
Cours d'Architecture des ordinateurs

L2 Informatique

29 août 2023

Séverine Fratani

Table des matières

1	Introduction	5
2	Circuits combinatoires	7
2.1	Algèbre booléenne binaire	7
2.2	Fonction Booléennes	7
2.3	Circuits combinatoires	8
2.3.1	Portes logiques	9
2.3.2	Quelques circuits utiles	10
3	Circuits séquentiels	13
3.1	Les bascules	13
3.1.1	Horloges	13
3.1.2	Modes de synchronisation	14
3.1.3	La bascule D	14
3.1.4	Quelques bascules	15
3.1.5	Forçage des bascules	16
3.2	Quelques circuits séquentiels	16
3.2.1	Les compteurs	16
3.2.2	Les registres	18
3.2.3	Banc de registres	19
3.2.4	Vers la construction d'un calculateur	20
4	Conception et analyse de circuits séquentiels	23
4.1	Automates finis déterministes	24
4.2	Machine de Moore	25
4.2.1	Représentation d'un circuit séquentiel par une machine de Moore	26
4.2.2	De la machine vers le circuit	28
4.3	Machines de Mealy	30
4.4	Comparaison de Modèles	32
4.4.1	Microprogrammation	34
4.4.2	Réalisation microprogrammée d'une machine de Mealy	35
4.4.3	Logique câblée ou micro-programmée ?	37
5	Assembleur	39
5.1	Programmation machine	39
5.1.1	Langage machine et langage d'assemblage	39
5.1.2	Fonctionnement général	39
5.1.3	Les deux principales architectures processeur	40
5.2	Assembleur MIPS	40
5.2.1	Le processeur MIPS R2000	40
5.2.2	Structure d'un programme assembleur MIPS	42
5.2.3	Instructions assembleur MIPS	42
5.2.4	Appel de procédures	46
5.3	Format des instructions MIPS	48

5.3.1	Format d'instructions I	48
5.3.2	Format d'instructions J	49
5.3.3	Format d'instructions R	49
5.3.4	Exemple	49

Chapitre 1

Introduction

Voici la structure classique d'un ordinateur simple :

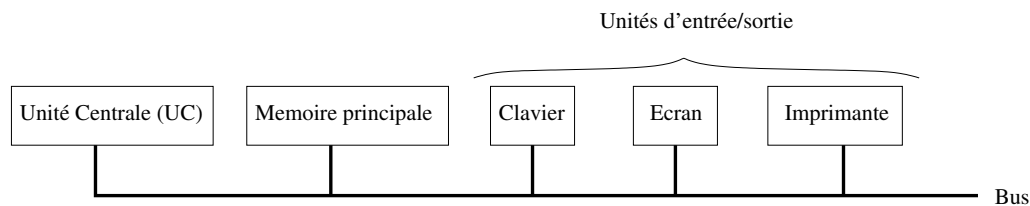


FIGURE 1.1 – Un ordinateur simple

Il possède au moins un processeur (Unité Centrale), au moins une mémoire et des dispositifs d'entrée/sortie. De plus, tous ces composants communiquent entre eux par le biais de bus. On distingue deux sortes de bus, les bus externes reliant par exemple l'unité centrale à la mémoire, et des bus internes à l'unité centrale.

Le processeur est le cerveau de l'ordinateur. Dans la quasi-totalité des ordinateurs modernes, les processeurs respectent l'architecture de Von Neumann, définie par le scientifique du même nom dans les années 40.

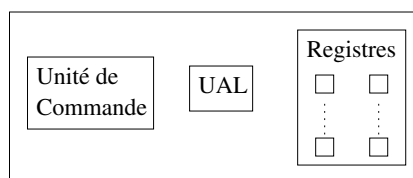


FIGURE 1.2 – L'Unité Centrale

Dans l'architecture de Von Neumann, le processeur est formé

- de l'UAL : l'unité arithmétique et logique qui effectue les opérations de base,
- d'une unité de commande chargée du séquençage des opérations.
- d'une mémoire, organisée en registres, ayant chacun une fonction particulière. Les registres peuvent être lus et écrits très rapidement, c'est la mémoire de travail de l'UC. Un registre particulier appelé PC (Program Counter) pointe sur la prochaine instruction à charger dans l'unité de commande pour exécution.

Le but du cours est de comprendre comment un ordinateur est capable d'exécuter un programme : programmation physique. Nous allons donc nous concentrer sur le fonctionnement de l'unité centrale. Vous savez déjà que les données sont codées en binaire par des signaux électriques. Ces données peuvent être stockées dans les registres, et modifiées par des opérations de bases effectuées par l'UAL. Il s'agit donc de comprendre comment l'unité de commande est capable de séquencer les opérations pour exécuter un programme.

Chapitre 2

Circuits combinatoires

2.1 Algèbre booléenne binaire

L'algèbre booléenne binaire est définie sur l'ensemble $\mathbb{B} = \{0, 1\}$, et composée de 3 opérations suivantes :

- la conjonction (ou produit) : opération binaire qui peut être notée “.”, “et” ou bien “and”.
- la disjonction (ou somme) : opération binaire qui peut être notée “+”, “ou” ou bien “or”.
- la négation (ou complément) : opération unaire qui peut être notée “non” ou “not” ou bien par une barre sur l'opérande.

Ces opérations sont définies de la façon suivante :

la négation :		la conjonction :			la disjonction :		
a	\bar{a}	a	b	$a \cdot b$	a	b	$a + b$
0	1	0	0	0	0	0	0
0	1	0	1	0	0	1	1
1	0	1	0	0	1	0	1
1	0	1	1	1	1	1	1

On peut vérifier facilement que ces opérations satisfont les axiomes de l'algèbre de Boole :

commutativité :	$a \cdot b = b \cdot a$	$a + b = b + a$
associativité :	$a \cdot (b \cdot c) = (a \cdot b) \cdot c$	$a + (b + c) = (a + b) + c$
distributivité :	$a \cdot (b + c) = a \cdot b + a \cdot c$	$a + (b \cdot c) = (a + b) \cdot (a + c)$
éléments neutres :	$1 \cdot a = a \cdot 1 = a$	$0 + a = a + 0 = a$
complément :	$a \cdot \bar{a} = \bar{a} \cdot a = 0$	$a + \bar{a} = \bar{a} + a = 1$

ainsi que les propriétés suivantes (valables pour toute algèbre de Boole) :

élément absorbant :	$a \cdot 0 = 0 \cdot a = 0$	$a + 1 = 1 + a = 1$
absorption :	$a \cdot (a + b) = a$	$a + (a \cdot b) = a$
idempotence :	$a \cdot a = a$	$a + a = a$
involution :	$\bar{\bar{a}} = a$	
lois de De Morgan :	$\overline{a \cdot b} = \bar{a} + \bar{b}$	$\overline{a + b} = \bar{a} \cdot \bar{b}$

2.2 Fonction Booléennes

Une fonction booléenne d'arité n est une fonction qui prend en arguments n booléens et qui retourne un booléen.

Une fonction booléenne (d'arité n) $f : \{0, 1\}^n \rightarrow \{0, 1\}$ peut être donnée

- de manière extensionnelle par sa **table de vérité**

x	y	$m = f(x, y)$
0	0	0
0	1	1
1	0	0
1	1	1

- par **une expression booléenne**, qui est une expression définie avec les constantes et les opérateurs de l'algèbre de Boole et un certain nombre de variables x_1, \dots, x_n . Par exemple :

$$\bar{x}yz + x\bar{y}z + xy\bar{z} + xyz$$

Théorème 2.1 *Toute fonction booléenne peut être représentée comme une expression booléenne.*

PREUVE. Soit f une fonction booléenne ayant pour variables x_1, \dots, x_n et représentée par sa table de vérité.

1. Supposons que f vaille 1 pour k n -uplets d'entrées. (et donc 0 pour les $2^n - k$ autres). Comme $0 + 1 = 1$ on peut écrire f comme $f_1 + \dots + f_k$ où $f_i : \{0, 1\}^n \rightarrow \{0, 1\}$ n'est égale à 1 que pour un seul n -uplet d'entrées.
2. Maintenant pour chaque f_i , on note b_1, \dots, b_n le n -uplet d'entrées pour lequel f_i vaut 1. Alors $f_i = y_1 \cdot \dots \cdot y_n$ où pour tout j :
 $y_j = x_j$ si $b_j = 1$ et $y_j = \bar{x}_j$ si $b_j = 0$.

□

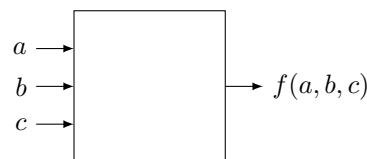
Voici un exemple de cette construction, pour une fonction booléenne f à trois paramètres x, y, z :

x	y	z	f	f_1	f_2	f_3	f_4
0	0	0	0	0	0	0	0
0	0	1	1	1	0	0	0
0	1	0	0	0	0	0	0
0	1	1	1	0	1	0	0
1	0	0	0	0	0	0	0
1	0	1	1	0	0	1	0
1	1	0	1	0	0	0	1
1	1	1	0	0	0	0	0

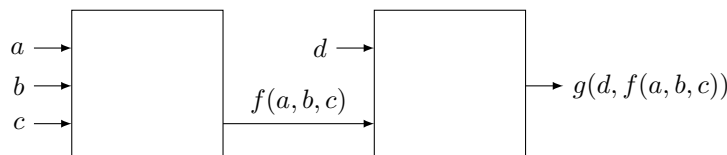
avec $f_1 = \bar{x}.\bar{y}.z$ $f_2 = \bar{x}.y.z$ $f_3 = x.\bar{y}.z$ $f_4 = x.y.\bar{z}$
 donc $f(x, y, z) = \bar{x}.\bar{y}.z + \bar{x}.y.z + x.\bar{y}.z + x.y.\bar{z}$

2.3 Circuits combinatoires

Un circuit combinatoire est un circuit physique élaboré à partir de composants électroniques et réalisant une (ou plusieurs) fonction booléenne. Il comporte des entrées et des sorties. Les entrées et sorties sont des valeurs booléennes et chaque sortie est la valeur d'une fonction booléenne fonction des entrées.



Les circuits peuvent être mis en série pour réaliser des compositions de fonction :




Les circuits combinatoires sont construits à partir de “portes logiques” qui sont les circuits combinatoires les plus simples.

2.3.1 Portes logiques

Les portes logiques réalisent les fonctions booléennes élémentaires ; elles disposent d'entrées (à gauche sur les dessins) et d'une sortie (à droite). Des signaux arrivent sur les entrées (0 ou 1) et un signal est produit sur la sortie.


Voici l'ensemble des portes logiques que nous utiliserons : pour chacune nous indiquons le nom, la représentation graphique, et la fonction booléenne réalisée par la porte sous forme d'une table de vérité.

Porte ET




0	0	0
0	1	0
1	0	0
1	1	1

Porte OU




0	0	0
0	1	1
1	0	1
1	1	1

Porte NON




0	1
1	0

Porte NON-ET




0	0	1
0	1	1
1	0	1
1	1	0

Porte NON-OU



0	0	1
0	1	0
1	0	0
1	1	0

Porte OU-X (ou XOR)



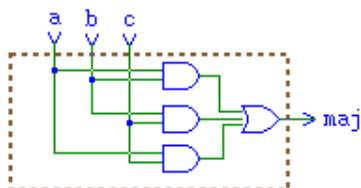
0	0	0
0	1	1
1	0	1
1	1	0

De façon purement physique, les deux états logiques 0 et 1 sont représentés par un signal électrique : le 0 correspond à un signal dans une plage d'intensité basse et le 1 dans une plage d'intensité haute. Les plages dépendent de la technologie utilisée pour réaliser les portes. Une porte logique est souvent composée de **transistors**. Un transistor est un composant électronique se comportant comme un interrupteur très rapide.

Exemple 2.2 La fonction majorité : elle associe au triplet de booléens a_3, a_2, a_1 la valeur 1 si il y a une majorité de 1 en entrée, et 0 sinon. Elle est donc donnée par l'expression booléenne suivante

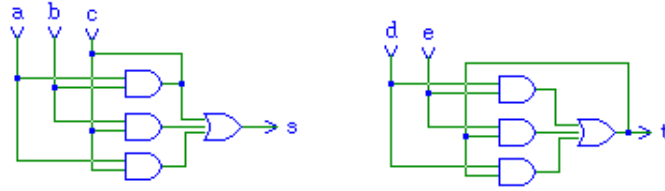
$$maj(a, b, c) = a \cdot b + b \cdot c + c \cdot a.$$

Le circuit correspondant est le suivant :



Notez que les signaux ont été dupliqués pour que les entrées puissent être utilisées plusieurs fois. Chaque duplication est matérialisée par un •. Remarquez également qu'on utilise ici une porte OU à plus de deux entrées. Cela est rendu possible par le fait que l'opérations OU est associative.

Règles de construction Un circuit combinatoire est donc obtenu à partir de portes logiques, en utilisant des duplications de signaux et des compositions. Attention, une sortie ne peut en aucun cas être redirigée sur une entrée. Les exemples suivants ne sont donc pas des circuits combinatoires, et ne peuvent pas être évalués (en fait ils provoquent des court-circuits).



De l'idéal à la réalité... Les circuits combinatoires sont **des idéalizations** dans lesquels le temps de propagation des signaux n'est pas pris en compte. La sortie est donc disponible dès que les entrées sont présentes.

En réalité il faut tenir compte du temps de parcours du courant électrique, et du temps de réponse des portes qui n'est pas instantané.

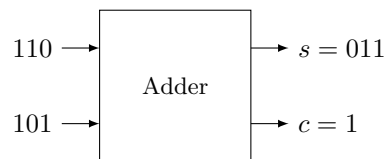
2.3.2 Quelques circuits utiles

Nous décrivons ici quelques circuits qui seront souvent utilisés dans le cours, mais sans forcément leur construction. Certains seront détaillés en travaux dirigés. Jusqu'à présent, nous avons considéré que les "câbles" utilisés pour les entrées, sorties et connexions transportaient un bit. Nous autoriserons à partir de maintenant des câbles de largeur arbitraire n , qui correspondent en fait à n câbles 1 bit ordonnés. Ainsi n entrées a_{n-1}, \dots, a_0 pourront être matérialisées par une entrée $a = (a_{n-1}, \dots, a_0)$ sur n bits. Lorsque la largeur d'un câble est supérieure à 1, elle sera indiquée sur le câble.

2.3.2.1 Additionneur n bits

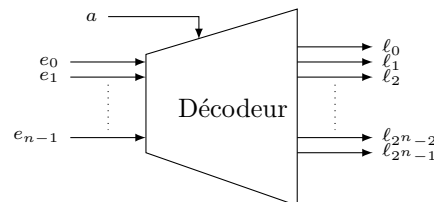
Un additionneur n bits est un circuit ayant 2 entrées sur n bits $a = a_{n-1} \dots a_0$, et $b = b_{n-1} \dots b_0$ et deux sorties : c sur 1 bit, et $s = s_{n-1} \dots s_0$ sur n bits telles que $cs_{n-1} \dots s_0$ est le codage binaire de la somme des deux nombres codés par a et b .

Exemple pour $n = 3$: $(101)_2 + (110)_2 = (1011)_2$



2.3.2.2 Le décodeur

Le schéma suivant représente un décodeur avec ses $n + 1$ entrées et ses 2^n sorties (toutes sur 1 bit).



L'entrée a sert de ligne d'activation : lorsque $a = 0$ toutes les sorties sont à 0. Lorsque $a = 1$, toutes les sorties sont à 0, sauf la ligne ℓ_i telle que i est le codage binaire de $e_{n-1} \dots e_0$:

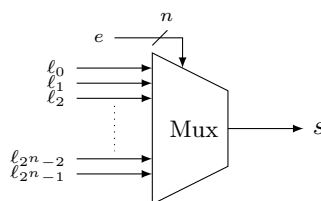
pour tout $i \in [0, 2^n - 1]$, si $i = (e_{n-1} \dots e_0)_2$ alors $\ell_i = 1$, sinon $\ell_i = 0$.

Notez qu'il existe aussi des décodeurs sans ligne d'activation. Il se comporte alors comme dans le cas $a = 1$, et a donc toujours une et une seule sortie égale à 1.

un décodeur 2 vers 4 : Voici un décodeur 2 vers 4 (2 entrées - 4 sorties) représenté par sa table de vérité et les expressions booléennes de chacune des sorties.

e_1	e_0	s_3	s_2	s_1	s_0	
0	0	0	0	0	1	$s_0 = \overline{e_0} \cdot \overline{e_1}$
0	1	0	0	1	0	$s_1 = \overline{e_1} \cdot e_0$
1	0	0	1	0	0	$s_2 = e_1 \cdot \overline{e_0}$
1	1	1	0	0	0	$s_3 = e_0 \cdot e_1$

2.3.2.3 Le multiplexeur



Il comporte 2^n entrées appelées lignes : l_0, \dots, l_{2^n-1} , une entrée e de largeur n et une sortie s . Les lignes et la sortie s doivent toutes avoir la même largeur de bit m . Son fonctionnement est simple, il recopie sur sa sortie la valeur de la ligne d'entrée dont le numéro est codé en binaire par e :

$$s = l_{(e)_2}$$

Par exemple, pour $n = 2$, on a la table (simplifiée) suivante :

e_1	e_0	s
0	0	l_0
0	1	l_1
1	0	l_2
1	1	l_3

2.3.2.4 Unité arithmétique et logique

L'unité arithmétique est logique (UAL ou ALU) est le composant d'un ordinateur chargé d'effectuer les calculs. Les ALU les plus simples travaillent sur des nombres entiers, et peuvent effectuer les opérations communes :

- Les opérations arithmétiques : addition, soustraction, changement de signe etc.,
- les opérations logiques : compléments à un, à deux, ET, OU, OU-exclusif, NON, NON-ET etc.,
- les comparaisons : test d'égalité, supérieur, inférieur, ...
- éventuellement des décalages et rotations (mais parfois ces opérations sont externalisées).

Certaines UAL sont spécialisées dans la manipulation des nombres à virgule flottante, en simple ou double précision (on parle d'unité de calcul en virgule flottante ou floating point unit (FPU)) ou dans les calculs vectoriels. Typiquement, ces unités savent accomplir les opérations suivantes :

- additions, soustractions, changement de signe,
- multiplications, divisions,
- comparaisons,
- modulus

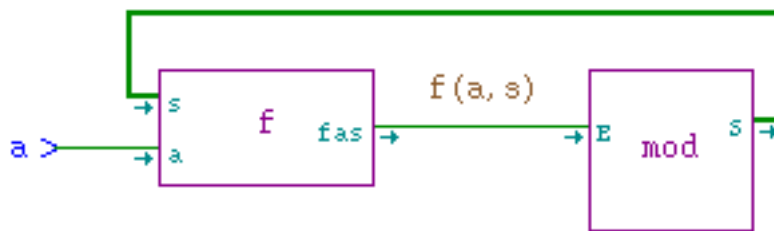
Certaines UAL, le plus souvent de la classe des FPUs, notamment celles des superordinateurs, sont susceptibles d'offrir des fonctions avancées : inverse ($1/x$), racine carrée, logarithmes, fonctions transcendantes ($\sin x$, $\cos x$, etc.), opération vectorielle (produit scalaire, vectoriel, etc.), etc.

Chapitre 3

Circuits séquentiels

La sortie d'un circuit combinatoire est (quasi) instantanée. Les changements de valeurs des entrées engendrent immédiatement la modification de la sortie. Aussi ces circuits ne permettent pas ni mémorisation, ni la synchronisation des sorties.

Les circuits séquentiels contiennent des modules dont les entrées dépendent **des valeurs antérieures des sorties** :



Supposons que f est un circuit combinatoire, on voit sur ce schéma que si module mod est un simple circuit combinatoire, la sortie risque d'osciller très rapidement (par exemple si $f(a, s) = \bar{s}$).

Le module mod doit donc avoir la capacité de stabiliser ses sorties pendant un temps donné. Le temps est donc un paramètre des circuits séquentiels. Ils permettent de stocker une information au cours du temps et sont donc l'élément principal des mémoires.

3.1 Les bascules

La majorité des circuits séquentiels sont réalisés à partir de **bascules** qui sont eux même des circuits séquentiels permettant de mémoriser un bit. Typiquement, une bascule peut-être dans deux états :

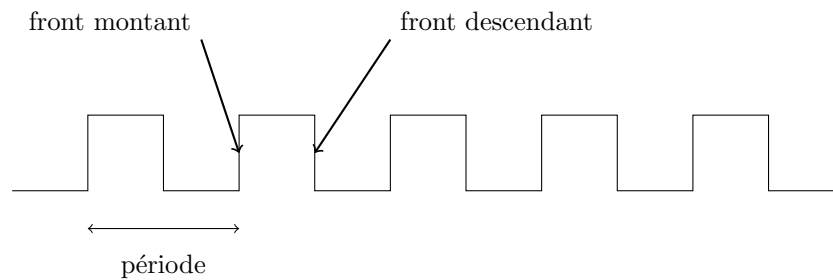
- verrouillée : lorsque une bascule est verrouillée, ses sorties ne peuvent plus être modifiées, et ce même si les entrées changent. Il s'agit donc d'un état de mémorisation.
- déverrouillée : les sorties sont modifiées en fonction des entrées.

Dans ce cours, nous nous focaliserons sur des horloges qui ne peuvent être déverrouillées que sur des petits laps de temps.

Une bascule est fréquemment pilotée par une horloge, qui va enclencher/déclencher son verrouillage.

3.1.1 Horloges

Une horloge produit un signal impulsionnel de fréquence fixe comme représenté dans le chronogramme suivant :



$$\text{fréquence} = \frac{1}{\text{période}}$$

Le signal passe donc alternativement et de manière périodique d'un niveau haut (1) à un niveau bas (0). On supposera que ce passage est instantané (ces passages sont appelés **front**). De plus, on suppose souvent que les temps passés au niveau haut et au niveau bas sont égaux.

Voici le symbole d'une horloge de période 100 :



3.1.2 Modes de synchronisation

Une bascule est pilotée par une horloge, qui va enclencher/déclencher son verrouillage selon deux modes :

- **Déclenchement sur front montant** : ce type de bascule est déclenchée (déverrouillée) lorsque le signal d'horloge est sur front montant. Le reste du temps la bascule est verrouillée.
- **Déclenchement sur front descendant** : ce type de bascule est déclenchée (déverrouillée) lorsque le signal d'horloge est sur front descendant. Le reste du temps la bascule est verrouillée.

Pour être cohérent, un circuit sera généralement synchrone, c'est à dire que toutes les bascules passent en mode mémorisation au même moment. Pour cela, elles sont synchronisées par une même horloge.

Représentation symbolique des modes de synchronisation Les schémas suivants montrent sur un module générique comment les modes de synchronisation seront représentés par la suite :



Déclenchement sur front montant · Déclenchement sur front descendant ·

3.1.3 La bascule D

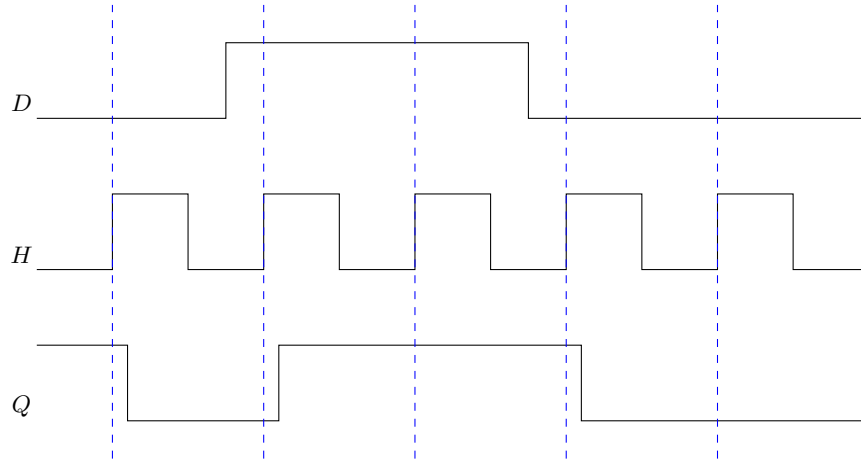
Il s'agit de la bascule la plus simple, et la plus couramment utilisée. Dans sa version minimale, elle dispose d'une entrée D , et d'une sortie Q (comme toutes les bascules elle dispose aussi d'une entrée H pour l'horloge.)

Puisque le temps est maintenant un paramètre, nous le faisons apparaître en utilisant la notation suivante : si Q est la valeur de la sortie à un instant donné, Q' est la valeur de cette même sortie à l'instant suivant.

Le fonctionnement de la bascule D est le suivant :

- en mode verrouillé : $Q' = Q$ (mémorisation)
- en mode déverrouillé : $Q' = D$ (l'entrée est recopiée sur la sortie)

Pour bien comprendre, examinons le chronogramme d'une bascule D à déclenchement sur front montant. Le chronogramme est une représentation graphique des entrées et des sorties au cours du temps.



On peut remarquer que la valeur de Q n'est modifiée qu'aux fronts montants de l'horloge, et à chaque modification, elle prend la valeur de D .

3.1.4 Quelques bascules

Nous résumons ici le fonctionnement des bascules les plus couramment utilisées. Elles pourront toutes être déclinées selon les 2 modes de synchronisation. En mode verrouillé, toutes ces bascules auront pour équation $Q' = Q$, nous ne présentons donc les fonctionnements qu'en mode actif.

La bascule D : Une entrée D , une entrée d'horloge, et une sortie Q d'équation $Q' = D$.

La bascule JK : Une entrée J , une entrée K , une entrée d'horloge, et une sortie Q d'équation $Q' = J\bar{Q} + \bar{K}Q$.

La table de fonctionnement est la suivante :

J	K	Q'
0	0	Q
0	1	0
1	0	1
1	1	\bar{Q}

On peut remarquer le fonctionnement suivant :

Pour passer de	Q	à	Q'	il suffit que	car
	0		0	$J = 0$	$Q' = 0\bar{0} + \bar{K}0 = 0$
	0		1	$J = 1$	$Q' = 1\bar{0} + \bar{K}0 = 1$
	1		0	$K = 1$	$Q' = J\bar{1} + \bar{1}1 = 0$
	1		1	$K = 0$	$Q' = J\bar{1} + \bar{0}1 = 1$

Donc pour passer d'une valeur Q à une valeur Q' , il suffit de n'agir que sur une seule entrée.

La bascule T Une entrée T , une entrée d'horloge, et une sortie Q d'équation $Q' = T\bar{Q} + \bar{T}Q$. En d'autres mots, lorsque $T = 1$, $Q' = \bar{Q}$ et lorsque $T = 0$, $Q' = Q$.

La table de fonctionnement est la suivante :

T	Q'
0	Q
1	\bar{Q}

3.1.5 Forçage des bascules

Sur les bascules, il existe toujours des entrées supplémentaires permettant l'initialisation lors de la mise sous tension. Ces entrées sont asynchrones : leur comportement est indépendant du signal d'horloge. On trouve généralement ;

- une entrée Clear (C) pour la mise à 0,
- une entrée Enable (E) pour activer la bascule

Pour des raisons de conception, ces entrées sont en général inversées (elles s'appellent alors \bar{C} et \bar{E}), leur effet est alors le suivant :

- si $\bar{C} = 0$, la sortie est forcée à 0
- si $\bar{E} = 1$, la bascule est verrouillée. En fonctionnement normal on a donc $\bar{E} = 0$.

3.2 Quelques circuits séquentiels

3.2.1 Les compteurs

On présente ici l'exemple d'un compteur modulo 8. Il dispose de 3 sorties Q_2, Q_1, Q_0 codant un entier entre 0 et 7. De façon périodique, les sorties sont modifiées de façon à afficher de façon cyclique les sorties 0,1,2,3,4,5,6,7,0,1, ...

Il s'agit d'un système synchrone. Si à un instant les sorties $(Q_2Q_1Q_0)$ codent en entier n , au prochain front (montant par exemple) de l'horloge, elle vaudront $n + 1$ (modulo 8). On a donc l'équation suivante : $(Q'_2Q'_1Q'_0)_2 = (Q_2Q_1Q_0)_2 + 1[8]$.

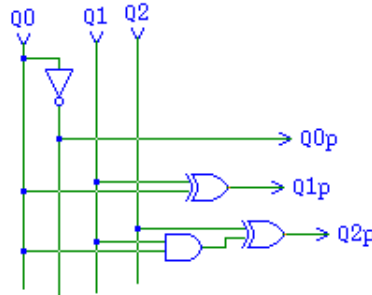
Nous pouvons donc représenter la valeur de chaque Q'_i par une fonction booléenne de (Q_2, Q_1, Q_0) .

La table de vérité est et les expressions correspondantes sont les suivantes :

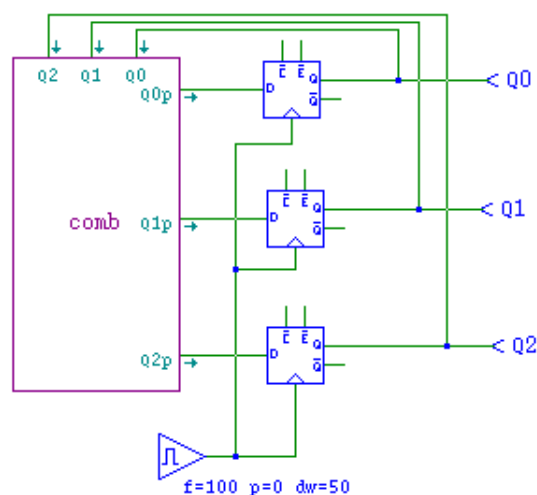
Q_2	Q_1	Q_0	Q'_2	Q'_1	Q'_0
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

$$\begin{aligned}
 Q'_0 &= \bar{Q}_0 \\
 Q'_1 &= Q_0 \cdot \bar{Q}_1 + Q_1 \cdot \bar{Q}_0 = Q_0 \oplus Q_1 \\
 Q'_2 &= \bar{Q}_2 \cdot (Q_1 \cdot Q_0) + Q_2 \cdot \overline{(Q_1 \cdot Q_0)} = Q_2 \oplus (Q_1 \cdot Q_0).
 \end{aligned}$$

Nous pouvons maintenant facilement construire un circuit combinatoire **comb** dont les entrées sont Q_2, Q_1, Q_0 et les sorties Q'_2, Q'_1, Q'_0 :



Chaque sortie Q_i devant être mémorisée, nous utiliserons une bascule D par sortie. En utilisant l'équation de la bascule : $Q' = D$, il nous suffit alors de brancher chaque sortie Q'_i sur l'entrée D de la i -ème bascule :



Réalisation avec des bascules JK La réalisation avec des D est la plus directe, mais pas toujours celle qui donne le circuit le plus simple. Donnons maintenant une construction du compteur en utilisant des bascules JK

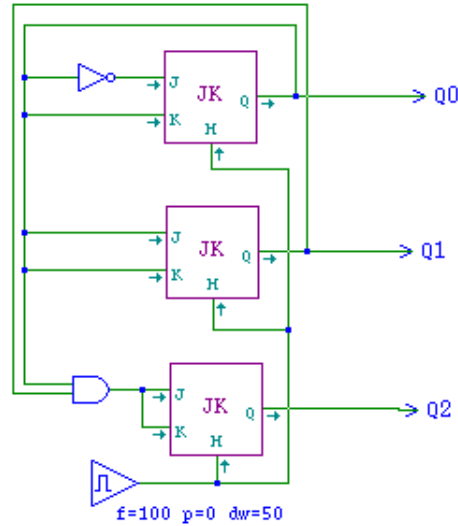
On rappelle le fonctionnement de la bascule JK :	Pour passer de	Q	à	Q'	il suffit que
		0		0	$J = 0$
		0		1	$J = 1$
		1		0	$K = 1$
		1		1	$K = 0$

On peut alors étendre la table pour obtenir les fonctions donnant les entrées des bascules (lorsque la valeur n'importe pas, elle est remplacée par un $-$) :

Q_2	Q_1	Q_0	Q_2'	Q_1'	Q_0'	J_2	K_2	J_1	K_1	J_0	K_0
0	0	0	0	0	1	0	-	0	-	1	-
0	0	1	0	1	0	0	-	1	-	-	1
0	1	0	0	1	1	0	-	-	0	1	-
0	1	1	1	0	0	1	-	-	1	-	1
1	0	0	1	0	1	-	0	0	-	1	-
1	0	1	1	1	0	-	0	1	-	-	1
1	1	0	1	1	1	-	0	-	0	1	-
1	1	1	0	0	0	-	1	-	1	-	1

On obtient alors les équations et le circuit suivant :

$$\begin{aligned}
J_2 &= Q_1 Q_0 \\
K_2 &= Q_1 Q_0 \\
J_1 &= Q_0 \\
K_1 &= Q_0 \\
J_0 &= \bar{Q}_0 \\
K_0 &= Q_0
\end{aligned}$$



Remarquez que le circuit obtenu est beaucoup plus simple que celui utilisant des bascules D.

On aurait pu aussi obtenir ces équations directement depuis celles de Q'_2, Q'_1, Q'_0 , en forçant l'apparition de l'équation de la bascule JK : $Q' = J\bar{Q} + \bar{K}Q$:

$$Q'_0 = \bar{Q}_0 = \bar{Q}_0 + 0 = 1 \cdot \bar{Q}_0 + 0 \cdot Q_0, \text{ donc : } J_0 = 1 \text{ et } K_0 = 1.$$

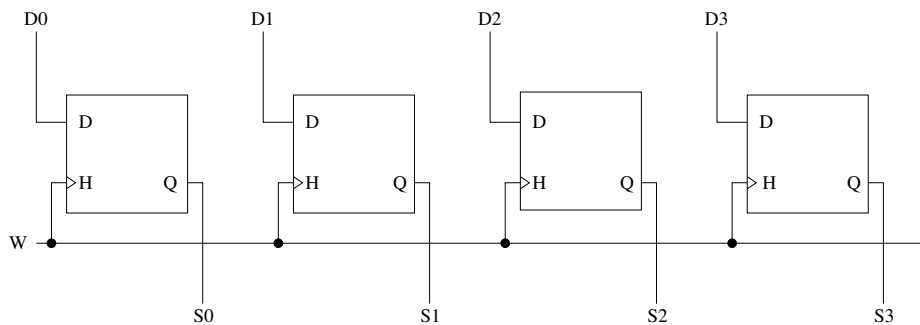
$$Q'_1 = Q_0 \cdot \bar{Q}_1 + Q_1 \cdot \bar{Q}_0, \text{ donc } J_1 = Q_0 \text{ et } K_1 = Q_0.$$

$$Q'_2 = \bar{Q}_2 \cdot (Q_1 \cdot Q_0) + Q_2 \cdot \overline{(Q_1 \cdot Q_0)}, \text{ donc } J_2 = Q_1 Q_0 \text{ et } K_2 = Q_1 Q_0.$$

3.2.2 Les registres

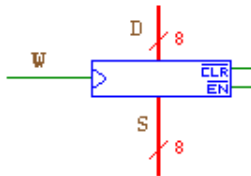
Puisque les bascules permettent la mémorisation de bits, elles sont le composant principal des registres.

Voici par exemple un registre 4 bits : c'est à dire un module capable de mémoriser un mot de 4 bits. L'entrée W ordonne l'écriture de l'entrée $D_3 D_2 D_1 D_0$ dans le registre : en synchronisme avec le signal d'écriture W, le registre mémorise les données présentes sur les entrées D_0, D_1, D_2 et D_3 . Elles sont conservées jusqu'au prochain signal de commande W. Les valeurs mémorisées peuvent être lues sur les sorties Q_0, Q_1, Q_2 et Q_3 .



L'entrée W pourra être un signal d'horloge, dans ce cas, le registre sera périodiquement mis à jour.

Par la suite, les registres seront représentés par un module de la forme suivante (ici un registre 8 bits) :



Remarques que comme les bascules qui comportent deux entrées \bar{C} et \bar{EN} qui ont exactement la même action que sur les bascules.

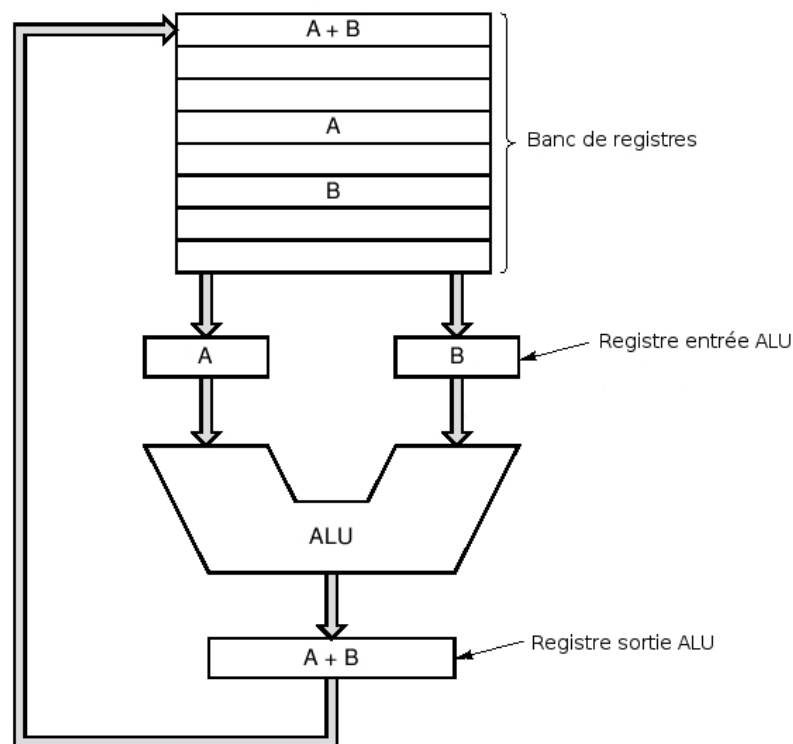
Il existe différentes variantes de registres, par exemple le registre à décalage permet de décaler le mot stocké en mémoire. Un registre à décalage à droite peut-être utilisé comme un diviseur par 2, alors qu'un registre à décalage à gauche agit comme un multiplicateur par 2.

3.2.3 Banc de registres

Les bancs de registres contiennent plusieurs registres. Il est possible de sélectionner un ou plusieurs registres pour la lecture et l'écriture via leur adresse. Pour adresser le bon registre on utilise des décodeurs, des multiplexeurs, et/ou des démultiplexeurs.

3.2.3.1 Fonctionnement : exemple un banc de registres 3 bits

Nous détaillons ici le fonctionnement et la conception d'un banc de $n + 1$ registres stockant des mots de taille 3. Il est possible d'écrire dans un registre, et de lire en parallèle le contenu de 2 registres. Ce type de banc de registres est couramment utilisé comme avec une ALU. Les opérandes d'un calcul sont contenues dans les registres et sont lues simultanément, le résultat d'une opération est ensuite stockée dans un registre, comme représenté dans le schéma ci-dessous :



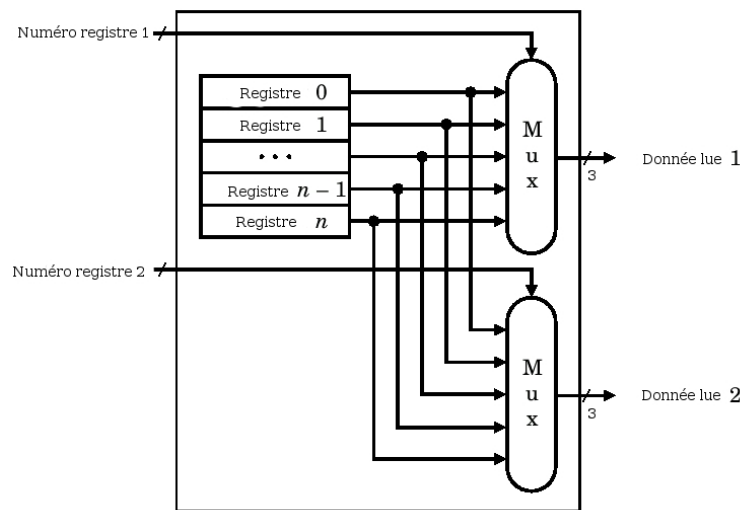
Sur ce banc de registres, on note les 2 (bus de) sorties, qui vont permettre de lire en même temps deux contenus de registres qui vont être stockés dans les deux registres A et B d'entrées de l'ALU. L'ALU réalise alors l'opération (ici une somme) et le résultat est placé dans le registre de sortie de l'ALU. Finalement, le contenu de ce registre de sortie d'ALU est placé sur l'entrée du banc de registre pour être lu par ce dernier.

Ici, on voit que les entrées et les sorties sont représentées par des flèches épaisses : il s'agit de **bus**, plusieurs fils en parallèle regroupant plusieurs bits d'information traités en parallèle.

Considérons un de $n + 1$ registres de 3 bits avec

- un port d'écriture
- deux ports de lecture (lors d'une lecture le contenu de deux registres est lu même temps.)

Fonctionnement en lecture : (2 registres en parallèle)

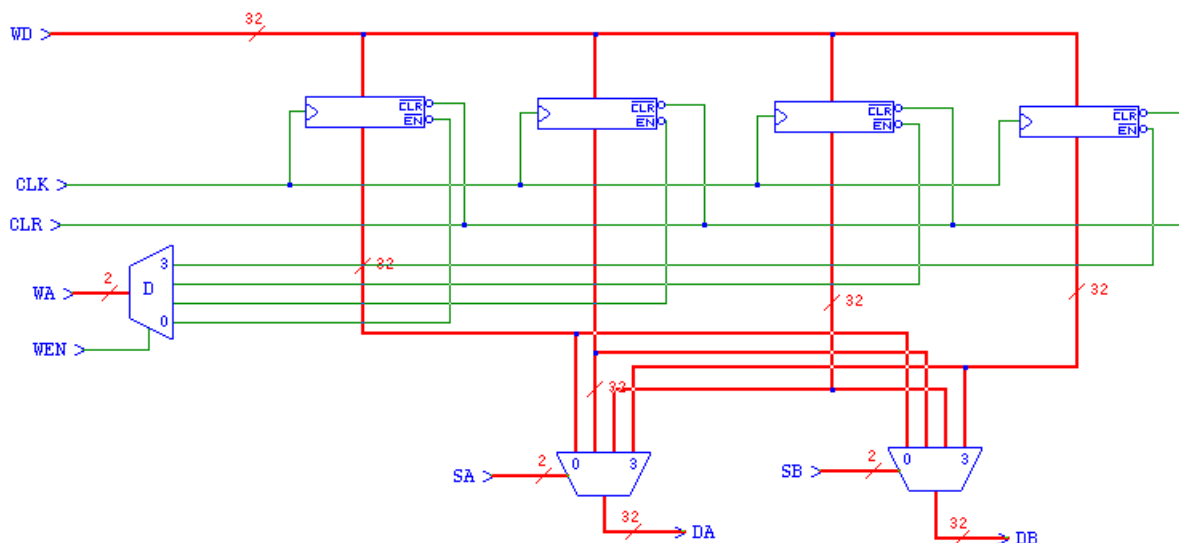


Nous avons vu tout à l'heure une façon explicite de sélectionner les sorties d'une ligne de bascule D dans un banc mémoire; on fait de même ici mais en utilisant un circuit "haut niveau", un multiplexeur. Selon la valeur de l'entrée *Numéro registre 1*, le multiplexeur sélectionne une de ses entrées pour la mettre sur sa sortie. On remarque que notre banc mémoire possède deux ports de lecture puisqu'il possède un second multiplexeur contrôlé par une seconde entrée, *Numéro registre 2*.

Nota bene : Les multiplexeurs utilisés ici ne fonctionnent pas sur des fils isolés mais sur des bus de largeur 3 (il sélectionne 3 fils pour une valeur de sélection).

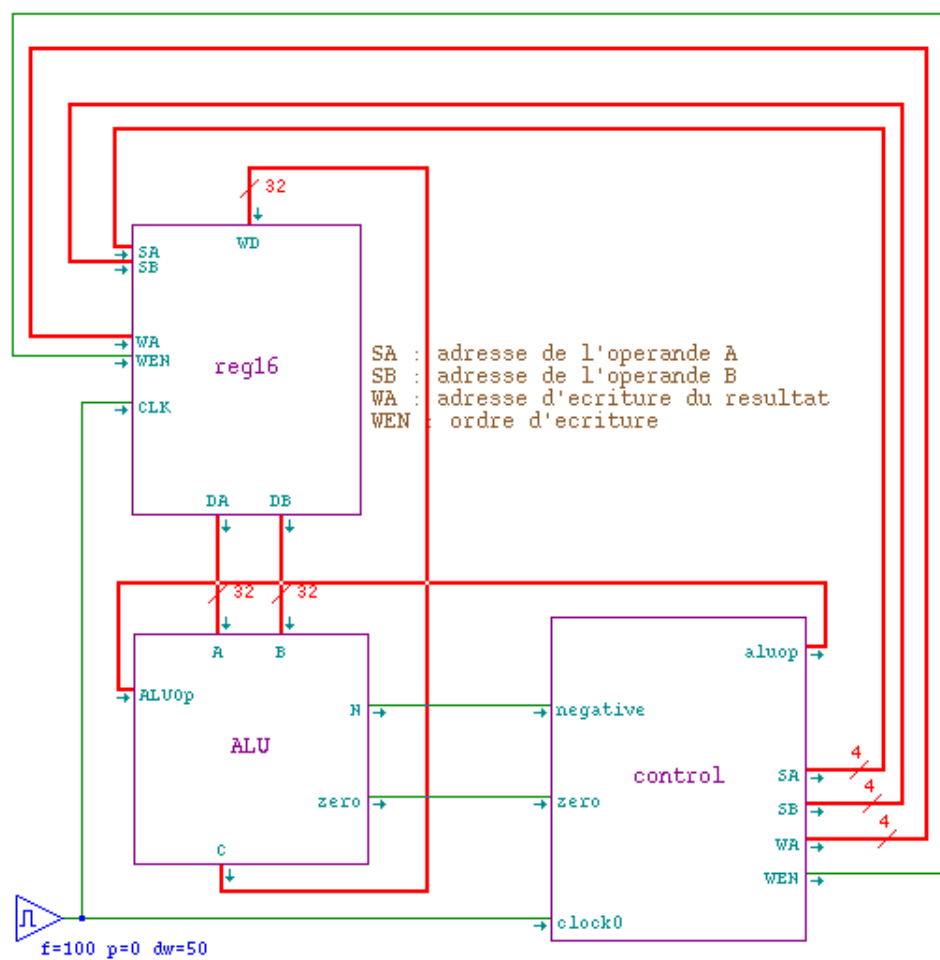
Fonctionnement en écriture : Une entrée *WEN* (Write ENable) déclenche l'écriture dans le registre d'adresse *WA* (Writing Adress). On utilise pour cela un décodeur, qui va décoder l'adresse *WA* et activer le registre d'adresse *WA*.

Voici une réalisation complète d'un banc de 4 registres 32 bits : *CLK* (CLOCK) est un signal d'horloge, l'écriture dans un sera registre se fera sur front montant. *CLR* (CLeaR) est un signal de mise à 0 de tous les registres. *WD* est la donnée à stocker dans un registre. *SA* et *SB* sont les adresses des deux registres dont on souhaite lire les valeurs.



3.2.4 Vers la construction d'un calculateur

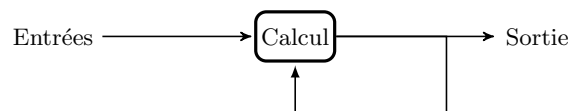
Nous savons maintenant comment construire un banc de registre et une ALU. Il nous reste à comprendre comment construire un module de contrôle contenant un programme en mémoire et capable d'exécuter ce programme en pilotant l'ALU et le banc de registre selon le schéma simplifié suivant :



Chapitre 4

Conception et analyse de circuits séquentiels

Dans le cours précédent, nous avons construit nos premiers circuits séquentiels, en utilisant des bascules. Les spécifications de ces circuits étaient relativement simples car elles avaient la particularité suivante : les sorties étaient modifiées périodiquement, en fonction de leurs valeurs précédentes et les entrées. Les spécifications étaient donc du type suivant :

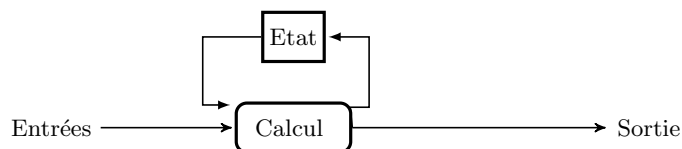


Pour construire le circuit relatif réalisant une telle spécification, il a suffi de mémoriser les sorties dans des bascules, et d'établir les tables de vérités des sorties en fonction du temps et des entrées.

Pour des spécifications plus élaborées, la conception du circuit peut être plus difficile car les sorties peuvent dépendre en plus d'un *état interne* du système.

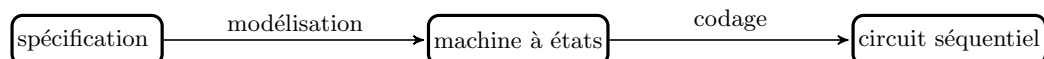
Imaginez par exemple un distributeur de boissons, il ne suffit pas d'appuyer sur un bouton pour obtenir une bouteille d'eau, il faut d'abord avoir introduit une somme suffisante. Le distributeur ne délivrera donc la bouteille que s'il a atteint un état interne stipulant que la somme nécessaire à distribuer à la bouteille a été insérée.

Ce type de spécification est du type suivant :

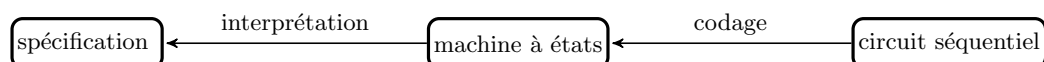


Pour ce genre de comportement complexe, il sera donc nécessaire de représenter préalablement le système par une *machine à états*. Nous considérerons deux types de machines : les machines de Mealy, et les machines de Moore, qui sont des automates finis particuliers.

Une fois la machine à état modélisant une spécification donnée réalisée, la construction du circuit séquentiel associé est directe et revient encore à l'élaboration de circuits combinatoires :



Inversement, un circuit séquentiel donné pourra être, par simple codage, représenté par une machine à état, qui elle-même pourra être en général interprétée par une spécification :



4.1 Automates finis déterministes

Un automate fini déterministe (AFD) est une machine abstraite permettant de reconnaître des mots. Ces mots peuvent être vus comme des suites d'actions autorisées par machine. Après chaque action, l'automate change son état interne, en fonction de l'action réalisée et de l'état courant. Une suite d'action (un mot) est acceptée par l'automate si, partant de l'état initial de l'automate, la suite d'actions le mène à un état dit *final*.

Formellement, un automate est une structure $\mathcal{A} = (Q, q_0, A, \delta, F)$ où :

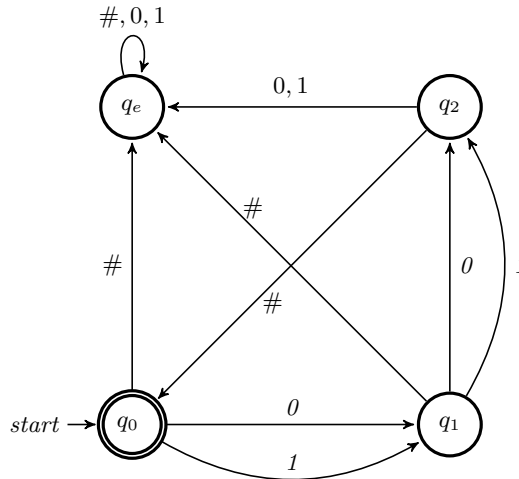
- Q est un ensemble fini d'états,
- $q_0 \in Q$ est l'état initial,
- A est l'alphabet d'entrées (les actions),
- $\delta : Q \times A \rightarrow Q$ est la *fonction de transition*,
- $F \subseteq Q$ est l'ensemble des états finaux.

La fonction de transition fait passer l'automate de l'état q à l'état $\delta(q, a)$ lorsqu'une action a est effectuée.

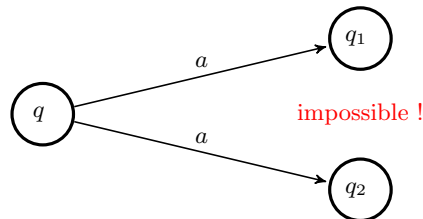
Exemple 4.1 Voici l'automate modélisant un récepteur qui lit séquentiellement des mots de 2 bits séparés par un symbole # :

$\mathcal{A} = (Q, q_0, A, \delta, F)$ où : $Q = \{q_0, q_1, q_2, q_e\}$, $A = \{0, 1, \#\}$, $F = \{q_0\}$. La fonction de transition est donnée par table ci-dessous. L'automate admet une représentation graphique comme représenté ci-dessous : les états sont les sommets du graphe, l'état initial est marqué par une flèche entrante et les états finaux par un double cercle. La fonction de transition correspond aux arcs étiquetés par les actions.

état q	action a	$\delta(q, a)$
q_0	0	q_1
q_0	1	q_1
q_0	#	q_e
q_1	0	q_2
q_1	1	q_2
q_1	#	q_e
q_2	0	q_e
q_2	1	q_e
q_2	#	q_0
q_e	0	q_e
q_e	1	q_e
q_e	#	q_e



Un automate fini déterministe est donc un graphe, dont les sommets sont des états, et dont les flèches sont étiquetées par des actions et représentent la fonction de transition. Étant donné un état q et une action a , il y a donc une et une seule flèche issue de q et étiquetée par a . On appelle cette propriété le *déterminisme* : le prochain état est déterminé, le système n'a pas de choix à faire.



A toute séquence $u = a_1 a_2 \dots a_n$ d'actions correspond donc un et un seul calcul dans l'automate, qui correspond à la suite des états successifs pris par l'automate pendant la lecture des actions.

Exemple 4.2 Reprenons l'automate de l'exemple 4.1.

Calcul du mot 00#01# : $q_0 \xrightarrow{0} q_1 \xrightarrow{0} q_2 \xrightarrow{\#} q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_2 \xrightarrow{\#} q_0$. Le calcul termine dans l'état final, la séquence 00#01# est donc acceptée.

Calcul du mot $0\#0 : q_0 0 q_1 \# q_e 0 q_e$. Le calcul termine dans l'état q_e qui n'est pas un état final, la sequence $0\#0$ est donc refusée.

Définition 4.3 Soit $\mathcal{A} = (Q, q_0, A, \delta, F)$ un AFD et $u = a_1 \dots a_n$ une séquence d'actions non vide. Le calcul de u par \mathcal{A} est l'unique séquence $q_0 a_1 q_1 \dots q_{n-1} a_n q_n$ telle que pour tout $i \in [1, n]$, $\delta(q_{i-1}, a_i) = q_i$.

Ce calcul est dit acceptant si q_n est un état final ($q_n \in F$). Si le calcul de u est acceptant, le mot u est dit accepté par l'automate, sinon il est rejeté.

Les AFD représentent donc des systèmes capables d'accepter ou de rejeter des suites d'actions mais ne pouvant pas répondre aux actions et aux changements d'états, en envoyant des signaux vers l'extérieur. Nous introduisons maintenant deux types de machines, les machines de Moore, et les machines de Mealy, assimilables à des AFD munis d'une fonction de sortie. Chaque type de machine modélise un certain type de circuit séquentiel.

4.2 Machine de Moore

Une machine de Moore est un AFD muni d'une fonction de sortie associant chaque état à une action de sortie. Pendant un calcul, la machine fournit une sortie à chaque changement d'état. A chaque action d'entrée, correspond donc une action de sortie.

Définition 4.4 Une machine de Moore est une structure $\mathcal{M} = (Q, q_0, A, S, \delta, \sigma)$ où $\mathcal{A}_{\mathcal{M}} = (Q, q_0, A, \delta, F)$ est un AFD, S est l'ensemble des actions de sorties et $\sigma : Q \rightarrow S$ est la fonction de sortie.

Etant donné un mot d'entrée $u = a_1 \dots a_n$, et $q_0 a_1 q_1 \dots q_{n-1} a_n q_n$ son calcul dans l'automate $\mathcal{A}_{\mathcal{M}}$, le fonctionnement de \mathcal{M} sur u est la paire

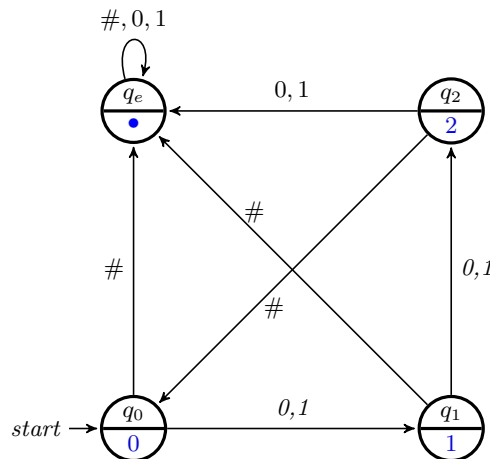
$$(a_1 \dots a_{n-1} a_n, \sigma(q_1) \dots \sigma(q_{n-1}) \sigma(q_n)).$$

Exemple 4.5 La machine suivante lit séquentiellement des mots de 2 bits séparés par un symbole $\#$ et sort la taille de la séquence déjà lue.

$\mathcal{M} = (Q, q_0, A, S, \delta, \sigma)$ où Q, A, δ sont donnés Exemple 4.1, $S = \{0, 1, 2, \bullet\}$ et σ est la fonction

q	$\sigma(q)$
q_0	0
q_1	1
q_2	2
q_e	\bullet

Cette machine est représentée par le graphe de l'automate sous-jacent dans lequel on a représenté la fonction de sortie dans les sommets :



Notez que puisque la sortie ne dépend que de l'état courant, l'action ayant mené à cet état, ainsi que l'état précédent n'influent pas directement sur la sortie. Par exemple, quand on passe de l'état q_0 à l'état q_1 , la sortie est indépendante du fait que l'action ait été un 0 ou un 1.

Voici quelques exemples de fonctionnements :

- *Fonctionnement du mot $00\#01\#$:*
Son calcul est $q_00q_10q_2\#q_00q_11q_2\#q_0$, donc son fonctionnement est

($00\#01\#$, 120220).

- *Fonctionnement du mot $0\#0$:*
Son calcul est $q_00q_1\#q_e0q_e$, donc son fonctionnement est

($0\#0$, 1●●).

4.2.1 Représentation d'un circuit séquentiel par une machine de Moore

Tout circuit de la forme suivante admet un codage direct vers une machine de Moore :

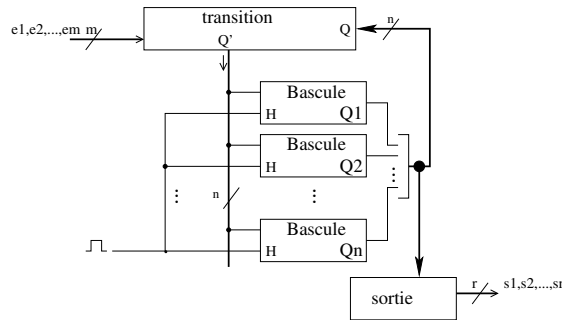


FIGURE 4.1 – un circuit de type Moore

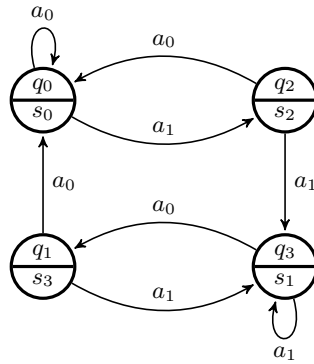
La caractéristique de ces circuits est que le circuit calculant les sorties ne dépend que de l'état (la sortie des bascules). Le circuit est donc totalement synchrone : les sorties sont modifiées toutes au même moment, sur le front d'horloge.

On peut remarquer facilement ce circuit est la réalisation d'une machine de Moore en notant qu'il est constitué des éléments suivants :

- une entrée composée de m bits e_1, \dots, e_m : donc le système admet 2^m valeurs d'entrées possibles. Notons A l'ensemble de ces valeurs ;
- une sortie codée sur r bits s_1, \dots, s_r : le système admet donc 2^r valeurs possibles de sortie. Notons S l'ensemble de ces valeurs ;
- n bascules D ayant pour sorties q_1, \dots, q_n : le système admet donc 2^n états internes possibles. Notons Q l'ensemble de ces états ;
- un circuit combinatoire **transition** représentant une fonction $\delta : Q \times A \rightarrow Q$;
- un circuit combinatoire **sortie** représentant une fonction $\sigma : Q \rightarrow S$.

On voit donc que ce circuit réalise exactement la machine de Moore $\mathcal{M} = (Q, q_0, A, S, \delta, \sigma)$ où q_0 est l'état d'initialisation des bascules (en général $00 \dots 0$).

Exemple 4.6 *Considérons le circuit suivant,*



On peut en déduire la spécification du circuit. Il fonctionne selon deux phases :

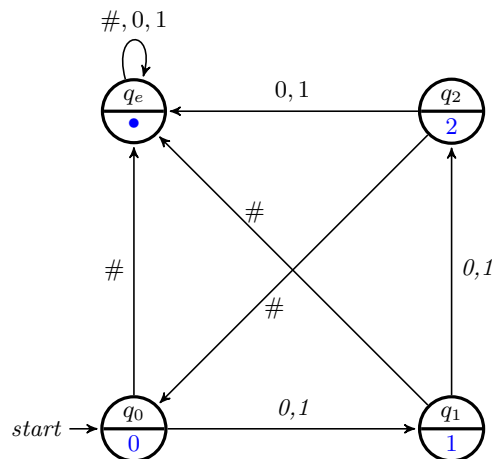
- la phase où sont déjà entrés deux a_0 l'affilée : le système sort ensuite s_2 si entre un a_1 isolé, et s_0 sinon.
- la phase où sont déjà entrés deux a_1 l'affilée : le système sort ensuite s_3 si entre un a_0 isolé, et s_1 sinon.

4.2.2 De la machine vers le circuit

Nous avons vu comment un circuit séquentiel pouvait être représenté par une machine de Moore. Pour passer d'une machine de Moore à un circuit, la démarche est symétrique :

1. Ecrire la fonction de transition et la fonction de sortie dans une table.
2. Coder en binaire les états, les entrées et sorties.
3. Injecter ce codage dans les tables écrites à l'étape 1 : on obtient des tables de vérités.
4. Tirer deux circuits combinatoires de ces tables de vérité
5. Enfin, relier ces circuits à des bascules D pour obtenir un circuit similaire à celui présenté Figure 4.1.

Exemple 4.7 Reprenons l'exemple de la machine lisant des sequences de bits :



1. Ecrire la fonction de transition et la fonction de sortie dans une table.

état q	action a	$\delta(q, a)$
q_0	0	q_1
q_0	1	q_1
q_0	#	q_e
q_1	0	q_2
q_1	1	q_2
q_1	#	q_e
q_2	0	q_e
q_2	1	q_e
q_2	#	q_0
q_e	0	q_e
q_e	1	q_e
q_e	#	q_e

q	$s(q)$
q_0	0
q_1	1
q_2	2
q_e	•

2. Coder en binaire les états et si besoin les entrées et sorties.

q	Q_1	Q_0
q_0	0	0
q_1	0	1
q_2	1	0
q_e	1	1

a	a_1	a_0
0	0	0
1	0	1
#	1	0

S	s_1	s_0
0	0	0
1	0	1
2	1	0
•	1	1

3. Injecter ce codage dans les tables écrites à l'étape 1 : on obtient des tables de vérités.

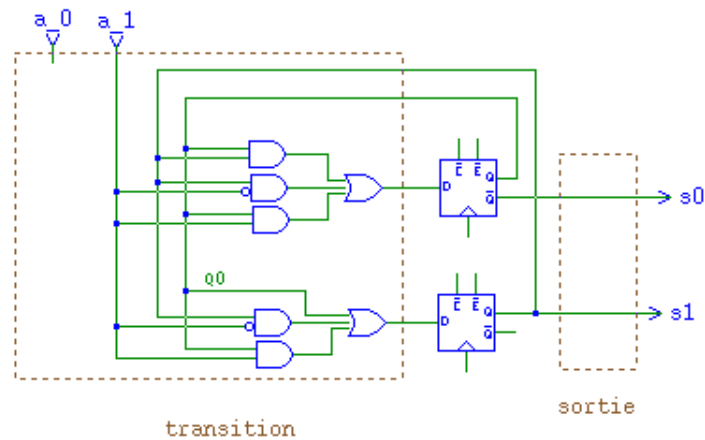
Q_1	Q_0	a_1	a_0	Q_1	Q_0
0	0	0	0	0	1
0	0	0	1	0	1
0	0	1	0	1	1
0	1	0	0	1	0
0	1	0	1	1	0
0	1	1	0	1	1
1	0	0	0	1	1
1	0	0	1	1	1
1	0	1	0	0	0
1	1	0	0	1	1
1	1	0	1	1	1
1	1	1	0	1	1

Q_1	Q_0	s_1	s_0
0	0	0	0
0	1	0	1
1	0	1	0
1	1	1	1

$$Q'_1 = Q_0 + Q_1 \bar{a}_1 + \bar{Q}_1 a_1 \quad Q'_0 = Q_1 Q_0 + Q_1 \bar{a}_1 + Q_0 a_1 \quad s_1 = Q_1 \quad s_0 = Q_0$$

4. De la table de la fonction de transition on obtient un circuit combinatoire **transition**, et de la table de la fonction de sortie, on tire un circuit combinatoire **sortie**.

5. Enfin, on relie ces circuits à des bascules D pour obtenir un circuit similaire à celui présenté Figure 4.1.



4.3 Machines de Mealy

Une machine de Mealy est un AFD muni d'une fonction de sortie dépendant de l'état courant et de l'entrée. Pendant un calcul, la machine fournit une sortie à chaque changement d'état. A chaque action d'entrée, correspond donc une action de sortie.

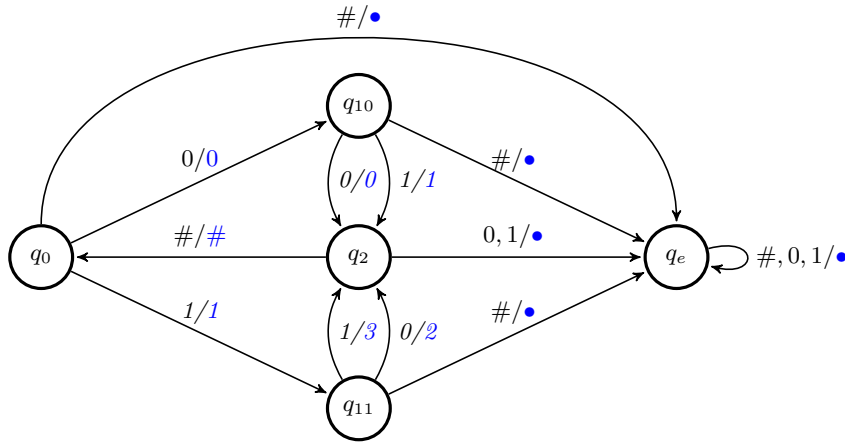
Définition 4.8 Une machine de Mealy est une structure $\mathcal{M} = (Q, q_0, A, S, \delta, \sigma)$ où $\mathcal{A}_{\mathcal{M}} = (Q, q_0, A, \delta, F)$ est un AFD, S est l'ensemble des actions de sorties et $\sigma : Q \times A \rightarrow S$ est la fonction de sortie.

Etant donné un mot d'entrée $u = a_1 \dots a_n$, et $q_0 a_1 q_1 \dots q_{n-1} a_n q_n$ son calcul dans l'automate $\mathcal{A}_{\mathcal{M}}$, le fonctionnement de \mathcal{M} sur u est la paire

$$(a_1, a_2, \dots, a_n, \sigma(q_0, a_1)\sigma(q_1, a_2) \dots \sigma(q_n, a_n)).$$

Exemple 4.9 La machine suivante lit séquentiellement des mots de 2 bits séparés par un symbole # et sort le nombre codé en binaire après la séquence déjà lue.

Cette fois, la fonction de sortie est représentée sur les arcs puisqu'elle dépend de aussi de l'action réalisée.



Notez que puisque la sortie ne dépend que de l'état courant, l'action ayant mené à cet état, ainsi que l'état précédent n'influent pas directement sur la sortie. Par exemple, quand on passe de l'état q_0 à l'état q_1 , la sortie est indépendante du fait que l'action ait été un 0 ou un 1.

Voici quelques exemples de fonctionnements :

— Fonctionnement du mot $10\#11\#$:

Son calcul est $q_0 1 q_{11} 0 q_2 \# q_0 1 q_{11} 1 q_2 \# q_0$, donc son fonctionnement est

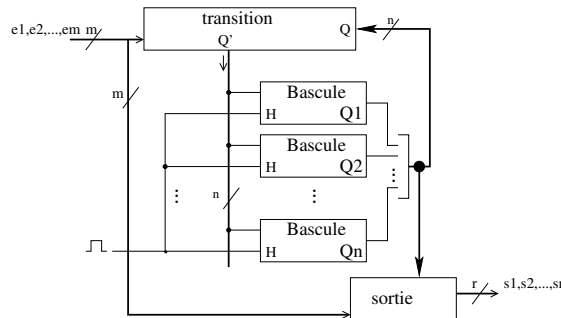
$$(10\#11\#, 12\#13\#).$$

— Fonctionnement du mot $0\#0$:

Son calcul est $q_0 0 q_1 \# q_e 0 q_e$, donc son fonctionnement est

$$(0\#0, 0 \bullet \bullet).$$

La synthèse du circuit à partir de la machine, et l'abstraction d'un circuit par une machine de Mealy se fait exactement de la même façon que dans le cas des machines de Moore, mais cette fois, le circuit est du type suivant :



Les caractéristiques de ce type de circuit sont les suivantes :

- le calcul des sorties dépend de l'état et de l'entrée
- les sorties sont modifiées à chaque fois que les entrées sont modifiées (sorties asynchrones)

Exemple 4.10 (Circuit d'arbitrage) Un système dispose d'une ressource qui ne peut être utilisée que par un agent à la fois, et deux agents sont susceptibles d'utiliser cette ressource. Afin d'éviter les conflits entre les deux agents, on utilise un **circuit d'arbitrage** qui va décider qui peut utiliser la ressource. Le circuit possède deux entrées r_0 et r_1 (les requêtes des agents : $r_i = 1$ ssi l'agent i demande à utiliser la ressource) et deux sorties s_0 et s_1 ($s_i = 1$ indique que l'agent i est autorisé à utiliser la ressource).

La condition de bon fonctionnement est

- si $r_0 = 1$ ou $r_1 = 1$ alors $s_0 = 1$ ou $s_1 = 1$
- si $s_0 = 1$ alors $r_0 = 1$
- si $s_1 = 1$ alors $r_1 = 1$
- si $r_0 = 0$ et $r_1 = 0$ alors $s_0 = 0$ et $s_1 = 0$
- A chaque instant, $s_0.s_1 = 0$ (exclusion mutuelle)

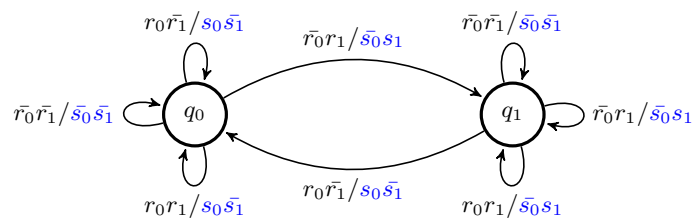
On ajoute une condition supplémentaire de stabilité : si $r_0 = r_1 = 1$ alors c'est le dernier agent i qui a utilisé la ressource qui est autorisé à l'utiliser (donc $s_i = 1$).

L'abstraction du système est donc la suivante :

- entrées : $E = \{r_0r_1, \bar{r}_0r_1, r_0\bar{r}_1, \bar{r}_0\bar{r}_1\}$,
- sorties : $S = \{s_0s_1, \bar{s}_0s_1, s_0\bar{s}_1, \bar{s}_0\bar{s}_1\}$,
- états internes : se sont les valeurs à mémoriser : "qui a été le dernier agent autorisé à utiliser la ressource", il y a donc deux états : $Q = \{q_0, q_1\}$,
- fonction de sortie f et fonction d'entrée g donnés par :

E	Q	σ	δ
$\bar{r}_0\bar{r}_1$	q_0	$\bar{s}_0\bar{s}_1$	q_0
$\bar{r}_0\bar{r}_1$	q_1	$\bar{s}_0\bar{s}_1$	q_1
$r_0\bar{r}_1$	q_0	$s_0\bar{s}_1$	q_0
$r_0\bar{r}_1$	q_1	$s_0\bar{s}_1$	q_0
\bar{r}_0r_1	q_0	\bar{s}_0s_1	q_1
\bar{r}_0r_1	q_1	\bar{s}_0s_1	q_1
r_0r_1	q_0	s_0s_1	q_0
r_0r_1	q_1	s_0s_1	q_1

Ce qui correspond à la machine suivante :

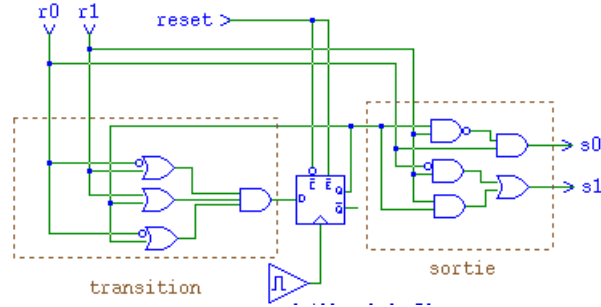


Les tables de vérités correspondantes sont les suivantes :

r_0	r_1	Q	s_0	s_1	Q'
0	0	0	0	0	0
0	0	1	0	0	1
1	0	0	1	0	0
1	0	1	1	0	0
0	1	0	0	1	1
0	1	1	0	1	1
1	1	0	1	0	0
1	1	1	0	1	1

$$s_0 = r_0(\bar{r}_1 + \bar{Q}) = r_0\bar{r}_1\bar{Q} \quad s_1 = \bar{r}_0r_1 + r_1Q \quad Q' = \bar{r}_0r_1 + r_1Q + \bar{r}_0Q.$$

Et le circuit est alors :



4.4 Comparaison de Modèles

Théorème 4.11 *Pour toute machine de Moore $\mathcal{M} = (Q, q_0, A, S, \delta, \sigma)$, il existe une machine de Mealy $\mathcal{M}' = (Q', q'_0, A', S', \delta', \sigma')$ qui possède les mêmes fonctionnements.*

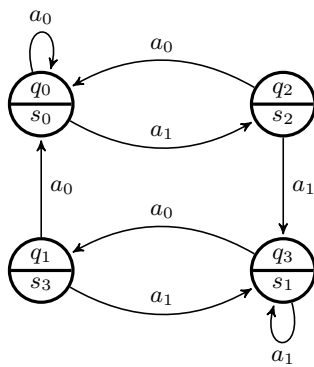
PREUVE. On pose :

- $A' = A$ et $S' = S$,
- $Q' = Q$ et $q'_0 = q_0$,
- $\delta' = \delta$
- pour tout $q \in Q$, $a \in A$, $\sigma'(q, a) = \sigma(\delta(q, a))$

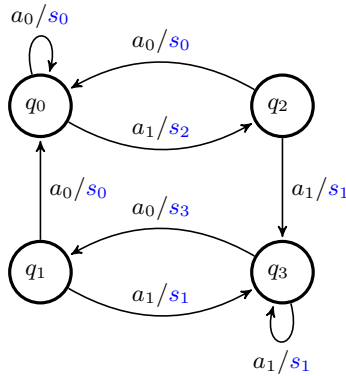
On prouve que tout fonctionnement de l'une est un fonctionnement de l'autre par récurrence sur la longueur des fonctionnements.

□

Le schéma suivant illustre cette construction :



Machine de Moore



Machine de Mealy ayant le même fonctionnement

Théorème 4.12 *Pour toute machine de Mealy $\mathcal{M} = (Q, q_0, A, S, \delta, \sigma)$, il existe une machine de Moore $\mathcal{M}' = (Q', q'_0, A', S', \delta', \sigma')$ qui possède les mêmes fonctionnements.*

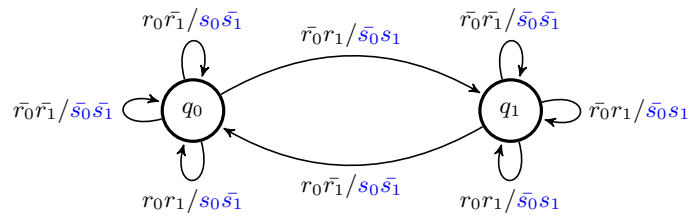
PREUVE. On pose :

- $A' = A$ et $S' = S$,
- $Q' = Q \times S$
- $q'_0 = (q_0, s_0)$,
- pour tout $(q, s) \in Q'$, $a \in A$, $\delta'((q, s), a) = (\delta(q, a), \sigma(q, a))$
- pour tout $(q, s) \in Q'$, $a \in A$, $\sigma'(q, s) = s$

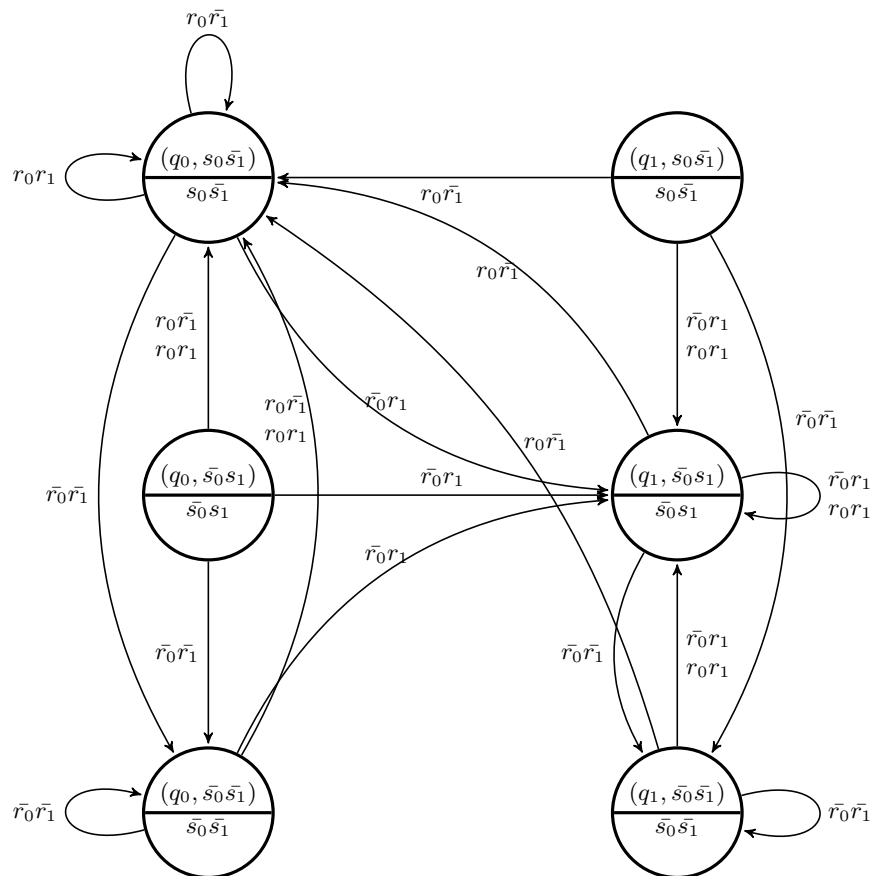
On prouve que tout fonctionnement de l'une est un fonctionnement de l'autre par récurrence sur la longueur des fonctionnements.

□

Par exemple, on peut transformer la machine du circuit d'arbitrage en machine de Moore :

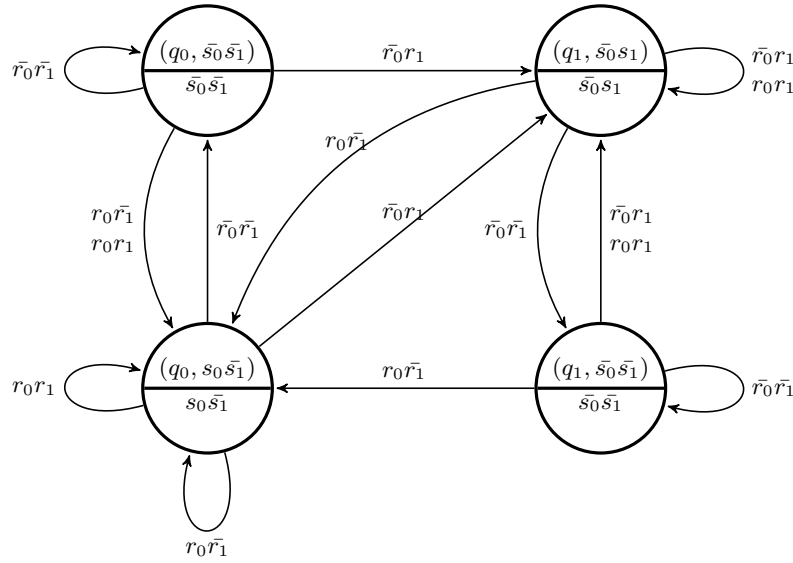


Machine de Mealy



Machine de Moore correspondante

On remarque que certains états ne peuvent jamais être atteints, on peut donc les retirer et on obtient la chine simplifiée suivante :



Machine de Moore simplifiée

4.4.1 Microprogrammation

La microprogrammation est une alternative à la solution câblée pour l'implantation de machines de Mealy/Moore inventée par Maurice Wilkes en 1951. On distingue 2 types de microprogrammation : la microprogrammation horizontale et la microprogrammation verticale. Nous considérons ici la microprogrammation horizontale.

L'idée est de remplacer un circuit combinatoire par une ROM dont l'ensemble des paires (adresse, contenu) code la table de vérité du circuit.

Une ROM (Read Only Memory) est un tableau mémoire à données persistantes. Les informations sont stockées au moment de la conception du circuit. Leur contenu est fixé et conservé en permanence même hors alimentation électrique.

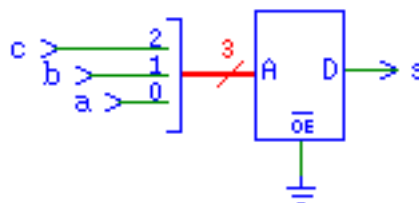
Une ROM dispose d'une entrée A (pour Adress) et d'une sortie D (pour Data). La sortie D fournit la donnée stockée dans la ROM à l'adresse A . Le nombre de bits de A détermine donc le nombre de cases mémoires (le nombre maximal de données dans la ROM). Si A est codée sur n bits, la capacité de stockage est de 2^n mots. Le nombre de bits de D détermine la taille des données stockées. Si D est codée sur m bits, chaque case de la ROM contient un mot de taille m .

Exemple 4.13 La fonction booléenne $s = \bar{c} + ab$ admet la table de vérité suivante :

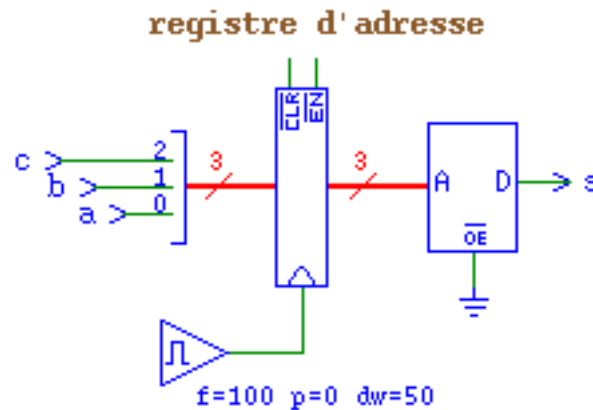
a	b	c	s
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

On peut réaliser cette fonction par un circuit combinatoire, ou bien simplement en mettant sa table dans une ROM de la façon suivante :

Adress	Data
0	1
1	0
2	1
3	0
4	1
5	0
6	1
7	1



Si on souhaite synchroniser les sorties sur une horloge, il suffira de stocker l'adresse dans un registre (on l'appelle alors le **registre d'adresse**) comme dans le schéma ci-dessous :



Conclusion Sauf contraintes de type physique spécifiées dans le cahier des charges, tout problème peut être résolu par une machine de type Mealy ou Moore, indifféremment.

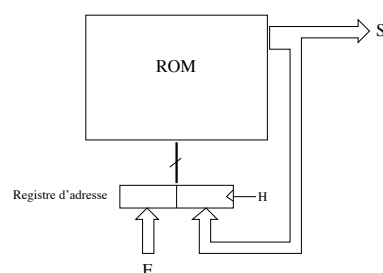
Chacune a ses avantages et ses inconvénients :

- en général, une machine de Moore demande plus d'états que la machine de Mealy équivalente,
- les sorties d'une machine Mealy ne sont pas synchronisées avec l'horloge. Ainsi, les entrées peuvent parfois agir sur les sorties *sans provoquer de changement d'états*, puisque ceux-ci sont synchrones. On a alors des brèves sorties parasites.

4.4.2 Réalisation microprogrammée d'une machine de Mealy

Toute machine de Mealy peut être réalisée par circuit comme celui-ci :

Les adresses seront présentes dans un "Registre d'adresse" (un simple registre utilisé pour stocker l'adresse de la prochaine donnée à lire). Cette adresse sera composée pour partie des entrées, le reste sera une partie de la dernière donnée lue (l'état suivant). A l'adresse correspondante dans la ROM sont stockés le prochain état et la sortie. Les sorties sont directement extraites des données.



Voici une réalisation d'un tel circuit.

Pour l'écriture dans la ROM dans TkGate, on utilise un fichier texte contenant les données de la ROM. Le fichier doit avoir une extension `.mem` et être chargé dans la ROM au moment de la simulation (voir la doc ici : <http://pageperso.lif.univ-mrs.fr/~peter.niebert/archi/tkgate-doc/gateSim.html#memory>).

Les adresses et données doivent être codées en hexadécimal. Les adresses sont précédées du symbole `@`.

Voici le fichier mémoire du circuit d'arbitrage : on indique les contenus des cases 0 à 7 à partir de l'adresse 0 :

`@0`

`0 1 4 4 3 3 4 3`

4.4.3 Logique câblée ou micro-programmée ?

Le choix dépend de plusieurs critères :

- Complexité/coût : la taille du circuit câblé augmente avec la complexité du problème. Le micro-programme nécessite des composants coûteux, mais sa taille n'augmente que peu en fonction de la complexité du problème.
- Evolutivité : En logique câblée, la moindre modification entraîne la mise au point d'un nouveau circuit, alors qu'en logique programmée, on peut souvent se contenter d'une modification du programme.
- Rapidité : les circuits câblés sont plus rapides.

Chapitre 5

Assembleur

5.1 Programmation machine

5.1.1 Langage machine et langage d'assemblage

Le langage machine est la suite de bits qui est interprétée par le processeur d'un ordinateur exécutant un programme informatique. C'est le langage natif d'un processeur, c'est-à-dire le seul qu'il puisse traiter. Il est composé d'instructions et de données à traiter codées en binaire. Chaque instruction correspond à un nombre (codé selon le cas sur un octet, un mot de 16 bits, ... : le **format de l'instruction**) et se décompose en

- une partie codant l'opération à exécuter appelée **opcode** ou **code opération**
- une partie pour les opérandes

Code op	Operandes
---------	-----------

Notez que les opérandes peuvent être spécifiées de différentes manières (on parle de **mode d'adressage**). Citons par exemple l'*adressage par registres* : **Operandes** contient le(s) numéro(s) du (des) registre(s) où se trouvent les données manipulées par l'instruction ; l'*adressage direct (ou direct restreint)* : **Operandes** est l'adresse (ou un fragment de l'adresse) où se trouve la donnée en mémoire ; ou encore l' *adressage immédiat* : **Operandes** est la valeur utilisée par l'instruction.

Un programme en langage machine est donc une suite de mots codant opérations et opérandes. Si vous ouvrez un fichier exécutable avec un afficheur hexadécimal, vous verrez un fichier du type de celui de la figure 5.1

```
01ebe814063727473747566662e6305f5f43544f525f4c
5f05f5f44544f525f4c4953545f5f05f5f4a43525f4c49
53545f5f05f5f646f5f676c6f62616c5f64746f72735f6
75780636f6d706c657465642e36353331064746f725f69
```

FIGURE 5.1 – Extrait d'un fichier exécutable (processeur MIPS)

Ce code machine est rendu compréhensible par un codage direct en **langage d'assemblage**. Le langage d'assemblage est un langage de bas niveau qui représente le langage machine sous une forme lisible par un humain. Les combinaisons de bits du langage machine sont représentées par des symboles faciles à retenir. Le programme assembleur convertit ces symboles en langage machine en vue de créer un fichier exécutable.

Par exemple, dans la figure 5.1, la première instruction est codée par le mot **01ebe814** qui se traduit en assembleur par **add \$t7, \$t3, \$sp**. Nous verrons la signification de cette commande dans la suite.

5.1.2 Fonctionnement général

D'un point de vue de la programmation, le processeur offre

- un certain **jeu d'instructions** qu'il sait exécuter.
- un certain nombre de **registres** :

- utilisables/modifiables directement par le programme : registres de travail - pointeur de segment
il s'agit de registres vus par le jeu d'instructions
- modifiables indirectement par le programme : compteur ordinal - pointeur de pile - registre d'instruction - registre d'états
ces registres sont manipulés implicitement par le jeu d'instructions
- d'un tas mémoire permettant entre autre de stocker le programme à exécuter et les données.

Voici les différentes étapes de l'exécution d'une instruction.

1. **Récupérer (en mémoire) l'instruction à exécuter** : $RI \leftarrow \text{Mémoire}[PC]$
L'instruction à exécuter est présente en mémoire à l'adresse contenue dans le compteur de programme PC et est placée dans le registre d'instruction RI .
2. **Le compteur de programme est incrémenté** : $PC \leftarrow PC + 4$
Par défaut, la prochaine instruction à exécuter sera la suivante en mémoire (sauf si l'instruction est un saut). Pour cet exemple nous avons choisi $PC + 4$, on suppose donc que les instructions sont codées sur 4 octets.
3. **L'instruction est décodée** : On identifie les opérations qui vont devoir être réalisées pour exécuter l'instruction.
4. **L'instruction est exécutée** : elle peut modifier les registres (opérations arithmétiques - lecture en mémoire), la mémoire (écriture), le registre PC (instructions de saut).

5.1.3 Les deux principales architectures processeur

Le **jeu d'instructions** est l'ensemble des opérations élémentaires qu'un processeur peut accomplir. Le type de jeu d'instructions d'un processeur détermine son architecture. On distingue deux types d'architectures :

- RISC (Reduced Instruction Set Computer)
PowerPC, MIPS, Sparc
 - jeu d'instructions de taille limitée
 - instructions simples
 - format des instructions petit et fixé
 - modes d'adressage réduits
- CISC (Complex Instruction Set Computer)
Pentium
 - jeu d'instructions de taille importante
 - instructions pouvant être complexes
 - format d'instructions variables (de 1 à 5 mots)
 - modes d'adressages complexes.

Ces deux architectures ont leurs avantages et leurs inconvénients :

- CISC
 - ⊕ programmation de plus haut niveau, plus compacte (écriture plus rapide et plus élégante des applications), et moins d'occupation en mémoire et à l'exécution ;
 - ⊖ complexifie le processeur, taille des instructions élevée et variable : pas de structure fixe, exécution des instructions complexe et peu performante.
- RISC
 - ⊕ instructions de format standard, traitement plus efficace, possibilité de pipeline plus efficace ;
 - ⊖ programmes plus volumineux et compilation plus compliquée.

5.2 Assembleur MIPS

5.2.1 Le processeur MIPS R2000

Nous détaillons ici les caractéristiques du processeur MIPS R2000, donc nous étudierons ensuite le langage d'assemblage. Ce processeur de type RISC est utilisé par exemple par NEC, SGI, Sony PSP, PS2. Le langage assembleur MIPS est proche des autres assembleurs RISC.

Il s'agit d'un processeur 32 bits (taille des registres) constitué de

- 32 registres de 32 bits
- une mémoire vive adressable de 2^{32} octets
- un compteur de programmes PC (Program Counter) sur 32 bits
- un registre d'instruction RI sur 32 bits

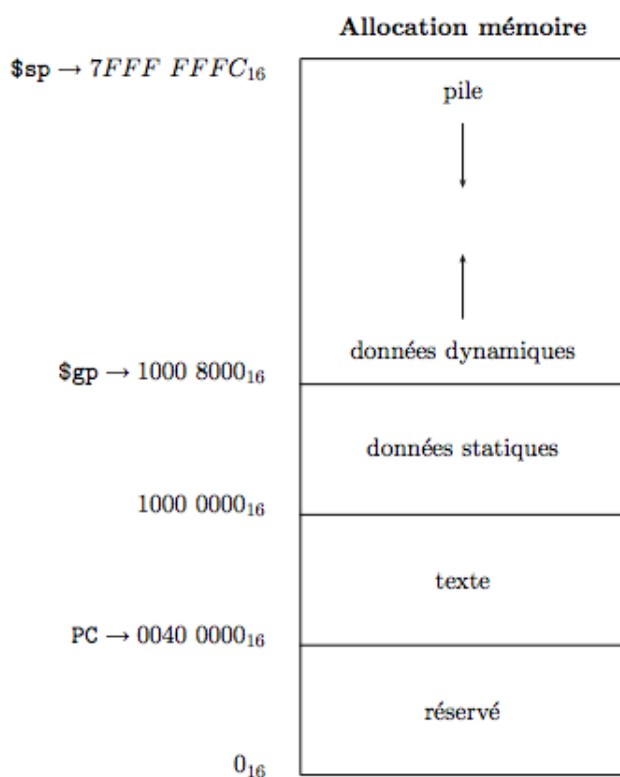
Le programme est stocké en mémoire, l'adresse de l'instruction en cours d'exécution est stockée dans le registre *PC* et l'instruction en cours d'exécution est stockée dans le registre *RI*. Ceci implique en particulier que chaque instruction est codée sur 32 bits.

5.2.1.1 Mémoire

La mémoire est un grand tableau dont les indices sont les adresses. Pour des raisons historiques, les adresses sont mesurées en octets (8 bits). Cependant, les lectures et écritures en mémoire se font sur des quantités de mémoire plus importantes que l'octet. Le mot est la taille des données que l'on peut transférer en une instruction : typiquement 32 ou 64 bits, selon les processeurs. Le mot est également la taille des adresses et des registres.

Mémoire de 2^{32} octets = 2^{30} mots de 32 bits. Les mots mémoires sont donc adressés par des adresses qui sont des multiples de 4.

- bus d'adresses de 32 bits
- bus de données de 8 bits



5.2.1.2 Registres MIPS

Le processeur MIPS contient 32 registres. Hormis le registre \$0, appelé \$zero, qui contient toujours la valeur 0, même après une écriture, les autres registres sont interchangeables, mais pour garantir l'interopérabilité entre programmes assembleurs produits par des compilateurs différents, on a fixé des conventions d'usage qui sont détaillées comme suit.

- Les registres \$a0...\$a3 sont utilisés pour passer les premiers 4 paramètres d'une fonction lors d'un appel.
- Les registres \$v0 et \$v1 sont utilisés pour le renvoi de résultats.
- Lors de l'appel de programme, les registres \$s0-\$s7 sont considérés comme sauvegardés par le programme appelant et les registres \$t0-\$t9 comme non sauvegardés.
- Les pointeurs \$sp, \$fp contiennent les pointeurs vers la pile, et \$gp des pointeurs vers les données.
- Les registres \$k0-\$k1 sont réservés par le noyau et \$at par l'assembleur.
- Enfin, le registre \$ra contient l'adresse de retour après l'appel à une fonction.

Voici le récapitulatif du nom des registres, de leur code et de leur description.

Nom	Numéro	Description
\$zero	0	constante 0
\$at	1	réservé à l'assembleur
\$v0,\$v1	2-3	résultats d'évaluation
\$a0,...,\$a3	4-7	arguments de procédure
\$t0,...,\$t7	8-15	valeurs temporaires
\$s0,...,\$s7	16-23	sauvegardes
\$t8,\$t9	24-25	temporaires
\$k0,\$k1	26-27	réservé pour les interruptions
\$gp	28	pointeur global
\$sp	29	pointeur de pile
\$fp	30	pointeur de bloc
\$ra	31	adresse de retour

5.2.2 Structure d'un programme assembleur MIPS

Commençons par détailler les différents éléments de structuration d'un programme MIPS :

- **Commentaires** : les commentaires sont tout ce qui suit sur une ligne le symbole #.
- **Labels** : les labels servent à référencer certaines parties du programme. Ils servent juste à donner un nom à une ligne du programme. Ceci permet de faire des saut dans le programme, et par suite, des boucles. Les labels n'apparaissent pas dans le code final, ils servent juste à faciliter l'écriture. Pour créer un label nommé `lab`, il suffit d'écrire `lab:` devant la ligne à nommer
- **Sections** : un programme est divisé en 2 types de sections pouvant s'entremêler : Les sections de données, qui débutent par `.data`, et les sections d'instructions qui débutent par `.text`.
- **Directives** : Les directives permettent de spécifier quelle section de la mémoire est concernée par les instructions, macro-instructions ou directives qui les suivent. Ainsi, les délimiteurs de section `.data` et `.text` définis ci-dessus sont des directives. Vous pouvez trouver les autres directives dans le manuel de référence, citons par exemple :

```
str_name: .asciiz "ma_chaine"           Met la chaîne ma_chaine en mémoire
array_name: .word w1,...,wn           Mets les  $n$  valeurs sur 32 bits dans des mots successifs.
```

5.2.3 Instructions assembleur MIPS

Nous donnons ici les principales instructions assembleur. Vous pouvez vous référer à la documentation MIPS pour avoir l'ensemble des instructions. Notez que certaines des instructions présentées sont en fait des *pseudo-instructions*, c'est-à-dire des instructions qui n'existent pas réellement dans le jeu d'instructions machine, mais que l'assembleur traduira en langage machine par une suite d'instructions.

5.2.3.1 Arithmétique

Le jeu d'instruction MIPS contient toutes les opérations arithmétique et booléennes de base. Prenons l'exemple de l'addition.

Code C	Assembleur
<code>s0 = s1 + s2</code>	<code>add \$s0, \$s1, \$s2</code>

Pour les instructions arithmétiques, le résultat est toujours placé dans la première opérande. Remarquez que dans notre exemple, toutes les opérandes se trouvent dans des registres, mais ce ne sera pas toujours le cas. L'instruction `add` (comme la plupart des autres instructions arithmétiques) admet différentes déclinaisons que nous détaillons ici :

- `add $t1,$t2, $t3`
Addition avec débordement (levée d'exception) : copie dans le registre `$t1` le résultat de la somme du contenu du registre `$t2` avec le contenu du registre `$t3`
- `add.d $f2, $f4, $f6`
Addition flottants double precision
- `add.s $f0, $f1, $f3`
Addition flottants simple precision
- `addi $t1,$t2, i`
Addition du contenu d'un registre avec un immédiat (i.e., un nombre donné directement) avec débordement. L'immédiat est ici un nombre signé codé sur 16 bits : `$t1 = $t2 + i`
- `addu $t1, $t2, $t3`
Addition non signée sans débordement.

— `addiu $t1,$t2, i`

Addition non signée et sans débordement d'un registre et d'un immédiat signé codé sur 16 bits.

Notez que la division entière peut être réalisée par la pseudo-instruction `div $t1, $t2, $t3` réalisant l'opération $\$t1 = \$t2 / \$t3$. Si on souhaite récupérer le reste de la division, on utilisera plutôt l'instruction `div $t2, $t3` qui place le résultat de la division dans le registre `lo` et le reste dans le registre `hi`. Ces deux registres ne font pas partie des 32 registres mentionnés précédemment, il ne peuvent pas être utilisés directement comme opérandes d'instructions mais leur contenu est accessible via les instructions suivantes : `mfhi $t1` copie le contenu de `hi` dans le registre `$t1` et `mflo $t1` copie le contenu de `lo` dans le registre `$t1`.

5.2.3.2 Ecriture dans un registre

L'affectation d'un registre par le contenu d'un autre registre est réalisée par la pseudo-instruction `move`.

Assembleur	Effet
<code>move \$s0, \$s1</code>	copie le contenu de <code>\$s1</code> dans le registre <code>\$s0</code>

Notez que cette instruction est traduite par l'assembleur comme l'instruction `add $s0, $0, $s1`. Le registre `$0` est un registre qui ne peut pas être modifié et dont le contenu est toujours 0.

L'affectation d'un registre par un immédiat est réalisée par la pseudo-instruction `li` (pour load immediate).

Assembleur	Effet
<code>li r, imm</code>	charge la valeur <code>imm</code> (sur 32 bits) dans le registre <code>r</code>

Cela ne peut pas correspondre à une instruction réelle car elle ne tiendrait pas sur 32 bits ! Cette instruction est (selon l'assembleur) assemblée comme ceci :

```
lui r, immh
ori r, imm1
```

où

- `immh,imm1` sont respectivement les 16 bits de poids fort et de poids faible de `imm`,
- `lui r, i`, où `i` est un immédiat sur 16 bit, est l'instruction qui charge $i \times 2^{16}$ dans `r` (par exemple `lui $t0, 0xFF14` revient à l'affectation `t0=0xFF14 << 16`),
- `ori r, imm1` : réalise un "ou logique" entre les 16 bits de `imm1` (étendus à 32 bits en mettant ceux de poids fort à 0) et le contenu de `r`, le résultat étant placé dans `r`.

5.2.3.3 Lecture-Ecriture dans la mémoire principale

Les deux instructions `lw` (load word = lecture) et `sw` (store word = écriture) permettent les échanges entre la mémoire centrale et les registres.

syntaxe :

`lw $2, 10($3)` copie dans le registre `$2` la valeur située dans la mémoire principale à l'adresse obtenue en ajoutant 10 au nombre stocké dans le registre `$3`.

`sw $2, -15($1)` copie la valeur présente dans le registre `$2` dans la mémoire principale à l'adresse obtenue en soustrayant 15 au nombre stocké dans le registre `$1`.

5.2.3.4 Les branchements

Les branchements sont les instructions qui permettent de casser la linéarité d'un programme en permettant d'exécuter une partie d'un programme, plutôt qu'une autre et de créer des boucles. Un branchement est toujours relatif à un endroit du programme (l'endroit où on veut « brancher ») référencé par un label ou une adresse.

Par soucis d'efficacité, avant d'effectuer le branchement, le processeur réalisera l'instruction (instruction machine!! attention donc aux pseudo-instructions) qui suit linéairement dans le programme. Ceci est du au fait de l'architecture pipeline du processeur : pendant qu'une instruction est effectuée, la suivante est chargée et décodée. Ne pas réaliser l'instruction déjà chargée ferait perdre du temps (le temps de charger, décoder et exécuter l'instruction référencée), le choix a donc été fait de l'exécuter par défaut. Il revient donc

au programmeur de tenir compte de cette spécificité. En général, on essaye d'exploiter cette instruction supplémentaire, si ce n'est pas possible, on peut toujours utiliser l'instruction `nop` qui ne fait rien (mais coûte tout de même un cycle). Plusieurs exemples décrivant ce mécanisme (appelé délai de branchement) sont donnés ci-après.

Notez que pour des raisons pédagogiques, la plupart des simulateurs disposent d'une option permettant de supprimer ce délai. Pour ces mêmes raisons, la plupart des exemples de programmes assembleurs que vous pouvez trouver sur internet (dans des cours principalement) le tiennent pas compte du délai de branchement.

On distingue deux types de branchements : le branchement conditionnel est effectué si une condition est réalisée ; le branchement inconditionnel est réalisé sans condition.

Les branchements conditionnels Il existe de nombreuses instructions de branchement conditionnels, permettant d'exprimer différentes conditions et permettant de brancher vers un endroit du programme référencé par un label. Nous en décrivons quelques unes ici, elles fonctionnent toutes sur le même mode.

- `bne $t0, $t1, Label` (branch if not equal)
Si la valeur contenue dans le registre `$t0` n'est pas égale à celle stockée dans le registre `$t1` alors la prochaine instruction à exécuter est celle placée après l'étiquette `Label` ;
- `beq $t0, $t1, Label` (branch if equal)
Si la valeur contenue dans le registre `$t0` est égale à celle stockée dans le registre `$t1` alors la prochaine instruction à exécuter est celle placée après l'étiquette `Label` ;
- `beqz $t0, Label` (branch if equal zero) (pseudo-instruction)
Si la valeur contenue dans le registre `$t0` est égale à 0, alors la prochaine instruction à exécuter est celle placée après l'étiquette `Label` ;
- `ble $t0, $t1, Label` (branch if less or equal) (pseudo-instruction)
Si la valeur contenue dans le registre `$t0` est inférieur ou égale à celle stockée dans le registre `$t1` alors la prochaine instruction à exécuter est celle placée après l'étiquette `Label`.

Voici quelques exemples : les deux derniers montrent l'importance de bien connaître les pseudo-instructions.

Assembleur	Code C
<pre>beqz \$t0, suite addi \$t1, \$t1, 12 addi \$t2, \$t2, 13 suite: addi \$t6, \$t6, 14</pre>	<pre>t1 = t1 + 12 if (t0!=0) {t2 = t2 + 13} t6 = t6 + 14</pre>

Dans ce premier exemple, l'instruction `t1 = t1 + 12` sera effectuée dans tous les cas à cause du délai de branchement. Comme `$t1` n'influe pas sur la condition de branchement, cela revient à l'exécuter avant le `if`.

Assembleur	Code C
<pre>beqz \$t0, suite addi \$t0, \$t1, 12 addi \$t2, \$t2, 13 suite: addi \$t6, \$t6, 14</pre>	<pre>t0 = t1 + 12 if (t0!=12) {t2 = t2 + 13} t6 = t6 + 14</pre>

Dans ce nouvel exemple, l'instruction `t0 = t1 + 12` sera effectuée dans tous les cas à cause du délai de branchement. Cette fois, `$t0` influe sur la condition de branchement, si on souhaite affecter `$t0` avant le `if`, il faut donc inclure la modification faite sur `$t0` dans la condition du `if`.

Assembleur	MIPS pur	Code C
<pre>beqz \$t0, suite li \$t2, 0x3BF20 suite: addi \$t6, \$t2, 10</pre>	<pre>beqz \$t0, suite lui \$t2, 0x0003 ori \$t2, \$t2, 0xBF20 suite: addi \$t6, \$t2, 10</pre>	<pre>if (t0!=0) {t2 = 0x3BF20} t2 = 0x00030000 t6 = t2 + 10</pre>

Dans cet exemple, il faut tenir compte du fait que `li $t2, 0x3BF20` est une pseudo instruction. L'instruction exécutée durant le délai de branchement sera donc `lui $t2, 0x0003`, ce qui crée un comportement certainement inattendu.

Assembleur	MIPS pur	Code C
<pre> beqz \$t0, suite subi \$t1, \$t1, 1 suite: addi \$t6, \$t2, 10 </pre>	<pre> beqz \$t0, suite addi \$at, \$0, 1 sub \$t1, \$t1, \$at suite: addi \$t6, \$t2, 10 </pre>	<pre> if (t0!=0) {t1 = t1 - 1} t6 = t2 + 10 </pre>

Dans cet exemple, l'instruction supplémentaire générée par le délai de branchement est `addi $at, $0, 1`, elle affecte le registre `$at` réservé à l'assemblage et n'a donc pas d'effet sur l'exécution.

Branchements inconditionnels Voici 3 exemples de branchements inconditionnels :

- `j Label` (jump)
La prochaine instruction à exécuter est celle placée après l'étiquette `Label` : $PC \leftarrow \text{Label}$.
- `jr r` (jump register)
L'adresse de la prochaine instruction à exécuter est dans le registre `r` : $PC \leftarrow r$.
- `jal Label` (jump and link register) Même effet que `j Label`, mais stock également l'adresse de l'instruction suivant l'instruction courante pour pouvoir y revenir ultérieurement. L'adresse de retour est stockée dans le registre d'adresse `$ra` : $\$ra \leftarrow PC, PC \leftarrow \text{Label}$.

En utilisant ces instructions, on peut effectuer des branchements plus complexes que le `if ... then`. Voici par exemple comment programmer un `if ... then ... else`.

Assembleur	Code C	Code C (simplifié)
<pre> bl \$t0, \$t2, else add \$t0, \$t0, \$t1 sub \$t2, \$t2, \$t0 j endif else: addi \$t2, \$t2, 1 add \$t2, \$t2, \$t0 endif: </pre>	<pre> if (t0 >= t2) { t0 = t0 + t1 t2 = t2 - t0 t2 = t2 + 1 } else { t0 = t0 + t1 t2 = t2 + 1 t2 = t2 + t0 } </pre>	<pre> t0 = t0 + t1 if (t0 - t1 >= t2) t2 = t2 - t0 else t2 = t2 + t0 t2 = t2 + 1 </pre>

On peut également programmer des boucles. Voici l'exemple d'une boucle `while`.

Assembleur	Code C
<pre> li \$t2, 0 while:beq \$t1, \$0, done nop add \$t2, \$t1, \$t2 subi \$t1, \$t1, 1 j while done: nop </pre>	<pre> t2=0 while (t1 != 0){ t2 = t2 + t1 t1=t1-1 } </pre>

Remarquez que du fait du délai de branchement, il peut-être plus efficace de programmer des boucles `do ... while`, comme l'illustre l'exemple suivant :

Assembleur	Code C
<pre> li \$t2, 0 do: add \$t2, \$t1, \$t2 bnez \$t1, do subi \$t1, \$t1, 1 </pre>	<pre> t2 = 0 do { t2 = t2 + t1 t1 = t1 - 1 } while (t1 != 0) </pre>

5.2.3.5 Les appels systèmes (syscall)

Les simulateurs fournissent un ensemble de services par l'intermédiaire de l'appel `syscall`. Pour demander un service, on charge le code du service (voir tableau 5.2) dans le registre `$v0` et ses arguments dans les registres `$a0, $a1` (ou `$f12` pour les valeurs flottantes). Les appels système qui retournent une valeur placent le résultat dans `$v0` (ou `$f0` pour les valeurs flottantes).

Par exemple, le code suivant imprime : `la réponse = 5` .

service	code	arguments	résultat
<code>print_in</code>	1	<code>\$a0</code> entier à afficher	
<code>print_float</code>	2	<code>\$f12</code> flottant à afficher	
<code>print_double</code>	3	<code>\$f12</code> double à afficher	
<code>print_string print_string</code>	4	<code>\$a0</code> chaîne à afficher	
<code>read_int</code>	5		un entier <code>\$v0</code>
<code>read_float</code>	6		un flottant <code>\$f0</code>
<code>read_double</code>	7		un double <code>\$f0</code>
<code>read_string</code>	8	<code>\$a0</code> adresse, <code>\$a1</code> longueur	
<code>sbrk</code>	9	<code>\$a0</code> longueur à allouer dans le tas	<code>\$v0</code> adresse
<code>exit</code>	10		

FIGURE 5.2 – Les appels système

```

.data
str: .asciiz "la réponse = "
.text

main: li $v0,4      #code appel système print_str
      la $a0,str    #adresse chaîne à imprimer
      syscall      #on imprime la chaîne
      li $v0,1      #code appel système print_int
      li $a0,5      #entier à imprimer
      syscall      #on l'imprime

```

5.2.4 Appel de procédures

Pour appeler une procédure, il faut sauver l'adresse de retour à l'appel général, en général dans le registre `$ra`, pour revenir après l'appel. A la fin d'une procédure, on retourne à l'appelant en sautant à l'adresse contenue dans `$ra`.

L'instruction `jal SP` permet d'exécuter la procédure de label `SP`, la sauvegarde de l'adresse de retour dans `$ra` étant réalisée automatiquement par cette instruction.

```

jal SP    # appel SP
....

```

```

SP: ....
....
....
....
jr $31    # return

```

Cependant,

- Le sous-programme peut affecter les valeurs contenues dans les registres au moment de l'appel : pas de notion de variables locales et de portée/masquage de variables.
- La sauvegarde de l'adresse de retour dans un registre ne permet pas l'enchaînement des appels à des sous-programmes, encore moins des sous-programmes récursifs.

Solution :

- Sauvegarder la valeur des registres (en mémoire) de l'appelant et restaurer ces valeurs à l'issue de l'appel
- Sauvegarder l'adresses de retour du programme appelant en mémoire

On sauvegarde les (une partie des) registres en mémoire dans **une pile**.

Gestion de la pile Une pile est une mémoire qui se manipule via deux opérations :

- `push` : empiler un élément (le contenu d'un registre) au sommet de la pile
- `pop` : dépiler un élément (et le récupérer dans un registre)

Ces deux instructions n'existent pas en assembleur MIPS, mais elles peuvent être "simulées" en utilisant les instructions `sw` et `lw`.

Par convention, la pile grossit vers les adresses décroissantes. Le registre `$sp` pointe vers le sommet de pile (le pointeur de pile).

Pour sauver un registre r sur la pile (empilement) :

```
sub $sp, $sp, 4    # alloue un mot sur la pile
sw r, 0($sp)      # écrit r sur le sommet de pile
```

Pour restaurer un mot de la pile dans le registre r (défilement)

```
lw r, 0($sp)      # lit le sommet de pile dans r
add $sp, $sp, 4    # désalloue un mot sur la pile
```

Politiques de gestion de la pile Deux politiques de sauvegarde des registres :

- *sauvegarde par l'appelant* : le programme appelant sauvegarde tous les registres sur la pile (avant l'appel).
- *sauvegarde par l'appelé* : le programme appelant suppose que tous les registres seront préservés par le sous-programme appelé.

Quelque soit la politique utilisée,

un sous-programme doit rendre la pile intacte

Conventions d'appels En général :

- les arguments sont passés dans les registres `a0` à `a3` puis dans la pile.
- la ou les valeurs de retour dans `v0` et `v1`.

De même :

- les registres t_i sont sauvés par l'appelant (l'appelé peut les écraser)
- les registres s_i sont sauvés par l'appelé (qui doit les remettre dans l'état où il les a pris)

Mais ce n'est qu'une convention ! (celle proposée par le fabriquant) La respecter permet de communiquer avec d'autres programmes (qui la respectent également). Choisir une autre convention est possible tant que l'on n'interagit pas avec le monde extérieur.

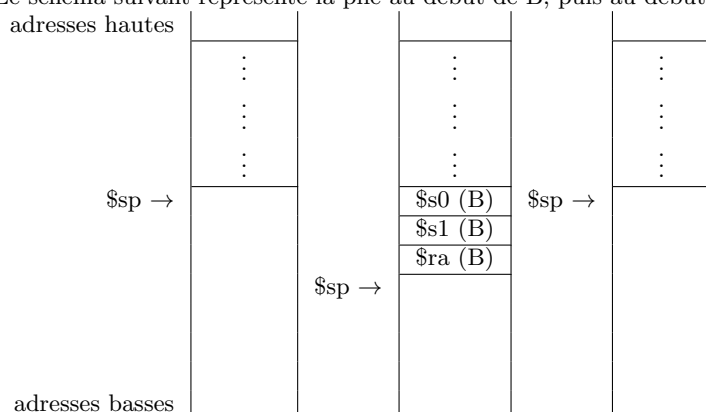
Exemple

On a deux sous-programmes B et C, le programme B appelle le programme C. Avant d'appeler C on souhaite sauvegarder les valeurs des registres `$t0`, `$t1`, et l'emplacement où se fera le retour de B dans le programme principal, stocké dans le registre `$ra`. Initialement, le sommet de la pile est pointé par `$sp`. On sauvegarde d'abord `$t0` à cet emplacement de la pile, puis `$t1` à l'emplacement suivant (donné par `$sp-4`) et enfin `$ra` est stocké en `$sp-8`. Avant d'appeler C, on place correctement le pointeur `$sp` pour qu'il puisse être utilisé par C, on le place au prochain emplacement libre (soit `$sp-12`). On appelle ensuite le sous-programme C (instruction `jal C`). Après exécution de C, le pointeur `$sp` est identique à avant l'appel (sinon le programme est incorrect). On restaure donc facilement les paramètres sauvegardés dans la pile en chargeant les valeurs contenues dans `$sp+4` pour `$ra`, `$sp+8` pour `$s1`, et `$sp+12` pour `$s0`. Enfin, avant de terminer B, on remet le pointeur `$sp` à sa position initiale `$sp+12`.

Voici un fragment du sous-programme B :

B	...	
...	...	debut de B
...	...	
	<code>sw \$t0,0(\$sp)</code>	sauvegarde de <code>\$t0</code> dans la pile
	<code>sw \$t1,-4(\$sp)</code>	sauvegarde de <code>\$t1</code> dans la pile
	<code>sw \$ra,-8(\$sp)</code>	sauvegarde de l'adresse de retour de B
	<code>subi \$sp,\$sp,12</code>	ajustement du sommet de pile
	<code>jal C</code>	appel du sous-programme C
	<code>lw \$ra,4(\$sp)</code>	restauration de l'adresse de retour de B
	<code>lw \$s1,8(\$sp)</code>	restauration de <code>\$s1</code>
	<code>sw \$s0,12(\$sp)</code>	sauvegarde de <code>\$s0</code>
	<code>addi \$sp,\$sp,12</code>	ajustement du sommet de pile
	<code>jr \$ra</code>	
	...	fin de B

Le schéma suivant représente la pile au début de B, puis au début de C et enfin à la fin de B.



5.3 Format des instructions MIPS

Rappel : les instructions du langage machine MIPS sont codées sur **32 bits**

6 bits	26 bits
Code op	Opérandes

$2^6 = 64$ opérateurs possibles

Trois formats d'instructions :

- Instructions de type immédiat (Format I) utilisé pour les instructions spécifiant un opérande immédiat, les transferts mémoire, les branchements conditionnels
- Instructions de type saut (Format J) utilisé pour les sauts inconditionnels
- Instructions de type registre (Format R) utilisé pour coder les instructions à trois registres (comme les opérations arithmétiques par exemple)

Les 6 bits du Code op déterminent le format de l'instruction.

5.3.1 Format d'instructions I

6 bits	5bits	5bits	16 bits
Code op	rs	rt	immédiat/adresse

- *rs* : registre source
- *rt* : registre cible / condition de branchement
- *immédiat/adresse* : opérande immédiate ou déplacement d'adresse

Ce format a trois usages : il est utilisé pour les instructions de saut et de branchement conditionnel, pour certaines instructions arithmétiques, ainsi que pour les instructions de transfert de données.

Pour un branchement conditionnel, les deux registres désignés par *rt* et *rs* sont comparés et le résultat sert de condition au saut à l'adresse imm

Pour les instructions arithmétiques et logiques à deux opérandes : le registre désigné par *rs* et la constante (immédiat) sont les opérandes sources, le registre désigné par *rt* est la destination.

La dernière utilisation de ce format concerne les instructions de transfert de données (load/store). Le champ immédiat contient le déplacement alors que le champ *rs* désigne le registre de base de l'adresse de l'opérande à transférer. Le champ *rt* représente, pour sa part, le registre source ou destination suivant le sens de la transaction.

	<i>op</i>	<i>rs</i>	<i>rt</i>	16 bits
lui \$1, 100	15	0	1	100
lw \$1, 100(\$2)	35	2	1	100
sw \$1, 100(\$2)	43	2	1	100
beq \$1, \$2, 100	4	1	2	100
bne \$1, \$2, 100	5	1	2	100

- `lui r, imm` (load upper immediate) utilise les 16 bits de `imm` pour initialiser les 16 bits de poids fort du registre `r`, les 16 bits de poids faible étant mis à 0.
- pour un branchement conditionnel, l'adresse de branchement est la somme de l'adresse de l'instruction courante plus la valeur immédiate sur 16 bits, avec extension de signe, décalée à gauche de 2 bits (donc multipliée par 4). Cela signifie que les 16 bits codent en fait le nombre de lignes dans le programme qu'il faut sauter.

5.3.2 Format d'instructions J

6 bits	26 bits
Code op	adresse

Ce format est utilisé pour les sauts et les branchements sans condition. Le champ cible ne contient que 26 bits : après un décalage à gauche de deux positions, on le concatène aux 4 bits de poids forts du compteur ordinal pour obtenir les 32 bits de l'adresse absolue. Lors d'une instruction Jump And Link (`jal`), l'adresse de retour est automatiquement placée dans le registre `R31`.

	op	adresse 26 bits
<code>j 1000</code>	2	1000
<code>jal 1000</code>	3	1000

5.3.3 Format d'instructions R

6 bits	5bits	5bits	5bits	5bits	6bits
Code op	rs	rt	rd	sa	fu

La principale utilisation de ce format concerne les instructions arithmétiques à trois ou quatre opérandes (`rs` et `rt` sont les identificateurs des registres sources alors que `rd` désigne le registre destination). La norme MIPS IV inclut des instructions de multiplication-addition flottante dont le format est aussi de ce type. Dans le cas d'une telle instruction, `fr`, `fd`, `fs` sont des registres flottants source, alors que `fd` désigne le registre destination.

- `rs` : registre source 1
- `rt` : registre source 2
- `rd` : registre destination
- `sa` : nombre de décalage à effectuer (shift amount)
- `fu` : identificateur de la fonction

	op	rs	rt	rd	sa	fu
<code>add \$1,\$2,\$3</code>	0	2	3	1	0	32
<code>sub \$1,\$2,\$3</code>	0	2	3	1	0	34
<code>slt \$1,\$2,\$3</code>	0	2	3	1	0	42
<code>jr \$31</code>	0	31	0	0	0	8

- `sub $1,$2,$3` : soustrait `$3` de `$2` et place le résultat dans `$1`.
- `slt $1,$2,$3` (set less than) : met `$1` à 1 si `$2` est inférieur à `$3` et à 0 sinon.

5.3.4 Exemple

Code C	Assembleur MIPS
<pre>while (tab[i] == k) i = i+j;</pre>	<pre>Loop : mul \$9, \$19, \$10 lw \$8 , Tstart(\$9) bne \$8 , \$21, Exit add \$19, \$19, \$20 j Loop Exit :</pre>

avec `i` \mapsto `$19`, `j` \mapsto `$20`, `k` \mapsto `$21` et `$10` est initialisé à 4.
Programme chargé à l'adresse 80000 et `Tstart` vaut 1000

Adresse	Contenu					
80000	28	19	10	9	0	2
80004	35	9	8	1000		
80008	5	8	21	2		
80012	0	19	20	19	0	32
80016	5	20000				
80020					

car

- $\overbrace{80008 + 4}^{PC} + \textcolor{red}{2} * 4 = 8020$
- $\textcolor{blue}{20000} * 4 = 80000$

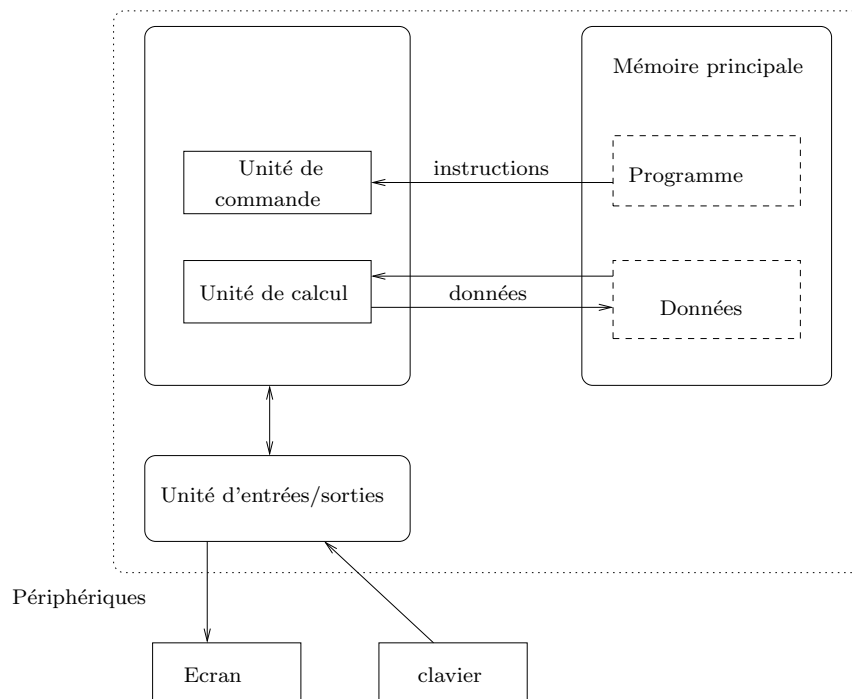
Chapitre 6

Processeur

Le processeur (ou CPU pour Central Processing Unit (Unité centrale de traitement)) est le composant qui exécute les instructions machine.

Il est essentiellement composé de :

- d'une unité de calcul, constituée de plusieurs unités : l'Unité Arithmétique et Logique (ALU), qui gère les calculs sur les entiers et opérations booléennes ; l' Unité de calcul flottant : (FPU - Floating Point Unit), qui effectue les calculs les flottants : sqrt, sin, ... ; l'Unité multimédia, qui gère le calcul vectoriel (même instruction sur plusieurs donnée en parallèle). Il n'est pas rare de trouver 3 ALU dans une unité de calcul. Ceci permet de faire 3 calculs en parallèle.
- d'une unité de contrôle (ou séquenceur) qui permet de synchroniser les différents éléments du processeur.
- d'unités de mémorisation : registres, mémoire cache (permet d'accélérer les traitements en diminuant les temps d'accès à la mémoire. Ces mémoires tampons sont en effet beaucoup plus rapides que la RAM et ralentissent moins le CPU.)
- d'une horloge synchronisant toutes les actions de l'unité centrale
- d'une unité d'entrée-sortie qui gère la communication avec les périphériques.
- des bus de communication : le bus de donnée détermine la taille des données pour les entrées-sorties (indépendant de la taille des registre), le bus d'adresse qui véhicule l'adresse d'une donnée à lire et à écrire (détermine donc la taille de la mémoire adressable), le bus de contrôle qui permet la gestion du matériel



6.1 Unité de contrôle/commande

Cette unité coordonne le fonctionnement des autres éléments pour exécuter la séquence d'instructions constituant le programme.

Cette unité est constituée :

- d'un ensemble de registres
 - registre d'instruction RI : permet de stocker l'instruction qui doit être exécutée
 - compteur programme PC : stocke l'adresse de la prochaine instruction à exécuter.
 - registre d'états (flag register) : permet de stocker des indicateurs sur l'état du système après l'exécution d'une instruction. par exemple,
 - C (pour carry) : vaudra 1 si une retenue est présente.
 - Z (pour Zero) : vaudra 1 si le résultat de la dernière opération réalisée est nul.
 - V (pour overflow) : vaudra 1 en cas de dépassement de capacité
 - N (pour Negative) : vaudra 1 si le résultat est négatif.
 - T (Trap flag) : mis à 1 le processeur fonctionne en mode pas à pas
 - IE (Interrupt Enable) : mis à 1 les interruptions sont prise en compte
 -
 - registre d'adresse RA : contient l'adresse de la donnée à lire ou à écrire en mémoire.
 - registres de données RD : contient temporairement la donnée lue ou à écrire en mémoire.
 - registre d'index XR (utilisé dans le mode d'adressage indexé) : l'adresse est obtenue en ajoutant son contenu à l'adresse contenue dans l'instruction ; peut être incrémenter/décrémenter automatiquement après son utilisation.
 - registre de base : contient l'adresse (le numéro de segment) à ajouter aux adresses (relatives) contenues dans les instructions.
- **d'une horloge** qui permet la synchronisation des éléments et des événements
- **un décodeur** qui détermine les opérations à exécuter en fonction du code de l'instruction.
- **un séquenceur** qui déclenche et coordonne les différentes opérations pour réaliser l'instruction.

Le séquenceur est une machine de Mealy recevant des informations du décodeur et des signaux d'états (entrées) et produisant des signaux de commandes contrôlant les différentes unités

Plusieurs réalisations sont possibles :

- séquenceur câblé :
 - circuit séquentiel (synchrone) réalisé avec des portes logiques
 - Un sous-circuit pour chaque instruction, sous-circuit activé selon le code envoyé par le décodeur.
- Séquenceur micro-programmé :
 - Une ROM contient des micro-programmes composés de micro-instructions
 - Le séquenceur sait exécuter les séquences de micro-instructions

L'exécution d'une instruction suit le cycle suivant :

1. recherche de l'instruction (FETCH)
2. décodage de l'instruction (opération et opérandes)
3. exécution de l'opération
4. écriture du résultat

recherche de l'instruction : L'adresse de l'instruction est stockée dans PC. L'instruction est stockée dans le registre d'instruction RI.

décodage : En général, une instruction est composé d'un premier segment appelé **opcode** codant l'opération à réaliser. La suite de l'instruction peut contenir des constantes (valeurs immédiates), ou coder l'emplacement des opérandes (selon le mode d'adressage de l'instruction) Dans les processeurs récents, un microprogramme est souvent utilisé pour traduire les instructions en différents ordres.

exécution : Les opérations et opérandes sont transmises à l'unité de calcul qui procède à l'exécution du calcul

écriture du résultat : Ecriture du résultat en mémoire.

Après l'exécution de l'instruction et l'écriture des résultats, tout le processus se répète, le prochain cycle d'instructions recherche la séquence d'instruction suivante puisque le compteur de programme avait été incrémenté. Si l'instruction précédente était un saut, c'est l'adresse de destination du saut qui est enregistrée dans le compteur de programme. Dans des processeurs plus complexes, plusieurs instructions peuvent être recherchées, décodées et exécutées simultanément, on parle alors d'architecture pipeline, aujourd'hui communément utilisée dans les équipements électroniques.

Exemple 6.1

1. Recherche de l'instruction :

PC	80000	80000	<i>add \$1, \$2, \$3</i>
		80004

2. Chargement de l'instruction à exécuter

- L'adresse de la prochaine instruction est placée dans le registre d'adresse : $RA \leftarrow PC$
- lecture à la mémoire : le contenu de la case pointée par RA est placée dans le registre d'instruction RI.

RI *add \$1, \$2, \$3*

3. Incrémentation du compteur ordinal PC

- Soit PC est muni d'un dispositif d'incrémentation
- Soit on utilise l'ALU

PC 80004

4. Décodage de l'instruction (**Décodeur**)

- identification d'une addition entre deux registres avec placement du résultat dans un registre
- Préparation des données (**Séquenceur**) : les contenus des registres \$2 et \$3 sont placés dans les deux registres d'entrée de l'ALU

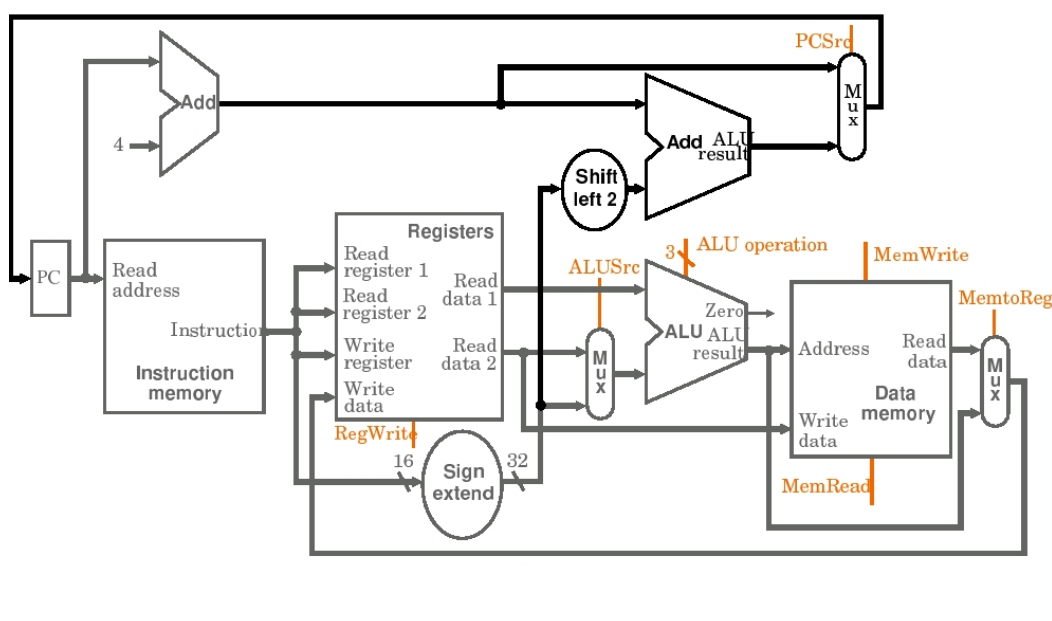
5. Execution

- Envoi du signal de l'opération d'addition à l'ALU
- L'ALU ajoute les deux opérandes et place le résultat dans son registre de sortie
- le contenu du registre de sortie de l'ALU est transféré dans le registre \$1

6.2 Processeur : fonctionnement

Nous illustrons le fonctionnement d'un processeur en prenant pour exemple d'étude un processeur MIPS. Pour simplifier l'exposé, nous choisissons de présenter ici un modèle ne permettant d'effectuer qu'une seule instruction par cycle d'horloge. De plus, ce modèle ne peut décoder que quelques instructions au format R ou I.

Voici un schéma simplifié de l'architecture MIPS.



Les différents composants sont légendés dans la Figure 6.1. Les entrées représentées en orange sont calculées par les unités de contrôles qui n'apparaissent pas sur ce schéma et qui seront introduits dans la Figure 6.2.

Le registre PC contient l'adresse de l'instruction à traiter. La partie haute du schéma est dédiée au calcul de la prochaine valeur du PC :

- PC est incrémenté de 4,
- puis si l'instruction décodée est un branchement, la nouvelle adresse est calculée en récupérant les 16 dernier bits de l'instruction, en les passant en 32 bits signés puis multipliant le résultat par 4 (Shift left 2) et enfin en additionnant ce nombre à PC.

C'est le multiplexeur en haut à droite qui va sélectionner un des 2 calculs en fonction de l'entrée **PCSrc**.

Partant maintenant linéairement du registre PC, le bloc **Instruction memory** prend en entrée l'adresse de l'instruction et donne en sortie l'instruction. Après décodage (transparent sur ce schéma), les opérandes de l'instruction sont récupérées dans les registres et envoyés sur les sorties **data1** et **data2**. Vient ensuite le calcul des opérandes qui seront envoyés à l'ALU, la première est **data1**, la seconde est, selon le type de l'instruction (déterminé par l'entrée **ALUSrc** du multiplexeur)

- soit **data2**
- ou bien (pour une instruction au format I) la valeur immédiate de l'instruction passée en 32 bits signés.

Selon le type de l'instruction réalisée, le résultat de l'ALU va être traité de 2 façons différentes (déterminé par l'entrée **MemoReg** du multiplexeur) :

- si le résultat du calcul était une adresse de donnée, la donnée est récupérée en mémoire et placée dans un registre
- sinon le résultat est directement placé dans un registre.

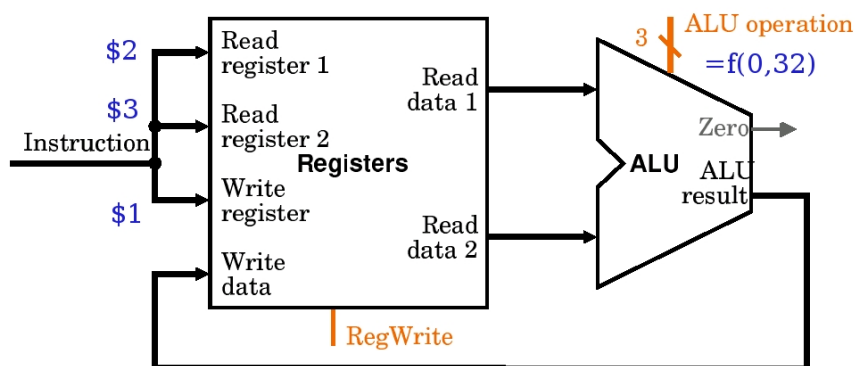
6.2.1 Exemples d'exécutions d'instructions

Nous examinons ici quelques exemples d'exécution d'une instruction par un processeur MIPS. Rappelons tout d'abord le format des instructions MIPS :

Format	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
Format R	Code op	rs	rt	rd	sa	funct
Format I	Code op	rs	rt	immédiat 16 bits		
Format J	Code op	immédiat 26 bits				

Exécution de : **add \$1,\$2, \$3**

Codeop	rs	rt	rd	sa	funct
0	2	3	1	0	32



- le signal **RegWrite** contrôle l'écriture dans le banc de registres
- **ALUoperation** décrit le type de calcul réalisé
- le signal **Zero** est émis si le calcul vaut 0.

Exécution de : **lw \$1, 100(\$2)**

Code op	rs	rt	immédiat 16 bits
35	2	1	100

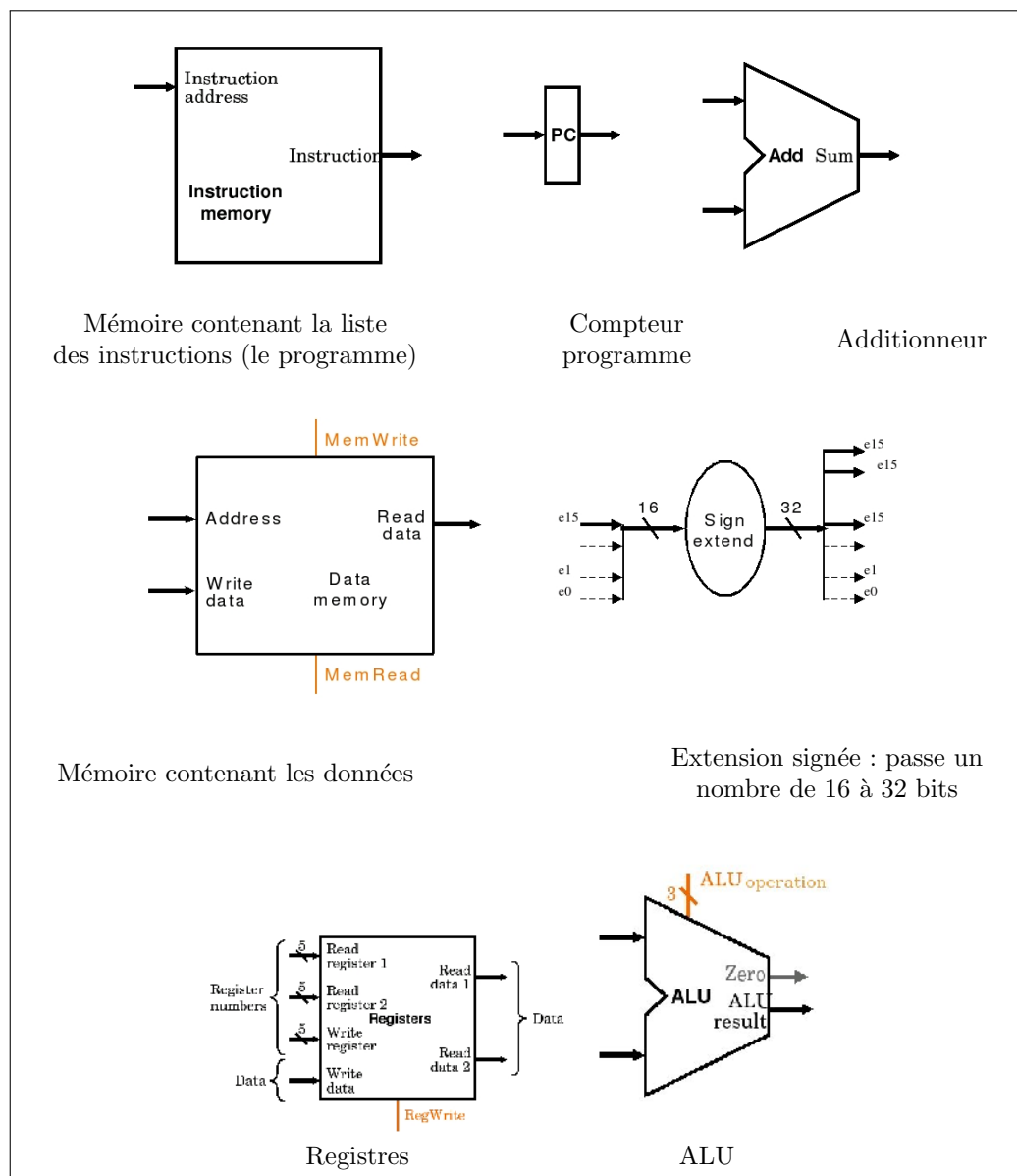
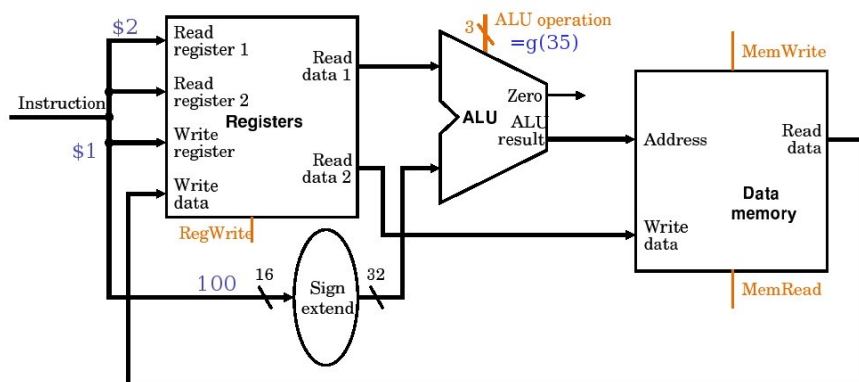


FIGURE 6.1 – Détail des composants

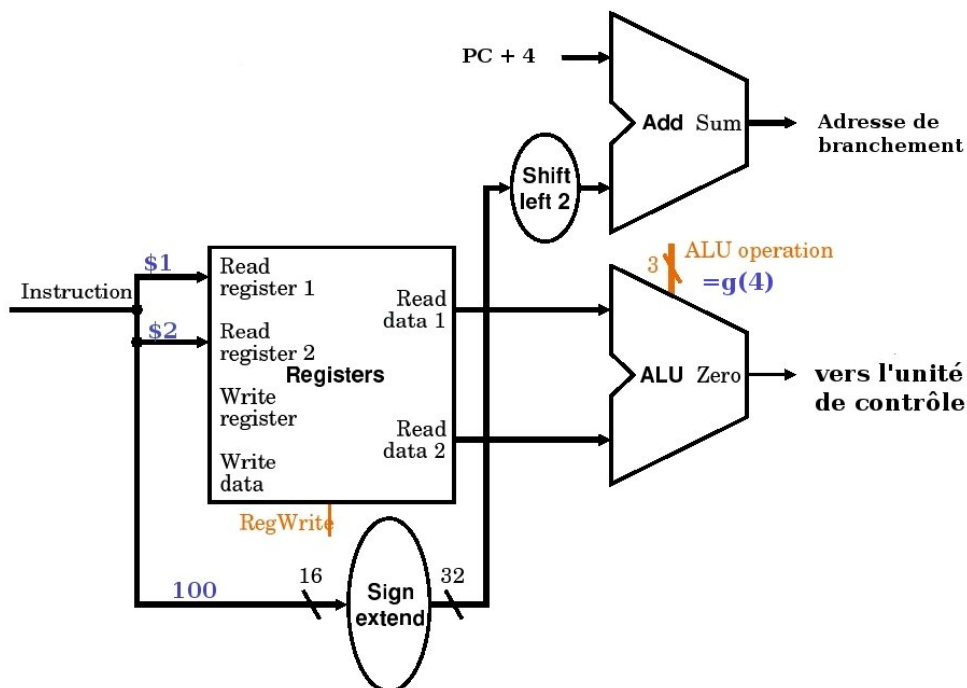


Le signal **MemRead** est activé.

- “immédiat 16 bits” est un déplacement relatif signé
- les signaux **MemWrite** et **MemRead** contrôlent respectivement l’écriture et la lecture dans la mémoire.

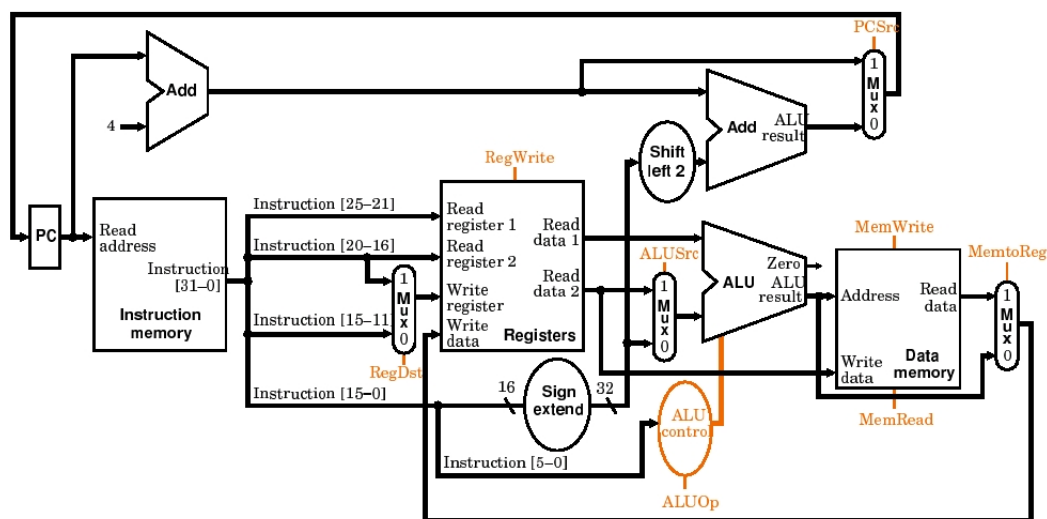
Exécution de : `beq $1,$2, 100`

Code op	rs	rt	adresse sur 16 bits
4	2	1	100



6.2.2 Contrôle de l’ALU

Nous détaillons maintenant un peu plus le schéma initial pour introduire le contrôle de l’ALU.



Sur ce schéma, on voit apparaître le traitement de l'instruction :

- bits 32 à 24 : il s'agit de l'opcode, il n'apparaissent pas encore sur le schéma, il seront considérés ultérieurement
- bits 25 à 21 : adresse de l'opérande 1
- bits 20 à 16 : adresse de l'opérande 2 ou du registre à écrire
- bits 15 à 0 : selon les cas :
 - Instruction R :
 - bits 15 à 11 : adresse du registre à écrire
 - bits 10 à 6 : décalage (non représenté sur le schéma)
 - bits 5 à 0 : code de l'opération
 - Instruction I : l'immédiat sur 16 bits est passé en 32 bits signé puis traité comme déjà détaillé plus haut.

Signaux de contrôle (ALUoperation)	Calcul réalisé
000	and
001	or
010	add
110	sub
111	slt

ALUoperation est calculé en fonction

- du champ **funct**, les 6 bits de poids faible de l'instruction exécutée
- du signal **ALUOp** sur 2 bits

Le signal **ALUOp** est calculé en fonction du **Codeop**, les 6 bits de poids fort de l'instruction exécutée

Codeop	ALUOp	funct	ALUoperation
lw	00		010
sw	00		010
beq	01		110
add	10	100000	010
sub	10	100010	110
and	10	100100	000
or	10	100101	001
slt	10	101010	111

6.2.3 L'unité de contrôle

Nous terminons de compléter le schéma en ajoutant l'unité de contrôle.

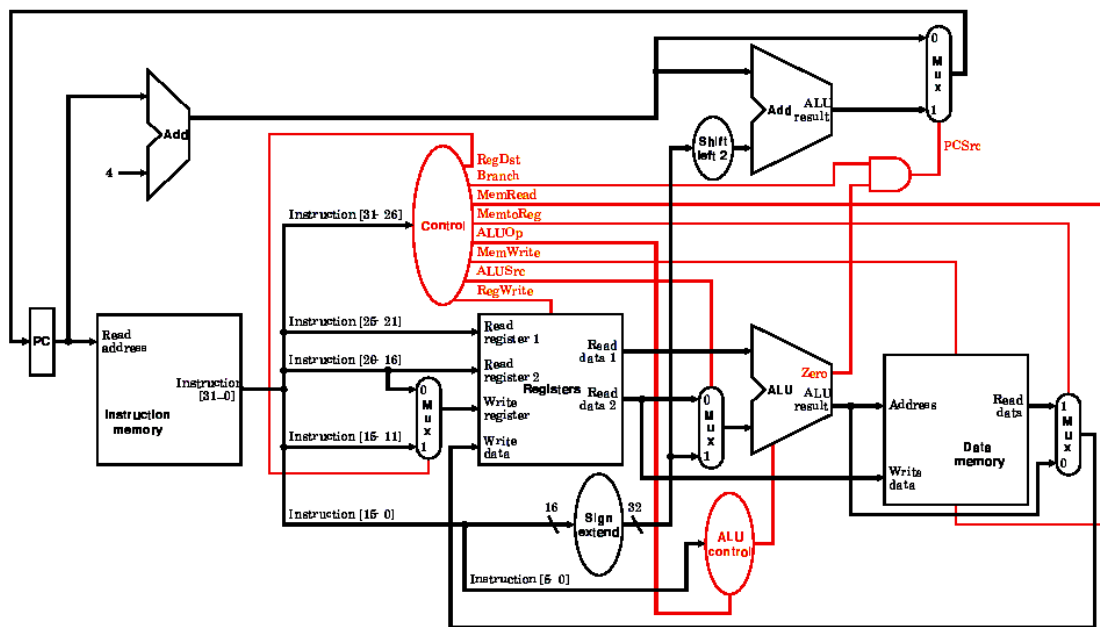


FIGURE 6.2 – Architecture MIPS avec unité de contrôle