédent.
Pour cela complétez le code contenu dans le fichier nommé :
nombres_signes_algo_enonce.java
Vous utiliserez les fonctions de l'API java *Character.forDigit* et *Character.digit*. Ces fonctions possèdent un deuxième argument qui est la base de conversion (voir documentation de l'API).

Question 2 : Circuits sur les nombres signés

1. **Décalage arithmétique à droite de un rang** Complétez le circuit permettant de la figure FIG 3.8 en utilisant l'équation de l'exercice théorique.

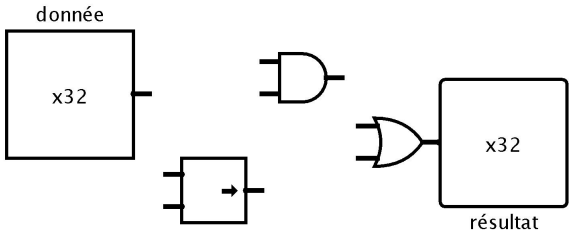


FIGURE 3.8 – Donnée >> 1, énoncé

2. **Décalage arithmétique à droite de n rang** Complétez le circuit permettant de la figure FIG 3.9 en utilisant l'équation de l'exercice théorique.

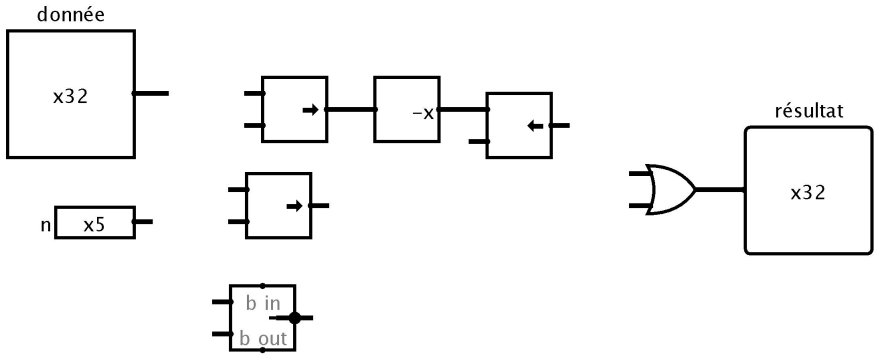


FIGURE 3.9 – Donnée >> n, énoncé

3. **Additionneur de logisim** La retenue de l'additionneur de logisim est elle signée ou non signée ? Complétez l'additionneur quatre bits non signé de la figure FIG 3.13 pour ajouter

une retenue signé qui sera utilisé pour les quatre bits de poids fort de l'additionneur 32 bits.

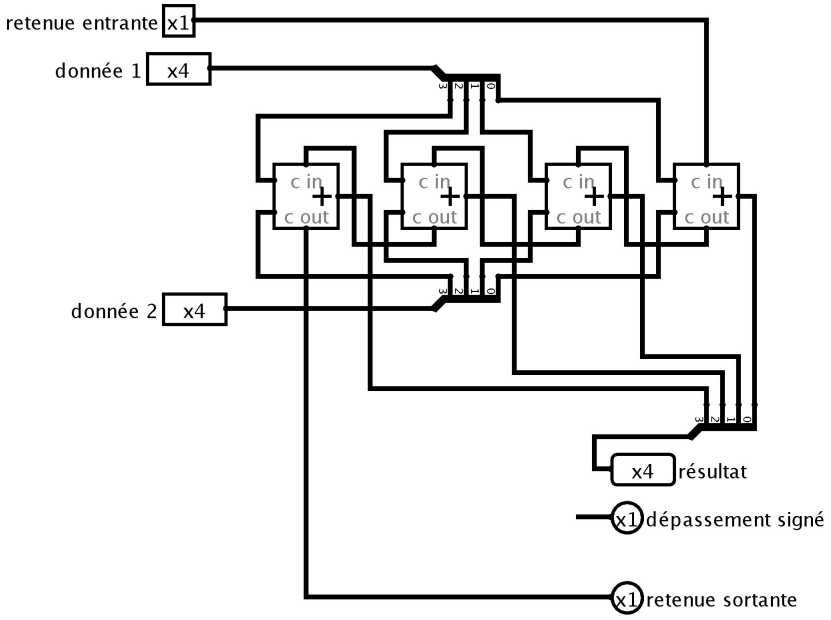


FIGURE 3.10 – Additionneur quatre bits de poids forts, énoncé

Correction exercice 4 :

Question 1 : Algorithmes sur les nombres signés

Listing 3.1 – Algorithme sur les nombres signés

```
1  import java.*;
2  public class nombres_signes_algo{
3    static int sra(int i){
4      return (i >>> 1) | (i  & 0x80000000);
5    }
6    static int sran(int i,int n){
7      return (i >>> n) | (-(i >>> 31)  << (31 - n));
8    }
9    static int signum(int i){
10     return (i>>31) | ((-i) >>> 31);
11   }
12   static int abs(int n){
13     //return (((n >> 31) & (-n)) | (((-n) >> 31) & n)); autre solution
14     return (n - ((n<<1) &(n>> 31)));
15   }
16   static int fastLowestOneBit(int i){
17     return i & (-i);
18   }
19   static String toString(int nombre,int base){
20     String chaine="";
21     boolean negatif=true;
22     if ((nombre >>> 31)==0){
23       nombre=~nombre+1;
24       negatif=false;
25     }
26     do{
27       chaine=Character.forDigit(-(nombre%base),base)+chaine;
28       nombre=nombre/base;
29     }while(nombre!=0);
30     if (negatif)
31       chaine='-'+chaine;
32     else
33       chaine='+'+chaine;
34
35     return chaine;
36   }
37   static int parseInt(String chaine,int base){
38     int position,resultat=0;
39     for(position=1;position<chaine. length();position++){
40       resultat = base * resultat + (-Character.digit(chaine. charAt(position),base));
41     }
42     if (chaine. charAt(0)=='+'){
43       resultat = ~resultat+1;
44     }
45     return resultat;
46   }
47   public static void main( String[] args ) {
48     System.out.println("Test de notre fonction toString");
49     System.out.println("Integer.toString(-1,2)):"+Integer.toString(-1,2));
50     System.out.println("toString(-1,2))           :"+toString(-1,2));
51     System.out.println("Test de notre fonction parseInt");
52     System.out.println("Integer.parseInt(+1111111,2):"+Integer.parseInt("+1111111",2));
53     System.out.println("parseInt(+1111111,2):"+parseInt("+1111111",2));
54     System.out.println("Integer.parseInt(-7f,16):"+Integer.parseInt("-7f",16));
55     System.out.println("parseInt(-7f,16):"+parseInt("-7f",16));
```

```
56        System.out.println("Integer.parseInt(+2147483647,10):"+Integer.parseInt("+2147483647",
57        System.out.println("parseInt(+2147483647,10):"+parseInt("+2147483647",10));
58        System.out.println("Integer.parseInt(-2147483648,10)"+Integer.parseInt("-2147483648",1
59        System.out.println("parseInt(-2147483648,10):"+parseInt("-2147483648",10));
60        System.out.println("Integer.parseInt(+7FFFFFFF,16):"+Integer.parseInt("+7FFFFFFF",16))
61        System.out.println("parseInt(+7FFFFFFF,16):"+parseInt("+7FFFFFFF",16));
62        System.out.println("Integer.parseInt(-10000000000000000000000000000000,2):"+Integer.pa
63        System.out.println("parseInt(-10000000000000000000000000000000,2)"+parseInt("-1000000
64        System.out.println("Test de notre fonction signum");
65        System.out.println("Integer.signum(8) :"+Integer.signum(8));
66        System.out.println("signum(8) :"+signum(8));
67        System.out.println("Integer.signum(-8) :"+Integer.signum(-8));
68        System.out.println("signum(8) :"+signum(-8));
69        System.out.println("Integer.signum(0) :"+Integer.signum(0));
70        System.out.println("signum(0) :"+signum(0));
71        System.out.println("Test de notre fonction sra");
72        System.out.println("8 >> 1: "+(8>>1));
73        System.out.println("sra(8)"+sra(8));
74        System.out.println("-8 >> 1: "+(-8>>1));
75        System.out.println("sra(-8)"+sra(-8));
76        System.out.println("Test de notre fonction sran");
77        System.out.println("8 >> 2: "+(8>>2));
78        System.out.println("sran(8,2)"+sran(8,2));
79        System.out.println("-8 >> 2: "+(-8>>2));
80        System.out.println("sran(-8,2)"+sran(-8,2));
81        System.out.println("Test de notre fonction abs");
82        System.out.println("abs(8) :"+abs(8));
83        System.out.println("abs(-8) :"+abs(-8));
84        System.out.println("abs(0) :"+abs(0));
85        System.out.println("Test de notre fonction fastLowestOneBit");
86        System.out.println("Integer.lowestOneBit(0) :"+Integer.lowestOneBit(0));
87        System.out.println("fastLowestOneBit(0) :"+fastLowestOneBit(0));
88        System.out.println("Integer.lfastLowestOneBit(0x76):"+Integer.lowestOneBit(0x76));
89        System.out.println("fastLowestOneBit(0x76) :"+fastLowestOneBit(0x76));
90    }
91 }
```

Question 2 : Circuits sur les nombres signés

1. Décalage arithmétique à droite de 1 rang

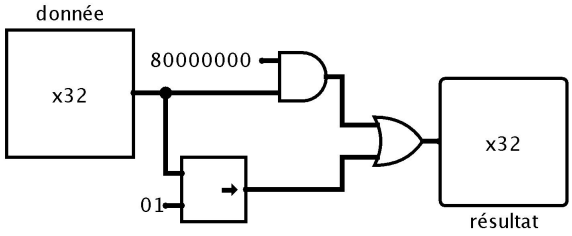


FIGURE 3.11 – Décalage arithmétique de 1 rang à droite, correction

2. Décalage arithmétique à droite de n rangs

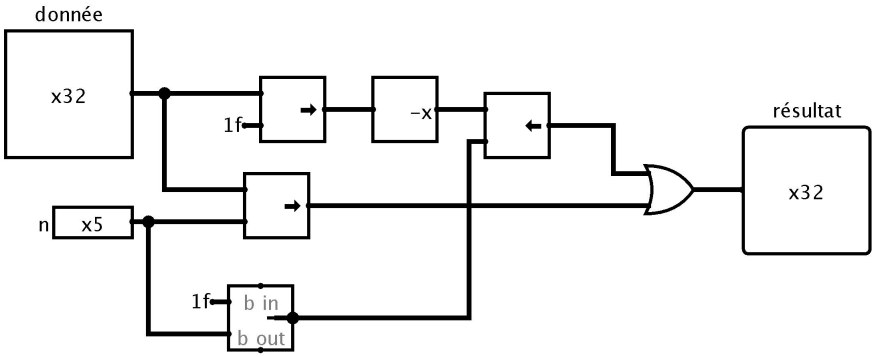


FIGURE 3.12 – Décalage arithmétique de n rangs à droite, correction

3. Retenue signée

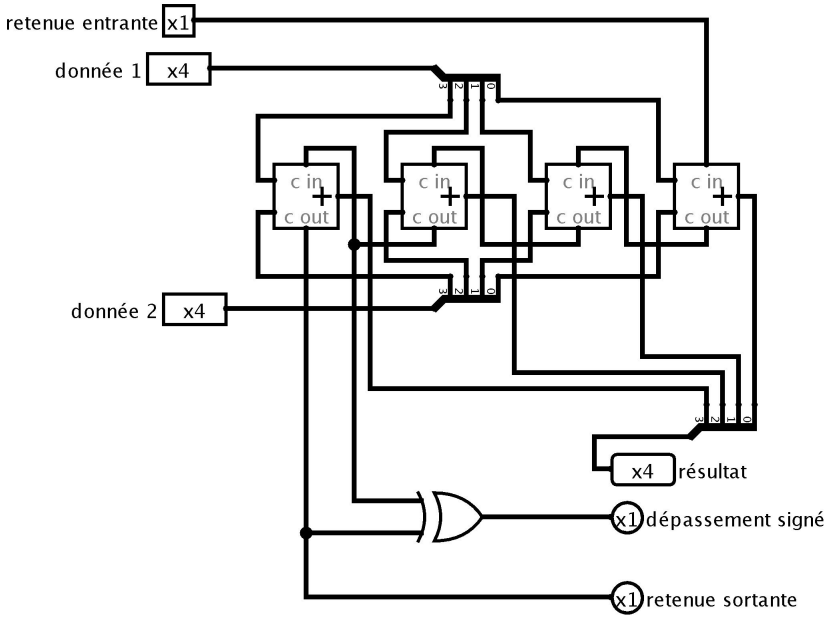


FIGURE 3.13 – Additionneur quatre bits de poids forts, correction

Nous devons remplacer l'additionneur pour les bits b_{28} à b_{31} avec celui de la figure FIG 3.13 .

Chapitre 4

Codage des nombres réels ($\in \mathbb{R}$)

4.1 Réels non signés codés en binaire

Un nombre réel x non signé est formé d'une partie entière non signée (notée $\lfloor x \rfloor$) et d'une partie fractionnaire non signée (notée $\{x\}$). Dans ce cas $x = \lfloor x \rfloor + \{x\}$.
Un nombre réel codé en binaire sur $n+m$ bits sera noté sous la forme suivante :

$$d_{n-1}d_{n-2}....d_i...d_1d_0, d_{-1}d_{-2}...d_{-m+1}d_{-m} \qquad \text{avec } 0 \leq d_i \leq 1 \tag{4.1}$$

4.1.1 Conversion en une valeur décimale

Un nombre réel aura pour expression décimale :

$$\mathrm{N} = d_{n-1} \times 2^{n-1} + ... + d_1 \times 2 + d_0 + d_{-1} \times 2^{-1} + ... + d_{-m} \times 2^{-m} \tag{4.2}$$

$d_{n-1}...d_0$ représente la partie entière (4.3)

$d_{-1}...d_{-m}$ représente la partie fractionnaire (4.4)

(4.5)

Remarque 19 :

- En binaire les poids des bits de la partie fractionnaire sont : $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}...$
- Le problème de ce codage est la perte de précision pour certaines valeurs. En effet, lors de la représentation d'un nombre très petit, les bits de la partie entière ne servent à rien. De la même façon, lors de la représentation d'un nombre très grand, les bits de la partie fractionnaire ne servent à rien.

Exemple 35 :

$$\begin{aligned} +1,25 &= \mathbf{0b}1{,}01 \\ &= 0\ldots01,010\ldots0 \text{ représentation binaire} \end{aligned} \tag{4.6}$$

4.1.2 Conversion d'une valeur décimale en sa représentation binaire

L'algorithme est proche de celui utilisé pour convertir un nombre entier non signé (algorithme des divisions successives voir Algorithme. 44) . La partie entière d'un nombre réel non signé

utilise cet algorithme, et la partie fractionnaire utilise une division par $\frac{1}{2}$. Il est plus simple de multiplier par 2 que de diviser par $\frac{1}{2}$. L'algorithme est nommé multiplications successives.

Algorithme 44 Multiplications successives

```
1: Variables
2:    base : entier
3:    nombre : réel 32 bits
4: Début
5: Lire (nombre,base)
6: Tant Que nombre ≠ 0 Faire
7:    Afficher (← ⌊nombre × base⌋)
8:    nombre ← {nombre × base}
9: Fin Tant Que
10: Fin
```

Exemple 36 :

Conversion de la partie fractionnaire du nombre 0.44_{10} en binaire :

$$\begin{aligned}
0,44 \times 2 &= 0,88 \text{ valeur bit } 0 = 0 \text{ (le poids de ce bit est 0,5)} \\
0,88 \times 2 &= 1,76 \text{ valeur bit } 1 = 1 \text{ (le poids de ce bit est 0,25)} \\
0,76 \times 2 &= 1,52 \text{ valeur bit } 1 = 1 \text{ (le poids de ce bit est 0,125)} \\
0,52 \times 2 &= 1,04 \text{ valeur bit } 1 = 1 \text{ (le poids de ce bit est 0,0625)} \\
0,04 \times 2 &= 0,08 \text{ valeur bit } 0 = 0 \text{ (le poids de ce bit est 0,03125)} \\
0,08 \times 2 &= \ldots \\
0,44_{10} &= \mathbf{0b}0,011100\ldots \\
\mathbf{0b}0,011100\ldots &= 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4} \\
&= 0,4375
\end{aligned}$$

Remarque 20 :

Le nombre $0,44$ qui se code avec un nombre de chiffres fini en décimal se code avec un nombre de chiffres infinis en binaire, d'où une perte de précision sur le résultat.

4.2 Réels signés codés en virgule flottante binaire

4.2.1 Principe

Un nombre réel signé différent de zéro peut être représenté sous forme d'une mantisse non signée (ayant un seul chiffre avant la virgule et plusieurs chiffres après la virgule), de son signe et d'un exposant signé.
Le nombre réel signé en base 2 sera codé sur n+m chiffres est noté :

$$\pm 1, d_{n-2}\ldots d_1 d_0 d_{-1} d_{-2}\ldots d_{-m} \times 2^{n-1} \tag{4.7}$$

Remarque 21 :

- Le premier chiffre de la mantisse (toujours à 1) ne sera pas représenté dans la mémoire de l'ordinateur. Cette opération se nomme *normalisation* du nombre réel.
- Le nombre 0 n'est pas normalisé. Le bit à gauche de la virgule n'est pas 1, mais 0. $0 = 0,0 \times 2^0$.

4.2.2 Spécification des formats IEEE 754

Dans la mémoire de l'ordinateur, pour coder un nombre réel signé nous devons représenter trois informations :

1. Le signe de la mantisse à gauche de la donnée,
2. La mantisse normalisée en valeur absolue à droite de la donnée,
3. L'exposant et son signe.

Signe mantisse	Exposant	Mantisse
MSB		LSB

FIGURE 4.1 – Position des champs au format IEEE 754

Deux formats ont été définis : le formats nommé simple précision ((codé sur 32 bits) et le format nommé double précision (codé sur 64 bits). La figure FIG. 4.2 spécifie les caractéristiques des nombres réels représentés au format *IEEE 754*

Précision	**simple**	**double**
TAILLE DE LA DONNEE	32 bits	64 bits
TAILLE SIGNE MANTISSE	1 bit	1 bit
TAILLE MANTISSE	23 bits	52 bits
TAILLE EXPOSANT	8 bits	11 bits
EXCEDENT	127	1023
EXP MINIMUM normalisé	-126	-1022
EXP MAXIMUM normalisé	+127	+1023
EXP MINIMUM Dénormalisé	-127	-1023
EXP MAXIMUM Dénormalisé	+128	+1024
| VALEUR MINIMALE |	$1 \times 2^{-149} = 1.4 \times 10^{-45}$	$1 \times 2^{-1023-52} \approx 10^{-324}$
| VALEUR MAXIMALE |	$\mathbf{0b}1{,}11... \times 2^{126} \approx 1,992^{126} = 10^{38}$	10^{307}

FIGURE 4.2 – Caractéristiques des nombres codés IEEE 754

4.2.3 Codage de l'exposant

Le codage de l'exposant signé n'utilise pas la méthode du complément à 2. Il utilise un codage avec excédent, c'est-à-dire que nous ajoutons une valeur entière positive à l'exposant signé. De cette façon, le nombre obtenu pour l'exposant est toujours positif.
L'excédent sera choisi de sorte que le nombre de valeurs de l'exposant positives et négatives soient identiques.
Au format simple précision, l'excédent est de $\lfloor \frac{254}{2} \rfloor = 127$. Le plus petit entier non signé représentable au format 8 bits est 0, le plus grand 255. La figure FIG. 4.3 spécifie les valeurs de l'exposant codé et de l'exposant réel au format simple précision.

Exposant codé	**Expression de l'exposant réel**	**Exposant réel**
0	$0 = 0 - 127$	-127
1	$1 = 1 - 127$	-126
x	$= x - 127$	
254	$254 = 254 - 127$	+127
255	$255 = 255 - 127$	+128

FIGURE 4.3 – Excédent au format simple précision

Remarque 22 :

— Les valeurs -127 et +128 sont réservées pour les nombres spéciaux et dénormalisés.
— Un nombre est représenté de façon normalisée lorsque l'exposant est donc compris entre -126 et +127 ,codé entre 1 et 254 au format simple précision.

Exemple 37 :

— Codage mémoire du réel 1,0 au format simple précision :

$$\begin{aligned} 1,0 &= \mathbf{0b}1{,}0 \times 2^0 \\ &= \mathbf{0x}1{,}0 \times 2^0 \\ \text{Mantisse} &= 0 \text{ normalisée} \\ \text{Exposant} &= 127 = \mathbf{0b}0111\,1111 \\ \text{Mémoire} &= \mathbf{0b}0\ 01111111\ 00000000000000000000000 \\ &= \mathbf{0b}0011\,1111\,1000\,0000\,0000\,0000\,0000\,0000 \\ &= \mathbf{0x}3F80\,0000 \end{aligned} \tag{4.8}$$

Signe mantisse	Exposant	Mantisse
1 bit	8 bits	23 bits
0	011 1111 1	000 0000...

— Codage mémoire du réel -2,5 au format simple précision :

$$\begin{aligned} -2,5 &= \mathbf{0b}\text{- }10{,}1 \\ &= \mathbf{0b}\text{- }1{,}010\,0 \times 2^1 \\ &= \mathbf{0x}\text{- }1{,}4 \times 2^1 \\ \text{Mantisse} &= \mathbf{0b}0100\,0000 \text{ ... normalisée} \\ \text{Exposant} &= 127 + 1 = \mathbf{0b}1000\,0000 \\ \text{Mémoire} &= \mathbf{0b}1\ 10000000\ 01000000000000000000000 \\ &= \mathbf{0b}1100\,0000\,0010\,0000\,0000\,0000\,0000\,0000 \\ &= \mathbf{0x}C020\,0000 \end{aligned} \tag{4.9}$$

Signe mantisse	Exposant	Mantisse
1 bit	8 bits	23 bits
1	100 0000 0	010 0000...

Remarque 23 :

La mantisse est alignée sur le bit de gauche de la représentation. Si elle était alignée à droite, après dénormalisation la valeur représentée serait **0b**1,000...01 ce qui est différent de **0b**1,01.

Le circuit de la figure Fig 4.4 illustre le décodage et l'affichage sous forme hexadécimale du nombre -2,5.

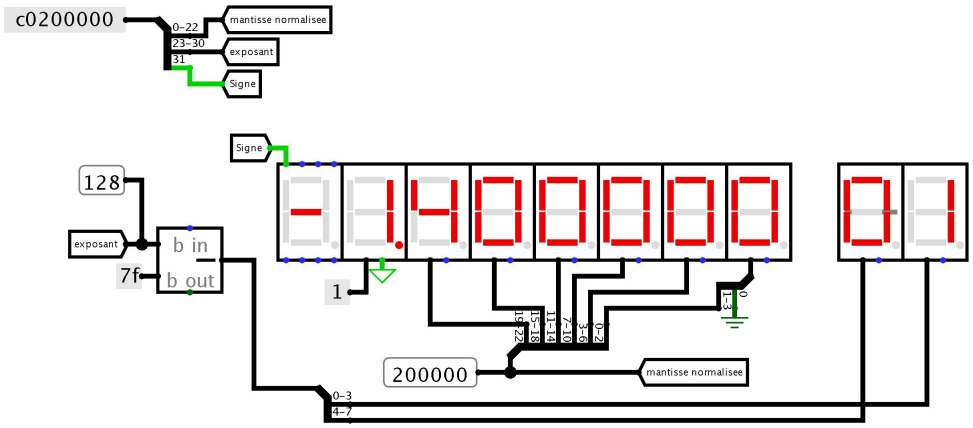


FIGURE 4.4 – Affichage du réel -2,5 en hexadécimal

4.2.4 Nombres spéciaux et dénormalisés au format simple précision

Les nombres normalisés sont tels que le bit à gauche de la virgule soit à 1.

$$|\min| = \mathbf{0b}1{,}000... \times 2^{-126} \tag{4.10}$$

$$|\max| = \mathbf{0b}1{,}11...1 \times 2^{127} \approx 2^{128} \tag{4.11}$$

Nombres très proches de zéro Lorsque le nombre à représenter devient trop petit pour être normalisé (c'est-à- dire inférieur à $1,0 \times 2^{-126}$, pas de 1 devant le nombre) la norme spécifie de le dénormaliser (plus de bit à 1 à gauche de la virgule mais un bit à 0). Le nombre est alors de la forme :

$$|min| = 0,000...11.. \times 2^{-126} \tag{4.12}$$

(4.13)

Le nombre le plus petit dénormalisé différent de zéro au format simple précision est :

$$0.000\,0000\,0000\,0000\,0000\,0001 \times 2^{-126} = 1 \times 2^{-126-23} \tag{4.14}$$

$$= 1 \times 2^{-149} \tag{4.15}$$

$$= 3.5 \times 10^{-46} \tag{4.16}$$

Remarque 24 :

Dans ce cas, l'exposant codé est -127, et non pas -126.

Zéro Zéro est codé avec mantisse et l'exposant codé à zéro

$$0.000\,0000\,0000\,0000\,0000\,0000 \times 2^{-126} = 0 \tag{4.17}$$

Remarque 25 :

Dans ce cas, l'exposant codé est -127, et non pas -126.

L'infini Lorsque le nombre à représenter devient trop grand pour être normalisé (c'est-à- dire supérieur à $1,1111..1112^{127} \approx 1,0 \times 2^{128}$) il est dénormalisé. Lorsque la mantisse est égale à 0 et que l'exposant est égal à 128 le nombre représenté est $\pm$ inf en fonction de la valeur du bit de signe.

$$Infini = \pm 1.000\,0000\,0000\,0000\,0000\,0000 \times 2^{128} \tag{4.18}$$

Nombres invalides Dans ce cas (exposant égal à 128) si la mantisse est différente 0, le nombre représenté n'est pas valide. Il est appelé Not A Number noté NaN(exemple racine carrée de -3).

$$NaN = 1.100\,0000\,0000\,0000\,0000\,0000 \times 2^{128} \tag{4.19}$$

4.2.5 Addition de deux nombres réels codés au format simple précision

Algorithme d'addition

$$\begin{aligned} N_1 &= m1 \times 2^{e1} \\ N_2 &= m2 \times 2^{e2} \\ N &= N_1 + N_2 \\ N &= m1 \times 2^{e1} + m2 \times 2^{e2} \\ N &= m1 \times 2^{e1} + m2 \times 2^{e2-e1+e1} \\ N &= (m1 + m2 \times 2^{e2-e1}) \times 2^{e1} \end{aligned} \tag{4.20}$$

Pour additionner deux nombres $N_1 > 0$ et $N_2 > 0$, il faut réaliser les opérations spécifiées par l'algorithme Algorithme. 45) .

Algorithme 45 Addition de deux nombres réels positifs format simple précision

```
1: Variables
2:    N1, N2, N : réel 32 bits
3:    m, m1, m2 : entier 24 bits
4:    e, e1, e2 : entier8 bits
5: Début
6: Lire (N1,N2)
7: m1 ← Mantisse de N1 avec bit normalisation
8: m2 ← Mantisse de N2 avec bit normalisation
9: e1 ← Exposant de N1
10: e2 ← Exposant de N2
11: Si e1 < e2 Alors
12:    m1 ← m1 >>> (e2 − e1)
13:    e ← e2
14: Sinon
15:    m2 ← m2 >>> (e1 − e2)
16:    e ← e1
17: FinSi
18: m ← m1 + m2 //Addition 2 × 24 bits
19: Si c_out = 1 Alors
20:    e ← e + 1
21:    m ← (m >>> 1) | 0x800000 //Décalage avec c_out
22: FinSi
23: Coder m et e dans N
24: Fin
```

Exemple 38 :

$$
\begin{aligned}
2,5 &= \mathbf{0b}1{,}01 \times 2^1 \\
1,0 &= 1,0 \times 2^0 = \mathbf{0b}0{,}10 \times 2^1 \\
2,5+1,0 &= (\mathbf{0b}1{,}01 + \mathbf{0b}0{,}10) \times 2^1 \\
&= \mathbf{0b}1{,}112^1 \\
\text{Mantisse} &= \mathbf{0b}11 \\
\text{Exposant} &= 127+1 = \mathbf{0b}1000\,0000 \\
\text{Mémoire} &= \mathbf{0b}0100\,0000\,0110\,0000\,0000\,0000\,0000\,0000 \\
&= \mathbf{0x}4060\,0000
\end{aligned}
$$

$$
\begin{aligned}
1,0 &= 1,0 \times 2^0 \\
1,0+1,0 = (1,00+1,00) \times 2^0 &= 10,0 \times 2^0 \\
\text{Mantisse} &= 0 \\
\text{Exposant} &= 127+1 = 128 = \mathbf{0b}1000\ 0000 \\
\text{Mémoire} &= \mathbf{0b}0100\,0000\,0000\,0000\,0000\,0000\,0000\,0000 \\
&= \mathbf{0x}4000\,0000
\end{aligned}
$$

Circuit d'addition

Le circuit qui réalise l'addition de deux nombres réels doit pouvoir effectuer l'opération d'alternative. Ceci pour tester la valeur de $e1$ par rapport à celle de $e2$, et pour tester la retenue. Pour cela nous avons besoin d'un circuit de sélection (nommé aussi multiplexeur) manipulant des données 32 bits. Pour réaliser ce décodeur nous avons besoin d'un décodeur.

Décodeur

Ce circuit permet d'activer une sortie parmi 2^n en fonction de la valeur présente sur les n bits d'entrées **an-1** ,...,**a2** , **a1** ,**a0** . La figure Fig 4.5 donne la table de vérité d'un décodeur 1 parmi 4.

a1	**a0**	**s3**	**s2**	**s1**	**s0**
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Figure 4.5 – Table de vérité d'un décodeur 2 bits

$$\mathbf{s3} = \mathbf{a1}.\mathbf{a0} \tag{4.21}$$

$$\mathbf{s2} = \mathbf{a1}.\overline{\mathbf{a0}} \tag{4.22}$$

$$\mathbf{s1} = \overline{\mathbf{a1}}.\mathbf{a0} \tag{4.23}$$

$$\mathbf{s0} = \overline{\mathbf{a1}}.\overline{\mathbf{a0}} \tag{4.24}$$

Le circuit représenté sur Fig 4.6 illustre le contenu interne du décodeur.

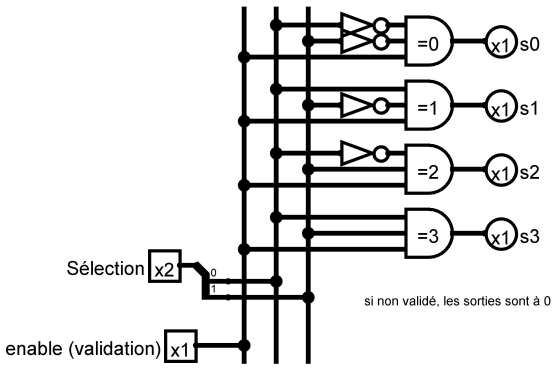


FIGURE 4.6 – Schéma interne du décodeur 2 bits

Le circuit de la FIG 4.7 est la représentation externe de ce décodeur en vue de son utilisation. Si l'entrée enable est à 0, les sorties sont à 0 dans ce cas.

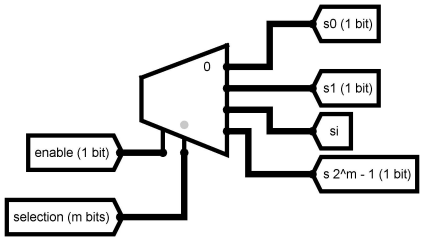


FIGURE 4.7 – Schéma externe du décodeur 2 bits

Remarque 26 :

- Si enable=0 sortie X ou à 0 en fonction de type de décodeur.
- Si enable=1 une des sorties est à 1

Multiplexeur

Un multiplexeur un bits permet d'envoyer sur la sortie **c** l'état d'une entrée **a** ou de l'autre **b** en fonction d'un signal de sélection **s**. La figure FIG 4.8 donne la table de vérité de ce multiplexeur.

s	**a**	**b**	**c**
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

FIGURE 4.8 – Table de vérité du multiplexeur 1 parmi 2

$$\mathbf{c} = \mathbf{a}.(\mathbf{b} + \overline{\mathbf{b}}).\overline{\mathbf{s}} + (\mathbf{a} + \overline{\mathbf{a}}).\mathbf{b}.\mathbf{s} \tag{4.25}$$

$$\mathbf{c} = \mathbf{a}.\overline{\mathbf{s}} + \mathbf{b}.\mathbf{s} \tag{4.26}$$

Le schéma de la figure Fig 4.9 illustre le contenu interne du multiplexeur sélectionnant une entrée parmi 2. Chaque variable est codée sur 1 bit.

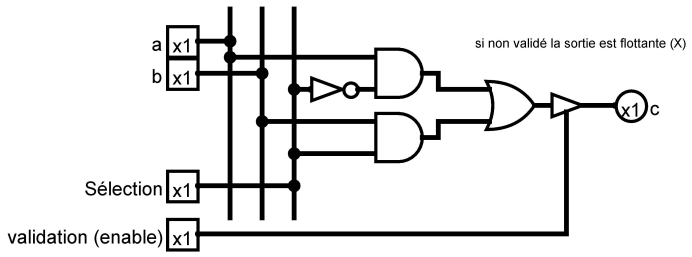


Figure 4.9 – Schéma interne d'un multiplexeur 1 bit

Remarque 27 :

— Si enable=0 la sortie est à l'état X

Le schéma de la figure Fig 4.9 l'utilisation de ce multiplexeur.

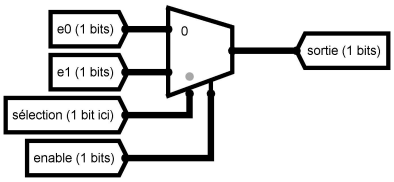


Figure 4.10 – Schéma externe d'un multiplexeur 1 bit

Les données que nous manipulons sont représentées sur 32 bits. Pour réaliser ce circuit, nous utiliserons un décodeur à 4 sorties. La figure Fig 4.11 donne le schéma interne de ce multiplexeur.

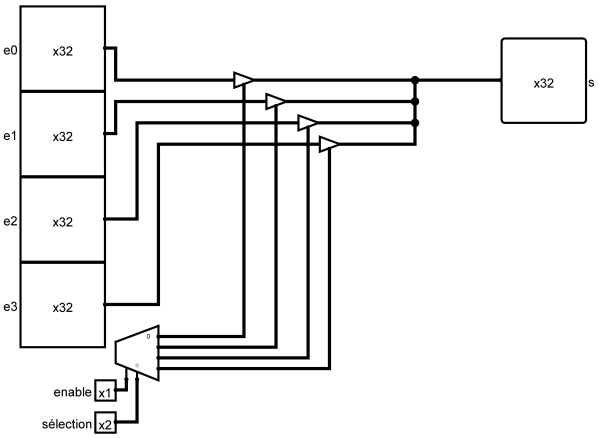


Figure 4.11 – Multiplexeur 1 parmi 4, 32 bits

Le décodeur pilote le contrôle des quatre portes OUI. Seule une des sorties du décodeur est à 1, et les bits de cette entrée sont placés sur la sortie.

Comparateur nombres non signé - 1 bit

1. Table de vérité

a_1	a_0	$a_1 < a_0$	$a_1 > a_0$	$a_1 = a_0$
0	0	0	0	1
0	1	1	0	0
1	0	0	1	0
1	1	0	0	1

FIGURE 4.12 – Table de vérité du comparateur 1 bit non signé

2. Équation

$$\begin{aligned} a_1 < a_0 &= \overline{a_1}.a_0 \\ a_1 > a_0 &= \overline{a_0}.a_1 \\ a_1 = a_0 &= \overline{a_0 \oplus a_1} \end{aligned}$$

3. Circuit

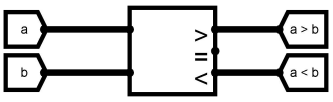


FIGURE 4.13 – Comparateur 1 bit non signé - vue externe

Addition de deux nombres réels normalisé

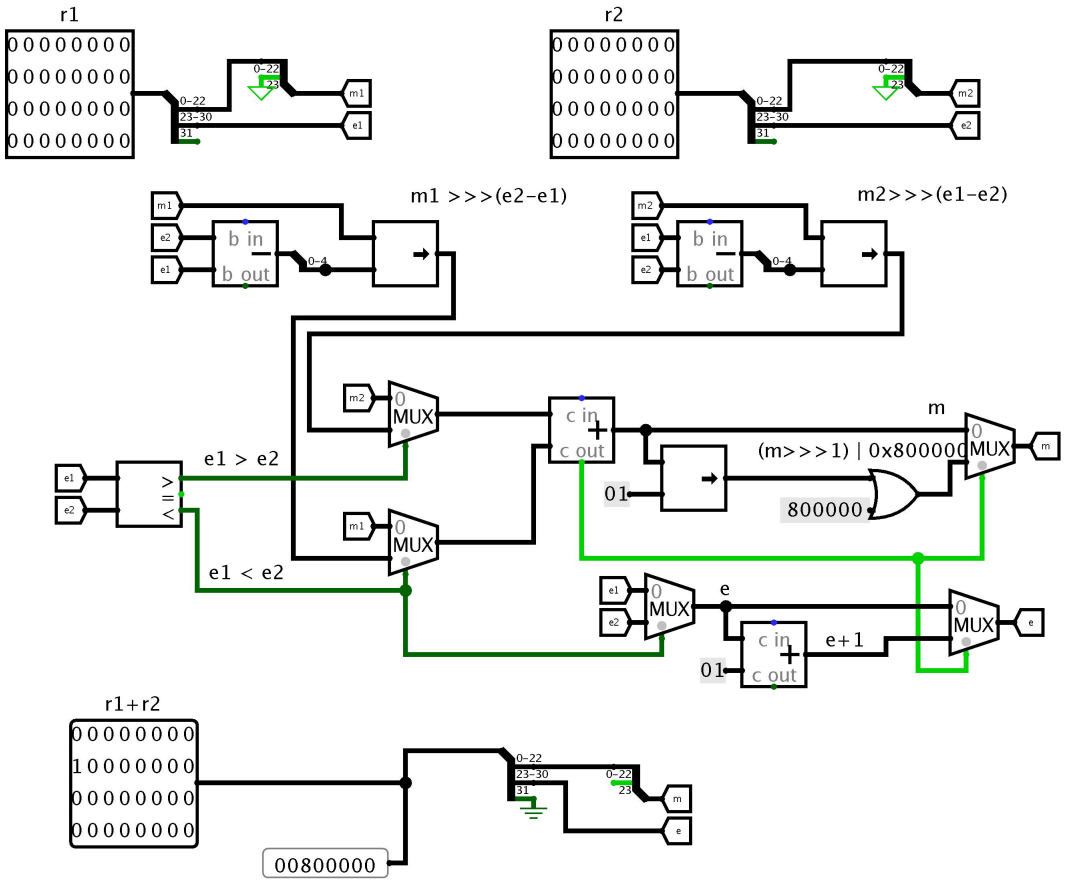


FIGURE 4.14 – Addition de deux nombres réels positifs normalisés

— A partir des nombres de départ nous devons extraire les exposant et les mantisses avec leurs bit de normalisation.
— Nous calculons en parallèle les résultats ($m1 >>> (e_2 - e_1)$ et de $m2 >>> (e_1 - e_2)$) de l'alternative ($e_1 < e_2$). Nous aurions aussi pu, inverser les deux mantisses et deux exposants et ne faire que le calcul de $m1 >>> (e_2 - e1)$
— En fonction e_1 et de e_2 nous plaçons les bonnes valeurs sur les entrées de l'additionneur 24 bits non signé
— Si une retenue non signée est générée, nous devons incrémenter la puissance et décaler le résultat de l'addition avec cette retenue non signée ($(m >>> 1) \mid \mathbf{0x}8000\,0000$).
— Nous devons ensuite former le résultat avec le nouvel exposant et la nouvelle mantisse normalisé (suppression du bit de gauche)
— Beaucoup plus complexe qu'une addition sur des nombres entiers
— Beaucoup plus lent car nous devons effectuer une suite d'opération.

4.3 Exercices

Exercice 5 : Théorique : Codage des nombres réels normalisés et algorithmes

Question 1 : Codage 125,25 et 0,375 Donnez au format simple précision et double précision les représentations mémoires hexadécimales des nombres 125,25 et 0,375 et -0.375.

Question 2 : Addition Donnez la méthode de calcul de 125,25+0,375 et de 125,25+125,25 au format simple précision.

Question 3 : y_{max} Si $x = 1,0$, $y = 2^{100}$ et $z = -y$ sont des réels codés au format simple précision, montrez que $(x + y) + z \neq x + (y + z)$. Donnez la valeur maximale de y_{max} (au format simple précision) pour que $(x + y_{max}) + z = x + (y_{max} + z)$.
Indication 1 : Pour cela effectuez les additions avec la priorité des parenthèses, au format simple précision, avec la mantisse codée sur 23 bits.
Indication 2 : Vous déterminerez y_{max} pour que la relation suivante soit vérifiée. y_{max} représente le nombre maximum que nous pouvons additionner à la valeur 1.0 sans perdre le bit de droite.

$$1 + y_{max} = y_{maxi} \tag{4.27}$$
$$y_{max} = y_{maxi} - 1 \tag{4.28}$$

Indication 3 : L'addition peut provoquer un résultat sur 24 bits (voir exercice précédent), dans ce cas la mantisse sera re-normalisée.

Question 4 : Plus grand normalisé Donnez la représentation mémoire au format simple précision du nombre normalisé positif le plus grand.

Question 5 : Plus petit normalisé Donnez la représentation mémoire hexadécimale au format simple précision du nombre normalisé positif le plus petit.

Question 6 : 2^{-130} Donnez la représentation mémoire hexadécimale au format simple précision du nombre réel dénormalisé 2^{-130}.

Question 7 : Pus petit dénormalisé Donnez la représentation mémoire hexadécimale au format simple précision du nombre dénormalisé positif le plus petit.

Question 8 : Algorithmes accès réels Complétez les algorithmes des six fonctions permettant d'accéder au différentes parties d'un nombre réel codé normalisé au format simple précision.

Fonction *getSigne* (*nombre* : réel 32 bits) : entier
Fonction *getExposant* (*nombre* : réel 32 bits) : entier
Fonction *getMantisse* (*nombre* : réel 32 bits) : entier
Fonction *setSigne* (*nombre* : réel 32 bits,*signe* : entier) : réel 32 bits
Fonction *setExposant* (*nombre* : réel 32 bits,*exposant* : entier) : réel 32 bits
Fonction *setMantisse* (*nombre* : réel 32 bits,*mantisse* : entier) : réel 32 bits

Algorithme 46 Obtenir ? ? ? d'un réel simple précision - énoncé

```
1: Fonction get ? ? ? (nombre : réel 32 bits) : entier
2: Début
3: Retourner ...
4: Fin fonction
```

Algorithme 47 Modifier ? ? ? d'un réel simple précision - énoncé

```
1: Fonction set ? ? ? (nombre : réel 32 bits,??? : entier) : réel 32 bits
2: Variables
3:    bits : entier
4: Début
5: bits ← FloatToIntBits(nombre)
6: bits ← ...
7: Retourner IntBitsToFloat(bits)
8: Fin fonction
```

Indication 1 :

- Le signe retourné sera égal à 1 si le nombre est négatif, 0 sinon
- L'exposant retourné sera l'exposant d'un réel normalisé ($-126 \leq exposant \leq +127$),
- La mantisse retournée sera codée sur 24 bits (le bit de poids b_{23} est à 1 car la mantisse est normalisée et les autres bits seront positionné à 0).
- Vous utiliserez les opérateurs bit à bit.
- Vous utiliserez les fonctions *Float.FloatToIntBits* et *Float.IntBitsToFloat* pour convertir un réel en sa représentation mémoire dans un nombre entier 32 bits.

Fonction *FloatToIntBits* (*nombre* : réel 32 bits) : entier
Fonction *IntBitsToFloat* (*nombre* : entier) : réel 32 bits

Question 9 : *toHexString* Complétez l'algorithme d'une fonction nommée *toHexString* permettant de convertir un nombre réel normalisé codé au format simple précision en une chaîne de caractères hexadécimaux au format suivant :

- Le signe '-' ou '+"
- les caractères "0x1. (nombre normalisé)
- les caractères hexadécimaux de mantisse
- le caractère 'p'
- la valeur décimale signée de l'exposant

Fonction *toHexString* (*nombre* : réel 32 bits) : chaîne de caractères
Exemple 39 :

$$toHexString(125.25) \quad = \quad \text{"} + 0x1.F5p6\text{"}$$

Indication 1 :

- La fonction placera le signe '-' ou '+' en utilisant la fonction *getSigne* puis la chaîne "0x1"
- En utilisant la fonction *getMantisse*,vous convertirez les 23 bits de la mantisse en hexadécimal à l'aide de la fonction *forDigit*. Pour convertir vous n'utilisez pas la fonction *ItoHexString*, car nous devons afficher les digits de gauche de la mantisse s'ils sont à zéro. Si les 30 bits de la mantisse sont à 0, vous afficherez un seul '0'. Vous affichez les digits en partant de celui de gauche (bit 22) et non pas de droite. En effet, avec la valeur **0b**111 1010 1000 0000 0000 0000 la fonction *toHexString*(**0x**7 A800) retourne la chaîne de caractère "7A8000". Dans ce cas, nous devrions afficher les digits formés par les ensemble de quatre bits, en partant de gauche et non pas la chaîne correcte "F5".
- Vous ajouterez "p", puis l'exposant en décimal signé. Pour cela, vous utiliserez la fonction *toString*.

Algorithme 48 Enoncé conversion nombre réel simple précision en une chaîne de caractères

```
1: Fonction toHexString (nombre : réel 32 bits) : chaîne de caractères
2: Variables
3:    chaîne : chaîne de caractères
4:    mantisse, exposant : entier
5: Début
6: Si getSigne(nombre) = 1 Alors
7:    chaîne ← ? ? ?
8: Sinon
9:    chaîne ← ? ? ?
10: FinSi
11: mantisse ← getMantisse(nombre) & 0x7F FFFF
12: Répéter
13:    chaîne ← chaîne + forDigit(???)
14:    mantisse ←???
15: Jusqu'à (mantisse = 0)
16: exposant ← getExposant(nombre)
17: chaîne ← chaîne +' p'+???
18: Retourner chaîne
19: Fin fonction
```

Question 10 : *parseFloat* Complétez l'algorithme d'une fonction nommée *parseFloat* permettant de convertir une chaîne de caractères hexadécimaux (supposés valide) en un nombre réel normalisé codé au format simple précision.
Format de la chaîne de caractères :

- Le signe '-' ou '+"
- les caractères "0x1."
- les caractères hexadécimaux de mantisse
- le caractère 'p'
- la valeur décimale signée de l'exposant

Fonction *parseFloat* (*chaîne* : chaîne de caractères) : réel 32 bits
Indication 1 :

- En fonction du caractère à la position 0 de la chaîne, vous placerez le signe dans le résultat en utilisant la fonction *setSigne*.
- Pour convertir la mantisse en hexadécimal, vous n'utilisez pas la fonction *Integer.parseInt* , car nous devons convertir les digits de gauche de la mantisse s'ils sont à zéro. Vous convertirez les caractères de la chaîne de la position 5 à la lettre 'p' dans la mantisse à l'aide de la fonction *digit*, zéro de gauche compris. En effet, parsInt("01",16) retourne le nombre hexadécimal **0x**1 et non pas le nombre hexadécimal **0x**0100 0000. Vous placerez la mantisse dans le résultat en utilisant la fonction. *setMantisse*.
- Pour l'exposant vous copierez le reste de la chaîne (après le caractère 'p') qui représente l'exposant dans autre chaîne en utilisant la fonction *String.substring(int indice)*. Cette fonction copie dans une chaîne de caractères les caractères à partir de l'indice spécifié. Vous convertirez cette chaîne en décimale à l'aide de la fonction *parseInt* avant de le placer dans le résultat en utilisant la fonction *setExposant*.

Algorithme 49 Enoncé conversion chaîne de caractères hexadécimaux en nombre réel simple précision

```
1: Fonction parseFloat (   chaîne : chaîne de caractères) : réel 32 bits
2: Variables
3:    ls : chaîne de caractères
4:    position, décalage, mantisse, exposant : entier
5:    nombre : réel 32 bits
6: Début
7: nombre ← 0,décalage ← 19,position ← 5, mantisse ← 0
8: Si  Caractère(chaîne,0) =' −') Alors
9:    nombre ←???
10: FinSi
11: Tant Que  Caractère(chaîne, position) ≠' p' Faire
12:    mantisse ←???
13:    position ← position + 1
14:    décalage ← décalage − 4
15: Fin Tant Que
16: nombre ← setMantisse(nombre, mantisse)
17: position ← position+???
18: ls ← String.subString(chaine,position)
19: exposant ← parseInt(???, 10)
20: nombre ← setExposant(nombre, exposant)
21: Retourner  nombre
22: Fin fonction
```

Correction exercice 5 :

Question 1 : Codage 125,25 et 0,375

$$
\begin{aligned}
125,25 &= \mathbf{0b}111\,1101{,}01 = \mathbf{0x}7D{,}4\\
&= \mathbf{0b}1{,}1111\,0101 \times 2^{6} = \mathbf{0x}1{,}F5 \times 2^{6}\\
\text{Exposant simple précision } 125{,}25 &= 127+6 = 133 = \mathbf{0b}1000\,0101\\
\text{Exposant double précision } 125{,}25 &= 1023+6 = 1029 = \mathbf{0b}100\,0000\,0101\\
\text{Mantisse normalisée } 125{,}25 &= \mathbf{0b}1111\,0101\ldots\\
\text{Mémoire simple précision } 125{,}25 &= \mathbf{0b}0100\,0010\,1111\,1010\,1000\,0000\,0000\,0000\\
&= \mathbf{0x}42FA\,8000\\
\text{Mémoire double précision } 125{,}25 &= \mathbf{0b}0100\,0000\,0101\,1111\,0101\,0000\,0000\ldots\\
&= \mathbf{0x}405F\,5000\,0000\,0000\\
0,375 &= \mathbf{0b}0{,}011 = \mathbf{0x}0{,}6\\
&= \mathbf{0b}1{,}1 \times 2^{-2} = \mathbf{0x}1{,}8 \times 2^{-2}\\
\text{Exposant simple précision } 0{,}375 &= 127-2 = 125 = \mathbf{0b}0111\,1101\\
\text{Exposant double précision } 0.375 &= 1023-2 = 1021 = \mathbf{0b}011\,1111\,1101\\
\text{Mantisse normalisée } 0{,}375 &= \mathbf{0b}1000\ldots\\
\text{Mémoire simple précision } 0{,}375 &= \mathbf{0b}0011\,1110\,1100\,0000\,0000\,0000\,0000\,0000\\
&= \mathbf{0x}3EC0\,0000\\
\text{Mémoire simple précision } -0{,}375 &= \mathbf{0b}1011\,1110\,1100\,0000\,0000\,0000\,0000\,0000\\
&= \mathbf{0x}BEC0\,0000\\
\text{Mémoire double précision } 0{,}375 &= \mathbf{0b}0011\,1111\,1101\,1000\,0000\,0000\,0000\ldots\\
&= \mathbf{0x}3FD8\,0000\,0000\,0000\\
\text{Mémoire double précision } -0{,}375 &= \mathbf{0b}1011\,1111\,1101\,1000\,0000\,0000\,0000\ldots\\
&= \mathbf{0x}BFD8\,0000\,0000\,0000
\end{aligned}
$$

Question 2 : Addition

$$
\begin{aligned}
125,25+0,375 &= \mathbf{0b}1{,}11110101 \times 2^{6} + \mathbf{0b}0{,}000000011 \times 2^{-2+(6-(-2))} \text{ Alignement mantisses}\\
&= \mathbf{0b}1{,}111101101 \times 2^{6} \text{ Addition des mantisses}\\
\text{Mémoire } 125{,}625 &= \mathbf{0b}0100\,0010\,1111\,1011\,0100\,0000\,0000\,0000\\
&= \mathbf{0x}42FB\,4000
\end{aligned}
$$

$$
\begin{aligned}
125,25+125,25 &= \mathbf{0b}1{,}11110101 \times 2^{6} + \mathbf{0b}1{,}11110101 \times 2^{6}\\
&= \mathbf{0b}11{,}1110101 \times 2^{6} \text{ Addition des mantisses}\\
&= \mathbf{0b}1{,}11110101 \times 2^{7} \text{ Normalisation}\\
\text{Mémoire } 250{,}5 &= \mathbf{0b}0100\,0011\,0111\,1010\,1000\,0000\,0000\,0000\\
&= \mathbf{0x}437A\,8000
\end{aligned}
$$

Question 3 : y_{max}

$$\begin{aligned}(x+y) &= 1\times 2^0 + 1\times 2^{100}\\ &= \mathbf{0b}0.000...01\times 2^{100} + 1\times 2^{100}\\ &= 1\times 2^{100} \text{ Mantisse codée sur 23 bits, dernier 1 non conservé}\\ (x+y)+z &= 1\times 2^{100} + (-1)\times 2^{100}\\ &= 0\end{aligned}$$

L'écart entre les deux nombres est trop important, il n'y a pas assez de place dans la mantisse pour mettre le dernier bit à 1 de la valeur réelle 1.0.

$$\begin{aligned}(y+z) &= 1\times 2^{100} + (-1)\times 2^{100}\\ &= 0\\ x+(y+z) &= 1\times 2^1 + 0\\ &= 1\end{aligned}$$

Dans ce cas, les deux nombres ne sont pas trop différents, le résultat est correct.

$$\begin{aligned}1\times 2^0 + y_{max} &= y_{maxi}\\ \mathbf{0b}0{,}00000000000000000000001\times 2^{23} + y_{max} &= \mathbf{0b}1{,}11111111111111111111111\times 2^{23}\\ y_{max} &= \mathbf{0b}1{,}11111111111111111111111\times 2^{23}\\ &- \mathbf{0b}0{,}00000000000000000000001\times 2^{23}\\ &= \mathbf{0b}1{,}11111111111111111111110\times 2^{23}\\ &= 2^{24}-2\end{aligned}$$

Nous avons vu que l'addition de deux mantisses génère une retenue, qui peut être utilisée en renormalisant le nombre réel résultat. Dans ce cas nous obtenons la valeur de y_{max} suivante :

$$\begin{aligned}1\times 2^0 + y_{max} &= y_{maxi} + 1\\ \mathbf{0b}0{,}00000000000000000000001\times 2^{23} + y_{max} &= 1\times 2^{24}\\ y_{max} &= 1\times 2^{24} - \mathbf{0b}0{,}00000000000000000000001\times 2^{23}\\ &= \mathbf{0b}1{,}11111111111111111111111\times 2^{23}\\ &= 2^{24}-1\end{aligned}$$

Question 4 : Plus grand normalisé

$$\begin{aligned}\text{Nombre max} &= \mathbf{0b}1.11.....1\times 2^{127}\\ \text{Mémoire} &= \mathbf{0b}0\ 111\ 1111\ 0111\ 1111\ 1111\ 1111\ 1111\ 1111\\ &= \mathbf{0x}7F7F\ FFFF\end{aligned}$$

Question 5 : Plus petit normalisé

$$\begin{aligned}\text{min} &= 1.0\times 2^{-126}\\ \text{Mémoire} &= \mathbf{0b}0000\ 0000\ 1000\ 0000\ 0000\ 0000\ 0000\ 0000\\ &= \mathbf{0x}80\ 0000\end{aligned}$$

Question 6 : 2^{-130} Le nombre est dénormalisé puisque la puissance est inférieure à -126.

$$\begin{aligned} 2^{-130} &= 0.0001 \times 2^{-126} \\ \text{Mémoire} &= \mathbf{0b}0000\,0000\,0000\,1000\,0000\,0000\,0000\,0000 \\ &= \mathbf{0x}8\,0000 \end{aligned}$$

Dans ce cas l'exposant est codé à 0 et le bit à gauche de la virgule vaut 0. La mantisse codée vaut donc 0001_2.

Question 7 : Pus petit dénormalisé

$$\begin{aligned} N_{min} &= \mathbf{0b}0.00000000000000000000001 \times 2^{-126} \\ \text{Mémoire} &= \mathbf{0b}0000\,0000\,0000\,0000\,0000\,0000\,0000\,0001 \\ &= 1 \end{aligned}$$

Question 8 : Algorithmes accès réels

Algorithme 50 Obtenir le signe d'un réel simple précision

```
1: Fonction getSigne (nombre : réel 32 bits) : entier
2: Variables
3: Début
4: Retourner FloatToIntBits(nombre) >>> 31
5: Fin fonction
```

Algorithme 51 Obtenir l'exposant d'un réel simple précision

```
1: Fonction getExposant (nombre : réel 32 bits) : entier
2: Début
3: Retourner ((FloatToIntBits(nombre) << 1) >>> 24) − 127
4: Fin fonction
```

Algorithme 52 Obtenir la mantisse d'un réel simple précision

```
1: Fonction getMantisse (nombre : réel 32 bits) : entier
2: Début
3: Retourner (FloatToIntBits(nombre) & 0x7F FFFF) | 0x80 0000
4: Fin fonction
```

Algorithme 53 Modifier le signe d'un réel simple précision

```
1: Fonction setSigne (nombre : réel 32 bits,signe : entier) : réel 32 bits
2: Variables
3:    bits : entier
4: Début
5: bits ← FloatToIntBits(nombre)
6: bits ← (bits & 0x7FFF FFFF) | (signe << 31)
7: Retourner IntBitsToFloat(bits)
8: Fin fonction
```

Algorithme 54 Modifier l'exposant d'un réel simple précision

```
1: Fonction setExposant (nombre : réel 32 bits,exposant : entier) : réel 32 bits
2: Variables
3:    bits : entier
4: Début
5: bits ← FloatToIntBits(nombre)
6: bits ← (bits & 0x807F FFFF) | ((exposant + 127) & 0xFF) << 23)
7: Retourner IntBitsToFloat(bits)
8: Fin fonction
```

Algorithme 55 Modifier la mantisse d'un réel simple précision

```
1: Fonction setMantisse (nombre : réel 32 bits,mantisse : entier) : réel 32 bits
2: Variables
3:    bits : entier
4: Début
5: bits ← FloatToIntBits(nombre)
6: bits ← (bits & 0xFF80 0000) | (mantisse & 0x7F FFFF)
7: Retourner IntBitsToFloat(bits)
8: Fin fonction
```

Question 9 : *toHexString*

Algorithme 56 Conversion nombre réel simple précision en une chaîne de caractères

```
1: Fonction toHexString (nombre : réel 32 bits) : chaîne de caractères
2: Variables
3:    chaîne : chaîne de caractères
4:    mantisse, exposant : entier
5: Début
6: Si getSigne(nombre = 1) Alors
7:    chaîne ← "-0x1."
8: Sinon
9:    chaîne ← "+0x1."
10: FinSi
11: mantisse ← getMantisse(nombre) & 0x7F FFFF
12: Répéter
13:    chaîne ← chaîne + forDigit(mantisse >>> 19)
14:    mantisse ← (mantisse << 4) & 0x7F FFFF
15: Jusqu'à (mantisse = 0)
16: exposant ← getExposant(nombre)
17: chaîne ← chaîne +' p' + toString(exposant, 10)
18: Retourner chaîne
19: Fin fonction
```

Question 10 : *parseFloat*

Algorithme 57 Conversion chaîne de caractères hexadécimaux en nombre réel simple précision

```
1: Fonction parseFloat (  chaîne : chaîne de caractères) : réel 32 bits
2: Variables
3:    ls : chaîne de caractères
4:    position, décalage, mantisse, exposant : entier
5:    nombre : réel 32 bits
6: Début
7: nombre ← 0,décalage ← 19,position ← 5, mantisse ← 0
8: Si  Caractère(chaîne,0) =' −') Alors
9:    nombre ← setSigne(nombre, 1)
10: FinSi
11: Tant Que  Caractère(chaîne, position) ≠' p' Faire
12:    mantisse ← mantisse | (digit(Caractère(chaîne, position)) << décalage)
13:    position ← position + 1
14:    décalage ← décalage − 4
15: Fin Tant Que
16: nombre ← setMantisse(nombre, mantisse)
17: position ← position + 1
18: ls ← String.subString(chaine,position)
19: exposant ← parseInt(ls, 10)
20: nombre ← setExposant(nombre, exposant)
21: Retourner  nombre
22: Fin fonction
```

Exercice 6 : Programmation : Codage et algorithme sur les nombres réels

Remarque 28 :

- Une constante réelle de type float (format 32 bits) se termine par la lettre f (exemple $125.25f$)
- La fonction *memoire* donnée ici permet de convertir en une chaîne de caractères hexadécimaux, le contenu mémoire associé à un nombre réel codé au format simple précision.

```
static String memoire(float réel){
    int bits;
    bits = Float.floatToIntBits(réel);
    return (Integer.toHexString(bits));
}
```

La documentation de l'API JAVA Java sur les nombres réels est disponible sur `http://docs.oracle.com/javase/7/docs/api/java/lang/Float.html`

Question 1 : Vérification des valeurs théoriques Analyser le code suivant (*nombres_reels_api.java*) et retrouver les différentes valeurs hexadécimales obtenues dans les exercices théoriques.

Listing 4.1 – API java des nombres réels

```
1  public class nombres_reels_api{
2    static String memoire(float reel){
3      int bits;
4      bits = Float.floatToIntBits(reel);
5      return (Integer.toHexString(bits));
6    }
7    public static void main( String[] args ) {
8      System.out.println("Constante de la classe Float en memoire ");
9      System.out.println("MIN_EXPOSANT="+Float.MIN_EXPONENT);
10     System.out.println("NaN           ="+memoire(Float.NaN));
11     System.out.println("- infini      ="+memoire(Float.NEGATIVE_INFINITY));
12     System.out.println("+infini       ="+memoire(Float.POSITIVE_INFINITY));
13     System.out.println("0             ="+memoire(0f));
14     System.out.println("MIN_NORMAL="+memoire(Float.MIN_NORMAL));
15     System.out.println("MAX_VALUE   ="+memoire(Float.MAX_VALUE));
16     System.out.println("MIN_VALUE    ="+memoire(Float.MIN_VALUE));
17     System.out.println("125.25        ="+memoire(125.25f));
18     System.out.println("-0.375        ="+memoire(-0.375f));
19     System.out.println("2p-130        ="+memoire(0x1p-130f));
20     System.out.println("Fonction Float.toHexString");
21     System.out.println("125.25        ="+Float.toHexString(125.25f));
22     System.out.println("-0.375        ="+Float.toHexString(-0.375f));
23     System.out.println("Fonction Float.parseFloat");
24     System.out.println("0x1.f5p6      ="+Float.parseFloat("0x1.f5p6"));
25     System.out.println("-0x1.8p-2     ="+Float.parseFloat("-0x1.8p-2"));
26   }
27 }
```

Question 2 : Programmation des fonctions Écrivez les six fonctions nommées *get...* et *set...*, la fonction *toHexString* et la fonction *parseFloat*

Question 3 : maximum Écrivez la fonction nommée **maximum** permettant de déterminer la valeur y_{max} de l'exercice théorique. Vous procéderez itérativement en partant de la valeur

$y_{max} = 1$ jusqu'à obtenir la valeur maximale. La condition de sortie de l'itération est le non respect de l'associativité.

Question 4 : Circuits de manipulation des nombres réels normalisé Complétez les circuits permettant :

— d'afficher en hexadécimal la valeur d'un nombre réel normaliser

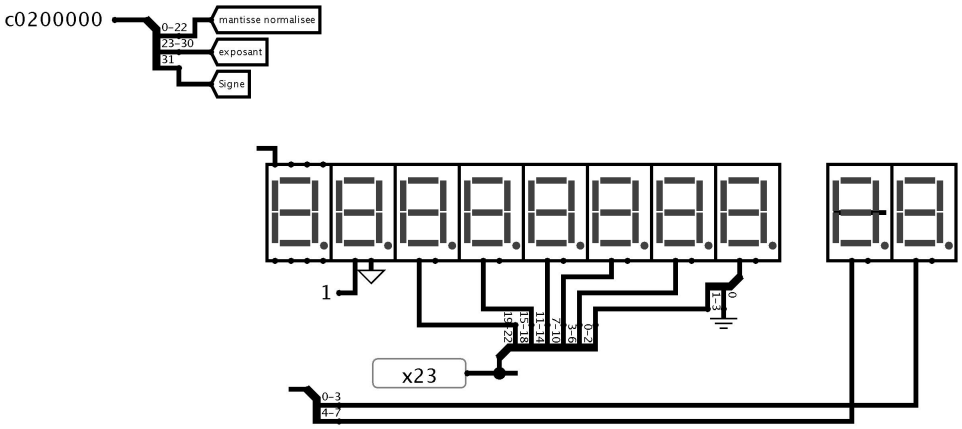


FIGURE 4.15 – Affichage d'un nombre réel normalisé en hexadécimal, énonce

— d'additionner deux nombres réels positifs normalisés

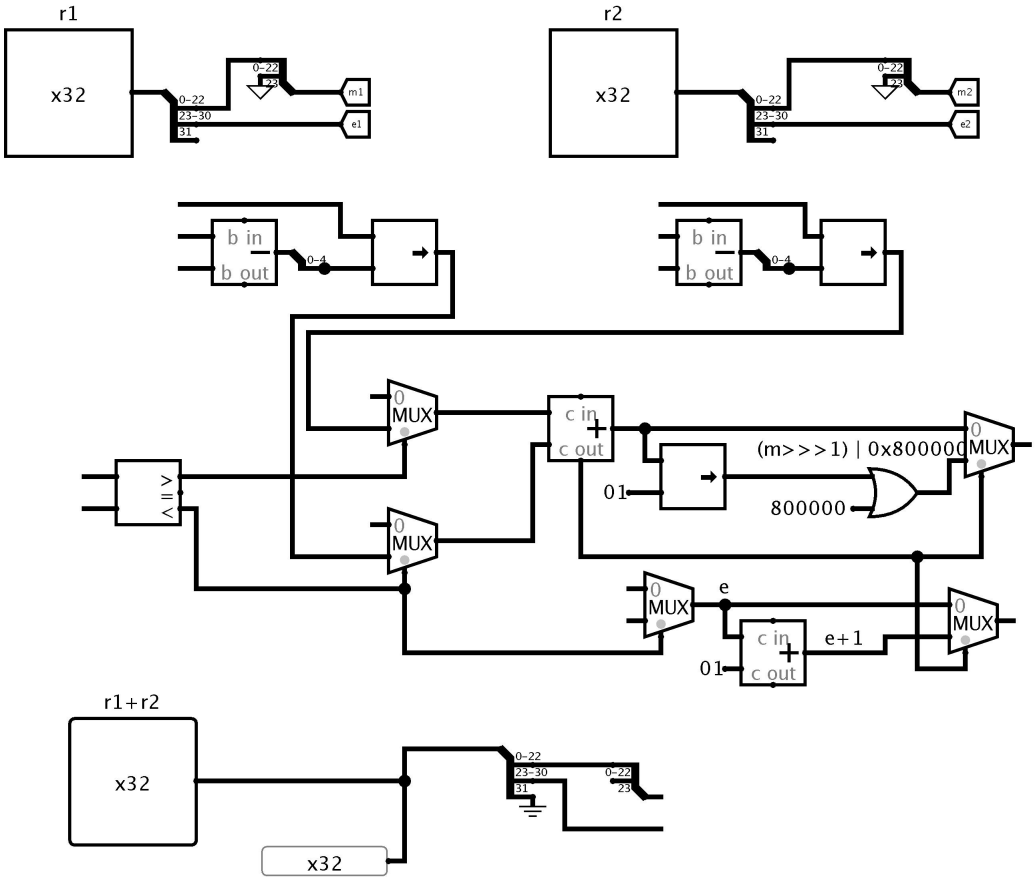


FIGURE 4.16 – Addition de deux réels normalisés positifs, énoncé

Correction exercice 6 :

Question 1 : Manipulation nombre réels

Listing 4.2 – Algorithmes sur les nombres réels

```
1  public class nombres_reels_algo{
2    static int getSigne(float reel){
3      return Float.floatToIntBits(reel) >>>31;
4    }
5    static int getExposant(float reel){
6      return (((Float.floatToIntBits(reel) << 1)>>>24)-127);
7    }
8    static int getMantisse(float reel){
9      return 0x800000 |(Float.floatToIntBits(reel) & 0x7FFFFF);
10   }
11   static float setSigne(float reel,int signe){
12     int bits;
13     bits = Float.floatToIntBits(reel);
14     bits=(bits & 0x7FFFFFFF)| (signe << 31);
15     reel = Float.intBitsToFloat(bits);
16     return reel;
17   }
18   static float setExposant(float reel,int exposant){
19     int bits;
20     bits = Float.floatToIntBits(reel);
21     bits=(bits & 0x807FFFFF) | (((exposant+127)&0xFF)<<23);
22     reel = Float.intBitsToFloat(bits);
23     return reel;
24   }
25   static float setMantisse(float reel,int mantisse){
26     int bits;
27     bits = Float.floatToIntBits(reel);
28     bits=(bits & 0xFF800000)| (mantisse & 0x7FFFFF);
29     reel = Float.intBitsToFloat(bits);
30     return reel;
31   }
32   static String toHexString(float reel){
33     String chaine;
34     int mantisse,exposant;
35     if (getSigne(reel)==1)
36       chaine="-0x1.";
37     else
38       chaine="0x1.";
39     mantisse= getMantisse(reel)   &0x7FFFFF;
40     do{
41       chaine=chaine+Character.forDigit((mantisse>>>19) ,16);
42       mantisse = (mantisse << 4)  & 0x7FFFFF;
43     } while(mantisse !=0);
44     exposant = getExposant(reel);
45     chaine=chaine+'p'+Integer.toString(exposant,10);
46     return chaine;
47   }
48   static float parseFloat(String chaine){
49     float reel=0.0f;
50     int mantisse=0,exposant=0,decalage=19;
51     int posChaine=5;
52     String ls;
53     if (chaine.charAt(0)=='-'){
54       reel=setSigne(reel,1);
```

```
55        }
56        while (chaine.charAt(posChaine)!='p') {
57          mantisse = mantisse |
58              (Character.digit(chaine.charAt(posChaine),16)<<decalage);
59          posChaine++;
60          decalage = decalage -4;
61        }
62        posChaine++;
63        ls = chaine.substring(posChaine);
64        reel=setMantisse(reel,mantisse);
65        exposant=Integer.parseInt(ls,10);
66        reel=setExposant(reel,exposant);
67        return reel;
68      }
69      static float maximum(){
70        float y=1.0f;
71        while(((1.0f+y)-y)==(1.0f+(y-y))){
72          y=y+1.0f;
73        }
74        return (y-1);
75      }
76      public static void main( String[] args ) {
77        float f=0;
78        System.out.println("Test get ...");
79        System.out.println("getSigne(125.25f)          ="+getSigne(125.25f));
80        System.out.println("getMantisse(125.25f)       ="+Integer.toHexString(getMantisse(125.25
81        System.out.println("getExposant(125.25f)       ="+getExposant(125.25f));
82        System.out.println("Test set ...");
83        f=setSigne(f,0); f=setExposant(f,6); f=setMantisse(f,0xFA8000);
84        System.out.println("125.25= " +f);
85        System.out.println("Test toHexString et parseFloat");
86        System.out.println("Float.toHexString(125.25f)       ="+Float.toHexString(+125.25f));
87        System.out.println("toHexString(125.25f)             ="+toHexString(+125.25f));
88        System.out.println("Float.toHexString(-0.375f)       ="+Float.toHexString(-0.375f));
89        System.out.println("toHexString(-0.375f)             ="+toHexString(-0.375f));
90        System.out.println("Float.parseFloat(+0x1.f5p6)      ="+Float.parseFloat("+0x1.f5p6"));
91        System.out.println("parseFloat(+0x1.f5p6)            ="+parseFloat("+0x1.f5p6"));
92        System.out.println("Float.parseFloat(-0x1.8p-2)      ="+Float.parseFloat("-0x1.8p-2"));
93        System.out.println("parseFloat(-0x1.8p-2)            ="+parseFloat("-0x1.8p-2"));
94        System.out.println("Test Maximum");
95        System.out.println("Maximum       ="+Float.toHexString(maximum()));
96      }
97    }
```

Question 3 : Circuits de manipulation des nombres réels normalisé

— d'afficher en hexadécimal la valeur d'un nombre réel normaliser

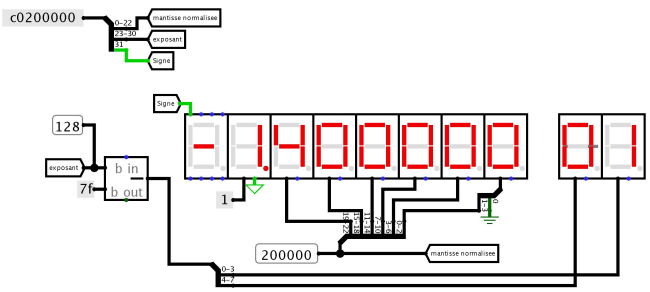


FIGURE 4.17 – Affichage d'un nombre réel normalisé en hexadécimal, correction

— d'additionner deux nombres réels positifs normalisés

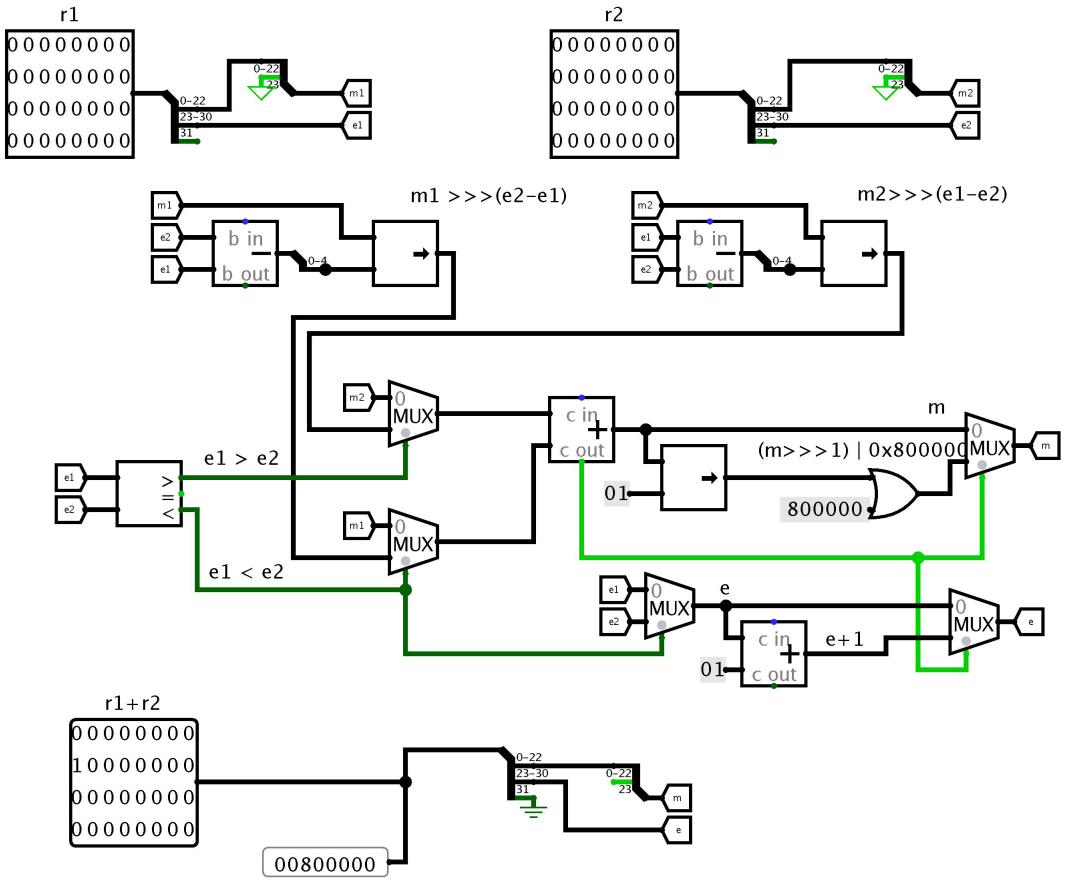


FIGURE 4.18 – Addition de deux réels normalisés positifs, correction

Deuxième partie

Programmation et Architecture des microprocesseurs MIPS

Bibliographie :

Sur le chapitre 6 Programmation MIPS :

1. **Organisation et conception des ordinateurs : L'interface matériel/logiciel DUNOD D Patterson J Hennessy** Chapitre 3
2. **Organisation et conception des ordinateurs : L'interface matériel/logiciel DUNOD D Patterson J Hennessy** Chapitre A (en ligne `http://pages.cs.wisc.edu/~larus/HP_AppA.pdf`
3. **Architecture des Ordinateurs : Une approche Quantitative ITP J Hennessy A Patterson** Chapitre 2.8
4. **Organisation et conception des ordinateurs : L'interface matériel/logiciel DUNOD A Patterson J Hennessy** Chapitre 5.6, 8.1 et annexe A.10
5. **Architecture de l'ordinateur Nicolas P Carter Schaum's** Chapitre 11.

Sur le chapitre 7 Organisation MIPS :

1. **Architecture logicielles et matérielles : Cours, études de cas et exercices corigés DUNOD Ouvrage collectif** Chapitre 2 pages 23 à 45
2. **Organisation et conception des ordinateurs : L'interface matériel/logiciel DUNOD D Patterson J Hennessy** Annexe B.1 à B.3 et B.5
3. **Organisation et conception des ordinateurs : L'interface matériel/logiciel DUNOD D Patterson J Hennessy** Chapitre 4.5
4. **Organisation et conception des ordinateurs : L'interface matériel/logiciel DUNOD D Patterson J Hennessy** Chapitre 5.1 et 5.2
5. **Architecture des Ordinateurs : Une approche Quantitative ITP J Hennessy A Patterson** Chapitre 3.1 à 3.5 et 3.10 à 3.13

Chapitre 5

Introduction

MIPS *Microprocessor without interlocked pipeline stages* est une architecture de microprocesseur de type jeu d'instruction réduit *RISC -Reduced Instruction-Set Computer*. Elle fut développée par la compagnie MIPS Computer Systems Inc. Les processeurs fabriqués selon cette architecture sont surtout utilisés dans les systèmes SGI `http://www.sgi.fr/index.shtml`, dans les systèmes embarqués, comme les ordinateurs de poche, les routeurs Cisco et les consoles de jeux vidéo (Nintendo 64 et Sony PlayStation, PlayStation 2 et PSP). Vers la fin des années 1990, on estimait que les processeurs dérivés de l'architecture MIPS occupaient le tiers des processeurs RISC produits.
Les premières implémentations de l'architecture MIPS étaient de 32 bits (registres et chemins de données), puis passèrent à 64 bits.
Le jeu d'instructions de base est simple, efficace ce qui facilite son apprentissage. Pour illustrer les concepts développés ici, nous utiliserons un simulateur multi-plateforme Java nommé Mars (**M**ips **a**rchitecture and **r**untine **s**imulator) en téléchargement sur `http://courses.missouristate.edu/KenVollmar/MARS/`
Vous trouverez d'autres informations sur la genèse de l'architecture MIPS sur cette page en Anglais : `http://en.wikipedia.org/wiki/MIPS_architecture`

Chapitre 6

La programmation en langage machine du MIPS

Le langage machine permet de programmer un ordinateur au plus bas niveau, à savoir directement le microprocesseur. Pour cela nous utiliserons des *instructions* groupées dans un *jeu d'instructions*.
Le **compilateur** vérifie la syntaxe et transforme le code écrit en *langage de haut niveau* en un code en *langage assembleur* qui sera traduit ensuite en *code objet binaire* par l'**assembleur**.
L'éditeur de liens inclut les bibliothèques pour obtenir un *fichier exécutable*. En cas de dysfonctionnement, le **débuggeur symbolique** permet de recherche les erreurs logiques faites lors de conception du code de haut niveau ou du code assembleur.
La figure Fig. 6.1 présente les étapes de la production d'un programme exécutable.

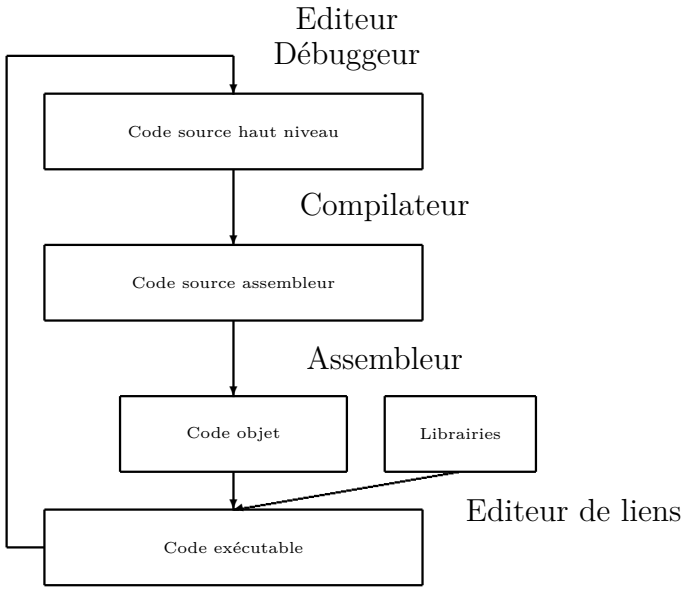


Figure 6.1 – Chaîne de production de programme

6.1 Principe de conception d'un jeu d'instructions

Les deux principes d'un ordinateur à programme enregistré sont la **représentation des instructions comme des nombres entiers non signés** et l'utilisation d'une **mémoire modifiable contenant les programmes et les données**.
La figure Fig. 6.2 présente l'architecture simplifiée d'un ordinateur. Cette architecture est nommée de "Von Neumann".

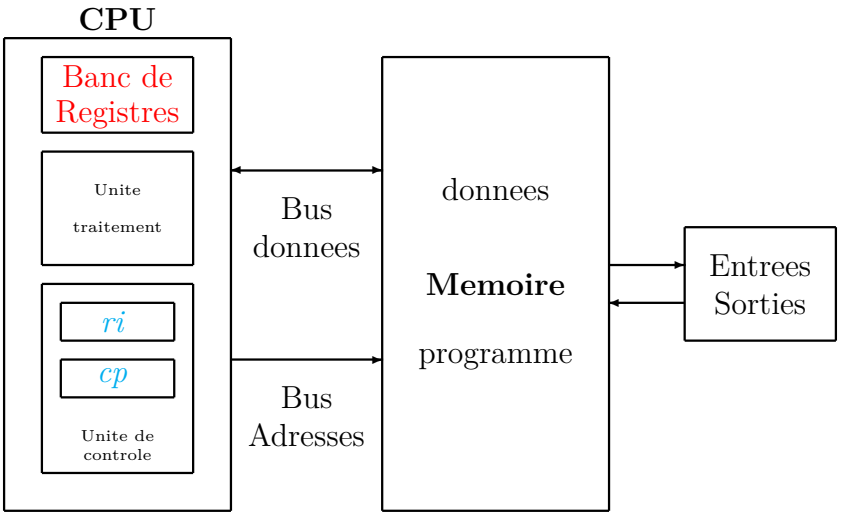


FIGURE 6.2 – Architecture de Von Neumann

Le programme est enregistré dans la mémoire. L'adresse (numéro) de l'instruction en cours de traitement est stockée dans une mémoire interne au microprocesseur nommée **registre compteur de programme**, noté *cp* . L'entier représentant l'instruction en cours de traitement est stocké dans une mémoire interne **nommé registre instruction**, noté *ri* . L'unité de traitement effectue les traitements sur des informations présentes dans des mémoire internes nommées **banc de registres** contenant un ensemble de registres. La mémoire contient aussi les données des programmes. Ces données seront placées dans le banc de registres avant utilisation. Après traitement, elles seront placée dans la mémoire.
Le choix d'un jeu d'instructions influe sur l'architecture matérielle, mais aussi sur le nombre d'instructions nécessaires pour effectuer une suite d'opération. Le jeu d'instruction conditionne le nombre de cycles d'horloge nécessaires à une instruction ainsi que la vitesse d'horloge séquencant les opérations.
Quatre règles guident les concepteurs dans la recherche d'un compromis entre simplicité et efficacité :

1. **Le plus petit est le plus rapide** Petit nombre de registres (32) et nombre d'instructions limité sont des caractéristiques fondamentales de l'architecture *RISC* (acronyme de Reduce Instruction Set Computer).
2. **La simplicité favorise la régularité** La plupart des instructions doivent utiliser la même méthode de codage :
   — taille identique des instructions (32 bits),
   — trois opérandes pour toutes les instructions arithmétiques et logiques
   — taille des données identiques (32 bits)
   — utiliser une même méthode de codage proche pour toutes les autres instructions.
3. **Une bonne conception requiert des compromis** Par exemple, les constantes ne seront pas codées sur 32 bits, puisque toutes les instructions sont elle mêmes codées sur 32 bits (principe de régularité).
4. **Faire en sorte que les cas fréquents soient les plus rapides** Par exemple, les opérations sur les nombres entiers doivent être très rapides, car elles sont utilisées dans tous les types de calcul (pointeurs, caractères, réels).

6.2 Syntaxe des instructions MIPS

Formes générales d'une instruction en langage assembleur MIPS

NomInstruction Opérande 1, Opérande 2, Opérande 3
ou
NomInstruction Opérande 1
ou
NomInstruction

— *NomInstruction* représente le nom symbolique de l'instruction.
— *Opérande* représente une donnée ou le résultat de l'instruction. Les opérandes utilisent les **modes d'adressages** pour savoir comment obtenir la valeur de la donnée recherchée, ou pour savoir où ranger le résultat de l'instruction.

6.3 Codage des instructions

Les instructions sont codées dans des entiers non signés 32 bits. Les opérandes et le code opération sont codés dans des ensembles de bits placés à des endroits spécifiques de l'instruction. Par souci de simplicité, seulement trois types d'instructions existent. Ils sont présentés sur la figure FIG. 6.3 :

type R : Trois opérandes registres,

type I : Deux opérandes registres et une opérande immédiate codée sur 16 bits,

Type J : Une opérande immédiate codée sur 26 bits.

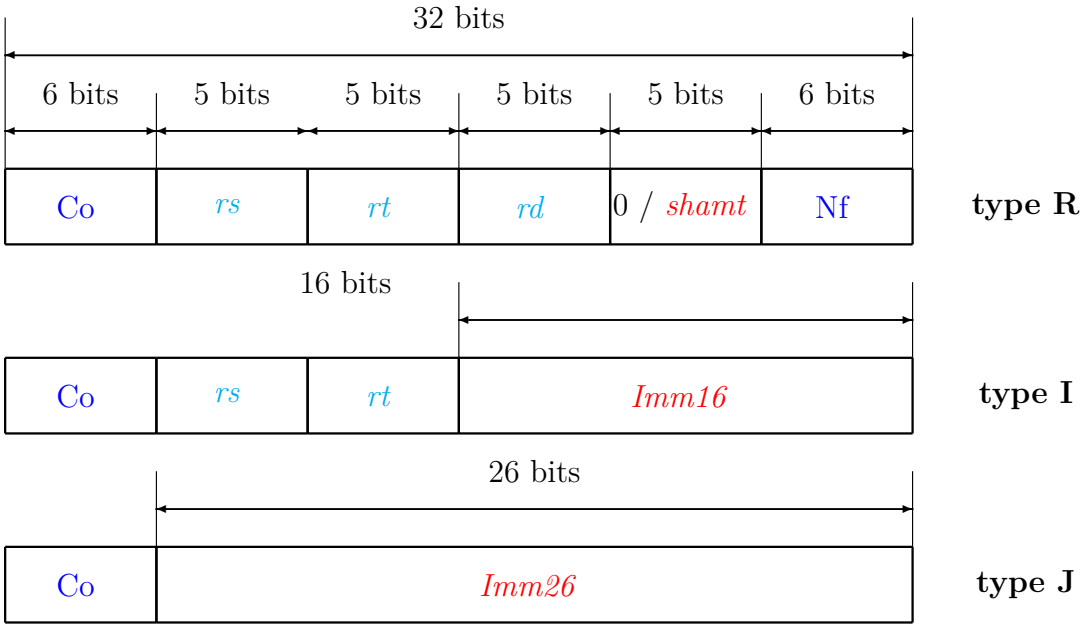


FIGURE 6.3 – Type d'instructions

Les champs de l'instruction sont :

Code Opération (noté Co) : Il est codé dans les bits 26 à 31 et éventuellement un numéro de fonction bits 5 à 0 (noté Nf). Le champs Nf n'existe que dans le cas d'instruction à trois registres.

2 ou 3 Registres : Les trois numéros sont codés dans des paquets de 5 bits, notés *rs* , *rt* et *rd* .

Valeur immédiate 16 bits : Les valeurs immédiates sont comprises entre $-32768 \leq$ *Imm16* ≤ 32767.

Valeur immédiate 26 bits : Ce type d'opérande est utilisé lors des branchements. Les valeurs immédiates sont comprises entre $-2^{25} \leq Imm26 \leq 2^{25} - 1$.

Les instructions avec trois opérandes registres sont codées au format R
Ce type d'instruction manipule uniquement trois registres.

bits 31 à 26 Code opération (noté Co).

bits 25 à 21 Numéro du registre (noté *rs*).

bits 20 à 16 Numéro du registre (noté *rt*).

bits 15 à 11 Numéro du registre (noté *rd*).

bits 10 à 6 Valeur 0 ou *shamt* qui est utilisée par les instruction de décalage (entre 0 et 31)

bits 5 à 0 Numéro de fonction (noté Nf).

Les instructions avec deux opérandes registres et une opérande immédiate sont codées au format I
Ce type d'instruction manipule deux registres et une donnée immédiate codée directement dans l'instruction.

bits 31 à 26 Code opération (noté Co).

bits 25 à 21 Numéro du registre (noté *rs*).

bits 20 à 16 Numéro du registre (noté *rt*).

bits 15 à 0 Valeur Immédiate 16 bits (noté *Imm16*).

Les instructions avec deux opérandes registres et une opérande immédiate sont codées au format J :
Ce type d'instruction manipule uniquement une donnée immédiate.

bits 31 à 26 Code opération (noté Co).

bits 25 à 0 Valeur Immédiate 26 bits (noté *Imm26*).

Remarque 29 :
Le nombre total d'instruction est de 127. En effet le code opération Co est codé sur 6 bits ce qui correspond à 64 instructions. La combinaison **0b**00 0000 est réservée pour les instructions à trois registres utilisant le champs Nf . Ce champs Nf est codé sur 6 bits, ce qui correspond a 64 instructions. Au total nous avons 63+64 soit 127 instructions.

6.4 Mode d'adressage

Les données d'une instruction peuvent se trouver à différents endroit de l'ordinateur. La figure Fig. 6.4 présente ceux ci :

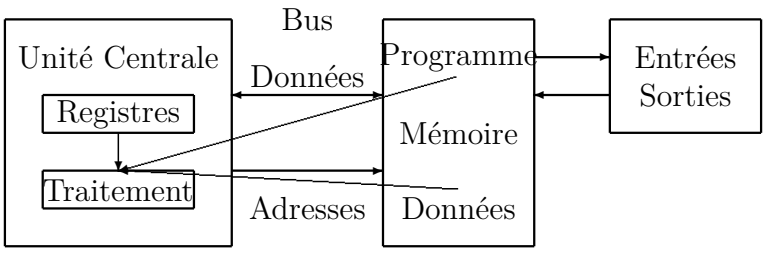


Figure 6.4 – Position des opérandes

L'opérande peut se situer :

— dans le banc de 32 registres,
— dans la mémoire contenant les programmes,
— dans la mémoire contenant les données du programme.

6.4.1 Adressage registre

L'opérande se situe dans un *registre* (mémoire interne au microprocesseur). Une valeur utilisant le mode d'adressage registre est obtenue en utilisant la notation *$i* avec $(0 \leq i \leq 31)$ ou un nom symbolique noté *$NomSymbolique* . Le nom symbolique permet d'accéder au registre en utilisant sa catégorie (temporaires, arguments ...). La figure Fig. 6.6 précise l'utilisation des 32 registres de l'architecture Mips. La figure Fig. 6.5 illustre la détermination d'une opérande registre. Dans l'architecture Mips, le numéro d'un registre est codé sur 5 bits (avec 5 bits nous pouvons représenter 32 combinaisons car $2^5 = 32$). Une instruction de type R possède trois opérande de type registre.

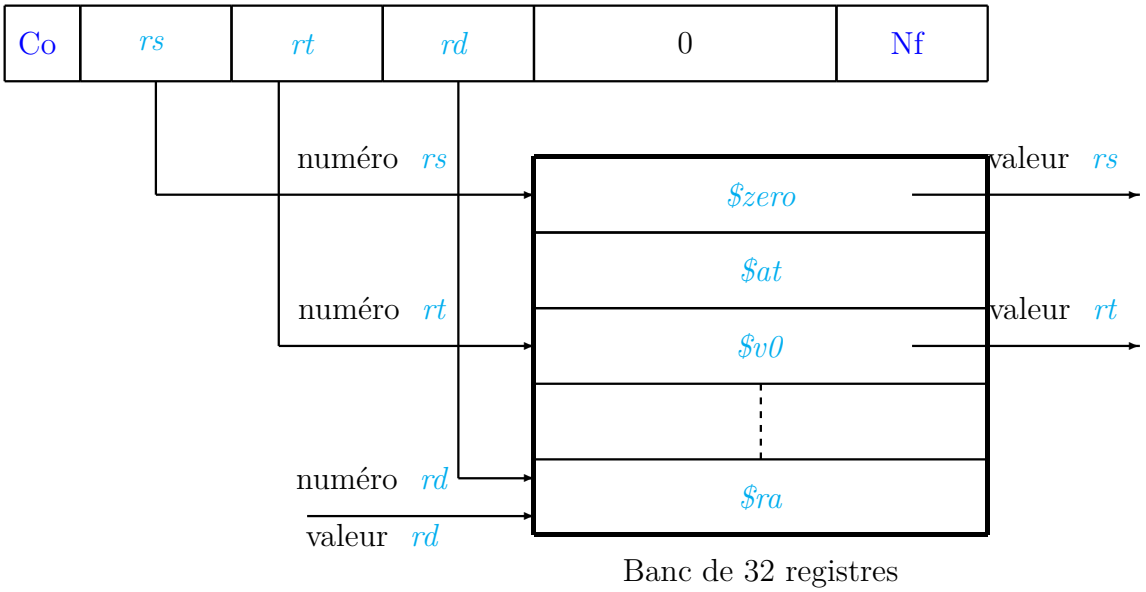


Figure 6.5 – Mode d'adressage registre

Nom symbolique	Numéro	Utilisation
$zero	*$0*	La valeur 0
$at	*$1*	réservé assembleur
$v0	*$2*	résultat de la fonction
$v1	*$3*	résultat de la fonction
$a0	*$4*	argument de la fonction
$a1	*$5*	argument de la fonction
$a2	*$6*	argument de la fonction
$a3	*$7*	argument de la fonction
$t0 à *$s7*	*$8* à *$15*	libres
	$16 à *$27*	Inutilisés dans ce cours
$k0 et *$k1*	*$26* à *$27*	OS kernel
	$28 à *$30*	Inutilisés dans ce cours
$ra	*$31*	adresse retour de fonction

Figure 6.6 – Numéros et noms explicites des registres

Remarque 30 :

Les noms symboliques sont préfixés avec une lettre significative : **a** pour argument, **v** pour value, **t** pour temporary, **k** pour kernel ...

Exemple 40 :

```
add $t0 , $t1 , $t2 # $t0 ← $t1 + $t2
jr $ra # retour d'une fonction à l'adresse stockée dans le registre $ra
```

6.4.2 Adressage immédiat

L'opérande est une **constante** située directement dans l'instruction. Dans le cas de l'architecture MIPScette constante est de taille 16 bits ou 26 bits mais ne peut pas être de taille 32 bits. Une valeur utilisant le mode d'adressage immédiat est obtenue en utilisant la notation *Imm16* ou *Imm26* . La figure FIG. 6.7 illustre la détermination d'une opérande immédiate. Une instrution de type I utilise deux opérandes registre et une valeur immédiate. Une instruction de type J utilise une valeur immédiate.

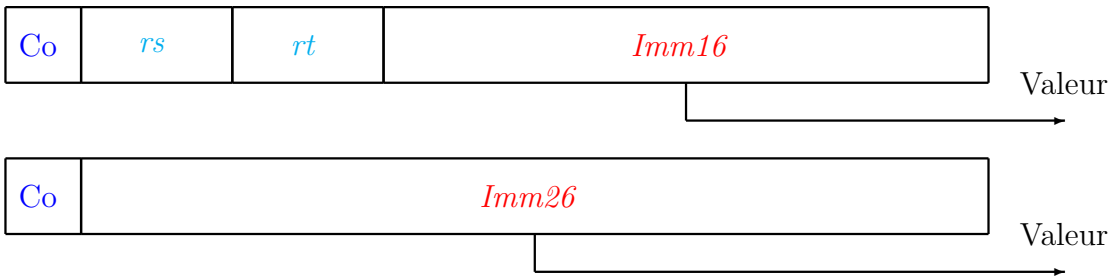


FIGURE 6.7 – Mode d'adressage immédiat

```
addi $t0 , $t1 ,0x7FFF # addi $t0 ← $t1 + 0x7FFF
j 0x37F 0000
```

Dans les exemples précédents , les valeurs immédiates sont **0x**7FFF au format 16 bits et **0x**37F 0000 au format 26 bits.

6.4.3 Adressage indirect immédiat

L'opérande se trouve dans la mémoire de l'ordinateur dont l'adresse est calculée en utilisant la valeur d'un registre et une valeur immédiate signée. Une valeur utilisant le mode d'adressage indirect est obtenue en utilisant la notation *Imm16* (*$NomSymbolique*).La figure FIG. 6.8 illustre la détermination d'une opérande indirecte immédiate. Les instructions manipulant ce mode d'adressage sont de type I ; un registre est une opérande du type registre, et l'autre registre et la valeur immédiate sont utilisée avec le mode d'adressage indirect immédiat.

(6.1) Adresse mémoire = *$i* + *Imm16*

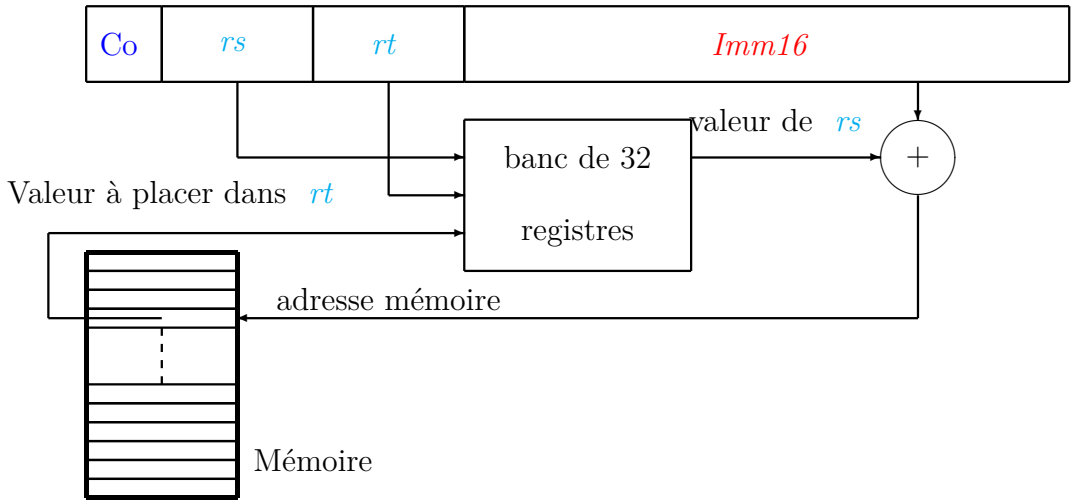


FIGURE 6.8 – Mode d'adressage indirect immédiat

Exemple 41 :

```
la  $t0 ,adresseVariable #  $t0  reçoit l'adresse de la variable
lw  $t1 ,4( $t0 ) #  $t1  reçoit le valeur contenue dans la mémoire d'adresse 4 +  $t0
sw  $t3 ,2( $t0 ) # la mémoire d'adresse 2 +  $t0  reçoit la valeur du registre  $t1
```

Remarque 31 :

L'adresse calculée doit être divisible par 4. En effet les données 32 bits étant représentée sur 4 octets de la mémoire, leur adresse est divisible par 4 (0,4,8,12,16 ...)

La figure FIG. 6.9 résume les notations des modes d'adressage.

Mode d'adressage	**Notation**	**Valeur**
Registre	*$i*	$0 \leq 31$
Immédiat 16 bits	*Imm16*	$-2^{15} \leq Imm16 \leq 2^{15} - 1$
Immédiat 26 bits	*Imm26*	$0 \leq Imm26 \leq 2^{26} - 1$
Indirect immédiat	*Imm16* (*$i*)	adresse = *Imm16* + *$i*

FIGURE 6.9 – Notation des modes d'adressage

Nous allons étudier les principales instructions du jeu d'instructions MIPS. La liste exhaustive des instructions et des pseudo-instructions se trouve dans l'aide du simulateur MARS.

6.5 Instructions arithmétiques et logiques

Les deux notations possibles sont :

1. *Instruction rd , rs , rt* qui effectue l'instruction *rd* ← *rs Instruction rt*
2. *Instruction rt , rs , Imm16* effectue l'instruction *rt* ← *rs Instruction Imm16*

Rôle	Syntaxe	Opération	Codage
addition	add *rd* , *rs* , *rt*	*rd* ← *rs* + *rt*	Co =0 Nf = **0x**20
addition immédiate	addi *rt* , *rs* ,*Imm16*	*rt* ← *rs* + *Imm16*	Co =8
soustraction	sub *rd* , *rs* , *rt*	*rd* ← *rs* - *rt*	Co =0 Nf =**0x**22
multiplication	mul *rd* , *rs* , *rt*	*rd* ← *rs* × *rt*	Co =**0x**1C Nf =2
et bit à bit	and *rd* , *rs* , *rt*	*rd* ← *rs* & *rt*	Co =0 Nf =**0x**24
et bit à bit immédiat	andi *rt* , *rs* ,*Imm16*	*rt* ← *rs* & *Imm16*	Co =**0x**C
ou bit à bit	or *rd* , *rs* , *rt*	*rd* ← *rs* | *rt*	. Co =0 Nf =**0x**25
ou bit à bit immédiat	ori *rt* , *rs* ,*Imm16*	*rt* ← *rs* | *Imm16*	Co =**0x**D
non ou bit à bit	nor *rd* , *rs* , *rt*	*rd* ← ∼ (*rs* | *rt*)	. Co =0 Nf =**0x**27
décalage à gauche	sll *rd* , *rt* ,shamt	*rd* ← *rt* << shamt	Co =0 Nf =0
décalage à gauche	sllv *rd* , *rt* , *rs*	*rd* ← *rt* << *rs*	Co =0 Nf =4
décalage droite arithm	sra *rd* , *rt* ,shamt	*rd* ← *rt* >> shamt	Co =0 Nf =3
décalage droite log	srl *rd* , *rt* ,shamt	*rd* ← *rt* >>> shamt	Co =0 Nf =2
charger demi mot poids fort	lui *rt* ,*Imm16*	*rt* ← *Imm16* << 16	Co =**0x**F

FIGURE 6.10 – Liste partielle des instructions arithmétiques et logiques

Le jeu d'instruction est **non destructif** car la syntaxe à trois opérandes préserve éventuellement les données en permettant le stockage du résultat dans une autre opérande.

Exemple 42 :

```
addi $t0 , $zero ,0 # $t0 ← $zero + 0
add $t1 , $0 , $0 # nous pouvons utiliser les numéro de registre ou nom
add $t2 , $at , $t0
```

Remarque 32 :

— Les **pseudo-instructions** permettent de manipuler des valeurs immédiates codées sur 32 bits. Celles-ci utilisent le registre nommé *$at* qui est réservé à l'assembleur et une séquence d'instructions est implémentée pour réaliser cette pseudo-instruction.

— Une constante 32 bits d'une instruction de type I est décomposée en deux constantes 16 bits. Pour affecter une constante 32 bits nous utiliserons une pseudo-instruction. Celle-ci est décomposée en une suite d'instructions ou nous trouvons cette constante divisée en deux parties.

```
addi $t0 , $zero ,0x1234 ABCD
```

est traduite par l'assembleur en la suite d'instructions suivante

```
lui $at ,0x1234
ori $at , $at ,0xABCD
add $t0 , $zero , $at
```

ou sans utiliser les noms symboliques des registres.

```
lui $1 ,0x1234
ori $1 , $1 ,0xABCD
add $8 , $0 , $1
```

— la est une pseudo instruction qui place dans le registre résultat l'adresse d'une donnée. En effet, les adresses sont codées sur 32 bits, nous ne pouvons pas les placer directement dans l'instruction. Les données commencent à l'adresse **0x**1001 0000. Soit une variable nommée var_x placée à l'adresse **0x**1001 0004, la pseudo instruction la *$t0* ,$var_x$ sera décomposée de la façon suivante :

```
lui $1 ,0x1001 # $at ← 0x1001 0000
ori $8 , $1 ,0x0004 # $t0 ← $1 | 4 = 0x1001 0004
```

6.6 Instructions de chargement et rangement

Le jeu d'instruction du MIPSest **chargement/rangement** car toutes les données manipulées par les instructions arithmétiques et logiques doivent être placées dans les registres. Les instructions de chargement et de rangement permettent d'effectuer les transferts entre la mémoire centrale et les registres et utilisent exclusivement le mode indirect immédiat. La notation [adresse] exprime le contenu de la mémoire dont l'adresse est spécifiée. Le chargement permet de prendre une valeur mémoire et de la placer dans un registre. Le rangement permet de prendre une valeur contenue dans un registre et de la placer dans la mémoire.

Rôle	Syntaxe	Opération	Codage
chargement 32 bits	lw *rt* ,*Imm16* (*rs*)	*rt* ← [*Imm16* + *rs*]	Co =**0x**23
chargement 8 bits	lb *rt* ,*Imm16* (*rs*)	*rt* ← [*Imm16* + *rs*]	Co =**0x**20
rangement 32 bits	sw *rt* ,*Imm16* (*rs*)	[*Imm16* + *rs*] ← *rt*	Co =**0x**2B
rangement 8 bits	sb *rt* ,*Imm16* (*rs*)	[*Imm16* + *rs*] ← *rt*	Co =**0x**28

FIGURE 6.11 – Liste partielle des instructions chargement/rangement

Exemple 43 :

```
lw  $t0 ,4( $a0 ) # Mettre en  $t0  la valeur 32 bits dans la mémoire d'adresse  $a0 +4
add  $t0 , $t0 , $t1
sw  $t0 ,0( $a0 )# Placer le contenu de  $t0  dans la mémoire d'adresse  $a0 +0
```

Remarque 33 :

— Les instructions manipulant des données 32 bits doivent accéder à des adresses mémoires alignées sur des mots de 32 bits, c'est-à-dire divisibles par 4. En effet, la mémoire physique du système contient des données 8 bits ; les données 32 bits sont stockées dans des groupes 4 octets consécutif.

Exemple 44 :

```
la  $t0 ,0x1001 1001
lw  $t1 ,3( $t0 ) #  $t1  ←[0x1001 1100]
```

6.7 Instructions de branchement

Les instructions de branchement rendent non séquentiel le déroulement d'un programme. Elles permettent de se rendre à une **instruction cible** autre que l'instruction suivante du programme. La taille des instructions étant de 32 bits, leurs adresses sont divisibles par 4 ce qui signifie que les deux bits de poids faibles *LSB* sont égaux à 0 et ne sont pas codés dans la valeur immédiate. Le registre **compteur de programme** (noté *cp*) indique l'instruction en cours de traitement. Il est modifié par les instructions de branchement si le branchement est pris. Si ce n'est pas le cas (branchement non pris), l'instruction suivante du programme (en *cp* + 4) sera traitée. L'adresse de l'instruction cible peut être obtenue de différentes façons.

6.7.1 Valeur immédiate 16 bits signée

La valeur immédiate 16 bits représente le nombre d'instructions entre l'instruction de branchement et l'instruction cible. Si ce nombre est négatif, l'instruction cible se trouve avant l'instruction de branchement, si ce nombre est positif après l'instruction suivante, s'il vaut 0 la cible est l'instruction suivante (aucun intérêt puisque l'instruction suivante est traitée que le branchement soit pris ou non pris).

Les instructions sont codées sur 32 bits et utilise 4 octets. L'adresse des instructions est donc divisible par 4. Nous devons donc convertir un nombre d'instructions en un nombre d'octets. Pour cela, nous devons effectuer une multiplication par 4 (décalage de deux rang à gauche). Ce mode d'adressage est nommé **relatif au compteur de programme** car nous ajoutons un déplacement au compteur de programme *cp* . Dans ce cas les deux registres sont comparés pour prendre la décision de branchement. La figure FIG. 6.12 présente ce mode d'adressage.

$$\text{Adresse de l'instruction cible} = (cp + 4) + Imm16 << 2$$

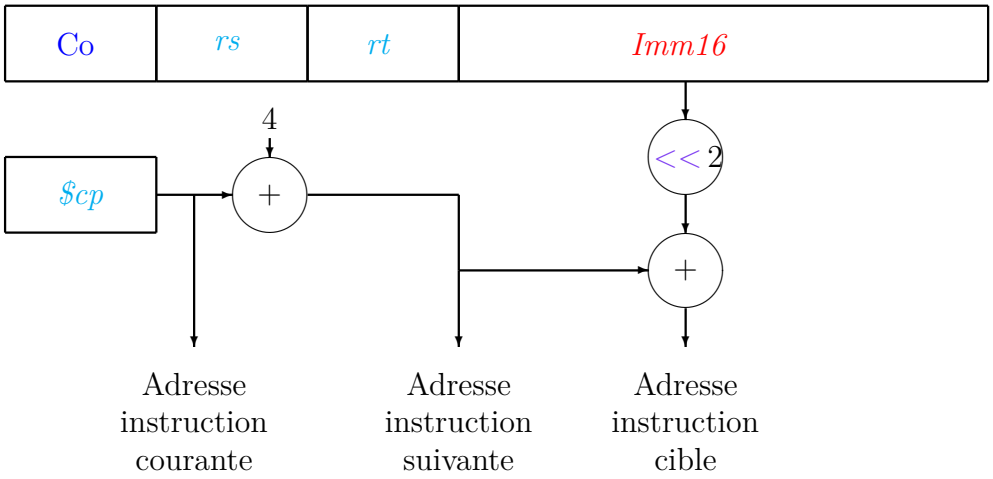


FIGURE 6.12 – Mode d'adressage relatif au compteur de programme

Exemple 45 :

Cible : ....
bne $t1 , $t0 ,Cible # $t1 ≠ $t0 aller à l'instruction cible

6.7.2 Valeur immédiate 26 bits non signée

Avec le décalage de deux bits vers la gauche de la valeur non signée 26 bits (2 bits LSB à 0), concaténée avec les quatre bits de poids fort du **compteur de programme**, les instructions cibles se situent dans $\frac{1}{16}$ de la mémoire autour du registre *cp* . Ce mode d'adressage est nommé **pseudo-direct**. Dans ce cas le branchement est inconditionnel (toujours pris) puisque nous ne pouvons pas coder des informations concernant la prise de décision. La figure FIG. 6.13 présente ce mode d'adressage.

$$\text{Adresse de l'instruction cible} = (cp \ \&\ \mathbf{0x}F000\,0000) \mid Imm26 << 2$$

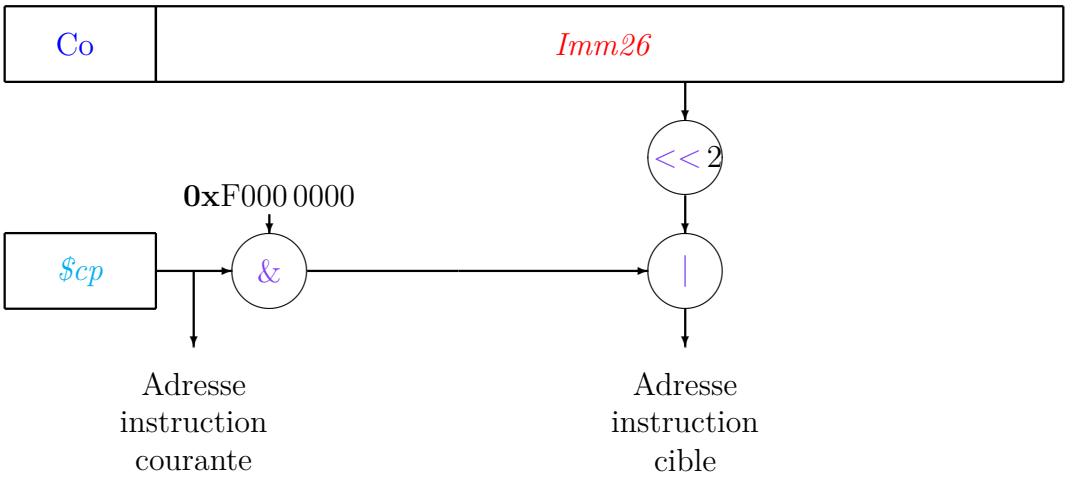


FIGURE 6.13 – Mode d'adressage pseudo-direct

j **0x**100 1234

6.7.3 Valeur 32 bits non signée

L'adresse de la cible est codée dans un registre et l'instruction cible se trouve n'importe où en mémoire.

$$\text{Adresse de l'instruction cible} \quad = \quad \$i$$

Exemple 46 :

jr $ra # cp ← $ra

Remarque 34 :

Le registre $ra est utilisé pour le retour de sous-programme. Tous les registres peuvent être aussi opérande de cette instruction.

6.7.4 Liste des instructions de branchement

La figure FIG. 6.14 présente les principales instructions de branchement. Elles sont de deux types :

1. Instructions de branchement conditionnelle : Dans ce cas le branchement est pris si la condition logique est vraie, elles utilisent le mode d'adressage relatif au compteur de programme.
2. Instruction de saut inconditionnelle : Dans ce cas le branchement est toujours pris, elles utilise les mode d'adressage pseudo-direct ou registre.

Rôle	Syntaxe	Opération	Codage
branchement si égal	beq *rs* , *rt* ,*Imm16*	si *rs* = *rt* alors *cp* ← (*cp* +4)+*Imm16* << 2	Co =4
branchement si non égal	bne *rs* , *rt* ,*Imm16*	si *rs* ≠ *rt* alors *cp* ← (*cp* +4)+(*Imm16* << 2)	Co =5
branchement si ⩽	ble *rs* , *rt* ,*Imm16*	si *rs* ⩽ *rt* alors *cp* ← (*cp* +4)+(*Imm16* << 2)	pseudo
branchement si positif	bgtz *rs* ,*Imm16*	si *rs* > 0 alors *cp* ← (*cp* +4)+(*Imm16* < < 2)	Co =7
branchement si négatif	blez *rs* ,*Imm16*	si *rs* ≤ 0 alors *cp* ← (*cp* +4)+(*Imm16* << 2)	Co =6
saut 26 bits	j *Imm26*	alors *cp* ← (*cp* & **0x**F000 0000) | (*Imm26* << 2	Co =2
saut 32 bits	jr *rs*	*cp* ← *rs*	Co =0 et Nf =8
appel de fonction	jal *Imm26*	*cp* ← (*cp* & **0x**F000 0000) | (*Imm26* << 2) puis *$ra* ← *cp* +4	Co =3

FIGURE 6.14 – Liste partielle des instructions de branchement

Remarque 35 :

— Il existe des pseudo-instructions de branchement permettant de faire une comparaison par rapport à une valeur immédiate.
La pseudo-instruction
bne *$t0* ,100,étiquette
est traduite par l'assembleur en la suite d'instructions :
addi *$at* , *$zero* ,100
bne *$8* , *$at* ,étiquette
— L'instruction jal permet d'appeler des fonctions en stockant l'adresse de l'instruction suivante celle ayant provoqué le branchement.
— L'instruction jr permet de revenir d'une fonction.

6.8 Instructions Diverses

Rôle	Syntaxe	Opération	Codage
No opération	nop	aucune	0
Appel Système	syscall	Voir ci dessous	Co =0 et Nf =**0x**C

FIGURE 6.15 – Liste des instructions diverses

Les appels système permettent de créer des liens entre le microprocesseur MIPSet l'extérieur (principalement gestion du clavier, de l'écran et des périphériques). Le système est accessible via une *API* simplifiée implémentant les fonctions suivantes :

Nom	*$v0*	Arguments	Résultat
Afficher entier	1	*$a0*	
Afficher chaine	4	*$a0* =adresse	
Lire entier	5		*$v0*
Lire chaine	8	*$a0* =adresse *$a1* =taille	
Afficher caractère	11	*$a0*	
Lire caractère	12		*$v0*
Fin programme	10		

FIGURE 6.16 – Liste partielle des appels système

Exemple 47 :

addi *$a0* , *$zero* ,10 # mise en place de l'argument valeur à afficher ici 10
addi *$v0* , *$zero* ,1 # Appel système numéro 1, afficher un entier
syscall # Affiche la valeur 10

Remarque 36 :

L'aide du simulateur MARS utilisé pour les exercices pratiques donne la liste complète des appels système et la liste complète des instructions et pseudo instructions

6.9 Programmation en assembleur

Dans cette partie nous allons étudier les directives de l'assembleur. Une directive n'est pas une instruction, ni une pseudo-instruction. Ce sont des indications données à l'assembleur pour que celui-ci puisse réaliser son travail correctement. Les directives commencent toujours par le caractère point ".
Forme générale d'une instruction en langage assembleur MIPS.
étiquette : *NomInstruction Opérande 1, Opérande 2, Opérande 3* #Commentaire
L'étiquette est un nom symbolique qui facilite la programmation. Il associe un nom à la ligne du programme assembleur. Ceci évite au programmeur de manipuler les adresses mémoire codées en hexadécimale. Le nom de l'étiquette doit être unique dans le code source.
Commentaire est une information non assemblée. Tout ce qui suit le caractère # est ignoré par l'assembleur.

6.9.1 Segments du programme

Comme dans un algorithme et dans tous les langages de programmation, un programme assembleur contient une **partie déclaration des données** et une partie contenant **les instructions**. Le LISTING 6.1présente les deux segments.
La directive .data indique le début du segment de données, et la directive .text le début des instructions du programme.

Listing 6.1 – Fichier assembleur minimal

```
1  .data # Le segment donnees du programme
2    # donnes ici
3  .text # Le segment text -- instructions du prog
4    # instructions ici
5    #  Fin du programme appel systeme
6    addi $v0, $zero, 10  # Syscall 10 pour terminer
7    syscall              # Sortir maintenant
```

La figure FIG. 6.17 énumère les directives de l'assembleur MIPS.

Contenu	Type	Exemple
Chaine	.asciiz str	msg : .asciiz "Bonjour"
Octet	.byte b1,..bn	b : .byte **0x**12
Réel 64 bits	.double d1,...,dn	d : .double 1.5
Réel 32 bits	.float f1,...fn	f : .float 1.5
Entier 32 bits	.word i1,...in	w : .word **0x**FF
Espace vide	.space n	.space **0x**FF
Alignement	.align n	.align 2

FIGURE 6.17 – Liste des types définis par l'assembleur

Remarque 37 :

- Comme en Java le préfixe 0x représente un nombre hexadécimal.
- La directive **.align** permet de faire commencer la donnée suivante à une adresse divisible par 2^n (indispensable pour utiliser les instructions de chargement/rangement 32 bits qui doit pointer sur une adresse ayant les deux bits *LSB* à 0).
- Une chaîne de caractères est terminée par le caractère de code ascii 0 (non pas le code ascii de 0 !).
- Le caractère $\backslash n$ dans une chaîne de caractères permet d'effectuer un passage à la ligne suivante.

Voici un premier programme : Le classique **Hello World**

Listing 6.2 – Hello World

```
1  .data #debute à l'adresse 0x1001 0000
2    Msg:    .asciiz "Hello World \n"
3  .text
4  main: #debute à l'adresse 0x0040 0000
5    #Affichage de la chaine de caractères
6    addi $v0, $zero, 4     # Sycall 4 afficher une chaine
7    la $a0, Msg            # Adresse de la chaine
8    syscall                # Afficher maintenant
9    # Fin
10   addi $v0, $zero, 10
11   syscall
```

6.9.2 Programmation de l'alternative

L'instruction algorithmique classique est le **Si** :

Algorithme 58 Alternative

```
1: Début
2: Si (condition = VRAI) Alors
3:    instruction(s) //Effectué si la condition est vraie
4: Sinon
5:    instruction(s) //Effectué si la condition est fausse
6: FinSi
7: Fin
```

Se traduit en assembleur MIPSpar la suite d'instructions :

beq *rs* , *rt* sinon # ou une autre instruction
alors : # étiquette non utilisée
instruction(s) - Effectuée(s) si le branchement n'est pas pris : *rs* ≠ *rd*
j finsi : # Aller à l'étiquette finsi.
sinon :
#*instruction(s)* Effectuée(s) si le branchement est pris : *rs* = *rt*
finsi :
#*instruction(s)* # Effectué(s) dans tous les cas

Exemple 48 :

Test de l'égalité de deux nombres :

Listing 6.3 – Programmation de l'alternative

```
1  .data
2    Lecture: .asciiz "Entrez un nombre ?\n"
3    Egaux:   .asciiz "Nombres egaux \n"
4    Diff:    .asciiz "Nombre differents \n"
5  .text
6    addi $v0, $zero, 4    # Afficher chaine
7    la $a0,Lecture
8    syscall
9    addi $v0, $zero, 5   # Lire un entier
10   syscall
11   add $t0, $v0, $zero
12   addi $v0, $zero, 4    # Afficher chaine
13   la $a0,Lecture
14   syscall
15   addi $v0, $zero, 5   # Lire entier
16   syscall
17   bne $t0, $v0, Sinon # si rs != rt aller a sinon
18 Alors:
19   la $a0,Egaux # Adresse de la chaine Egaux
20   j FinSi
21 Sinon:
22   la $a0,Diff # Adresse de la chaine Diff
23 FinSi:
24   addi $v0, $zero, 4    # Afficher chaine
25   syscall
26   addi $v0, $zero, 10  # Terminer
27   syscall
```

6.9.3 Programmation des itérations

L'instruction algorithmique la plus générale est l'itération nommé **Tant Que** :

Algorithme 59 itération Tant Que

1: **Début**
2: **Tant Que** (*condition* = *VRAI*) **Faire**
3: *instruction(s)*
4: **Fin Tant Que**
5: **Fin**

Se traduit en assembleur MIPSpar la suite d'instructions :

TantQue :
bne *rs* , *rt* FinTantQue # si (*rs* = *rt*) aller à la fin de l'itération
Faire : # étiquette non utilisée
#instruction(s)
j TantQue # retour au début de l'itération
FinTantQue :

Exemple 49 :

Calcul de $n!$

Listing 6.4 – Factorielle n

```
1  .data
2    Lecture: .asciiz "Entrez un nombre ?\n"
3    Fact:  .asciiz "Fact("
4    Egal:     .asciiz ")="
5  .text
6    addi $v0, $zero, 4    # Afficher chaine
7    la $a0,Lecture
8    syscall
9    addi $v0, $zero, 5   # Lire entier
10   syscall
11   add $t1, $v0 , $zero # Garder la valeur lue pour l'afficher
12   addi $t0, $zero, 1
13 TantQue:
14   beq $v0,$zero, FinTantQue # si $v0=0 aller a FinTantQue
15   mul $t0,$t0, $v0 # $t0 <- $t0 * $v0
16   addi $v0, $v0, -1 # $v0 <- $v0 - 1
17   j TantQue
18 FinTantQue:
19   addi $v0, $zero, 4    # Afficher "Fact("
20   la $a0, Fact
21   syscall
22   addi $v0, $zero, 1    # Afficher n
23   add $a0, $t1, $zero
24   syscall
25   addi $v0, $zero, 4    # Afficher ")="
26   la $a0, Egal
27   sisal
28   addi $v0, $zero, 1    # Afficher entier
29   add $a0, $t0, $zero
30   syscall
31   addi $v0, $zero, 10  # Fin
32   syscall
```

6.9.4 Notion de fonctions

Une fonction contient un bloc de code réutilisable. Elle possède des arguments et elle renvoie une valeur scalaire.
Lors de l'appel d'une fonction, le programme doit réaliser les opérations suivantes :

1. Placer les paramètres d'entrée dans les arguments de la fonction.
2. Sauvegarder l'adresse de l'instruction suivante ayant appelé la fonction (pour pouvoir y revenir ensuite).
3. Exécuter les instructions de la fonction.
4. Revenir à l'instruction suivante du programme ayant appelé la fonction

Exemple 50 :

Algorithme d'une fonction et de son appel :

Algorithme 60 Fonction

1: *res* : entier
2: **Fonction** *NomFonction* (liste des paramètres formels :entier) : entier
3: **Début**
4: *instruction(s)*
5: return ...
6: **Fin fonction**
7: **Début**
8: *res* ← *NomFonction*(liste des paramètres réels)
9: **Fin**

En assembleur MIPS, les paramètres sont placés dans les registres *$a0* à *$a3* . Le résultat de la fonction est placé dans le registre *$v0* et *$v1*

```
#mise en place des paramètres
add $a0 , ....
add $a1 ,
...
#Appel de la fonction
jal NomFonction # mettre l'adresse instruction suivante dans $ra puis traiter la fonction
#instruction suivante
....

NomFonction :
#instructions de la fonction
...
add $v0 , ... #placer le résultat dans $v0
jr $ra # aller à l'adresse contenue dans le registre $ra
```

Remarque 38 :

Les étiquettes doivent être unique dans le code assembleur. Pour cela, elles seront préfixées avec le nom de la fonction. Si la fonction s'appelle *factorielle* et que nous désirons appeler l'étiquette finsi, l'étiquette du programme sera nommée factorielle_finsi : ou factorielleFinsi :

Exemple 51 :

Listing 6.5 – n ! avec une fonction

```
1  .data
2    Lecture: .asciiz "Entrez un nombre ?\n"
3    MFact:   .asciiz "Fact("
4    Egal: .asciiz ")="
5  .text
6    addi $v0, $zero, 4     # Afficher chaine
7    la $a0,Lecture
8    syscall
9    addi $v0, $zero, 5    # Lire entier
10   syscall
11   add $a0, $v0, $zero   # Argument de fact dans $a0
12   add $t0, $a0, $zero   # Sauvergarder a0 pour l'afficher
13   jal Fact
14   add $t1, $v0, $zero   # Sauvergarde $v0
15   addi $v0, $zero, 4    # Afficher chaine
```

```
16    la $a0, MFact
17    syscall
18    addi $v0, $zero, 1   # Afficher entier
19    add $a0, $t0, $zero
20    syscall
21    addi $v0, $zero, 4   # Afficher chaine
22    la $a0, Egal
23    syscall
24    addi $v0, $zero, 1   # Afficher entier
25    add $a0, $t1, $zero
26    syscall
27   # Fin
28    addi $v0, $zero, 10
29    syscall
30 #Fonction Fact Parametre entree dans $a0, retour dans $v0
31 Fact:
32    addi $v0, $zero, 1   # Fact(0)=1
33 Fact_TantQue:
34    beq $a0, $zero, Fact_FinTantQue
35    mul $v0, $v0, $a0
36    addi $a0, $a0, -1
37    j Fact_TantQue
38 Fact_FinTantQue:
39    jr $ra # retour au programme appelant
```

6.10 Exercices

Exercice 7 : Exercice théorique : Instructions mips et programmation assembleur

Question 1 : Codage des instructions Mips Donner les valeurs hexadécimales 32 bits représentant les instructions MIPSsuivantes :

1. add *$t0* , *$a0* , *$v0*
2. sra *$t0* , *$v0* ,31
3. addi *$v0* , *$a0* ,-1
4. lw *$v0* ,4(*$a0*)
5. sw *$v0* ,4(*$a0*)
6. bne *$a0* , *$v0* , instruction courante
7. bne *$a0* , *$v0* , deux instructions plus loin

Indication 1 :

- Pour les instructions de décalage, le nombre de décalage est codé dans le champs *shamt* .
- Pour les branchements, la valeur immédiate représente l'écart entre l'adresse de l'instruction suivant le branchement et l'adresse de l'instruction cible, qui est décalé à droite de deux rangs.
- Vous observerez avec attention l'ordre des registres dans la description des instructions.

Question 2 : Instruction(s) des pseudo-instructions Mips Donnez l'instruction (ou les instructions) MIPSremplaçant les pseudo-instructions nommées :

1. **move** *$t1* , *$t0* qui effectue le transfert du registre *$t0* dans le registre *$t1* .

2. **not** *$t1* , *$t0* qui effectue l'inversion bit à bit du registre *$t0* et place le résultat dans le registre *$t1* .Vous utilisez l'instruction nor .
3. beq *$t1* ,*Imm16* ,étiquette qui utilise l'instruction beq *rs* , *rt* ,étiquette et le registre réservé pour les pseudo-instruction nommé *$at* (ou *$1*).

Question 3 : Analyse des pseudo-instructions Expliquez le code des pseudo-instructions suivantes :

1. rol *$t1* , *$t0* ,1 qui est décomposée en :
   srl *$1* , *$8* ,**0x**001F
   sll *$9* , *$8* ,1
   or *$9* , *$9* , *$1*
2. ror *$t1* , *$t0* ,1 qui est décomposée en :
   sll *$1* , *$8* ,**0x**001F
   srl *$9* , *$8* ,1
   or *$9* , *$9* , *$1*
3. abs *$t1* , *$t0* qui est décomposée en :
   sra *$1* , *$8* ,**0x**001F
   xor *$9* , *$1* , *$8*
   subu *$9* , *$9* , *$1* # soustraction sans tenir compte de la retenue
   Pour cette dernière, vous regarderez le résultat pour un nombre négatif puis un nombre positif

Pour les algorithmiques des questions suivantes, vous utilisez les instructions ci-dessous.

- Vous n'utiliserez pas les structures algorithmiques classiques (Si, Pour, Tant Que et Répéter Jusqu'à), mais uniquement la structure :
  - **SiCond** (*$rs* opérateur *Imm16*) **Aller à** Etiquette
  - **SiCond** (*$rs* opérateur *$rt*) **Aller à** Etiquette
- Les expressions arithmétiques seront de la forme $a \leftarrow b\ Op\ c$.
- Vous accèderez aux variables mémoire 32 bits ou 8 bits en utilisant les fonctions de chargement et rangement suivantes :
  - *$t0* ← Mem32[*$t1* +*Imm16*]
  - Mem32[*$t0* +*Imm16*]← *$t1*

  Ou pour une information codée sur 8 bits,
  - *$t0* ← Mem8[*$t1* +*Imm16*]
  - Mem8[*$t0* +*Imm16*]← *$t1*
- Les fonctions se termineront avec le mot réservé **Retourner**.
- Pensez à effectuer une indentation correcte de vos algorithmes.

Question 4 : Algorithme orienté assembleur *forDigit* Complétez l'algorithme de la fonction *forDigit* .

Indication 1 :

- Le sinon de l'alternative sera supprimée en calculant dans tous les cas un résultat, puis en vérifiant la validité de celui-ci.
- Le code ascii du caractère 0 est **0x**30 et celui du caractère A est de **0x**41

Fonction *forDigit* (*$a0*) : *$v0*

Complétez l'algorithme suivant :

Algorithme 61 Conversion d'un digit en un caractère optimisé (énoncé)

```
1: Fonction forDigit ( $a0 ) : $v0
2: Début
3:     $v0 ← $a0 + ....
4:   SiCond (a0...) Aller à forDigitFinsi
5:     $v0 ← $v0 + ...
6: forDigitFinsi :
7: Retourner
8: Fin fonction
```

Question 5 : Algorithme orienté assembleur *digit* Sur le principe de l'algorithme précédent, modifier l'algorithme de la fonction *digit* pour utiliser une structures **SiCond** () **Aller à** et deux soustractions. Le code ascii passé est supposé valide c'est à dire représentant un digit hexadécimal. Nous effectuerons uniquement les tests nécessaires à un caractère supposé valide. La fonction n'utilisera pas de base de conversion.
Fonction *digit* (*$a0*) : *$v0*

Question 6 : Algorithme orienté assembleur *initMem* Complétez l'algorithme d'une fonction nommée *initMem* permettant d'initialiser une zone mémoire débutant à l'adresse *$a0* avec la valeur entière 32 bits *$a2* . Le registre *$a1* (*$a1* ≥ 0) contient la taille de la zone mémoire à initialiser, exprimée en nombre de valeurs entières 32 bits (codées dans 4 octets). Vous utiliserez deux structures **SiCond** () **Aller à** , un rangement mémoire Mem32[]← et deux opérations arithmétiques.
Fonction *initMem* (*$a0* , *$a1* , *$a2*)

Algorithme 62 Initialisation d'une zone mémoire énoncé

```
1: Fonction InitMem ( $a0 , $a1 , $a2 )
2: Début
3:   SiCond ( $a1 = 0) Aller à ...
4: initMemBoucle :
5:   Mem32[...]← ...
6:     $a0 ← $a0 + ...
7:     $a1 ← $a1 - ...
8:   SiCond (...) Aller à initMemBoucle
9: initMemFin :
10: Retourner
11: Fin fonction
```

Question 7 : Algorithme orienté assembleur *copieMem* Modifier l'algorithme d'une fonction nommée *initMem* pour copier une zone mémoire débutant à l'adresse *$a0* dans une zone mémoire débutant à l'adresse *$a1* . La taille de cette zone mémoire sera exprimée en nombre de valeurs entières 32 bits à copier dans *$a2* (avec *$a2* > 0).
Fonction *copieMem* (*$a0* , *$a1* , *$a2*)

Question 8 : Algorithme orienté assembleur *copieChaine* Modifier l'algorithme d'une fonction nommée *copieMem* pour copier une chaîne de caractères débutant à l'adresse *$a0* dans une une chaîne de caractères débutant à l'adresse *$a1* .
Indication 1 :

— Les caractères de la chaîne sont codés dans des octets.
— La chaîne de caractères est terminée par le caractère de code ASCII zéro.
— Vous devez effectuer au moins une fois l'itération afin de copier le caractère de fin de chaîne lorsqu'elle est vide.

Question 9 : Algorithme orienté assembleur *toHexString* Modifier l'algorithme de la fonction *toHexString* sur les nombres non signés pour utiliser une structure **SiCond** () **Aller à** .

Algorithme 63 Conversion d'un nombre non signé en une chaîne de caractères hexadécimale - énoncé

```
1: Fonction toHexString ( $a0 , $a1 ) :
2: Variables
3:     $t0 , $t1 , $t2 : entier
4: Début
5:     $t0 ← ... //Compteur d'itération
6:     $t1 ← ... //Sauvegarde du nombre à convertir
7:     $t2 ← ... //Sauvegarder de l'adresse de retour
8: toHexStringBoucle :
9:     $a0 ← ... //Octet de gauche
10:    $v0 ← forDigit( $a0 )
11:   Mem8[...]← ... //Ranger le caractère dans la chaîne
12:    $a1 ← ... //Modifier l'adresse pour le caractère suivant
13:    $t1 ← ... //Modifier la valeur du nombre à convertir pour la prochaine itération
14:    $t0 ← ... //Modifier le compteur de boucle
15:   SiCond (...) Aller à ... //Condition d'arrêt
16:   Mem8[...]← ... //Mettre le caractère de fin de chaîne
17:    $ra ← $t2 //Restaurer l'adresse de retour
18: Retourner
19:   Fin fonction
```

Indication 1 :

— La chaîne de caractères résultat contiendra 8 caractères (même s'ils sont à zéro)
— Vous traiterez les digits du nombre de gauche à droite. En effet, nous pouvons ajouter simplement des caractères à la fin de la chaîne et non pas au début.
— Le registre $a0$ contient l'entier non signé à convertir et le registre $a1$ contient l'adresse de la chaîne de caractères.
— La chaîne de caractère se terminée par l'entier valant 0 dans le neuvième caractère.
— Vous utiliserez une variable permettant de compter les itérations.
— Vous utiliserez la fonction *forDigit*. Le paramètre de cette fonction étant stocké dans le même registre que celui de la fonction *toHexString* nous devons le sauvegarderons dans un registre.
— Vous sauvegarderez la valeur du registre ra car nous devons appeler la fonction *forDigit*.

Fonction *toHexString* ($a0$, $a1$) :

Question 10 : Algorithme orienté assembleur *parseHexInt* Modifier l'algorithme de la fonction *parseHexInt* sur les nombres non signés pour utiliser deux structures **SiCond** () **Aller à** .

Algorithme 64 Conversion d'une chaîne de caractères hexadécimal en un nombre non signé - enoncé

```
1: Fonction parseHexInt ( $a0 ) : $v0
2: Variables
3:     $t0 , $t1 , $t2 : entier
4: Début
5:     $t0 ← ... //Sauvegarde de l'adresse de la chaîne
6:     $t1 ← ... //Résultat partiel
7:     $t2 ← ... //Sauvegarder l'adresse de retour
8:    ... ← Mem8[...] //Charger le premier caractère de la chaîne
9:    SiCond (...) Aller à ... //Condition d'arrêt
10: parseHexIntBoucle :
11:    $v0 ←digit( $a0 )
12:    $t1 ← ... //Modifier le résultat
13:    $t1 ← ... //Modifier le résultat
14:    $t0 ← ... //Adresse du caractère suivant
15:    ... ← Mem8[...] //Charger le caractère suivant de la chaîne
16:    SiCond (...) Aller à .... //Condition d'itération
17: parseHexIntFin :
18:    $v0 ← ... //Affecter le résultat
19:    $ra ← ... //Restaurer l'adresse de retour
20: Retourner
21: Fin fonction
```

Indication 1 :

- Les codes ascii sont supposé valides c'est à dire représentant un digit hexadécimal.
- $a0 contient l'adresse de la chaîne.
- L'itération se termine lorsque la valeur chargée depuis la mémoire est 0.
- Vous utiliserez la fonction *digit*. Le paramètre et le résultat de cette fonction étant stockés dans les mêmes registre que ceux de la fonction *parseInt* nous devrons les sauvegarder dans des registres.

Fonction *parseHexInt* ($a0) : $v0

Correction exercice 7 :

Question 1 : Codage des instructions Mips

1. add *$t0* , *$a0* , *$v0* Les champs sont :Co =**0b**00 0000, *rs* =**0b**0 0100, *rt* =**0b**0 0010, *rd* =**0b**0 1000,**0b**0 0000,Nf =**0b**10 0000 et l'instruction est codée avec le nombre **0x**0082 4020.
2. sra *$t0* , *$v0* ,31 Les champs sont :Co =**0b**00 0000, *rs* =**0b**0 0000, *rt* =**0b**0 0010, *rd* =**0b**0 1000,shamt=**0b**1 1111,Nf =**0b**00 0011 et l'instruction est codée avec le nombre **0x**0002 47C3.
3. addi *$v0* , *$a0* ,-1 Les champs sont : Co =**0b**00 1000, *rs* =**0b**0 0100, *rt* =**0b**0 0010,*Imm16* =**0b**1111 1111 1111 1111 et l'instruction est codée **0x**2082 FFFF
4.
5. lw *$v0* ,4(*$a0*) Les champs sont : Co =**0b**10 0011, *rs* =**0b**0 0100, *rt* =**0b**0 0010, *Imm16* = **0b**0000 0000 0000 0100= **0x**4 . L'instruction est codée avec le nombre **0x**8C82 0004.
6.
7. sw *$v0* ,4(*$a0*) Les champs sont : Co =**0b**10 1011, *rs* =**0b**0 0100, *rt* =**0b**0 0010, *Imm16* = **0b**0000 0000 0000 0100= **0x**4 . L'instruction est codée avec le nombre **0x**AC82 0004.
8. bne *$a0* , *$v0* , instruction courante Les champs sont : Co =**0b**00 0101, *rs* =**0b**0 0100, *rt* =**0b**0 0010, *Imm16* = **0b**1111 1111 1111 1111= **0x**FFFF . L'instruction courante se trouve à -4 de l'adresse de l'instruction suivant bne . Décalée arithmétiquement de 2 rangs à droite, la valeur immédiate codée dans l'instruction sera donc -1. L'instruction est codée avec le nombre **0x**1482 FFFF.
9. bne *$a0* , *$v0* , deux instructions plus loin Les champs sont : Co =**0b**00 0101, *rs* =**0b**0 0100, *rt* =**0b**0 0010, *Imm16* = **0b**0000 0000 0000 0001= **0x**1 . L'instruction se trouve à 4 octets de l'adresse de l'instruction suivant bne . Décalée arithmétiquement de 2 rangs à droite, la valeur immédiate codée dans l'instruction sera donc 1. L'instruction est codée avec le nombre **0x**1482 0001.

Question 2 : Instruction(s) des pseudo-instructions Mips

1. **move** *$t1* , *$t0* : addi *$t1* , *$t0* ,0 ou **move** *$t1* , *$t0* : add *$t1* , *$t0* , *$zero*
2. **not** *$t1* , *$t0* : nor *$t1* , *$t0* , *$zero*
3. beq *$t1* ,*Imm16* ,étiquette n'existe pas car il n'y a pas la place de coder une valeur immédiate 16 bits et une étiquette 16 bits plus le code opération sur 32 bits. Le code de la pseudo-instruction est le suivant :

   addi *$1* , *$zero* ,*Imm16*
   beq *$1* , *$t1* ,étiquette

Question 3 : Analyse des pseudo-instructions

1. rol *$t1* , *$t0* ,1

$$\$t1 \quad = \quad ((\ \$t0 \ggg 31) \mid (\ \$t0 \ll 1)$$

2. ror *$t1* , *$0* ,1

$$\$t1 \quad = \quad ((\ \$t0 \ggg 1) \mid (\ \$t0 \ll 31)$$

3. abs $t1 , $0

$$\begin{aligned} \$t1 &= ((\ \$t0 >> 31) \hat{}\ \$t0\ ) - (\ \$t0 >> 31) \\ \$t1 &= (0 \hat{}\ \$t0\ ) - 0 = \$t0\ si\ \$t0 \geqslant 0) \\ \$t1 &= \sim \$t0\ - (-1) = -\ \$t0\ si\ \$t0 < 0. \end{aligned}$$

Question 4 : Algorithme orienté assembleur *forDigit*

Algorithme 65 Conversion d'un digit en un caractère optimisé

```
1: Fonction forDigit ( $a0 ) : $v0
2: Début
3:     $v0 ← $a0 +'0'
4:   SiCond (a0 ≤ 9) Aller à forDigitFinsi
5:     $v0 ← $v0 + 7 //car 'A'-'0'-10=7
6: forDigitFinsi :
7: Retourner
8: Fin fonction
```

Question 5 : Algorithme orienté assembleur *digit*

Algorithme 66 Conversion d'un caractère en un digit

```
1: Fonction digit ( $a0 ) : $v0
2:     $v0 ← $a0 - '0'
3:   SiCond ( $v0 ≤ 9) Aller à digitFin
4:     $v0 ← $v0 - 7 //car 'A'-'0'-10=7
5: digitFin :
6: Retourner
7: Fin fonction
```

Question 6 : Algorithme orienté assembleur *initMem*

Algorithme 67 Initialisation d'une zone mémoire

```
1: Fonction InitMem ( $a0 , $a1 , $a2 )
2: Début
3:   SiCond ( $a1 = 0) Aller à initMemFin
4: initMemBoucle :
5:   Mem32[ $a0 ]← $a2
6:     $a0 ← $a0 + 4
7:     $a1 ← $a1 - 1
8:   SiCond ( $a1 ≠ $zero ) Aller à initMemBoucle
9: initMemFin :
10: Retourner
11: Fin fonction
```

Question 7 : Algorithme orienté assembleur *copieMem*

Algorithme 68 Copie d'une zone mémoire

```
1: Fonction copieMem ( $a0 , $a1 , $a2 )
2: Variables
3:     $t0 : entier
4: Début
5: copieMemBoucle :
6:     $t0 ← Mem32[ $a0 ]
7:   Mem32[ $a1 ]← $t0
8:     $a0 ← $a0 + 4
9:     $a1 ← $a1 + 4
10:    $a2 ← $a2 - 1
11:   SiCond ( $a2 ≠ $zero ) Aller à copieMemBoucle
12: Retourner
13: Fin fonction
```

Question 8 : Algorithme orienté assembleur *copieChaine*

Algorithme 69 Copie d'une chaîne de caractères

```
1: Fonction copieChaine ( $a0 , $a1 )
2: Variables
3:     $t0 : entier
4: Début
5: copieChaine :
6:     $t0 ← Mem8[ $a0 ]
7:   Mem8[ $a1 ]← $t0
8:     $a0 ← $a0 + 1
9:     $a1 ← $a1 + 1
10:   SiCond ( $t0 ≠ $zero ) Aller à copieChaine
11: Retourner
12: Fin fonction
```

Question 9 : Algorithme orienté assembleur *toHexString*

Algorithme 70 Conversion d'un nombre non signé en une chaîne de caractères hexadécimale

```
1:  Fonction toHexString ( $a0 , $a1 ) :
2:  Variables
3:      $t0 , $t1 , $t2 : entier
4:  Début
5:      $t0 ← 8 //Compteur d'itération
6:      $t1 ← $a0 //Sauvegarde du nombre à convertir
7:      $t2 ← $ra //Sauvegarder de l'adresse de retour
8:  toHexStringBoucle :
9:      $a0 ← $t1 >>> 28 //Octet de gauche
10:     $v0 ← forDigit( $a0 )
11:    Mem8[ $a1 ]← $v0 //Ranger le caractère dans la chaîne
12:     $a1 ← $a1 +1 //Modifier l'adresse pour le caractère suivant
13:     $t1 ← $t1 << 4 //Modifier la valeur du nombre à convertir pour la prochaine itération

14:     $t0 ← $t0 -1 //Modifier le compteur de boucle
15:    SiCond ( $t0 ≠ $zero ) Aller à toHexStringBoucle //Condition d'arrêt
16:    Mem8[ $a1 ]← $zero //Mettre le caractère de fin de chaîne
17:     $ra ← $t2 //Restaurer l'adresse de retour
18: Retourner
19:    Fin fonction
```

Question 10 : Algorithme orienté assembleur *parseHexInt*

Algorithme 71 Conversion d'une chaîne de caractères hexadécimaux en un nombre non signé

```
1:  Fonction parseHexInt ( $a0 ) : $v0
2:  Variables
3:      $t0 , $t1 , $t2 : entier
4:  Début
5:      $t0 ← $a0
6:      $t1 ← 0
7:      $t2 ← $ra
8:  parseHexIntBoucle :
9:      $a0 ← Mem8[ $t0 ]
10:    SiCond ( $a0 = $zero ) Aller à parseHexIntFin
11:     $v0 ←digit( $a0 )
12:     $t1 ← $t1 << 4
13:     $t1 ← $t1 | $v0
14:     $t0 ← $t0 +1
15:    SiCond ( $zero = $zero ) Aller à parseHexIntBoucle
16: parseHexIntFin :
17:     $v0 ← $t1
18:     $ra ← $t2
19: Retourner
20: Fin fonction
```

Nous pouvons optimiser le nombre de branchement par itérations :

Algorithme 72 Conversion d'une chaîne de caractères hexadécimal en un nombre non signé

```
1: Fonction parseHexInt ( $a0 ) : $v0
2: Variables
3:     $t0 , $t1 , $t2 : entier
4: Début
5:     $t0 ← $a0 //Sauvegarde de l'adresse de la chaîne
6:     $t1 ← 0 //Résultat partiel
7:     $t2 ← $ra //Sauvegarder l'adresse de retour
8:     $a0 ← Mem8[ $t0 ] //Charger le premier caractère de la chaîne
9:   SiCond ( $a0 = $zero ) Aller à parseHexIntFin //Condition d'arrêt
10: parseHexIntBoucle :
11:     $v0 ←digit( $a0 )
12:     $t1 ← $t1 <<4 //Modifier le résultat
13:     $t1 ← $t1 | $v0 //Modifier le résultat
14:     $t0 ← $t0 +1 //Adresse du caractère suivant
15:     $a0 ← Mem8[ $t0 ] //Charger le caractère suivant de la chaîne
16:   SiCond ( $a0 ≠ $zero ) Aller à parseHexIntBoucle //Condition d'itération
17: parseHexIntFin :
18:     $v0 ← $t1 //Affecter le résultat
19:     $ra ← $t2 //Restaurer l'adresse de retour
20: Retourner
21: Fin fonction
```

Exercice 8 : Exercice pratique : programmation Assembleur

Pour réaliser vous devez télécharger et installer la dernière version du simulateur MARS (Mips Architecture Runtime Simulator) `http://courses.missouristate.edu/KenVollmar/MARS/`. La documentation de la partie Help de ce simulateur est très complète. Vous y trouverez les explications concernant le fonctionnement de l'IDE, les instructions et pseudo-instructions, les appels systèmes.

Question 1 : Prise en main Testez le codage des instructions des exercices théoriques, le codage des pseudo-instructions des exercices théoriques et les exemples du cours.

Question 2 : Fonctions d'accès à la mémoire Écrivez et testez les fonctions suivantes (étudiées dans l'exercice précédent).

```
# $a0  adresse zone mémoire
# $a1  taille de le zone mémoire
# $a2  valeur à placer en mémoire
initMem :
# $a0  adresse zone mémoire src
# $a1  adresse zone mémoire dest
# $a2  taille de le zone mémoire
copieMem :
# $a0  adresse chaine source
# $a1  adresse chaine destination
# $v0  retourne le nombre de caractères copiés
CopieChaine
```

Remarques :

- Vous utiliserez les instructions de transfert d'octets lb et sb pour la fonction *CopieChaine*
- Les résultats ne sont pas affichés par le programme. Vous observerez les résultats dans le data segment à partir de l'adresse **0x**1001 0000.

Question 3 : Fonctions sur les nombres non signés

```
# $a0  digit à convertir
# $v0  retourne le caractère
digit :
# $a0  caractère à convertir
# $v0  retourne le digit correspondant digit ou -1
forDigit :
# $a0  entier à convertir
# $a1  adresse de la chaine résultat
toHexString :
# $a0  adresse de la chaine à convertir
# $v0  retourne l'entier résultat
parseHexInt :
```

Question 4 : Et pour finir Donnez et testez le code des deux fonctions nommées *bitCount* et *fastBitCount*.

```
# $a0  nombre de départ
# $v0  résultat
```

```
bitCount :
#  $a0  nombre de départ
#  $v0  résultat
fastBitCount :
```

Correction exercice 8 :

Listing 6.6 – Manipulation zone mémoire

```
1  .data
2    source : .space 32
3    destination : .space 32
4    ChaineSource: .asciiz "Hello world!"
5    ChaineDest:   .space 16
6
7
8  .text
9    #Test InitMem
10   la $a0,source
11   addi $a1,$zero,8
12   addi $a2,$zero,0xFF00FF00
13   jal InitMem
14   #Test CopieMem
15   la $a0,source
16   la $a1,destination
17   addi $a2,$zero,8
18   jal CopieMem
19   #Test Copie chaine de caracteres
20   la $a0, ChaineSource
21   la $a1, ChaineDest
22   jal CopieChaine
23   # -- exit
24   addi $v0, $zero, 10
25   syscall
26
27 # Fonction Initialisation zone memoire
28 # $a0 adresse zone memoire
29 # $a1 taille de le zone memoire
30 # $a2 valeur a place en memoire
31 InitMem:
32   beq $a1,$zero,InitMemFin
33   sw $a2,0($a0)
34   addi $a1,$a1,-1
35   addi $a0,$a0,4
36   j InitMem
37 InitMemFin:
38   jr $ra
39
40 # Fonction Copie zone memoire
41 # $a0 adresse zone memoire src
42 # $a1 adresse zone memoire dest
43 # $a2 >0 taille de le zone memoire
44 CopieMem:
45   lw $t0,0($a0)
46   sw $t0,0($a1)
47   addi $a2,$a2,-1
48   addi $a0,$a0,4
49   addi $a1,$a1,4
50   bne $a2,$zero,CopieMem
51 CopieMem_Fin:
52   jr $ra
53
54 # Fonction Copie chaine version
55 # $a0 adresse chaine source
56 # $a1 adresse chaine destination
```

```
57 # retourne le nombre de caracteres copies en $v0
58 CopieChaine:
59   #valeur retournee par la fonction
60   addi $v0, $zero, -1
61 CopieChaineBoucle:
62   lb $t0, 0($a0)
63   sb $t0, 0($a1)
64   addi $a0, $a0, 1
65   addi $a1, $a1, 1
66   addi $v0, $v0, 1
67   bne $t0,$zero, CopieChaineBoucle
68   jr $ra
```

Listing 6.7 – Conversion hexadécimale non signé

```
1  .data
2    #sauvergarder la chaine resultat 8 caract?res maximum + 0
3    chaine : .space 9
4  .text
5    #Test de la fonction toHexString
6    addi $a0,$zero,0x1234ABCD
7    la   $a1,chaine
8    jal toHexString
9    #Test de la fonction parseHexInt
10   la   $a0,chaine
11   jal parseHexInt
12   # Affichage du resultat des deux conversions successives.
13   add $a0,$zero,$v0
14   addi $v0,$zero,34
15   syscall
16   #-- exit
17   addi $v0,$zero,10
18   syscall
19
20 #a0 digit a convertir
21 #v0 retourner le caractere
22 forDigit:
23   addi $v0,$a0,'0'
24   ble $a0,9,forDigitChiffre
25   addi $v0,$v0,7
26 forDigitChiffre:
27   jr $ra
28
29 #$a0 caractere valide a convertir
30 #v0 retourne le digit
31 digit:
32   sub $v0,$a0,'0'
33   ble $v0,9,digitFin
34   sub $v0,$v0,7
35 digitFin:
36   jr $ra
37
38 #$a0 entier ? convertir
39 #a1 adresse chaine resultat
40 toHexString:
41   # garder la valeur de a0 car utilise par fordigit argument de forDigit
42   add $t0,$zero,$a0
43   #Compteur de boucle
```

```
44    addi $t1,$zero,8
45    #sauvegarde registre d'adresse de retour
46    add $t2,$ra,$zero
47  toHexStringBoucle:
48    #mettre le digit de poids fort dans celui de poids faible dans a0
49    add $a0,$t0,$zero
50    srl $a0,$a0,28
51    jal forDigit
52    sb $v0,0($a1)
53    sll $t0,$t0,4
54    addi $a1,$a1,1
55    addi $t1,$t1,-1
56    bne $t1,$zero,toHexStringBoucle
57    #Caract?re de fin de chaine
58    sb $zero,0($a1)
59    #restitution adresse de retour
60    add $ra,$t2,$zero
61    jr $ra
62
63  #a0 adresse chaine a convertir
64  #v0 retourne entier
65  parseHexInt:
66    #sauvegarde registre d'adresse de retour
67    add $t2,$ra,$zero
68    #a0  param de digit - adresse en $t0
69    add $t0,$a0,$zero
70    #vO resultat de digit - resultat en $t1
71    add $t1,$zero,$zero
72  parseHexIntBoucle:
73    lb $a0,0($t0)
74    beq $a0,$zero,parseHexIntFin
75    jal digit
76    sll $t1,$t1,4
77    or $t1,$t1,$v0
78    addi $t0,$t0,1
79    j  parseHexIntBoucle
80  parseHexIntFin:
81    addi $v0,$t1,0
82    #restitution adresse de retour
83    add $ra,$t2,$zero
84    jr $ra
```

Listing 6.8 – Comptage des bits d'une donnée

```
1   .data
2   .text
3     addi $a0,$zero,0x7F000001
4     jal fastBitCount
5     addi $v0,$zero,10
6     syscall
7
8   #parametre a0
9   #resultat v0
10  bitCount:
11    addi $v0,$zero,0
12  bitCount_boucle:
13    beqz $a0,bitCount_Fin
14    andi $t0,$a0,1
```

```
15    add  $v0,$v0,$t0
16    srl  $a0,$a0,1
17    j bitCount_boucle
18 bitCount_Fin:
19    jr $ra
20
21 #parametre a0
22 #resultat v0
23 fastBitCount:
24    andi $v0,$a0,0x55555555
25    srl  $a0,$a0,1
26    andi $a0,$a0,0x55555555
27    add $a0,$v0,$a0
28
29    andi $v0,$a0,0x33333333
30    srl  $a0,$a0,2
31    andi $a0,$a0,0x33333333
32    add $a0,$v0,$a0
33
34    andi $v0,$a0,0x0F0F0F0F
35    srl  $a0,$a0,4
36    andi $a0,$a0,0x0F0F0F0F
37    add $a0,$v0,$a0
38
39    andi $v0,$a0,0x00FF00FF
40    srl  $a0,$a0,8
41    andi $a0,$a0,0x00FF00FF
42    add $a0,$v0,$a0
43
44    andi $v0,$a0,0x0FFFF
45    srl  $a0,$a0,16
46    add $v0,$v0,$a0
47    jr $ra
```

Chapitre 7

Architecture des processeurs MIPS

7.1 Introduction

Dans ce chapitre, nous allons concevoir l'implémentation matérielle d'une architecture permettant de traiter le minimum d'instructions nécessaires à tous les programmes. Ce minimum contient les instructions suivantes :

Instructions de chargement rangement : lw et sw .

Instructions arithmétiques et logiques : add , and , lui , nor , or , sll ,sra ,srl ,sub , xor ,addi ,andi ,ori ,xori .

Instruction de branchement : instruction bne .

Pour traiter une instruction, le microprocesseur doit effectuer les opérations suivantes :

1. Envoyer la valeur d'un registre nommé *compteur de programme* ou *cp* qui contient l'adresse de l'instruction à la mémoire et ensuite calculer la position de l'instruction suivante qui se trouve quatre octet plus loin (codage des instructions sur 32 bits)
2. Charger depuis la mémoire contenant les instructions la valeur de l'entier 32 bits correspondant à l'instruction à traiter.
3. Lire la valeur de un (*rs*) ou deux (*rs* et *rt*) registres opérandes.
4. Ensuite en fonction de la nature de l'instruction :
   - **Instruction arithmétiques et logiques à 3 registres** : Calculer le résultat de l'opération à partir des valeurs contenues dans les registres *rs* et *rt* et ranger le résultat dans le registre destination *rd* .
   - **Instruction arithmétiques et logiques à 2 registres** : Calculer le résultat de l'opération à partir des valeurs contenues dans le registre *rs* et la valeur immédiate *Imm16* *rt* et ranger le résultat dans le registre destination *rt* .
   - **Instruction chargement/rangement** : Calculer l'adresse de la donnée en mémoire (opération d'addition de *rs* et la valeur immédiate *Imm16*)
     - **Instruction de rangement** : écrire le contenu du registre *rt* à l'adresse calculée ci-dessus
     - **Instruction de chargement** : lire le contenu de la mémoire dont l'adresse est calculée ci-dessus puis ranger le résultat dans le registre destination *rt*
   - **Instruction de branchement** : Calculer l'adresse de l'instruction cible (opération d'addition de *rs* et la valeur (*Imm16* $<<$ 2). Tester l'égalité des registres *rs* et *rt* , puis sélectionner l'adresse de l'instruction suivante en fonction du résultat du test.

7.2 Éléments de base du chemin de données

7.2.1 Unité Arithmétique et Logique (UAL)

L'unité arithmétique et logique permet d'effectuer les opérations arithmétiques et logiques bit à bit sur les données. En fonction du signal nommé *op* permettant de sélectionner une opération parmi un ensemble possible. Le circuit détermine le résultat d'une opération arithmétique ou logique. Le circuit présenté sur la figure Fig 7.2 réalise les opérations figurant sur la figure Fig 7.1 .

Op	Opération
0	$opérande_1 + opérande_2$
1	$opérande_1$ & $opérande_2$
...	...
4	$opérande_2 << shamt$
...	...
9	$opérande_1 \ \hat{} \ opérande_2$

Figure 7.1 – Opérations UAL

Tous les résultats sont calculés en même temps. Nous utiliserons un multiplexeur pour sélectionner un résultat parmi ceux disponibles. La figure Fig 7.2 représente le schéma interne d'une UAL effectuant une partie des opérations nécessaires pour notre architecture. Les signaux $opérande_1$, $opérande_2$ et *shamt* sont les entrées, *numéro d'opération UAL* est la commande *op*, et *résultat UAL* le résultat de l'unité arithmétique et logique.

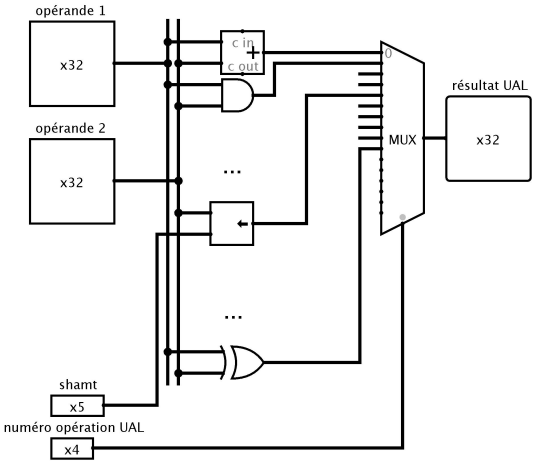


Figure 7.2 – Schéma interne simplifié de l'UAL

7.2.2 Banc de registres - noté REG

Le banc de registre contient les données et les résultats d'une instruction. Pour une instruction, nous avons besoin de deux valeurs contenues dans les registres et nous devons stocker un résultat dans un registre. Nous avons donc besoin de lire le contenu de deux registre et d'en écrire un. L'architecture contient 32 registres spécifiés dans la partie concernant l'architecture matérielle. Un registre est un circuit qui stocke la valeur codée sur n bit présente sur l'entrée nommé D sur la sortie Q en fonction des entrées suivantes :

— **Front montant et front descendant** : Lorsque le signal d'horloge (noté avec le symbole $\wedge$) passe de l'état 0 à 1 (propriété Trigger à la valeur rising edge dans logisim) ou de 1 à

0 (propriété Trigger à la valeur falling edge dans logisim) et que le signal validation noté *en* (enable) est à 1. Si ce signal *en*= 0 la sortie ne change pas.

— Un signal de remise à 0 noté 0 permet de mettre la sortie à 0, quelque soient l'état des autres entrées.

Le banc de registre possède les entrées et sorties suivantes :

— *num rs* contient le numéro du premier registre à lire et *rs* la valeur contenue dans ce registre.

— *num rt* contient le numéro du deuxième registre à lire et *rt* la valeur contenue dans ce registre.

— *num rd/rt* contient le numéro du registre à écrire et *val ec* la valeur à écrire dans ce registre si l'instruction le nécessite.

La figure Fig 7.3 présente une vue externe du banc de registres incluant les signaux définis ci-dessus.

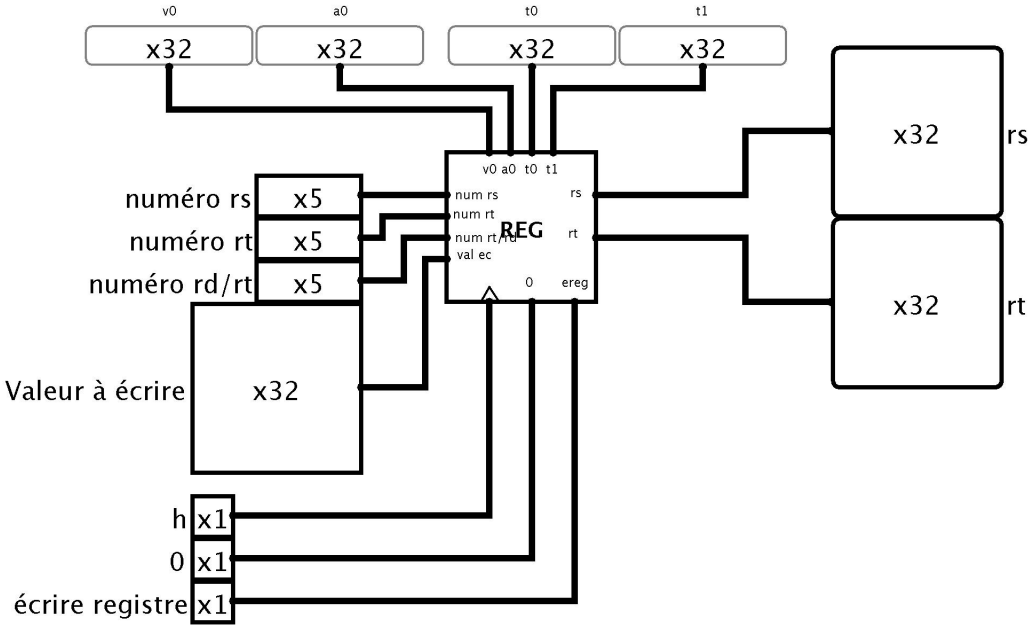


Figure 7.3 – Schéma externe du banc de registres

L'intérieur du banc de registres est composé de 32 registres 32 bits, d'un décodeur 5 bits qui permet de sélectionner le registre à écrire et deux multiplexeurs 5 bits qui permettent de placer sur les sorties les valeurs des deux registres lus.

Les entrées D des 32 registres sont connectées à la valeur à écrire dans le registre *val ec*. Elle sera écrite uniquement dans le registre dont le numéro est placé en *num rd/rt*, à l'aide d'un décodeur qui sélectionnera uniquement l'entrée de validation correspondante. La valeur sera écrite dans le registre seulement si le signal $ereg = 1$ (certaines instructions ne nécessite pas d'écriture dans le registre).

Les sorties des registres sont envoyées à deux multiplexeurs, qui à partir des numéros des registres *num rs* et *num rt*, placent sur *rs* et *rt* les sorties des registres correspondants. Le schéma partiel de la figure Fig 7.4 représente l'intérieur du banc de registres.

Les sorties *$v0* , *$a0* , *$t0* , *$t1* permettent de visualiser à l'extérieur le contenu de ces registres.

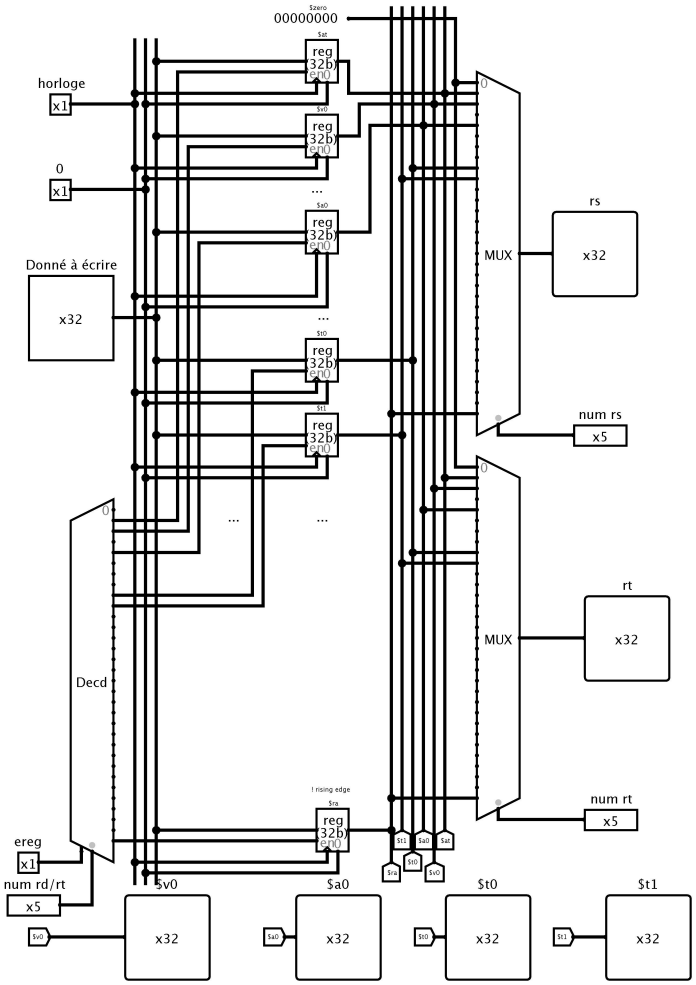


FIGURE 7.4 – Schéma interne du banc de registres

7.2.3 Mémoire contenant les instructions du programme

Vous avez peut être remarqué que dans le simulateur Mars, le programme démarre à l'adresse **0x**0040 0000. Afin de simplifier l'architecture, nous utiliserons une mémoire de 64 kilo-octets (16 kilo-mot de 32 bits). Les instructions étant codées sur 32 bits (4 octets) nous utiliserons uniquement les adresses divisibles par 4 (c'est à dire avec les deux bits de poids faibles à 0). Nous utiliserons donc 14 bits d'adresse comme le suggère le schéma de la mémoire d'instruction présenté sur la figure FIG 7.5 . L'adresse mémoire est sur 14 bits et les données sur 32 bits. Cette mémoire est du type lecture seule (ROM - read only memory) et peut être seulement lue.

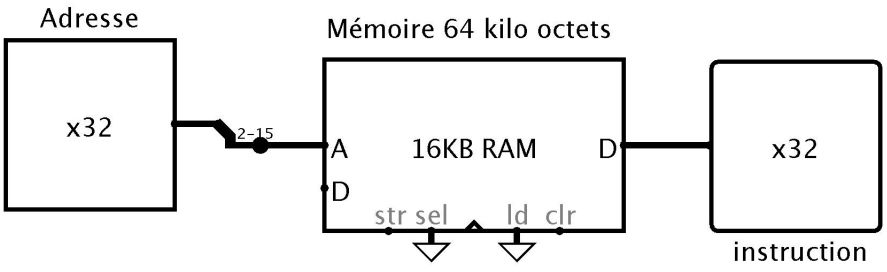


FIGURE 7.5 – Schéma mémoire d'instructions

7.2.4 Mémoire contenant les données du programme

Les données du programme démarre à l'adresse **0x**1001 0000. Afin de simplifier l'architecture, nous utiliserons une mémoire de 64 kilo-octets. Les données étant codées sur 32 bits (4 octets) nous utiliserons uniquement les adresses divisibles par 4 (c'est à dire avec les deux bits de poids faibles à 0). Nous utiliserons donc 14 bits d'adresse comme le suggère le schéma de la mémoire d'instruction présenté sur la figure Fig 7.6 . L'adresse mémoire est sur 14 bits et les données sur 32 bits. Nous ne pourrons pas accéder aux données sous forme d'octet car notre mémoire contient des informations 32 bits. Le signal *ecriture mémoire* permet d'enregistrer une donnée en mémoire.

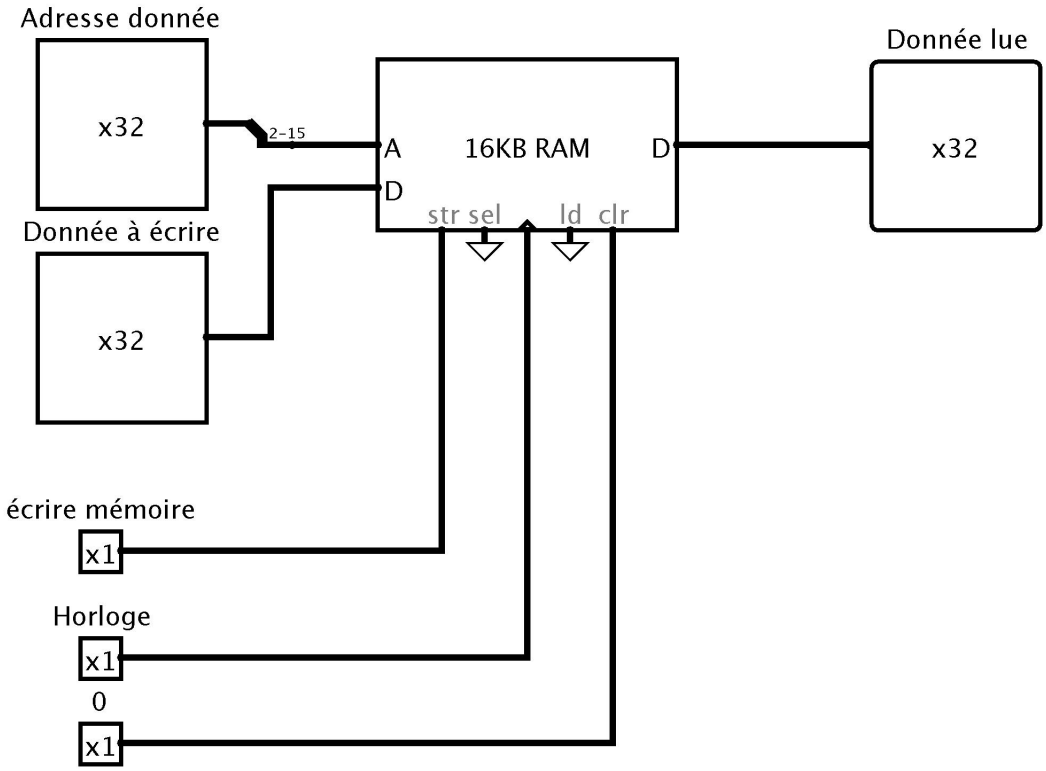


Figure 7.6 – Schéma mémoire de donnée

7.2.5 Compteur de programme

Le compteur de programme est sur 32 bits (même si nous n'en utiliserons que 14) et sensible au front descendant. Il contient l'adresse de l'instruction en cours de traitement. Le changement de l'adresse de l'instruction en cours de traitement se fera lors du passage de 1 à 0 du signal d'horloge (front descendant). La figure Fig 7.7 présente le schéma interne du registre compteur de programme. Nous remarquons que les 16 bits de poids fort sont à **0x**0040 et ne peuvent pas être modifiés.

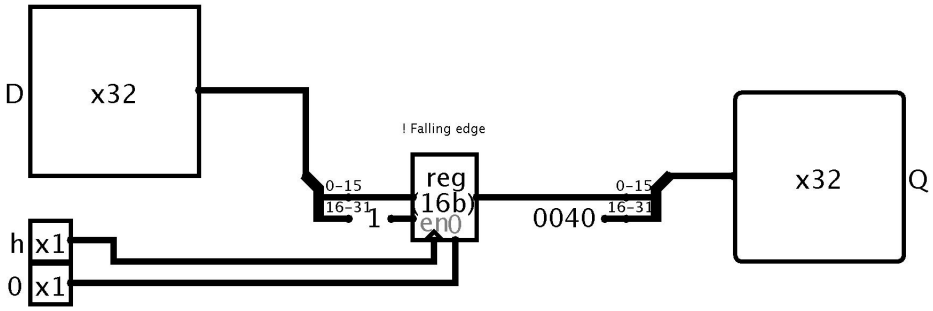


FIGURE 7.7 – Schéma compteur de programme

7.3 Construction du chemin de données

A partir des éléments définis précédemment nous allons construire le schéma de l'architecture MIPS. Nous placerons dans le chemin de données les instructions suivantes :

1. Instructions à trois registres
2. Instructions à deux registre et une valeur immédiate
3. Instructions de chargement et rangement
4. Instruction de branchement

A chaque étape, le circuit deviendra un peu plus complexe.

Pour toutes les instructions, le registre compteur de programme *cp* contient l'adresse de l'instruction mémoire à traiter. La mémoire d'instruction est accédée et nous calculerons *cp* pour qu'il pointe sur la prochaine instruction. La mémoire contenant des informations codées sur 8 bits et l'instruction étant elle même codée sur 32 bits, le registre *cp* doit être augmenté de 4 pour passer à l'instruction suivante.

7.3.1 Instructions arithmétiques et logiques à trois registres

Les instructions à trois registres utilisent le banc de registres et l'UAL. Les champs correspondants aux numéros des registres *rs* et *rt* sont placés sur les entrées du banc de registre *num rs* et *num rt*. A la sortie de ceux-ci, les valeurs de *rs* et *rt* contenues dans les registres sont placées sur les entrées de l'UAL. Celle ci effectue l'opération codée dans les champs Co et Nf de l'instruction. Le résultat de l'UAL est placé dans le registre *rd* dont le numéro est celui du registre *num rd/rt*. Le schéma de la figure FIG 7.8 représente le chemin de données pour les instructions UAL à trois registres. L'instruction dans la mémoire **0x**0040 0000 à pour valeur **0x**0082 4020. C'est l'instruction add *$t0* , *$a0* , *$v0* . Observer les valeurs des numéros des registres(2 pour *$v0* , 4 pour *$a0* et 8 pour *$t0*), des entrées de l'UAL (valeurs 1 et 2) et de la sortie de l'UAL(valeur 3). Le code opération Co est à 0 et le numéro de fonction Nf à **0x**20, ce qui correspond bien à une addition.

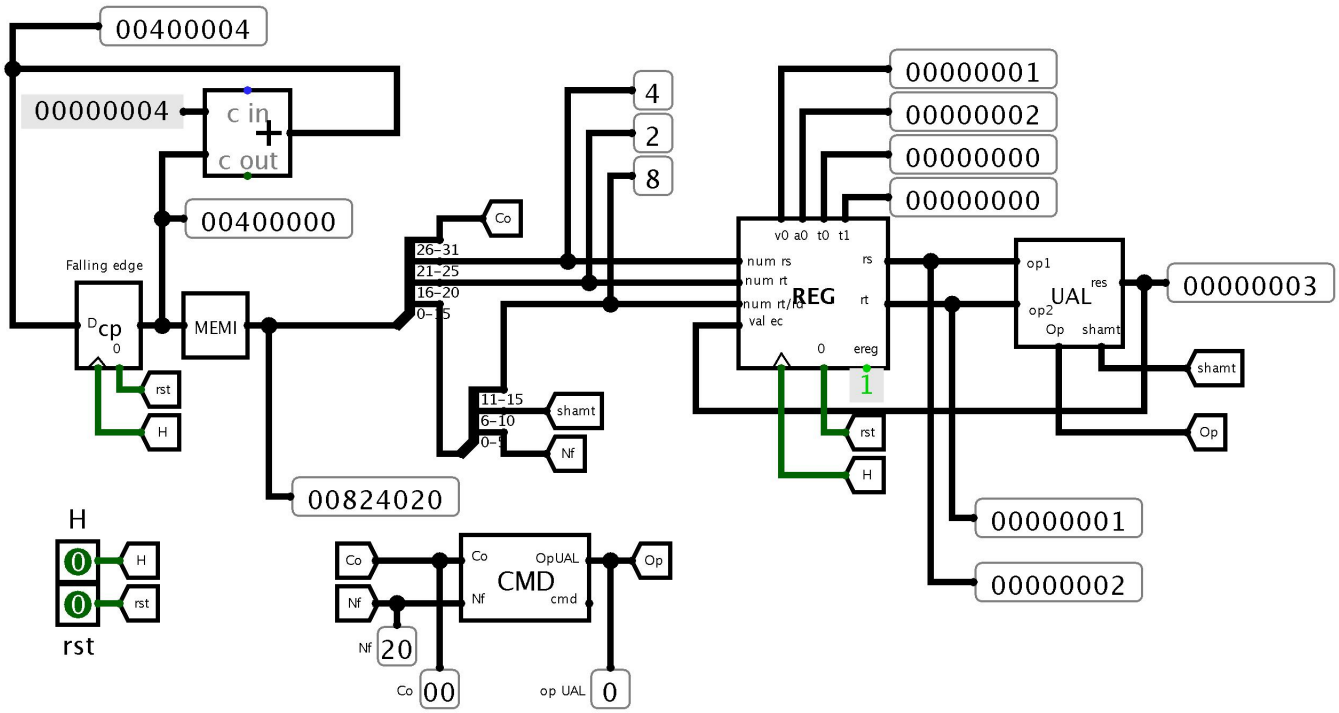


FIGURE 7.8 – Chemin de données instructions à trois registres

Le schéma de la figure FIG 7.9 représente l'écriture du résultat dans le banc de registre (registre *$t0*) au front montant du signal d'horloge. La valeur 3 est écrite dans le registre *$t0* qui est le numéro du registre *rd* .

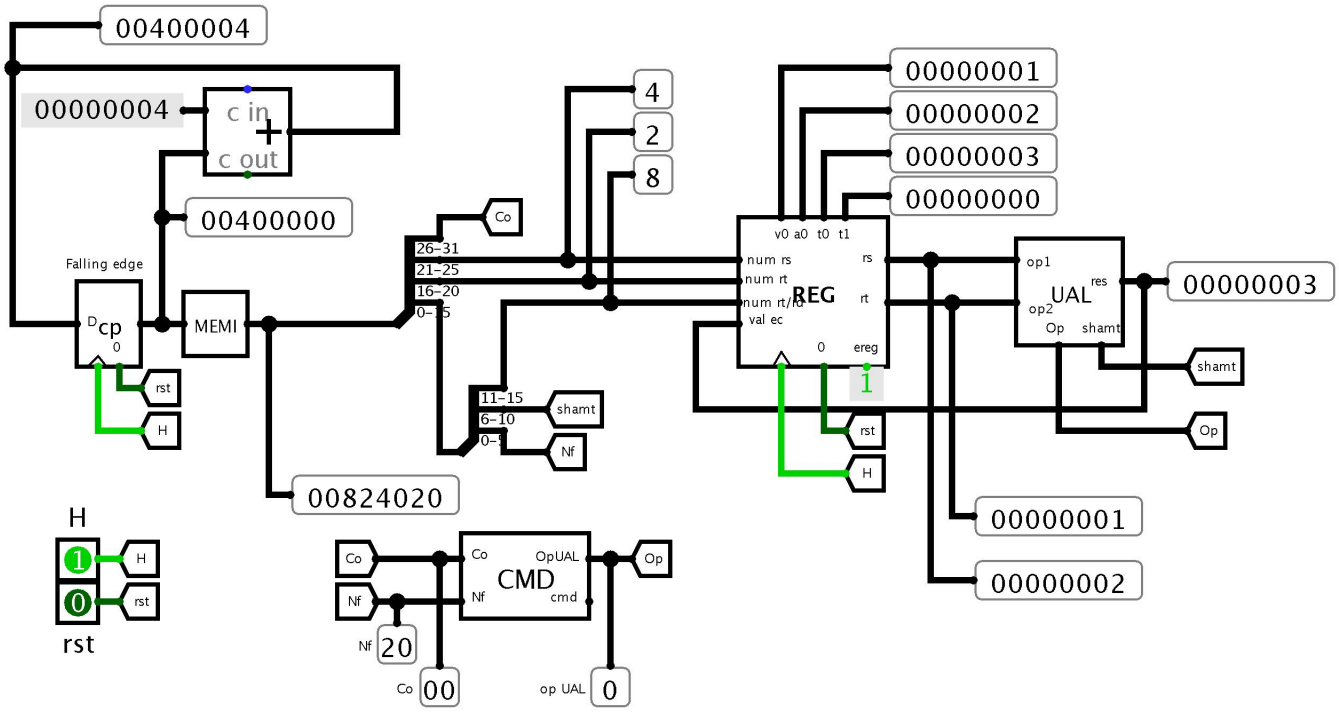


FIGURE 7.9 – Chemin de données instructions à trois registres

Remarque 39 :

— Le signal *ereg*=1 car nous devons enregistrer le résultat de l'instruction dans le banc de registre

7.3.2 Instructions arithmétiques et logiques utilisant deux registres et une valeur immédiate

Dans ce cas, la valeur immédiate est placée sur la deuxième entrée de l'UAL à la place de la valeur du registre *rt* . Le résultat est placé dans le registre dont le numéro est celui du registre *rt* . Pour combiner avec les instructions à trois registres, nous ajoutons un multiplexeur permettant de sélectionner la valeur contenue dans le registre *rt* ou la valeur immédiate *Imm16* . Nous ajoutons aussi un multiplexeur permettant de sélectionner le numéro du registre résultat *num rt* ou *num rd*. La valeur immédiate signée 16 bits sera transformée en un valeur immédiate signée 32 bits (signée pour les opérations arithmétiques, non signée pour les opération logiques). Le schéma de la figure Fig 7.10 représente le chemin de données pour les instructions à trois registres ou à deux registres et une valeur immédiate. L'instruction dans la mémoire **0x**0040 0000 à pour valeur **0x**2082 ffff. C'est l'instruction addi *$v0* , *$a0* ,-1. Observer les valeurs des numéros des registres, des entrées de l'UAL (2 et -1) et de la sortie de l'UAL (1). Nous utiliserons une extension de la valeur immédiate signée.

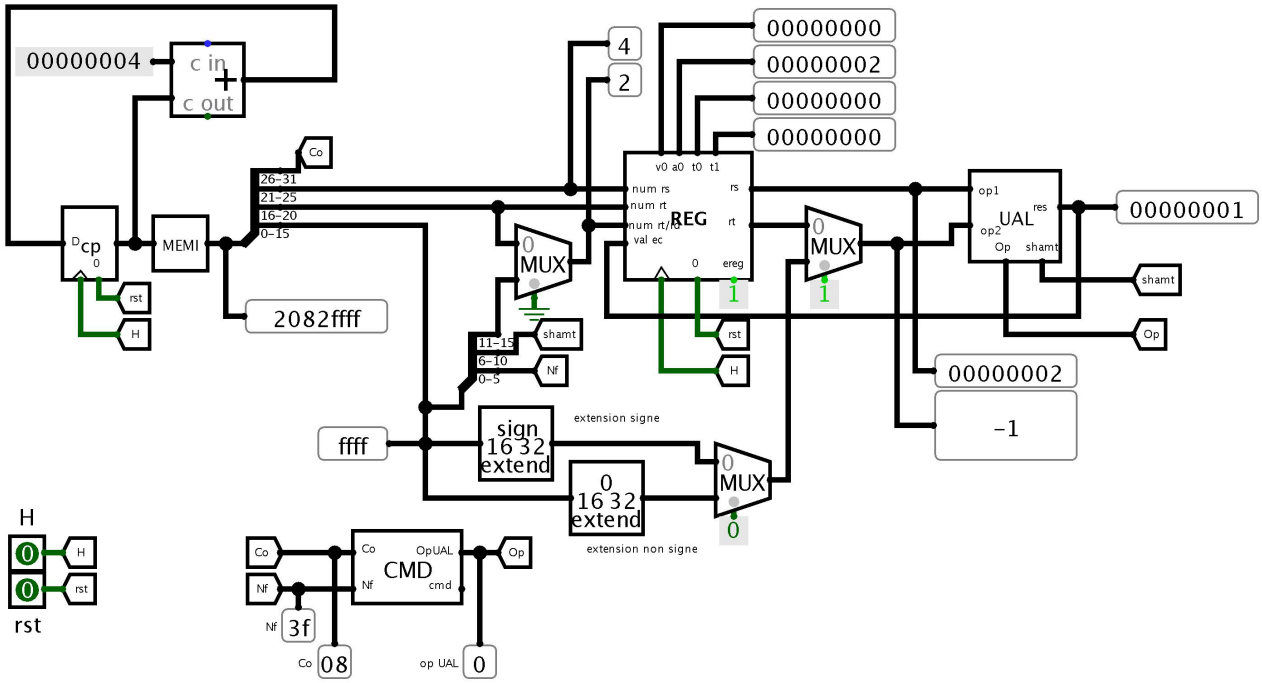


Figure 7.10 – Chemin de données pour les opérations à deux registres et une valeur immédiate

Remarque 40 :

- Le signal *ereg*=1 car nous devons enregistrer le résultat de l'instruction dans le banc de registre
- Le signal de commande du multiplexeur placé avant le banc de registre sera nommé *rtrd*. Ici, cas il sera à 0 pour sélectionner le numéro du registre résultat, a savoir le registre *rt* . (*rtrd* signifie que si la commande est à 0, le registre écrit sera *rt* , *rd* sinon)
- Le signal de commande du multiplexeur placé après les extension de signe sera nommé *arithLog*. Ici, il sera positionné à 0 car nous souhaitons une extension signée de la valeur immédiate sur 32 bits.
- Le signal de commande du multiplexeur placé sur l'entrée basse de l'UAL sera nommé *rtImm*. Ici, il sera à 1 pour sélectionner la valeur immédiate 32 bits.
- Pour ces instructions Nf n'est pas utilisé, mais il a la valeur de 6 bits de poids faible de la valeur immédiate.

7.3.3 Instructions de chargement et de rangement

Les instructions lw et sw accèdent à la mémoire de données du système. Dans ce cas, l'UAL est utilisée pour calculer l'adresse de la mémoire de donnée qui sera lue ou écrite en ajoutant la valeur immédiate *Imm16* à la valeur du registre *rs* (c'est à dire une opération d'addition avec *Op*=0)
La donnée contenue dans le registre *rt* sera écrite en mémoire ou celle lue dans la mémoire sera écrite dans le registre *rt* . La mémoire peut être vue comme un grand ensemble de registres. L'entrée de la mémoire *donnée* à *écrire* reçoit les valeurs à écrire dans la mémoire en provenance du registre *rt* pour un rangement. La sortie de la mémoire *donnée lue* reçoit la valeur lue en mémoire, qui sera écrite dans le registres *rt* lors d'un chargement.
Lorsque nous combinons les opérations UAL et les opérations de chargement/rangement nous devons ajouter un multiplexeur permettant de sélectionner la donnée écrite dans le banc de registres : soit la valeur lue en mémoire, soit la valeur calculée par l'UAL. Le schéma de la figure Fig 7.11 représente le chemin de données pour les instructions lw accédant à la mémoire. L'instruction dans la mémoire **0x**0040 0000 à pour valeur **0x**8c82 0004. C'est l'instruction lw *$v0* ,4(*$a0*). Observer les valeurs des numéros des registres, des entrées de l'UAL et de la sortie de l'UAL ainsi que la valeur lue en mémoire qui sera écrite dans le banc de registres

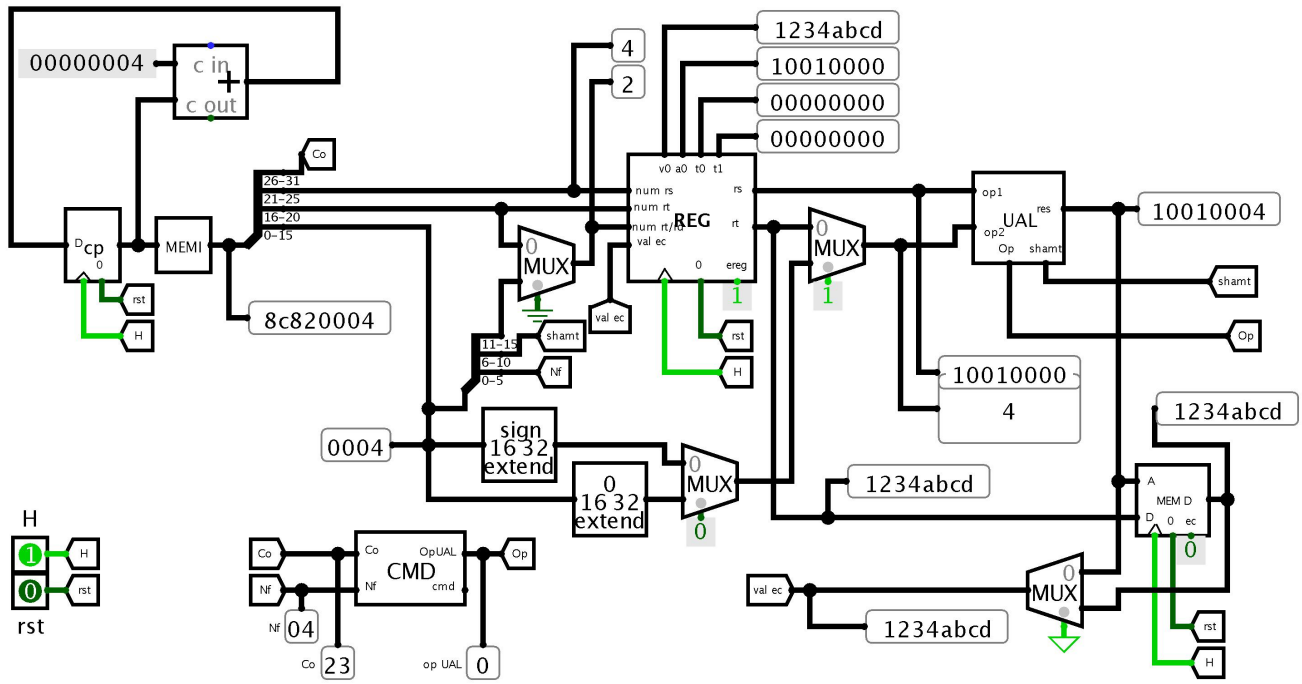


Figure 7.11 – Chemin de données pour une instruction de chargement

Remarque 41 :

- Le signal *ereg*=1 car nous devons enregistrer le résultat de l'instruction dans le banc de registre.
- Le signal *ec*=0 car nous ne devons pas modifier la valeur contenue dans la mémoire de données. Si signal *ec* = 1 l'écriture de la valeur présente sur les donnée est effectuée.
- Le signal de commande du multiplexeur placé avant le banc de registre sera nommé *rtrd*. Dans ce cas il sera à 0 pour sélectionner le numéro du registre résultat, a savoir le registre *rt* . (*rtrd* signifie que si la commande est à 0, le registre écrit sera *rt* , *rd* sinon)
- Le signal de commande du multiplexeur placé à l'entrée sera nommé *rtImm*. Dans ce cas il sera à 1 pour sélectionner la valeur immédiate *Imm16* transformée en un valeur 32 bits.

— Le signal de commande du multiplexeur placé à la sortie de la mémoire sera nommé *ualmem*. Dans ce cas il sera à 1 pour sélectionner la valeur lue en mémoire.

Le schéma de la figure FIG 7.12 représente le chemin de données pour les instructions sw accédant à la mémoire. L'instruction dans la mémoire **0x**0040 0000 à pour valeur **0x**ac82 0008. C'est l'instruction sw *$v0* ,8(*$a0*). Observer les valeurs des numéros des registres, des entrées de l'UAL et de la sortie de l'UAL ainsi que l'adresse mémoire.

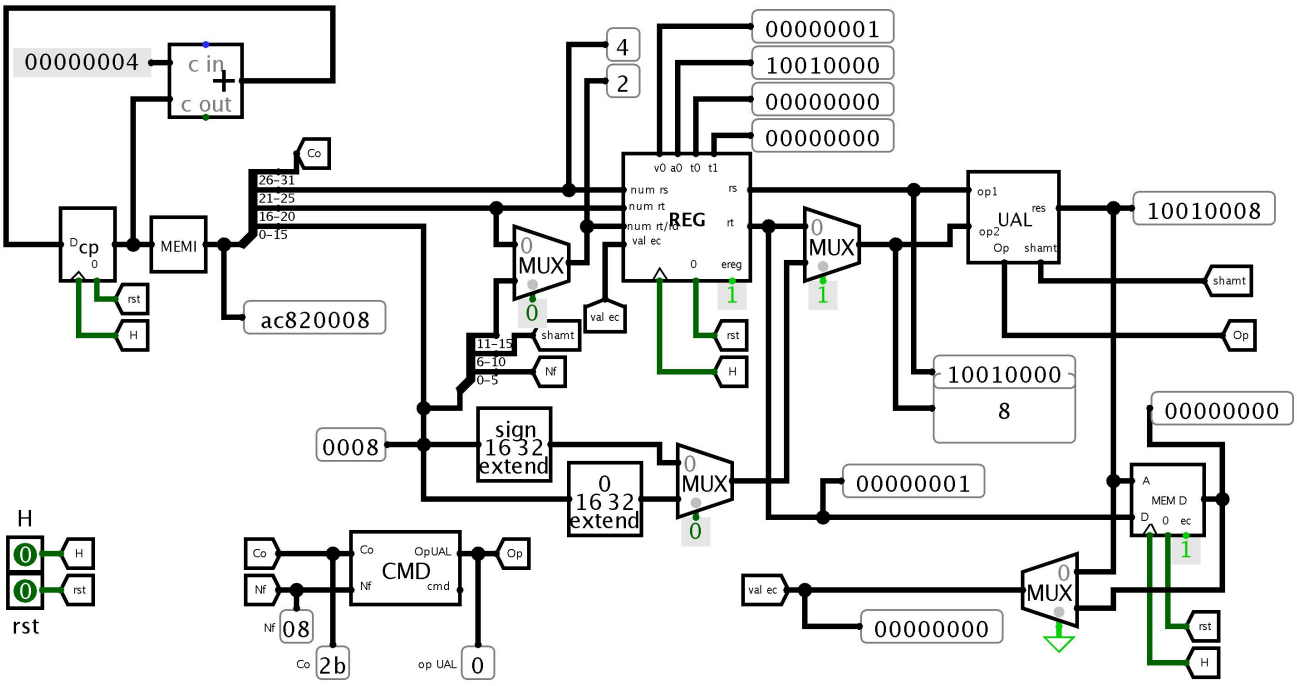


FIGURE 7.12 – Chemin de données pour une instruction de rangement

Remarque 42 :

— Le signal *ereg*=0 car nous ne devons pas enregistrer le résultat de l'instruction dans le banc de registre.

— Le signal *ec*=1 car nous devons modifier la valeur contenue dans la mémoire de données.

— Le signal de commande du multiplexeur placé avant le banc de registre sera nommé *rtrd*. Dans ce cas il sera à 0 pour sélectionner le numéro du registre résultat, a savoir le registre *rt* . (*rtrd* signifie que si la commande est à 0, le registre écrit sera *rt* , *rd* sinon)

— Le signal de commande du multiplexeur placé à l'entrée sera nommé *rtImm*. Dans ce cas il sera à 1 pour sélectionner la valeur immédiate *Imm16* transformée en un valeur 32 bits.

— Le signal *ualmem* est ici indifférent car nous n'écrivons aucune valeur dans le banc de registre.

7.4 Instructions de branchement conditionnel

L'instruction bne *rs* , *rt* ,*Imm16* réalise l'opération de branchement en *cp* = (*cp* +4)+ (Imms << 2) si *rs* ≠ *rt* . Sinon *cp* = *cp* +4

Lors du branchement conditionnel, l'UAL est utilisée pour effectuer le calcul de l'adresse de branchement. Un comparateur teste l'égalité des deux opérandes *rs* et *rt* .

Le multiplexeur piloté par le résultat du comparateur positionne sur sa sortie la valeur calculée par l'UAL dans le cas d'un branchement pris (si *rs* ≠ *rt*), c'est à dire (*cp* +4)+(*Imm16* << 2), ou (*cp* +4) dans le cas d'un branchement non pris.

Pour calculer l'adresse de la cible, nous devons utiliser maintenant la valeur immédiate décalée à gauche de deux rangs sur l'entrée de l'UAL. Nous ajoutons un multiplexeur pour sélectionner cette valeur en cas de branchement.

Le schéma de la figure Fig 7.13 représente le chemin de données pour les instructions bne lorsque le branchement est pris. L'instruction dans la mémoire **0x**0040 0000 à pour valeur **0x**1482 ffff. C'est l'instruction bne *$v0* , *$a0* ,-1. Observer l'état du comparateur et la valeur de l'adresse de l'instruction suivante.

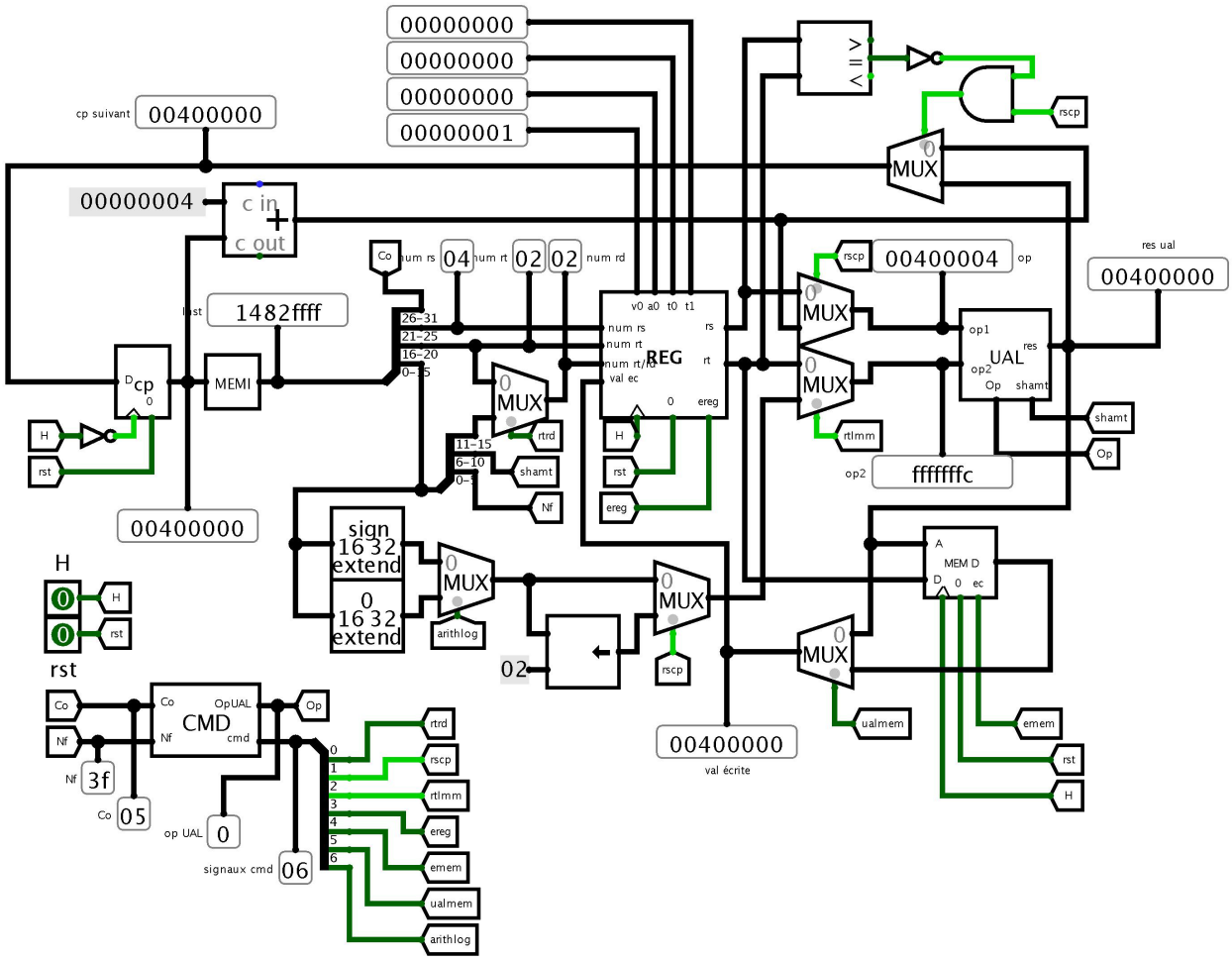


Figure 7.13 – Chemin de données pour un branchement pris

Le schéma de la figure Fig 7.14 représente le chemin de données pour les instructions bne lorsque le branchement n'est pas pris. L'instruction dans la mémoire **0x**0040 0000 à pour valeur **0x**1482 ffff. C'est l'instruction bne *$v0* , *$a0* ,-1. Observer l'état du comparateur et la valeur de l'adresse de l'instruction suivante.

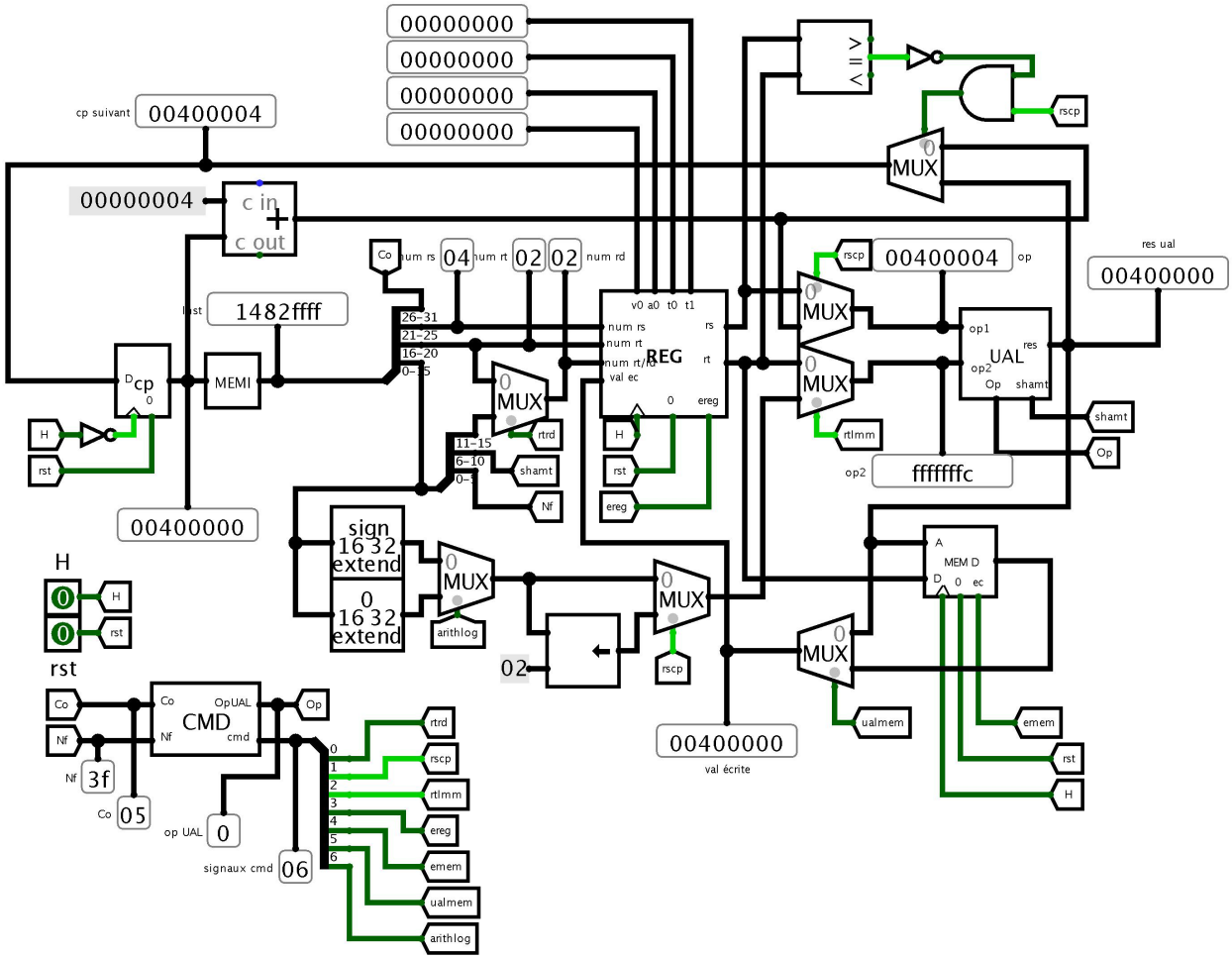


FIGURE 7.14 – Chemin de données pour un branchement non pris

Remarque 43 :

- Le signal *ereg*=0 car nous ne devons pas enregistrer le résultat de l'instruction dans le banc de registre.
- Le signal *ec*=0 car nous ne devons pas modifier la valeur contenue dans la mémoire de données.
- Le signal de commande du multiplexeur placé avant le banc de registre sera nommé *rtrd*. Dans ce cas il sera à 0 pour sélectionner le numéro du registre résultat, a savoir le registre *rt* . (*rtrd* signifie que si la commande est à 0, le registre écrit sera *rt* , *rd* sinon)
- Le signal de commande du multiplexeur placé à l'entrée sera nommé *rtImm*. Dans ce cas il sera à 1 pour sélectionner la valeur immédiate *Imm16* transformée en un valeur 32 bits.
- Le signal de commande du multiplexeur placé à l'entrée supérieure du banc de registre sera nommé *rscp*. Dans ce cas il sera à 1 pour sélectionner la valeur immédiate du registre *cp* . Ce signal est utilisé aussi pour décaler la valeur immédiate.

7.5 Le contrôle

Afin d'assurer le fonctionnement correct du chemin de données, nous devons ajouter un circuit séquentiel calculant les commandes des divers multiplexeurs et signaux d'écriture. Ce circuit utilise le code opération Co et le numéro fonction Nf et détermine les valeurs binaires des signaux suivant :

1. *rtrd* : positionne la commande du multiplexeur permettant de sélectionner le numéro du registre à écrire : *num rt* (cas 0) ou *num rd* (cas 1)

2. *rscp* : positionne la commande du multiplexeur permettant de placer la valeur contenue dans le registre *rs* (cas 0) ou la valeur de *cp* +4 (cas 1) sur l'entrée haute de l'UAL.

3. *rtimm* : positionne la commande du multiplexeur permettant de placer la valeur contenue dans le registre *rt* (cas 0) ou une des valeurs immédiates (cas 1) sur l'entrée base de l'UAL.

4. *ereg* : si l'instruction modifie un registre, ce signal indique au banc de registre que la valeur présente sur l'entrée *val ec* doit être stockée dans le registre *num rt/rd*.

5. *emem* : indique si la mémoire de données doit être écrite.

6. *ualmem* : positionne la commande du multiplexeur permettant d'écrire dans le banc de registre, la valeur lue en mémoire (cas 1) ou le résultat de l'opération effectuée par l'UAL (cas 0).

7. *arithlog* : positionne la commande du multiplexeur permettant de choisir le type d'extension de signe, signé (cas 0) et non signé (cas 1) pour les instructions logiques.

8. *op* : signal permettant de déterminer l'opération effectuée par l'UAL

Le schéma de la figure Fig 7.15 représente le principe du décodage des instructions. Dans la partie supérieur du schéma, nous utilisons quatre décodeurs pour décoder les 32 valeurs du code opération et les 32 valeurs du numéro de fonction. Si le code opération est à 0, nous décodons le numéro de fonction. Chaque sortie de ces décodeur correspond à une instructions MIPS comme suggéré avec add et addi .

La partie basse de gauche utilise un encodeur (noté PRI). La valeur de sa sortie correspond à la valeur de l'entrée active (si l'entrée 5 est activée, la sortie est égale à 5). Nous devons envoyer aux entrée du décodeur la ou les instructions (avec des portes logiques) pour obtenir le numéro d'opération compatible avec l'instruction. La partie basse de droite permet de calculer les valeurs des signaux permettant de piloter l'architecture. Nous les étudierons plus précisément ces signaux en exercice théorique et pratique.

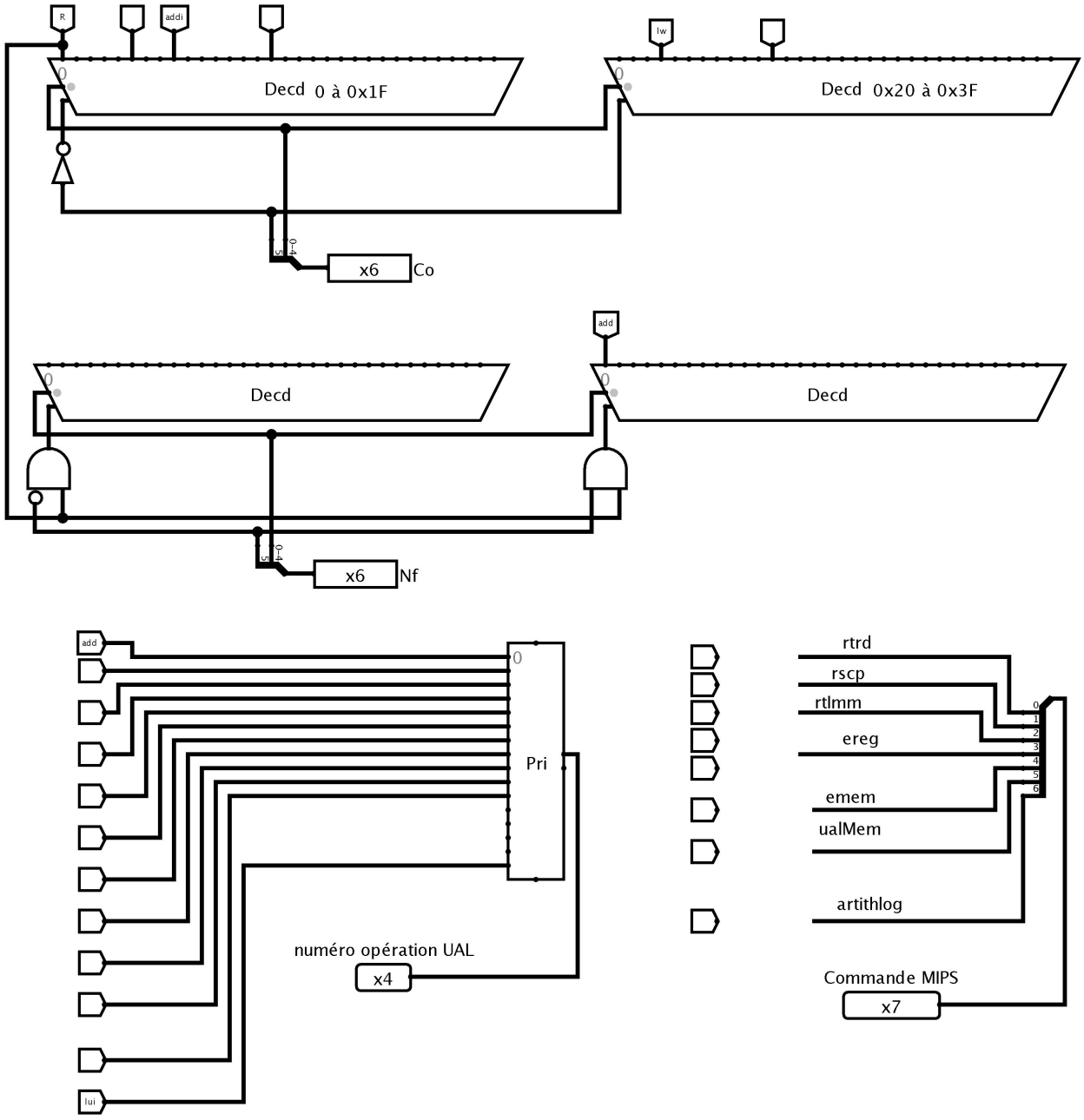


FIGURE 7.15 – Commande MIPS incomplète

Remarque 44 :

Avec logisim, les quatre décodeurs utilisés doivent avoir leurs propriétés **Disable Ouput = Zero**.

7.6 Exercices

Exercice 9 : Théorique : Étude du chemin de données et des commandes

Question 1 : Analyse d'un programme Soit les instructions d'un programme placées à partir de l'adresse **0x**0040 0000 : **0x**00004020, **0x**3c041001, **0x**8c820000, **0x**20840004,**0x**01024020, **0x**1440fffc, **0x**ac880000

Soit les données du programme placées à l'adresse **0x**1001 0000 (valeurs 32 bits) : 1,2,0,0
Ces instructions correspondent aux instructions suivantes

Listing 7.1 – Test architecture

```
1 add $t0,$zero,$zero #00004020
2 lui $a0,0x1001 #3c041001
3 boucle:
4 lw $v0,0($a0) #8c820000
5 addi $a0,$a0,4 #20840004
6 add $t0,$t0,$v0 #01024020
7 bne $v0,$zero,boucle #1440fffc
8 sw $t0,0($a0) #ac880000
```

Pour chaque instruction, donnez les valeurs présentes au différents points de l'architecture ci-dessous

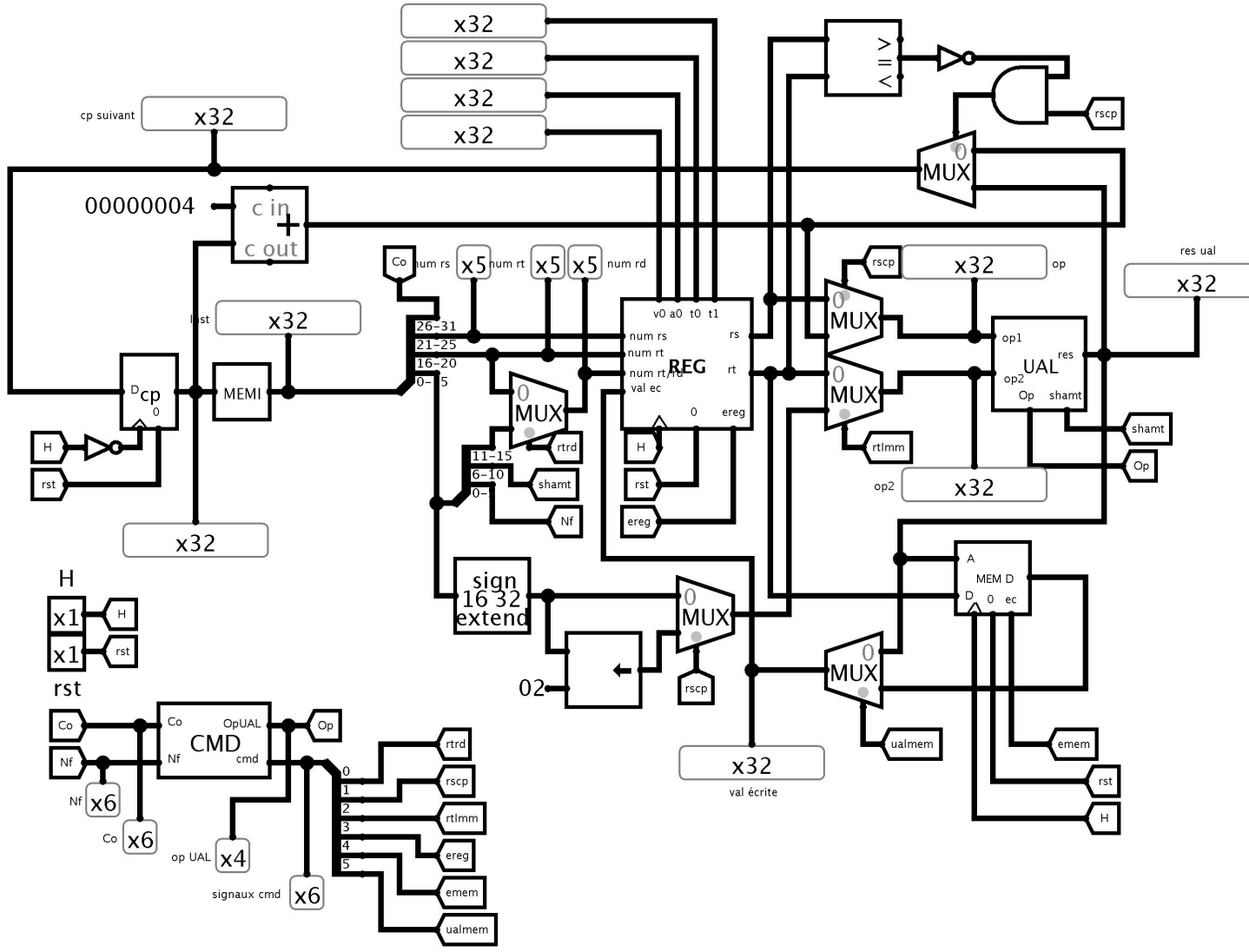


FIGURE 7.16 – Schéma complet de l'architecture

Nous désirons implémenter une architecture MIPS permettant de traiter les instructions suivantes : add , addi , and , andi , bne , lui , lw , nor , or , ori , sll , sra , srl , sub , sw , xor , xori .
Remarque

— Pour l'instruction lui vous effectuerez l'opération Imms << 16.

Question 2 : Décodage du numéro d'opération UAL Pour chaque instruction, donnez la nature de l'opération UAL à réaliser (addition, soustraction, opération logique) et compléter les numéros d'opération UAL (noté *Op* sur les schéma) . Donnez ensuite, pour chaque valeur du numéro d'opération Op, les équations ci-dessous.

Instruction	Co	Nf	type d'opération	Numéro opération *Op*
add	0	32	+	0
addi	8			
and	0	36	ET	1
andi	12			
bne	5			
lui	15			5
lw	35			
nor	0	39		
or	0	37		3
ori	13			
sll	0	0		4
sra	0	3		6
srl	0	2		7
sub	0	34		8
sw	43			
xor	0	38		9
xori	14			

FIGURE 7.17 – Table de vérité numéro d'opération UAL

$Op = 0 \quad si \quad ((Co = ???).(Nf = ???)) + (Co = ???) + (Co = ???) + (Co = ???) \mid (Co = ???)$
$Op = 1 \quad si \quad ((Co = ???).(Nf = ???)) + (Co = ???)$
$Op = 2 \quad si \quad ((Co = ???).(Nf = ???)$
$Op = 3 \quad si \quad ((Co = ???).(Nf = ???)) + (Co = ???)$
$Op = 4 \quad si \quad ((Co = ???).(Nf = ???))$
$Op = 5 \quad si \quad (Co = ???)$
$Op = 6 \quad si \quad ((Co = ???).(Nf = ???))$
$Op = 7 \quad si \quad ((Co = ???).(Nf = ???))$
$Op = 8 \quad si \quad ((Co = ???).(Nf = ???))$
$Op = 9 \quad si \quad ((Co = ???).(Nf = ???)) + (Co = ??)$

Question 3 : Signaux de contrôle de l'architecture Donnez les équations des signaux suivant permettant de piloter les différents multiplexeur en utilisant le modèle ci-dessous. Dans la syntaxe *rtrd* le registre *rd* est positionné sur la sortie du multiplexeur si la valeur *rtrd*=1, *rt* sinon.

Instruction	Co	*rtrd*	*rscp*	*rtimm*	*ereg*	*emem*	*ualmem*	*arithlog*
3 registres arithmétique	0	1	0	0	1	0	0	0
3 registres logique								
Immédiate arithmétique								
Immédiate logique								
bne	5							
lw	35							
sw	43							

FIGURE 7.18 – Table de vérité des signaux de commande - Énoncé

$$
\begin{aligned}
rtrd &= (Co = ???) \\
rscp &= (Co = ???) \\
rtImm &= \overline{(Co = ???)} \\
ereg &= \overline{(Co = ???)???(Co = ???)} \\
emem &= (Co = ???) \\
ualmem &= (Co = ???) \\
arithlog &= (Co = 0).((Nf = ???) + (Nf = ???) + (Nf = ???) + (Nf = ???)) \\
&\quad +(Co = ???) + (Co = ???) + (Co = ???)
\end{aligned}
$$

Remarque 45 :

- les instructions à trois registres sont de la forme $rd \leftarrow rs\ op\ rt$
- les instructions avec une valeur immédiate sont de la forme $rt \leftarrow rs\ op$ Imm
- l'instruction bne est de la forme bne rs , rt ,$Imm16$. Si $rs \neq rt$ alors $cp \leftarrow (cp + 4) + (Imm16 << 2)$
- l'instruction lw est de la forme lw rt ,$Imm16$ (rs). $rt \leftarrow$ Mem32[$Imm16 + rs$]
- l'instruction sw est de la forme sw rt ,$Imm16$ (rs). Mem32[$Imm16 + rs$]$\leftarrow rt$

Remarque 46 :

- Pour chaque type d'instruction, observer quelle opération doit être effectuée par l'UAL.

Question 4 : jr Expliquez ce que nous devons ajouter au chemin de données précédent pour implémenter l'instruction jr du jeu d'instruction MIPS.

Indication 1 : Vous ajouterez un multiplexeur dans le chemin du le registre cp

Correction exercice 9 :

Question 1 : Analyse d'un programme Les différentes figures ci-dessous donnent l'état du processeur lors des trois itérations de la boucle.

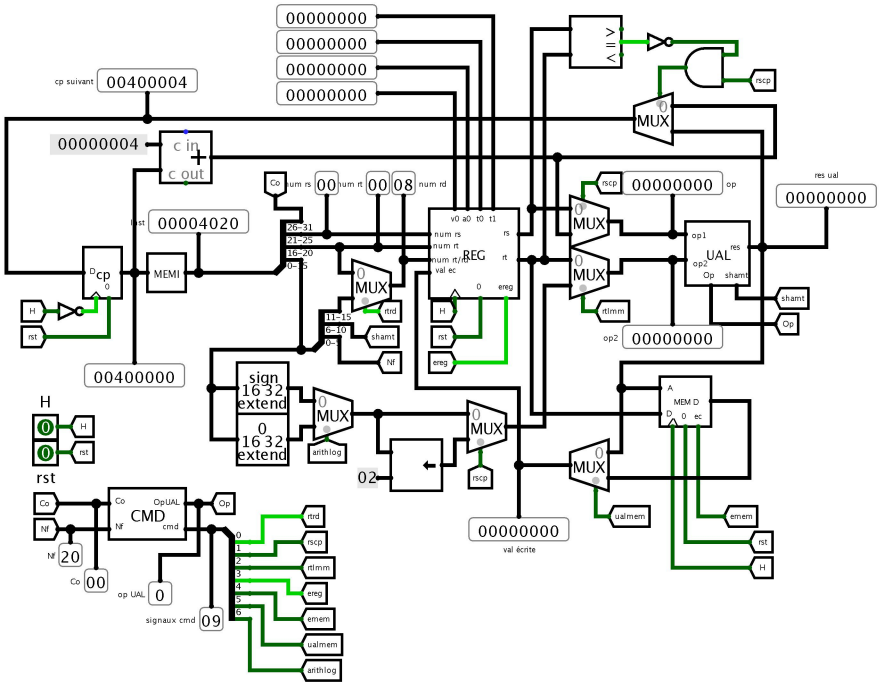


FIGURE 7.19 – addi *$t0* , *$zero* , *$zero*

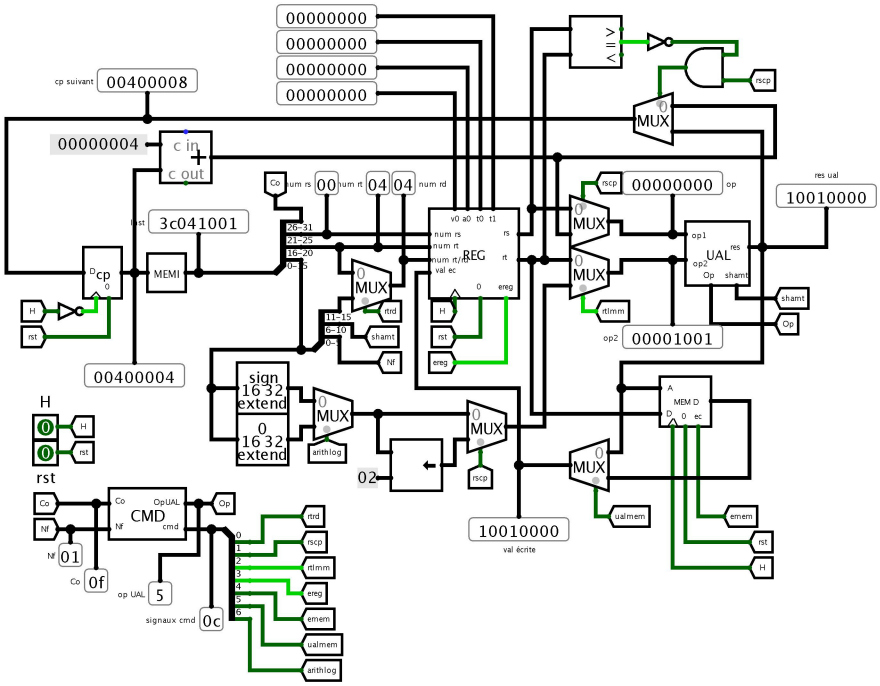


FIGURE 7.20 – lui *$a0* ,**0x**1001

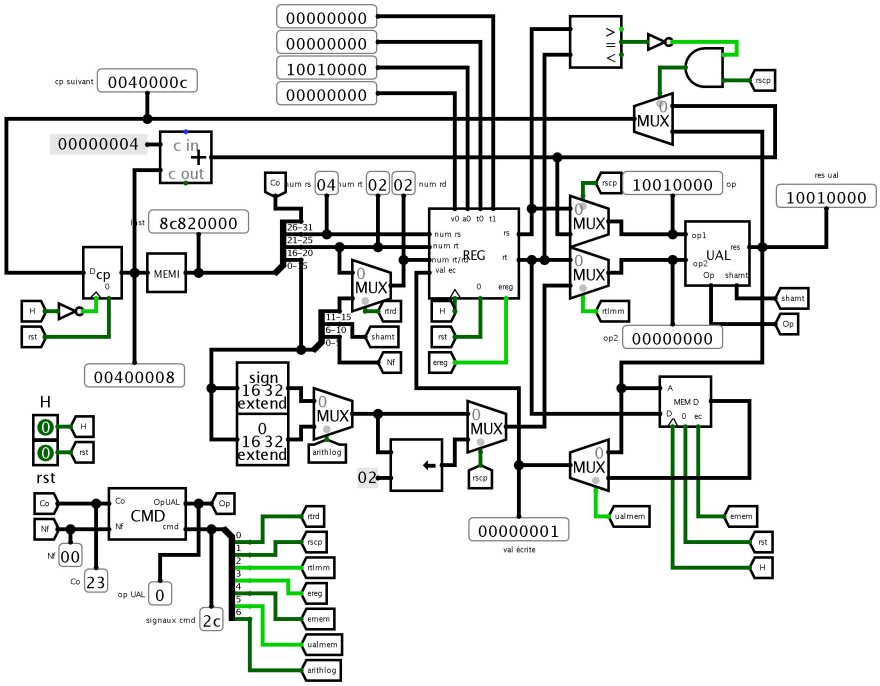


FIGURE 7.21 – lw *$v0* ,0(*$a0*)

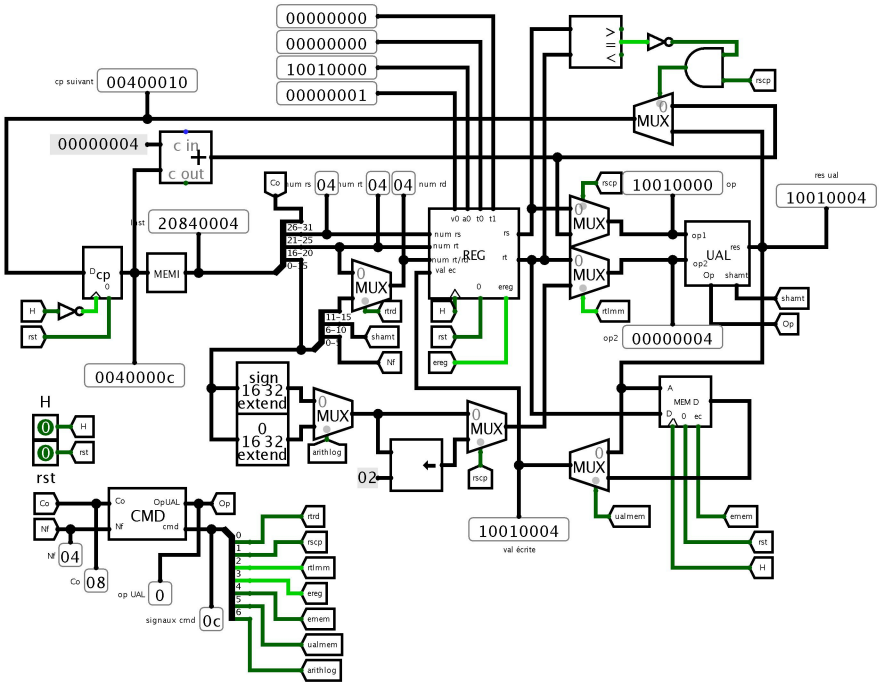


FIGURE 7.22 – addi *$a0* , *$a0* ,4

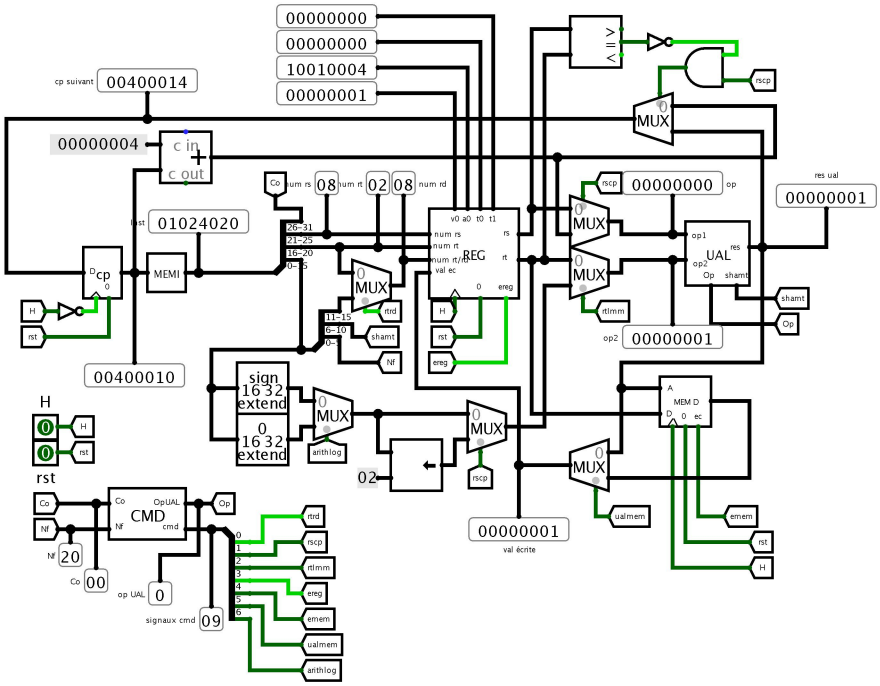


FIGURE 7.23 – add $t0 , $t0 , $v0

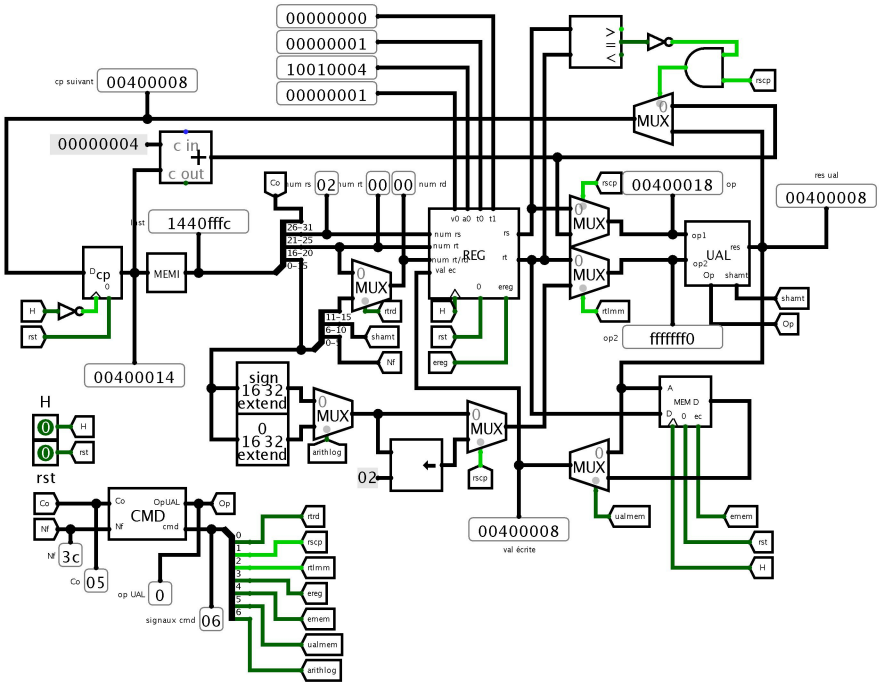


FIGURE 7.24 – bne $v0 , $zero ,boucle

Idem pour la deuxième et la troisième itération. Le dernier bne ne sera pas pris car la valeur lue en mémoire est égale à 0.

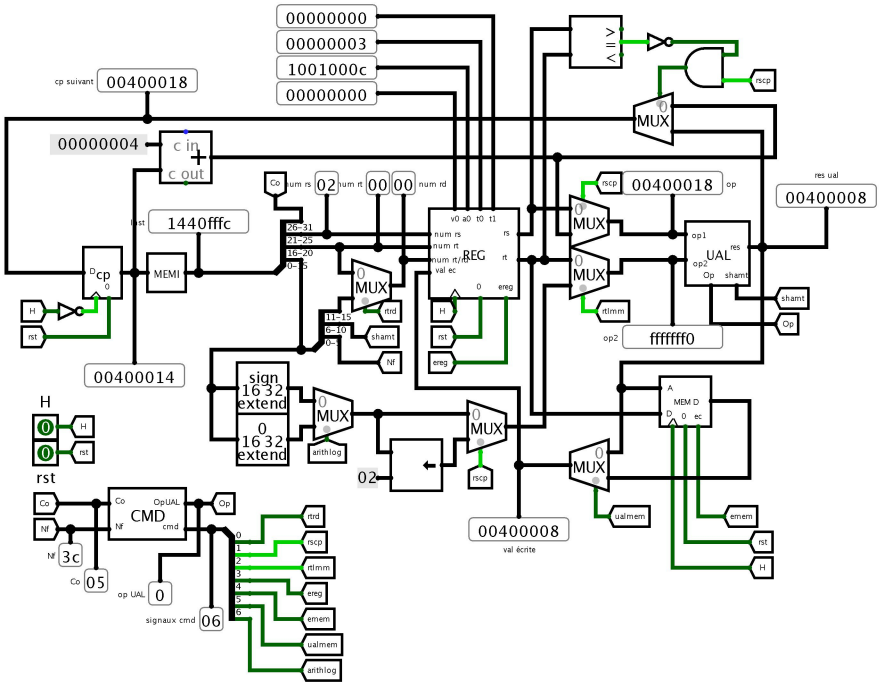


FIGURE 7.25 – bne *$v0* , *$zero* ,boucle

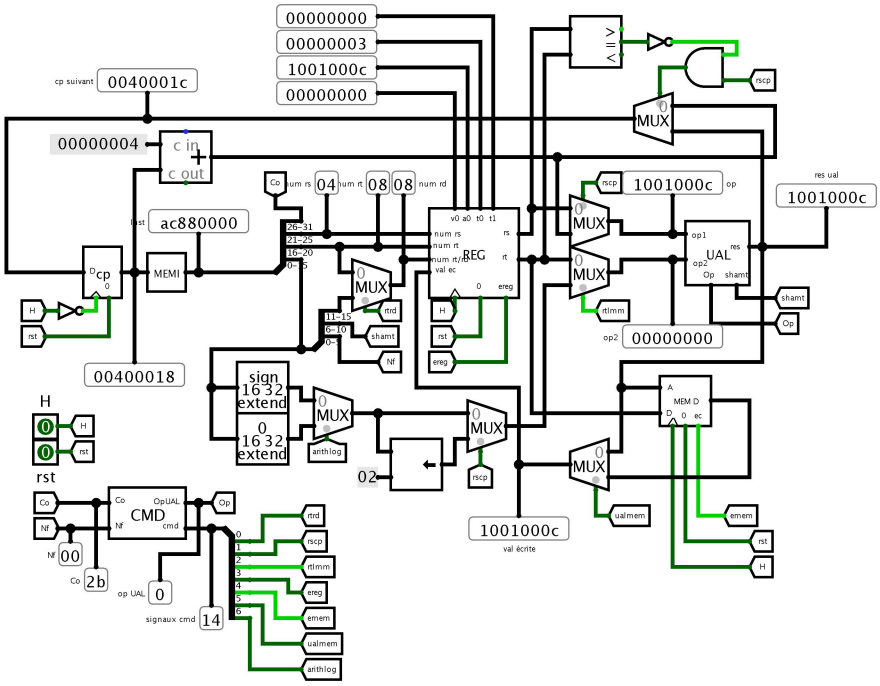


FIGURE 7.26 – sw *$v0* ,0(*$a0*)

La valeur stockée dans la mémoire d'adresse **0x**1001 000C est donc la valeur 3.

Instruction	Co	Nf	type d'opération	Numéro opération *Op*
add	0	32	+	0
addi	8		+	0
and	0	36	&	1
andi	12		&	1
bne	5		+	0
lui	15		<< 16	5
lw	35		+	0
nor	0	39	~ |	2
or	0	37	|	3
ori	13		|	3
sll	0	0	<<	4
sra	0	3	>>	6
srl	0	2	>>>	7
sub	0	34	-	8
sw	43		+	0
xor	0	38	^	9
xori	14		^	9

FIGURE 7.27 – Correction table de vérité numéro d'opération UAL

Question 2 : Numéro d'opération UAL

$$
\begin{aligned}
&Op = 0 \quad si \quad ((Co = 0).(Nf = 32)) + (Co = 8) + (Co = 5) + (Co = 35) + (Co = 43)\\
&Op = 1 \quad si \quad ((Co = 0).(Nf = 35)) + (Co = 12)\\
&Op = 2 \quad si \quad ((Co = 0).(Nf = 39)\\
&Op = 3 \quad si \quad ((Co = 0).(Nf = 37)) + (Co = 13)\\
&Op = 4 \quad si \quad ((Co = 0).(Nf = 0))\\
&Op = 5 \quad si \quad (Co = 15)\\
&Op = 6 \quad si \quad ((Co = 0).(Nf = 3))\\
&Op = 7 \quad si \quad ((Co = 0).(Nf = 2))\\
&Op = 8 \quad si \quad ((Co = 0).(Nf = 34))\\
&Op = 9 \quad si \quad ((Co = 0).(Nf = 38)) + (Co = 14)
\end{aligned}
$$

Question 3 : Signaux de contrôle de l'architecture simplifiée La figure FIG 7.28 donne la table de vérité des signaux de contrôle de l'architecture simplifiée.

Instruction	Co	*rtrd*	*rscp*	*rtimm*	*ereg*	*emem*	*ualmem*	*arithlog*
3 registres arithmétique	0	1	0	0	1	0	0	0
3 registres logique	0	1	0	0	1	0	0	1
Immédiate arithmétique		0	0	1	1	0	0	0
Immédiate logique		0	0	1	1	0	0	1
bne	5	0	1	1	0	0	0	0
lw	35	0	0	1	1	0	1	0
sw	43	0	0	1	0	1	0	0

FIGURE 7.28 – Correction table de vérité signaux de commande

$$
\begin{aligned}
rtrd &= (Co = 0)\\
rscp &= (Co = 5)\\
rtimm &= \overline{(Co = 0)}\\
ereg &= \overline{(Co = 5) + (Co = 43)}\\
emem &= (Co = 43)\\
ualmem &= (Co = 35)\\
arithlog &= (Co = 0).((Nf = 36) + (Nf = 37) + (Nf = 38) + (Nf = 39))\\
&\quad +(Co = 12) + (Co = 13) + (Co = 14)
\end{aligned}
$$

Question 4 : jr Nous devons ajouter un multiplexeur sur l'entrée du registre *cp* . Une entrée de ce multiplexeur est reliée au contenu du registre *rs* et l'autre à la sortie du multiplexeur piloté par le signal *rscp* et la porte logique ET.

Exercice 10 : Pratique : Implémentation de l'architecture

Vous compléterez les circuits du fichier mips32-enonce.cir :

- L'unité arithmétique et logique (placer les différentes fonctions réalisant les opérations)
- Le circuit de commande de l'architecture : pour le tester vous utiliserez les instructions étudiées dans les exercices théoriques.
- Le banc de registres (vous placerez uniquement les registres *$zero* , *$v0* , *$a0* , *$t0* et *$t1*)
- L'architecture complète : vous implémenter et testerez les instructions ayant trois registres, les instructions ayant une valeur immédiate, puis les accès mémoire et enfin le branchement ; pour le tester vous utiliserez les instructions étudiées dans les exercices théoriques.

Remarque 47 :

- Vous utiliserez tous les composants de logisim.
- Pour placer une valeur dans la mémoire (d'instructions ou des données) vous cliquerez avec le bouton de droite sur la mémoire. Avec l'item du menu contextuel *Edit contents* vous entrerez la ou les valeurs dans la mémoire aux adresses désirées. Avec le menu *Save image* vous pourrez sauvegarder le contenu de la mémoire dans un fichier d'extension .hex.
- Les circuits ne vérifient pas que les adresses commencent en **0x**0040 0000 pour les instructions et **0x**1001 0000 pour les données (nous ne vérifions pas la valeurs de 16 bits de poids fort). Vous placerez donc instructions et données à partir de l'adresse **0x**0000 0000 des mémoire d'instruction et de données.
- La mémoire d'instructions est de type ROM (*read only memory*). Cette mémoire peut être juste lue et non écrite. Lorsque vous rechargez le circuit, le contenu de la mémoire est automatiquement chargé avec les dernières valeurs. Vous utiliserez le programme de l'exercice théorique pour tester le fonctionnement de votre architecture.

Correction exercice 10 :

Vous trouverez tout dans le fichier nommé mips32-correction.circ. Le programme de test vu en exercice pratique se nomme mips-td.asm et les codes hexadécimaux mips-td.hex que vous

pouvez charger dans la mémoire d'instruction. N'oubliez pas de modifier les valeur contenues dans la mémoire de données pour effectuer plusieurs itération.

Chapitre 8

Système d'interruptions et d'entrées/sorties

8.1 Organisation

Un système informatique est organisé de la façon suivante présentée sur la figure Fig 8.1

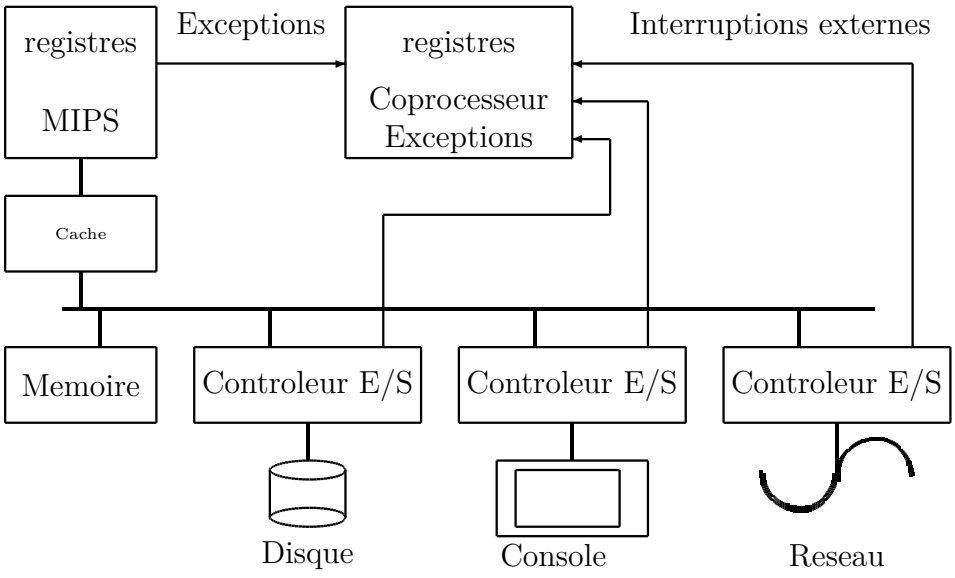


Figure 8.1 – Organisation d'un système

Nous avons étudié le fonctionnement du processeur et nous étudierons la mémoire cache dans la partie suivante. Le système informatique contient aussi des dispositifs d'entrées/sorties (nommé aussi I/O) et un système d'interruptions. Le coprocesseur d'exception gère les diverses interruptions externes et exceptions.

Les interruptions permettent de gérer le système d'entrées/sorties de façon efficace. Prenons par exemple la gestion du périphérique d'entrée (clavier). Nous ne devons pas perdre de caractères en provenance de ce périphérique si le processeur est occupé à une autre activité.

Deux techniques de gestion des événements sont possibles :

L'Interrogation utilisant le principe de l'Algorithme 73.

Algorithme 73 Gestion par interrogation du clavier

```
1: Début
2: Tant Que VRAI Faire
3:    Si Touche clavier appuyée ? Alors
4:       Stocker en mémoire le code ascii du caractère lu
5:    FinSi
6:    Attendre 100 ms
7: Fin Tant Que
8: Fin
```

L'Interruption utilisant le principe de l'ALGORITHME 74.

Algorithme 74 Gestion par interruption du clavier

```
1: Début
2: Si Touche clavier appuyée ? Alors
3:    stocké dans le tampon mémoire le code ascii du caractère lu
4: FinSi
5: Fin
```

L'interrogation effectue une lecture répétitive de l'état du clavier alors qu'avec la technique d'interruption, c'est le clavier qui signale au processeur qu'il doit traiter un caractère et ne pas le perdre.

8.2 Principe

— Les exceptions sont des événements inattendus qui sont générés par le processeur. Le débordement de la capacité de calcul, les erreurs d'accès à la mémoire sont des exceptions. Les appel systèmes (syscall) génèrent aussi des exceptions.
— Les interruptions sont générées par le système d'entrées/sorties pour signaler qu'une donnée d'un périphérique est prêt et requiert l'attention du processeur.

Les interruptions et les exceptions contrairement au branchement peuvent arriver à n'importe quel endroit du programme. Elle ne doivent pas modifier les registres du programme en cours. Pour cela les sous-programmes gérant les interruptions disposent de deux registres nommés *$k0* et *$k1* .
La figure FIG 8.2 présente quelques exemples de sources d'interruptions et d'exceptions.

Nature de l'événement	**Provenance**	Type
Requête E/S	Externe	Interruption
Sycall	Interne	Exception
dépassement arithmétique signé	Interne	Exception
Instruction invalide	Interne	Exception
Problème matériel	Les deux	Les deux

FIGURE 8.2 – Sources des interruptions

Lors de l'arrivée d'un événement le processeur doit terminer l'instruction en cours de traitement, puis exécuter un sous-programme nommé d'exception ou d'interruption qui effectue un traitement spécifique.

8.3 Gestion des exceptions

Dans l'implémentation de l'architecture MIPS, le coprocesseur d'interruption nommé **coprocesseur 0** gère les exceptions et les interruptions. Ce coprocesseur est accessible via quatre registres listés sur la figure Fig 8.3 .

Nom	Numéro	Usage
AdrErrRef	*$8*	Contient l'adresse mémoire où l'erreur s'est produite
Statut	*$12*	Bits de masquage et validation interruptions
Cause	*$13*	Type d'exceptions et interruption en attente
CEP	*$14*	Adresse de l'instruction ayant provoqué l'exception

Figure 8.3 – Registre du coprocesseur c0

Après une exception le registre *$14* nommé registre **compteur programme exception** contient l'adresse de l'instruction en cours d'exécution lors de l'arrivée de l'événement. Si l'instruction ayant provoqué l'exception à fait un accès mémoire, le registre *$8* nommé **AdrErrRef** contient l'adresse de la mémoire référencée.
Les bits du registres *$13* nommé **registre cause** contiennent différentes informations comme le suggère la figure Fig 8.4 .

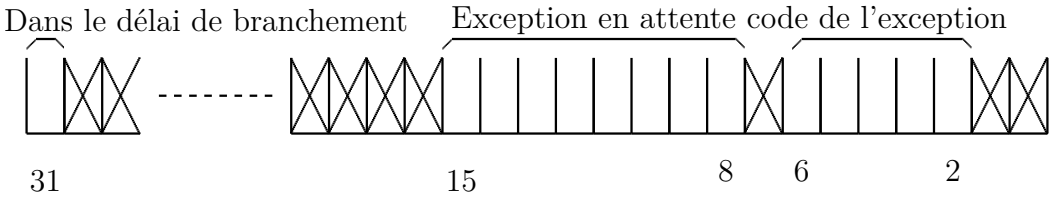


Figure 8.4 – Registre *$13* ou cause du coprocesseur 0

Les bits 2 à 6 indiquent la nature de l'exception. La figure Fig 8.5 liste une partie des causes d'exceptions possibles. Les autres indiquent le périphérique ayant provoqué une interruption. Les bits 8 et 9 seront utilisés pour signaler des événement en provenance de la console (écran et clavier).

Bit 2 à 6 de *$13*	Nom	Description
0	I/O	Dispositif d'entrées/sorties
4	ADDRL	Erreur d'adresse Load
5	ADDRS	Erreur d'adresse Store
7	DBE	Erreur sur le bus données
8	SYSCALL	Appel système
10	RI	Instruction erronée
12	OVF	Dépassement arithmétique signé

Figure 8.5 – Causes de l'exception

Les bits du registres *$12* nommé **registre statut** contiennent différentes informations comme le suggère la figure Fig 8.6 .

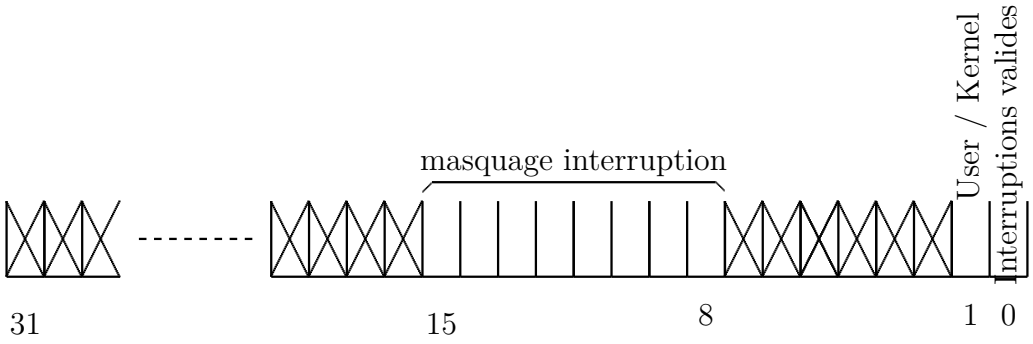


FIGURE 8.6 – Registre *$12* ou statut du coprocesseur 0

Le bits 0 permet de valider le traitement des exceptions.
Pour accéder aux quatre registres du coprocesseur, le MIPS possède deux instructions nommées mfc0 et mtc0 . La syntaxe des ces instructions est donnée sur la figure FIG 8.7 .

Rôle	Syntaxe	Opération	Codage
déplacer depuis c0	mfc0 *$rt* , *$rd*	*$rt* ← *$rd* de c0	Co=**0x**10 et *$rs* =0
déplacer vers c0	mtc0 *$rd* , *$rt*	*$rd* de c0 ← *$rt*	Co=**0x**10 et *$rs* =4

FIGURE 8.7 – Liste partielle des instructions du coprocesseur c0

Lors de l'arrivée d'une exception ou d'une interruption le MIPS lance le programme se trouvant à l'adresse **0x**8000 0180 (dans l'espace du noyau, pas de l'utilisateur), appelé **gestionnaire d'interruptions**. Le système effectuera alors l'opération voulue pour répondre au problème puis retourner au programme ou le terminer.

8.4 Gestionnaire d'interruption

Le programme doit être écrit dans un segment spécial nomme .ktext **0x**8000 0180, et utilise les données stockées dans .kdata. Les registres permettant de transférer les données entre le gestionnaire d'interruption et le programme utilisateurs sont nommés *$k0* et *$k1* .
Le gestionnaire doit effectuer les opérations décrite dans l'ALGORITHME 75.

Algorithme 75 Gestionnaire d'exception

1: **Début**
2: Charger les registres du coprocesseur c0 dans *$k0* et *$k1* .
3: Isoler la cause de l'exception.
4: Traiter l'exception.
5: Retour au programme utilisateur ou le terminer,
6: **Fin**

8.5 Entrées/Sorties de caractères par interrogation

Dans l'architecture MIPS la console (système clavier / écran) est "mappée" en mémoire. Cela signifie que les entrées/sorties sont accessibles via des adresses mémoires spécifiques. Cet espace mémoire est nommé MMIO (Memory Manager IO) et débute à l'adresse **0x**FFFF 0000.
Pour la réception de caractères (depuis le clavier) :

— l'adresse de réception du caractère est **0x**FFFF 0004,
— l'adresse du contrôleur de réception est **0x**FFFF 0000. Si le bit 0 est à 1 cela signifie qu'un caractère est en attente. Ce bit repassera à 0 lors de l'instruction de lecture mémoire à l'adresse de réception.

Pour l'envoi de caractères (vers l'écran) :

— l'adresse d'envoi du caractère est **0x**FFFF 000C,
— l'adresse du contrôleur d'émission est **0x**FFFF 0008. Le bit 0 passera à 0 lors de l'écriture mémoire à l'adresse d'envoi. Il reviendra à 1 après un temps nécessaire au traitement du caractère. En effet, le gestionnaire d'exception devra lire le code de la touche appuyée, éventuellement le convertir et stocker le résultat en mémoire.

La figure Fig 8.8 résume le rôle des adresses et des bits associés à la console lors d'un traitement des événements clavier par interrogation.

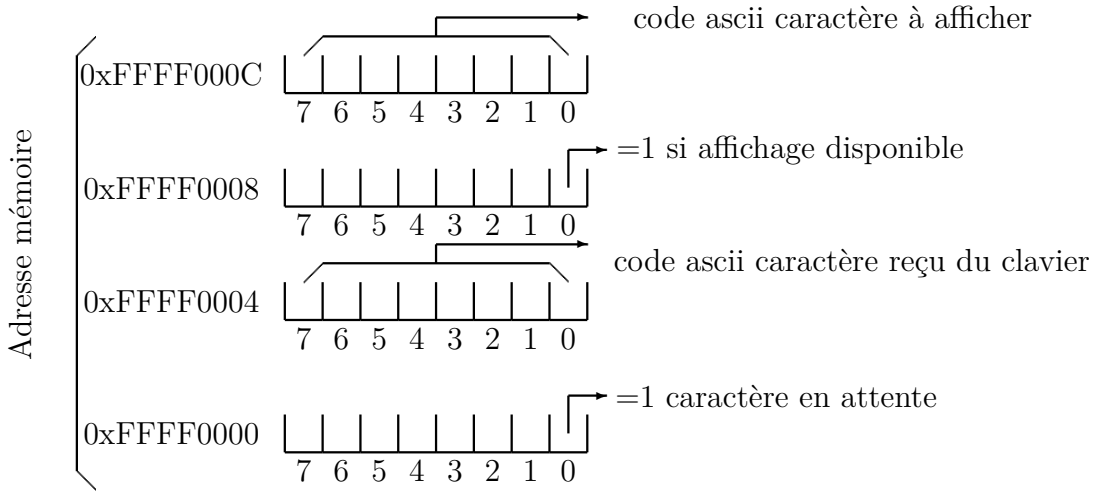


Figure 8.8 – Console gérée par interrogation

8.6 Entrées/Sorties de caractères par interruption

Le bit numéro 1 des deux contrôleurs de la console permet de valider les interruptions lors de l'envoi ou de la réception d'un caractère. Lors d'un événement venant des entrées/sorties, une interruption de cause numéro 0 est générée. Le bit 8 (respectivement 9) du registre cause indique la réception d'un caractère du clavier (respectivement la fin de l'envoi d'un caractère à l'écran). La figure Fig 8.9 résume le rôle des adresses et des bits associés à la console lors d'un traitement des événements clavier par interruption.

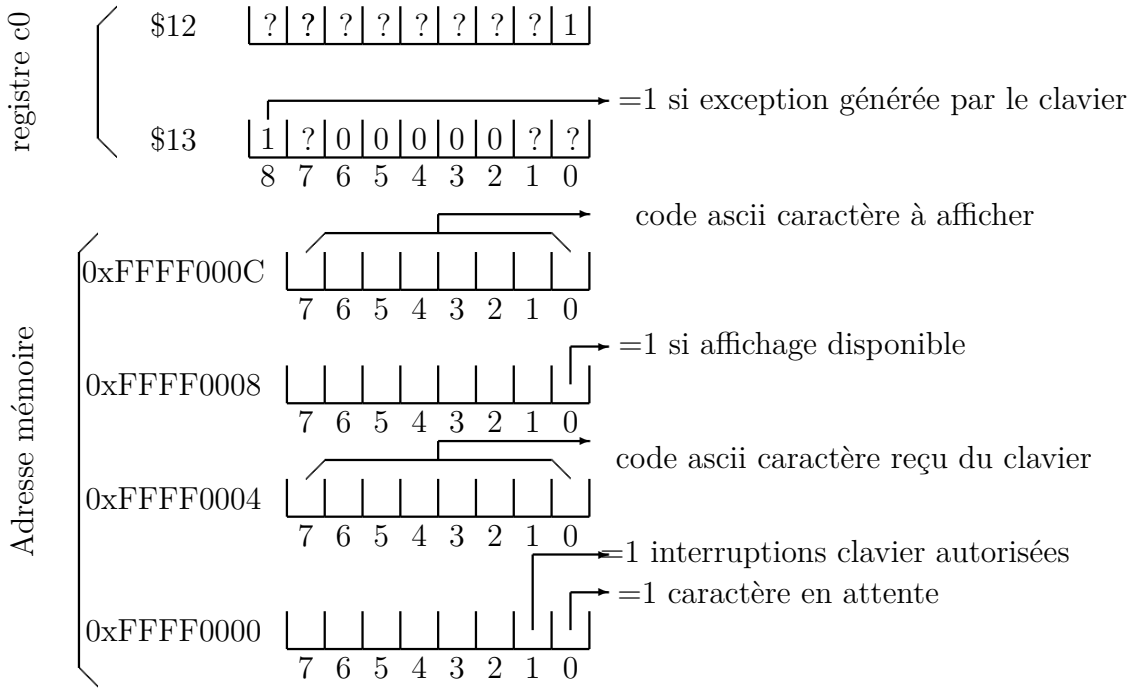


FIGURE 8.9 – Organisation de la console

8.7 Exercices

Exercice 11 : Théorique : Gestion des exceptions

Question 1 : Déclencher exceptions Donnez les instructions MIPSgénérant les exceptions suivantes.

1. Un dépassement arithmétique signé (changement de signe) en utilisant l'incrémentation du registre *$t0* .
   ```
   addi $t0 , $t0 ,...
   addi $t0 , $t0 ,1 #exception ici
   ```
2. Une erreur d'adresse en écriture d'une donnée 32 bits en utilisant l'adresse d'une donnée non alignée (non divisible par quatre) placée dans le registre *$t0* .
   ```
   la $t0 ,NonAligne
   sw $t1 ,...( $t0 ) #exception ici
   ```
3. Une erreur d'adresse en lecture d'une donnée 32 bits en utilisant l'adresse d'une donnée alignée (divisible par quatre) placée dans le registre *$t0* .
   ```
   la $t0 ,Aligne
   lw $t1 ,...( $t0 ) #exception ici
   ```

Le segment de donnée contient les définitions suivantes :

```
.data
Aligne : .word 1 # adresse divisible par 4
inutilise : .byte 0
NonAligne : .byte 1,2,3,4 # adresse non divisible par 4
```

Question 2 : Affichage numéro exception Complétez l'algorithme du programme d'exception qui affiche le numéro de l'exception générée par le coprocesseur c0 en utilisant les bits 2 à 6 du registre cause. Il est accessible en utilisant la syntaxe *c0$13* .

Algorithme 76 Afficher numéro d'exception

```
1: Début du programme d'exception
2:     $k0  ← c0$13
3:     $k0  ← ( $k0  >>> ???)
4:   ??? ← $k0  & ???
5: Lancer l'appel système numéro 1
6:    Terminer le programme
7: Fin
```

Question 3 : Gestion de la console par interrogation En utilisant la technique d'interrogation, complétez l'algorithme du programme qui attend un caractère (attente que le bit 0 du contrôleur de réception **0x**FFFF 0000 = 1) et l'affiche (attente que le bit 0 du contrôleur d'émission **0x**FFFF 0008 = 1). Vous utiliserez deux structures **SiCond** () **Aller à** .

Algorithme 77 Énoncé Gestion de la console par interrogation

```
1: Variables
2:     $t0 , $t1  : entier
3: Début
4: AttenteLecture :
5:     $t0  ← Mem32[???]
6:     $t0  ← $t0  & ???
7:   SiCond ( $t0  =???) Aller à AttenteLecture
8:     $t1  ← Mem32[???]
9: AttenteEcriture :
10:    $t0  ← Mem32[???]
11:    $t0  ← $t0  & ???
12:  SiCond ( $t0  =????) Aller à AttenteEcriture
13:  Mem32[???]←???
14: Fin
```

Question 4 : Gestion de la console par interruption Complétez l'algorithme :

- d'un programme qui validera la réception des interruptions émises depuis le clavier (bit 1 du contrôleur de réception **0x**FFFF 0000 = 1), puis effectuera une boucle infinie. Dans cette boucle, vous afficherez le contenu du registre *$k1* s'il est différent de zéro (en utilisant l'interrogation pour l'affichage), puis mettrez le registre *$k1* à zéro.
- d'un gestionnaire d'exceptions qui place le caractère reçu dans le registre *$k1* si le numéro de l'exception vaut 0 et si le bit 8 du registre *$13* vaut 1 (clavier qui à émis une demande d'interruption)

Algorithme 78 Énoncé gestion de la console par exception

```
1: Variables
2:     $t0 , $k0 , $k1  : entier
3: Début
4:     $t0  ← Mem32[0xFFFF 0000]
5:     $t0  ←  $t0   |  ???
6:    Mem32[0xFFFF 0000]←  $t0
7:     $k1  ← 0
8: Attente :
9:    SiCond ( $k1  = 0) Aller à Attente
10: AttenteEcriture :
11:    $t0  ← Mem32[???]
12:    $t0  ←  $t0   &  ???
13:   SiCond ( $t0  = 0) Aller à AttenteEcriture
14:   Mem32[???]←???
15:    $k1  ← 0
16:   SiCond (???) Aller à Attente
17: Fin
18: Débutdu programme d'exception
19:    $k0  ← c0$13
20:    $k0  ← ( $k0  >>> ???)
21:    $k0  ←  $k0   &  ???
22:   SiCond ( $k0  ≠???) Aller à Fin
23:   ??? ← Mem32[???]
24:    Retourner au programme utilisateur.
25: Fin
```

Question 5 : Console et appels système Nous souhaitons piloter la console en utilisant deux appels système.

1. L'appel système numéro $v0 =**0x**60 lit le code ascii d'un caractère depuis le clavier et le place dans le registre $v0 en utilisant l'interrogation.

2. L'appel système numéro $v0 =**0x**61 affiche le code ascii du caractère placé dans le registre $a0 en utilisant l'interrogation.

— Donnez l'algorithme du programme qui lance les deux appels système ci-dessus et effectue un boucle infinie.
— Donnez l'algorithme du programme gestionnaire d'exceptions qui utilise la technique d'interrogation pour lire et afficher un caractère. L'appel système génère une cause valant 8, et la valeur du registre $v0 permet de sélectionner l'action à effectuer.

Indication 1 : Le retour au programme utilisateur se fait à l'instruction suivant l'appel système. Pour cela vous modifierez la valeur du registre c0$14 .

Algorithme 79 Énoncé Gestion de la console par appels systèmes

```
1:  Variables
2:      $t0 , $k0 , $v0 , $a0  : entier
3:  Début
4:     Lancer l'appel système 0x60
5:      $a0  ← ???
6:     Lancer l'appel système 0x61
7:     SiCond ( $0  =  $0 ) Aller à début
8:  Début du programme d'exception
9:      $k0  ← c0$13
10:     $k0  ← ( $k0  >>> 2)  & 0x1F
11:    SiCond ( $k0  ≠ ???) Aller à modifcep
12:    SiCond ( $v0  ≠ ???) Aller à suite
13: AttenteLecture :
14:     $t0  ← Mem32[0xFFFF 0000]
15:     $t0  ←  $t0  & 1
16:    SiCond ( $t0  = 0) Aller à AttenteLecture
17:    ??? ← Mem32[0xFFFF 0004]
18:    SiCond ( $0  =  $0 ) Aller à modifcep
19: suite :
20:    SiCond ( $v0  ≠ ???) Aller à modifcep
21: AttenteEcriture :
22:     $t0  ← Mem32[0xFFFF 0008]
23:     $t0  ←  $t0  & 1
24:    SiCond ( $t0  = 0) Aller à AttenteEcriture
25:    Mem32[0xFFFF 000C]← ???
26: modifcep :
27:     $k0  ← c0$14
28:     $k0  ←  $k0  + ???
29:    c0$14  ←  $k0
30: Fin
```

Correction exercice 11 :

Question 1 : Déclencher exceptions

1. addi $t0 , $t0 ,**0x**7FFF FFFF
   addi $t0 , $t0 ,1 #exception ici changement de signe

2. sw $t1 ,0($t0)

3. lw $t1 ,1($t0)

Question 2 : Affichage numéro exception

Algorithme 80 Afficher numéro d'exception

```
1: Débutdu programme d'exception
2:     $k0 ← c0$13
3:     $k0 ← ( $k0 >>> 2)
4:     $a0 ← $k0 & 0x1F
5:   Lancer l'appel système 1
6:   Terminer le programme
7: Fin
```

Question 3 : Gestion de la console par interrogation

Algorithme 81 Gestion de la console par interrogation

```
1: Variables
2:     $t0 , $t1 : entier
3: Début
4: AttenteLecture :
5:     $t0 ← Mem32[0xFFFF 0000]
6:     $t0 ← $t0 & 1
7:   SiCond ( $t0 = 0) Aller à AttenteLecture
8:     $t1 ← Mem32[0xFFFF 0004]
9: AttenteEcriture :
10:    $t0 ← Mem32[0xFFFF 0008]
11:    $t0 ← $t0 & 1
12:  SiCond ( $t0 = 0) Aller à AttenteEcriture
13:  Mem32[0xFFFF 000C]← $t1
14: Fin
```

Question 4 : Gestion de la console par interruption

Algorithme 82 Gestion de la console par exception

```
1:  Variables
2:      $t0 , $k0 , $k1  : entier
3:  Début
4:      $t0  ← Mem32[0xFFFF 0000]
5:      $t0  ← $t0  | 2
6:     Mem32[0xFFFF 0000]← $t0
7:      $k1  ← 0
8:  Attente :
9:     SiCond ( $k1  = 0) Aller à Attente
10: AttenteEcriture :
11:     $t0  ← Mem32[0xFFFF 0008]
12:     $t0  ← $t0  & 1
13:    SiCond ( $t0  = 0) Aller à AttenteEcriture
14:    Mem32[0xFFFF 000C]← $k1
15:     $k1  ← 0
16:    SiCond ( $0  =  $0 ) Aller à Attente
17: Fin
18: Débutdu programme d'exception
19:     $k0  ← c0$13
20:     $k0  ← ( $k0  >>> 2)
21:     $k0  ← $k0  & 0x1F
22:    SiCond ( $k0  ≠ 0x40) Aller à Fin
23:     $k1  ← Mem32[0xFFFF 0004]
24:    Retourner au programme utilisateur.
25: Fin
```

Question 5 : Appels systèmes

Algorithme 83 Gestion de la console par appels systèmes

```
1: Variables
2:     $t0 , $k0 , $v0 , $a0  : entier
3: Début
4:    Lancer l'appel système 0x60
5:     $a0  ← $v0
6:    Lancer l'appel système 0x61
7:    SiCond ( $0  =  $0 ) Aller à début
8: Début du programme d'exception
9:     $k0  ← c0$13
10:    $k0  ← ( $k0  >>> 2)  & 0x1F
11:   SiCond ( $k0  ≠ 8) Aller à modifcep
12:   SiCond ( $v0  ≠ 0x60) Aller à suite
13: AttenteLecture :
14:    $t0  ← Mem32[0xFFFF 0000]
15:    $t0  ← $t0  & 1
16:   SiCond ( $t0  = 0) Aller à AttenteLecture
17:    $v0  ← Mem32[0xFFFF 0004]
18:   SiCond ( $0  =  $0 ) Aller à modifcep
19: suite :
20:   SiCond ( $v0  ≠ 0x61) Aller à modifcep
21: AttenteEcriture :
22:    $t0  ← Mem32[0xFFFF 0008]
23:    $t0  ← $t0  & 1
24:   SiCond ( $t0  = 0) Aller à AttenteEcriture
25:   Mem32[0xFFFF 000C]← $a0
26: modifcep :
27:    $k0  ← c0$14
28:    $k0  ← $k0  +4
29:   c0$14  ← $k0
30: Fin
```

Exercice 12 : Pratique : Gestion des exceptions

Question 1 : Déclenchement exception Testez le programme de l'exercice théorique.

Question 2 : Affichage numéro d'exception Testez le programme de l'exercice théorique. Pour tester les entrées/sorties de la console, vous utiliserez l'outil intégré à MARS. Cet outils se trouve dans le menu **tools** et se nomme **keyboard and display simulator**. L'outil doit être connecté pour fonctionner. Dans le segment de données vous afficherez le segment MMIO. Le bouton reset de l'outil permet de réinitialisé les contrôleurs (le bit de 0 du contrôleur d'émission est mis à 1 pour dire qu'un caractère peut être envoyé)

Question 3 : Gestion de la console par interrogation Testez le programme de l'exercice théorique.

Indication 1 :

— L'affichage dans la fenêtre d'envoi se fait avec une latence de cinq instructions (temps de traitement du caractère). Votre programme contiendra donc au moins 5 instructions après l'envoi du caractère à écrire.

— Vous devez vérifier que le mot stocké dans la mémoire **0x**FFFF 0008 vaut 1, ceci indique que l'affichage est prêt. Si ce n'est pas le cas, pensez à réinitialiser l'outil **keyboard and display simulator**.

Question 4 : Gestion de la console par exception Testez le programme de l'exercice théorique.

Indication 1 :

— Vous lancerez le programme en pas à pas et vous observerez le registre *$k1* . Vous pouvez aussi modifier la position du curseur nommé "run speed" afin d'observer le déroulement du programme.

— Vous terminez le gestionnaire d'interruption avec la suite d'instructions suivantes, détaillée dans le fichier HP_AppaA.pdf.

```
mtc0  $zero , $13
mfc0  $k0 ,  $12
andi  $k0 , 0xFFFD
ori  $k0 , 1
mtc0  $k0 ,  $12
eret
```

Question 5 : Appels systèmes Testez le programme de l'exercice théorique.

Correction exercice 12 :

Listing 8.1 – Générer exceptions

```
1  .data
2    #.align 1
3    Aligne: .word 1
4    inutilise : .byte 0
5    NonAligne : .byte 0x12,0x34,0xAB,0xCD,0xEF
6
7  .text
8  main:
9    #Decommenter l'exception désirée.
10   #Exception 12 Erreur arithmétique
11   addi $t1,$zero,0x7FFFFFFF
12   addi $t1,$t1,1 #-> exception 12
13   #Exception 4 (lw) ou 5 (sw)
14   la $t0,Aligne
15   #lw $t2,1($t0) #exception 4
16   la $t0,NonAligne
17   #sw $t2,0($t0) #exception 5
```

Listing 8.2 – Afficher le numéro d'une exceptions

```
1  .ktext 0x80000180
2    #Charger le registre cause
3    mfc0 $k0,$13 #Cause
4    #Extraction ExcCode
5    srl $k0,$k0,2 #bit 0 et 1 inutile
6    andi $a0,$k0,0x1F #masque sur 5 bits
7    addi $v0,$zero,1 #afficher un entier
8    syscall
9    addi $v0,$zero,10 #exit
10   syscall
```

Listing 8.3 – Entrées / sorties par interrogation

```
1  .text
2  main:
3  boucle:
4    la $t0,0xffff0000 #Adresse de base
5    #Attendre qu'un caractère soit présent
6  AttenteLecture:
7    lw $t1,0($t0)
8    andi $t1,$t1,1
9    beq $t1,$zero,AttenteLecture
10   #Charger la valeur lue au clavier depuis RDR
11   lw $t2,4($t0)
12   #Attendre que le buffer soit disponible ecriture
13 AttenteEcriture:
14   lw $t1,8($t0) #TCR
15   andi $t1,$t1,1
16   beq $t1,$zero,AttenteEcriture:
17   #Ecrire le caractère lu
18   sw $t2,0xC($t0)
19   #Attendre que 5 instructions pour voir l'affichage
```

```
20    nop
21    nop
22    nop
23    nop
24    nop
25    #exit
26    add $v0,$zero,10
27    syscall
```

Listing 8.4 – Entrées / sorties interruption

```
1  .text
2  main:
3    #valider interruption reception dans controleur
4    la $t0,0xffff0000
5    lw $t1,0($t0)
6    ori $t1,$t1,2
7    sw $t1,0($t0)
8    #attendre
9  Attendre:
10   nop #afin de voir le programme tourner !
11   beq $k1,$zero,attendre
12 Attente_Ecriture:
13   nop #afin de voir le programme tourner !
14   lw $t1,0x8($t0)
15   andi $t1,$t1,1
16   beq $t1,$zero,Attente_Ecriture
17   sw  $k1,0xC($t0)
18   add $k1,$zero,$zero
19   beq $zero,$zero,Attendre
20 .ktext 0x80000180
21   #Charger le registre cause
22   mfc0 $k0,$13
23   #tester les bit 2 à 6 qui contiennent la cause
24   srl $k0,$k0,2
25   andi $k0,$k0,0x5F
26   #Cause = 0 et Clavier bit8=1?
27   bne $k0,0x40,fin
28   la $k0,0xffff0000
29   lw $k1,4($k0)
30 fin:
31   mtc0 $zero,$13
32   mfc0 $k0, $12 # Fix Status register
33   andi $k0, 0xfffd # Clear EXL bit
34   ori $k0, 0x1 # Enable interrupts
35   mtc0 $k0, $12
36   eret # Return to EPC
```

Listing 8.5 – Entrées / sorties avec syscall

```
1  .text
2  bouclei:
3    addi $v0,$zero,0x60
4    syscall
5    addi $a0,$v0,0
6    addi $v0,$zero,0x61
```

```
 7     syscall
 8     j bouclei
 9   .ktext 0x80000180
10     mfc0 $k0,$13
11     srl $k0,$k0,2
12     andi $k0,$k0,0x1F
13     bne  $k0,8,Fin
14     bne  $v0,0x60,suite
15     la $k0,0xFFFF0000
16   boucle: lw $k1,0($k0)
17     andi $k1,$k1,1
18     beq $k1,$zero,boucle
19     lb $v0,4($k0)
20     j Fin
21   suite:  bne  $v0,0x61,Fin
22     la $k0,0xFFFF0000
23   boucle2:lw $k1,8($k0)
24     andi $k1,$k1,1
25     beq $k1,$zero,boucle2
26     sw $a0,0xC($k0)
27   Fin:
28     mtc0 $zero,$13
29     mfc0 $k0, $12 # Fix Status register
30     andi $k0, 0xfffd # Clear EXL bit
31     ori $k0, 0x1 # Enable interrupts
32     mtc0 $k0, $12
33     mfc0 $k0, $14 # Next Instruction in EPC
34     addi $k0,$k0,4
35     mtc0 $k0, $14
36     eret # Return to EPC
```

Troisième partie

Accélération des processeurs

Bibliographie :

Sur le chapitre 9 hiérarchie mémoire :

1. **Architecture des Ordinateurs : Une approche Quantitative ITP J Hennessy A Patterson** Chapitre 5.1 à 5.2 et 5.11 à 5.14
2. **Organisation et conception des ordinateurs : L'interface matériel/logiciel DUNOD A Patterson J Hennessy** Chapitre 7.1, 7.2 et 7.5 à 7.8.
3. **Architecture de l'ordinateur Nicolas P Carter Schaum's**Chapitre 9.

Sur le chapitre 10 pipeline :

1. **Architecture des Ordinateurs : Une approche Quantitative ITP J Hennessy A Patterson** Chapitre 3.1 à 3.5 et 3.10 à 3.13
2. **Organisation et conception des ordinateurs : L'interface matériel/logiciel DUNOD A Patterson J Hennessy** Chapitre 6.1 à 6.7 et 6.10 à 6.13.
3. **Architecture de l'ordinateur Nicolas P Carter Schaum's**Chapitre 6.

Chapitre 9

La hiérarchie mémoire

Pour que le processeur fonctionne correctement, nous devons l'alimenter en instructions et en données. Nous allons présenter dans ce chapitre les techniques permettant d'obtenir ces informations de façon optimale.

9.1 Principe de la hiérarchie mémoire

De nos jours, les microprocesseurs sont très rapides. Nous devons donc leur fournir des instructions et des données à traiter en grand nombre. Or la technologie des mémoires n'évolue pas si rapidement et nous constatons un écart entre les possibilités de fonctionnement de ces deux éléments de l'architecture de l'ordinateur. La figure Fig 9.1 présente l'évolution des performances de ces deux parties.

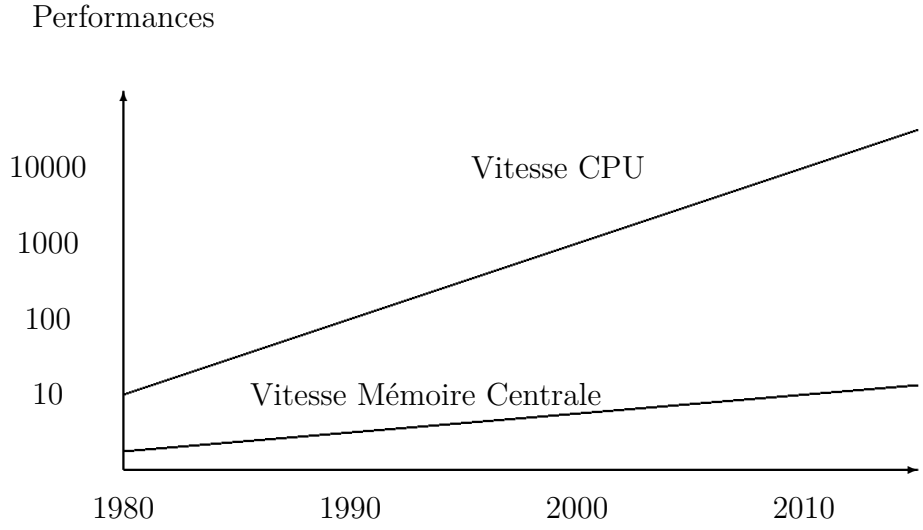


Figure 9.1 – Evolution vitesse microprocesseur - mémoire centrale

9.1.1 Principe

L'écart de performance augmente de 50% d'année en année. La mémoire rapide existe mais elle est beaucoup trop chère pour pouvoir être utilisée de façon extensive. Pour adapter la vitesse du microprocesseur à celle de la mémoire, nous plaçons entre ces deux éléments une mémoire (nommé mémoire cache) rapide qui contient uniquement les instructions et données en cours d'utilisation. Cette mémoire rapide fait le lien entre les deux parties ayant des vitesses de fonctionnement incompatibles. Plus de la moitié de la surface des microprocesseurs actuels

est réservée à cette mémoire cache, ce qui prouve son extrême importance. La figure Fig. 9.2 présente l'architecture de la machine incluant la mémoire cache.

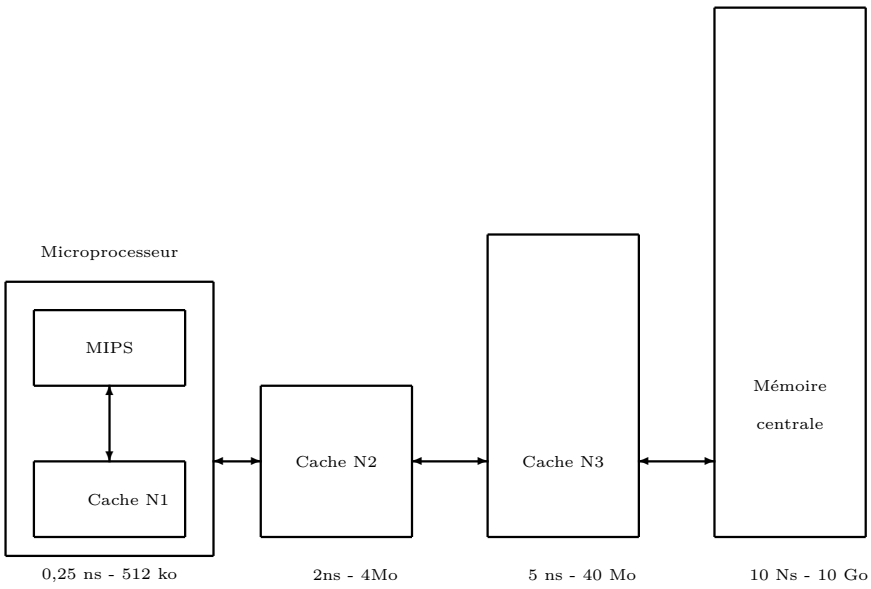


Figure 9.2 – Hiérarchie mémoire

Avec le microprocesseur, nous intégrons un cache nommé de **niveau 1** qui fonctionne à la même vitesse. Il contient un petit nombre d'informations et fonctionne à la même vitesse d'horloge que le microprocesseur. Les caches de **niveau 2** et **niveau 3**, intégrés ou non avec le processeur, sont plus gros, mais plus lents aussi. Vient ensuite la mémoire centrale de l'ordinateur qui est beaucoup plus lente que le microprocesseur. Nous remarquons que plus nous nous éloignons du microprocesseur, plus la taille de la mémoire augmente, et plus sa vitesse diminue.

9.1.2 Localité

Pour organiser les informations stockées dans le cache, nous utiliserons deux principes :

La localité temporelle : les données ou instructions utilisées récemment le seront dans un futur proche (itérations). Des mesures montrent que les microprocesseurs passent la plus grande partie du temps sur un petit nombre d'instructions.

Les données qui viennent d'être utilisées ont une probabilité d'être réutilisées plus grande que celles auxquelles nous n'avons pas fait référence depuis longtemps. La mémoire cache fonctionne en utilisant ce principe. Les données placées dans la cache seront certainement réutilisées dans un futur proche.

La localité spatiale : les données ou instructions proches de celles en cours de traitement seront bientôt utilisées. Les données d'un programme sont stockées dans un espace mémoire limité. Les instructions d'un programme sont traitées séquentielles sauf en cas de branchement.

Un programme accède donc à de petites portions de l'espace mémoire. Pour respecter ce principe, les caches manipulent des **blocs** d'informations et non pas des informations élémentaires (généralement un bloc se compose de quelques kilo-octets). Toutes les tailles des informations sont exprimées sous la forme de puissance de deux.

Remarque 48 :

— Le nombre d'octets par bloc sera nommé NOPB,

— La taille de la mémoire centrale (en octet) sera nommée TMC,
— La taille de la mémoire cache (en octet) sera nommé TC,
— Le nombre de blocs contenus dans la mémoire centrale sera nommé NBMC=$\frac{\text{TMC}}{\text{NOPB}}$,
— Le nombre de blocs contenus dans la mémoire cache sera nommé NBC=$\frac{\text{TC}}{\text{NOPB}}$.

9.2 Placer un bloc dans le cache

La mémoire cache étant beaucoup plus petite que la mémoire centrale, nous devons y placer efficacement les informations. Pour cela, nous utiliserons une **fonction de correspondance** qui spécifie la relation entre le numéro du bloc en mémoire centrale et sa place dans la mémoire cache.
Trois fonctions de correspondance existent :

1. La fonction **directe** aussi nommée *direct mapping*,
2. La fonction **associative** aussi nommée *fully associative*,
3. La fonction **associative par ensemble** aussi nommée *N-way set associative*.

9.2.1 Fonction de correspondance directe

Chaque bloc de la mémoire centrale possède une place **unique** dans le cache. Le bloc x de la mémoire centrale, aura pour place dans le cache le bloc numéro y. La relation est de la forme relation y=f_d(x,NBC).

$$\text{y} = \text{x} \mod \text{NBC} \tag{9.1}$$
$$= \text{x} \,\&\, (\text{NBC} - 1) \tag{9.2}$$

Inconvénient : Si nous prenons deux blocs concurrents de la mémoire centrale, accédés séquentiellement, par exemple le bloc numéro x=0 et le bloc x=NBC, ils se trouveront tous les deux dans le bloc numéro 0 de la mémoire cache. Si nous utilisons l'un puis l'autre, nous devrons recharger chaque fois un nouveau bloc dans le cache, et il ne fonctionnera pas correctement.

$$\begin{aligned} \text{y} &= 0 \,\&\, (\text{NBC} - 1) \\ &= 0 \\ \text{y} &= \text{NBC} \,\&\, (\text{NBC} - 1) \\ &= 0 \end{aligned}$$

Le principe de localité temporelle n'étant pas respecté, la mémoire cache ne fonctionnera pas.

Avantage : Pour savoir si une information se trouve dans la mémoire cache, nous savons précisément où la chercher dans le cache.

Exemple 52 :
Soit une mémoire centrale contenant 16 blocs et un cache contenant 4 blocs. La figure Fig. 9.3 présente les associations possibles entre numéro du bloc mémoire centrale et numéro du bloc dans la mémoire cache. Chaque bloc de la mémoire cache contient un bloc parmi quatre potentiels de la mémoire centrale.

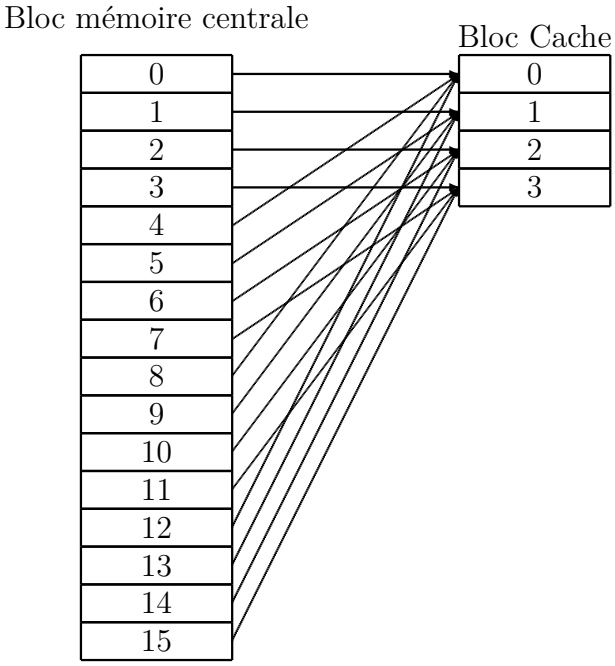


FIGURE 9.3 – Fonction de correspondance directe

9.2.2 Fonction de correspondance associative

Dans ce cas, un bloc x de la mémoire centrale peut se trouver dans n'importe quel bloc de la mémoire cache. Aucune relation n'est définie entre les valeurs de x et de y.

Inconvénient : Pour savoir si un bloc est dans le cache, nous devons parcourir (dans le pire des cas) tous les blocs de celui-ci, ce qui peut être très long. En moyenne, nous devrons parcourir la moitié des blocs du cache pour rechercher l'information recherchée.

Avantage : N'importe quel bloc de la mémoire cache pouvant contenir n'importe quel bloc de la mémoire centrale simultanément, il n'y a plus de problème de concurrence de bloc.

Remarque 49 :

Lorsque la mémoire cache est pleine, nous devons créer un espace en supprimant un bloc. Le principe de localité temporelle nous indique qu'un bloc utilisé récemment à de forte chance de l'être rapidement. Nous devrons donc supprimer le bloc qui n'a pas été utilisé depuis le plus longtemps. Cette politique de remplacement efficace est nommée LRU (less recent used). Une autre politique de remplacement, plus simple à mettre en oeuvre, est basée sur un tirage aléatoire du bloc à supprimer du cache.

9.2.3 Fonction de correspondance associative par ensemble

Dans ce cas, chaque bloc x de la mémoire centrale peut se trouver dans un ensemble de blocs du cache. Pour manipuler les ensembles, nous définissons les variables suivantes ayant pour valeur une puissance de deux :

- Le nombre de blocs par ensemble sera nommé NBPE,
- Le nombre d'ensembles contenus dans la mémoire centrale sera nommé NEMC=$\frac{\text{NBMC}}{\text{NBPE}}$,
- Le nombre d'ensembles contenus dans la mémoire cache sera nommé NEC=$\frac{\text{NBC}}{\text{NBPE}}$.

Nous devons chercher l'information x dans tous les blocs de l'ensemble z. La relation est de la forme z = f_e(x, NEC).

$$z = x \mod NEC \tag{9.3}$$
$$= x \,\&\, (NEC - 1) \tag{9.4}$$

Avantage : Prenons deux blocs concurrents de la mémoire centrale accédés séquentiellement, le bloc numéro x=0 et le bloc x=NEC.

$$\begin{aligned} z &= 0 \,\&\, (NEC - 1) \\ &= 0 \\ z &= NEC \,\&\, (NEC - 1) \\ &= 0 \end{aligned}$$

Ces deux blocs se trouveront tous les deux dans l'ensemble 0 de la mémoire cache, mais dans des blocs différents. Nous devrons alors chercher dans l'ensemble pour savoir si le bloc recherché est présent. Cette recherche, limitée à la taille de l'ensemble, est beaucoup plus rapide que dans le cas de la fonction associative. Cette fonction est très utilisée dans les implémentations de mémoire cache avec des ensembles de 4 ou 8 blocs.

Exemple 53 :

Soit une mémoire centrale contenant 16 blocs et un cache contenant 4 blocs ayant une fonction de correspondance associative par ensemble 2. La figure FIG. 9.4 présente les associations possibles entre blocs mémoire centrale et blocs du cache. Chaque bloc de la mémoire cache contient deux blocs parmi huit potentiels de la mémoire centrale.

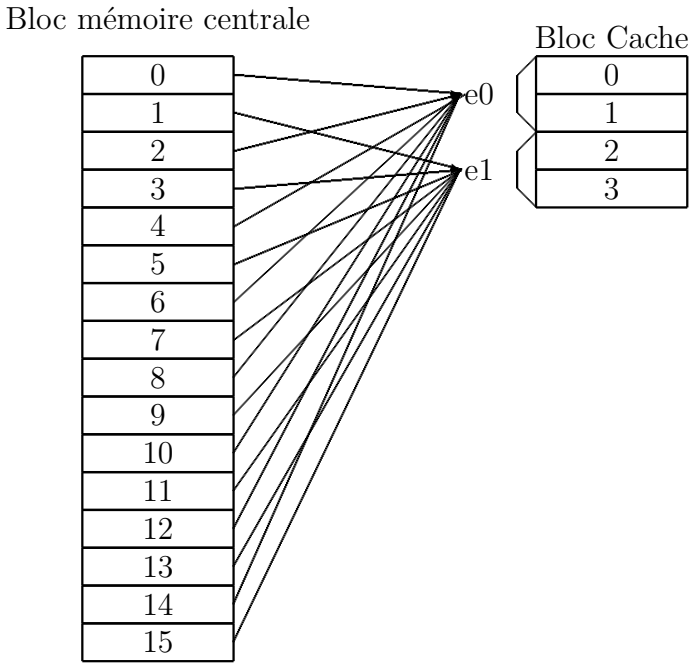


FIGURE 9.4 – Fonction de correspondance directe

Remarque 50 :

— Dans certaines implémentations, les blocs d'un ensemble ne sont pas contigus. Dans l'implémentation du simulateur MARS, l'ensemble e0 sera composé de blocs 0 et 2 et l'ensemble e1 des blocs 1 et 3.

9.3 Trouver un bloc dans le cache

Après avoir placé le bloc dans le cache, nous devons le retrouver. Pour cela nous recherchons sa position dans le cache, puis nous devons l'identifier. Nous utiliserons l'adresse de la mémoire référencée, décomposée en trois champs de bits. La figure FIG. 9.5 montre le position des champs.

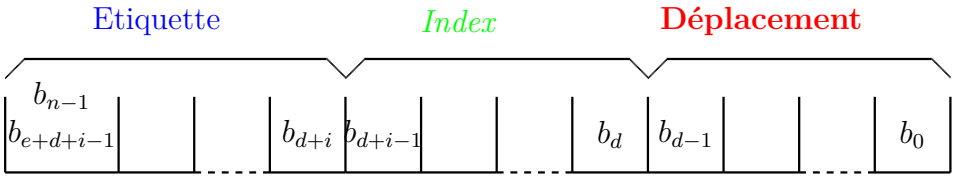


FIGURE 9.5 – Décomposition de l'adresse mémoire

Les champs sont utilisés pour accéder à l'information recherchée.

Le déplacement permet de trouver une information à l'intérieur d'un bloc. La taille d'un bloc étant une puissance de 2, les d bits de gauche de l'adresse permettent de sélectionner une information parmi celles présentes dans le bloc.

L'index permet de savoir dans quel bloc ou ensemble de la mémoire cache se trouve l'information recherchée. L'index dépend de la fonction de correspondance utilisée. Les i bits permettent de repérer l'endroit ou se trouve le bloc recherché dans le cache.

L'étiquette permet de savoir quel bloc de la mémoire centrale se trouve dans un bloc de la mémoire cache. Cette valeur codée sur e bits est stockée dans le cache, en plus des données et pourra ainsi être comparée à l'étiquette du bloc recherché. Si elles sont égales, le bloc est dans la mémoire cache : c'est un succès ; sinon : c'est un défaut de cache.

Exemple 54 :

Soit un système ayant un espace mémoire de 4 giga octets octets (2^{32} octets), et une mémoire cache contenant 4 Méga octets (2^{22} octets). La taille des blocs est fixée à 4 kilo octets (2^{12} octets).

1. Avec la fonction de correspondance directe

   Le déplacement est codé sur 12 bits car les blocs contiennent 4 Kilo octets (2^{12}),

   L'index est codé sur 10 bits car le cache contient 1024 blocs (2^{10}),

   L'étiquette est donc codée sur $32 - 10 - 12 = 10$ bits, ce qui signifie qu'un bloc du cache contient un des 1024 (2^{10}) blocs de la mémoire centrale.

2. Avec la fonction de correspondance associative

   Le déplacement est codé sur 12 bits car les blocs contiennent 4 Kilo octets (2^{12}),

   L'index est codé sur 0 bit car non utilisé,

   L'étiquette est donc codée sur $32 - 12 = 20$ bits, ce qui signifie qu'un bloc du cache contient un des 2^{20} blocs de la mémoire centrale.

3. Avec la fonction associative par ensemble de 4 blocs :

   Dans la mémoire cache il y a $\frac{1024}{4} = 256$ ensembles

   Le déplacement est codé sur 12 bits car les blocs contiennent 4Ko (2^{12}),

   L'index est codé sur 8 bits car le cache contient 256 ensembles (2^8),

   L'étiquette est donc codée sur $32 - 8 - 12 = 12$ bits, ce qui signifie qu'un bloc du cache contient un des 2^{10} blocs de la mémoire centrale.

9.4 Implémentation

9.4.1 Fonction de correspondance directe

L'implémentation de la mémoire cache présentée sur la figure FIG 9.6 correspond à un cache ayant une fonction de correspondance directe. La mémoire cache contient quatre blocs. Chaque bloc contient n données, l'étiquette du bloc dans le cache, et un bit de validité.

L'index permet de spécifier numéro du bloc de la mémoire cache où chercher la donnée. L'étiquette présente dans ce bloc est comparée à celle recherchée. Si celles-ci sont identiques et que le bit de validité est positionné à vrai, la recherche dans le cache est un succès. La donnée recherchée sera sélectionnée à l'aide d'un multiplexeur n → 1, est envoyée sur la sortie nommée donnée.

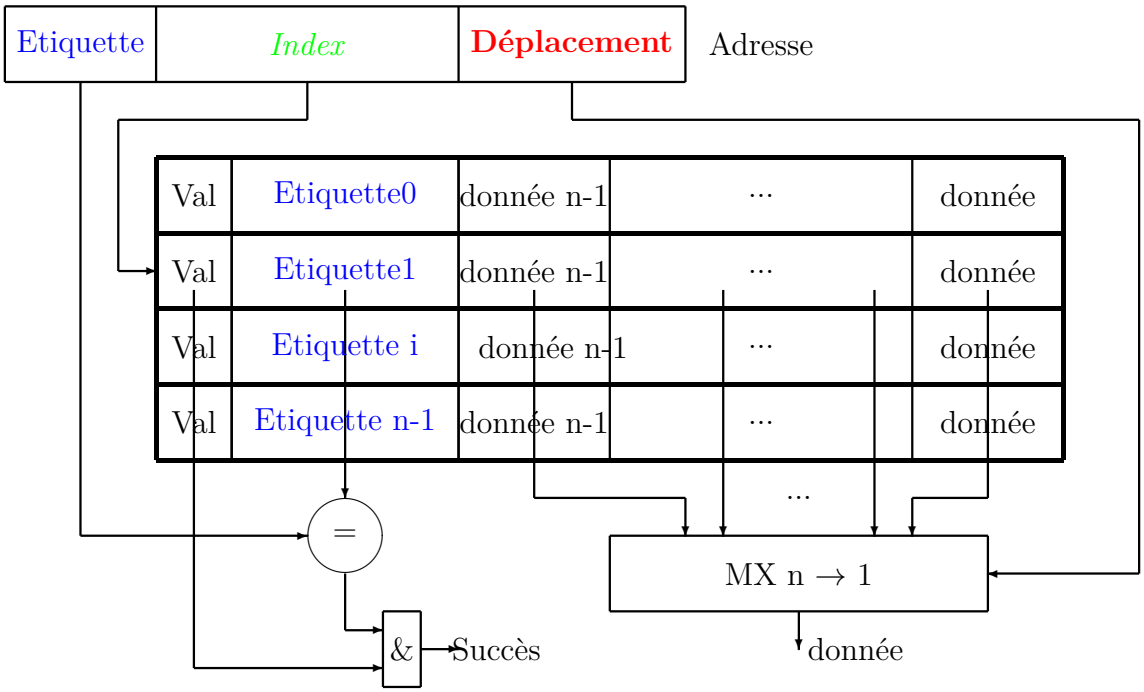


FIGURE 9.6 – Implémentation d'une mémoire cache directe

9.4.2 Fonction de correspondance associative par ensemble

L'implémentation de la mémoire cache présentée sur la figure FIG 9.7 correspond à un cache ayant une fonction de correspondance associative par ensemble de deux blocs. Le cache contient deux ensembles. Chaque ensemble contient $2 \times n$ données, deux étiquettes des blocs présents dans le cache, et deux bits de validité.

L'index permet de spécifier l'ensemble ou chercher la donnée. Les étiquettes présentes dans l'ensemble sont comparées à celle recherchée. Si une de celles-ci est identique et que son bit de validité est positionné à vrai, la recherche dans le cache est un succès. Le bloc recherché sera sélectionné à l'aide d'un multiplexeur 2 → 1. La donnée recherchée sera sélectionnée à l'aide d'un multiplexeur n → 1, est envoyée sur la sortie nommée donnée.

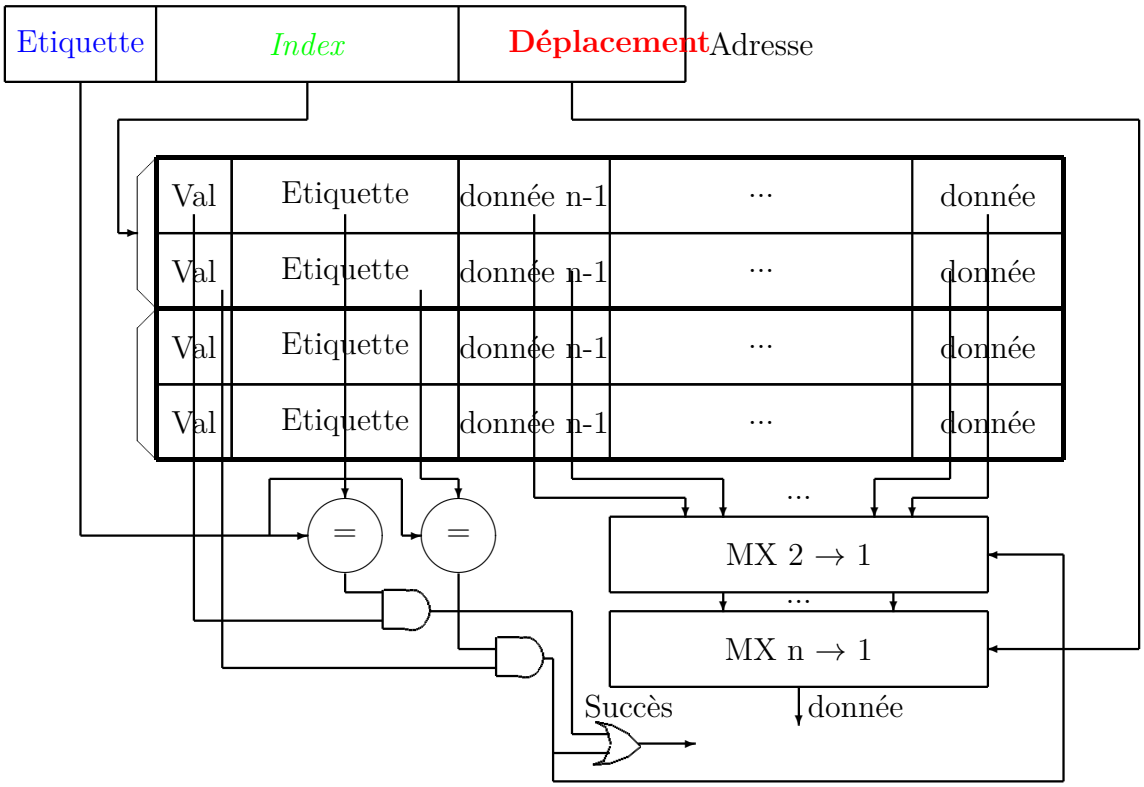


FIGURE 9.7 – Implémentation d'une mémoire cache associative par ensemble de deux

9.5 Exercices

Exercice 13 : Théorique : Performance des caches

Question 1 : Cache pédagogique Nous utiliserons les fonctions *initMem* et *copieMem* de la partie précédente et un cache théorique contenant 32 octets. La taille des adresses est de 32 bits. La taille des blocs est de 4, 8, 16 et 32 octets. La fonction de correspondance sera

1. directe,
2. associative,
3. associative par ensemble de deux blocs.

Pour chaque fonction de correspondance, pour chaque taille du bloc donnez :

1. Le nombre de blocs (éventuellement d'ensembles), la taille du déplacement, de l'index et de l'étiquette exprimés en bits
2. Le taux de succès lors des accès à la mémoire cache.

Indication 1 :

— La taille des zones mémoire manipulées par fonctions *initMem* et *copieMem* est de 32 octets.
— Les données de la fonction *initMem* débutent à l'adresse **0x**1001 0000 et se terminent à l'adresse **0x**1001 001F.
— Celles de la fonction *copieMem* débutent à l'adresse **0x**1001 0000 et se terminent à l'adresse **0x**1001 001F pour la source et débutent à l'adresse **0x**1001 0020 et se terminent à l'adresse **0x**1001 003F pour la destination.
— Lorsque nous devons supprimer un bloc du cache, nous utiliserons la technique LRU (less recent used)

— Au démarrage du programme, le cache est vide.
— Lors d'un accès à la mémoire (instruction sw ou lw), si la donnée recherchée est dans le cache c'est un succès d'accès au cache, sinon c'est un défaut d'accès au cache.
— Le taux de succès est relatif à tous les accès mémoire effectués par la fonction *initMem* puis par la fonction *copieMem*.
— Le taux de succès est défini de la façon suivante :

$$\text{taux succès} = \frac{\text{nombre d'accès réussi à la mémoire cache}}{\text{nombre d'accès à la mémoire cache}} \tag{9.5}$$

Question 2 : Cache réel Nous utiliserons un cache de 256 kilo octets, organisé en bloc de 8 kilo octets, utilisant les fonctions de correspondance directe et associative par ensemble de 4. Nous utiliserons les fonctions *initMem* et *copieMem* pour traiter 512 kilo octet de données. Les données des fonctions débutent à l'adresse **0x**1001 0000 et sont consécutives (la destination suis directement la source).
Pour chaque fonction de correspondance donnez :

1. Le nombre de blocs (éventuellement d'ensembles), la taille du déplacement, de l'index et de l'étiquette exprimés en bits
2. Le taux de succès lors des accès à la mémoire cache.

Correction exercice 13 :

Question 1 : Cache pédagogique La figure FIG 9.8 présente les taux de succès pour une fonction de correspondance directe.

Taille blocs	4	8	16	32
nombre de blocs	8	4	2	1
déplacement	2	3	4	5
index	3	2	1	0
étiquette	27	27	27	27
succès *initMem*	$\frac{0}{8}$	$\frac{4}{8}$	$\frac{6}{8}$	$\frac{7}{8}$
succès *copieMem*	$\frac{0}{16}$	$\frac{0}{16}$	$\frac{0}{16}$	$\frac{0}{16}$

FIGURE 9.8 – Fonction de correspondance directe

Exemple du calcul de la taille des champs pour la fonction de correspondance directe avec des blocs de 8 octets :

- **Nombre de blocs dans le cache :** $\frac{32}{8} = 4$
- **Taille du déplacement** : Les blocs contenant $8 = 2^3$, nous avons donc besoin 3 bits pour identifier une information dans le cache.
- *Taille de l'index* : Dans le cache, nous avons 4 blocs (2^2), nous avons donc besoin de 2 bits pour identifier la place d'un bloc dans le cache.
- Taille de l'étiquette : Les adresses étant codées sur 32 bits, nous avons donc $32-2-3 = 27$ bits à comparer pour déterminer si le bloc dans le cache est celui recherché.

Exemple du calcul du taux de succès pour la fonction de correspondance directe avec des blocs de 8 octets pour la fonction *initMem* :

Accès en écriture aux 32 bits mémoire d'adresse :

- **0x**1001 0000 (**0b**0001 0000 0000 0001 0000 0000 000*0 0***000**)
- **0x**1001 0003 (**0b**0001 0000 0000 0001 0000 0000 000*0 0***011**)

La donnée n'est pas présente dans le bloc numéro 0 du cache indiqué par l'index car le cache est vide. Les 8 octets du bloc (de **0x**1001 0000 à **0x**1001 0007) sont chargés dans le bloc numéro 0 spécifié par l'index. La donnée n'étant pas dans le cache, **c'est un défaut d'accès cache.**

Accès en écriture aux 32 bits mémoire adresse :

- **0x**1001 0004 (**0b**0001 0000 0000 0001 0000 0000 000*0 0***100**)
- **0x**1001 0007 (**0b**0001 0000 0000 0001 0000 0000 000*0 0***111**)

Les 4 octets recherchés (de **0x**1001 0004 à **0x**1001 0007) sont présents dans le bloc numéro 0 spécifié par l'index car l'étiquette de la donnée est la même que celle stockée dans le cache. **C'est un succès d'accès cache.**

Accès en écriture aux 32 bits mémoire d'adresse

- **0x**1001 0008 (**0b**0001 0000 0000 0001 0000 0000 000*0 1***000**)
- **0x**1001 000B (**0b**0001 0000 0000 0001 0000 0000 000*0 1***011**)

La donnée n'est pas présente dans le bloc numéro 1 du cache indiqué par l'index car le cache est vide. Les 8 octets du bloc (de **0x**1001 0008 à **0x**1001 000F) sont chargés dans le bloc numéro 1 spécifié par l'index. La donnée n'étant pas dans le cache, **c'est un défaut d'accès cache.**

Accès en écriture aux 32 bits mémoire d'adresse

- **0x**1001 000C (**0b**0001 0000 0000 0001 0000 0000 000*0 1***100**)

— **0x**1001 000F (**0b**0001 0000 0000 0001 0000 0000 000*0 1* **111**)

Les 4 octets recherchés (de **0x**1001 000C à **0x**1001 000F) sont présents dans le bloc numéro 1 spécifié par l'index car l'étiquette de la donnée est la même que celle stockée dans le cache. **C'est un succès d'accès cache.**

...

Dans ce cas, pour la fonction *initMem*, 8 accès sont effectués, un accès au cache sur deux est un succès, donc le taux de succès est de 50%.

Exemple du calcul du taux de succès pour la fonction de correspondance directe avec des blocs de 8 octets pour la fonction *copieMem* :

Accès en lecture aux 32 bits mémoire d'adresse

— **0x**1001 0000 (**0b**0001 0000 0000 0001 0000 0000 000*0 0* **000**)

— **0x**1001 0003 (**0b**0001 0000 0000 0001 0000 0000 000*0 0* **011**)

Les 8 octets du bloc (de **0x**1001 0000 à **0x**1001 0007) sont chargés dans le bloc numéro 0. La donnée n'étant pas dans le cache, **c'est un défaut d'accès cache.**

Accès en écriture aux 32 bits mémoire d'adresse

— **0x**1001 0020 (**0b**0001 0000 0000 0001 0000 0000 001*0 0* **000**)

— **0x**1001 0023 (**0b**0001 0000 0000 0001 0000 0000 001*0 0* **011**)

Les 8 octets du bloc (de **0x**1001 0020 à **0x**1001 0027) sont chargés dans le bloc numéro 0 spécifié par l'index car l'étiquette du bloc actuellement dans le cache n'est pas la bonne. La donnée n'étant pas dans le cache, c'est un défaut d'accès cache.

...

Dans ce cas, pour la fonction *copieMem*, 16 accès au cache sont réalisés, tous les accès au cache sont des défauts, donc le taux de succès est de 0%. Ceci est du à a concurrence des données utilisées.

Remarque 51 :

— Dans le cas d'un bloc de taille 4 octets, le principe de localité spatiale n'est pas respecté, car une seule donnée est chargée dans un bloc, le taux de succès est toujours à 0.

— Pour la fonction nommée *initMem* le cache fonctionne correctement. Plus la taille du bloc est grande, plus nous chargeons de données simultanément, plus le taux de succès est grand.

— Pour la fonction nommée *copieMem* le cache ne fonctionne pas. Les données source et destination sont en concurrence, c'est-à-dire chargée dans le même bloc du cache.

La figure FIG 9.9 présente les taux de succès pour une fonction de correspondance associative.

Taille blocs	4	8	16	32
nombre de blocs	8	4	2	1
déplacement	2	3	4	5
index	0	0	0	0
étiquette	30	29	28	27
succès *initMem*	$\frac{0}{8}$	$\frac{4}{8}$	$\frac{6}{8}$	$\frac{7}{8}$
succès *copieMem*	$\frac{0}{16}$	$\frac{8}{16}$	$\frac{12}{16}$	$\frac{0}{16}$

FIGURE 9.9 – Fonction de correspondance associative

Exemple du calcul de la taille des champs pour la fonction de correspondance associative avec des blocs de 8 octets :

— **Nombre de blocs dans le cache :** $\frac{32}{8} = 4$

— **Taille du déplacement** : Les blocs contenant $8 = 2^3$, nous avons donc besoin 3 bits pour identifier une information dans le cache.
— *Taille de l'index* : 0 car un bloc n'a pas de place définie.
— Taille de l'étiquette : Les adresses étant codées sur 32 bits, nous avons donc $32 - 3 = 29$ bits à comparer pour déterminer si le bloc dans le cache est celui recherché.

Exemple du calcul du taux de succès pour la fonction de correspondance associative avec des blocs de 8 octets pour la fonction *copieMem* :

Accès en lecture aux 32 bits mémoire d'adresse

— **0x**1001 0000 (**0b**0001 0000 0000 0001 0000 0000 0000 0**000**)
— **0x**1001 0003 (**0b**0001 0000 0000 0001 0000 0000 0000 0**011**)

Les 8 octets du bloc (de **0x**1001 0000 à **0x**1001 0007) sont chargés dans le premier bloc libre (numéro 0). La donnée n'étant pas dans le cache, c'est un **défaut d'accès cache**.

Accès en écriture aux 32 bits mémoire d'adresse :

— **0x**1001 0020 (**0b**0001 0000 0000 0001 0000 0000 0010 0**000**)
— **0x**1001 0023 (**0b**0001 0000 0000 0001 0000 0000 0010 0**011**)

Les 8 octets du bloc (de **0x**1001 0020 à **0x**1001 0027) sont chargés dans le premier bloc libre (numéro 1). La donnée n'étant pas dans le cache, **c'est un défaut d'accès cache**.

Accès en lecture aux 32 bits mémoire d'adresse

— **0x**1001 0004 (**0b**0001 0000 0000 0001 0000 0000 0000 0**100**)
— **0x**1001 0007 (**0b**0001 0000 0000 0001 0000 0000 0000 0**111**)

Les 4 octets recherchés(de **0x**1001 0004 à **0x**1001 0007) sont présents dans le bloc numéro 0 car l'étiquette est la bonne. La donnée est dans le cache, **c'est un succès d'accès cache**.

Accès en écriture aux 32 bits mémoire d'adresse

— **0x**1001 0024 (**0b**0001 0000 0000 0001 0000 0000 0010 0**100**)
— **0x**1001 0027 (**0b**0001 0000 0000 0001 0000 0000 0010 0**111**)

Les 4 octets recherchés (de **0x**1001 0024 à **0x**1001 0027) sont présents dans le bloc numéro 1. La donnée est dans le cache, c'est un succès d'accès cache.

Lorsque le cache est plein, nous supprimons les plus anciennes données qui ne sont plus utilisées.

Dans ce cas, pour la fonction *copieMem*, 16 accès au cache sont réalisés, un accès sur deux est un succès, donc le taux de succès est de 50%.

Remarque 52 :

— Dans le cas d'un bloc de taille 4 octets, le principe de localité spatiale n'est pas respecté, une seule donnée est chargée dans un bloc.
— Pour la fonction nommée *initMem* le cache fonctionne correctement. Plus la taille du bloc est grande, plus nous chargeons de données simultanément, plus le taux de succès est grand.
— Pour la fonction nommée *copieMem* le cache fonctionne correctement. Les données source et destination sont en concurrence, mais placées n'importe où. La recherche d'un bloc est longue (dans le pire des cas, il faut chercher dans tous les blocs du cache)
— Dans le cas où il n'y à qu'un bloc dans le cache, le principe du cache n'est pas respecté, ce qui conduit à un non fonctionnement.

La figure Fig 9.10 présente les taux de succès pour une fonction de correspondance associative.

Taille blocs	4	8	16	32
nombre de blocs	8	4	2	1
nombre d'ensemble	4	2	1	
déplacement	2	3	4	
index	2	1	0	
étiquette	28	28	28	
succès initMem	$\frac{0}{8}$	$\frac{4}{8}$	$\frac{6}{8}$	
succès copieMem	$\frac{0}{16}$	$\frac{8}{16}$	$\frac{12}{16}$	

FIGURE 9.10 – Fonction de correspondance associative par ensemble de deux

— **Nombre de blocs dans le cache :** $\frac{32}{8} = 4$
— **Nombre d'ensemble dans le cache :** $\frac{4}{2} = 2$
— **Taille du déplacement** : Les blocs contenant $8 = 2^3$, nous avons donc besoin 3 bits pour identifier une information dans le cache.
— *Taille de l'index* : Dans le cache, nous avons 2 ensemble (2^1), nous avons donc besoin de 1 bits pour identifier la place d'un bloc dans le cache.
— Taille de l'étiquette : Les adresses étant codées sur 32 bits, nous avons donc $32-1-3 = 28$ bits à comparer pour déterminer si le bloc dans le cache est celui recherché.

Exemple du calcul du taux de succès pour la fonction de correspondance associative par ensemble de deux avec des blocs de 8 octets pour la fonction *copieMem* :

Accès en lecture aux 32 bits mémoire d'adresse

— **0x**1001 0000 (**0b**0001 0000 0000 0001 0000 0000 0000 *0***000**)
— **0x**1001 0003 (**0b**0001 0000 0000 0001 0000 0000 0000 *0***011**)

Les 8 octets du bloc (de **0x**1001 0000 à **0x**1001 0007) sont chargés dans l'ensemble numéro 0 spécifié par l'index, dans le premier bloc de l'ensemble. La donnée n'étant pas dans le cache, **c'est un défaut d'accès cache**.

Accès en écriture aux 32 bits mémoire d'adresse

— **0x**1001 0020 (**0b**0001 0000 0000 0001 0000 0000 0010 *0***000**)
— **0x**1001 0023 (**0b**0001 0000 0000 0001 0000 0000 0010 *0***011**)

Les 8 octets du bloc (de **0x**1001 0020 à **0x**1001 0027) sont chargés dans l'ensemble numéro 0 spécifié par l'index, dans le deuxième bloc de l'ensemble car le premier est occupé. La donnée n'étant pas dans le cache, **c'est un défaut d'accès cache**.

Accès en lecture aux 32 bits mémoire d'adresse

— **0x**1001 0004 (**0b**0001 0000 0000 0001 0000 0000 0000 *0***100**)
— **0x**1001 0007 (**0b**0001 0000 0000 0001 0000 0000 0000 *0***111**)

Les 4 octets recherchés(de **0x**1001 0004 à **0x**1001 0007) sont présents dans le premier bloc de l'ensemble numéro 0. La donnée est dans le cache, **c'est un succès d'accès cache**.

Accès en écriture aux 32 bits mémoire d'adresse

— **0x**1001 0024 (**0b**0001 0000 0000 0001 0000 0000 0010 *0***100**)
— **0x**1001 0027 (**0b**0001 0000 0000 0001 0000 0000 0010 *0***111**)

Les 4 octets recherchés (de **0x**10010024 à **0x**10010027) sont présents dans le deuxième bloc de l'ensemble numéro 0. La donnée est dans le cache, **c'est un succès d'accès cache**.

Quand un ensemble est plein, nous supprimons le bloc qui n'a pas été utilisé depuis le plus longtemps.

Dans ce cas, pour la fonction *copieMem*, 16 accès au cache sont réalisés, un accès sur deux est un succès, donc le taux de succès est de 50%.

Remarque 53 :

— Dans le cas d'un bloc de taille 4 octets, le principe de localité spatiale n'est pas respecté, une seule donnée est chargée dans un bloc.
— Pour la fonction nommée *initMem* le cache fonctionne correctement. Plus la taille du bloc est grande, plus nous chargeons de données simultanément, plus le taux de succès est grand.
— Pour la fonction nommée *copieMem* le cache fonctionne correctement. Les données source et destination sont en concurrence, mais placées n'importe où. La recherche d'un bloc est limité à un ensemble.
— Dans le cas où il n'y à qu'un bloc dans le cache, nous ne pouvons pas créer d'ensemble

Question 2 : Cache réel Les figures FIG 9.11 et FIG 9.12 présentent les taux de succès pour les fonctions *initMem* et *copieMem*.

nombre de blocs	$\frac{256}{8} = 32$
déplacement	$8 \times 2^{10} = 2^{13} \rightarrow 13$ bits
index	$32 = 2^5 \rightarrow 5$ bits
étiquette	$32 - 13 - 5 = 14$ bits
succès *initMem*	$\frac{\frac{2^{13}}{2^2}-1}{\frac{2^{13}}{2^2}} \approx 1$
succès *copieMem*	$\frac{0}{\frac{2^{13}}{2^2}}$

FIGURE 9.11 – Fonction de correspondance directe

nombre de blocs	$\frac{256}{8} = 32$
nombre d'ensembles	$\frac{32}{4} = 8$
déplacement	$8 \times 2^{10} = 2^{13} \rightarrow 13$ bits
index	$8 = 2^3 \rightarrow 3$ bits
étiquette	$32 - 13 - 3 = 16$
succès *initMem*	$\frac{\frac{2^{13}}{2^2}-1}{\frac{2^{13}}{2^2}} \approx 1$
succès *copieMem*	$\frac{\frac{2^{13}}{2^2}-1}{\frac{2^{13}}{2^2}} \approx 1$

FIGURE 9.12 – Fonction de correspondance associative par ensemble de 4

Exercice 14 : Pratique : Performances des caches

Question 1 : Cache pédagogique Nous allons vérifier les résultats obtenus dans les exercices théoriques en utilisant l'outil nommé **Data Cache Simulator** intégré au simulateur **MARS**.

Indication 1 :

— Dans tous les cas, la taille du cache est de 32 octets. Pour cela ajuster la taille d'un bloc (ici spécifiée en mots de 32 bits) et le nombre de blocs de la mémoire cache

— Dans le cas d'un cache associatif par ensemble (N way Set associatif), le paramètre de l'outil **Data Cache Simulator** nommé **Set Size** permet de définir la taille de l'ensemble en nombre de blocs.
— Vous cocherez la case **enable** de la partie **Runtime log** et observerez les messages affichés.

1. Pour la fonction de correspondance directe, analysez en détails les messages affichés dans **Runtime log** ?
2. Pour la fonction de correspondance associative, analysez en détails les messages affichés dans **Runtime log** ?
3. Pour la fonction de correspondance associative par ensemble de 2, analysez en détails les messages affichés dans **Runtime log** ?
4. Pour la fonction de correspondance associative par ensemble de 2, analysez en détails les messages affichés dans **Runtime log** si l'on utilise la technique aléatoire pour remplacer un bloc. Lancez plusieurs fois la fonction est observez le taux de succès.

Question 2 : Cache réels

1. Modifiez le programme pour traiter 512 kilo octet de mémoire ?
2. En utilisant un cache de 256 ko organiser en blocs de 8ko vérifiez les taux de succès pour les deux fonctions de correspondance (directe et associative par ensemble de 4). Vous utiliserez la politique de remplacement LRU (least recent use et random). Que remarquez vous dans le cas du random ?

Correction exercice 14 :

Question 1 : Cache pédagogique

1. Fonction directe

   Pour la taille des blocs 8 octets nous obtenons les messages suivants pour la fonction *initMem* :

   Signification des messages :

   — *(1)address : 0x10010000 (tag 0x00800800) block range : 0-0 trying block 0 empty – MISS* : Recherche dans le bloc 0 de la donnée d'adresse **0x**1001 0000 ayant pour l'étiquette la valeur **0x**0080 0800. Le bloc 0 étant vide, c'est un défaut d'accès au cache.

   — *(2) address : 0x10010004 (tag 0x00800800) block range : 0-0 trying block 0 tag 0x00800800 – HIT.* Recherche dans le bloc 0 de la donnée d'adresse **0x**1001 0004 ayant pour l'étiquette la valeur **0x**0080 0800. Dans le bloc 0, il y à une information ayant la même étiquette, c'est un succès d'accès au cache.

   Nous retrouvons le taux de succès de 50

   Pour la taille des bloc 8 octets nous obtenons les messages suivants pour la fonction *copieMem* :

   Signification des messages :

   — *(2) address : 0x10010020 (tag 0x00800801) block range : 0-0 trying block 0 tag 0x00800800 – OCCUPIED MISS due to FULL SET.* Dans ce cas les étiquettes du bloc présent dans le cache et celle de l'information recherchée sont différente. La fonction de correspondance étant direct, nous recherchons l'information dans un seul bloc (bloc range 0-0), le numéro 0. Le message indique que le bloc cache est occupé avec une autre donnée et que le défaut est causé par le l'ensemble plein. Pour la fonction de correspondance directe la taille de l'ensemble vaut 1.

2. associative

   Pour la taille des bloc 8 octets nous obtenons les messages suivants pour la fonction *initMem* :

   Signification des messages :

   — *(7) address : 0x10010018 (tag 0x02002003) block range : 0-3 trying block 0 tag 0x02002000 – OCCUPIED trying block 1 tag 0x02002001 – OCCUPIED trying block 2 tag 0x02002002 – OCCUPIED trying block 3 empty – MISS* : Dans ce cas le taille nous recherchons dans l'ensemble du cache l'information. Nous devons parcourir tous les blocs du cache pour placer placer le bloc dans le premier libre (ici le dernier, bloc numéro 3).

   — *(8) address : 0x1001001c (tag 0x02002003) block range : 0-3 trying block 0 tag 0x02002000 – OCCUPIED trying block 1 tag 0x02002001 – OCCUPIED trying block 2 tag 0x02002002 – OCCUPIED trying block 3 tag 0x02002003 – HIT.* Nous devons rechercher, ici dans tous les blocs du cache pour trouver celui dont l'étiquette et celle recherchée.

3. associative par ensemble de 2

   Pour la taille des bloc 8 octets nous obtenons les messages suivants pour la fonction *copieMem* :

   (13) address : 0x10010018 (tag 0x01001001) block range : 2-3 trying block 2 tag 0x01001000 – OCCUPIED trying block 3 tag 0x01001002 – OCCUPIED MISS due to FULL SET – LRU replace block 2 ; unused since (7) (14) address : 0x10010038 (tag

0x01001003) block range : 2-3 trying block 2 tag 0x01001001 – OCCUPIED trying block 3 tag 0x01001002 – OCCUPIED MISS due to FULL SET – LRU replace block 3 ; unused since (8) (15) address : 0x1001001c (tag 0x01001001) block range : 2-3 trying block 2 tag 0x01001001 – HIT

Signification des messages :

— *(13) address : 0x10010018 (tag 0x01001001) block range : 2-3 trying block 2 tag 0x01001000 – OCCUPIED trying block 3 tag 0x01001002 – OCCUPIED MISS due to FULL SET – LRU replace block 2; unused since (7)* : Dans ce cas le taille nous recherchons dans l'ensemble de l'ensemble l'information. Nous devons parcourir les blocs 2 et 3 du cache pour placer placer le bloc. L'ensemble étant plein, nous placerons celui-ci dans celui qui n'a pas été utilisé depuis le plus longtemps.

— *(16) address : 0x1001003c (tag 0x01001003) block range : 2-3 trying block 2 tag 0x01001001 – OCCUPIED trying block 3 tag 0x01001003 – HIT*. La recherche si limite à la taille de l'ensemble.

4. associative par ensemble de 2, aléatoire pour remplacer un bloc.

   Le taux de succès est variable, car nous pouvons effacer un bloc qui sera encore utilisé par la suite. Ce politique n'est pas optimale.

Question 2 : Cache réels

1. Modifiez le programme pour traiter 512 kilo octet de mémoire ?

Listing 9.1 – InitMem et copieMem avec 512 kilo octets de données

```
1  .data
2    source : .space 0x80000
3    destination : .space 0x80000
4  .text
5  main:
6    #Test InitMem
7    la $a0,source
8    addi $a1,$zero,0x20000
9    addi $a2,$zero,0xFF00FF00
10   jal InitMem
11   #Test CopieMem
12   la $a0,source
13   la $a1,destination
14   addi $a2,$zero,0x20000
15   #jal CopieMem
16   # -- exit
17   addi $v0, $zero, 10
18   syscall
19
20 # Fonction Initialisation zone memoire
21 # $a0 adresse zone memoire
22 # $a1 taille de le zone memoire
23 # $a2 valeur a placer en memoire
24 InitMem:
25   beq $a1,$zero,InitMem_Fin
26   sw $a2,0($a0)
27   addi $a1,$a1,-1
28   addi $a0,$a0,4
29   j InitMem
30 InitMem_Fin:
31   jr $ra
32
33 # Fonction Copie zone memoire
```

```
34 # $a0 adresse zone memoire src
35 # $a1 adresse zone memoire dest
36 # $a2 taille de le zone memoire
37 CopieMem:
38 CopieMem_boucle:
39    beq $a2,$zero,CopieMem_Fin
40    lw $t0,0($a0)
41    sw $t0,0($a1)
42    addi $a2,$a2,-1
43    addi $a0,$a0,4
44    addi $a1,$a1,4
45    j CopieMem_boucle
46 CopieMem_Fin:
47    jr $ra
```

2. Vérifier les taux de succès pour les deux fonctions de correspondance ?

   Dans ce cas, la taille du bloc est de 2048 mots de 32 bits (8 kilos octets), et le bloc contient 32 bloc. Pour vérification, la taille du cache est de 262144 octets. Nous retrouvons les taux de succès prévu dans les exercices théoriques.

Chapitre 10

Le pipeline d'instructions

10.1 Principe

Le traitement d'une instruction nécessite un ensemble d'opérations effectués séquentiellement :

1. Lire l'instruction dans la mémoire de code et calculer l'adresse de l'instruction suivante.
2. Décoder l'instruction et **rechercher les opérandes de type registre** dans le banc de registres du MIPS.
3. Effectuer l'opération arithmétique ou logique dans le circuit nommé "unité arithmétique et logique".
4. Suivant le cas, accéder à la mémoire de données pour lire ou écrire une information.
5. En cas de branchement, mettre l'adresse de l'instruction suivante dans le registre compteur de programme.
6. Suivant le cas, ranger le résultat dans le banc de registres du MIPS.

La figure Fig 10.1 présente ce principe de fonctionnement.

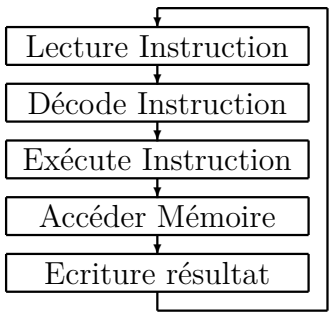


Figure 10.1 – Traitement séquentiel d'une instruction

Pour accélérer le fonctionnement, le principe est de traiter simultanément plusieurs instructions. Ces instructions seront à des étapes différentes de leur traitement. De cette façon, le temps de traitement global (aussi appelé **latence**) reste le même, mais les instructions sortent plus rapidement. Ce principe est nommé **Pipeline d'instructions**.
Dans l'architecture MIPS, le traitement de l'instruction est décomposé en cinq **étages** :

Etage LI : Lire l'instruction dans la mémoire de code et calculer l'adresse de l'instruction suivante.

Etage DI : Décoder l'instruction et rechercher les opérandes dans le banc de registres du MIPS.

Etage EX : Effectuer l'opération dans l'UAL : calculer le résultat ou calculer l'adresse ou tester deux valeurs suivant l'instruction à traiter. En cas de branchement, calculer l'adresse de l'instruction suivante.

Etage M : Accéder à la mémoire de données en lecture ou en écriture. En cas de branchement préparer l'adresse de l'instruction suivante.

Etage ER : Écriture du résultat dans le jeu de registres et en cas de branchement charger l'instruction cible.

La figure FIG 10.2 présente ce principe de fonctionnement

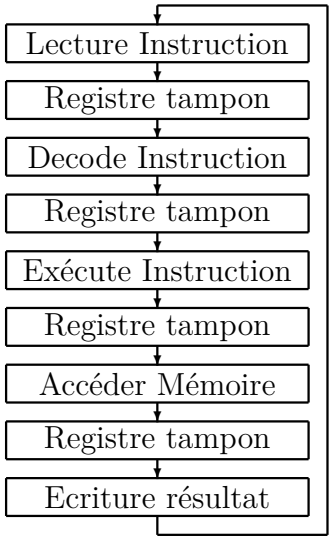


FIGURE 10.2 – Principe du pipeline d'instruction

Entre les étages du pipeline sont ajoutés des **registres tampons** qui assurent l'indépendance entre les instructions placées dans le pipeline. Les données de l'instruction dans l'étage sont stockées dans le registre tampon précédent et les résultats de l'instruction dans l'étage sont stockés dans le registre tampon suivant.
Si une version du processeur non pipeliné traite une instruction en un temps nommé TSP , le temps de traitement d'une instruction sur un processeur pipeliné TP contenant n étages sera calculé en utilisant la relation suivante :

$$\text{TP} = \frac{\text{TSP}}{n} \tag{10.1}$$

L'introduction d'un registre tampon augmente le temps de l'instruction dans l'étage. Si nous créons un pipeline avec beaucoup d'étages (aussi nommé **pipeline profond**) le temps passé dans l'étage devient faible et le temps du stockage/lecture des informations dans le registre tampon ne doit pas devenir prépondérant. En tenant compte du temps passé dans le registre tampon TT, la relation devient donc :

$$\text{TP} = \frac{\text{TSP}}{n} + \text{TT} \tag{10.2}$$

La latence est modifiée par l'introduction de registre tampon de la façon suivante :

$$\begin{aligned} \text{Latence} &= n \times TP \\ &= n \times (\frac{\text{TSP}}{n} + \text{TT}) \end{aligned} \tag{10.3}$$

Soit la suite d'instructions :

```
addi $t0 , $zero , 1
addi $t1 , $zero , 2
addi $t2 , $zero , 3
addi $t3 , $zero , 4
addi $t4 , $zero , 5
```

En considérant que les instructions présentes dans chaque étage du pipeline soient exécutées en un **cycle d'horloge** on obtient le diagramme de déroulement temporel. A l'intersection d'une ligne (instruction) et d'une colonne (numéro de cycle) on obtient l'étage dans lequel se trouve l'instruction.

Instruction / Cycle	1	2	3	4	5	6	7	8	9
addi $t0 , $zero , 1	LI	DI	EX	M	ER				
addi $t1 , $zero , 2		LI	DI	EX	M	ER			
addi $t2 , $zero , 3			LI	DI	EX	M	ER		
addi $t3 , $zero , 4				LI	DI	EX	M	ER	
addi $t4 , $zero , 5					LI	DI	EX	M	ER

FIGURE 10.3 – Diagramme temporel régulier des instructions précédentes

Le processeur sans pipeline utilise 5 cycles d'horloge pour effectuer une instruction, il faudra $5 \times 5 = 25$ cycles pour effectuer les cinq instructions.
Avec le processeur pipeliné, une instruction sort du MIPS tous les cycles d'horloge, il faudra cinq cycles pour la première et un cycle pour les instructions suivantes soit : $5 + 4 = 9$ cycles
Le gain réel sur cette séquence d'instructions est de $\frac{25}{9} = 2,78$ au lieu du gain théorique de 5, puisque le pipeline n'est pas complètement rempli au début.

10.2 Résolutions des aléas

Les aléas sont des conditions particulières qui bloquent les instructions du pipeline. Une suite d'instructions particulière ne peut être placées correctement dans le pipeline.

10.2.1 Aléas de données lecture après écriture

Les aléas de données sont dues aux dépendances entre les opérandes des instructions.

```
1 : addi $t0 , $zero , 1
2 : addi $t1 , $zero , 2
3 : add  $t2 , $t1   , $t0
```

Sur la suite d'instructions précédente nous remarquons des dépendances de données pour l'instruction 3 : les registres $t1$ et $t0$ sont calculés par les instructions 1 et 2 . Les opérandes de l'instruction de la ligne 3 étant lues dans le banc de registres durant la phase DI, elles ne sont pas encore disponibles. Elles le seront lorsque les instructions produisant les opérandes seront dans l'étage ER. En attendant l'instruction 3 attend à l'étage DIque les opérandes soient disponibles.

Instruction / Cycle	1	2	3	4	5	6	7	8	9	10
1 :addi $t0 , $zero , 1	LI	DI	EX	M	ER					
2 :addi $t1 , $zero , 2		LI	DI	EX	M	ER				
3 :add $t2 , $t1 , $t0			LI	**DI**	**DI**	DI	EX	M	ER	

FIGURE 10.4 – Diagramme temporel aléa LEA instructions UAL

L'instruction **add** *$t2* , *$t1* , *$t0* utilise les registres *$t1* et *$t2* qui sont en cours de calcul dans les instructions précédentes. Le décodage des opérandes de cette instruction ne peut pas s'effectuer correctement à l'étage DI. En effet, elles ne sont pas encore connues. Le décodage sera donc effectué au cycle numéro 6, quand les valeurs de *$t0* et *$t1* auront été écrites dans le banc de registres du MIPS. L'aléa provoque la perte de 2 cycles dans cette suite d'instructions.

10.2.2 Résolution des aléas lecture après écriture

Ce problème peut se résoudre matériellement en ne décodant plus les opérandes manquantes à l'étage DI, et en plaçant sur les entrées de l'UAL de l'étage EX, les résultats présents à l'entrée des étages M où ER. Cette technique nommée **envoi** permet de supprimer une partie des temps d'attente. Les envois sont symbolisés en encadrant la notation de l'étage (***EX***,***M***,***ER***), la donnée envoyée partant de l'étage de l'instruction la plus avancée et se dirigeant vers l'instruction la moins avancée.

Instruction / Cycle	1	2	3	4	5	6	7
1 :addi *$t0* , *$zero* , 1	LI	DI	EX	M	ER		
2 :addi *$t1* , *$zero* , 2		LI	DI	EX	***M***	ER	
3 :add *$t2* , *$t1* , *$t0*			LI	DI	***EX***	M	ER

FIGURE 10.5 – Diagramme temporel envoi et instructions UAL

Dans ce cas il n'y a plus de perte de cycle.

Remarque 54 :

- L'envoi peut être effectué entre les étages M (source) et EX (destination)
- L'envoi peut être effectué entre les étages ER (source) et M ou EX (destination)
- L'envoi de l'étage ER vers l'étage DI est implicite. En effet, lors de la modification d'un registre, nous pouvons lire la bonne valeur.

Par contre lors d'une lecture mémoire, nous devons attendre que l'opérande provenant de la mémoire soit disponible. Ceci est effectif au début de l'étage ER. Dans ce cas l'attente d'un cycle est effectuée à l'étage EX.

1 : lw *$t1* , 0(*$t0*)
2 : addi *$t1* , *$t1* , 1

Instruction / Cycle	1	2	3	4	5	6	7
1 :lw *$t1* , 0(*$t0*)	LI	DI	EX	M	***ER***		
2 :addi *$t1* , *$t1* , 1		LI	DI	EX	***EX***	M	ER
3 :addi *$t0* , *$t0* ,4			LI	DI	DI	EX	M

FIGURE 10.6 – Diagramme temporel chargement

Remarque 55 :

- Dans ce cas de la lecture mémoire (instruction lw) , il faut attendre la fin de la lecture (étage M) pour pouvoir utiliser la valeur lue en mémoire. Il reste donc un cycle perdu. Pour supprimer cette attente, nous devons réorganiser afin de ne pas utiliser une opérande lue en mémoire lors de l'instruction suivante. Cette technique est appelée **chargement retardé** et permet de ne jamais mettre un étage en attente.

Instruction / Cycle	1	2	3	4	5	6	7
1 :lw *$t1* , 0(*$t0*)	LI	DI	EX	M	***ER***		
2 :addi *$t0* , *$t0* ,4		LI	DI	EX	M	ER	
3 :addi *$t1* , *$t1* ,1			LI	DI	***EX***	M	ER

FIGURE 10.7 – Diagramme temporel chargement retardé

Dans ce cas, l'instruction 2 n'utilisant pas le résultat de l'instruction 1, nous n'aurons pas de cycle d'attente.

— Dans le cas d'une écriture mémoire précédé d'une lecture mémoire, l'envoi s'effectuera directement entre l'entre l'étage ***ER*** et ***M***.

```
1 : lw  $t1 , 0( $t0 )
2 : sw  $t1 ,0( $t0 )
```

Instruction / Cycle	1	2	3	4	5	6	7
1 :lw *$t1* , 0(*$t0*)	LI	DI	EX	M	***ER***		
2 : sw *$t1* ,0(*$t0*)		LI	DI	EX	***M***	ER	

FIGURE 10.8 – Diagramme temporel chargement - rangement

10.2.3 Aléas de branchement ou de contrôle

Lors d'un branchement, la décision de prendre le branchement ou non est connue au début de l'étage ER. L'instruction cible ou l'instruction suivante démarre seulement à ce moment là : trois cycles sont perdus.

```
boucle :
  # autre instruction
  addi  $t1 , $t1 ,-1
  add  $a0 , $a0 , $t1
  bne  $t1 ,  $zero , boucle
  #inst +1
  #inst +2
  #inst +3
```

Instruction / Cycle	1	2	3	4	5	6	7	8	9	10	11
addi *$t1* , *$t1* ,-1	LI	DI	EX	***M***	***ER***						
addi *$a0* , *$a0* , *$t1*		LI	DI	***EX***	M	ER					
bne *$t1* , *$zero* ,boucle			LI	DI	***EX***	M	ER				
inst +1				LI	DI	EX					
inst +2					LI	DI					
inst +3						LI					
addi *$t1* , *$t1* ,-1							LI	DI	EX	M	ER

FIGURE 10.9 – Diagramme temporel aléas de branchement pris

Dans ce cas d'un branchement pris, trois cycles seront perdus et le pipeline sera vidé.

Instruction / Cycle	1	2	3	4	5	6	7	8	9	10	11
addi $t1 , $t1 ,-1	LI	DI	EX	***M***	***ER***						
addi $a0 , $a0 , $t1		LI	DI	***EX***	M	ER					
bne $t1 , $zero ,boucle			LI	DI	***EX***	M	ER				
inst +1				LI	DI	EX	M	ER			
inst +2					LI	DI	EX	M	ER		
inst +3						LI	DI	EX	M	ER	
addi $t1 , $t1 ,-1							LI	DI	EX	M	ER

FIGURE 10.10 – Diagramme temporel aléas de branchement non pris

Dans ce cas d'un branchement non pris, aucun cycle ne sera perdu.

Pour supprimer cette attente de trois cycles quand le branchement est pris, nous devons décider au plus tôt (dans l'étage DI) si le branchement doit être pris. Cette technique est nommée **branchement au plus tôt**. Pour cela nous devons ajouter un additionneur pour calculer l'adresse de la cible et déplacer le comparateur et la prise de décision concernant le branchement dans l'étage DI.

Dans le cas du branchement au plus tôt, le délai de branchement est réduit à un cycle si le branchement est pris.

Instruction / Cycle	1	2	3	4	5	6	7	8	9	10	11
addi $t1 , $t1 ,-1	LI	DI	EX	***M***	ER						
addi $a0 , $a0 , $t1		LI	DI	***EX***	M	ER					
bne $t1 , $zero ,boucle			LI	***DI***	EX	M	ER				
inst +1				LI							
addi $t1 , $t1 ,-1					LI	DI	EX	M	ER		

FIGURE 10.11 – Diagramme temporel aléas de branchement au plus tôt pris

Pour supprimer ce cycle perdu lorsque le branchement est pris, nous devons réorganiser le code pour que l'instruction qui suit le branchement soit **toujours traitée** . Cette technique est nommée **branchement retardé** ou **delayed branch**. Dans le délais de branchement, nous devons placer un instruction et non pas une pseudo-instruction décomposée en plusieurs instructions simples.

```
boucle :
  addi  $t1 , $t1 ,-1
  # autre instruction
  bne  $t1 ,  $zero , boucle
  add  $a0 , $a0 , $t1  #instruction dans le délais de branchement
```

Instruction / Cycle	1	2	3	4	5	6	7	8	9	10	11
b :addi $t1 , $t1 ,-1	LI	DI	EX	***M***	ER						
# autre instruction		LI	DI	EX	M	ER					
bne $t1 , $zero ,b			LI	***DI***	EX	M	ER				
addi $a0 , $a0 , $t1				LI	DI	EX	M	ER			
b :addi $t1 , $t1 ,-1					LI	DI	EX	M	ER		

FIGURE 10.12 – Diagramme temporel branchement retardé d'un cycle

Dans ce cas, le fonctionnement du pipeline est optimal, puisque tous les cycles sont utilisés.

Remarque 56 :

- Pour le branchement au plus tôt, les opérandes de l'instruction de branchement doivent être disponibles dans l'étage DI. L'envoi doit donc placer un résultat dans l'étage DI, cela se traduit avec la notation ***DI***.
- Dans le cas du branchement au plus tôt nous devons ajouter un envoi entre les étages M (source) et DI (destination) pour récupérer le résultat d'une instruction précédente dés que possible.

10.3 Exercices

Exercice 15 : Théorique : Performances du pipeline

Question 1 : Nombre d'étage Un processeur sans pipeline possède un temps de cycle de 1 nano-seconde. Les registres tampon ont un TT de 0.05 ns. Quels seront les temps de cycle et de latence des versions avec un pipeline à 2,4,8 et 16 étages si le temps passé dans chaque étage est identiques ? Combien d'étages sont requis pour obtenir un temps de cycle de 0.2ns et 0.1ns ?

Question 2 : *Comptage* Soit le programme assembleur MIPS :

```
1 : andi  $t0 , $a0 ,1
2 : add   $v0 , $v0 , $t0
3 : srl   $a0 , $a0 ,1
4 : bne   $a0 , $zero , 1 :
```

1. Complétez le diagramme temporel de la figure Fig 10.13 et donnez le nombre de cycles pour une itération avec un pipeline sans optimisation.

Inst / Cycle	01	02	03	04	05	06	07	08	09	10	11	12
1 : andi $t0 , $a0 ,1	LI	DI	EX	M	ER							
2 : add $v0 , $v0 , $t0												
3 : srl $a0 , $a0 ,1												
4 : bne $a0 , $zero ,1												
1 : andi $t0 , $a0 ,1												LI

Figure 10.13 – Énoncé *comptage* :Diagramme temporel sans optimisation

2. Complétez le diagramme temporel de la figure Fig 10.14 et donnez le nombre de cycles pour une itération avec un pipeline avec l'envoi.

Inst / Cycle	01	02	03	04	05	06	07	08	09	10	11	12
1 : andi $t0 , $a0 ,1	LI	DI	EX	M	ER							
2 : add $v0 , $v0 , $t0												
3 : srl $a0 , $a0 ,1												
4 : bne $a0 , $zero ,1												
1 : andi $t0 , $a0 ,1								LI	DI	EX	M	ER

Figure 10.14 – Énoncé *comptage* :Diagramme temporel avec envoi

3. Complétez le diagramme temporel de la figure Fig 10.15 et donnez le nombre de cycles pour une itération avec un pipeline avec l'envoi et le branchement au plus tôt.

Inst / Cycle	01	02	03	04	05	06	07	08	09	10	11	12
1 : andi $t0 , $a0 ,1	LI	DI	EX	M	ER							
2 : add $v0 , $v0 , $t0												
3 : srl $a0 , $a0 ,1												
4 : bne $a0 , $zero ,1												
1 : andi $t0 , $a0 ,1							LI	DI	EX	M	ER	

Figure 10.15 – Énoncé *comptage* :Diagramme temporel avec envoi et branchement au plus tôt

4. Réorganisez le code sur la figure Fig 10.16 , pour obtenir avec l'envoi et le branchement au plus tôt, 5 cycles par itération.

Inst / Cycle	01	02	03	04	05	06	07	08	09	10	11	12
1 : andi $t0 , $a0 ,1	LI	DI	EX	M	ER							
1 : andi $t0 , $a0 ,1						LI	DI	EX	M	ER		

Figure 10.16 – Énoncé *comptage* :Diagramme temporel réorganisé avec envoi et branchement au plus tôt : 5 cycles

Question 3 : *copieChaine* Soit le programme assembleur MIPS :

```
1 : lb   $t0 , 0( $a0 )
2 : sb   $t0 , 0( $a1 )
3 : addi  $a0 , $a0 ,1
4 : addi  $a1 , $a1 ,1
5 : bne   $t0 , $zero , 1 :
```

1. Complétez le diagramme temporel de la figure Fig 10.17 et donnez le nombre de cycles pour une itération avec un pipeline sans optimisation.

Inst / Cycle	01	02	03	04	05	06	07	08	09	10	11
1 : lb $t0 , 0($a0)	LI	DI	EX	M	ER						
2 : sb $t0 , 0($a1)											
3 : addi $a0 , $a0 ,1											
4 : addi $a1 , $a1 ,1											
5 : bne $t0 , $zero ,1 :											
1 : lb $t0 , 0($a0)											LI

Figure 10.17 – Énoncé *copieMem* :Diagramme temporel sans optimisation

2. Complétez le diagramme temporel de la figure Fig 10.18 et donnez le nombre de cycles pour une itération avec un pipeline utilisant l'envoi et le branchement au plus tôt.

Inst / Cycle	01	02	03	04	05	06	07	08	09	10	11
1 : lb *$t0* , 0(*$a0*)	LI	DI	EX	M	ER						
2 : sb *$t0* , 0(*$a1*)											
3 : addi *$a0* , *$a0* ,1											
4 : addi *$a1* , *$a1* ,1											
5 : bne *$t0* , *$zero* ,1 :											
1 : lb *$t0* , 0(*$a0*)							LI	DI	EX	M	ER

FIGURE 10.18 – Énoncé *copieMem* : Diagramme temporel avec envoi et branchement au plus tôt

3. Donnez le code réorganisé sur la figure FIG 10.19 et le diagramme temporel avec un pipeline optimisé (envoi et branchement au plus tôt) et utilisant le branchement retardé, pour obtenir 5 cycles par itération.

Inst / Cycle	01	02	03	04	05	06	07	08	09	10	11
1 : lb *$t0* , 0(*$a0*)	LI	DI	EX	M	ER						
1 : lb *$t0* , 0(*$a0*)						LI	DI	EX	M	ER	

FIGURE 10.19 – Énoncé *copieMem* : Diagramme temporel avec envoi et branchement au plus tôt et branchement retardé : 5 cycles

Correction exercice 15 :

Question 1 : Nombre d'étages

$$\begin{aligned} \mathrm{TP} &= \frac{\mathrm{TSP}}{\mathrm{n}} + \mathrm{TT} \\ &= \frac{1}{n} + 0.05 \\ \mathit{Latence} &= \mathrm{TP} \times n \end{aligned}$$

Nombre d'étages	Temps de cycle	Latence
2	0.55 ns	1.1ns
4	0.3 ns	1.2 ns
8	0.175 ns	1.4 ns
16	0.1125 ns	1.8 ns

$$n = \frac{\mathrm{TSP}}{\mathrm{TP} - \mathrm{TT}}$$

Pour 0,2ns nous trouvons 6,667 étages, le nombre entier immédiatement supérieur est 7 étages.

Pour 0,1ns nous trouvons 20 étages.

Question 2 : *Comptage* Les diagrammes temporels sont représentés sur le figures Fig 10.20 à Fig 10.23 . Dans ce programme nous avons deux dépendances : *$t0* entre les lignes 1 : et 2 : et *$a0* entre les lignes 3 : et 4 :.

Inst / Cycle	01	02	03	04	05	06	07	08	09	10	11	12
1 : andi ***$t0*** , *$a0* ,1	LI	DI	EX	M	ER							
2 : add *$v0* , *$v0* , ***$t0***		LI	DI	DI	DI	EX	M	ER				
3 : srl ***$a0*** , *$a0* ,1			LI	LI	LI	DI	EX	M	ER			
4 : bne ***$a0*** , *$zero* ,1						LI	DI	DI	DI	EX	M	ER
1 : andi *$t0* , *$a0* ,1												LI

Figure 10.20 – Correction *comptage* :Diagramme temporel sans optimisation : 11 cycles

Inst / Cycle	01	02	03	04	05	06	07	08	09	10	11	12
1 : andi ***$t0*** , *$a0* ,1	LI	DI	EX	***M***	ER							
2 : add *$v0* , *$v0* , ***$t0***		LI	DI	***EX***	M	ER						
3 : srl ***$a0*** , *$a0* ,1			LI	DI	EX	***M***	ER					
4 : bne ***$a0*** , *$zero* ,1				LI	DI	***EX***	M	ER				
1 : andi *$t0* , *$a0* ,1								LI	DI	EX	M	ER

Figure 10.21 – Correction *comptage* :Diagramme temporel avec envoi : 7 cycles

L'instruction bne , lors du branchement au plus tôt est effectuée complètement dans l'étage DI. Le registre *$a0* doit être placé dans le banc de registres pour que le branchement puisse être effectué.

Inst / Cycle	01	02	03	04	05	06	07	08	09	10	11	12
1 : andi ***$t0*** , *$a0* ,1	LI	DI	EX	***M***	ER							
2 : add *$v0* , *$v0* , ***$t0***		LI	DI	***EX***	M	ER						
3 : srl ***$a0*** , *$a0* ,1			LI	DI	EX	***M***	ER					
4 : bne ***$a0*** , *$zero* ,1				LI	DI	***DI***	EX	M	ER			
1 : andi *$t0* , *$a0* ,1							LI	DI	EX	M	ER	

FIGURE 10.22 – Correction *comptage* :Diagramme temporel avec envoi et branchement au plus tôt : 6 cycles

Inst / Cycle	01	02	03	04	05	06	07	08	09	10	11	12
1 : andi *$t0* , *$a0* ,1	LI	DI	EX	M	***ER***							
3 : srl *$a0* , *$a0* ,1		LI	DI	EX	***M***	ER						
2 : add *$v0* , *$v0* , *$t0*			LI	DI	***EX***	M	ER					
4 : bne *$a0* , *$zero* ,1				LI	***DI***	EX	M	ER				
1 : andi *$t0* , *$a0* ,1						LI	DI	EX	M	ER		

FIGURE 10.23 – Correction *comptage* :Diagramme temporel réorganisé avec envoi et branchement au plus tôt : 5 cycles

Dans ce cas, deux envois sont réalisés dans l'architecture. L'une du registre *$t0* et l'autre du registre *$a0* .

Question 3 : *copieChaine* Les diagrammes temporels sont représentés sur le figures FIG 10.24 à FIG 10.26 . Dans ce programme nous avons deux dépendances : *$t0* entre les lignes 1 : et 2 :.

Inst / Cycle	01	02	03	04	05	06	07	08	09	10	11
1 : lb ***$t0*** , 0(*$a0*)	LI	DI	EX	M	ER						
2 : sb ***$t0*** , 0(*$a1*)		LI	DI	DI	DI	EX	M	ER			
3 : addi *$a0* , *$a0* ,1			LI	LI	LI	DI	EX	M	ER		
4 : addi *$a1* , *$a1* ,1						LI	DI	EX	M	ER	
5 : bne *$t0* , *$zero* ,1 :							LI	DI	EX	M	ER
1 : lb *$t0* , 0(*$a0*)											LI

FIGURE 10.24 – Correction *copieMem* :Diagramme temporel sans optimisation : 10 cycles

Inst / Cycle	01	02	03	04	05	06	07	08	09	10	11
1 : lb *$t0* , 0(*$a0*)	LI	DI	EX	M	***ER***						
2 : sb *$t0* , 0(*$a1*)		LI	DI	EX	***M***	ER					
3 : addi *$a0* , *$a0* ,1			LI	DI	EX	M	ER				
4 : addi *$a1* , *$a1* ,1				LI	DI	EX	M	ER			
5 : bne *$t0* , *$zero* ,1 :					LI	DI	EX	M	ER		
1 : lb *$t0* , 0(*$a0*)							LI	DI	EX	M	ER

FIGURE 10.25 – Correction *copieMem* : Diagramme temporel avec envoi et branchement au plus tôt : 6 cycles

Inst / Cycle	01	02	03	04	05	06	07	08	09	10	11
1 : lb *$t0* , 0(*$a0*)	LI	DI	EX	M	***ER***						
2 : sb *$t0* , 0(*$a1*)		LI	DI	EX	***M***	ER					
3 : addi *$a0* , *$a0* ,1			LI	DI	EX	M	ER				
4 : bne *$t0* , *$zero* ,1 :				LI	DI	EX	M	ER			
5 : addi *$a1* , *$a1* ,1					LI	DI	EX	M	ER		
1 : lb *$t0* , 0(*$a0*)						LI	DI	EX	M	ER	

FIGURE 10.26 – Correction *copieMem* : Diagramme temporel avec envoi et branchement retardé d'un cycle : 5 cycles

Exercice 16 : Pratique : Performances du pipeline

Question 1 : Mars : branchement retardé d'un cycle Le branchement retardé est activé dans le menu **setting** du **simulateur MARS**. Réorganisez le code des fonctions nommées *initMem*, *copieMem* et dans le programme d'appel de la partie précédente pour obtenir un fonctionnement correct du programme lorsque l'option **delayed branch** est activée.

Remarque 57 :

- Dans le délai de branchement, ne pas positionner une pseudo-instruction décomposée en plusieurs instructions.
- Si aucune instruction n'est disponible le délai sera rempli avec l'instruction *nop*

Correction exercice 16 :

Listing 10.1 – InitMem et copieMem avec delayed branch

```
1  .data
2    source : .space 32
3    destination : .space 32
4  .text
5  main:
6    #Test InitMem
7    la $a0,source
8    addi $a2,$zero,0xFF00FF00 # pas dans le delais car pseudo instruction
9    jal InitMem
10   addi $a1,$zero,8 #delais branchement
11   #Test InitMem
12   la $a0,source
13   la $a1,destination
14   jal CopieMem
15   addi $a2,$zero,8 #delais branchement
16   # -- exit
17   addi $v0, $zero, 10
18   syscall
19
20 # Fonction Initialisation zone memoire
21 # $a0 adresse zone memoire
22 # $a1 taille de le zone memoire
23 # $a2 valeur a placer en memoire
24 InitMem:
25   beq $a1,$zero,InitMem_Fin
26   addi $a1,$a1,-1 #delais branchement
27   sw $a2,0($a0)  # pas sw dans le delais ...
28   j InitMem
29   addi $a0,$a0,4 #delais branchement
30 InitMem_Fin:
31   jr $ra
32   nop  #delais branchement nop car rien d'autre à faire
33
34 # Fonction Copie zone mŐmoire
35 # $a0 adresse zone memoire src
36 # $a1 adresse zone memoire dest
37 # $a2 taille de le zone mémoire
38 CopieMem:
39 CopieMem_boucle:
40   beq $a2,$zero,CopieMem_Fin
41   addi $a2,$a2,-1 #delais branchement
42   lw $t0,0($a0)
43   addi $a0,$a0,4
44   sw $t0,0($a1) #evite d'utiliser un autre registre
45   j CopieMem_boucle
46   addi $a1,$a1,4 #delais de branchement
47 CopieMem_Fin:
48   jr $ra #delais branchement
49   nop  #delais branchement nop car rien d'autre à faire
```

Chapitre 11

Additions rapides

11.1 Additionneur complet

La figure Fig 2.6 donne la table de vérité d'une additionneur complet. Les équations de ce circuit additionnant trois bits nommés $a_{i-1}, b_{i-1}, c_{i-1}$, et générant deux bits nommés r_i, c_i en utilisant une somme de produit logique, sont définies par les équations (11.1) et (11.2).

$$c_i = a_{i-1}.b_{i-1} + a_{i-1}.c_{i-1} + b_{i-1}.c_{i-1} = a_{i-1}.b_{i-1} + (a_{i-1} + b_{i-1}).c_{i-1} \tag{11.1}$$

$$r_i = a_{i-1}.\overline{b_{i-1}}.\overline{c_{i-1}} + \overline{a_{i-1}}.b_{i-1}.\overline{c_{i-1}} + \overline{a_{i-1}}.\overline{b_{i-1}}.c_{i-1} + a_{i-1}.b_{i-1}.c_{i-1} \tag{11.2}$$

Pour calculer la valeur de la retenue sortante c_i, les signaux entrant nommés $a_{i-1}, b_{i-1}, c_{i-1}$, doivent traverser deux couches de portes logiques (les portes logiques ET sont traversées en parallèle puis leurs résultats sont envoyés à une porte logique OU). En même temps, nous calculons la valeur du signal r_i en utilisant parallèlement deux autres couches de portes logiques.

$$\text{Nombre de couches logiques additionneur complet} = 2 \tag{11.3}$$

11.2 Additionneur à propagation de retenue (RCA)

L'additionneur à propagation de retenue (nommé aussi RCA, *ripple carry adder*) est lent. En effet, pour obtenir le résultat, nous devons attendre que les diverses retenues soient calculées, puis propagées au bit suivant, avant d'obtenir le résultat final.
Nous calculons la retenue c_1, puis quand cette valeur est obtenue, la valeur de c_2 et ceci jusqu'à c_n,retenue sortante . Pour cette dernière valeur, les signaux doivent traverser deux couches de portes logiques pour calculer la valeur de c_1, puis deux autres couches de portes logiques pour calculer c_2, soit quatre au total. En généralisant à n rangs, les signaux doivent traverser $2n$ couches de portes logiques pour obtenir c_n. Le temps de calcul du résultat de l'addition est lié directement au temps de transit dans chaque couche de porte logique. Plus le nombre de couches est réduit, plus les résultats sont obtenus rapidement.

$$\text{Nombre de couches logiques propagation de retenue } n \text{ rangs} = 2n \tag{11.4}$$

11.3 Additionneur à anticipation de retenue (CLA)

Le problème de l'addition à propagation de retenue est le calcul des retenues. En utilisant l'équation (11.1), nous pouvons calculer directement la retenue entrante au rang i nommée c_i

en fonction de la retenue entrante nommée c_0 sans attendre la propagation de la retenue dans les rangs.

$$c_i = g_{i-1} + p_{i-1}.c_{i-1} \text{ avec } g_{i-1} = a_{i-1}.b_{i-1} \text{ et } p_{i-1} = a_{i-1} + b_{i-1} \quad (11.5)$$

$$c_1 = g_0 + p_0.c_0 \quad (11.6)$$

$$c_2 = g_1 + p_1.c_1 = g_1 + p_1(g_0 + p_0.c_0) = g_1 + p_1.g_0 + p_1.p_0.c_0 \quad (11.7)$$

$$c_i = g_{i-1} + p_{i-1}.c_{i-1} = g_{i-1} + p_{i-1}.g_{i-2} + p_{i-1}.p_{i-2}.g_{i-3} + \ldots + p_{i-1}.p_{i-2}\ldots p_1.g_0 + p_{i-1}.p_{i-2}\ldots p_1.p_0.c_0 \quad (11.8)$$

Les équations des retenues sortantes des rangs 0 à 3 sont données ci-dessous :

$$\begin{aligned} c_1 &= g_0 + p_0.c_0 \\ c_2 &= g_1 + p_1.g_0 + p_1.p_0.c_0 \\ c_3 &= g_2 + p_2.g_1 + p_2.p_1.g_0 + p_2.p_1.p_0.c_0 \\ c_4 &= g_3 + p_3.g_2 + p_3.p_2.g_1 + p_3.p_2.p_1.g_0 + p_3.p_2.p_1.p_0.c_0 \end{aligned}$$

Lors de l'addition à anticipation de retenue, nous évaluons simultanément toutes les retenues. Pour cela, nous calculons pour chaque rang les valeurs de p_i et g_i ; g_i signifie générée, si les deux bits du rang sont à 1 ; p signifie propagée, si un ou deux bits du rang sont à 1.
La retenue entrante au rang i vaut 1 :

- si le rang $i-1$ à généré la retenue ($g_{i-1} = 1$)
- ou si le rang $i-1$ à propagé la retenue générée au rang $i-2$ ($p_{i-1} = 1$ et $g_{i-2} = 1$)
- ou si les rangs $i-1$ et $i-2$ ont propagés la retenue générée au rang $i-3$($p_{i-1} = p_{i-2} = 1$ et $g_{i-3} = 1$)
- ...
- ou si tous les rangs inférieurs ont propagés la retenue entrante dans l'addition ($p_{i-1} = p_{i-2} = ... = p_0 = c_0 = 1$).

L'additionneur qui calcule les retenues en utilisant l'équation (11.8) s'appelle à retenue anticipée (nommée aussi CLA, *carry Lookahead Adder*).
Le nombre de couches logiques traversées est de cinq :

- une pour calculer les signaux p_i et g_i,
- deux pour calculer les retenues,
- deux couches pour calculer les résultat.

La figure Fig 11.1 illustre le principe de réalisation de l'addition. La partie nommée CLU (*carry Lookahead Unit*) effectue le calculs de toutes les retenues.

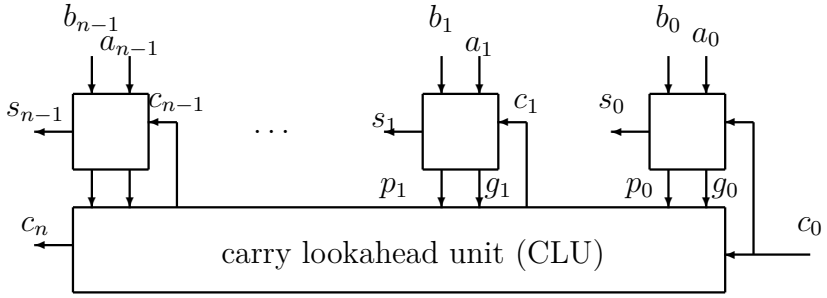


Figure 11.1 – Additionneur à anticipation de retenue

C'est une amélioration considérable par rapport aux $2 \times n$ couches logiques de l'addition à propagation simple de retenue.

Malheureusement, une addition à anticipation de retenue n rang à besoin de portes logiques à $n+1$ entrées, ce qui est difficile à réaliser pratiquement. De plus, le signal p_{n-1} est connecté à n entrées logique, ce qui demande une intensité électrique forte. Une addition 32 bits ou 64 bits utilisant ce principe est irréalisable.

Nous pouvons utiliser l'idée de l'anticipation de retenue pour construire un circuit un peu plus lent que l'anticipation précédente mais beaucoup plus rapide que la propagation simple de retenue. L'idée est de construire graduellement les retenues en utilisant un arbre avec $i \leq j, j+1 < k$ (sans démonstration). L'ensemble des rangs i à k est décomposé en deux parties :

— La partie haute entre $j+1$ et k,

— La partie base entre i et j,

$$c_{k+1} = G_{i,k} + P_{i,k}.c_i \tag{11.9}$$

$$G_{i,k} = G_{j+1,k} + P_{j+1,k}G_{i,j} \tag{11.10}$$

$$P_{i,k} = P_{j+1,k}.P_{i,j} \tag{11.11}$$

$$\text{Si } i = j < k, G_{i,i} = g_i, P_{i,i} = p_i. \tag{11.12}$$

Remarque 58 :

— L'équation (11.9) indique de la retenue au rang $k+1$ vaut 1, si elle générée entre les rangs i et k, ou si elle est propagée entre les rangs i et k.

— L'équation (11.10) indique que la retenue au rang k vaut 1, si elle est générée dans la partie haute ou si elle générée dans la partie basse et propagée dans la partie haute.

— L'équation (11.11) indique que la retenue au rang k est propagée, si elle l'est dans la partie haute et dans la partie basse.

Nous allons maintenant concevoir un additionneur à anticipation de retenue réalisable pratiquement. Il est constitué de deux parties :

— La première partie, représentée sur la figure Fig 11.2 calcule les valeurs des P et G à partir des signaux p_i et g_i en utilisant les équations (11.10) et (11.11). Les signaux P et G sont calculés du haut vers le bas. En bas de l'arbre nous avons toutes les valeurs des P et G.

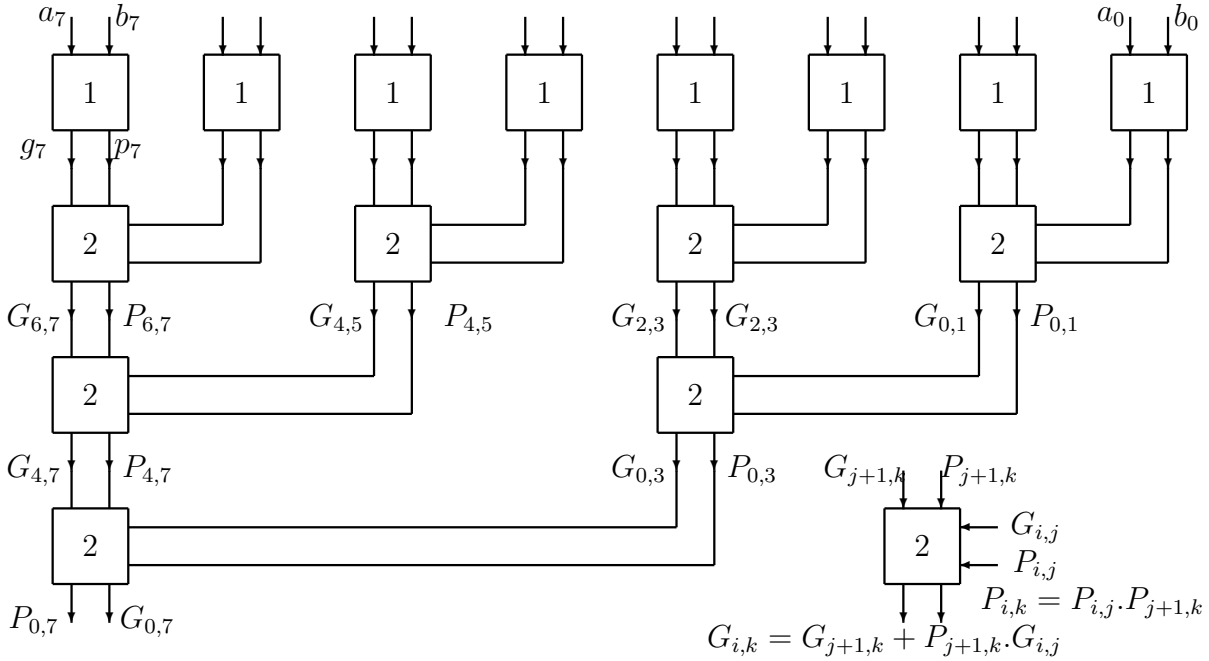


Figure 11.2 – Arbre d'anticipation de retenue partie 1

— La seconde partie, représentée sur la figure FIG 11.3 utilise l'équation (11.9). Les retenues sont calculées depuis le bas (à partir de c_0) vers le haut. Elles sont disponibles en haut de l'arbre et sont utilisées pour calculer le résultat de l'addition.

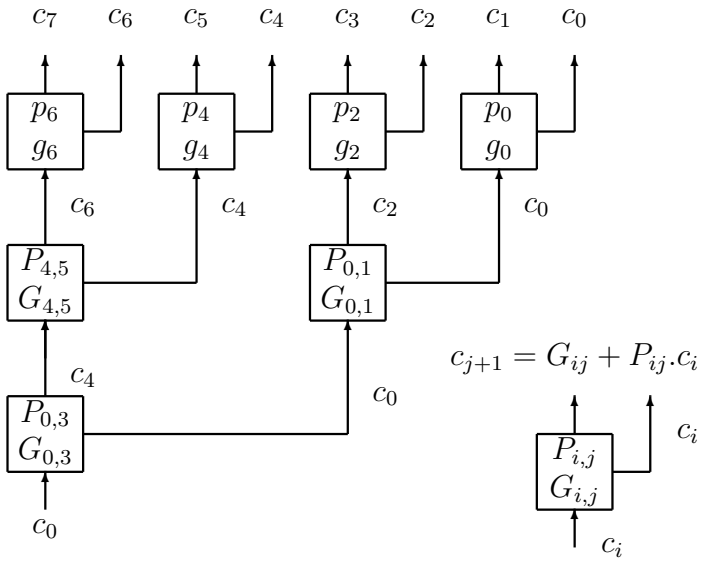


FIGURE 11.3 – Arbre d'anticipation de retenue partie 2

La figure FIG 11.4 représente la combinaison des deux parties précédentes. Les $p_i$et g_i à additionner se propagent de haut en bas de l'arbre, se combinent à c_0 en bas et remontent l'arbre pour donner les retenues et permettre de calculer les résultats.

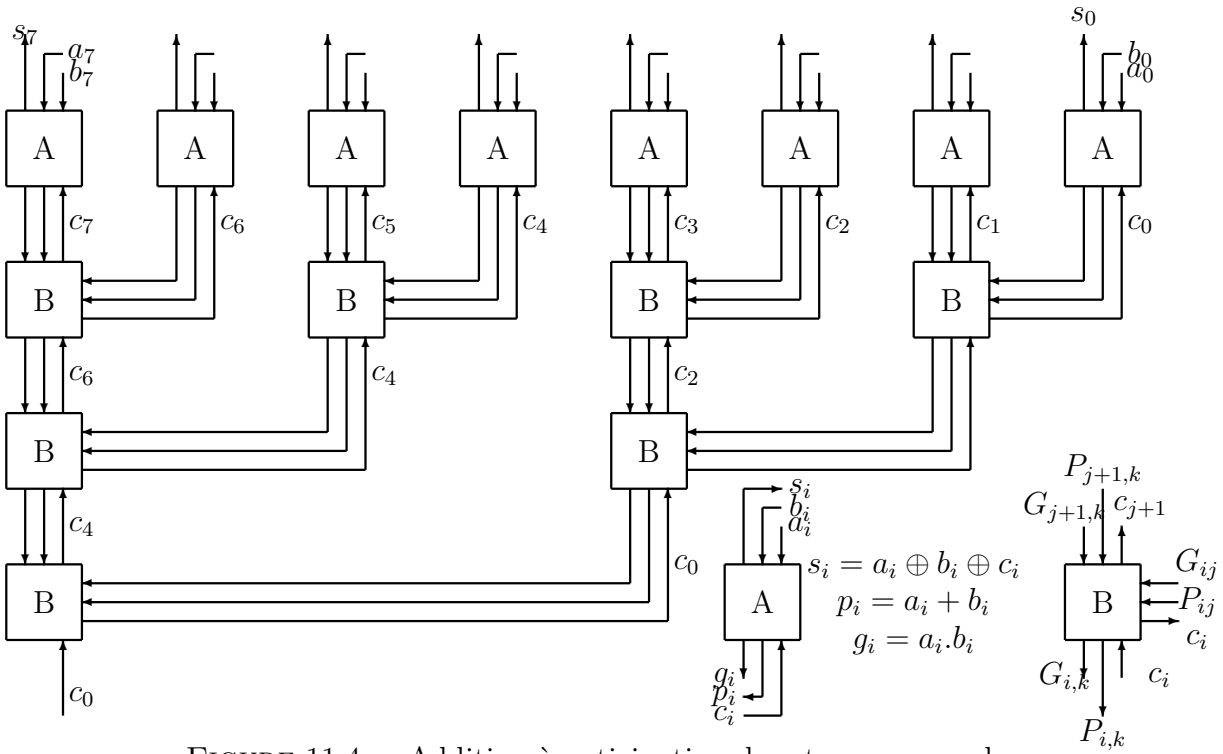


FIGURE 11.4 – Addition à anticipation de retenue avec arbre

Le nombre de couches logiques à traverser pour une addition n rangs à arbre anticipation (avec $n = 2^j$) est de :

— une pour calculer les p_i et g_i,
— deux par niveau pour descendre dans l'arbre logique (bloc B) sur j niveaux,soit $2j$
— deux par niveau pour remonter dans l'arbre logique (bloc B) sur j niveaux,soit $2j$

— deux couches par calculer le résultat.

(11.13) Nombre de couches logiques arbre anticipation retenue 2^j rangs $= 4 \times j + 3$

En contrepartie l'addition à arbre d'anticipation de retenue est légèrement plus complexe à réaliser que celui à propagation de retenue. En effet, un addition n bits à propagation utilise n cellules alors qu'une addition à arbre d'anticipation en utilise $2n$.
Lorsque n est grand, une conception hybride peut être utilisée. Deux options sont possibles

RCLA (ripple CLA Pour diminuer la complexité de l'arbre d'anticipation, des blocs d'addition CLA k rangs sont utilisés pour réaliser l'addition n rangs. Entre ces blocs de k rangs, les retenues sont propagées.

BCLA (Blocks CLA Les blocs d'addition k rangs utilisent la propagation de retenue. Entre ces blocs de k rangs, les retenues sont générées par arbre d'anticipation.

11.4 Additionneur à saut de retenue (CSA)

Une addition à saut de retenue (nommé aussi CSA, *carry skip adder*) se situe à mi-chemin en vitesse entre un addition à propagation et un addition à anticipation de retenue avec une complexité de réalisation proche de celle de la propagation de retenue. Les équations précédentes suggèrent que le calcul des P est plus simple que celui des G (pour P une porte ET suffit). L'addition à saut de retenue calcule seulement les valeurs de $P_{i,j}$. La figure Fig 11.5 illustre la réalisation de ce type d'addition à saut de retenue. Au début, les retenues sont calculées pour chaque addition k rang, avec leurs retenues entrantes à 0. Les retenues sortantes de ces additions k rangs peuvent être vues comme les signaux $G_{i,j}$. Si elles sont à 1, cela sera toujours exacte ; en effet, si les valeurs $a + b + 0$ génèrent une retenue, alors $a + b + 1$ en générera une aussi.
La retenue entrante de l'addition k rang suivant sera obtenue à partir de :

— la retenue générée dans l'addition k rangs. Nous pouvons considérer que ce résultat est toujours correct,

— de la retenue précédente si elle est propagée dans l'addition k rangs.

Si la retenue entrante et le signal propagé sont positionnés à un , alors la retenue saute l'addition k rangs suivante (d'où le nom de la technique). Cela permet de gagner du temps ; en cas de génération d'une retenue nous devons recalculer les valeurs de sortie de l'addition k rangs suivant.

Pour calculer la vitesse de l'addition n rangs à saut de retenue, nous supposerons que nous utilisons une addition k rangs à propagation ($2k$ couches de portes logique) et de deux couches logiques pour que la retenue saute un bloc.
Le temps le plus long, sera obtenu lorsqu'une retenue sera générée par la première addition k rangs qui sera transmisse aux additions k rangs suivantes. Dans ce cas, cette retenue devra éventuellement sauter $\frac{n}{k} - 2$ additions k rangs. Nous devons calculer la première retenue ($2k$ couches logiques), sauter les additions k rangs, soit $2(\frac{n}{k} - 2)$ couches logique et recalculer le dernier résultat ($2k$ couches logique). Dans ce cas le nombre de couches logiques à traverser pour calculer le résultat est de :

(11.14) Nombre de couches logiques saut de retenue n rangs $= 4k + 2(\frac{n}{k} - 2)$

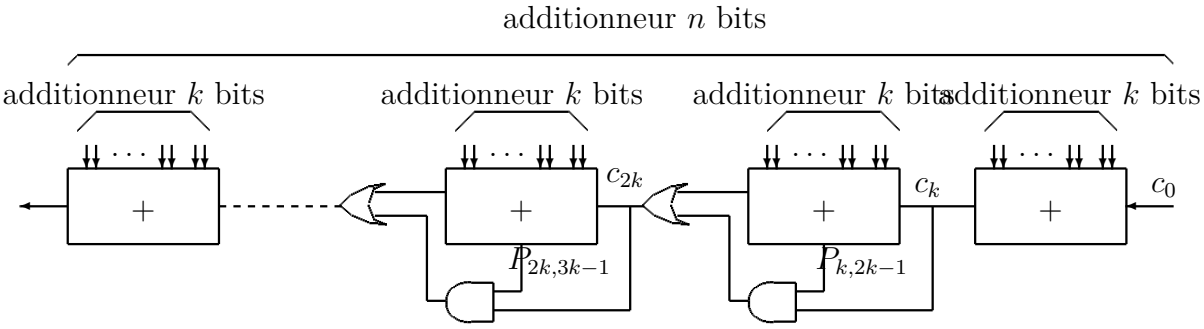


FIGURE 11.5 – Addition à saut de retenue

11.5 Addition à sélection de retenue

Ce type d'additionneur est aussi nommé *carry select adder*. Une addition à sélection de retenue fonctionne selon les deux principes suivants :

1. deux additions (supposées à propagation de retenue) sont effectuées en parallèle, l'une en supposant que la retenue entrante vaut 0, l'autre que la retenue entrante vaut 1,

2. quand la retenue entrante sera connue, la somme correcte déjà calculée est choisie.

La figure FIG 11.6 illustre ce principe pour un additionneur $2k$ rangs, décomposé en deux addition k rangs. La retenue sortante de l'addition k rangs de poids faible est utilisée pour sélectionner le résultat de l'addition k rangs de poids forts. La complexité est environ deux fois supérieure à celle d'une addition à propagation.

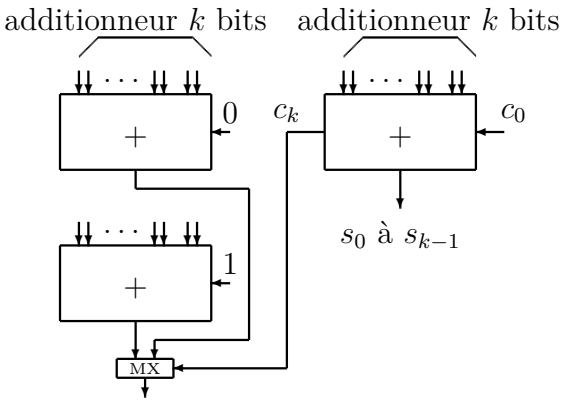


FIGURE 11.6 – Additionneur à sélection de retenue, principe

La figure FIG 11.7 généralise ce principe, pour un addition à sélection de retenue n bits formé de $\frac{n}{k}$ addition de k rangs.

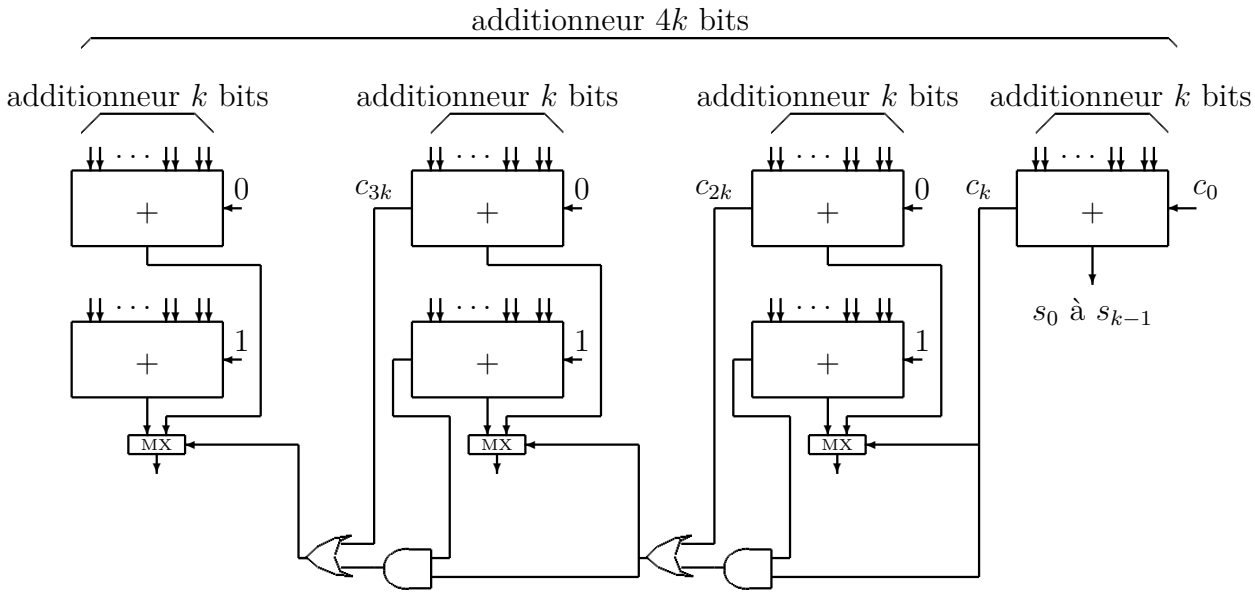


FIGURE 11.7 – Additionneur à sélection de retenue

Dans ce cas, en supposant que nous ayons besoin de traverser $2k$ couches logiques pour l'addition k rangs, et deux couches logique pour calculer la valeur de l'entrée du multiplexeur

(11.15) Nombre de couches logiques sélection de retenue n rangs $= 2k + 2(\frac{n}{k} - 2)$

11.6 Résumé

Soit à additionner n bits ($n = 2^j$), en utilisant des addition k bits ($k < n$) à propagation de retenue. La figure FIG 11.8 résumé le nombre de couches à traverser et la complexité de chaque type d'addition.

Type	Unité de temps	Complexité
Propagation	$2n$	n
Arbre anticipation	$4j + 3$	$\approx 2n$
Saut de retenue	$4k + 2(\frac{n}{k} - 2)$	$\approx n$
Sélection de retenue	$2k + 2(\frac{n}{k} - 2)$	$\approx 2n$

FIGURE 11.8 – Comparaison vitesse et complexité

En posant $n = 32 = 2^5$ et $k = 8$ nous obtenons les résultats de la figure FIG 11.9 .

Type	Unité de temps	Complexité
Propagation	64	32
Arbre anticipation	23	64
Saut de retenue	36	32
Sélection de retenue	20	64

FIGURE 11.9 – Comparaison numérique vitesse et complexité

11.7 Exercices

Exercice 17 : Théorique : Additions rapides

Pour notre architecture simplifiée nous souhaitons implémenter un additionneur rapide 16 rangs.

Question 1 : Addition à propagation de retenue Donnez le nombre de couches traversées pour obtenir le résultat.

Question 2 : Addition RCLA En utilisant des additions interne de 4 bits, donnez le nombre de couches traversées pour obtenir le résultat. Est ce une bonne solution

Question 3 : Addition BCLA En utilisant des additions interne de 4 bits à propagation (voir figure Fig 11.10 , donnez les équations de $P_{0,3}$ $G_{0,3}$ et donnez le nombre de couches traversées pour obtenir le résultats.

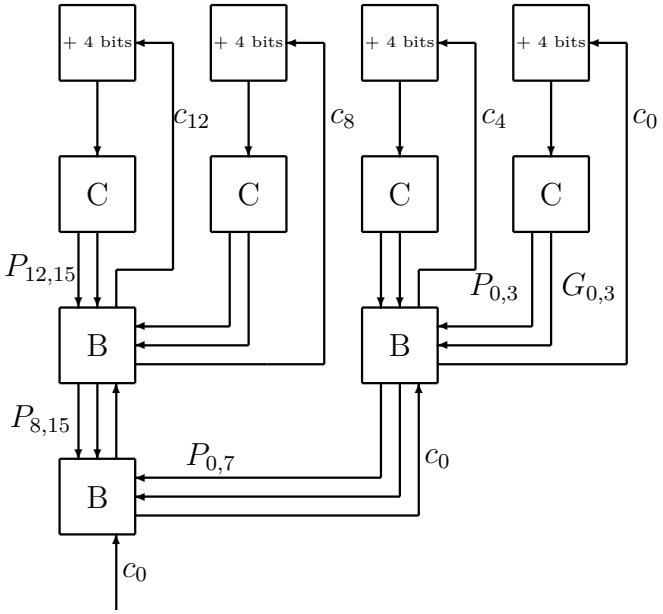


Figure 11.10 – Addition BCL

Correction exercice 17 :

Question 1 : Addition à propagation de retenue $2 \times 16 = 32$ couches logiques traversées.

Question 2 : Addition RCLA Dans ce cas j=2. Pour une addition interne nous avons besoin de $4 \times 2 + 3 = 11$ couches logiques. La propagation de la retenue entre addition internes fait que les signaux doivent traverser $11 \times 4 = 44$ couches logiques. Cette solution n'est pas efficace.

Question 3 : Addition BCLA

$$P_{0,3} = P_{0,1}P_{2,3} = P_{0,0}P_{1,1}P_{2,2}P_{3,3} = p_3p_2p_1p_0 \quad (11.16)$$

$$G_{0,3} = G_{2,3} + P_{2,3}G_{0,1} \quad (11.17)$$

$$= (G_{3,3} + P_{3,3}G_{2,2}) + (P_{2,2}P_{3,3})(G_{1,1} + P_{1,1}G_{0,0}) \quad (11.18)$$

$$= g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 \quad (11.19)$$

Les signaux traversant le bloc C doivent donc traverser deux couches logiques.
Au total les signaux traversent 19 couches logiques.

- une couche pour les p_i et g_i,
- deux couches pour les P, G,
- quatre couches pour descendre l'arbre,
- quatre couches pour remonter l'arbre,
- 8 couches dans les additionneur à propagation

Exercice 18 : Pratique : Additions rapides

Implémentez et tester les différents type d'additions rapides en utilisant *logisim* :

- CLA : 4 bits avec CLU
- CLA avec arbre. Implémentez aussi les blocs A et B
- CSA 16 bits avec des additions 4 bits.
- Sélection de retenue 16 bits avec des additions 4 bits.
- BCLA 16 bits avec des additions 4 bits : Implémenter aussi le bloc C.

Correction exercice 18 :

Voir le fichier *fast-add.circ*

Quatrième partie

Annexe

Table des figures

1.1 Codage analogique et digital . . . 5
1.2 Noms des ensembles de bits . . . 6
1.3 Portes logiques . . . 6
1.4 Fonction Non . . . 7
1.5 Fonctions logiques . . . 7
1.6 Portes logiques réalisées avec des NAND . . . 7
1.7 Fonctions logiques . . . 8

2.1 Valeurs maximales des nombres non signés . . . 11
2.2 Valeurs maximales des nombres hexadécimaux non signés . . . 11
2.3 Poids binaires et hexadécimaux . . . 12
2.4 Divisions successives en binaire . . . 13
2.5 Table de conversion hexadécimale en binaire . . . 13
2.6 Addition de deux et de trois bits . . . 14
2.7 Principe de l'addition à propagation de retenue . . . 15
2.8 Addition binaire de deux nombres non signés . . . 15
2.9 Addition hexadécimale de deux nombres non signés . . . 15
2.10 Table de vérité de l'addition 2 bits . . . 16
2.11 Schéma du demi-additionneur . . . 16
2.12 Schéma de l'additionneur complet . . . 17
2.13 Schéma de l'additionneur 2×4 bits . . . 17
2.14 Opérateur non bit à bit . . . 17
2.15 Valeur du bit 1 . . . 18
2.16 Forcer bit 1 à 1 . . . 19
2.17 Ou exclusif bit à bit . . . 19
2.18 Opérateur ou exclusif bit à bit . . . 19
2.19 Décalage logique d'un rang à gauche et à droite . . . 20
2.20 Circuit de l'opérateur et bit à bit . . . 21
2.21 Circuit décalage logique de 1 rang . . . 21
2.22 Comptage du nombre de donnée à 1 . . . 22
2.23 Circuit de comptage du nombre de donnée à 1 . . . 22
2.24 Comptage rapide du nombre de donnée à 1 . . . 23
2.25 Circuit de comptage rapide du nombre de donnée à 1 . . . 23
2.26 Table des codes ascii . . . 24
2.27 Bit 31 de x à 0 . . . 26
2.28 Bit 31 de x à 1 . . . 26
2.29 Placer le bit 31 de y dans le bit 0 de x . . . 26
2.30 Placer le bit 0 de y dans le bit 31 de x . . . 26
2.31 Masquer bit 30 à 23 de x . . . 27
2.32 Placer les bits 30 à 23 de y dans les bits 7 à 0 de x . . . 27

2.33 Placer les bits 7 à 0 de y dans les bits 30 à 23 de x . . . 27
2.34 Masquer bit 22 à 0 de x . . . 27
2.35 Placer les bits 22 à 0 de y dans 22 à 0 de x . . . 28
2.36 Placer les bits 22 à 0 de y dans les bits 22 à 0 de x . . . 28
2.37 Rotation d'un bit à droite . . . 28
2.38 Rotation d'un bit à gauche . . . 29
2.39 Principe de l'inversion des bits d'une donnée . . . 33
2.40 Principe de l'inversion des octets d'une donnée . . . 33
2.41 Comptage rapide du nombre de bit à 1 d'un octet . . . 34
2.42 Exemple de comptage rapide des bits d'une donnée . . . 35
2.43 Table de vérité de l'additionneur complet, énoncé . . . 38
2.44 Table de vérité égalité un bit, énoncé . . . 38
2.45 Table de vérité de l'additionneur complet, correction . . . 46
2.46 Table de vérité égalité un bit, correction . . . 46
2.47 Implémentation Java . . . 48
2.48 Tutorial logisim 1 . . . 49
2.49 Tutorial logisim 2 . . . 50
2.50 Tutorial logisim 3 . . . 50
2.51 Tutorial logisim 4 . . . 51
2.52 Tutorial logisim 5 . . . 51
2.53 Tutorial logisim 6 . . . 52
2.54 Tutorial logisim 7 et 8 . . . 52
2.55 Rotation un rang, énoncé . . . 53
2.56 Rotation n rangs, énoncé . . . 53
2.57 Egalité de deux nombres, énoncé . . . 53
2.58 Egalité de deux nombres, énoncé . . . 54
2.59 Comptage rapide, énoncé . . . 55
2.60 Rotation un rang, correction . . . 59
2.61 Rotation n rangs, correction . . . 60
2.62 Egalité de deux nombres, correction . . . 60
2.63 Egalité de deux nombres, correction . . . 61
2.64 Comptage rapide, correction . . . 62

3.1 Étendue des valeurs des nombres signés . . . 64
3.2 Addition de deux nombres signés . . . 65
3.3 Décalage arithmétique à droite . . . 65
3.4 Principe du décalage arithmétique à droite . . . 67
3.5 Codage de nombre signés . . . 70
3.6 Opérations sur des nombres signés . . . 70
3.7 Opérateurs Java signés . . . 73
3.8 Donnée $>>1$, énoncé . . . 73
3.9 Donnée $>>n$, énoncé . . . 73
3.10 Additionneur quatre bits de poids forts, énoncé . . . 74
3.11 Décalage arithmétique de 1 rang à droite, correction . . . 76
3.12 Décalage arithmétique de n rangs à droite, correction . . . 77
3.13 Additionneur quatre bits de poids forts, correction . . . 77

4.1 Position des champs au format IEEE 754 . . . 81
4.2 Caractéristiques des nombres codés IEEE 754 . . . 81
4.3 Excédent au format simple précision . . . 81
4.4 Affichage du réel -2,5 en hexadécimal . . . 83

4.5 Table de vérité d'un décodeur 2 bits . . . 85
4.6 Schéma interne du décodeur 2 bits . . . 86
4.7 Schéma externe du décodeur 2 bits . . . 86
4.8 Table de vérité du multiplexeur 1 parmi 2 . . . 86
4.9 Schéma interne d'un multiplexeur 1 bit . . . 87
4.10 Schéma externe d'un multiplexeur 1 bit . . . 87
4.11 Multiplexeur 1 parmi 4, 32 bits . . . 87
4.12 Table de vérité du comparateur 1 bit non signé . . . 88
4.13 Comparateur 1 bit non signé - vue externe . . . 88
4.14 Addition de deux nombres réels positifs normalisés . . . 88
4.15 Affichage d'un nombre réel normalisé en hexadécimal, énonce . . . 99
4.16 Addition de deux réels normalisés positifs, énoncé . . . 99
4.17 Affichage d'un nombre réel normalisé en hexadécimal, correction . . . 101
4.18 Addition de deux réels normalisés positifs, correction . . . 102

6.1 Chaîne de production de programme . . . 109
6.2 Architecture de Von Neumann . . . 110
6.3 Type d'instructions . . . 111
6.4 Position des opérandes . . . 112
6.5 Mode d'adressage registre . . . 113
6.6 Numéros et noms explicites des registres . . . 113
6.7 Mode d'adressage immédiat . . . 114
6.8 Mode d'adressage indirect immédiat . . . 115
6.9 Notation des modes d'adressage . . . 115
6.10 Liste partielle des instructions arithmétiques et logiques . . . 116
6.11 Liste partielle des instructions chargement/rangement . . . 117
6.12 Mode d'adressage relatif au compteur de programme . . . 118
6.13 Mode d'adressage pseudo-direct . . . 119
6.14 Liste partielle des instructions de branchement . . . 120
6.15 Liste des instructions diverses . . . 120
6.16 Liste partielle des appels système . . . 121
6.17 Liste des types définis par l'assembleur . . . 122

7.1 Opérations UAL . . . 144
7.2 Schéma interne simplifié de l'UAL . . . 144
7.3 Schéma externe du banc de registres . . . 145
7.4 Schéma interne du banc de registres . . . 146
7.5 Schéma mémoire d'instructions . . . 146
7.6 Schéma mémoire de donnée . . . 147
7.7 Schéma compteur de programme . . . 148
7.8 Chemin de données instructions à trois registres . . . 149
7.9 Chemin de données instructions à trois registres . . . 149
7.10 Chemin de données pour les opérations à deux registres et une valeur immédiate 150
7.11 Chemin de données pour une instruction de chargement . . . 151
7.12 Chemin de données pour une instruction de rangement . . . 152
7.13 Chemin de données pour un branchement pris . . . 153
7.14 Chemin de données pour un branchement non pris . . . 154
7.15 Commande MIPS incomplète . . . 156
7.16 Schéma complet de l'architecture . . . 157
7.17 Table de vérité numéro d'opération UAL . . . 158
7.18 Table de vérité des signaux de commande - Énoncé . . . 159

7.19 addi $t0 , $zero , $zero . . . 160
7.20 lui $a0 ,0x1001 . . . 160
7.21 lw $v0 ,0($a0) . . . 161
7.22 addi $a0 , $a0 ,4 . . . 161
7.23 add $t0 , $t0 , $v0 . . . 162
7.24 bne $v0 , $zero ,boucle . . . 162
7.25 bne $v0 , $zero ,boucle . . . 163
7.26 sw $v0 ,0($a0) . . . 163
7.27 Correction table de vérité numéro d'opération UAL . . . 164
7.28 Correction table de vérité signaux de commande . . . 164

8.1 Organisation d'un système . . . 167
8.2 Sources des interruptions . . . 168
8.3 Registre du coprocesseur c0 . . . 169
8.4 Registre $13 ou cause du coprocesseur 0 . . . 169
8.5 Causes de l'exception . . . 169
8.6 Registre $12 ou statu du coprocesseur 0 . . . 170
8.7 Liste partielle des instructions du coprocesseur c0 . . . 170
8.8 Console gérée par interrogation . . . 171
8.9 Organisation de la console . . . 172

9.1 Evolution vitesse microprocesseur - mémoire centrale . . . 187
9.2 Hiérarchie mémoire . . . 188
9.3 Fonction de correspondance directe . . . 190
9.4 Fonction de correspondance directe . . . 191
9.5 Décomposition de l'adresse mémoire . . . 192
9.6 Implémentation d'une mémoire cache directe . . . 193
9.7 Implémentation d'une mémoire cache associative par ensemble de deux . . . 194
9.8 Fonction de correspondance directe . . . 196
9.9 Fonction de correspondance associative . . . 197
9.10 Fonction de correspondance associative par ensemble de deux . . . 199
9.11 Fonction de correspondance directe . . . 200
9.12 Fonction de correspondance associative par ensemble de 4 . . . 200

10.1 Traitement séquentiel d'une instruction . . . 205
10.2 Principe du pipeline d'instruction . . . 206
10.3 Diagramme temporel régulier des instructions précédentes . . . 207
10.4 Diagramme temporel aléa LEA instructions UAL . . . 207
10.5 Diagramme temporel envoi et instructions UAL . . . 208
10.6 Diagramme temporel chargement . . . 208
10.7 Diagramme temporel chargement retardé . . . 209
10.8 Diagramme temporel chargement - rangement . . . 209
10.9 Diagramme temporel aléas de branchement pris . . . 209
10.10 Diagramme temporel aléas de branchement non pris . . . 210
10.11 Diagramme temporel aléas de branchement au plus tôt pris . . . 210
10.12 Diagramme temporel branchement retardé d'un cycle . . . 210
10.13 Énoncé *comptage* :Diagramme temporel sans optimisation . . . 211
10.14 Énoncé *comptage* :Diagramme temporel avec envoi . . . 211
10.15 Énoncé *comptage* :Diagramme temporel avec envoi et branchement au plus tôt . . . 212
10.16 Énoncé *comptage* :Diagramme temporel réorganisé avec envoi et branchement au plus tôt : 5 cycles . . . 212

10.17Énoncé *copieMem* :Diagramme temporel sans optimisation 212
10.18Énoncé *copieMem* : Diagramme temporel avec envoi et branchement au plus tôt 213
10.19Énoncé *copieMem* : Diagramme temporel avec envoi et branchement au plus tôt et branchement retardé : 5 cycles 213
10.20Correction *comptage* :Diagramme temporel sans optimisation : 11 cycles 214
10.21Correction *comptage* :Diagramme temporel avec envoi : 7 cycles 214
10.22Correction *comptage* :Diagramme temporel avec envoi et branchement au plus tôt : 6 cycles 215
10.23Correction *comptage* :Diagramme temporel réorganisé avec envoi et branchement au plus tôt : 5 cycles 215
10.24Correction *copieMem* :Diagramme temporel sans optimisation : 10 cycles 215
10.25Correction *copieMem* : Diagramme temporel avec envoi et branchement au plus tôt : 6 cycles 215
10.26Correction *copieMem* : Diagramme temporel avec envoi et branchement retardé d'un cycle : 5 cycles 216

11.1 Additionneur à anticipation de retenue 220
11.2 Arbre d'anticipation de retenue partie 1 221
11.3 Arbre d'anticipation de retenue partie 2 222
11.4 Addition à anticipation de retenue avec arbre 222
11.5 Addition à saut de retenue 224
11.6 Additionneur à sélection de retenue, principe 224
11.7 Additionneur à sélection de retenue 225
11.8 Comparaison vitesse et complexité 225
11.9 Comparaison numérique vitesse et complexité 225
11.10Addition BCL 226

Liste des algorithmes

1 Divisions successives . . . 12
2 Tester la valeur du bit 1 . . . 18
3 Positionner le bit 1 à 1 . . . 19
4 Valeurs minimale et maximale en utilisant ^ . . . 20
5 Conversion d'un caractère majuscule en minuscule . . . 24
6 Conversion d'un nombre non signé en un caractère . . . 25
7 Comptage du nombre de bits à 1 - énoncé . . . 30
8 Nombre de zéros à droite - énoncé . . . 30
9 Poids du bit à 1 le plus à droite - énoncé . . . 31
10 Inversion des bits - énoncé . . . 32
11 Comptage rapide du nombres de bit à 1 - énoncé . . . 34
12 Conversion d'un digit en un caractère - énoncé . . . 35
13 Conversion d'un caractère en un digit - énoncé . . . 36
14 Conversion d'un nombre non signé en une chaîne de caractères base donnée - énoncé 36
15 Conversion d'une chaîne de caractère base quelconque en un nombre non signé - énoncé . . . 37
16 Comptage du nombre de bits à 1 . . . 41
17 Nombre de zéros à droite . . . 42
18 Nombre de zéros à gauche . . . 42
19 Poids du bit à 1 le plus à droite . . . 43
20 Poids du bit à 1 le plus à gauche . . . 43
21 Inversion des bits . . . 43
22 Inversion des octets . . . 44
23 Comptage rapide du nombres de bit à 1 . . . 44
24 Conversion d'un digit en un caractère . . . 44
25 Conversion d'un caractère en un digit . . . 44
26 Conversion d'un nombre non signé en une chaîne de caractères base donnée . . . 45
27 Conversion d'un nombre non signé en une chaîne de caractères hexadécimaux . . . 45
28 Conversion d'un nombre non signé en une chaîne de caractères binaire . . . 45
29 Conversion d'une chaîne de caractère base quelconque en un nombre non signé . . . 46
30 Conversion d'un nombre signé en une chaîne de caractères, énoncé . . . 66
31 Conversion d'une chaîne de caractères en un nombre signé, énoncé . . . 67
32 Décalage arithmétique à droite de 1 rang, énoncé . . . 67
33 Décalage arithmétique à droite de n rang, énoncé . . . 68
34 Signe d'un nombre entier, énoncé . . . 68
35 Valeur absolue d'un nombre entier, énoncé . . . 69
36 Poids du bit à 1 le plus à droite rapide, énoncé . . . 69
37 Conversion d'un nombre signé en une chaîne de caractères . . . 71
38 Conversion d'une chaîne de caractères en un nombre signé . . . 71
39 Décalage arithmétique à droite de 1 rang . . . 71

40 Décalage arithmétique à droite de n rangs . . . 72
41 Signe d'un nombre entier . . . 72
42 Valeur absolue d'un nombre entier . . . 72
43 Poids du bit à 1 le plus à droite rapide . . . 72
44 Multiplications successives . . . 80
45 Addition de deux nombres réels positifs format simple précision . . . 84
46 Obtenir ? ? ? d'un réel simple précision - énoncé . . . 90
47 Modifier ? ? ? d'un réel simple précision - énoncé . . . 90
48 Enoncé conversion nombre réel simple précision en une chaîne de caractères . . . 91
49 Enoncé conversion chaîne de caractères hexadécimaux en nombre réel simple précision . . . 92
50 Obtenir le signe d'un réel simple précision . . . 95
51 Obtenir l'exposant d'un réel simple précision . . . 95
52 Obtenir la mantisse d'un réel simple précision . . . 95
53 Modifier le signe d'un réel simple précision . . . 95
54 Modifier l'exposant d'un réel simple précision . . . 96
55 Modifier la mantisse d'un réel simple précision . . . 96
56 Conversion nombre réel simple précision en une chaîne de caractères . . . 96
57 Conversion chaîne de caractères hexadécimaux en nombre réel simple précision . 97
58 Alternative . . . 122
59 itération Tant Que . . . 123
60 Fonction . . . 125
61 Conversion d'un digit en un caractère optimisé (énoncé) . . . 128
62 Initialisation d'une zone mémoire énoncé . . . 128
63 Conversion d'un nombre non signé en une chaîne de caractères hexadécimale - énoncé . . . 129
64 Conversion d'une chaîne de caractères hexadécimal en un nombre non signé - enoncé . . . 130
65 Conversion d'un digit en un caractère optimisé . . . 132
66 Conversion d'un caractère en un digit . . . 132
67 Initialisation d'une zone mémoire . . . 132
68 Copie d'une zone mémoire . . . 133
69 Copie d'une chaîne de caractères . . . 133
70 Conversion d'un nombre non signé en une chaîne de caractères hexadécimale . . . 134
71 Conversion d'une chaîne de caractères hexadécimaux en un nombre non signé . 134
72 Conversion d'une chaîne de caractères hexadécimal en un nombre non signé . . . 135
73 Gestion par interrogation du clavier . . . 168
74 Gestion par interruption du clavier . . . 168
75 Gestionnaire d'exception . . . 170
76 Afficher numéro d'exception . . . 173
77 Énoncé Gestion de la console par interrogation . . . 173
78 Énoncé gestion de la console par exception . . . 174
79 Énoncé Gestion de la console par appels systèmes . . . 175
80 Afficher numéro d'exception . . . 176
81 Gestion de la console par interrogation . . . 176
82 Gestion de la console par exception . . . 177
83 Gestion de la console par appels systèmes . . . 178

Listings

2.1 Expressions contenant des opérateurs des bit à bit 56
2.2 Algorithmes utilisant les opérateurs bit à bit 57
3.1 Algorithme sur les nombres signés 75
4.1 API java des nombres réels 98
4.2 Algorithmes sur les nombres réels 100
6.1 Fichier assembleur minimal 121
6.2 Hello World 122
6.3 Programmation de l'alternative 123
6.4 Factorielle n 124
6.5 n ! avec une fonction 125
6.6 Manipulation zone mémoire 138
6.7 Conversion hexadécimale non signé 139
6.8 Comptage des bits d'une donnée 140
7.1 Test architecture 157
8.1 Générer exceptions 180
8.2 Afficher le numéro d'une exceptions 180
8.3 Entrées / sorties par interrogation 180
8.4 Entrées / sorties interruption 181
8.5 Entrées / sorties avec syscall 181
9.1 InitMem et copieMem avec 512 kilo octets de données 203
10.1 InitMem et copieMem avec delayed branch 218