Introduction à l'architecture des ordinateurs
notes de cours et exercices
DIU-EIL

Emmanuelle Encrenaz & Karine Heydemann
Equipe pédagoqique DIU EIL

2019

Ressource éducative libre distribuée sous Licence Creative Commons Attribution
- Pas d'Utilisation Commerciale -
Partage dans les Mêmes Conditions 4.0 International

Table des matières

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Plan de lecture de ce document

Ce document est issu de cours dispensés en Licence d'Informatique, il a été augmenté et ajusté, pour présenter un panorama accessible des notions d'architecture d'un ordinateur, de la représentation et de l'exécution d'un programme.

Il contient des éléments considérés comme prérequis pour la formation (DIU bloc B3 thème architecture et machines séquentielles) et un support pour les notions visées par la formation.

Il présente plusieurs niveaux de lecture, adaptables selon vos connaissances préalables. Pour faciliter votre parcours du document nous proposons un code couleur (titres de sections, sous-sections et paragraphes) pour identifier les parties indispensables en prérequis à la formation, et celles indispensables pour valider la formation.

- Rouge vif : notions pré-requises à maîtriser avant la formation
- Rouge foncé : contenu de la formation à lire avant la séance
- Marron clair : exemples divers permettant de comprendre les notions, à lire pour les parties pré-requises et préparatrices de la formation
- Bleu : preuve et éléments d'explication utiles à la compréhension mais non requises
- Vert : notions non indispensables pour la formation, permettant d'aller plus loin.

Logiciels supports de ce cours

- Logisim est un logiciel d'édition et simulation de schémas logiques souvent utilisé dans les travaux pratiques autour de la réalisation de circuits logiques. Il est libre et gratuit, et programmé en Java, il fonctionne sous tous les systèmes d'exploitation usuels. Plus d'informations : `http://www.cburch.com/logisim/`.
- Mars un logiciel d'assemblage et de simulation de programmes assembleurs MIPS, développé au Missouri State University. Il est disponible à l'adresse suivante : `http://courses.missouristate.edu/KenVollmar/MARS/`.

Autres ressources Quelques liens vers des ressources permettant d'appronfondir et d'aller plus loin sur cette thématique.

- Livre *Architectures matérielles et logicielles*. P. Amblard, J.-C. Fernandez, F. Lagnier, F. Maraninchi, P. Sicard, Ph. Waille. Chez Dunod. Une version pdf est disponible en ligne `http://www-verimag.imag.fr/~carrier/LivreALM.pdf`
- Notes de cours utiles pour complémenter les transparents de cours : site de Super-Boole `http://www.groupes.polymtl.ca/circuits-logiques/php/manuel.php`, le cours de SupInfo `https://www.supinfo.com/cours/1CPA`
- Le cours en ligne de J.-L. Danger sur l'architecture des microprocesseurs : `https://perso.telecom-paristech.fr/danger/ELEC223/ARCHI_ARM.pdf`
- Plusieurs videos sont disponibles aussi. On ne peut que conseiller les cours de Gérard Berry au collège de France : `https://www.college-de-france.fr/site/gerard-berry/_course.htm`, et notamment son cours de 2008 intitulé *Des circuits aux systèmes sur puce* : `https://www.college-de-france.fr/site/gerard-berry/course-2008-02-01-10h30.htm`. Aussi, le séminaire de Olivier Temam *L'évolution des microprocesseurs* en 2013 `https://www.college-de-france.fr/site/gerard-berry/seminar-2013-04-09-11h00.htm`.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Chapitre 1

Architecture générale d'un ordinateur

Un ordinateur, dont l'architecture générale est représentée sur la figure 1.1 comprend :
— un processeur (ou CPU)
— une mémoire centrale (ou RAM ou mémoire vive)
— un bus
— des périphériques

Le CPU est l'unité de traitement de l'information (instructions et données), il exécute des programmes (des instructions qui définissent le traitement à appliquer à des données).

La mémoire centrale est une unité de stockage temporaire des informations nécessaires à l'exécution d'un programme. Elle est externe au processeur. Elle stocke en particulier d'une part les instructions du programme en cours d'exécution ou à exécuter et d'autre part les données du programme (nombre, caractères alphanumériques, adresses mémoires...).

Les périphériques sont des unités connexes permettant de communiquer avec l'ensemble processeur-mémoire. Exemples : clavier, écran, disque dur, CD-ROM, réseau, imprimante/scanner...

Le bus est le support physique des transferts d'informations entre les différentes unités.

L'architecture d'un processeur séquentiel, comme représentée sur la figure 1.2 comporte 2 parties :
— une unité de commande qui analyse les instructions à exécuter et qui séquence les actions élémentaires permettant leur réalisation,
— une partie opérative qui comprend les outils permettant de réaliser les actions élémentaires (ordonnées par l'unité de commande). Elle comprend une unité arith-

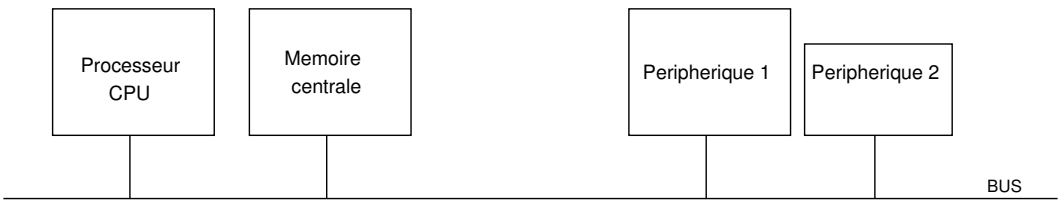


FIGURE 1.1 – Architecture générale d'un ordinateur

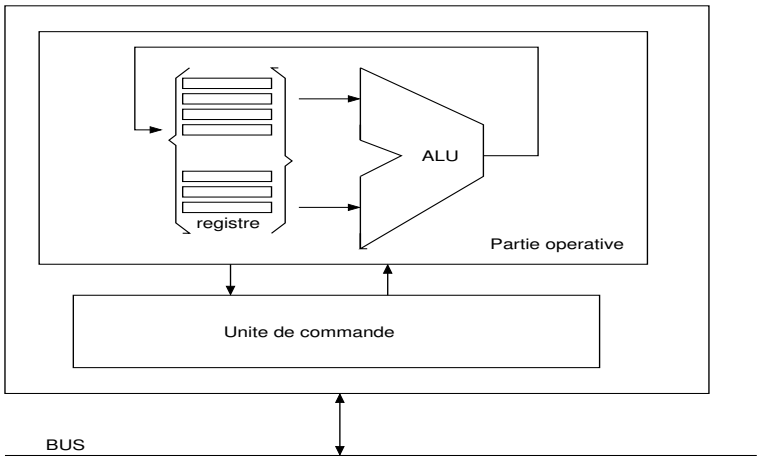


FIGURE 1.2 – Aperçu de l'architecture interne d'un processeur

métique et logique (UAL ou encore ALU) permettant de réaliser des opérations arithmétiques sur des entiers et des opérations logiques sur des vecteurs de bits. Elle comprend également des registres internes permettant de stocker de façon très temporaire des opérandes ou résultats intermédiaires de calcul.

La réalisation physique de ces éléments est électronique (et mécanique pour certains périphériques). Les grandeurs manipulées au sein des composants sont des tensions électriques qui sont stockées dans des éléments mémorisants ou qui transitent sur des fils et traversent des portes combinant les tensions et réalisant des fonctions logiques. On distingue deux niveaux de tension (0V et 1.5V) qui représentent deux valeurs distinctes nommées 0 et 1. Toutes les informations manipulées par l'ordinateur sont représentées par des mots composés de 0 et de 1. La représentation des informations est dite binaire car formé sur l'alphabet {0,1}. Les traitements réalisés sur ces informations sont faites à partir d'une représentation binaire (des données et des traitements à réaliser) sont des fonctions logiques.

Cette organisation se retrouve dans de très nombreux systèmes numériques : ordinateurs (portables), téléphones mobiles, ... comme l'illustre la figure 1.3. L'élement central est le micro-processeur , auquel sont reliés la mémoire (FLASH et SRAM), et différents périphériques comme l'écran et le clavier (LCD Screen, keypad) mais également des dispositifs permettant l'émission et la réception radio, nécessitant des modules de convertion analogique-digitale. Dans cette vue, les éléments mémoire et périphériques sont directement reliés au processeur, le bus n'est pas représenté, mais il est bien présent en réalité.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

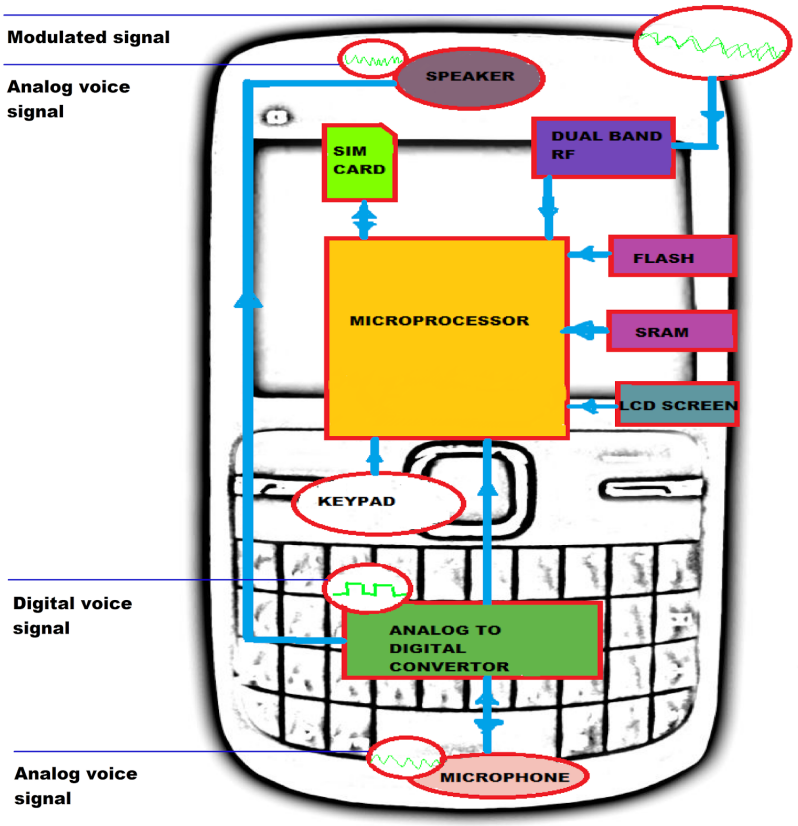


FIGURE 1.3 – Vue de l'architecture interne d'un téléphone mobile

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Chapitre 2

Logique booléenne et circuit combinatoire

Contents

Les traitements réalisés par l'ordinateur sont faits sur des représentations binaires des données et des traitements à réaliser, et ce en calculant des fonctions logiques.

Ce chapitre présente la logique booléenne et l'algèble de Boole, ainsi que les circuits combinatoires.

Définitions

- **mot binaire** : un mot formé sur l'alphabet {0,1}
- **bit** : 0 ou 1
- **octet** : mot binaire composé de 8 bits. Exemple : 01011111
- **quartet** : mot binaire composé de 4 bits. Exemple : 0000, 1010, ...
- **mot MIPS** : mot de 32 bits un mot de 4 octets

2.1 Logique booléenne et algèbre de Boole

La logique booléenne est une formalisation des raisonnements basés sur des éléments qui peuvent être soit vrais, soit faux.

Soit B l'alphabet B= {FAUX, VRAI} = {F,V} = {0,1}.
On définit un ordre sur les éléments de B : 0 < 1

Définitions

- **variable booléenne** : une variable pouvant contenir soit vrai, soit faux.
- **fonction booléenne** : une fonction de $B^n \rightarrow B$
- **table de vérité** : énumération ligne à ligne des valeurs prises par une fonction f en fonction de la valeur de ses paramètres

Exemple : $f : B^2 \rightarrow B$

x,y	$\rightarrow$	si	x = 0,	y = 0	alors	f = 1
			x = 0	y = 1		f = 0
			x = 1	y = 0		f = 0
			x = 1	y = 1		f = 1

Table de vérité de f :

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

Dénombrement du nombre de fonction booléennes à n variables il y a $2^4 = 16$ fonctions booléennes de 2 variables : il y a 2 valeurs possibles pour chaque paramètre, soit 2^2 configurations possibles pour les valeurs d'entrées. Pour chacune des configurations des valeurs d'entrées, il y a 2 valeur de sortie possible (0 ou 1). Il y a donc 2^{2^2} fonctions différentes.

De manière générale, il y a 2^{2^n} fonctions booléennes de n variables : il y a 2 valeurs possibles pour chaque paramètre, soit 2^n configurations possibles des valeurs d'entrées. Il y a 2 valeurs possibles pour la sortie (0 ou 1) pour chacune des 2^n configurations. Il y a donc 2^{2^n} fonctions booléeens différentes à n variables.

2.1.1 Opérations logiques

- Le **complément, appelé aussi NON/NOT et noté par le surlignage**, est une fonction unaire. Si $a = 0$ alors $\overline{a} = 1$ et si $a = 1$ alors $\overline{a} = 0$
- L'**addition, appelée aussi OU/OR et notée +**, est une fonction binaire et définie par $a + b = max(a,b)$
- La **multiplication, appelée aussi ET/AND et notée .**, est une fonction binaire et définie par $a.b = min(a,b)$.

Voici la table de vérité de ces opérations :

x	y	$\overline{x}$	$x + y$	$x.y$
0	0	1	0	0
0	1	1	1	0
1	0	0	1	0
1	1	0	1	1

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

2.1.2 Algèbre de Boole

Définition

$< B, 0, 1, +, ., - >$ forme une algèbre de Boole car il respecte les axiomes suivants :

- L'addition et la multiplication sont associatives :
  $\forall x, y, z \in B^3 \ (x + y) + z = x + (y + z)$
  $\forall x, y, z \in B^3 \ (x.y).z = x.(y.z)$
- L'addition et la multiplication sont commutatives :
  $\forall x,y \in B^2 \ x + y = y + x$
  $\forall x,y \in B^2 \ x.y = y.x$
- 0 est l'élément neutre pour l'addition et 1 pour la multiplication :
  $\forall x \in B \ x + 0 = x$
  $\forall x \in B \ x.1 = x$

- L'addition et la multiplication sont distributives l'une par rapport à l'autre (ce qui est différent de l'algèbre sur les nombres) :
  $\forall x, y, z \in B^3 \ x.(y + z) = x.y + x.z$
  $\forall x, y, z \in B^3 \ x + (y.z) = (x + y).(x + z)$
- La somme d'un élément et de son complément est 1 :
  $\forall x \in B \ x + \overline{x} = 1$
- Le produit d'un élément et de son complément est 0 :
  $\forall x \in B \ x.\overline{x} = 0$

Autres propriétés

Théorème de dualité :

si $< B, 0, 1, +, ., - >$ est une algèbre de Boole alors $< B, 1, 0, ., +, - >$ en est une également.

Loi de De Morgan :

$\overline{x + y} = \overline{x}.\overline{y}$

$\overline{x.y} = \overline{x} + \overline{y}$

Règle de simplification

loi d'involution : $\forall x \in B \quad \overline{\overline{x}} = x$

éléments absorbants : $\forall x \in B \quad x.0 = 0$ et $x + 1 = 1$

idempotence : $\forall x \in B \ x.x = x$ et $x + x = x$

Exemple : $\forall x, y \in B^2 \quad x.(x + y) = x + x.y$ (distributivité et idempotence)

2.1.3 Expression algébrique d'une fonction booléenne

Toute fonction booléenne f peut s'exprimer à partir des constantes 0 et 1, des noms des variables booléennes paramètres de f et des opérations +, . et - de l'algèbre de Boole.

On peut construire une expression algébrique d'une fonction f à partir de sa table de vérité.

2.1.3.1 Forme normale disjonctive

On peut construire une expression de f représentant les configurations des paramètres pour lesquelles f vaut 1. La fonction booléenne f vaut 1 quand on est dans une des combinaisons de valeurs des paramètres aboutissant à 1 en sortie. Chacun de ces cas peut être traduit en une expression algébrique. Par exemple, si x doit valoir 0 et y doit valoir 0 alors $\overline{x}.\overline{y}$ vaut 1 si et seulement si $x = 0$ et $y = 0$.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Exemple avec une fonction f :

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

$\rightarrow \quad \overline{x}.\overline{y}$

$\Rightarrow f = \overline{x}.\overline{y} + x.y$

$\rightarrow \quad x.y$

En pratique : une expression de f peut être construite en réalisant la disjonction (OU) des termes représentant les lignes où f vaut 1. Chaque terme est le produit (ET) des noms de variables de f, complémentés si la contribution de la variable est 0.

2.1.3.2 Forme normale conjonctive

On peut aussi construire une expression de f à partir des lignes de sa table de vérité qui valent 0. C'est la forme normale conjonctive.

La fonction f vaut 1 si et seulement si on n'est dans aucun des cas à 0. Soit e_i l'expression algébrique qui vaut 1 lorsque les paramètres ont une valeur qui correspond à la ième ligne de la table de vérité. Si f vaut 0 pour les lignes j, k et l alors on a $f = \overline{e_j}.\overline{e_k}.\overline{e_l}$: f vaut 1 si l'on n'est pas dans un des cas qui vaut 0.

Comme l'expression des e_i est un produit des variables paramètres ou de leur complément, alors en appliquant la loi de De Morgan $\overline{e_i}$ est une somme des compléments des variables (dont la contribution est 1) ou de leur compléments (si la contribution est 0). En appliquant alors la loi d'involution, on en déduit que $\overline{e_i}$ est la somme des variables paramètres si leur contribution est 0 ou de leur complément si leur contribution est 1.

Exemple avec la fonction f :

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

$\rightarrow \quad \overline{\overline{x}.y} = x + \overline{y} \quad \Rightarrow f = (x + \overline{y}).(\overline{x} + y)$

$\rightarrow \quad \overline{x.\overline{y}} = \overline{x} + y$

En pratique : la forme normale conjonctive est le produit des termes représentant les lignes où f vaut 0. Chaque terme est la sommes des noms de variable, complémentés si la contribution est 1.

Remarque : on peut retrouver la forme normale conjonctive à partir de la forme normale disjonctive en utilisant la loi d'involution $\overline{\overline{f}} = f$.

$\overline{f}$ vaut 1 quand f vaut 0 et FNC($\overline{f}$) = $\overline{x}.y + x.\overline{y}$.
Comme f vaut $\overline{\overline{f}}$ on a : $f = \overline{\overline{x}.y + x.\overline{y}}$
En appliquant la loi de De Morgan plusieurs fois :
$f = \overline{\overline{x}.y}.\overline{x.\overline{y}}$
$f = (\overline{\overline{x}} + \overline{y}).(\overline{x} + \overline{\overline{y}})$
f = $(x + \overline{y}).(\overline{x} + y)$

On retrouve le produit des termes correspondant aux lignes de la table de f où celle-ci vaut 0.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

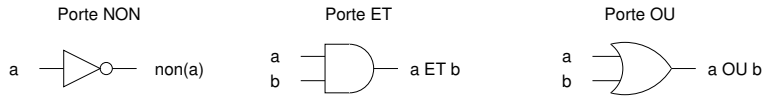


FIGURE 2.1 – Représentation schématique des portes ET, OU et NON

2.1.3.3 Equivalence d'expressions algébriques booléennes

On peut montrer l'équivalence d'expressions algébriques booléennes en utilisant une des deux équivalences ci-dessous.

1. Deux fonctions booléennes sont équivalentes si elles ont la même table de vérité.
2. Deux expressions algébriques booléennes sont équivalentes si elles peuvent se réécrire en une même expression.

Exemple. On cherche à montrer que :

$$a + (b.c) = (a + b).(a + c)$$

On peut le montrer avec la table de vérité de chacun des expressions :

a	b	c	bc	$a+bc$	$a+b$	$a+c$	$(a+b)(a+c)$
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	0	1	0	0
0	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1
1	0	1	0	1	1	1	1
1	1	0	0	1	1	1	1
1	1	1	1	1	1	1	1

On peut aussi le montrer en développant le membre droit de l'égalité puis en utilisant des propriétés/simplifications de l'algèble de Boole pour aboutir au membre gauche de l'égalité :

Développement de la partie de droite : $a.a + a.b + a.c + b.c$
Simplification par la mise en facteur de a : $a.(1 + b + c) + b.c$
Comme $1 + b + c = 1$ on a : $a.1 + b.c$
Soit $a + b.c$.

2.2 Représentation schématique des fonctions booléennes

Un circuit est une représentation schématique de l'évaluation d'une fonction booléenne à partir de l'une de ses expressions algébriques.

Porte logique Une *porte logique* représentant une fonction booléenne f à n variables est un élément possédant n signaux d'entrée, un signal de sortie et produisant sur le signal de sortie la valeur de f pour la configuration fournie sur les signaux d'entrée.

La figure 2.1 illustre la représentation schématique des portes ET, OU et NON.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

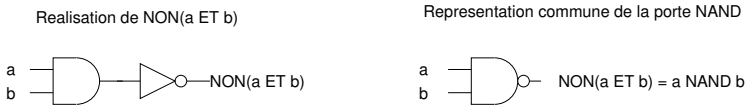


FIGURE 2.2 – Réalisation de la fonction booléenne $\overline{a.b}$ à partir de porte ET, OU et NON ainsi que la représentation schématique de la porte NAND

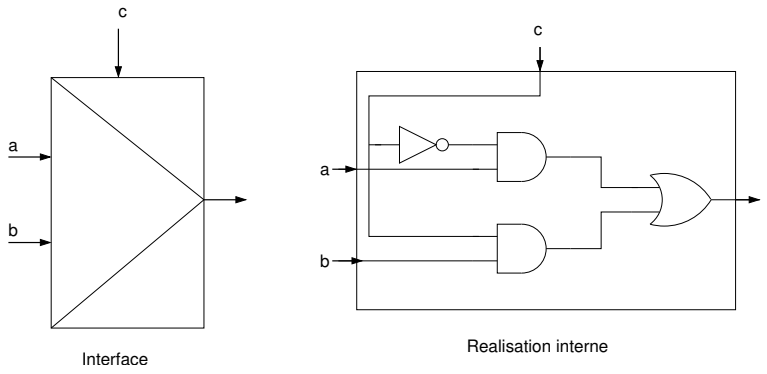


FIGURE 2.3 – Interface et réalisation interne d'un multiplexeur à deux entrées

Circuit logique Un *circuit logique* est un diagramme orienté composé de signaux d'entrée, de portes logiques, de signaux de sortie et respectant les règles de connectique suivantes :

— les entrées des portes logiques sont connectées aux signaux d'entrée du circuit ou aux sorties d'autres portes du circuit,
— les signaux de sortie sont connectés aux sorties de portes du circuit,
— la composition de fonction $g \circ f$ est représentée par la mise en séquence de f et de g : les signaux de sortie de f sont connectés aux signaux d'entrée de g

Exemple

La figure 2.2 montre la réalisation de la fonction booléenne $\overline{a.b}$ à partir de porte ET, OU et NON ainsi que la représentation schématique de la porte NAND.

Multiplexeur Un multiplexeur à deux entrées a et b et une commande c est un circuit permettant de mettre en sortie la valeur de l'entrée a si c vaut 0 ou la valeur de l'entrée b sinon (si c vaut 1). Un multiplexeur permet de sélectionner une des entrées. On a mux = $a.\overline{c} + b.c$

La figure 2.3 montre l'interface et la réalisation interne d'un tel multiplexeur à deux entrées.

Circuit combinatoire C'est un circuit logique dont le graphe orienté ne contient pas de cycle. La valeur du signal de sortie ne dépend que des valeurs des signaux d'entrée à l'instant présent (dès qu'une entrée change, la sortie change – modulo le temps de propagation de l'information qui est très rapide).

Logisim Logisim est un logiciel d'édition et simulation de schémas logiques souvent utilisé dans les travaux pratiques autour de la réalisation de circuits logiques. Il est libre et

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

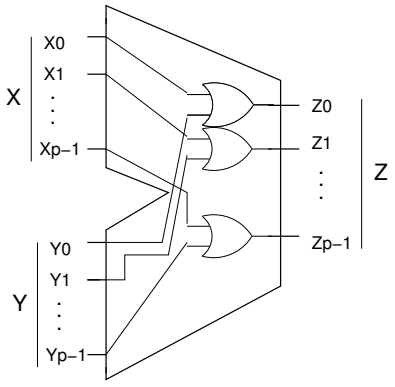


FIGURE 2.4 – Réalisation d'une opération OU sur deux vecteurs de p bits dans l'ALU

gratuit, et programmé en Java, il fonctionne sous tous les systèmes d'exploitation usuels. Plus d'informations : `http://www.cburch.com/logisim/`.

2.3 Unité arithmétique et logique

Les processeurs disposent d'instructions permettant de réaliser des opérations logiques sur des mots de p bits. L'application d'une fonction booléenne (ET, OU, NON) sur des vecteurs (mots) de p bits revient à l'application de la fonction sur chacun des bits du mots ou sur les bits de même rang dans les mots d'entrée pour les opérations n-aires. Cela revient à la mise en parallèle de p opérateurs logiques dans l'ALU.

Si f est une fonction booléenne binaire alors sur deux mots de p bits :

$f(a_{p-1}...a_1a_0, b_{p-1}...a_1a_0) = f(a_{p-1},b_{p-1})f(a_{p-2},b_{p-2})...f(a_1,b_1)f(a_0,b_0)$

Par exemple, `or $3, $2, $1` est une instruction des processeurs MIPS qui réalise un OU logique entre deux mots de 32 bits (ici ceux contenus dans les registres `$2` et `$1`) du processeur). Le résultat est placé dans un registre du processeur (ici dans `$3`). Certains registres du processeur sont explicitement adressables dans les instructions, nous reviendrons dessus dans les chapitres sur les registres et le jeu d'instructions.

Dans l'ALU, le circuit logique réalisant un OU sur des mots de p bits est activé lorsque le processeur exécute une telle instruction `or $3, $2, $1`. Pour ce faire, les opérandes (valeurs contenues dans `$2` et `$1`) sont placés sur les entrées X et Y de L'ALU et le résultat est émis sur Z. Il est ensuite écrit dans le registre `$3`. La figure 2.4 montre la réalisation d'une opération OU sur des vecteurs de p bits dans l'ALU.
Par exemple si p vaut 4 : 1001 OU 0100 = 1101 (OU logique sur les bits de même rang)

Les processeurs disposent d'autres opérations logiques. Par exemple `and $1, $2, $3` est l'instruction qui place dans `$1` le résultat du ET logique entre les mots binaires contenus dans les registres `$2` et `$3`.
L'ALU contient le circuit logique permettant de réaliser cette opération (soit p portes ET mises en parallèle).

L'ALU dispose de plusieurs circuits logiques précablés permettant de construire les résultats de calculs prédéfinis (spécifiques, ET, OU, NON, XOR...) Lors de l'exécution d'une instruction, le résultat du circuit associé à l'instruction est présenté sur la sortie de l'ALU, ce résultat est sélectionné (parmi les résultats des circuits précablés existant dans l'ALU) par un multiplexeur commandé avec l'instruction réalisée.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

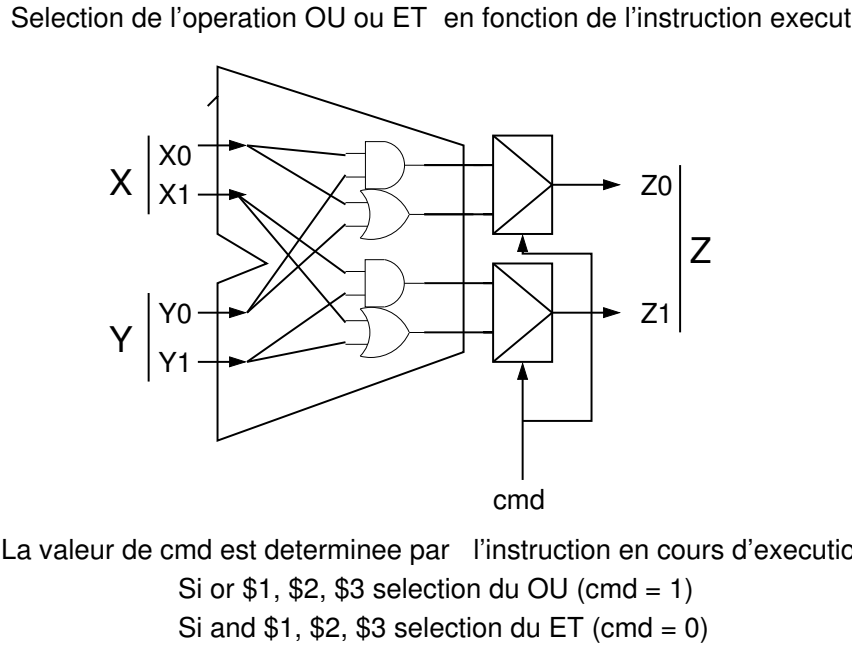


FIGURE 2.5 – Réalisation de la sélection de l'opération (OU ou ET) à la sortie de l'ALU en fonction de l'instruction exécutée

La figure 2.4 montre la réalisation de la sélection de l'opération (OU ou ET) à la sortie de l'ALU en fonction de l'instruction exécutée.

2.4 Exercices

Exercice 1 – Prise en main de logisim

Lancez Logisim en tapant dans un terminal la commande "logisim &"
Une documentation (en anglais, espagnol, russe mais pas en francais) est disponible en passant par le menu Help de Logisim.
L'interface avec quelques élements est présentée dans la figure 2.6.

Question 1
Construisez un circuit qui réalise la fonction f = (a ET NON b).

Pour information, la couleur des fils dans logisim vous donne des indications :

- vert foncé : valeur 0 transitant sur le fil de largeur 1
- vert clair : valeur 1 transitant sur le fil de largeur 1
- noir : indique une nappe de fils (plusieurs bits)
- bleu : aucune valeur (pas de connexion)
- rouge : erreur
- orange : problème de compatibilité entre la largeur d'une nappe de fils/port d'un circuit ou d'une porte

Selectionnez une porte AND et ajustez le nombre des entrées pour qu'il n'y en ait que 2. La figure 2.7 vous montre où changer le nombre d'entrées.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

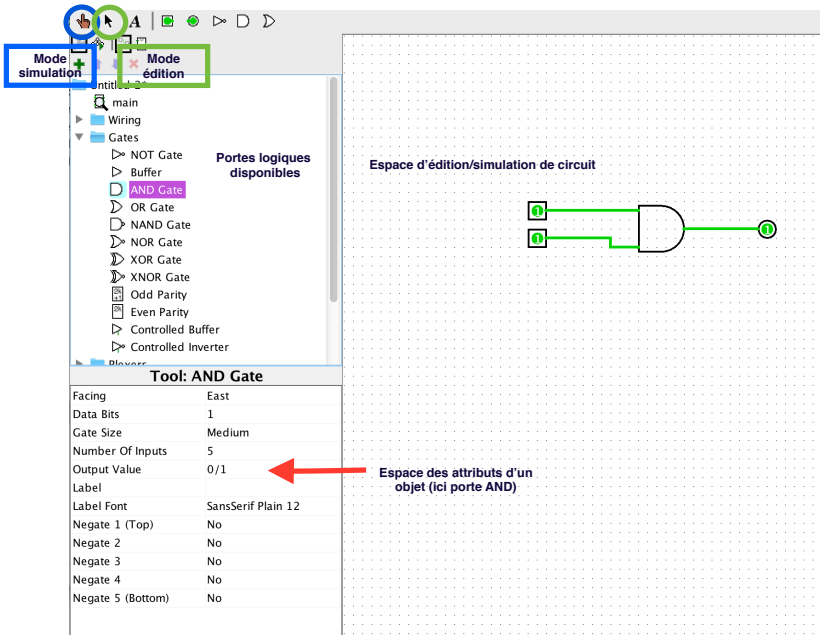


FIGURE 2.6 – Interface de logisim avec un exemple de circuit composé d'une porte AND

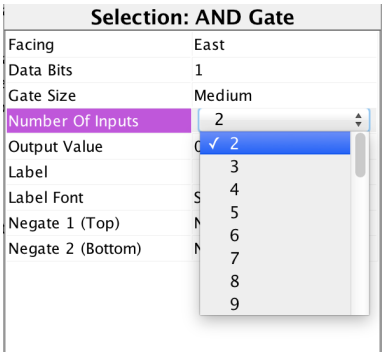


FIGURE 2.7 – Sélection du nombre d'entrée d'un AND

Etiquetez les deux entrées de votre circuit en indiquant leur nom dans le champ label des attributs des entrées. La figure 2.8 vous montre où est ce champ.

Simulez le circuit et comparez le résultat obtenu avec la table de vérité de la fonction f. Pour cela, mettez vous en mode simulation et faites varier les valeurs en entrée (en cliquant sur les entrées) pour couvrir l'ensemble des configuration possible.

Question 2

Trouvez une expression algébrique pour la fonction f = a XOR b. Complétez votre circuit pour réaliser cette fonction. Nommez mon_xor le circuit construit et sauvegardez le projet contenant le circuit sous le nom TME1.

Exercice 2 – Construction d'un multiplexeur à 2 entrées

Un multiplexeur est un circuit à plusieurs entrées (notées e_0, e_1, ...) avec un sélecteur dont la taille est fonction du nombre des entrées. Le rôle du sélecteur est de coder un numéro

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

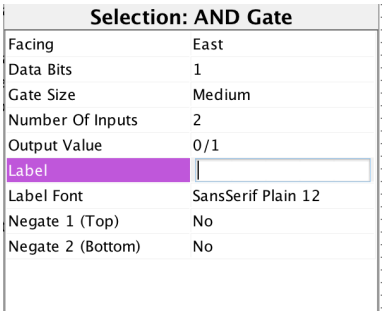


FIGURE 2.8 – Labelisation d'un AND

d'entrée. Le multiplexeur doit alors produire en sortie la valeur de cette entrée. Dans cas d'un circuit à 2 entrées numérotées 0 et 1, on prend donc un sélecteur i sur 1 bit. La sortie s du circuit peut alors s'écrire :

— s = e_0 ssi i = 0
— s = e_1 ssi i = 1

Question 1
Donnez l'expression booléenne du multiplexeur décrit ci-dessus.

Question 2
Construisez un circuit qui réalise la fonction s. Pour cela, ajouter un circuit à votre projet que vous nommerez MUX2. Vérifiez votre circuit par simulation.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

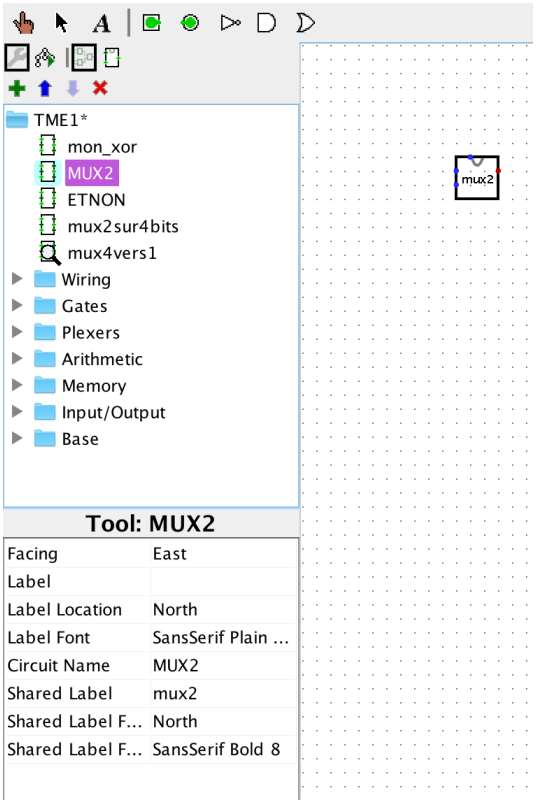


FIGURE 2.9 – Utilisation de mux2

Exercice 3 – Construction d'un multiplexeur à 2 entrées sur 4 bits

Question 1
Construisez un multiplexeur à 2 entrées A=$a_3a_2a_1a_0$ et B=$b_3b_2b_1b_0$ sur 4 bits UNIQUEMENT à partir de multiplexeurs 2 entrées. La sortie S=$s_3s_2s_1s_0$ est aussi sur 4 bits (et prend soit la valeur de A, soit la valeur de B), et il y a un bit de sélection i.

Vous utiliserez comme multiplexeur à 2 entrées sur 1 bit le circuit que vous avez réalisé à la question précédente. Pour cela, cliquez sur la petite boite à coté de MUX2 et ajouter le circeuit dans la partie édition. La figure 2.9 vous montre comment faire. Le circuit mux2 a comme interface : 3 entrées (sur le schéma 2 à gauche et 1 en haut) repérables par les points bleus sur le contour et une sortie repérable par le point rouge.
Vous pouvez aussi voir sur la figure que le projet TME1 contient tous les circuits à réaliser pour ce TP.

Exercice 4 – Construction d'un multiplexeur à 4 entrées

On désire maintenant construire un multiplexeur à 4 entrées, chacune sur 1 bit. Le sélecteur doit donc pouvoir prendre 4 valeurs différentes, et la valeur du sélecteur détermine le numéro de l'entrée sélectionnée.

Question 1
Quelle doit être la taille (en nombre de bits) du sélecteur ?

Question 2
Construisez un multiplexeur à 4 entrées 1 bits UNIQUEMENT à partir de multiplexeurs 2

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

entrées : pour cela donner l'expression booléenne du multiplexeur 4 entrées e_0, e_1, e_2, e_3, et manipuler cette expression pour y voir apparaître (plusieurs fois) celle d'un multiplexeur 2 entrées.

Question 3
Comment généraliser la construction de multiplexeurs à 2^n entrées à partir de multiplexeurs à 2 entrées ? Expliquez la solution pour un multiplexeur à 8 entrées puis à 16 entrées et généraliser.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Chapitre 3

Représentation des données

Contents

Les informations traitées par un ordinateur sont de différents types mais sont toujours représentées sous forme binaire. C'est l'utilisation de l'information qui en est faite qui détermine le type.

L'information élémentaire est le bit et les informations plus complexes telles qu'un caractère, un nombre se ramène à un ensemble de bits.

Le codage d'une information consiste à établir une correspondance entre la représentation externe de l'information (caractère A ou nombre 36) et sa représentation binaire c'est-à-dire une suite de bits.

La représentation binaire est facile à réaliser (2 états d'équilibre) et les opérations fondamentales sont relativement simples à effectuer sous forme de circuit logique (cf. cours 1 pour traduire une fonction en une expression algébrique puis en circuit).

Les informations traitées par l'ordinateur sont des instructions : on parle de codage des instructions et des données qui peuvent être numériques (N, Z, ...) ou alphanumérique (ASCII, UTF-8, ...).

3.1 Représentation des entiers naturels N

Système de numération Un **système de numération** fait correspondre à un nombre N un certain formalisme écrit et oral.

Représentation en base B Dans un système de base avec $B > 1$ les nombres 0, 1, 2, ..., $B-1$ sont appelés **chiffres**.

Tout entier naturel peut être exprimé comme une somme de multiples de puissance de la base, les multiples étant des chiffres (soit inférieurs à la base strictement). En base B :

$$N_d = \sum_i a_i B^i = a_n B^n + a_{n-1} B^{n-1} + ... + a_1 B + a_0 \text{ avec } \forall\ i\ a_i < B$$

La notation condensée est $N_B = a_n a_{n-1} ... a_1 a_0{}_B$.
On utilisera les indices b pour la base 2 (ex : 111_b) ou le binaire, d pour la base 10 (ex : 111_d) ou le décimal et h pour la base 16 ou l'hexadécimal (ex : 111_h).

On appelle notation par position pondérée cette notation : chaque position correspond à une puissance de la base B, il y a un chiffre pour chaque position (éventuellement 0). Cela est différent des chiffres romains par exemple.

Représentation en base 2
Les informations manipulées par un ordinateur étant des mots binaires, les nombres entiers sont représentés en base 2, les chiffres sont alors 0 et 1.
En base 2 :

$$N_d = a_n a_{n-1} ... a_1 a_0{}_b \text{avec } \forall\ i\ a_i < 2$$

$$N_d = \sum_i a_i 2^i = a_n 2^n + a_{n-1} 2^{n-1} + ... + a_1 2 + a_0$$

Exemple :
$N = 110011_b = (1 * 2^5 + 1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^0 + 1 * 2^0)_d$
$N_d = (2^5 + 2^4 + 2^1 + 2^0)_d = 51_d$

Représentation en base 16
Les nombres entiers exprimés en binaire sont souvent composés d'un grand nombre de bits. On préfère les exprimer en base 16 dite en hexadécimal car la conversion depuis et vers le binaire est simple. En base 16, on utilise les symboles 0, 1, ..., 8, 9, A, B, C, D, E,F pour les 15 chiffres avec la correspondance donnée dans la figure **??**

En hexadécimal, on a :

$$N_d = a_n a_{n-1} ... a_1 a_0{}_h \text{avec } \forall\ i\ a_i < 16$$

$$N_d = a_{n-1} 16^{n-1} + a_{n-2} 16^{n-2} + ... + a_1 16 + a_0$$

Il y a quatre fois moins de symboles que dans la notation en binaire.
Attention $1001_h \neq 1001_b$!

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Hexadécimal	Décimal	Binaire
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

FIGURE 3.1 – Chiffre hexadécimal et leur représentation en décimal et binaire

Taille bornée Les nombres sont représentés sur des mots de taille bornée (par exemple sur 32 ou 64 bits). En conséquence, tous les nombres ne sont pas représentables dans un ordinateur.

Intervalle de représentation Sur p symboles, on peut représenter les entiers en base B compris dans l'intervalle $[0, B^p - 1]$.
Exemple :
Sur 3 digits en décimal [0, 999]
Sur 3 bits en binaire [0,7]
Sur 3 bits en hexadécimal [0, 4095]

Extension Quelque soit la base B, un entier représenté sur p symboles peut être étendu sur $n > p$ symboles en introduisant des 0 sur les symboles des rangs p à n - 1.
Exemple extension de 4 à 8 bits :
$0011_b \rightarrow 00000011_b$
$1001_b \rightarrow 00001001_b$

3.2 Changement de base

Dans cette partie, on s'intéresse aux algorithmes pour passer d'une base B_1 à une base B_2.

3.2.1 Algorithme de conversion par divisions successives

Algorithme L'algorithme de conversion par divisions successives donné ci-dessous permet de passer de la base 10 à une base $B > 1$ pour un nombre N donné.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

$i \leftarrow 0$
while $i \leq 0$ **do**
  $(Q,R) \leftarrow \frac{N}{B}$
  $a_i \leftarrow R$
  $N \leftarrow Q$
  $i \leftarrow i + 1$
end while
$a_i \leftarrow 0$
Return $a_{j,j\in[0,i]}$

Par division successives, on obtient à la ième itération la valeur de a_i. Lors de la première itération, on divise l'entier par 2, et on obtient alors le reste R_0 (tel que $N = Q_0 * 2 + R_0$) qui correspond à a_0. Lors de la deuxième itération, on divise $N/2$ (ou Q_0) par deux, on obtient alors le reste R_1 tel que $Q_0 = Q_1 * 2 + R_1$. R_1 correspond à a_1. On voit apparaitre la décomposition de N que l'on est en train d'obtenir $N = (Q_1 * 2 + R_1) * 2 + R_0 = Q_1 * 4 + R_1 * 2 + R_0$.

Exemple La figure 3.2 montre la conversion de 25_d en binaire par divisions successives.

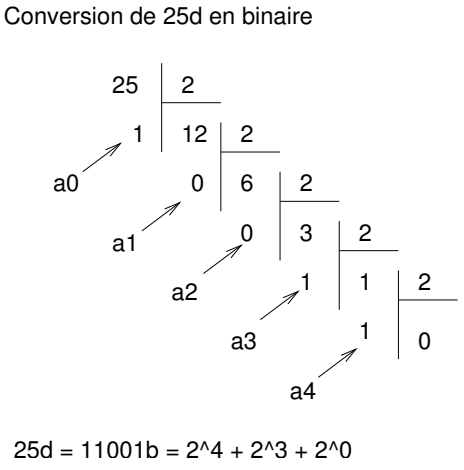


FIGURE 3.2 – Conversion de 25_d en binaire par divisions successives

3.2.2 Conversion de la base 10 vers la base B sur (k+1) symboles

Algorithme L'algorithme donné ci-dessous permet de convertir un nombre N donné dans une base $B > 1$ sur $(k + 1)$ symboles en cherchant les multiples des puissances de B.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

$i \leftarrow k$
while $N >= 0$ and $i \geq 0$ **do**
  if $N \geq \alpha.B^i$ and $N < (\alpha+1).B^i, \alpha \in [0, B-1]$) **then**
    $b_i \leftarrow \alpha$
    $N \leftarrow N - \alpha.B^i$
  else
    $b_i \leftarrow 0$
  end if
  $i \leftarrow i - 1$
end while
Return $b_{i,i\in[0,k]}$

Exemple Conversion de $N = 687_d$ en hexadécimal.
On suppose que k vaut 3 et on rappelle que $16^3 = 4096$ et $16^2 = 256$.

- Au début on a donc k = 3 et i = 3.
- Pour i = 3 : comme $N = 687_d \leq 16^3$ on a $b_3 = 0_h$.
- Pour i = 2 : on a
  - $687_d > 256_d$,
  - $687_d > 2 * 256_d = 512_d$ et
  - $687_d < 3 * 256_d = 768_d$
  - donc $b_2 \leftarrow 2_h$ et $N \leftarrow 687_d - 512_d$ soit $N \leftarrow 175$.
- Pour i = 1 : on a
  - $175_d > 10 * 16_d = 160$ et
  - $175_d < 11 * 16_d = 176_d$
  - donc $b_1 \leftarrow 10_d = A_h$ et $N \leftarrow 175_d - 160_d$ soit $N \leftarrow 15_d$.
- Pour i = 0 : $b_o = 15_d = F_h$.

On a donc : $687_d = 02AF_h$.

Utilisation d'un tableau de puissances À la main, il est aisé de réaliser cette conversion avec un tableau de puissances. Par exemple pour convertir en binaire la valeur 39_d sur 6 bits :

			2^6	2^5	2^4	2^3	2^2	2^1	2^0
			a_6	a_5	a_4	a_3	a_2	a_1	a_0
$39 < 2^6$	$\Longrightarrow$	$a_6 = 0$	0	?	?	?	?	?	?
$39 > 2^5$	$\Longrightarrow$	$a_5 = 1$	0	1	?	?	?	?	?
$39 - 2^5 = 7 < 2^4$	$\Longrightarrow$	$a_4 = 0$	0	1	0	?	?	?	?
$7 < 2^3$	$\Longrightarrow$	$a_3 = 0$	0	1	0	0	?	?	?
$7 > 2^2$	$\Longrightarrow$	$a_2 = 1$	0	1	0	0	1	?	?
$7 - 2^2 = 3 > 2^1$	$\Longrightarrow$	$a_1 = 1$	0	1	0	0	1	1	?
$3 - 2^1 = 1\ ge\ 2^0$	$\Longrightarrow$	$a_0 = 1$	0	1	0	0	1	1	0

3.2.3 Conversion de la base 2 à la base 16

Algorithme Pour passer de la base 2 à la base 16, il faut séparer le nombre binaire en quartet (paquet de 4 bits) en partant de la droite puis convertir chaque quartet en son symbole hexadécimal.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Exemple $1010111011110010_b = 1010\ 1110\ 1111\ 0010_b = AEF2_h$

3.2.4 Conversion de la base 16 à la base 2

Algorithme Pour passer de la base 16 à la base 2, il faut convertir chaque symbole hexadécimal en quartet/mot de 4 bits.

Exemple $7DB3_b = 0111\ 1101\ 1011\ 0011_b = 0111110110110011_b$

Exercice 5 – Représentation des entiers naturels

Question 1
Convertissez les nombres naturels suivants.

base 2 (sur 8 bits)	base 10	base 16 (sur 1 octet)
	10_d	
$0000\ 0010_b$		
		10_h
	57_d	
	127_d	
		17_h
		$5B_h$
$0010\ 1001_b$		
$1010\ 1010_b$		

Question 2
Sur combien de bits au minimum peut-on coder en binaire les nombres suivants : 127_d et 32_d.
En déduire combien de nombres on peut coder en binaire dans des mots de longueur n ? Quel est l'intervalle des valeurs entières représentées par un mot binaire non signé de longueur n ?

Question 3
Combien de nombres peut-on coder en hexadécimal dans des mots hexadécimaux de longueur n ? Quel est l'intervalle des valeurs entières représentées par un mot hexadécimal non signé de longueur n ?

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Exercice 6 – Exercices de conversion
Conversion binaire => hexadécimal
$0001\ 1101_b =$
$1001\ 1000\ 0011\ 1100_b =$
Conversion hexadécimal => binaire
$75_h =$
$1A_h =$
$34DF_h =$
Conversion binaire => décimal
$1001\ 0110_b =$
$1100\ 0110_b =$
Conversion décimal => binaire
$57_d =$
$1272_d =$

3.3 Addition et soustraction d'entiers naturels

3.3.1 Addition

En décimal

Addition :
$$\begin{array}{rcc} & {}^{1}7 & 5 \\ + & 1 & 7 \\ \hline & 9 & 2 \end{array}$$

"7 + 5 = 12 je pose 2 et je retiens 1" L'addition des unités dépasse 10, il faut propager une retenue à gauche.

$$\begin{array}{rcc} & 7 & 5 \\ - & 1^{1} & 7 \\ \hline & 5 & 8 \end{array}$$

Lorsque l'on soustrait les unités, on soustrait une valeur plus grande : "7 est plus grand que 5, je pique une dizaine à gauche"
Remarque : la soustraction n'est pas définie si le nombre à soustraire est supérieur au nombre dont on le soustrait. Soit $A - B$ non définie sur les entiers naturels si $B > A$.

3.3.1.1 Addition en base 2

Règles d'addition sur les bits

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 2_d = 10_b donc "je pose 0 et je retiens 1" ;

Lorsqu'il y a une retenue, il faut intégrer la retenue dans le calcul sur les bits à gauche. On a donc non pas 2 mais 3 chiffres à sommer. On peut avoir l'addition de trois 1, et 1 + 1 + 1 = 3_d=11_b, soit "je pose 1 et je retiens 1".

Exemples sur 8 et 4 bits :

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

$$\begin{array}{cccccccccc} & 0^1 & 1^1 & 1 & 0 & 0 & 0 & 1 & 1 \\ + & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ \hline & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \end{array}$$

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

Le résultat ne tient pas sur 4 bits car il y a une retenue sortante pour l'addition du dernier rang (rang 3). Le résultat n'est pas représentable sur 4 bits (ici on additionne 12_d et 6_d et la valeur du résultat sur 4 bits est 2_d).

Dépassement de capacité On dit qu'il y a un *dépassement de capacité* lorsque l'addition de 2 entiers naturels représentés sur n bits ne donne pas un résultat représentable sur n bits. Comme la taille de réprésentation est fixée (n bits), il est nécessaire de pouvoir détecter les dépassements de capacité afin de pouvoir déterminer que le résultat (qui est sur n bits) n'est pas correct (et donc pas utilisable).

3.3.1.2 Addition en n base 16

L'addition en base 16 peut se faire sans passer par la représentation binaire en additionnant les chiffres de même rang et en propageant une retenue lorsque le résultat est supérieur à 15.

Exemple : $\begin{array}{cccc} & 0 & D & 2 \\ + & 5 & 2 & 9 \\ \hline & 5 & F & B \end{array}$ $\begin{array}{cccc} & 1^1 & F & A \\ + & 7 & 3 & 4 \\ \hline & 9 & 2 & E \end{array}$ $\begin{array}{cccc} ^1 & 5 & 6^1 & 5 \\ + & C & 2 & E \\ \hline 1 & 1 & 9 & 3 \end{array}$

Pour la 3ème opération, il y a un dépassement de capacité car le résultat ne tient pas sur 4 symboles en héxadécimal.

3.3.1.3 Circuit logique représentant les calculs réalisés par l'addition de 2 opérandes 1 bit

La figure 3.3 représente l'interface d'un additionneur de deux opérandes 1 bit avec 1 bit de retenue entrante. Il y a 3 entrées : a_i, b_i et c_{in} la retenue entrante (sortant de l'addition des bits de droite). Il y a deux fonctions à calculer : s pour le bit de somme et c_{out} pour le bit de retenue.

La table de vérité de ces deux fonctions est :

a_i	b_i	c_{in}	s	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Exercice 7 – Expressions algébriques pour l'additionneur 1 bit

Donnez une expression booléenne pour c_{out} et s.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

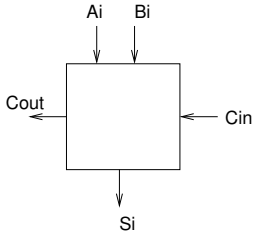


FIGURE 3.3 – Interface de l'additionneur de 2 opérandes un bit avec 1 bit de retenue entrante

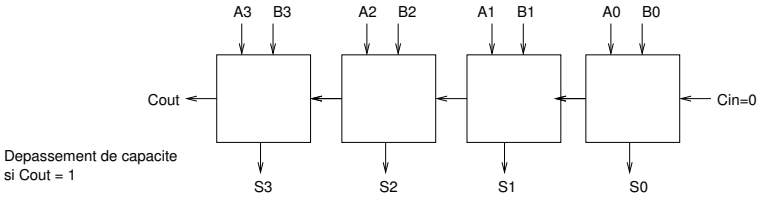


FIGURE 3.4 – Additionneur de 2 opérandes 4 bits réalisé à partir d'additionneurs 1 bit

3.3.1.4 Additionneur n bits

Un additionneur n bits est obtenu en connectant n additionneurs 1 bit de telle sorte que la retenue sortante du rang i soit la retenue entrante du rang $i+1$.
Le circuit final a 2n bits d'entrées (n pour A + n pour B) et ($n+1$) sorties (n bits pour la somme + la retenue sortante du dernier rang).
Il y a un dépassement de capacité lors de l'addition de deux entiers naturels lorsque $c_{out_{n-1}} = 1$.

La figure 3.4 montre la réalisation d'un additionneur 4 bits à partir d'additionneurs 1 bit.

3.3.2 Soustraction

La soustraction suit le même principe que l'addition.
Règle sur les bits :

0 - 0 = 0

1 - 0 = 0

1 - 1 = 0

0 - 1 = impossible : il faut piquer une deuzaine à gauche que l'on retranche ensuite, il y a donc une retenue à gauche qu'il faut soustraire ensuite. Ici le résultat est 1 et il y a une retenue à retrancher à gauche qui vaut 1.

Cela signifie que lors de la soustraction de b_i à a_i dans le calcul de $A - B$ on a a priori à soustraire 2 valeurs, la retenue sortante du rang $i-1$ et b_i et pas une. La soustraction de deux bits produit donc aussi une sortie et une retenue.

Rappel : $A - B$ non défini sur les entiers naturels si $B \geq A$

Exemples sur 4 et 8 bits . Ces deux exemples montrent que lorsque l'on pique une deuzaine à gauche, on l'indique sur l'opérande dont on soustraire et il faut retrancher

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

cette deuzaine lors du calcul du rang juste à gauche, ce que l'on indique en ajoutant un '1' à cote du bit du nombre à soustraire du rang juste avant.

	0	1	$^{1}1$	$^{1}0$	$^{1}0$	0	1	1
-	0	0^{1}	1^{1}	1^{1}	1	0	0	0
	0	0	1	0	1	0	1	1

	$^{1}1$	$^{1}0$	0	0
- ^{1}	1^{1}	1	0	0
	1	1	0	0

3.3.3 Multiplication et division

Les opérations de multiplication et division de 2 entiers sont complexes et les circuits les réalisant ne sont pas implantés dans l'ALU : un circuit spécialisé ou un coprocesseur est ajouté pour les réaliser, sinon elles sont logiciellement émulées. Ces opérations sont réalisées par addition et décalage.

En binaire, pour les multiplications ou divisions par des puissances de 2, elles reviennent simplement à des décalages de n bits à gauche ou à droite respectivement. Soit :
Si $N = a_n a_{n-1}...a_1 a_0$ alors $2N = a_{n-1}...a_1 a_0 0$ et $N/2 = 0 a_n a_{n-1}...a_1$.
De manière générale, dans une base B, décaler à gauche de 1 (respectivement n) revient à multiplier par la base B (respectivement par B^n) et décaler à droite de 1 (respectivement n) à diviser par la base B (respectivement par B^n).

Exemple en base 10 :
$1024/10 = 102$: décalage à droite de 1 (division par 10^1)
$1024/100 = 10$: décalage à droite de 2 (division par 10^2)
$1024 * 10 = 10240$: décalage à gauche de 1

Exemple en base 2 :
$1100_b >> 1_d = 0110_b = 6_d = 12_d/2_d$
$0110_b >> 2_d = 0001_b = 1_d = 6_d/4_d$
$0011_b << 1_d = 0110_b = 6_d = 3_d * 2_d$
$0011_b << 2_d = 1100_b = 12_d = 3_d * 4_d$

3.4 Exercices

Objectif(s)

- ✯ Réalisation d'un circuit additionneur 4 bits à partir de l'additionneur 1 bit, en utilisant l'outil LOGISIM.
- ✯ Réalisation d'un soustracteur 4 bits
- ✯ Assemblage des deux circuits pour réaliser un additionneur-soustracteur.
- ✯ Réalisation d'un opérateur de comparaison d'entiers naturels

Exercice 8 – Additionneur 4 bits

Question 1
Sous LOGISIM, construisez un additionneur dont les entrées a, b et cin sont sur un bit,

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

et de sorties s et $cout$ sur un bit, en respectant la forme normale disjonctive des fonctions représentant le calcul des sorties s et $cout$. Vous utiliserez des portes logiques à 3 et 4 entrées, et complémenterez certaines entrées. Déterminez la taille du circuit en nombre de portes, ainsi que le chemin le plus long (il s'agit du chemin partant d'une entrée et aboutissant à une sortie, traversant le plus grand nombre de portes logiques). Testez le bon fonctionnement de votre circuit.

Question 2
En appliquant les règles d'équivalence de l'algèbre de Boole, proposez une expression réduite des fonctions calculant s et $cout$, et implantez les sous forme de circuit à partir de portes logiques à 2 entrées (éventuellement complémentées) ; vous chercherez à minimiser le nombre de portes de votre additionneur 1 bit, en réutilisant des sous-circuits communs aux deux fonctions. Assurez vous du bon fonctionnement de votre circuit en simulant toutes les configurations d'entrées.

Question 3
En utilisant l'additionneur 1 bits créé précédemment, construisez un additionneur 4 bits. L'additionneur devra disposer de l'interface suivante : deux entrées A (a3 a2 a1 a0) et B (b3 b2 b1 b0) sur 4 bits, une sortie S (s3 s2 s1 s0) sur 4 bits et une sortie Cout sur un bit.

Indication : Utilisez 4 additionneurs 1 bits en connectant la retenue sortante de l'additionneur de rang i à la retenue entrante de l'additionneur de rang i+1. Vous utiliserez également le composant "splitter" qui permet de convertir une nappe de n fils en n fils distincts, et inversement.

Question 4
Déterminer un jeu de tests couvrant et vérifier le bon fonctionnement de votre additionneur 4 bits. Vous testerez particulièremenst les cas de dépassement de capacité sur entier naturels.

Question 5
Quel est le chemin le plus long de ce circuit ? Quelle serait la longueur de ce chemin, en nombre de portes traversées, pour un additionneur de n bits ?

Exercice 9 – Réalisation d'un soustracteur 4 bits pour entiers naturels

Question 1
Réalisez les opérations arithmétiques suivantes, en base deux :
- 0100 0110 - 0000 0100
- 0100 0110 - 0010 1100
- 0100 0110 - 0110 1100

Donnez une interprétation, sur entier naturel, des opérandes et résultats de l'opération. Quel est l'intervalles des valeurs manipulables par ce soustracteur sur entier naturel ? Quel est l'indicateur de dépassement soustractif sur entier naturel ?

Question 2
Construisez la table de vérité des signaux d et $bout$, représentant respectivement la différence et la retenue soustractive propagée des entrées a, b et bin. En déduire une expression booléenne réduite de d et $bout$.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Question 3
Construisez le circuit d'un soustracteur 1 bit et testez-le.

Question 4
Construisez le circuit d'un soustracteur 4 bits et testez-le sur les trois opérations précédentes.

Exercice 10 – Additionneur-Soustracteur sur entiers naturels

Question 1
Construisez un circuit combinant un additionneur 4 bits, un soustracteur 4 bits et un multiplexeur permettant de sélectionner sur la sortie le résultat de A+B ou de A-B selon une commande Add/Sub sur 1 bit. Le circuit a l'interface suivante :

- signaux d'entrée sur 4 bits : A et B les opérandes
- signal d'entrée sur 1 bit : Add/Sub la commande
- signal de sortie sur 4 bits : S le résultat de A+B ou de A-B (tronqué sur 4 bits)
- signaux de sortie sur 1 bit : CY (resp. BR) les drapeaux de dépassement de capacité additive (resp. soustractive) sur entier naturels.

Question 2
Testez votre circuit sur quelques exemples d'additions / soustractions sur entiers naturels ; vérifiez le bon fonctionnement des drapeaux !

Exercice 11 – Comparateur d'entiers naturels sur 4 bits

On souhaite réaliser un circuit qui prend en entrée deux **entiers naturels** A et B codés sur 4 bits ($a_3a_2a_1a_0$ et $b_3b_2b_1b_0$) et qui indique `1` en sortie si B est strictement plus grand que A, `0` sinon.

Question 1
En utilisant des opérateurs de comparaison ($>, =$) bit à bit, donnez une expression booléenne qui vaut `1` si et seulement si B est plus grand que A.

Question 2
Quel opérateur logique permet de coder l'expression $b_i = a_i$?
Trouvez une expression logique qui code la comparaison $b_i > a_i$.

Question 3
Construisez un circuit qui réalise la comparaison de A et B.

3.5 Représentation des entiers relatifs et arithmétique entière

3.5.1 Représentation des entiers relatifs en complément à 2

Définition
Cette représentation est adoptée dans tous les processeurs pour représenter les entiers relatifs.

En complément à deux, tout entier est représenté par deux parties :

- un terme négatif constant ou nul

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

— une correction positive

Soit N_b un mot binaire de n bits, noté $N_b = a_{n-1}...a_1a_0$, alors son interprétation en complément à deux est $N_d = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i = -a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + ... + a_1 2 + a_0$

a_{n-1} est appelé bit de signe puisque sa valeur indique le signe de l'entier, $-a_{n-1}2^{n-1}$ est le terme négatif constant ou nul (si l'entier est positif) et $\sum_{i=0}^{n-2} a_i B^i$ correspond à la correction positive.
ATTENTION : l'interprétation entiere non signée de N est $\sum_{i=0}^{n-1} a_i B^i$

Exemples Sur 3 bits $a_2a_1a_0$ la valeur est $-a_2 * 2^2 + a_1 * 2 + a_0$
$N = 000_b = 0_d$ $N = 101_b = -2^2 + 1 = -3$ $N = 110_b = -2^2 + 2 = -2$ $N = 111_b = -2^2 + 2 + 1 = -1$ $N = 011_b = 2 + 1 = 3$ entier positif, le bit de signe a_2 est nul.

Intervalle de représentation Sur n bits, en complément à 2, l'intervalle de représentation est $[-2^{n-1},2^{n-1} - 1]$.

Preuve En effet si a_{n-1} vaut 0 alors c'est un naturel sur $n-1$ bits et est donc dans l'intervalle $[0,2^{n-1} - 1]$ (voir la section sur les entiers naturels). Sinon, a_{n-1} vaut 1, la correction est un entier dans l'intervalle $[0,2^{n-1} - 1]$, on a donc le plus petit entier négatif lorsque la correction est nulle, soit l'entier -2^{n-1} et le plus grand lorsqu'elle est maximale ie $2^{n-1}-1$ et donc vaut -1. Donc, l'ensemble des entiers relatifs représentés est en complément à 2 sur n bit est bien $[-2^{n-1},2^{n-1} - 1]$.

3.5.1.1 Détermination de l'opposé d'un nombre

Soit $N_b = a_{n-1}a_{n-2}...a_1a_0$ un entier représenté en complément à 2 :
l'opposé de N_b est $\overline{N_b} + 1$.

Preuve Le complément bit à bit de N_b est $\overline{N_b}$ soit $\overline{a_{n-1}a_{n-2}}..\overline{a_1a_0}$.
Leur somme vaut $N_b + \overline{N_b} = a_{n-1}a_{n-2}...a_1a_0 + \overline{a_{n-1}a_{n-2}}..\overline{a_1a_0}$. Comme $\forall i a_i + \overline{a_i} = 1$ on a :$N_b + \overline{N_b}$=11....$11_b = -1_d$.
En ajoutant 1 des deux cotés de l'égalité on obtient $N_b + \overline{N_b} + 1_d = -1_d + 1_d$, soit $-N_b = \overline{N_b} + 1_d$. Ainsi l'opposé de N_b est $\overline{N_b} + 1$.
Remarque : l'intervalle de définition des nombres représentables en complément à 2 sur n bits n'est pas symétrique : on ne peut pas déterminer l'opposé de -2^{n-1} sur n bits.

3.5.1.2 Extension du format d'un nombre

Un entier relatif représenté sur p bits peut être étendu sur $n > p$ bits en laissant les bits de rang 0 à $n-1$ inchangés et en plaçant dans les bits de rang p à $n-1$ la valeur du bit de rang $p-1$, c'est-à-dire la valeur du bit de signe.
Si $N = a_{p-1}a_{p-2}...a_1a_0$ alors sur n bits $N = a_{p-1}...(n - pfois)...a_{p-1}a_{p-1}a_{p-2}...a_1a_0$.

Exemple $N = 1000_b = 11000_b = 111000_b = 1111000_b$

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Preuve

$N_p = -a_{p-1}2^{p-1} + \sum_{i=0}^{p-2} a_i 2^i$
$N_{p+1} = -a_{p-1}2^p + \sum_{i=0}^{p-1} a_i 2^i = -a_{p-1}2^p + a_{p-1}2^{p-1} + \sum_{i=0}^{p-2} a_i 2^i$
$N_{p+1} = a_{p-1}(-2^p + 2^{p-1}) + \sum_{i=0}^{p-2} a_i 2^i$
$N_{p+1} = a_{p-1}(-2^{p-1}) + \sum_{i=0}^{p-2} a_i 2^i$
$N_{p+1} = N_p$

Exemples

Extension de 4 à 8 bits :
$N_1 = 1001_b = 11111001_b$
$N_2 = 0110_b = 00000110_b$
Extension de 16 à 32 bits en hexadécimal :
$N_3 = 90B2_h = FFFF90B2_h$
$N_4 = 1110_h = 00001110_h$

3.5.2 Addition et soustraction d'entiers relatifs

L'addition et la soustraction sur entiers relatifs suivent le même algorithme que celui défini pour les entiers naturels (on additionne les parties négatives et les corrections, ou on soustrait les parties négatives et les corrections).
L'additionneur n bits vu précédemment peut donc servir pour réaliser les additions d'entiers naturels ou relatifs.

Dépassement de capacité Le résultat de la somme de 2 entiers relatifs représentés sur n bits en complément à 2 n'est pas toujours représentable sur n bits. On parle dans ce cas de *dépassement de capacité* sur entiers relatifs.

Détermination d'un dépassement de capacité lors d'un addition Il y a un dépassement de capacité lors de l'addition de deux entiers représentés en compléments à 2 lorsque le `XOR` des retenues sortantes vaut 1.

Ce test est implanté dans l'ALU pour détecté les dépassements de capacité comme illustré sur la figure 3.5.
ATTENTION : la détection d'un dépassement de capacité lors de l'addition sur entiers relatifs est différentes de celle lors de l'addition d'entiers naturels (retenue sortante à 1).

On peut aussi détecter qu'il y a un dépassement de capacité lors de l'addition de deux entiers représentés en compléments à 2 lorsque le résultat de l'addition de deux entiers de même signe donne un résultat de signe opposé.

Explication de la détection d'un dépassement de capacité En y regardant de plus près, si le bit de signe des deux opérandes A et B est :

— $a_{n-1} = b_{n-1} = 0$ alors cela revient à l'addition de deux entiers naturels sur $n-1$ bits, comme $a_{n-1} = b_{n-1} = 0$ on aura forcément $c_{out_{n-1}} = 0$. Le résultat doit être positif et donc représentable sur $n-1$ bits. Sur $n-1$ bits il y a un dépassement de capacité sur entier naturel si la retenue sortant du dernier rang vaut 1, soit $c_{out_{n-2}} = 1$. On sort donc des entiers naturels représentable sur $n-1$ bits si $c_{out_{n-2}} = 1$. Cette retenue étant additionnée avec les bits de rang $n-1$ c'est-à-dire les bits de signe, le bit de signe du résultats vaut donc 1. Ainsi, lorsque l'on additionne deux nombres

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

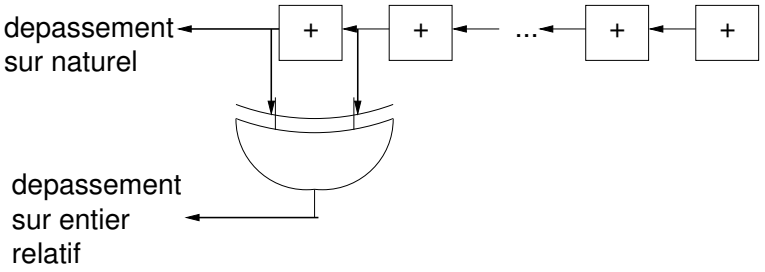


FIGURE 3.5 – Détection de dépassement de capacité lors d'une addition sur entiers naturels ou sur entiers relatifs

positifs, il y a un dépassement de capacité sur entiers relatifs si le résultats est un nombre négatif en complément à deux.

— $a_{n-1} = b_{n-1}$ = 1 alors il s'agit de l'addition de deux entiers négatifs.
Comme $a_{n-1} = b_{n-1}$ = 1 on a forcément $c_{out_{n-1}} = 1$.
L'addition de deux valeurs négatives est négative, donc on doit avoir $s_{n-1} = 1$ c'est-à-dire $c_{out_{n-2}} = 1$ pour rester dans l'intervalle de représentation. Si $c_{out_{n-2}} = 0$, alors $s_{n-1} = 0$ c'est-à-dire le résultat est un nombre positif en complément à 2 sur n bits, il y a un dépassement de capacité.
Lorsque l'on additionne deux valeurs négatives, il y a un dépassement de capacité si le résultat est une valeur positive, et on a forcément $c_{out_{n-2}} = 0$ et $c_{out_{n-1}} = 1$
$A + B = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i - b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$
$A + B = -2^{n-1} - 2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i + \sum_{i=0}^{n-2} b_i 2^i$
$A + B = -2^n + \sum_{i=0}^{n-2} a_i 2^i + \sum_{i=0}^{n-2} b_i 2^i$
Pour rester dans l'intervalle $[-2^{n-1},0[$ il faut que la somme des deux corrections positives soit supérieure ou égale à 2^{n-1} donc $c_{out_{n-2}} = 1$.

— $a_{n-1} \neq b_{n-1}$ alors il s'agit d'une soustraction puisqu'un des deux nombres est négatif. On a alors
$A + B = -2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i + \sum_{i=0}^{n-2} b_i 2^i$
Or $\sum_{i=0}^{n-2} a_i 2^i + \sum_{i=0}^{n-2} b_i 2^i$ est compris entre 0 et 2^{n-1} inclus (valeur sur n bits avec la retenue sortante). On a donc :
$-2^{n-1} \leq A + B \leq -2^{n-1} + 2^n - 1$ c'est-à-dire
$-2^{n-1} \leq A + B \leq 2^{n-1} - 1$.
Ainsi, quelles que soient les valeurs de A et B, l'addition de deux entiers relatifs de signe opposé est toujours représentable sur n bits.
Dans ce cas de figure, on a $a_{n-1} + b_{n-1} = 1$.
Si $c_{out_{n-2}} = 1$ alors $s_{n-1} = 0$ et $c_{out_{n-1}} = c_{out_{n-2}} = 1$. Le résultat est positif.
Si $c_{out_{n-2}} = 0$ alors $s_{n-1} = 1$ et $c_{out_{n-1}} = c_{out_{n-2}} = 0$. Le résultat est négatif.

En reprenant les différents cas, on s'aperçoit que le résultat d'une addition sur entiers relatifs est correct, c'est-à-dire si le résultat sur les bits $n-1$ à 0 représente bien en complément à 2 le résultat de l'addition ssi $c_{out_{n-1}} = c_{out_{n-2}}$ et donc qu'il y a dépassement de capacité ssi $c_{out_{n-1}} \neq c_{out_{n-2}}$.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Instructions d'addition et soustraction MIPS En MIPS, les instructions d'addition et de soustraction sont :

- `add Rd, Rs, Rt` ou `addu Rd, Rs, Rt` avec `Rx` des registres et réalisant `Rd` ← `Rs + Rt`.
- `addi Rt, Rs, imm` ou `addiu Rt, Rs, imm` avec `Rx` des registres, `imm` un immédiat sur 16 bits et réalisant `Rt` ← `Rs + extension-16-vers-32(imm)` avec l'immédiat étendu sur 32 bits (de manière signée)
- `sub Rd, Rs, Rt` ou `subu Rd, Rs, Rt` avec `Rx` des registres et réalisant `Rd` ← `Rs - Rt`.

Les instructions dont le nom se termine par `u` sont sans détection de capacité alors que les autres oui. Dans tous les cas, l'immédiat et les valeurs contenues dans les registres sont interprétées en complément à 2 et vues comme des valeurs signées, qu'il y ait ou non détection de dépassement de capacité.

Exemples d'addition d'entiers relatifs

$$\begin{array}{cccccccccc} {\scriptstyle 1} & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ + & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ \hline 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \end{array} \qquad \begin{array}{cccccccccc} & {\scriptstyle 0^1} & {\scriptstyle 1^1} & 1 & {\scriptstyle 0^1} & 1 & 1 & 1 & 0 \\ + & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ \hline & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{array}$$

Dans l'exemple de gauche, on additionne deux nombres négatifs et le résultat est un nombre positif, cela signifie qu'il y a un dépassement de capacité. Cela se voit aussi avec la formule $c_{out_{n-1}}(=1)\ xor\ c_{out_{n-2}}(=0)=1$. Comme $c_{out_{n-1}}=1$ si ces nombres étaient interprétés commes des naturels, il y aurait aussi un dépassement de capacité.

Dans l'exemple de droite, on additionne deux nombres positifs et le résultat est négatif. Il y a un dépassement de capacité. Cela se voit aussi avec la formule $c_{out_{n-1}}(=0)\ xor\ c_{out_{n-2}}(=1)=1$. Comme $c_{out_{n-1}}=0$ si ces nombres étaient interprétés commes des naturels, il n'y aurait pas de dépassement de capacité.

$$\begin{array}{cccccccccc} {\scriptstyle 1} & {\scriptstyle 1^1} & {\scriptstyle 0^1} & {\scriptstyle 0^1} & {\scriptstyle 1^1} & {\scriptstyle 0^1} & {\scriptstyle 1^1} & {\scriptstyle 1^1} & 1 \\ + & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \hline & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{array}$$

Dans cet exemple, on additionne un nombre positif et un nombre négatif, il n'y a donc pas de dépassement de capacité sur entiers relatifs, et l'on a $c_{out_{n-1}}=c_{out_{n-2}}=1$. Sur entier naturel, comme $c_{out_{n-1}}=1$ on aurait un dépassement de capacité.

$$\begin{array}{cccccccccc} & 1 & {\scriptstyle 1}0 & {\scriptstyle 1}0 & {\scriptstyle 1}1 & {\scriptstyle 1}0 & 1 & 1 & 1 \\ - & {\scriptstyle 0^1} & {\scriptstyle 1^1} & {\scriptstyle 1^1} & {\scriptstyle 1^1} & 1 & 1 & 1 & 1 \\ \hline & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \end{array} = \begin{array}{cccccccccc} {\scriptstyle 1} & 1 & 0 & 0 & 1 & {\scriptstyle 0^1} & {\scriptstyle 1^1} & {\scriptstyle 1^1} & 1 \\ + & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \end{array}$$

Dans cet exemple, il s'agit d'une soustraction, on retranche un nombre positif à un nombre négatif. A gauche, le calcul est fait directement (soustraction bit à bit avec propagation des retenues, des deuzaines piquées). A droite, on calcule l'opposé du nombre à retrancher et on effectue une addition. On s'apercoit que l'on additionne bien deux nombres negatifs. Le résultat est un nombre positif ce qui signifie qu'il y a dépassement de capacité sur entiers relatifs. On a bien $c_{out_{n-1}}=1\neq c_{out_{n-2}}=0$.

Comme $c_{out_{n-1}}=1$ sur entier naturel cette addition provoquerait un dépassement de capacité. Toutefois cette addition ne correspond pas à la soustraction demandée sur entier naturel puisque l'opposé d'un entier naturel n'existe pas dans N! Du coté de la

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

soustraction, en considérant des entiers naturel, comme le nombre à retrancher étant plus petit que celui dont on le retranche, la soustraction est bien définie et il n'y a pas de dépassement de capacité sur entier naturel.

3.6 Exercices

Objectif(s)

- ✯ Représentation et additions d'entiers relatifs ;
- ✯ Réalisation d'opérateurs d'addition et de soustraction ;
- ✯ Retour sur les entiers naturels.
- ✯ Réalisation d'une unité arithmétique et logique permettant l'addition et la soustraction de deux entiers de 4 bits, avec indicateurs de dépassement sur entiers naturels et relatifs
- ✯ Réalisation d'opérateurs de décalage

Exercice 12 – Représentation des entiers relatifs

La représentation binaire des entiers relatifs est utilisée sur des écritures de nombres de **longueur donnée** (nombres écrits couramment sur 8, 16, 32 ou 64 bits). Dans une telle écriture on utilise le bit de poids fort (bit le plus à gauche) du nombre pour contenir la représentation de son signe (positif ou négatif, le zéro étant considéré comme positif).

Question 1

Les première et troisième colonnes du tableau ci-dessous contiennent les entiers relatifs **codés en complément à 2 sur 3 bits**. Donnez pour chaque entier relatif sa valeur en décimal.

Complément à 2	Décimal	Complément à 2	Décimal
000_b		100_b	
001_b		101_b	
010_b		110_b	
011_b		111_b	

Sommez une valeur avec son inverse et une valeur avec la valeur zéro. Qu'en déduisez-vous ?
Quel est l'intervalle des valeurs représentées en complément à 2 par un mot de n bits ?
Quand peut on avoir un dépassement de capacité lors d'une addition de deux nombres relatifs et quand est on sur qu'il n'y aura pas de dépassement de capacité ? Justifiez votre réponse.

Question 2

Calculez l'opposé des nombres relatifs codés sur 16 bits $ABCD_h, FFFF_h, 5A72_h$ en base

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

16 et en base 2. Trouvez une autre règle de conversion. Pour chacun des nombres relatifs suivants codés sur 16 bits, indiquez si celui-ci est positif ou négatif, puis calculez l'opposé en base 2 et en base 16.

base 16	signe	base 2	opposé en base 2	opposé en base 16
$0B24_h$				
$ABCD_h$				
$FFFF_h$				
$5A72_h$				
72_h				

Exercice 13 – Opérations arithmétiques

Question 1
Soit l'addition binaire suivante : $01101001 + 10000000 = 11101001$. Quelle opération réalise-t-on dans le cas où les opérandes et résultat sont des entiers naturels ? Le résultat est-il codable sur 8 bits ? Quelles sont les valeurs des bits $Carry$ (= CY_n) et Ov (= CY_n xor $C_Yn - 1$) ? Même question dans le cas où les opérandes et le résultat sont des entiers relatifs en complément à 2.

Question 2
Soit l'opération $0xE2 + 0x9C$. Calculer le résultat. Quelle opération effectue-t-on si les opérandes sont des entiers naturels ? Le résultat est-il valide ? Mêmes questions si les opérandes sont des relatifs. Quelles sont les valeurs des bits $Carry$ et Ov ?

Question 3
Soit l'opération sur entiers naturels $136 + 250$. Comment est-elle traduite au niveau de l'additionneur (en binaire ou en hexa) ? Le résultat est-il valide, et pourquoi ? Si maintenant les mêmes valeurs en entrée de l'additionneur sont interprétées comme des entiers relatifs, que vaut le résultat ? Est-il valide ?

Question 4
Trouver des valeurs d'opérandes en entrée de l'additionneur qui produisent un résultat valide s'ils sont interprétés comme entiers naturels et invalide s'ils sont interprétés comme entiers relatifs.

Exercice 14 – Opérations arithmétiques

Question 1
Dans les opérations suivantes, les opérandes sont des entiers relatifs représentés en complément à 2 sur 8 bits. Effectuez les opérations suivantes en binaire sur des mots de 8 bits.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Pour chaque opération mettez en évidence la retenue sortante (de rang n) ainsi que la retenue de rang n-1 et indiquez s'il existe un dépassement de capacité sur entiers relatifs. Donnez l'equation booléenne permettant de dire s'il y a ou non dépassement de capacité sur entiers relatifs.

$72_h - 45_h$
$FE_h - 02_h$
$64_h + 7A_h$
$A3_h - 72_h$
$10011011_b - 11100000_b$

Exercice 15 – Entiers naturels vs entiers relatifs

Question 1
Reprenez les opérations de la question précédente en considérant que les nombres manipulés sont des entiers naturels. Indiquez quelles sont les opérations qui génèrent un dépassement de capacité sur entiers naturels.

Exercice 16 – Extension d'entiers naturels et relatifs

Question 1
Soit le nombre hexadécimal FF_h. Donnez sa valeur en décimal dans le cas d'un entier naturel et d'un entier relatif. Etendez ce nombre sur 32 bits. Obtenez-vous la même chose si l'on considère que c'est un entier naturel ou un entier relatif ? En déduire les règles d'extension d'un entier naturel et d'un entier relatif.

Exercice 17 – Additionneur-soustracteur 4 bits pour entiers naturels et relatifs

Question 1
Pour cet exercice de TP, vous devez partir de l'additionneur 4 bits construit précédemment pour construire une ALU (unité arithmétique et logique) pouvant effectuer soit une addition, soit une soustraction d'opérandes sur 4 bits *sans utiliser de circuit spécifique de soustraction*. Vous considérerez donc que la soustraction est réalisée par addition du nombre opposé : A - B est réalisé par A + (not B + 1).

L'ALU à réaliser aura deux entrées X (sur 4 bits) et Y (sur 4 bits) qui seront les opérandes, une entrée Aluop (sur 1 bit) qui permet de selectionner le type d'opération à effectuer (0 $\rightarrow$ addition, 1 $\rightarrow$ soustraction), une sortie S sur 4 bits et une sortie Cout correspondant à la retenue sortante.

Conseils :
— Pour effectuer plus efficacement vos simulations, votre instance finale devra avoir l'interface de la figure 3.6.
— Vous pourrez utiliser des multiplexeurs (ceux que vous avez construits en 1ere séance ou disponibles dans la bibliothèque), ou mieux, des portes XOR.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

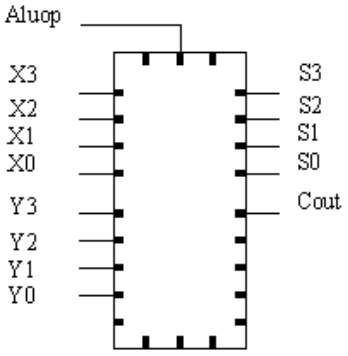


FIGURE 3.6 – Interface de l'ALU

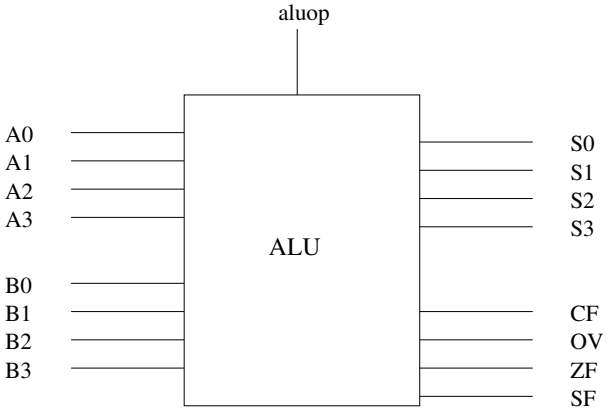


FIGURE 3.7 – Interface de l'ALU à réaliser

— Vous pourrez aussi avoir besoin de générateurs de constantes logiques 0 et 1 disponibles dans LOGISIM.

Question 2
Vérifiez que votre ALU fonctionne correctement en effectuant par simulation un ensemble d'opérations couvrant les différents cas possibles. Vous pourrez reprendre des opérations précédemment réalisées.

Question 3
Déterminez l'intervalle des valeurs des opérandes X et Y pour lesquelles S représente effectivement X+Y et X-Y.

Question 4
Modifiez votre ALU pour que celle-ci génère en sortie les signaux suivants (nommez les avec une étiquette) :

— un signal OV qui vaut 1 lorsque l'opération génère un dépassement de capacité sur entiers relatifs
— un signal ZF qui vaut 1 lorsque le résultat de l'opération est nul
— un signal SF qui vaut 1 lorque le résultat est négatif
— un signal CF qui correspond à la retenue sortante du dernier rang

L'interface de votre ALU est alors comme représenté sur la figure 3.7.

Question 5
Vérifiez le bon fonctionnement de votre nouvelle ALU par simulation, en proposant trois

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

opérations bien choisies. Comparez les résultats obtenus avec ceux obtenus par un calcul à la main.

Exercice 18 – Décalage à droite

On souhaite réaliser un opérateur qui effectue un décalage à droite signé de 0, 1, 2 ou 3 bits d'un entier codé sur 4 bits en complément à 2. Le décalage à droite est dit signé lorsqu'il y a propagation du bit de signe sur les bits de poids forts libérés par le décalage.

Cet opérateur aura en entrée un entier codé sur 4 bits $a_3a_2a_1a_0$ et un entier codé sur 2 bits d_1d_0 indiquant le nombre de bits à décaler. On notera $s_3s_2s_1s_0$ la sortie du décaleur.

Question 1

On note $>>$ l'opération de décalage à droite signé (comme en C). Quel est le résultat des opérations suivantes en binaire et en décimal ? Quel est l'effet d'un décalage à droite de i bits d'un entier codé sur n bits ? Vous distinguerez le cas où l'entier est positif du cas où il est négatif.

- $0011 >> 0 =$
- $0011 >> 1 =$
- $1011 >> 1 =$
- $0011 >> 2 =$
- $1011 >> 2 =$
- $0011 >> 3 =$
- $1011 >> 3 =$

Question 2

Donnez l'expression booléenne de chacun des bits de sortie (les s_i) en fonction des entrées. Il est fortement conseillé de vous aider d'une table de vérité.

Question 3

Réalisez le circuit sous LOGISIM et vérifiez son fonctionnement par simulation en testant, pour plusieurs valeurs de l'entier sur 4 bits, les différentes valeurs possibles de d_1d_0. Vérifiez par simulation votre réponse à la question 1 de cet exercice.

Exercice 19 – Décalage à gauche

On souhaite réaliser un opérateur qui effectue un décalage à gauche de 0, 1, 2 ou 3 bits d'un entier naturel codé sur 4 bits. Cet opérateur aura en entrée un entier naturel codé sur 4 bits $a_3a_2a_1a_0$ et un entier codé sur 2 bits d_1d_0 indiquant le nombre de bits à décaler. On notera $s_3s_2s_1s_0$ la sortie du décaleur.

Question 1

On note $<<$ l'opération de décalage à gauche (comme en C). Quel est le résultat des opérations suivantes en binaire et en décimal ? Quel est l'effet d'un décalage à gauche de i bits d'un entier naturel codé sur n bits ?

- $0011 << 0 =$
- $0011 << 1 =$
- $0011 << 2 =$
- $0011 << 3 =$

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Codage ASCII

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

Extension iso-latin

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8																
9																
A		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
B	°	±	2	3	´	µ	¶	·	¸	1	º	»	¼	½	¾	¿
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

FIGURE 3.8 – Table de codage ASCII et l'extension iso-latin

Question 2
Donnez l'expression booléenne de chacun des bits de sortie (les s_i) en fonction des entrées. Vous pourrez vous aider d'une table de vérité si besoin.

Question 3
Réalisez le circuit sous LOGISIM et vérifiez son fonctionnement par simulation en testant, pour plusieurs valeurs de l'entier sur 4 bits, les différentes valeurs possibles de $d_1 d_0$.
Vérifiez par simulation votre réponse à la question 1 de cet exercice.

3.7 Représentation des caractères alphanumériques

Les caractères alphanumériques permettent de représenter des mots en langage naturel, écrit par l'utilisateur d'un ordinateur. Cela peuvent être les mots d'un texte tapé dans un logiciel de traitement de texte, les mots décrivant un programme (C ou autre), les mots d'une commande tapée dans un terminal ou une console, les mots d'un e-mail,...

Les caractères sur le clavier sont désignés par leur position (numéro de ligne, numéro de colonne). Une table de codage associe à chaque position (éventuellement en comninaison avec d'autres touches) un caractère. Ce caractère lorsqu'il est rangé en mémoire ou lorsqu'il est manipulé par un programme est codé suivant le code international ASCII. Chaque caractère est codé sur 1 octet, suivant une table qui est donnée en haut sur la figure 3.8.
Le codage est sur 7 bits et ne prend pas en compte les lettres accentuées, les cédilles,... Il

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

n'est pas adapté pour les alphabets autres que romains. Des extensions de codage incorporant le 8ème bit de poids fort permettent d'avoir un sous ensemble commun (caractères codés de 00_h à $7F_h$) puis des caractères spécifiques à une langue (codé de 80_h à FF_h). L'extension pour les caractères français est iso-latin. La table correspondante est illustrée dans le bas de la figure 3.8.

Des extensions à 16 bits ont été proposées, notamment l'UNICODE très répandu et admis aujourd'hui. Il permet de coder toutes les langues et des tas de symboles. Il y a plusieurs variantes : UTF32 sur 32 bits (de longueur fixe), UTF16 et UTF8 à longueur variable. L'UTF8, le plus compact, contient le code ascii (et les autres caractères sont codés sur 16 ou 32 bits). L'iso-latin c'est du passé (en tout cas pour tous les textes que l'on tape aujourd'hui, comme le fichier source de ce cours qui est en utf8). Sur le web c'est l'UTF8 qui prédomine, en Java le choix a été fait pour l'UTF16, en Scheme (version 6) c'est de l'UTF32. Bien sûr il y a des bibliothèques de conversions d'un codage dans un autre.

Codage de mots

Pour coder un mot en ASCII il faut coder chacune des lettres et symboles le composant. Par exemple :
"Lundi" se code avec les valeurs 0x4C (pour le caractère L) 0x75 (u) 0x6E (n) 0x64 (d) 0x69 (i) et 0x00 qui est le caractère de fin de chaînes.
"123" se code avec les valeurs 0x31 (pour le caractère 1) 0x32 (pour le caractère 2) 0x33 (pour le caractère 3) 0x00 (pour le caractère de fin de chaînes).
Attention $123_d = 64 + 32 + 16 + 8 + 2 + 1 = 01111011_b = 0x7B_h = 0x0000007B_h$ ce qui est très différent du codage de la chaîne de caractères "123".

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Chapitre 4

Opérande registre et mémoire

Contents

Dans ce chapitre, on s'intéresse au stockage des informations dans l'ordinateur ainsi qu'à la localisation des unités de stockage.

4.1 Registres

Les registres correspondent à de la mémoire très temporaire. A tout instant, les données et les instructions du programme en cours d'exécution sont stockées en mémoire et dans le processeur.

Dans le processeur, toute donnée (temporaire) est stockée dans des registres.

Un registre n bits permet de stocker un mot binaire de n bits. Il est composé de la mise en parallèle de n registres 1 bit. Le registre 1 bit est la cellule mémorisante élémentaire.

La figure 4.1 montre l'interface d'un registre 1 bit. Sur cette figure, CK designe le signal d'horloge globale cadençant le processeur. WE signigie Write Enable. D'un point de vue macroscopique, le registre 1 bit mémorisant sur front montant a la fonctionnalité suivante : lorsque CK passe de 0 à 1 (front montant) si WE = 1 alors la valeur de Din est

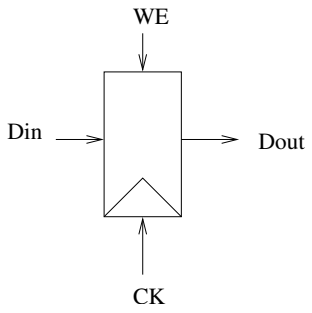


FIGURE 4.1 – Interface d'un registre 1 bit

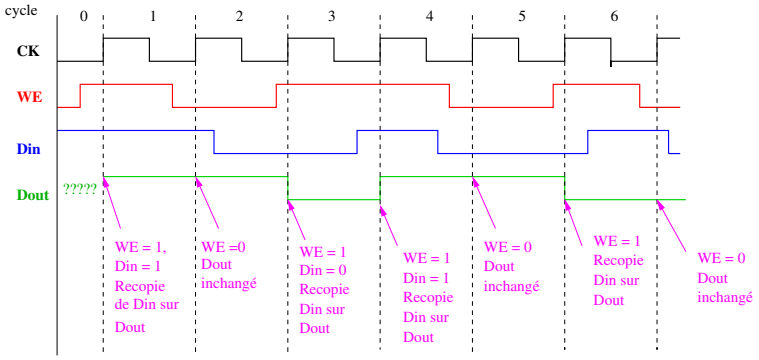


FIGURE 4.2 – Chronogramme pour un registre 1 bit

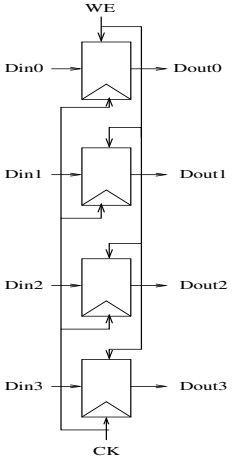


FIGURE 4.3 – Registre 4 bits

recopiée sur Dout. Sinon Dout est inchangé (garde la valeur précédente). A tout autre instant (front descendant donc) ou si WE = 0 alors Dout est inchangé.

La figure 4.2 montre un chronogramme avec les signaux d'entrées et la valeur de sortie Dout d'un registre 1 bit sur plusieurs cycles en fonction de ses entrées.

Pour réaliser un registre de n bits, il suffit de mettre en parallèle n registres 1 bit et de connecter les signaux WE et CK de tous ces registres au même signal. La figure 4.3 représente un registre 4 bits à partir de 4 registres 1 bit.

Comment stocker le mot 1100 dans le registre 4 bits ?

1. Placer Din0 à 0, Din1 à 0, Din2 à 1 et Din3 à 1
2. Positionner WE à 1
3. Lors du prochain front montant de CK (cycle suivant) on aura Dout=Din c'est à dire Dout0 = 0, Dout1 = 0, Dout2 = 1, Dout3 = 1
4. Si on change les valeurs en entrée et WE = 0 alors Dout vaudra toujours 1100.

Si 1) et 2) ont lieu au cycle i, alors 3 a lieu au cycle i + 1

La figure 4.4 montre le transfert d'informations dans l'ALU.

L'architecture du processeur définit le nombre, la taille et le nom des registres du processeur. Certains peuvent être manipulés explicitement par le programmeur.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

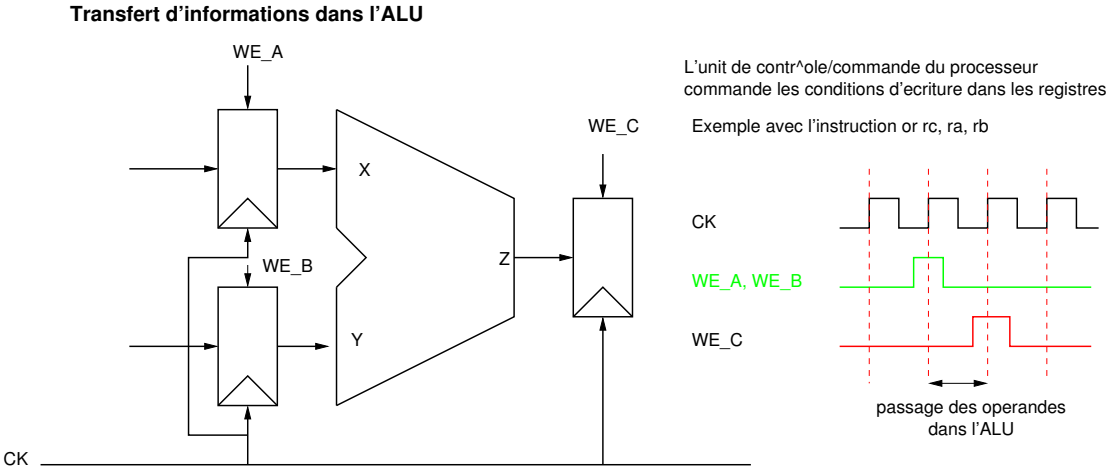


FIGURE 4.4 – Transfert d'informations dans l'ALU

Manipulation des registres : on peut affecter une valeur à un registre par le biais d'instructions du programme.

- Placer explicitement une valeur dans R : $R \leftarrow 2$
- Placer le résultat d'un calcul dans R : $R \leftarrow A + B$
- Placer une donnée qui était stockée en mémoire : $R \leftarrow Mem[@]$

Registres du MIPS

- Les registres du MIPS font 32 bits
- Les registres R0,..., R31 sont des registres de travail et R0 vaut toujours 0
- PC (Programme Counter) contient l'adresse de l'instruction en cours d'exécution (ou la suivante)
- Hi/Lo (High/Low) sont les registres pour le résultat d'opérations de multiplication ou de division

4.2 Architecture de la mémoire

La mémoire stocke toutes les informations utiles à l'exécution d'un programme, c'est-à-dire les instructions et les données.

La mémoire, comme illustrée sur la figure 4.5, est un tableau de n lignes de p bits, une suite de n mots de p bits. La position du mot dans la suite/le numéro de ligne définit l'adresse d'un mot. Le premier mot a l'adresse 0, le 2ème l'adresse 1, le 3ème l'adresse 2, ... et le dernier a l'adresse (n-1).

En général les mots sont des octets, soit $p = 8$, on dit que l'unité adressable est l'octet. Le mots plus longs (multiples d'octets/du mot de base) sont rangés sur des octets contigües en mémoire.

Rangement de mots de plus de n bits Une donnée de plus de n octets est rangée à des adresses contigües. Soit $M = o_3o_2o_1o_0$ un mot de 4 octets, o_0 étant l'octet de poids faible. On suppose que M est rangé à l'adresse A, il occupe donc les adresse $A, A+1, A+2, A+3$. Il y a deux ordres possibles de rangement, comme indiqué ci-dessous :

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

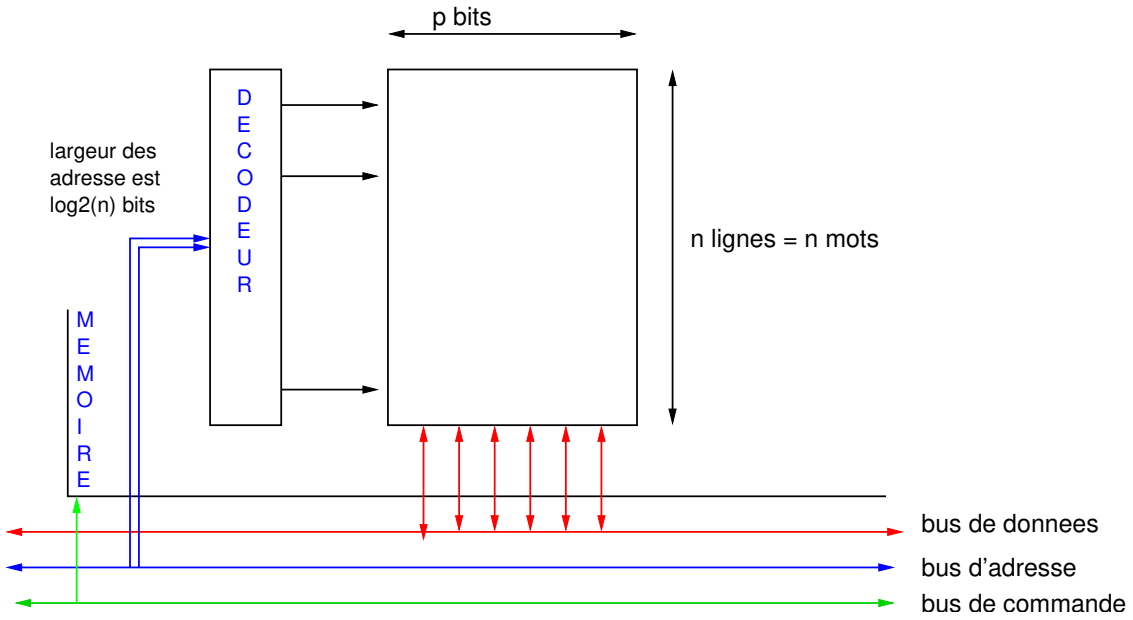


FIGURE 4.5 – Un mémoire et ses connexions aux différents bus nécessaires pour tout transfert avec le processeur

Adresse	A	$A+1$	$A+2$	$A+3$
Rangement petit boutien	o_0	o_1	o_2	o_3
Rangement gros boutien	o_3	o_2	o_1	o_0

Dans le rangement petit boutien, l'octet de poids faible est rangé à l'adresse la plus petite, alors que dans le rangement gros boutien c'est l'octet de poids fort est rangée à l'adresse la plus petite.
Si $M = 0xABCDEF12$ alors :

Adresse	A	$A+1$	$A+2$	$A+3$
Rangement petit boutien	0x12	0xEF	0xCD	0xAB
Rangement gros boutien	0xAB	0xCD	0xEF	0x12

L'ordre de rangement des mots de plusieurs octets (*endianness* en anglais) est fixe pour une architecture. En MIPS, le rangement est en petit boutien (*little endian*.

Capacité mémoire La capacité mémoire correspond au nombre de mots qu'elle peut stocker, c'est un nombre d'octets. On a :
- 1 kilo-octets ou 1 ko = 2^{10} octets
- 1 mega-octets ou 1 Mo = 2^{20} octets
- 1 giga-octets ou 1 Go = 2^{30} octets
- 1 tera-octets ou 1 To = 2^{40} octets

Fonctionnement de la mémoire La figure 4.5 représente la mémoire avec son décodeur d'adresse et les différents bus nécessaires pour les transferts de/vers le processeur. Lors d'un transfert,
- le décodeur d'adresse sélectionne la ligne correspondant à l'entrée demandée/indiquée sur le bus d'adresse,

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

— la commande indique si l'opération à effectuer est une lecture ou une écriture,
— enfin, les données lues sont mises sur le bus de données, celles à écrire dont celles présentes sur le bus de données.

Transfert entre la mémoire et le processeur Le processeur initie les transferts de données entre le processeur et la mémoire en définissant :

— L'adresse du mot à transférer
— La taille du mot à transférer
— Le sens du transfert : si processeur $\rightarrow$ mémoire alors c'est une écriture ou un store, si mémoire $\rightarrow$ processeur alors c'est une lecture ou un load (chargement).
— La donnée à écrire si écriture

Protocole d'écriture de v à l'adresse A

— Le processeur place A sur le bus d'adresse, v sur le bus de données, indique qu'il s'agit d'une écriture sur le bus de commande ainsi que la taille de la donnée v.
— Le processeur passe ensuite à l'instruction suivante s'il s'agit d'une écriture non bloquante ou attend un acquittement de la mémoire si les écritures sont bloquantes
— La mémoire lit l'adresse A sur le bus d'adresse et la donnée v sur le bus de donnée puis place v à l'adresse A.

Remarque : une fois l'écriture effectuée on a $Mem[A] = v$ (la valeur du mot d'adresse A est v) et est inchangée jusqu'à la prochaine écriture à cette adresse (ou à une des adresses occupée par le mot v si de taille supérieure à 1 octet).

Protocole de lecture d'un mot à l'adresse A

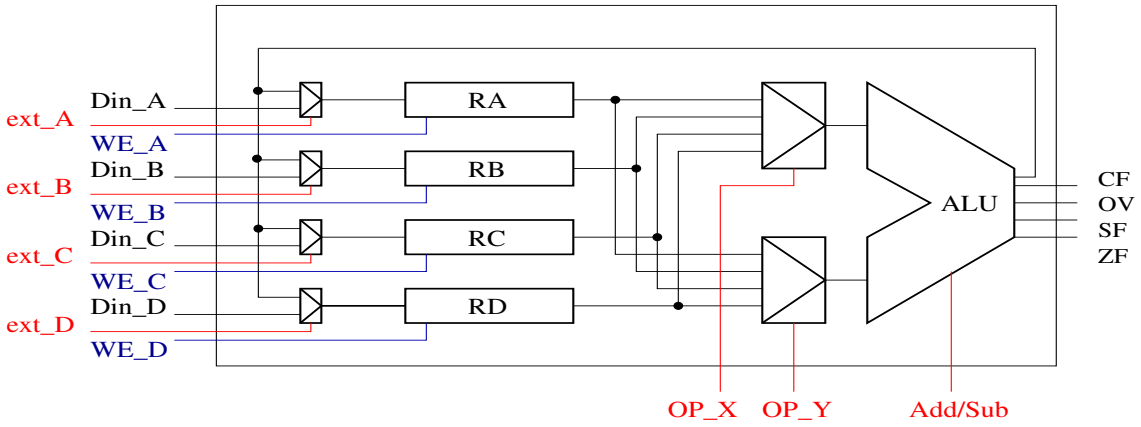
— Le processeur place A sur le bus d'adresse, indique qu'il s'agit d'une lecture ainsi que la taille de la donnée à lire sur le bus de commande.
— La mémoire lit l'adresse A sur le bus d'adresse, place le contenu de $Mem[A]$ soit la valeur v sur le bus de données et acquitte le transfert.
— Le processeur récupère v sur le bus de données et le place dans le regsitre destination

Remarque : la lecture à l'adresse A retourne la valeur contenue dans le mot d'adresse A à l'instant de la lecture. C'est la valeur de la derniere écriture à cette adresse. La lecture n'est pas destructrice : elle laisse le mot inchangé en mémoire.

4.3 Notion de chemin de données

Dans un système logique, un **chemin de données** est un ensemble d'unités fonctionnelles électroniques, telles que des unités arithmétiques et logiques, des multiplicateurs, des registres, qui permettent d'effectuer des opérations de traitement de données. Il représente de manière abstraite toutes les voies de circulation des données disponibles pour réaliser des traitements spécifiés par des commandes de ce systèmes (instructions).
Exemple de chemin de données :

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International



Sur ce chemin de données, comment réaliser RD ← 12 - 45 + 2 ?
Comment peut-on calculer de la valeur absolue d'un entier relatif ?
Quel sera le chronogramme des signaux de contrôle dans ces deux exemples ? Les valeurs contenues dans les registres au cours du temps ?

4.4 Exercices

Objectif(s)

- ✯ Compréhension du fonctionnement d'un registre via sa manipulation
- ✯ Comprésion du fonctionnement d'une mémoire via sa manipulation
- ✯ Manipulation d'un chemin de données simplifié d'un processeur

Exercice 20 – Manipulation d'un registre
Ce petit exercice a pour but de vous familiariser avec la manipulation des registres sous LOGISIM.

Dans LOGISIM, sélectionnez dans la rubrique *Memory*, le composant *Register*.

Modifiez ses paramètres pour que la donnée mémorisée soit sur 1 bit. Connectez sur ses ports d'entrée les signaux *Din* (Data In), *WE* (Write Enable, actif à 1), *CK* (Clock, signal d'horloge) et la constante *O* sur l'entrée Clear.

Connectez sur son port de sortie le signal *Dout*. Vous devez obtenir un schéma conforme à la figure 4.6.

Le signal d'horloge peut être manipulé manuellement ou par l'intermédiaire du menu *Simulate*, en activant les onglets "Simulation Enabled" et "Tick Once". Si l'onglet "Tick Enabled" est activé, l'horloge évolue périodiquement, la fréquence de battement pouvant être réglée dans l'onglet "Clock Frequency".

L'évolution du contenu du registre au cours du temps peut être visualisé dans un tableau : dans le menu *Simulate*, l'onglet "logging" permet d'accéder à une fenêtre permettant de sélectionner les signaux à observer (onglet "Selection"), de visualiser l'évolution des signaux au fil des cycles d'horloge (onglet "Table"), et de sauvegarder cette visualisation dans un fichier (onglet "File").

Question 1

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

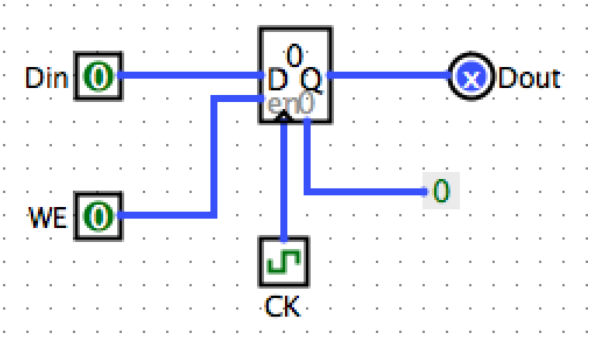


FIGURE 4.6 – Connexion du registre 1 bit.

— Quel enchainement d'actions doit on réaliser pour écrire la valeur 1 dans le registre et maintenir cette valeur mémorisée pendant au moins trois cycles ?

— Paramétrez le service "logging" pour visualiser l'évolution des signaux d'entrée et sortie du registre, puis, en utilisant la commande "Tick Once", réalisez cet enchainement d'actions.
— Affichez la fenêtre montrant l'évolution des signaux et assurez vous que l'enchainement des actions que vous avez réalisé produit bien la mémorisation de la valeur 1 pendant au moins 3 cycles dans le registre.

Question 2
Créez un registre 4 bits en modifiant la largeur des nappes de fils Din et Dout. Créez un scenario permettant de mémoriser le mot $(0100)_b$ pendant 2 cycles, puis le mot $(1111)_b$ pendant 3 cycles. Construisez les tables d'évolutions des signaux CK, Din, WE, Dout pour ce scenario.

Exercice 21 – Manipulation d'une mémoire

Ouvrez un nouveau circuit dans logisim et sélectionnez le composant *RAM* dans la rubrique *Memory*.

Question 1
Paramétrez le composant pour disposer d'une mémoire de 16 mots de 8 bits avec deux ports de données distincts (*Separate load and store ports*). Quelles doivent être les largeurs des nappes de fils ADDRESS, DATA et COMMAND ? Placez des pins sur les ports d'entrées et des sondes sur le port de sortie.

Question 2

— Que représente les mots apparaissant à l'intérieur de la mémoire ? dans quelle base sont-ils écrits ?
— Comment connaitre le contenu du mot d'adresse $(0000)_b$, du mot d'adresse $(0001)_b$?
— Sur une feuille notez les actions à réaliser pour stocker la valeur $(10)_h$ dans le mot d'adresse $(0001)_b$, la valeur 3 dans le mot 5. Réalisez-les.
— Comment placer sur le bus de données le mot stocké en mémoire à l'adresse 5 ? Réalisez cette opération.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Question 3
Construisez une mémoire comprenant 8 mots de 4 octets chacuns, adressable en octet. Réalisez les commandes d'écriture suivantes.

1. écrire le mot 0xFEDCBA98 à partir de l'adresse 0x00
2. écrire le mot 0x76543210 à partir de l'adresse 0x04
3. écrire le demi-mot 0x2222 à partir de l'adresse 0x10
4. écrire l'octet 0xAA à l'adresse 0x1C
5. écrire le mot 0xABABABAB à partir de l'adresse 0x07
6. écrire le demi-mot 0xABAB à partir de l'adresse 0x0E

Note : avec la mémoire créée ici, il n'est pas possible de réaliser en un transfert des écritures d'un ou deux octets (la valeur qui transite vers la mémoire est nécessairement sur 4 octets). Les octets non précisés dans les valeurs à écrire sont supposés être à zero.

Question 4
Repérez sur la figure précédente la cellule de mémorisation M4b et décrivez son interface, sans regarder sa réalisation interne.

La cellule élémentaire M4b comprend trois parties :

1. la cellule de mémorisation mem4b (cf. Figure 4.7),
2. l'élaboration de la commande d'écriture dans mem4b qui écrit le contenu de Din dans les points mémoire
3. l'élaboration de la commande de lecture qui place sur la sortie de la cellule le mot binaire mémorisé par M4b.

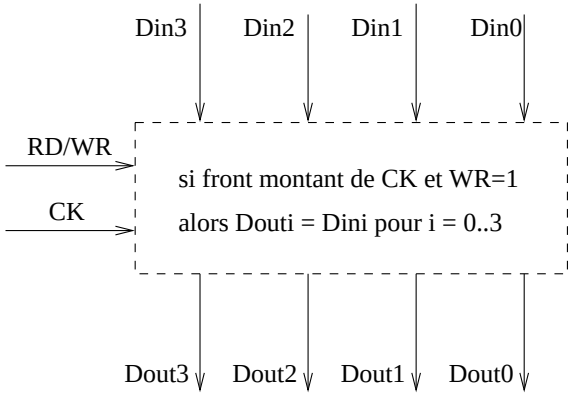


FIGURE 4.7 – Interface et contenu de la cellule de mémorisation M4b.

Question 5
Quelle est la condition d'écriture dans la cellule M4b ? Quel est l'effet de cette écriture ?

Question 6
Quelle est la condition de lecture dans la cellule M4b ?

Question 7
Que se passerait-il si la sortie de la cellule M4b représentait *à tout instant* la dernière valeur écrite dans cette cellule ?

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Question 8
Construisez dans un nouveau sous-projet une instance de la cellule M4b et validez le fonctionnement de la cellule sur le scénario suivant :

1. cycle 1 : Ecriture du mot $(1010)_b$ dans M4b. En phase d'écriture, la sortie de M4b est à 0.
2. cycle 2 : Lecture du contenu de la cellule M4b. La sortie de mem4b contient le mot mémorisé : $(1010)_b$
3. cycle 3 : Ecriture du mot $(0101)_b$ dans M4b. En phase d'écriture, la sortie de M4b est à 0.
4. cycle 4 : Lecture du contenu de la cellule M4b. La sortie de mem4b contient le mot mémorisé : $(0101)_b$

Considérez la mémoire de 4 mots de 4 bits et validez son fonctionnement :

Question 9
a) Vérifiez que dans les phases d'écriture, seule une cellule de mémorisation écrit effectivement la donnée Din.
b) Ecrivez les mots $(0001)_b, (0010)_b, (0011)_b, (0100)_b$ aux adresses $(00)_b, (01)_b, (10)_b et (11)_b$ respectivement. Vous détaillerez les suites de commandes appliquées à la mémoire.

Question 10
a) Vérifiez que dans les phases de lecture, seule une cellule de mémorisation produit sur sa sortie son contenu et que les sorties des autres cellules de mémorisation produisent 0.
b) Lisez séquentiellement les mots préalablement rangés aux adresses $(00)_b, (01)_b, (10)_b$ et $(11)_b$.

Question 11
Modifiez l'architecture proposée pour

1. construire une mémoire de quatre mots de deux quartets. Indication : la mémoire est structurée en 4 lignes (4 mots) et 2 colonnes (2 quartets) ; il faut donc étendre Din à 2 quartets et Addr à 3 bits, 2 pour les lignes et 1 pour les colonnes. Dans cette question, on suppose que le seul chargement possible est un chargement par mot.
2. considérer un chargement par quartet ou double quartet alignés. Indication : Il faut distinguer la taille du transfert (half ou word) à ajouter sur l'interface de la mémoire ; le chargement d'un mot complet a lieu dans une ligne. Le chargement d'un quartet nécessite d'identifier, dans une ligne, la colonne dans laquelle il faut placer le quartet de poids faible de Din. Il faut également s'assurer, dans un chargement mot, que l'adresse produite est alignée.

Exercice 22 – Manipulation d'un petit chemin de données composé d'une ALU et de 4 registres

Le petit chemin de données correspondant au circuit représenté sur la figure 4.8 discuté dans le cours vous est donné. Sur cette figure, n'est pas représentée l'horloge mais elle est une entrée de tous les registres pour qu'il fonctionne sur la même horloge.

Question 1
On souhaite réaliser l'addition de 3 nombres (par exemple -3, 5 et 6) avec ce chemin de

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

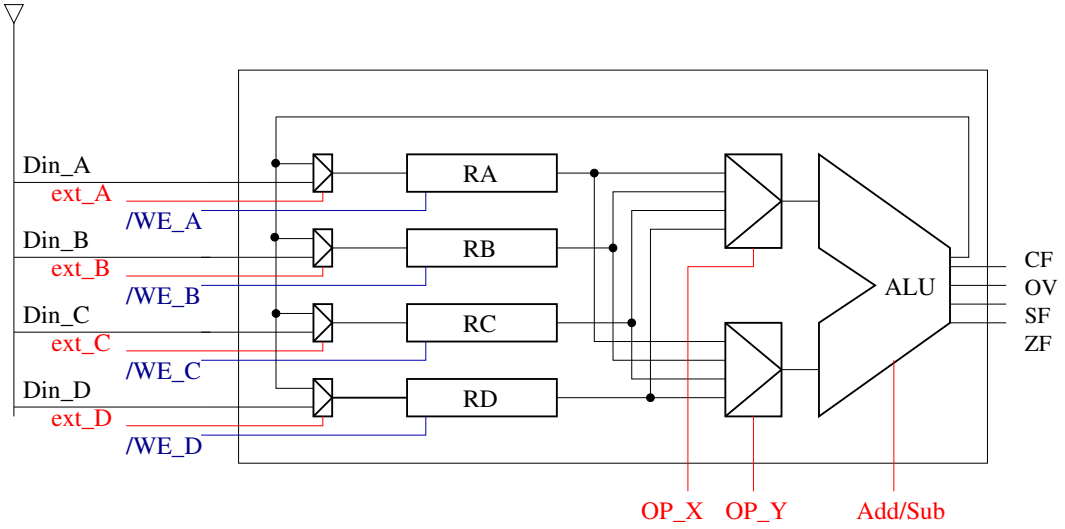


FIGURE 4.8 – Petit chemin de données sans son horloge.

données.
Sur une feuille, proposez une suite de mouvements de données permettant de réaliser l'addition de 3 nombres n, m et p. Vous utiliserez les registres RA, RB et RC pour charger les 3 valeurs à additionner et RD pour contenir le résultat.
Réalisez la suite d'opérations sur le chemin de données en positionnant les signaux qui permettent de réaliser chacune des opérations. Vérifiez le résultat, vous pouvez utiliser la fonctionnalité de traçage des signaux sous le menu *Simulate/logging* pour construire la table d'évolution des signaux.

Remarque :

— Initialement chaque registre contient la valeur 0

— **IMPORTANT : MODE OPERATOIRE POUR LA MANIPULATION DES REGISTRES.** Le contenu des registres étant potentiellement modifié lors de front montant d'horloge, il est préférable de manipuler les signaux lorsque l'horloge est à 0. Pour chaque manipulation, vous partirez de l'état où rien n'est autorisé en remettant tous les signaux d'écriture dans les registres à 1. Une fois les signaux positionnés pour un cycle vous ferez passer l'horloge de 0 à 1 puis de 1 à 0. Vous pourrez ensuite regarder l'effet de la manipulation réalisée.

Question 2
On souhaite réaliser sur ce chemin de données le calcul du maximum de 3 nombres n, m et p. On souhaite avoir le résultat du calcul dans le registre RD, et on suppose que les valeurs des 3 entiers dont on veut le maximum seront placées dans les registres RA, RB, RC.

Proposez la suite d'opérations à réaliser pour aboutir au résultat voulu sur une feuille (pensez que l'on peut utiliser les drapeaux pour déterminer les actions à réaliser en fonction du résultat de calcul !).

Simulez votre proposition avec 3 valeurs et avant de commencer la simulation n'oubliez pas de mettre à 0 le registre RD utilisé à la question précédente.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Chapitre 5

Systèmes séquentiels et machines à états

Contents

5.1 Introduction aux systèmes séquentiels

La partie commande du processeur est souvent réalisée à partir d'un ou plusieurs systèmes séquentiels synchronisés, de même que les contrôleurs de bus (vers la mémoire).

Les systèmes séquentiels sont des systèmes dont le comportement dépend :
— de la valeur des entrées courantes,
— de la séquence des valeurs entrées appliquée depuis une origine.

Les **fonctions combinatoires** sont de la forme $o_j = f(i_j)$ pour tout instant j (les délais de propagation des signaux au sein du circuit étant considérés comme négligeables).
Le graphe d'un circuit combinatoire est composé de portes logiques (combinatoires) et ne présente pas de cycle. La sortie dépend uniquement de l'entrée à l'instant courant.

Lorsque l'on a besoin de construire une sortie dépendant de l'instant courant mais également d'autres instants passés, il faut utiliser des systèmes séquentiels.

Exemple. Détecteur de fronts montants et descendants.
On considère un dispositif signalant par une alarme impulsionnelle le changement de valeur logique d'un signal ; l'alarme est déclenchée un instant après la détection du front.

Le dispositif, représenté sur la figure 5.1 analyse un signal d'entrée booléen i, évoluant au cours du temps, et produit les alarmes sur un signal de sortie booléen o.

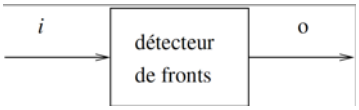


FIGURE 5.1 – Détecteur de fronts

Besoins → Il faut identifier différents instants permettant de localiser les valeurs successives du signal : le système est nécessairement cadencé sur une horloge et on considère que le front montant du signal d'horloge indique le début d'un nouvel instant.

Reprise de l'exemple. Un exemple de séquence d'entrée i/sortie o est de notre détecteur de fronts montants et descendant est donné dans le tableau ci-dessous. Les alarmes produites par un front montant survenant sur le signal i sont indiquées en rouge, celles induites par un front descendant sont indiquées en bleu :

instant	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
signal scruté i	0	0	0	0	1	1	0	0	0	0	1	1	1	0	1	1	1
alarme o	0	0	0	0	0	1	0	1	0	0	0	1	0	0	1	1	0

Besoins → Pour réaliser ce dispositif, il faut pouvoir analyser deux valeurs successives du signal d'entrée et les comparer ; en fonction du résultat de la comparaison, une alarme pourra être émise à l'instant suivant (positionnant le signal o à 1 pour une durée de 1 instant). La scrutation du signal d'entrée i se poursuit, afin de comparer plus tard deux

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

autres valeurs consécutives du signal d'entrée.

Les **fonctions séquentielles** sont de la forme $o_j = f(i_j, i_{j-1}, \ldots i_0)$, i_0 désignant la valeur des signaux d'entrée appliquée à l'instant origine. En pratique, la séquence d'entrées appliquée depuis l'origine est représentée sous la forme d'un état interne s_j, évoluant à chaque nouvelle entrée : $(o_j,s_{j+1}) = f'(i_j,s_j)$ et s_0 = état initial (origine). Le graphe d'un circuit séquentiel, lorsque les portes élémentaires sont des portes logiques combinatoires, contient des cycles qui réalisent la fonction de mémorisation de l'état (initial s_0 ou courant s_j) du système. On introduit dans la représentation des circuits logiques un symbole particulier représentant ces structures de mémorisation, et abstrayant ces cycles formés d'opérateurs combinatoires. Dans notre cas, ce symbole sera celui du registre.

Il existe de nombreuses réalisations possibles de ces systèmes séquentiels. La Machine de Moore est une représentation opérationnelle : elle décrit l'évolution de l'état interne sous la forme d'une fonction combinatoire des entrées et de l'état interne, et l'évolution des signaux de sortie sous la forme d'une fonction combinatoire de l'état interne.

5.2 Définition d'une Machine de Moore

Une machine de Moore M est la donnée des éléments $M =< S, I, O, T, G, S_0 >$ avec
- S = ensemble fini d'états (états de contrôle)
- I = ensemble fini des signaux d'entrée
- O = ensemble fini des signaux de sortie
- T = fonction de transition calculant l'état suivant l'état suivant à partit de l'état courant et de l'entrée courante : $S \times 2^I \rightarrow S$
- G = fonction de génération calculant la sortie courante à partir de l'état courant : $S \rightarrow 2^O$
- S_0 = état initial

Remarque 1 : T et G sont des fonctions combinatoires, S désigne l'état interne de la machine de Moore, et regroupe les structures de mémorisation de l'état.

Remarque 2 : On se focalise sur les systèmes déterministes et complets.

La figure 5.2 représente un schéma générique d'un circuit séquentiel.
L'ensemble des comportements admissibles par le système séquentiel peut être représenté en sous la forme d'un graphe orienté et étiqueté (appelé automate, ou Machine à états), qui décrit de façon explicite la succession des états et des sorties produites par la machine lorsqu'elle est soumise à n'importe quelle séquence d'entrée.

5.3 Représentation graphique des séquences d'exécution de la Machine de Moore - représentation sous forme de Machine à états (MAE)

L'ensemble des comportements admissibles par la Machine de Moore peut être décrit sous la forme d'un graphe orienté étiqueté : Le graphe $GR =< E,V,L_V, E_0 >$ associé à

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

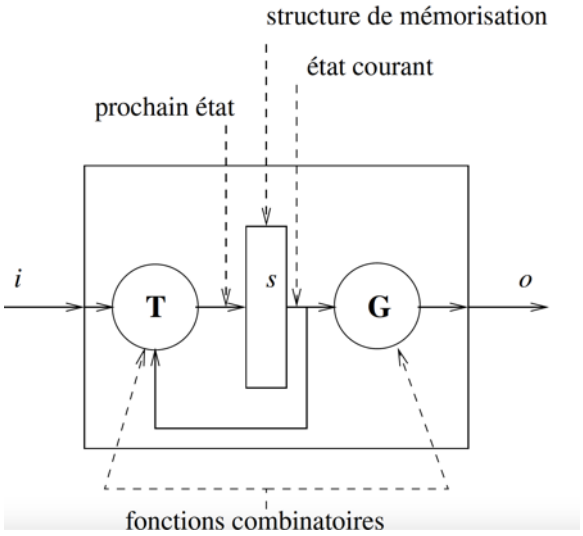


FIGURE 5.2 – Détecteur de fronts

une machine de Moore M est tel que :

- E = ensemble des sommets étiquetés par les états de M
- V = arcs : transitions $\subseteq E \times E'$
  $v = (s,s') \in V \iff < s,e_i,s' > \in S \times 2^I \times S$ et $s' = T(s,e_i)$
  s est l'état source et s' l'état destination de la transition ; elle est franchie lorsque, la machine étant dans l'état s, la configuration e_i est présente sur les signaux d'entrée.
- L_V = étiquettes des états : $E \rightarrow 2^O$
  $\forall\ s \in\ V, L_V(s) = e_o \iff\ e_o = G(s)$
  e_o est la configuration des signaux de sortie appliquée par la machine lorsqu'elle est dans l'état s.
- E_0 : sommet correspondant à l'état initial de M : S_0

Reprise de l'exemple. La figure 5.3 décrit le graphe représentant les comportements induits par une Machine de Moore associée aux comportements du détecteur de fronts.

Les états sont représentés par les cercles, nommés $s_0, \ldots\ s_4$. s_0 est l'état initial.

Chaque arc reliant deux états s_i et s_j est étiqueté par une valeur appliquée sur les entrées, et représente une transition de la machine à états. Par exemple, la transition reliant les états s_0 et s_1, étiquetée par la condition $i = 0$ indique que lorsque le système est dans l'état (courant) s_0, si la valeur sur l'entrée à l'instant courant est « 0 », alors l'état à l'instant suivant sera l'état s_1.

L'union des transitions représente la fonction de transition de la machine de Moore (la machine est déterministe : d'un état courant, il n'est pas possible d'aller vers deux états différents pour une même valeur d'entrée).

Chaque état contient également une étiquette indiquant la valeur à appliquer sur la sortie à l'instant courant (qui ne dépend que de l'état courant). La fonction de génération est composée de l'union des valeurs des sorties positionnées dans chaque état. Par

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

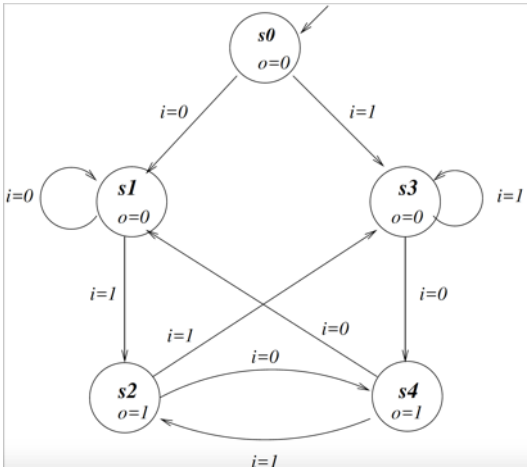


FIGURE 5.3 – Graphe représentant les comportements induits par une machine de Moore associée aux comportements du détecteur de fronts

exemple, lorsque la machine est dans l'état s_1, la sortie o est positionnée à « 0 » et lorsque la machine est dans l'état s_2, la sortie est positionnée à « 1 ».

Les comportements réalisés par la Machine correspondent à la suite des états visités en suivant une transition du graphe à chaque nouvel instant. Par exemple, de l'état initial s_0, la machine peut réaliser la séquence suivante :

s_0, s_1, s_1, s_1, s_1,s_2,s_3, s_4, s_1,s_1,s_1,s_2,s_3,s_3,s_4,s_2,s_3, s_3.

Cela se produit lorsque la séquence i=0, i=0, i=0, i=0,i=1, i=1, i=0, i=0, i=0, i=0, i=1, i=1, i=1, i=0, i=1, i=1, i=1 est appliquée à partir de l'état initial.

La séquence de valeurs sur la sortie est alors déterminée par la valeur de la sortie dans chaque état visité de la séquence :

$(s_0, o = 0) \rightarrow (s_1, o = 0) \rightarrow (s_1, o = 0) \rightarrow (s_1, o = 0) \rightarrow (s_1, o = 0) \rightarrow (s_2, o = 1) \rightarrow (s_3, o = 0) \rightarrow (s_4, o = 1) \rightarrow (s_1, o = 0) \rightarrow (s_1, o = 0) \rightarrow (s_1, o = 0) \rightarrow (s_2, o = 1) \rightarrow (s_3, o = 0) \rightarrow (s_3, o = 0) \rightarrow (s_4, o = 1) \rightarrow (s_2, o = 1) \rightarrow (s_3, o = 0) \rightarrow (s_3, o = 0)$

En effet, on obtient bien :

instant	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
signal scruté i	0	0	0	0	1	1	0	0	0	0	1	1	1	0	1	1	1
état courant	s_0	s_1	s_1	s_1	s_1	s_2	s_3	s_4	s_1	s_1	s_1	s_2	s_3	s_3	s_4	s_2	s_3
alarme o	0	0	0	0	0	1	0	1	0	0	0	1	0	0	1	1	0
état suivant	s_1	s_1	s_1	s_1	s_2	s_3	s_4	s_1	s_1	s_1	s_2	s_3	s_3	s_4	s_2	s_3	s_3

La lecture se fait par colonne. A l'instant t : de l'état courant s_j, la lecture l'entrée i induit l'état suivant s_k, et l'état courant s_j produit la sortie o. L'état suivant s_k de la colonne à l'instant t devient l'état courant de la colonne à l'instant $t + 1$.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

5.4 Construction du circuit séquentiel associé à une machine de Moore représentée sous forme d'une machine à états.

A partir de la description d'un système séquentiel sous forme de Machine à états, on peut construire el ciruit séquentiel produisant les mêmes séquences d'entrées/sorties. Le mode opératoire est le suivant :

1. Dénombrer les sommets de GR et leur associer un codage binaire dans M. Ce codage détermine le nombre d'éléments mémorisants qui seront présents dans le circuit. La sortie de chaque élément mémorisant désigne une partie de l'état courant de M.
2. Construire la table d'adjacence des sommets de GR (association état courant – entrée courante - état successeur), l'étendre avec leur représentation binaire.
3. Pour chaque bit b du mot binaire codant un état successeur de S : identifier les états courants induisant $b = 1$, l'union des états courants représente la condition de passage ou maintien à 1 du bit b au prochain cycle ; elle peut être mise sous forme d'une expression booléenne représentable par un circuit combinatoire dépendant de l'état courant, et des signaux d'entrées. L'ensemble des circuits combinatoires associés à tous les bits b représente la fonction de transition T.
4. Pour chaque signal de sortie o :

   déterminer les états courants induisant $o = 1$, leur union représente la condition de passage ou de maintien à 1 du signal o au cycle courant ; elle peut être mise sous forme d'une expression booléenne représentable par un circuit combinatoire dépendant de l'état courant. L'ensemble des circuits combinatoires associés à tous les signaux de sortie o représente la fonction de génération G.

5.5 Exemple complet

On se propose de construire le circuit séquentiel décrivant le comportement du détecteur de fronts montants et descendants.

Étape 1 On se donne un codage binaire des états $s_0, \ldots s_4$, par exemple le codage sur 3 bits donné dans la table ci-dessous.

Chaque bit du mot binaire d'état sera mémorisé dans un registre (structure mémorisante élémentaire), dénommé respectivement R2, R1 et R0 (R2 pour le bit de poids fort et R0 pour le bit de poids faible).

État du graphe	Codage sur 3 bits		
	R2	R1	R0
s_0	0	0	0
s_1	0	0	1
s_2	0	1	0
s_3	0	1	1
s_4	1	0	0

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Étape 2 La table d'adjacence de la machine découle directement de la structure du graphe et du codage adopté. Chaque ligne de la table représente une transition de la machine de Moore.

Etat courant	Entrée courante	Etat suivant	R2	R1	R0	i	R'2	R'1	R'0
s_0	$i=0$	s_1	0	0	0	0	0	0	1
s_0	$i=1$	s_3	0	0	0	1	0	1	1
s_1	$i=0$	s_1	0	0	1	0	0	0	1
s_1	$i=1$	s_2	0	0	1	1	0	1	0
s_2	$i=0$	s_4	0	1	0	0	1	0	0
s_2	$i=1$	s_3	0	1	0	1	0	1	1
s_3	$i=0$	s_4	0	1	1	0	1	0	0
s_3	$i=1$	s_3	0	1	1	1	0	1	1
s_4	$i=0$	s_1	1	0	0	0	0	0	1
s_4	$i=1$	s_2	1	0	0	1	0	1	0

Étape 3 Construction d'une expression Booléenne de la fonction de transition associée à chaque bit du mot d'état binaire :

Pour le bit R0 : scruter la colonne R0' (contribution du bit R0 pour l'état suivant), et construire une expression booléenne de la fonction :

R2	R1	R0	i	R0'
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0

Une expression d'évolution de R0' est :

$$R0' = \overline{R2}.\overline{R1}.\overline{R0}.(i+\bar{i}) + \overline{R2}.\overline{R1}.R0.i + \overline{R2}.R1.\overline{R0}.i + \overline{R2}.R1.R0.i + R2.\overline{R1}.\overline{R0}.\bar{i}.$$

On peut simplifier cette expression et en déduire un circuit combinatoire, d'entrées R0, R1, R2, i (l'état et l'entrée courants), et dont la sortie est la valeur du bit R0 à l'état suivant.

$$\begin{aligned} R0' &= \overline{R2}.\overline{R1}.\overline{R0} + \overline{R2}.\overline{R1}.R0.\bar{i} + \overline{R2}.R1.i + R2.\overline{R1}.\overline{R0}.\bar{i} \\ &= \overline{R1}.\overline{R0}.(\overline{R2} + R2.\bar{i}) + \overline{R2}.(\overline{R1}.R0.\bar{i} + R1.i) \\ &= \overline{R1}.\overline{R0}.(\overline{R2} + \bar{i}) + \overline{R2}.(\overline{R1}.R0.\bar{i} + R1.i) \end{aligned}$$

Il faut procéder de la même manière pour les bits R1 et R2 :

$$R1' = i$$
$$R2' = \overline{R2}.R1.\overline{R0}.\bar{i} + \overline{R2}.R1.R0 = \overline{R2}.R1.\bar{i}$$

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Étape 4 Pour la fonction de génération, on construit la table associant à chaque état courant la valeur de sortie :

Etat du graphe	Codage sur 3 bits			Sortie o
	R2	R1	R0	
s_0	0	0	0	0
s_1	0	0	1	0
s_2	0	1	0	1
s_3	0	1	1	0
s_4	1	0	0	1

On en déduit une expression booléenne de la fonction de sortie :

$G = \overline{R2}.R1.\overline{R0} + R2.\overline{R1}.\overline{R0}$

On peut simplifier cette expression et en déduire un circuit combinatoire, d'entrées R0, R1 et R2 (l'état courant), et de sortie o (la sortie du dispositif).

Une réalisation du détecteur de front est donnée dans la figure 5.5

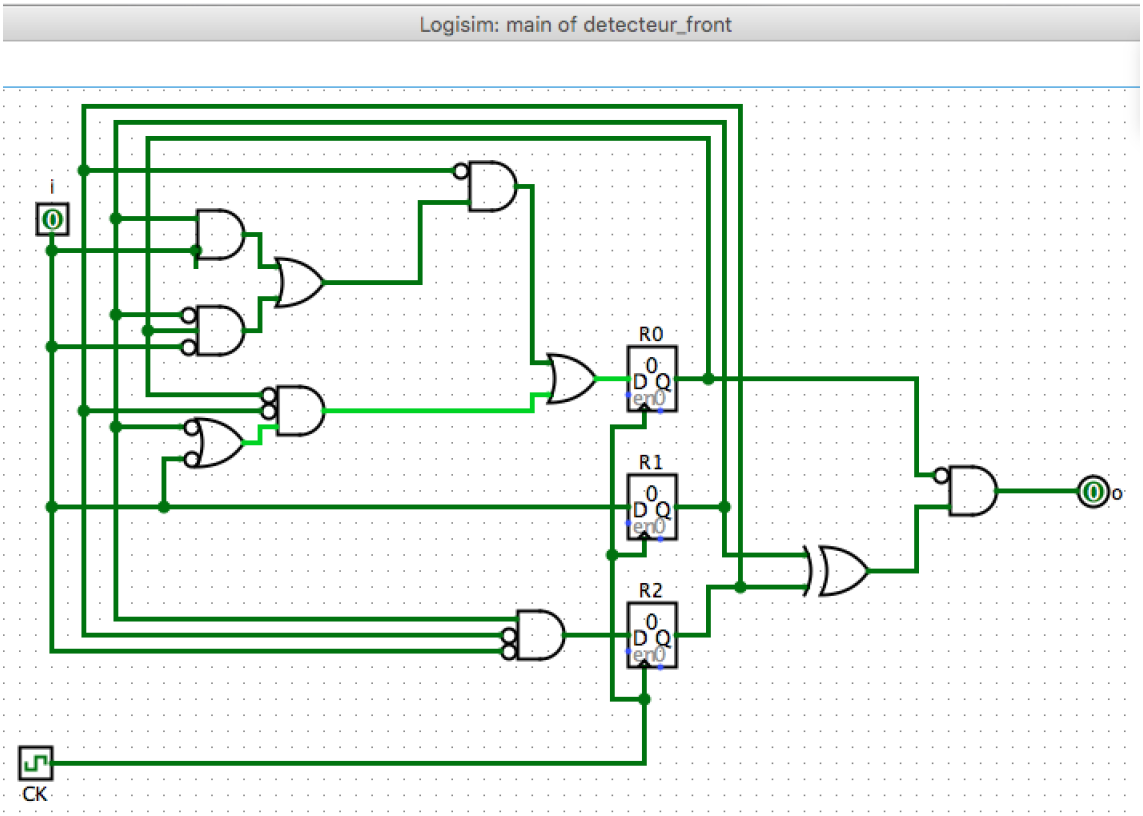


FIGURE 5.4 – Réalisation du détecteur de fronts

5.6 Exercices

Exercice 23 – Détecteur de fronts montants.
En reprenant l'exemple donné précédemment, décrire la machine à états d'un circuit dé-

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

tecteur de fronts montants ; en déduire la table d'adjacence et les fonctions de transition et génération associées. Implanter le circuit sur logisim et testez son fonctionnement sur la séquence fournie en exemple :

instant	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
signal scruté i	0	0	0	0	1	1	0	0	0	0	1	1	1	0	1	1	1
alarme o	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0

Exercice 24 – Transferts de données entre le processeur et la mémoire.

Lors des lectures et écriture, le processeur et la mémoire s'échangent des informations sur le bus. Ces échanges sont assurés par des contrôleurs de bus (dans la partie opérative du processeur et dans la partie contrôle de la mémoire) qui implantent un protocole de communication précis. Ces protocoles sont complexes car supposent plusieurs types d'échanges (1 seul mot à la fois ou des rafales de mots, 1 seul composant pouvant initier les transferts ou plusieurs). On se place dans un cas très simplifié où le processeur est le seul composant pouvant initier des transferts, ces transferts sont d'un seul mot, et le processeur garde la possession du bus tant que le transfert n'est pas achevé.

Le protocole considéré est le suivant :
LECTURE :

1. processeur : placer l'adresse et le sens du transfert (LECTURE) sur le bus, puis attendre l'acquittement de la mémoire (signal ACK sur le bus)
2. mémoire : récupérer les informations sur le bus et sélectionner la donnée, la placer sur le registre de sortie
3. mémoire : recopier le registre de sortie sur le bus et positionner le signal ACK et le code de retour (si OK, pas d'erreur, sinon ERR)
4. processeur : à la réception du signal ACK et code retour OK, recopier la donnée du bus dans un registre interne <fin du transfert>, puis le recopier dans le registre résultat. si le code d'erreur était ERR, lever une exception vers le processeur avec l'information « erreur lecture ».

ECRITURE :

1. processeur : placer l'adresse, le sens du transfert (ECRITURE) et la donnée sur le bus, puis attendre l'acquittement de la mémoire (signal ACK sur le bus)
2. mémoire : récupérer les informations sur le bus et recopier la donnée à son emplacement mémoire.
3. mémoire : positionner le signal ACK et le code d'erreur
4. processeur : à la réception du signal ACK et code retour OK, <fin du transfert>. Si le code d'erreur était ERR, lever une exception vers le processeur avec l'information « erreur écriture».

Question 1
Détaillez les informations à véhiculer sur les nappes de fils du bus

Question 2
Décrivez les machines d'états du contrôle de bus du processeur et de la mémoire, pour les opérations de lecture et d'écriture.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Question 3
Quels problèmes pourraient se poser avec ce protocole si il y a plusieurs composants qui peuvent initier des transferts sur le bus (par exemple plusieurs processeurs partageant une même mémoire, ou un DMA) ? quel composant faudrait-il ajouter à la structure du bus ? Quel impact cela pourrait avoir sur les machines d'états décrites précédemment (Q2) ?

Question 4
Est-il intéressant de réaliser des transferts d'un seul mot sur le bus ? que peut-on proposer ? quelles conséquences cela pourrait avoir sur les machines d'états déjà décrites précédemment (Q2) ?

Exercice 25 – Commande des mouvements sur un petit chemin de données.

On considère le petit chemin de donnée présenté au chapitre 4 (p.51). On suppose qu'il est complété par une unité de décodage d'instruction, fournissant le code opération, les registres sources et destination, les constantes immédiates et les adresses de saut. De plus, un mécanisme externe fait progresser le compteur de programme d'une instruction à la suivante.

On considère le jeu d'instructions (fictif) suivant :

— `li Rdest, Imm (étendu 32b)`
— `add Rdest, Rsc1, Rsc2` et `sub Rdest, Rsc1, Rsc2`
— `bz label` et `cmp R1, R2, label,`
— `j label`

Et deux programmes réalisés au moyen de ce langage :

```
prog1 :         li RA, 12          # RD <- 12 - 45 + 2
                li RB, 45
                li RC, 2
                sub RD, RA, RB
                add RD, RC, RD
```

```
prog2 :         li RA, 12        1 RA <- pgcd(12, 4)
                li RB, 4
        boucle :
                sub RC, RA, RB
                bz fin
                cmp RA, RB, ds_RA
                sub RB, RB, RA
                j boucle
        ds_RA :
                sub RA, RA, RB
                j boucle
        fin :   # le pgcd est dans RA et dans RB
```

On souhaite construire la Machine de Moore permettant d'enchainer les commandes à appliquer sur le chemin de données pour exécuter les suites d'instructions définies par les programmes 1 et 2.

Question 1
Proposez une interface pour cette machine : quelles doivent être les informations en entrées de la machine ? Quels sont les signaux qu'elle positionne en sortie ?

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Question 2
Décrivez la suite de transitions et d'états de la machine permettant de réaliser l'opération `li Rdest, Imm`.

Question 3
Ajoutez à la machine les suites de transitions et d'états permettant de réaliser les opérations `add Rdest, Rsc1, Rsc2` et `sub Rdest, Rsc1, Rsc2`.

Question 4
Ajoutez à la machine la suite de transitions et d'états permettant de réaliser l' opération `j label`. Remarque : Si vous n'aviez pas intégré le compteur ordinal en sortie de votre machine, il faudra l'intégrer ici, et on suppose que `label` désigne directement l'adresse de la prochaine instruction à exécuter.

Question 5
Ajoutez à la machine la suite de transitions et d'états permettant de réaliser l' opération `bz label`.

Question 6
Ajoutez à la machine la suite de transitions et d'états permettant de réaliser l' opération `cmp Rsc1, Rsc2, label`.

Exercice 26 – Scrutation d'un signal d'entrée.

on souhaite construire un dispositif, analysant un signal binaire évoluant dans le temps. Le dispositif émet une alarme (sa sortie passe de « 0 » à « 1 ») si le signal analysé comprend un caractère non doublé consécutivement autrement dit, il existe une suite maximale de « 0 » ou de « 1 » consécutifs de longueur impaire.

instant	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
signal scruté	0	0	0	0	1	1	0	0	0	0	1	1	1	0	0	1	1
alarme	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Chapitre 6

Instructions et jeu d'instruction MIPS

Contents

6.1 Notion d'instruction machine et jeu d'instructions

Un programme de haut niveau (écrit en C par exemple) est traduit en un ensemble d'instructions compréhensibles par le processeur qui doit exécuter le programme. Ces instructions sont dites en langage machine. Chaque processeur dispose d'un ensemble de traitements qui s'appuient sur son architecture (registres qu'il contient, organisation de la mémoire qu'il supporte, format des instructions qui a été défini en même temps que son architecture).

Jeu d'instructions La vue externe d'un processeur peut être définie par l'ensemble des instructions qu'il est capable de traiter. Ces instructions sont stockées en mémoire et codées en binaire.

Le jeu d'instructions d'un processeur (ISA) est la donnée :

1. de l'ensemble des instructions qu'il peut effectuer
2. du codage de ces instructions en binaire

Qu'est ce qu'une instruction ? Une instruction en langage machine est une commande donnée au processeur qui définit :

1. Le traitement à effectuer maintenant, à savoir :
   — l'opération mise en jeu (addition, opération logique, opération mémoire,...)
   — les opérandes sur lesquelles elle porte : la ou les opérandes sources, l'opérande destination s'il y en a un.
2. Quelle sera la prochaine instruction à exécuter : il existe un modèle implicite qui est séquentiel. La prochaine instruction à exécuter est celle qui est implantée en mémoire à la suite de l'instruction courante. De ce fait, l'adresse de la prochaine instruction à exécuter n'est pas représentée si elle suit le modèle d'exécution implicite. Du coup, il existe des instructions spéciales qui indiquent l'adresse de la prochaine instruction lorsque ce n'est pas systématiquement celle qui suit celle en cours d'examen.

6.2 Jeu d'instructions MIPS

6.2.1 Les différentes classes d'instructions

6.2.1.1 Instructions arithmétiques et logiques

Ces instructions utilisent l'ALU pour réaliser le calcul.

Instructions arithmétiques

— `add $2, $3, $4` dont l'effet est `$2` ← `$3+$4`
— `addi $2, $3, 0x25` dont l'effet est `$2` ← `$3+0x00000025` ; l'immédiat est étendu de 16 à 32 bits de manière signée
— `addi $2, $3, 0xFF` dont l'effet est `$2` ← `$3+0xFFFFFFFF` ; l'immédiat est étendu de 16 à 32 bits de manière signée
— `mult $4, $2` dont l'effet est `(HI,LO)` ← `$4*$2`

Instructions logiques

— `or $2, $3, $4` dont l'effet est `$2` ← `$3|$4` (ou logique bit à bit)
— `ori $2, $3, 0xFF00` dont l'effet est `$2` ← `$3|0x0000FF00` ; l'extension de l'immédiat est non signée
— `and $4, $2, $3` dont l'effet est `$2` ← `$3&$4` (et logique bit à bit)
— `andi $2, $3, 0xFF00`dont l'effet est `$2` ← `$3&0x0000FF00` ; l'extension de l'immédiat est non signée
— `xor $4, $2, $3` dont l'effet est `$4` ← `$2`$\hat{}$`3` (ou exclusif bit à bit)

Instructions de décalage

— `sllv $2, $3, $4` dont l'effet est `$2` ← `$3«$4` (décalage à gauche de $4 bits)
— `sll $2, $3, 2` dont l'effet est `$2` ← `$3«2` (décalage à gauche de 2 bits)
— `srav $2, $3, $4` dont l'effet est `$2` ← `$3»$4` (décalage à droite signé de $3 bits)
— `sra $2, $3, 4` dont l'effet est `$2` ← `$3»4` (décalage à droite signé de 4 bits)
— `srlv, $2, $3, $4` dont l'effet est `$2` ← `$3»$4` (décalage à droite non signé de $4 bits)
— `srl $4, $3, 4` dont l'effet est `$2` ← `$3»$4` (décalage à droite non signé de 4 bits)

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Instruction d'affectation de registre

— `lui $2, 0xABCD` dont l'effet est `$2` ← `0xABCD000` (chargement sur les 16 bits de poids fort et mise à zéro des 16 bits de poids faible)
— `mfhi $4` dont l'effet est `$4` ← `HI` (le contenu de HI est copié dans $4)
— `mflo $4` dont l'effet est `$4` ← `LO` (le contenu de LO est copié dans $4)

6.2.1.2 Instruction de transferts mémoire

Ces instructions lisent ou écrivent des valeurs en mémoire, en indiquant l'adresse mémoire (contenu d'un registre + un immédiat) et le registre contenant la valeur à écrire/où écrire la valeur lue en mémoire. La taille du mot à lire/écrire est indiqué dans le code de l'opération.

Instruction de lecture mémoire

— `lw $4, 0($3)` dont l'effet est `$4` $\leftarrow_{4octets}$ `MEM[$3+0]` (lecture mémoire de 4 octets depuis l'adresse `$3+0` et le résultat est mis dans $4)
— `lh $4, 2($3)` dont l'effet est `$4` $\leftarrow_{2octets}$ `MEM[$3+2]` (lecture mémoire de 2 octets depuis l'adresse `$3+2` et le résultat est mis dans $4 ; le demi-mot lu est étendu sur 32 bits de manière signée)
— `lb $4, -12($3)` dont l'effet est `$4` $\leftarrow_{1octet}$ `MEM[$3-12]` (lecture mémoire de 1 octet depuis l'adresse `$3-12` et le résultat est mis dans $4 ; l'octet lu est étendu sur 32 bits de manière signée)

Instruction d'écriture mémoire

— `sw $4, 0($3)` dont l'effet est `$4` $\rightarrow_{4octets}$ `MEM[$3+0]` (écriture en mémoire des 4 octets de $4 à partir de l'adresse `$3+0`)
— `sh $4, 2($3)` dont l'effet est `$4` $\rightarrow_{2octets}$ `MEM[$3+2]` (écriture en mémoire des 2 octets de poids faible de $4 à partir de l'adresse `$3+0`)
— `sb $4, -12($3)` dont l'effet est `$4` $\rightarrow_{1octet}$ `MEM[$3-12]` (écriture en mémoire de l'octet de poids faible de $4 à partir de l'adresse `$3+0`)

6.2.1.3 Rupture de séquence

Ces instructions indiquent quelle est l'adresse de la prochaine instruction. Il y a deux types de saut, les sauts inconditionnels qui ont toujours lieu ou les conditionnels qui sont effectués si et seulement une condition est vérifiée.

Instruction de sauts inconditionnels

— `j etiquette_inst` dont l'effet est `PC` ← `etiquette_inst`
— `jal ma_fonction` dont l'effet est `$31` ← `PC+4` et `PC` ← `ma_fonction`
— `jr $31` dont l'effet est `PC` ← `$31`

Instruction de sauts conditionnels

— `beq $0, $4, etiquette_inst` dont l'effet est `PC` ← `PC+4` si `$0!=$4`, si `$0=$4` alors `PC` ← `etiquette_inst`
— `bltz $4, etiquette_inst` dont l'effet est `PC` ← `PC+4` si `$4<0`, si `$4`$\geq$`0` alors `PC` ← `etiquette_inst`

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

6.2.1.4 Instructions système

Elles correspondent à une demande de service fourni par le système. C'est l'instruction `syscall` qui réalise cette demande de service. Le système offre un certains nombres de service, chacun a un numéro qu'il faut placer dans le registre `$2` avant d'exécuter l'instruction `syscall`.

6.2.2 Codage binaire des instructions du MIPS

Jusqu'à présent, nous avons vu le codage des données (entiers naturels, relatifs, et caractères alphanumériques). On s'intéresse ici au codage des instructions. Celui-ci est donné avec l'ISA d'un processeur qui décrit les instructions que peut exécuter le processeur et leur codage binaire.

En MIPS toutes les instructions sont codées sur 32bits.

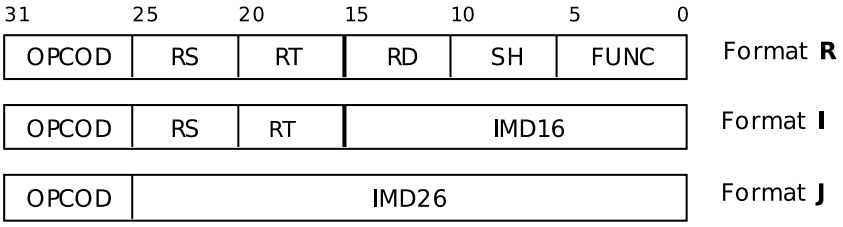


FIGURE 6.1 – Format de codage des instructions MIPS

Format de codage

Il y a 3 formats de codage, appelés R, I et J. Ils sont illustrés sur la figure 6.1.

- Le format R est utilisé pour les instructions arithmétiques et logiques avec 2 opérandes sources de type registre et pour toutes les instructions de décalage (qu'il y ait 1 ou 2 opérandes sources de type registre).
- Le format I est utilisé pour toutes les instructions arithmétiques et logiques avec 2 registres opérandes et un immédiat (sauf opération de décalage), les accès mémoire et les sauts conditionnels.
- Le format J est utilisé par les instructions de saut inconditionnel avec un adressage direct.

Chaque format contient différents champs. Chaque champ contient une information (opération, opérande, ...). Tous les formats ont un champ OPCOD qui occupe les bits 31 à 26. Le champ OPCOD code l'opération de l'instruction. Sa valeur est donnée par une table faisant la correspondance entre la valeur de OPCOD qui est sur 6 bits et le mnémonique de l'opération dans l'instruction assembleur. Ce champ OPCOD permet au processeur de déterminer l'opération à effectuer en fonction d'une table de codage ainsi que la la manière de décoder les 26 bits restant (donc le format de (dé)codage).

Exemples

Soit le mot 0x2062ABCD = 0010 0000 0110 0010 1010 1011 1100 1101. Il correspond à une instruction dont le code opération (OPCOD) vaut 001000 soit un addi. Un addi a la forme `addi Rs, Rt, Imm`. Il est codé avec le format I : on en déduit donc que les bits 21 à 25 et 16 à 20 correspondent aux registres Rs et Rt, qui sont respectivement 00011 soit $3 et 00010

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

DECODAGE OPCOD

INS 31 : 29 \ INS 28 : 26	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								

FIGURE 6.2 – Table de correspondance entre la valeur de OPCOD et le mnémonique d'une instruction. Par exemple, l'opération addi a pour code opération 001000. Et le code 100011 correspond à l'instrustion lw.

OPCOD = SPECIAL

INS 5 : 3 \ INS 2 : 0	000	001	010	011	100	101	110	111
000	SLL		SRL	SRA	SLLV		SRLV	SRAV
001	JR	JALR			SYSCALL	BREAK		
010	MFHI	MTHI	MFLO	MTLO				
011	MULT	MULTU	DIV	DIVU				
100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
101			SLT	SLTU				
110								
111								

FIGURE 6.3 – Table de correspondance entre le code FUNC et le mnémonique d'une instruction lorsque OPCOD = SPECIAL dans la table 6.2.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

soit $2. L'immédiat est 1010 1011 1100 1101 soit 0xABCD. L'instruction correspondante est donc `addi $3, $2, 0xABCD`.

OPCOD SPECIAL

Dans certains cas, le code opération désigne une famille d'opérations qui sont précisées par d'autre champ. C'est le cas lorsque pour la valeur de l'OPCOD 000000 = SPECIAL qui nécessite le décodage des bits 5 à 0 de l'instruction pour déterminer l'opération (champ FUNC pour function). L'opération est alors donnée par une deuxième table illustrée sur la figure 6.3.

Exemples

Si le champ opcode = 000000 et INS[5..0] = 100 000 alors il s'agit d'un add.

Soit 0x00641020 = 0000 0000 0110 0100 0001 0000 0010 0000.

On a opcode = 000000 il s'agit du format R. Le champ FUNC vaut 100000 il s'agit d'un add. Un add a la forme `add rd, rs, rt`. Il faut alors décoder les champs INS[25..21] désignant Rs, INS[20..16] désignant Rt et INS[15..11] désignant Rd. On a dans cet exemple Rs = 00011, Rt = 00100 et Rd = 00010.

L'instruction correspondante est donc `add $2, $3, $4`

6.3 Cycle d'exécution d'une instruction et chemin de données

6.3.1 Cycle d'exécution d'une instruction

L'exécution d'une instruction nécessite plusieurs étapes :

— Lire l'instruction en mémoire
— Décoder l'instruction (quelle opération, quelles opérandes)
— Exécuter l'instruction (réaliser le traitement correspondant à l'instruction)
— Déterminer l'adresse de la prochaine instruction

6.3.2 Chemin de données du MIPS

Chemin de données d'un processeur La plupart des processeurs sont composés d'un chemin de données et d'un séquenceur, ce dernier ayant pour rôle de commander le chemin de données et réguler ses interactions avec la mémoire vive. Le chemin de données d'un processeur représente de manière abstraite toutes les voies de circulation des données disponibles dans le processeur pour réaliser les traitements spécifiés par une instruction en langage machine. Il permet notamment de représenter les mouvements de données au sein du processeur lors de l'exécution d'une instruction. La figure 6.4 représente le chemin de données du MIPS que nous considérons dans ce cours.

Toutes les voies ne sont pas ouvertes simultannéement (court-circuit sinon). L'ouverture et la fermeture d'une voie est commandée par une porte 3 états dont l'état est commandé par la partie unité de contrôle en fonction de l'instruction à exécuter. De même, les commandes d'écriture dans les registres sont positionnées en fonction de l'instruction en cours de réalisation.

C'est l'unité de contrôle du processeur qui positionne les signaux et cadence les diverses opérations à réaliser sur le chemin de données pour l'exécution d'une instruction.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

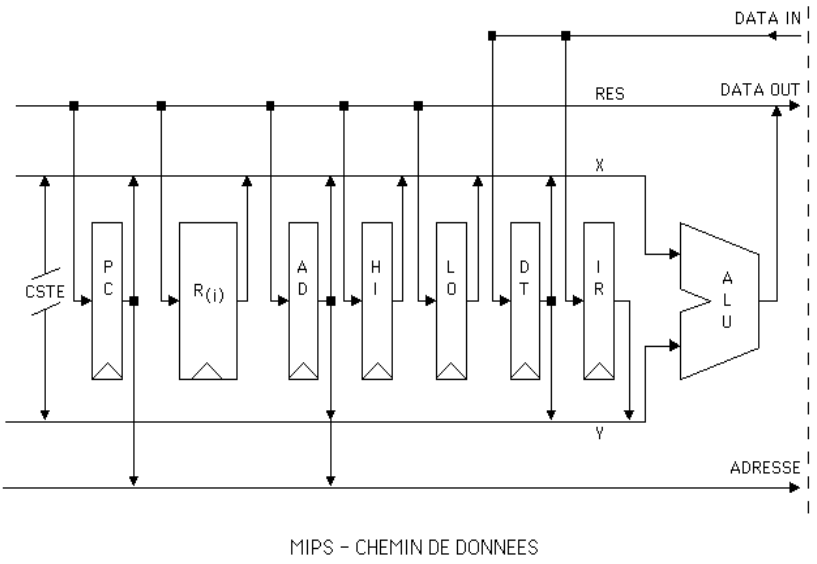


FIGURE 6.4 – Chemin de données du MIPS

Registres du chemin de données du MIPS En plus des registres accessibles par le programmeur, $0, $1,..., $30, $31, hi et lo, le chemin de données représenté sur la figure 6.4 contient les registres suivants.

- Le registre PC ou *Program Counter* qui contient l'adresse de l'instruction à exécuter/en cours d'exécution/de la prochaine instruction.
- Le registre IR ou *Instruction Register* qui contient l'instruction en cours d'exécution, c'est la derniere instruction lue en mémoire dans l'étape "lire l'instruction en mémoire" et qui provient de DATA_IN, le bus de données permettant des transfert entre la mémoire et le processeur.
- Le registre DT ou *Data Transfert*. C'est le registre dans lequel est rangée toute donnée lue en mémoire par une instruction de lecture mémoire, soit dans l'étape "executer l'instruction" du cycle d'exécution d'une telle instruction. Les données lues en mémoire arrivent au processeur via le bus de données DATA_IN.
- Le registre AD ou *ADdress*. Il contient les adresses à placer sur le bus d'adresse pour réaliser les transferts avec la mémoire. C'est également un registre qui peut stocker des résultats intermédiaires.

Illustration de l'exécution d'une instruction sur le chemin de données
Soit l'instruction `addu $4, $3, $2` d'adresse 0x04000118 et de codage 0x00622021. Voici son exécution détaillée :

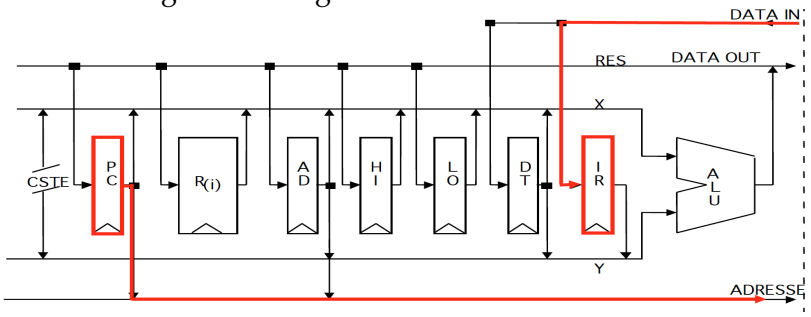
1 - Lire l'instruction

- Le PC contient l'adresse de l'instruction à exécuter, soit 0x04000118.
- Pour réaliser le transfert mémoire en lecture de l'instruction, il faut :
  1. mettre l'adresse 0x4000118 (contenu de PC) sur le bus d'adresse avec la commande de lecture d'un mot (4 octets)
  2. récupérer le mot 0x00622021 sur DATA_IN et l'écrire dans le registre IR

  Les mouvements de données engendrés pour lire une instruction sont repré-

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

sentés en rouge dans la figure ci-dessous :



2 - Décoder l'instruction

— L'instruction codée en binaire est 0x00622021.

— Le décodage de l'opération (champ opcod) indique que c'est un `addu`, le codage de l'instruction suit donc le format R. Il y a donc 2 opérandes sources : les registres R3 et R2. Il y a un opérande destination : le registre R4.

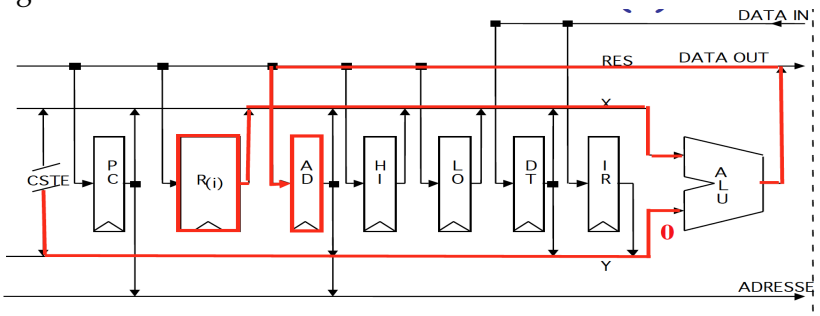
Ce décodage permet de déterminer les mouvements de données à réaliser sur le chemin de données, quelle commande donner à l'ALU, ...

3 - Exécuter l'instruction Il s'agit d'effectuer l'addition demandée par l'instruction. Comme le banc de registres ne peut écrire que sur le bus X, on ne peut réaliser une opération arithmétique et logique avec deux opérandes sources de type registre en une seule fois. Il faut mettre dans le registre AD (qui peut écrire sur le bus Y) le contenu d'une opérande puis réaliser l'opération voulue. Soit les deux actions suivantes :

1) AD ← 0 + $2

2) $4 ← AD + $3

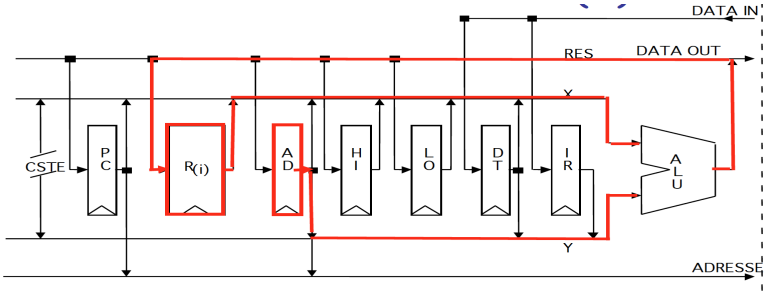
Pour l'action 1), le contenu de R2 est mis sur X et 0 est mis sur Y. Il y a sélection de l'addition dans l'ALU, et autorisation d'écriture de la sortie d'ALU (RES) dans AD. Les mouvements de données engendrés sont représentés en rouge dans la figure ci-dessous :



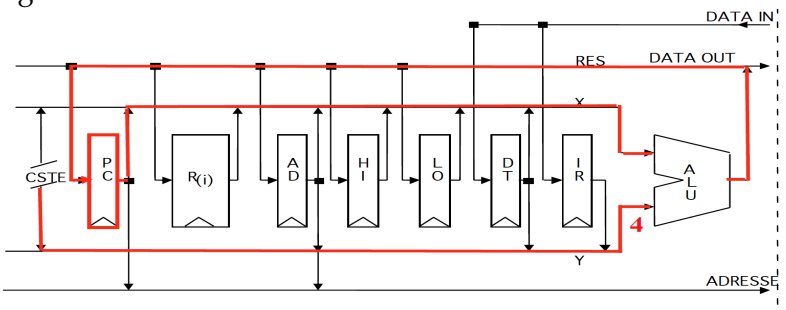
Pour l'action 2), le contenu de R3 est placé sur X et celui de AD sur Y. Il y a encore la sélection de l'addition dans l'ALU. L'autorisation d'écriture du registre R4 du banc de registre est mise à 1 pour enregistrer la sortie d'ALU dans R4.

Les mouvements de données engendrés sont représentés en rouge dans la figure ci-dessous :

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International



4 - Déterminer l'adresse de l'instruction suivante. Le modèle d'exécution implicite est séquentiel. L'adresse de l'instruction suivante est simplement l'adresse de l'instruction courante plus la taille en octet d'une instruction soit :
PC ← PC + 4.
Lorsque l'instruction spécifie l'adresse de l'instruction suivante, alors PC prend une valeur dont le calcul dépend de l'instruction (adresse relative à PC ou adresse absolue). Les mouvements de données engendrés sont représentés en rouge dans la figure ci-dessous :



Notion de transfert On appelle *transfert* un mouvement de données sur le chemin de données réalisable **en une seule fois**. L'exécution d'une instruction peut être écrite sous forme de transferts dans le chemin de données.
Les notations utilisées pour représenter les transferts de données dans le chemin de données sont les suivantes :

r	nom du registre
r[n]	registre n du banc de registres
=	test d'égalité
+	addition entière en complément à deux
-	soustraction entière en complément à deux
and	opérateur et bit à bit
or	opérateur ou bit à bit
nor	opérateur non-ou bit à bit
xor	opérateur ou exclusif bit à bit
m[a]	contenu de la case mémoire d'adresse a
←	assignation
||	concaténation de chaînes de bits
b^n	réplication du bit b dans une chaîne de n bits
$x_{p...q}$	sélection des bits p à q de la chaîne de bits x

L'exemple précédent s'écrit comme suit sous forme de transfert :

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

IR ← m[PC] ; lecture de l'instruction
AD ← r[2] (+ 0) ; réalisation de l'instruction
r[4] ← r[3] + AD
PC ← PC + 4 ; calcul de l'adresse de l'instruction suivante

Remarque : lors du lancement d'un programme, PC contient l'adresse de la première instruction du programme.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Chapitre 7

Uniformité et structuration de la mémoire

7.1 La mémoire est uniforme.

La mémoire est modélisée par un tableau de mots de 4 octets adressable en octets. Elle stocke sous forme de suite d'octets tout ce qui est nécessaire à l'exécution d'un programme.

Cette uniformité est ce qui confère à l'ordinateur (**machine de Von Neumann**) son universalité : les traitements exécutés par le processeur sont précablés mais l'agement de traitements est infini.

Les données et les instructions sont stockées sur le même support : ce qui est interprété à un instant donné comme une donnée peut être interprété comme une instruction en langage machine plus tard. Un programme peut être utilisé pour écrire de nouveau programme.

Exemple :

1. Rédaction de programme source : la donnée est le programme écrit en texte, donc codé en ASCII, et le programme est l'éditeur de texte.
2. Compilation d'un programme source en un binaire exécutable : la donnée d'entrée est le programme source, un texte codé en ASCII représentant le programme ; le programme est le compilateur ; la sortie est une donnée : le programme binaire exécutable.
3. Exécution d'un programme binaire : pour exécuter un programme binaire, il faut d'abord le charger en mémoire. C'est le travaille du 'loadeur' : le programme est le loader, la donnée le programme binaire. Une fois chargée, le loader donne la main au programme binaire exécutable : le programme est l'exécutable, les données celles du programme correspondant.

7.2 La mémoire est structurée.

La raison de cette structuration est qu'un programme n'est pas le seul objet présent en mémoire, il y a les données et programmes de l'OS, éventuellement d'autres utilisateurs de la machine. Il faut pouvoir protéger :

— les informations du système vis-à-vis des programmes utilisateurs
— les différents types d'informations au sein d'un même programme

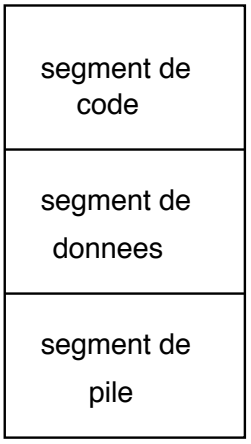


FIGURE 7.1 – Représentation des 3 segments mémoire associés à tout programme lors de son exécution

— les programmes vis à vis d'autres programmes

Pour protéger les informations système vis-à-vis des programmes utilisateur, la mémoire est découpée en deux zones distinctes :

1. celle réservée au système (OS), on ne peut y accéder qu'avec des instructions spéciales utilisables seulement en mode superviseur/super utilisateur,
2. celle réservée aux utilisateurs dont l'accès n'est pas restreint.

En MIPS, ces deux zones sont distinguées par leur adresse :

1. zone users : 0x00000000 $\rightarrow$ 0x7FFFFFFF
2. zone system : 0x80000000 $\rightarrow$ 0xFFFFFFFF

Pour protéger les différents types d'informations au sein d'un même programme, celles-ci ne sont pas rangées au même endroit en mémoire. Les différents types d'informations d'un programme sont regroupés en zones distinctes appelées segments. Un segment est un espace d'adressage contigü muni d'une taille maximale dans lequel sont stockées les informations de même nature.

Un programme possède au moins 3 segments comme illustré sur la figure 7.1 :

1. segment de code : il contient les instructions codées en binaire selon l'ISA du processeur
2. segment de données : il contient les données codées en binaire selon leur type et correspond aux variables globales
3. segment de pile : c'est une mémoire temporaire allouée à la demande au cours de l'exécution pour stocker les variables locales, les contextes d'exécution des sous-programmes et leurs paramètres d'appel.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Chapitre 8

Programmation assembleur

Contents

8.1 Niveaux de programmation et traduction de programmes

Il existe différents niveaux de programmation qui correspondent aux différents niveaux d'abstraction des traitements à exécuter.

Programme de haut niveau :
- Instruction du langage ∀ ISA cible
- Definition d'objets structurés
- Définition de variables nommées utilisables dans les instructions
- Structuration des traitements
- Gestion d'erreurs

Langage d'assemblage/assembleur :
- Suite d'instructions assembleur
- Présence d'étiquettes pour désigner les adresses (données ou instructions)

Programme binaire :
- Suite d'instructions en langage machine

Traduction programme haut niveau → programme assembleur La traduction entre programme de haut niveau et langage d'assemblage est réalisée par un compilateur. Les instructions de haut niveau sont traduites en une suite d'instructions assembleur. Les registres sont utilisés pour réaliser les traitements.

Traduction programme assembleur → programme binaire exécutable La traduction assembleur-binaire ou l'assemblage réalise la traduction binaire des instructions. Elle nécessite 2 phases :

1. Traduction en binaire et assignation des adresses d'implantation
2. Résolution des adresses : les étiquettes dans les instructions sont converties en adresses absolues ou relatives (exemple : le champ des sauts conditionnels/inconditionnels).

Exemples

ASM		Binaire		Desassemblé
Instruction assembleur + présence d'étiquettes		Instructions converties en langage machine		Instruction asm avec valeur pour les adresses relatives et absolues. Plus d'étiquettes.

`beq $5, $0, fin`
`etiq: andi $4, $4, 0x0FF0`
`addi $4, $4, -1`
`j etiq`
`fin: add $8, $4, $6`
	→	`0x10A00003`
		`0x30840FF0`
		`0x2084FFFF`
		`0x08100001`
		`0x00864020`
			→	`beq $5, $0, 3`
				`andi $4, $4, 4080`
				`addi $4, $4, -1`
				`j 4194308`
				`add $8, $4, $6`

Programme C		ASM		Binaire

`{`
`...`
`int a = 2;`
`int b = 3;`
`int c = (a + b) * 2;`
`...`
`}`
	→	`...`
		`ori $4, $0, 2`
		`ori $5, $0, 3`
		`add $6, $5, $4`
		`sll $6, $6, 1`
		`...`
			→	`0x34040002`
				`0x34050003`
				`0x00A43020`
				`0x00063040`

En assembleur, on utilise des registres pour stocker les valeurs des variables locales (simples) et les résultats de calculs intermédiaires.

8.2 Programmation assembleur et structuration d'un fichier asm MIPS

On peut écrire des programmes directement en assembleur. Il faut alors décrire les traitements à réaliser avec les instructions de jeu d'instructions du processeur cible, il faut aussi déclarer/allouer les données.

8.2.1 Structuration d'un programme assembleur MIPS

En MIPS, tout programme assembleur est constitué de 2 sections, la section de données et la section de code.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

La directive `.text` désigne la section de code, et spécifie que ce qui suite sera implanté dans le segment de code. Les instructions doivent se trouver dans cette section : dans un programme assembleur elles doivent apparaître après cette directive.
La directive `.data` désigne le section de données, elle spécifie que ce qui suit correspond aux données globales, celles-ci seront implantée dans le segment de données. Les données globales doivent être déclarées/allouées dans cette section du programme. Ils existent plusieurs directives permettant d'allouer des emplacements mémoires pour des variables et de les initialiser potentiellement. Les plus importantes sont décrites plus bas.

Il est possible d'entrelacer des sections de code et de données dans un fichier assembleur, et lors de la création du fichier exécutable, elles sont assemblées entre elles) . Toutefois, dans ce cours, pour des raisons pédagogiques nous nous efforcerons d'écrire des fichiers assembleur ne comportant qu'une seule section de données au début du fichier et une seule section de code ensuite.

Etiquettes Une étiquette à la forme **etiq :**, son nom est `etiq`. Sa sémantique est "adresse de ce qui suit". Elle peut apparaitre dans la section de code ou de données. Son nom peut être utilisé dans les instructions : les étiquettes permettent de désigner des adresses (de données ou d'instruction) dans le code assembleur.

8.2.2 Assemblage et chargement d'un programme en mémoire

Une fois un programme assembleur écrit, il faut l'assembler afin de créer un programme binaire exécutable. Pour exécuter un programme, il faut le lancer. Lors de son lancement, le programme est chargé en mémoire (par un loader). La section de code est rangée dans le segment de code et la section de données est rangée dans le segment de données. La première instruction de la section .text est implantée à la premiere adresse du segment de code et les suivantes sont allouées consécutivement en mémoire dans le segment de code, dans leur ordre d'apparition dans le fichier assembleur. La première donnée de la section .data est implantée à la première adresse du segment de données et les données déclarées/allouées ensuite sont implantée en séquence dans le segment de données. L'adresse d'implantation des segments de code et de données peuvent varier d'une plateforme cible à un autre, et sont configurables. Nous considèrerons que l'adresse d'implantation pour le segment de code est 0x0040000 et qu'elle vaut 0x10010000 pour le segment de données (valeurs par défaut considéré par Mars le logiciel utilisé pour les travaux pratiques).

La figure 8.1 présente la structure d'un fichier assembleur et l'implantation des sections dans les segments mémoire lors de l'exécution.

8.2.3 Exécution et terminaison d'un programme assembleur

La première adresse de la section de code correspond au point d'entrée du programme. Lors du lancement d'un programme, l'adresse du point d'entrée du programme est mise par le loader dans le registre PC, la premiere instruction est alors lue en mémoire, puis exécutée. L'exécution continue ensuite séquentiellement (avec parfois des sauts si exécution d'une instruction spécifiant l'adresse de la prochaine instruction qui n'est pas celle qui suit dans le programme source/asm). Le programme s'arrête avec un appel système demandant la terminaison du programme.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

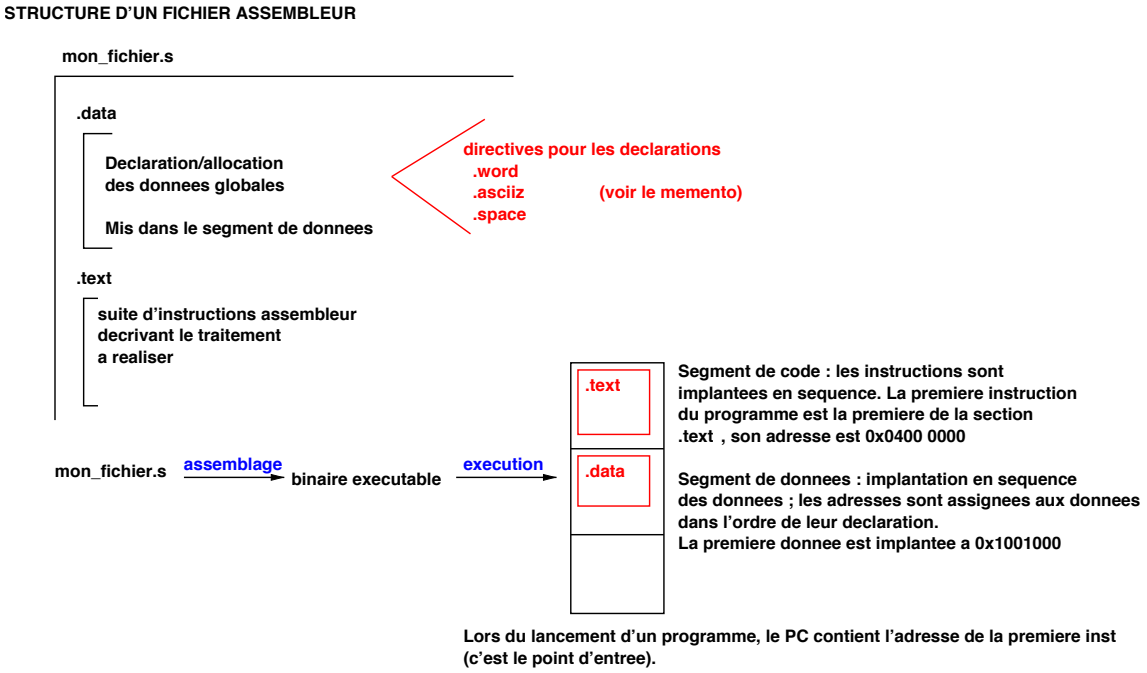


FIGURE 8.1 – Structure d'un fichier assembleur et l'implantation des sections dans les segments mémoire lors de l'exécution.

Appel système Un appel système, comme déjà mentionné, est une demande de service fourni par le système. En MIPS, ils ont des numéros ; pour les utiliser il faut mettre le numéro de l'appel système voulu dans le registre $2 avant d'exécuter l'instruction `syscall`. Voici quelques appels système et leur numéro.

— Terminaison d'un programme : numéro 10.
— Affichage d'un entier : numéro 1 ; il faut mettre l'entier à afficher dans le registre $4 avant l'appel.
— Affichage d'une chaine de caractères : numéro 4 ; l faut mettre dans le registre $4 l'adresse du premier caractère de la chaîne, celle-ci doit se terminer par le caractère de fin de chaîne.

8.2.4 Allocation et initialisation de mémoire de données

Les directives d'allocation d'emplacement mémoire de données sont les suivantes :

— `.word VAL1,[VAL2, VAL3,...]` : alloue un mot mémoire (4 octets) initialisé avec la valeur `VAL1` (donnée en décimal ou hexadecimal si préfixé par `0x`), ou plusieurs mots consécutifs initialisés avec les valeurs `VAL1`, `VAL2`, `VAL3`,...
— `.half VAL1,[VAL2, VAL3,...]` : alloue un demi-mot mémoire (2 octets) initialisé avec la valeur `VAL1` (donnée en décimal ou hexadecimal si préfixé par `0x`), ou plusieurs demi-mots consécutifs initialisés avec les valeurs `VAL1`, `VAL2`, `VAL3`,...
— `.byte VAL1,[VAL2, VAL3,...]` : alloue un octet mémoire (4 octets) initialisé avec la valeur `VAL1` (donnée en décimal ou hexadecimal si préfixé par `0x`), ou plusieurs octets consécutifs initialisés avec les valeurs `VAL1`, `VAL2`, `VAL3`,...
— `.space NB` alloue `NB` octets (non initialisés)
— `.asciiz CH` alloue le nombre d'octets nécessaire pour l'implantation consécutive

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

de chaque caractère de la chaîne de caractères `CH` (doit être donnée entre "") en incluant le caractère de fin de chaîne (dont la valeur est 0x00).

— `.ascii CH` alloue le nombre d'octets nécessaire pour l'implantation consécutive de chaque caractère de la chaîne de caractères `CH` (doit être donnée entre "").

Voici un exemple de section de données préambule d'un fichier assembleur :

```
.data
n:    .word 10            # entier sur 4 octets - adresse 0x10010000
m:    .word 0xFFFF        # entier sur 4 octets
                          # adresse 0x10010000 + 4 = 0x10010004
p:    .byte 255, 0x01     # 2 octets initialisés à 255 et 0x01
                          # l'adresse du premier vaut  0x10010004 + 1 = 0x10010005
ch1: .asciiz "coucou"     # allocation et initialisation d'une chaine de caractères.
                          # l'adresse de l'étiquette ch1 est 0x10010005 + 2 = 0x10010007

ch2: .space 12            # allocation d'un espace de 12 octets
                          # d'adresse 0x10010007 (adresse ch1) + taille (ch1)
                          # soit 0x10010007 + 7 = 0x1001000E
.text
...
```

8.3 Exemples de programme assembleur

8.3.1 Programme qui affiche la valeur 2

```
.data
.text
        ori $4, $0, 2    # on met l'entier à afficher dans $4
        ori $2, $0, 1    # numéro de l'appel systeme 1 (affichage entier) dans $2
        syscall          # on effectue la demande d'appel systeme
        ori $2, $0, 10   # numéro de l'appel systeme 10 (fin de prog) dans $2
        syscall          # on effectue la demande d'appel systeme
```

8.3.2 Programme qui affiche "Hello World"

```
.data
ch:     .asciiz "Hello World!"  #@0x1001000
.text
        lui $4, 0x1001
        ori $4, $4, 0X0000   # on met l'adresse ch dans $4
        ori $2, $0, 4        # numero appel systeme 4 dans $2
        syscall              # affichage de la chaine par appel systeme
        ori $2, $0, 10       # terminaison
        syscall
```

8.3.3 Petit programme traduit d'un programme C

```
/*Variables globales*/
int n1 = 0x10;
int n2 = -10;
int n3 = 128;

/*programme principal */
int main(){
  n1 = n1 + 1;
  n2 = n2 - 1;
  n3 = n1 + n2 + n3
  printf("\%d\n", n3); /*affichage de la nouvelle valeur de n3 */
  return 0;
}
```

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

L'instruction `n1 = n1 + 1` signifie : la variable `n1` recoit une nouvelle valeur, la sienne additionnée de 1. Comme la variable `n1` est une variable globale, cela nécessite de lire la valeur de la variable `n1` en mémoire, de lui ajouter 1 et de ranger en mémoire sa nouvelle valeur.

```
.data
n1:     .word 0x10              #@ = 0x10010000
n2:     .word -10               #@ = 0x10010004
n3:     .word 128               #@ = 0x10010008

.text

#lire n1, lui ajouter 1 et ranger la valeur
        lui $4, 0x1001
        ori $4, $4, 0x0000   #$4 contient l'adresse de n1
        lw  $5, 0($4)        #$5 contient la valeur de n1
        addi $5, $5, 1       #n1 + 1
        sw $5, 0($4)         #ecriture en memoire de la nouvelle valeur

#lire n2, lui ajouter -1 et ranger la valeur
        lui $3, 0x1001
        ori $3, $3, 0x0004   #$3 contient l'adresse de n2
        lw  $6, 0($3)        #$6 contient la valeur de n2
        addi $6, $6, -1      #n2 - 1
        sw $6, 0($3)         #ecriture en memoire de la nouvelle valeur
#lire n3,
        lw  $7, 4($3)        #$7 contient la valeur de n3

#faire la somme, $8 est un temporaire pour contenir la valeur n1 + n2 + n3
        add $8, $5, $6        # n1 + n2
        add $8, $8, $7        # n1 + n2 + n3
        sw $8, 4($3)          # n3 = n1 + n2 + n3
#affichage
        ori $4, $8, 0
        ori $2, $0, 1
        syscall
#fin du programme
        ori $2, $0, 10
        syscall
```

Règle sur les variables globales Lorsqu'une variable globale apparaît dans une expression C (par exemple la variable `x` dans `x + 2`), il faut lire sa valeur en mémoire pour la mettre dans un registre, à moins que cette valeur ne soit déjà dans un registre ; cela nécessite de

— mettre l'adresse de la variable (à calculer de tête par rapport au début de la section de données) dans un registre (avec les instructions `lui` puis `ori`),
— faire une lecture mémoire (avec une instruction de lecture `lw`/`lh`/`lb`...).

Lorsqu'une variable globale est à gauche d'un signe d'affectation (par exemple la variable `x` dans la variable `x = expr`), il faut écrire sa valeur en mémoire. Cela nécessite de :

— mettre l'adresse de la variable (à calculer de tête par rapport au début de la section de données) dans un registre (avec les instructions lui/ori),
— faire une écriture mémoire (avec une instruction d'écriture `sw`/`sh`/`sb`...).

Remarque : on ne doit pas utiliser les registres $29 et $31 dans les programmes. $29 correspond au pointeur de pile et $31 contient l'adresse de retour des sous-programmes (les appels de fonction et convention d'appels ne sont pas traités dans ces notes de cours).

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

8.4 Exercices

Objectif(s)

- ✯ Prise en main de Mars (lancement, interface, actions possibles, debug) ;
- ✯ Écriture et exécution de programmes assembleur.
- ✯ Familiarisation avec les appels système (`syscall`) de lecture et d'affichage d'une chaîne de caractères ou d'un nombre entier.
- ✯ Familiarisation avec les déclarations de variables globales et la réservation d'espace mémoire pour les données
- ✯ Écriture de programmes assembleur comportant des instructions de lecture et d'écriture en mémoire

Exercice 27 – Prise en main de MARS, visualisation du codage d'une instruction

Lancez Mars (commande donnée par vos enseignants ou sur le site web).
La figure 8.2 montre l'interface graphique de MARS.

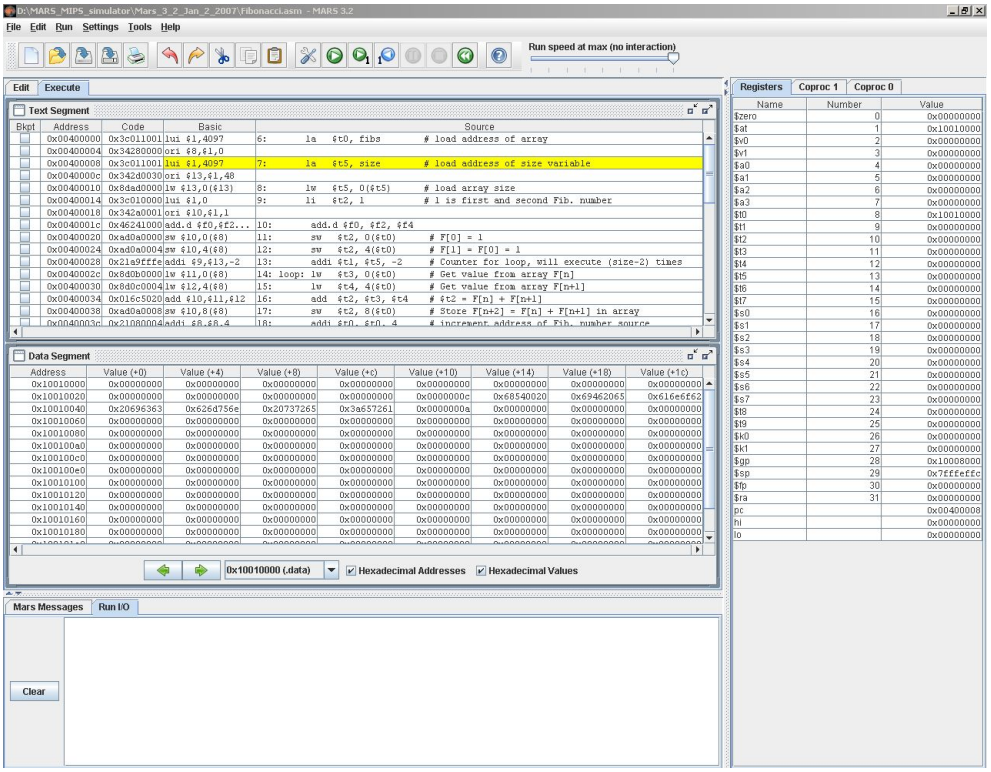


FIGURE 8.2 – MARS

Sur la droite de l'interface, vous pouvez apercevoir le contenu des registres généraux (`$0,$1,..., hi, lo`), ainsi que des registres spéciaux comme le compteur de programme (pc) etc.
Au centre de l'interface on peut voir deux onglets : **Edit** et **Execute**.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

— L'onglet **Edit** permet d'éditer et de modifier un code assembleur. C'est là qu'apparait le code assembleur après chargement d'un fichier.
— L'onglet **Execute** contient 2 parties (cliquez sur l'onglet pour voir) : **Text Segment** et **Data Segment**.
  — **Text Segment** contient le code assemblé : codage des instructions, adresses où elles sont implantées.
  — **Data Segment** contient les données (variables globales, pile, ...).

Au bas de l'interface, vous avez les onglets **Mars Messages** et **Run I/O**. Les messages d'erreur du simulateur seront affichés dans **Mars Messages**. Les sorties du simulateur, en particulier les affichages effectués par le programme, se feront dans **Run I/O**.

En haut de l'interface, vous avez les différentes actions possibles dans les menus **File, Edit, ...**. Des raccourcis sont disponibles sous la forme de petites icônes juste en dessous des menus. A droite de ces icônes se trouve un curseur réglable qui se nomme **Run speed** qui permet de régler la vitesse d'exécution des instructions.

IMPORTANT : Dans le menu Settings :

1. décochez l'utilisation des pseudo-instructions (*"Permit extended (pseudo) instructions and formats"*) : vous devrez toujours utiliser uniquement les instructions de base dans votre programme.
2. cochez l'affichage des valeurs et des adresses en hexadécimal (*"Values (resp. Addresses) displayed in hexadecimal"*).

Question 1
Où trouver dans le simulateur les instructions, directives et appels système supportés par celui-ci ?

Question 2

1. Dans Mars, ouvrez un nouveau fichier (menu File, New) et écrivez-y l'instruction `addi $12, $18, 33.`
2. La saisie de cette instruction modifie-t-elle les informations se trouvant dans l'onglet Execute ?
3. Sauvegardez votre fichier sous le nom test.s. Cette action modifie-t-elle les informations se trouvant dans l'onglet Execute ?
4. Assemblez votre programme avec la commande Assemble du menu Run. Que se passe-t-il ? Que lit-on dans **Mars Messages** ?
5. Où pouvez vous trouver le codage hexadécimal de cette instruction ? Quel est-il ? À quelle adresse est implantée cette instruction ?

Question 3

1. Sur votre feuille, donnez le codage en binaire de l'instruction `addi $12, $18, 33`. Inversez (i.e., prenez a négation de) la valeur du 29ème bit (les bits sont numérotés de droite à gauche, le bit le plus à droite est le bit 0).
2. À l'aide de votre mémento, déterminez l'instruction correspondant à ce codage binaire. Donnez le codage hexadécimal correspondant.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

3. Saisissez cette instruction après l'instruction précédente. Vérifiez que votre réponse à la question précédente est correcte en regardant, dans l'onglet `Execute`, la valeur hexadécimale associée à l'instruction que vous venez de saisir.
4. Exécutez cette série d'instructions. Que remarquez-vous ?

Question 4
Modifiez l'instruction `addu $0, $18, $12` afin de générer une erreur de syntaxe. Vous pouvez par exemple mettre 12 au lieu de $12. Que se passe-t-il ?

Exercice 28 – Exécution de programme

Le but de cet exercice est d'exécuter des programmes assembleur avec Mars et d'apprendre comment manipuler Mars pour lancer un programme, l'exécuter pas à pas, l'exécuter sans arrêt, revenir en arrière dans l'exécution.

Question 1

1. Dans votre éditeur de texte préféré, écrivez un programme assembleur qui met dans le registre $16 la valeur *décimale* 137, affiche cette valeur avec l'appel système d'affichage d'un entier (affichage en décimal) avant de quitter. Sauvegardez le fichier avec l'extension `.s` (extension classique des fichiers assembleur). Aidez-vous du mémento pour savoir comment utiliser l'appel système permettant d'afficher un entier.
2. Chargez le programme dans Mars et assemblez-le. Si la fenêtre Text Segment ne s'affiche pas, c'est qu'il y a une erreur de syntaxe. Corrigez l'erreur et recommencez l'opération de chargement et de correction de syntaxe jusqu'à ce que votre programme puisse être assemblé.
3. Exécutez le programme pas à pas en observant le contenu des différents registres après l'exécution de chaque instruction.
4. Réinitialisez le simulateur (double flèche vers la gauche).
5. Ré-exécutez le programme mais sans arrêt cette fois.
6. Réinitialisez le simulateur (double flèche vers la gauche).
7. Diminuez la vitesse d'exécution et ré-exécutez le programme. Au cours du programme faites pause après le premier syscall. Revenez en arrière d'une instruction et modifiez la valeur du registre $16 (dans la zone d'édition des registres, double-cliquez sur le registre $16 et entrez la valeur 1). Relancez le programme. Que se passe-t-il ?

Question 2
Reprenez le programme de la question précédente en mettant cette fois dans $16 la valeur $65537 = 2^{16} + 1$. Rappelez, avant de modifier votre programme, le nombre de bits affectés au stockage d'une valeur immédiate dans le format I (instructions avec un opérande de type immédiat) du jeu d'instructions MIPS. Exécutez votre programme et vérifiez le contenu de $16.

Exercice 29 – Instructions de multiplication et de division

Les instructions de multiplication et division sont un peu particulières. Elles ont deux

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

opérandes sources mais pas d'opérande explicite de destination. La raison est que la multiplication de deux valeurs sur 32 bits donne un résultat sur 64 bits (il faut donc 2 registres) et la division a deux résultats : le quotient et le reste. C'est pourquoi ces instructions n'ont pas d'opérande de destination explicite, elles produisent leur résultat dans deux registres spéciaux, les registres `lo` et `hi`.

L'instruction `div ri, rj` effectue la division de `ri` par `rj`. Elle met le quotient dans le registre `lo` et le reste dans le registre `hi`.

L'instruction `mult ri, rj` effectue le produit de `ri` par `rj`. Elle met les 32 bits de poids fort du résultat (sur 64 bits) dans le registre `hi` (high) et les 32 bits de poids faible dans le registre `lo` (low).

Il est possible de copier le contenu du registre spécial `lo` dans un registre général `$r` avec l'instruction `mflo $r`. De même, l'instruction `mfhi $r` permet de copier le contenu du registre `hi` dans le registre `$r`. Ces deux instructions sont les deux seules instructions permettant de récupérer les valeurs contenues dans les registres `lo` et `hi`, c'est-à-dire le résultat des instructions `mult` et `div`.

Question 1

1. Écrivez un programme assembleur qui charge les valeurs (décimales) 84 et 10 dans les registres $9 et $10, puis effectue la division de 84 par 10, récupère le quotient et le reste dans les registres $11 et $12 et les affiche.
2. Assemblez le programme et exécutez le pas à pas en regardant le contenu des différents registres tout au long de l'exécution, notamment les registres `lo` et `hi`.
3. Complétez le programme pour qu'il reconstruise la valeur 84 dans le registre $13 à partir du quotient et du reste. Le programme doit afficher le résultat recalculé.
4. Assemblez le programme et exécutez-le pas à pas en regardant le contenu des différents registres tout au long de l'exécution.

Question 2

1. Modifiez votre programme pour que les deux entiers de départ soient lus au clavier.
2. Testez le programme pour différentes valeurs et vérifiez qu'il est correct (la valeur reconstruite est bien égale à la valeur initiale).

Exercice 30

Rangement mémoire des données

Question 1

Déclarez 4 octets $o1, o2, o3$ et $o4$ valant respectivement 1, 2, 3 et 4 puis un mot $m1$ valant `0xAABBCCDD`. Assemblez ce programme et regardez comment ces variables sont implantées en mémoire.

Question 2

Donnez les adresses correspondant aux étiquettes du programme $o1, o2, o3$, $o4$ et $m1$.

Question 3

Dans l'onglet *Settings*, activez **Show Labels Window**. Qu'observez-vous ? Vérifiez ce que vous avez répondu à la question précédente.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Exercice 31 – Programme avec accès mémoire

Question 1
Écrivez un programme assembleur comportant l'allocation aux adresses v1 et v2 de deux mots mémoire initialisés respectivement à -1 et 0xFF. Le programme doit charger les valeurs contenues aux adresses v1 et v2 dans les registres $5 et $6, puis afficher les deux valeurs. Assemblez et testez votre programme.

Question 2
Modifiez le programme pour qu'il ajoute 1 à v1 et à v2 puis range les nouvelles valeurs en mémoire. Assemblez et exécutez le programme. Vérifiez le contenu de la mémoire à la fin de l'exécution (l'exécution a-t-elle bien modifié les valeurs implantées en mémoire aux adresses v1 et v2 ?).

Question 3
Écrivez un programme qui déclare un octet de valeur `0xFF`, qui charge cet octet dans le registre $7 en considérant sa valeur comme signée et qui charge cet octet dans $8 en considérant sa valeur comme non signée. Le programme affiche ensuite le contenu des deux registres.

Exécutez le programme et regardez le contenu des registres $7 et $8 après chargement. Quelles sont les valeurs affichées ? Expliquez.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Chapitre 9

Rupture de séquence : les instructions de saut

Contents

9.1 La rupture de séquence est nécessaire

Ci-dessous est donné un exemple de programme binaire. Initialement, lors du lancement de ce programme PC = 0x00400000, puis il croît de 4 en 4 à l'exécution de chaque instruction.

Code assembleur	Adresse mémoire : contenu	A l'exécution
`.text`		
`__start:`		PC ← 0x00400000
`xor $3, $3, $3`	0x00400000 : 0x00631826	
`xor $4, $4, $4`	0x00400004 : 0x00842026	
`addi $3, $3, 1`	0x00400008 : 0x20630001	
`add $4, $4, $3`	0x0040000c : 0x00832020	
`addi $3, $3, 1`	0x00400010 : 0x20630001	
`add $4, $4, $3`	0x00400014 : 0x00832020	
`addi $3, $3, 1`	0x00400018 : 0x20630001	
`add $4, $4, $3`	0x0040001c : 0x00832020	
`ori $2, $0, 10`	0x00400020 : 0x3402000a	
`syscall`	0x00400024 : 0x0000000c	

Son exécution pas à pas permet de voir que toutes les instructions sont exécutées les unes après les autres. L'objectif ici est de mettre en évidence qu'implicitement, la prochaine instruction exécutée est celle qui est implantée juste après l'instruction courante. Cela pose un problème lorsque l'on veut pouvoir réexécuter des instructions (taille du code, on ne sait pas forcément a priori le nombre de ré-exécution à appliquer (dépend des données)).
→ Il y a besoin d'instructions permettant d'agir directement sur la valeur de PC.

pliage de code : ré-exécuter des instructions (réduit la place occupée par le programme)

Code assembleur	Adresse mémoire : contenu	A l'exécution
`.text`		
`__start:`		PC ← 0x00400000
`xor $3, $3, $3`	0x00400000 : 0x00631826	
`xor $4, $4, $4`	0x00400004 : 0x00842026	
`debut:`		
`addi $3, $3, 1`	0x00400008 : 0x20630001	
`add $4, $4, $3`	0x0040000c : 0x00832020	
`j debut`	0x00400010 : 0x08100002	
`ori $2, $0, 10`	0x00400014 : 0x3402000a	
`syscall`	0x00400018 : 0x0000000c	

L'instruction `j debut` modifie explicitement la valeur de PC, en lui attribuant la valeur

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

0x00400008 alors que le modèle implicite aurait affecté à PC la valeur 0x00400014. C'est un **saut inconditionnel** (qui modifie systématiquement la valeur de PC en lui attribuant la valeur spécifiée en argument, ici : 0x00400008).
Cela pose un problème : on a créé une boucle infinie. Comment peut on sortir après trois additions exactement ? Il faut une instruction plus souple que le `j debut`, qui place dans PC la valeur 0x00400008 les trois premières fois qu'elle est exécutée, et place dans PC la valeur 0x00400014 lors de sa quatrième exécution. C'est un **saut conditionnel**, qui n'est réalisé que si une condition est vraie. La condition à élaborer ici doit être vraie lors de ses trois premières évaluations et fausse à la quatrième.

Les conditions de saut portent sur des tests d'égalité ou d'inégalité entre deux valeurs. Ces deux valeurs sont rangées dans des registres. Il est possible de tester l'égalité ou l'inégalité à 0 en utilisant le registre zéro (`$0`) qui vaut toujours zéro. Les conditions de saut incluent aussi des comparaisons à la valeur zéro (inférieur ou supérieur, égalité incluse ou exlue). La valeur à comparer doit être dans un registre. Les conditions plus compliquées doivent être évaluées préalablement et rangées dans un registre qui sera opérande de l'instruction de branchement conditionnel.

Par exemple, pour compter 3 itérations.
- Utiliser un registre initialisé à 3, lui soustraire 1 avant chaque branchement conditionnel, se brancher en début de boucle si le registre compteur n'a pas atteint la valeur 0.
- Utiliser un registre initialisé à 0, lui ajouter 1 avant chaque branchement conditionnel, se brancher en début de boucle si le registre compteur n'a pas atteint la valeur 3.

9.2 Instructions de saut conditionnel et inconditionnel

9.2.1 Les sauts inconditionnels

La valeur de PC est systématiquement modifiée. On lui affecte la valeur qui est argument de l'instruction de saut. Il y a deux formes de saut inconditionnel : `j label` ou `jr Ri`. Dans la première, l'adresse est donnée directement par `label`, dans la deuxième le saut est dit *indirect* car l'adresse de la cible du saut est dans le registre `Ri`.

9.2.2 Les sauts conditionnels

Les instructions de branchement conditionnel sont présentées ci-dessous en fonction de la condition de saut.

Tests d'égalité ou différence de contenu de deux registres :

- `beq $2, $3, label`, (branch if equal) si $2=$3 alors branchement à l'étiquette `label`
- `bne $2, $3, label`, (branch in not equal) si $2 $\neq$ $3 alors branchement à l'étiquette `label`

Comparaison du contenu d'un registre par rapport à 0 :

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

— `bgez $3, label`, (branch if greater or equal than zero) si $3≥0 alors branchement à `label`
— `bgtz $3, label`, (branch if greater than zero) si $3>0 alors branchement à l'étiquette `label`
— `blez $3, label`, (branch if less or equal than zero) si $3<= 0 alors branchement à l'étiquette `label`
— `bltz $3, label`, (branch if less than zero) si $3<0 alors branchement à l'étiquette `label`

Comparaison du contenu de deux registres :

— `slt $2, $3, $4`, (set if less than) comparaison signée, met 1 dans $2, si $3 < $4, met 0 sinon
— `sltu $2, $3, $4`, (set if less than) comparaison non signée, met 1 dans $2, si $3 < $4, met 0 sinon

Le saut sera effectué ou non après comparaison à 0 du registre destination de l'instruction set (`slt(i)(u)`) par une instruction de saut conditionnel avec comparaison à 0.
Par exemple, pour sauter à l'instruction d'étiquette « debut » si $var2 < var3$ (contenu de $var2$ dans $2 et de $var3$ dans $3) :

1. déterminer si $2 < $3 : `slt $4, $2, $3` → $4 contiendra 1 si $2 < $3 (interprétés en entiers relatifs)
2. sauter si $4 est à 1 : `bgtz $4, debut` → comparaison de $4 à 0

9.2.3 Détermination de l'adresse de saut

Comment traduire j label en « mettre dans PC l'adresse xxxx » ?
Deux formes d'adressage :
— absolu : label (sur 26 bits – cf format J) est une partie de l'adresse à laquelle il faut se brancher. (ex : `j label` → PC := PC 31 :28 | | IMD * 4)
— relatif : label (sur 16 bits) est un déplacement relatif par rapport à la valeur actuelle de PC (label reste sur seize bits, cf. format I) (ex : `beq Rs, Rt, label` → PC := PC + 4 + (IMD * 4))

Il faut connaître les adresses d'implantation des instructions pour pouvoir déterminer la valeur du champ IMD dans les formats J et I pour les sauts : la compilation nécessite deux passes : l'une implante toutes les instructions, sur 32 bits, en laissant les champs IMD à une valeur qui n'est pas définitive, puis une fois toutes les instructions implantées, les adressages absolus sont résolus (il suffit de placer la partie de l'adresse de l'instruction vers laquelle on saute dans le champ IMD de l'instruction de saut), et les décalages des adressages relatifs sont calculés (en fonction de l'adresse de l'instruction de saut et de l'adresse de l'instruction vers laquelle on saute).

9.3 Réalisation des structures de contrôle des langages de haut niveau en assembleur

Cette partie présente comment implanter en assembleur des structures de contrôle des langages de haut niveau, notamment les alternatives et les boucles. Un code générique

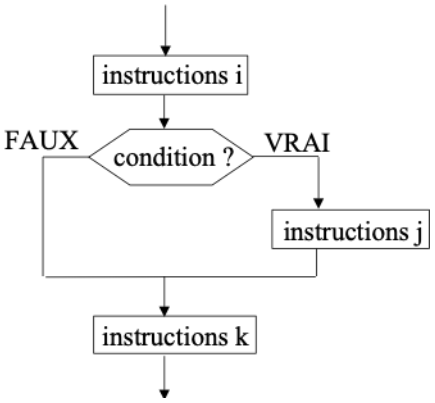
Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

est donné ainsi que des réalisations possibles pour chaque cas. Des exemples à partir de codes simples écrits en C sont aussi donnés.

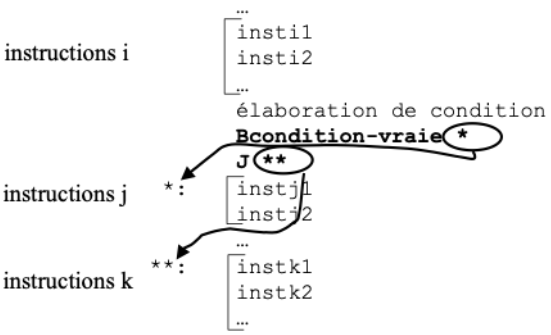
9.3.1 Alternatives

9.3.1.1 Si-alors

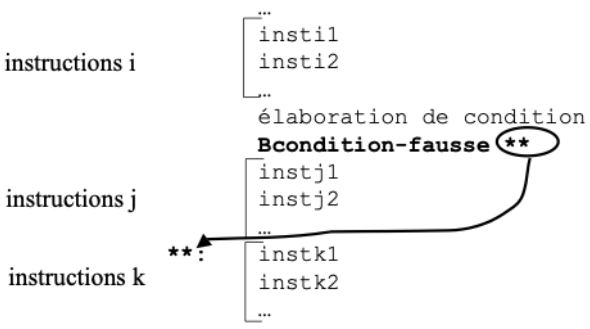
```
…
instructions i
si condition alors
        instructions j
fin-si
instructions k
…
```



Première réalisation possible :



Deuxième réalisation possible :



Exemple On considère l'extrait de code ci-dessous

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

```
...
if (a > b) {
  a = a + 2;
}
```

En considérant que l'adresse de `a` est dans le registre `$3`, la valeur de `a` et de `b` dans respectivement `$4` et `$5`, une réalisation possible est :

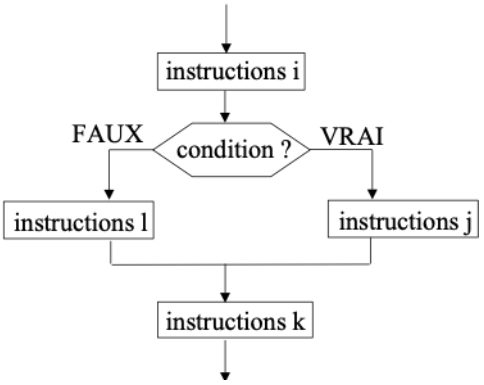
```
.data
   ...
.text
 ...
 #$3 contient l'adresse de a, $4 contient a, $5 contient b
 slt $6, $5, $4     # set $6 si b < a
 beq $6, $0, suite  # saut à suite si b>=a
 add $4, $4, 2      # a + 2
 sw $4, 0($3)       # a = a + 2
suite:
 ...
```

Ou une deuxième moins efficace (un saut de plus) :

```
.data
   ...
.text
 ...
 #$3 contient l'adresse de a, $4 contient a, $5 contient b
 slt $6, $5, $4     # set $6 si b < a
 bne $6, $0, then   # saut à then si b < a
 j suite            # saut à suite
then:
 add $4, $4, 2    # a + 2
 sw $4, 0($3)     # a = a + 2
suite:
 ...
```

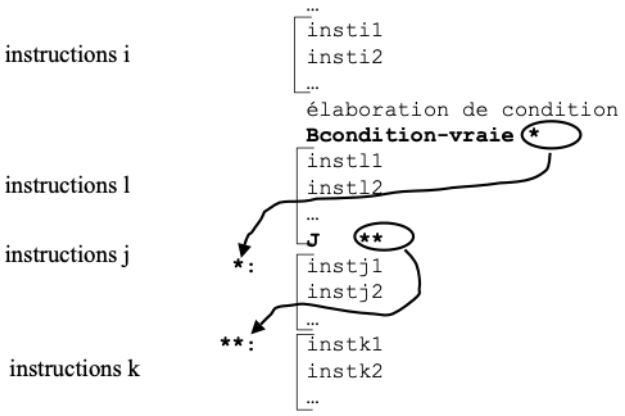
9.3.1.2 Si-alors-sinon

```
…
instruction i
si condition alors
        instruction j
sinon
        instruction l
fin-si
instruction k
…
```



Réalisation possible :

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International



Exemple

On considère l'extrait de code ci-dessous

```
...
if (a > b) {
  a = a + 2;
}
else {
  b = b + 2;
}
...
```

En considérant que l'adresse de `a` et de `a` sont respectivement dans les registre `$3` et `$7`, que la valeur de `a` et de `b` sont respectivement dans `$4` et `$5`, une réalisation possible est :

```
.data
   ...
.text
 ...
 #$3 contient l'adresse de a, $4 contient a, $5 contient b
 slt $6, $5, $4  # set $6 si b < a
 beq $6, $0, else # saut à else si b >= a
 add $4, $4, 2   # a + 2
 sw $4, 0($3)    # a = a + 2
 j suite         # saut à suite pour éviter le cas else
else:
 add $5, $5, 2   # b + 2
 sw $5, 0($7)    # b = b + 2
suite:
 ...
```

Ou une deuxième qui inverse les deux cas :

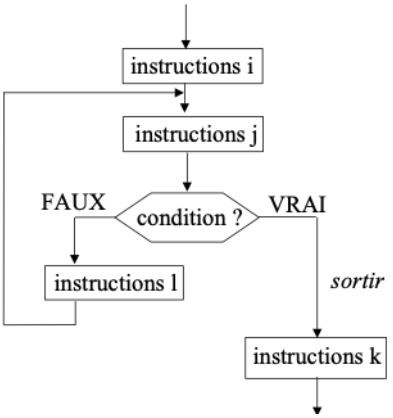
Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

```
.data
   ...
.text
 ...
 #$3 contient l'adresse de a, $4 contient a, $5 contient b
 slt $6, $5, $4  # set $6 si b < a
 bne $6, $0, then # saut à then si b < a
 add $5, $5, 2   # b + 2
 sw $5, 0($7)    # b = b + 2
 j suite         # saut à suite pour éviter le cas then
then:
 add $4, $4, 2   # a + 2
 sw $4, 0($3)    # a = a + 2
suite:
 ...
```

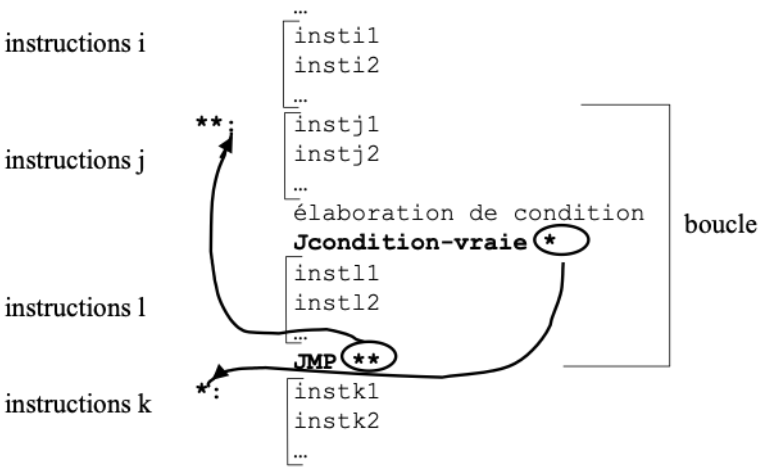
9.3.2 Boucles

Boucle générale

```
…
instructions i
boucle
        instructions j
        si condition sortir
        instructions l
fin-boucle
instructions k
…
```



Réalisation possible :



La boucle Tant Que Tant que `<condition == VRAI>` itérer.
La condition de sortie est inversée par rapport à celle de la boucle générale.
Le bloc d'instructions j est absent.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

La boucle Répéter Jusqu'à Itérer Jusqu'à `<condition == VRAI>`
Le bloc d'instructions l est absent.

La boucle Répéter n fois C'est une boucle TantQue pour laquelle la condition porte sur la valeur d'un compteur qui progresse de 1 à chaque passage dans la boucle, jusqu'à atteindre une borne qui conditionne la sortie.

Le saut calculé Les adresses de sauts peuvent être rangées dans des tables d'indirection, ou bien calculées dynamiquement et rangées dans des registres. Dans le cas du MIPS, cela s'applique aux sauts inconditionnels uniquement (instruction `jr Rd`).

Exemple de boucle tant que / while On considère l'extrait de code ci-dessous. Ce programme ajoute 10 fois la valeur de la variable `a` à la valeur de la variable `n` et range le resultat dans la variable `n`.

```
int n = 10;
int a = 2;

int main(){
  int res = n;
  int i = 0;
  while (i < 10){
    res = res + a;
    i = i + 1;
  }
  n = res;
  printf("%d", res); /* affichage de res */
}
...
```

Une réalisation possible est :

```
.data
n:  .word 10  # allocation et initialisation de n - adresse 0x1001000
a:  .word 2   # allocation et initialisation de a - adresse 0x1001004

.text
 lui $3, 0x1001    # adresse de n dans $3
 lw $4, 0($3)      # lecture valeur de n dans $4 (res)
 lw $5, 4($3)      # lecture valeur de a dans $5

 xor $6, $6, $6    # i = 0

boucle:
 slti $7, $6, 10        # $7 vaut 1 si i < 10
 beq $7, $0, fin_boucle # saut fin de la boucle si $7 vaut 0
 add $4, $4, $5         # res = res + a
 addi $6, $6, 1         # i = i + 1
 j boucle               # retour au début de la boucle

fin_boucle:
 sw $4, 0($3)           # n = res
 ori $2, $0, 1          # $2 = numero appel système affichage entier
 syscall                # demande d'appel système : affichage contenu $4
 ori $2, $0, 10         # $2 = numéro appel système fin de prog
 syscall                # demande d'appel système : terminaison
```

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Exemple de boucle for On considère le code C suivant. La boucle sert à accumuler `n` fois la valeur de la variable `p` dans `res`. La valeur finale devient la valeur de la variable `m` et est affichée.

```
int n = 10;
int p = 3;
int m;

int main(){
  int res = 0;
  int i;
  for (i = 0; i < n; i++){
    res = res + p;
  }
  m = res;
  printf("%d", m); /* affichage de m */
}
...
```

Une réalisation possible est :

```
.data
n:  .word 10  # allocation et initialisation de n - adresse 0x1001000
p:  .word 3   # allocation et initialisation de p - adresse 0x1001004
m:  .space 4  # allocation de m - adresse 0x10010008

.text
 lui $3, 0x1001    # adresse de n dans $3
 lw $4, 0($3)      # lecture valeur de n dans $4
 lw $5, 4($3)      # lecture valeur de p dans $5

 xor $4, $4, $4    # res = 0

 xor $6, $6, $6    # i = 0
boucle_for:
 slt $7, $6, $4             # $7 vaut 1 si i < n
 beq $7, $0, fin_boucle_for # saut fin de la boucle si $7 vaut 0
 add $4, $4, $5             # res = res + p
 addi $6, $6, 1             # i = i + 1
 j boucle_for               # retour au début de la boucle

fin_boucle_for:
 sw $4, 8($3)           # m = res
 ori $2, $0, 1          # $2 = numero appel système affichage entier
 syscall                # demande d'appel système : affichage contenu $4
 ori $2, $0, 10         # $2 = numéro appel système fin de prog
 syscall                # demande d'appel système : terminaison
```

9.4 Exercices

Objectif(s)

- ✯ Écriture de programmes assembleur comportant des boucles et des conditionnelles.
- ✯ Comparaison à une valeur pour les sauts conditionnels

Exercice 32 – Boucle pour avec comparaison à une valeur non nulle

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International

Question 1
Écrivez un programme assembleur correspondant au programme C suivant en mettant des commentaires pour donner la correspondance entre les variables locales du programme C et les registres que vous utilisez pour stocker le contenu de ces variables.

```
int main(){
  int i;
  int somme = 0;

  for (i = 1; i < 11; i++) {
    somme = somme + i;
  }
  printf("%d", somme);
  return 0;
}
```

Question 2
Modifiez le programme précédent pour lire au clavier la valeur minimale de i et maximale (non incluse).
Testez votre programme avec plusieurs valeurs (par exemple pour effectuer la somme des entiers entre -4 et 5, entre 100 et 201, entre 15 et 12).

Exercice 33 – Calcul du PGCD

Question 1
Voici un programme C permettant de calculer le PGCD de deux nombres.

```
int a = 123;
int b = 245;
int pgcd;
 int main(){
  int aa = a;
  int bb = b;
  while (aa != bb) {
    if (aa > bb) {
      aa = aa - bb;
    }
    else {
      bb = bb - aa;
    }
  }
  pgcd = aa;
  printf("%d", pgcd);
  return 0;
}
```

Écrivez le code assembleur équivalent.

Question 2
Modifier le programme précédent pour que les valeurs de a et b soient lues au clavier et rangées en mémoire avant le calcul principal du pgcd entre les deux valeurs des variables.

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International