Cours d'Architecture des ordinateurs

L2 Informatique 2014/2015

version du 23 septembre 2014

Séverine Fratani
Peter Niebert

Table des matières

Introduction

A refaire!!!

Chapitre 1

Codage

1.1 Systèmes de numération

Les nombres sont usuellement représentés en base 10. Chaque chiffre apparaissant dans un nombre est le coefficient d'une puissance de 10. Par exemple, le nombre 145 correspond au nombre obtenu par l'opération suivante : $1 \times 10^2 + 4 \times 10^1 + 5 \times 10^0$. Ce type de numération peut-être appliqué à n'importe quelle autre base.

1.1.1 Numeration en base b

Étant donné un entier positif b, chaque nombre entier x peut être représenté de manière unique par un nombre $a_n a_{n-1} \cdots a_0$, tel que $a_n \neq 0$ et pour tout $i \in [0, n]$, $a_i \in [0, b-1]$ et

$$x = a_n \times b^n + \cdots a_0 \times b^0.$$

Toutefois, pour les bases supérieures ou égales à 11, les symboles (ou chiffres) usuels (0, 1 ,..., 9) ne permettent pas une écriture non ambigüe. Par exemple, en base 11, on ne sais pas si le nombre 10 désigne 10×11^0 ou $1 \times 11^1 + 0 \times 11^0$. Pour ces bases, il faut donc enrichir l'ensemble des symboles (ou chiffres) utilisés pour le codage. Par exemple, en base 16, très utilisée en informatique, les chiffres sont $0, 1, \ldots, 9, a, b, c, d, e, f$, où $a = 10$, $b = 11$, $c = 12$, $d = 13$, $e = 14$ et $f = 15$.

Dans toute la suite, afin d'éviter toute confusion, nous utiliserons la notation x_b pour indiquer que le nombre x est représenté en base b.

1.1.2 Taille des codages

En informatique, les nombres ne peuvent pas avoir une taille arbitrairement grande. Ils ont donc toujours une taille fixée.

Déterminons la plage de nombres que l'on peut écrire en base b avec des nombres de taille n : il y a n places possibles pouvant contenir chacune un chiffre entre 0 et $b-1$, soit b^n nombres différents. Sur n chiffres, on écrit donc les nombres compris entre 0 et $b^n - 1$.

1.1.3 Comment obtenir une écriture en base b

1.1.3.1 À partir d'un nombre en base 10

Voyons par exemple comment écrire le nombre 145 en base 8. On remarque facilement la suite d'égalités suivante :

$$\begin{aligned} 145 &= 1 + 8 \times 18 \\ &= 1 + 8 \times (2 + 8 \times 2) \\ &= 1 + 8 \times (2 + 8 \times (2 + 8 \times 0)) \\ &= 1 + 2 \times 8 + 2 \times 8 \times 8 \\ &= 1 \times 8^0 + 2 \times 8^1 + 2 \times 8^2 \end{aligned}$$

On obtient donc $145_{10} = 221_8$.

On en déduit facilement un algorithme général utilisant les opérations suivantes :
la division entière : *div*, le modulo : *mod*.

Data: une base b, un entier x à écrire en base b
Result: un nombre $a_{n-1} \cdots a_0$ avec $x \in [b^{n-1}, b^n - 1]$ et chaque $a_i \in [0, b-1]$.
$i = 0$;
while $x \neq 0$ **do**
  $a_i = x \, mod \, b$;
  $x = x \, div \, b$;
  $i = i + 1$;
end

Algorithm 1: Ecriture en base b

1.1.3.2 À partir d'un nombre en base c

Dans le cas général, si on dispose de l'écriture d'un nombre en base c, le plus simple pour obtenir l'écriture en base b est de calculer la valeur du nombre puis d'appliquer l'algorithme décrit précedement. Il existe cependant des cas où la transformation est plus simple. Imaginons que b est une puissance de c, i.e., $b = c^k$ (par exemple $c = 2$ et $b = 16 = 2^4$), alors on obtient l'écriture en base b à partir de l'écriture en base c en groupant les chiffres par k élements à partir du chiffre de poids faible (i.e., le chiffre le plus à droite). Chaque groupe représente alors en base c un nombre entre 0 et $b - 1$. Par exemple si on veut passer de la base 2 à la base 16 :

$$(101110000011)_2 = ((1011)\ (1000)\ (0011))_2 = (b83)_{16}$$

puisque $(1011)_2 = 11_{10} = b_{16}$, $1000_2 = 8_{10} = 8_{16}$, $0011_2 = 3_{10} = 3_{16}$.

1.2 Codage de l'information

Dans un ordinateur, l'information est codé en "binaire", i.e., en base 2. Les chiffres binaires sont appelés des bits. Un bit est donc soit un 0, soit un 1, et une information est représentée par une séquence de bits. Une séquence de 8 bits est appelée un "octet".

1.2.1 L'arithmétique binaire

L'arithmétique binaire ressemble à l'arithmétique décimale. Voici la table d'addition des nombres binaires :

	0	0	1	1
+	0	1	0	1
	—	—	—	—
Somme	0	1	1	0
Retenue	0	0	0	1

Voici un exemple d'addition et un exemple de soustraction :

	1	0	1	1	0
+	1	1	0	1	1
1	1	0	0	0	1

	1	1	0	1	0
-	1	0	0	1	1
	0	0	1	1	1

Les multiplications et divisions se font sur le même mode, en adaptant les règles de l'arithmétique décimale.

1.2.2 Représentation des nombres entiers en binaires

Dans les ordinateurs, tous le nombres sont représentés par des nombres binaires d'une taille fixée. Les entiers positifs sont representés par le codage "binaire pur non-signé" découlant directement de la numération en binaire vue précédement.

1.2.2.1 Représentation avec un bit de signe

Une idée simple pour représenter les entiers positifs et négatifs est de réserver un bit (par exemple celui de gauche) pour coder le signe. Supposons qu'on code sur 8 bits, 3 sera codé 00000011 et -3 sera codé 10000011. Ce codage n'est en fait pas utilisé car il comporte de nombreux inconvenients. D'abord, la présence de deux valeurs pour 0 (00000000) et -0 (10000000), ensuite, l'addition est compliquée : il faut examiner les signes, et faire une addition ou une soustraction selon les cas.

1.2.2.2 Représentation complément à un

On note $(x)_{1cn}$ la représentation en complément à un sur n bits de l'entier x :

si x est un nombre positif, alors on donne sa représentation binaire avec la restriction que le bit le plus à gauche doit valoir 0. On ne peut donc coder sur n bits que les entiers positifs de 0 à $2^{n-1}-1$. Pour un nombre négatif $x=-y$, on inverse tous les bits de $(y)_{1cn}$ (on remplace les 1 par des 0 et les 0 par des 1). Le bit le plus à gauche est donc 1.

L'addition est simple à réaliser : on ajoute les deux nombres puis on ajoute la retenue éventuelle.

Voici par exemple les additions $6+(-4)=2$ et $-5+3=-2$:

	0	1	1	0
+	1	0	1	1
1 |	0	0	0	1
+				1
	0	0	1	0

	1	0	1	0
+	0	0	1	1
0—	1	1	0	1
+				0
	1	1	0	1

L'inconvénient de ce codage est qu'il y a deux représentations de 0 : par exemple sur 4 bits, 0000 et 1111.

1.2.2.3 Représentation complément à deux

La méthode réellement utilisée est la représentation par "complément à deux". On note $(x)_{2cn}$ la représentation en complément à deux sur n bits de l'entier x :

$$\text{si } (x)_{2cn}=x_{n-1}\cdots x_0 \text{ alors } x=-x_{n-1}2^{n-1}+\sum_{i=0}^{n-2}x_i2^i$$

Les nombres positifs sont donc représentés en binaire classique mais sont seulement codables les entiers allant de 0 à $2^{n-1}-1$. Etant donné un entier positif x, on obtient $-x$ de la facon suivante : on remplace les 1 par des 0 et les 0 par des 1, puis on ajoute 1 au résultat. Si une retenue est crée sur le dernier bit, elle est effacée. On passe de la même façon d'un nombre négatif à positif : on inverse tous les bits et on ajoute 1 au résultat.

Sur 8 bits, on peut coder tous les nombres de 127 à - 128 : 127=01111111 et -128=10000000. Tous les nombres positifs ou nuls ont leur bit le plus à gauche à 0, et tous les nombres négatifs ont leur bit le plus à gauche à 1.

Sur n bits, on peut coder tous les nombres de $2^{n-1}-1$ à -2^{n-1}. Si a_{2cn} est la représentation d'un nombre positif, alors $-a_{2cn}$ correspond à l'écriture binaire classique de $2^8 - a$

Comme pour le complément à 1, l'addition et la soustraction se font sans se préoccuper des signes : les circuits de calcul en sont grandement simplifiés. De plus, il n'y a qu'une seule représentation de 0.

Pour l'addition, on additionne les deux nombres et on omet l'éventuelle retenue. Voic par exemple les additions $6-5=1$ et $-6+3=-3$:

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

Pour faire une soustraction, on utilise l'identité

$$(a-b)_{2cn} = a + (-b)_{2cn}$$

Par exemple, pour $n=8$, supposons qu'on veuille faire l'opération $(01010110 - 00110111)_{2c8}$, on effectue l'opération $01010110 + 11001001 = 100011111$, le nombre obtenu dépasse les 8 bits, on supprime le bit de gauche supplémentaire. Finalement $(01010110 - 00110111)_{2c8} = (00011111)_{2c8}$.

1.2.2.3.1 Débordement de capacité Le problème de coder sur un nombre fixé de bits est que l'on peut déborder lors de calculs. Par exemple, si on effectue l'opération $(01000000 + 01000000)_{2c8}$ on obtient $(10000000)_{2c8}$ c'est à dire un nombre négatif alors qu'on a additionné deux nombres positifs ! Le résultat est donc faux, on dit qu'il y a débordement (overflow).

Pour le codage en complément à deux, on peut facilement détecter un débordement : il engendre forcément un erreur de signe. Il suffit donc d'observer les règles suivantes

- si on additionne deux nombres de signes contraires, il ne peut pas y avoir de débordement.
- si on additionne deux nombres positifs, il y a débordement si et seulement si le résultat est négatif, i.e., si le bit de gauche est à 1.
- si on additionne deux nombres negatifs, il y a débordement si et seulement si le résultat est positif, i.e., si le bit de gauche est à 0.

1.2.3 Représentation des nombres à virgule

1.2.3.1 Représentation en virgule fixe

Reprenons l'exemple de la base 10, et considérons le nombre $0,135$. Cette écriture désigne le nombre obtenu par le calcul suivant :

$$1 \times 10^{-1} + 3 \times 10^{-2} + 5 \times 10^{-3}$$

De la même façon, on peut représenter des nombres à virgule en binaire. Par exemple, le nombre $(1,11)_2$ désigne le nombre $2^0 + 2^{-1} + 2^{-2} = 1 + 0,5 + 0,25 = 1,75$.

Voici un algorithme permettant d'obtenir le codage d'un nombre $0 < x \leq 1$:

Data: une base b, un reel x dans $[0,1[$ à écrire en base b, une précision $n \geq 1$
Result: un nombre $0, a_1 \cdots a_n$ où chaque $a_i \in [0, b-1]$
Variables : y reel ;
$i = 0$;
$y = x$;
while $i < n$ **do**
  $y = y \times b$;
  $a_i =$ partie_entière(y);
  $y = y - a_i$;
  $i = i + 1$;
end

Algorithm 2: Codage en base b d'un nombre réel avec précision n

Lorsqu'on veut coder des nombres très grands, ou très petits, le codage binaire classique n'est plus utilisable puisqu'il faut n bits pour coder 2^n nombres.

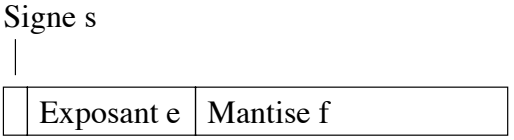
1.2.3.2 Représentation en virgule flottante

Elle correspond en fait à la notation dite "scientifique" des grands nombres comme 3×10^{27} ou encore 8×10^{-18}.

Pour des raisons évidentes d'espace mémoire, il n'est possible de représenter qu'un nombre borné de rééls, on parle alors plutôt de *flottants*

Depuis les années 70, il existe un standard pour la représentation des flottants. Aujourd'hui la plupart des ordinateurs utilisent ce standard. C'est la **représentation IEEE 754**.

Un nombre flottant est codé par 3 nombres représentés de la façon suivante :



Le coefficient f est appelé la *mantise*, e est appelé l'*exposant* et s représente le signe : positif si $s = 0$ et négatif si $s = 1$.

Le standard inclu deux représentations : simple précision et double précision.

Nombre de bits	taille de s	taille de f	taille de e	E_{min}	E_{max}
32 (simple précision)	1	23	8	-126	+127
64 (double précision)	1	52	11	-1022	+1023

où E_{min} et E_{max} représentent respectivement le plus petit et le plus grand exposant codable dans la représentation.

1.2.3.3 Codage

Considérons le codage sur 32 bits. On commence par écrire la valeur absolue du réel r à coder en binaire à virgule fixe. On décale ensuite la virgule en multipliant par des puissances de 2, jusqu'à avoir un et un seul chiffre avant la virgule.

Prenons par exemple $r = -123,5$. On le code par $-1111011,1$ puis on décale la virgule et on obtient $-(1,1110111) \times 2^6$. On en déduit

— le bit de signe $s = 1$

— la mantisse $M = 1110111$ qu'on complète pour obtenir un mot f sur 23 bits :
$f = 111\,0111\,0000\,0000\,0000\,0000$

— l'exposant $E = 6$ que l'on code en excès à 127 : le nombre e codé sur 8 bits est donc $e = E + 127 = 133 = (1000\,0101)_2$

Le codage de r est donc

1	1000 0101	111 0111 0000 0000 0000 0000

1.2.3.4 Décodage

La valeur d'un nombre est donnée par :

$$(-1)^s \times (1 + \sum_{i=1}^{\text{taille de } f} f_i 2^{-i}) \times 2^{e - E_{max}} \text{ où } f_i \text{ est le bit } i \text{ de la représentation de } f$$

Par exemple en simple précision, considérons la représentation

1	1000 0010	001 1000 0000 0000 0000 0000

On a $s = 1$, $e = 130 - 127 = 3$ et $f = 2^{-3} + 2^{-4} = 0,125 + 0,0625 = 0,1875$. Le nombre codé est donc $-1,1875 \times 2^3 = -9,5$.

1.2.3.5 Cas particuliers

Afin de traiter certains cas spéciaux, la norme prévoit un schéma de codage comme suit (pour le codage sur 32 bits) :

exposant e	partie fractionnaire f	valeur de r	
$0 < e < 255$	f	$(-1)^s \times (1, f)_2 \times 2^{e-127}$	normalisé
$e = 0$	$f \neq 0$	$(-1)^s \times (0, f)_2 \times 2^{e-126}$	dénormalisé
$e = 0$	$f = 0$	0	dénormalisé
$e = 255$	$f = 0$	$+/-/infty$	réservé
$e = 255$	$f \neq 0$	NaN	réservé

Lorsqu'on travaille sur de très petits nombres (exposant minimum), on n'effectue pas la normalisation (le chiffre avant la virgule est alors 0). Le nombre est alors qualifie de dénormalisé. NaN (Not a Number) est une valeur spéciale destinée à coder le résultat d'opérations incorrectes (comme la division par zéro).

1.2.4 Codage des caractères

Différents standards ont étés définis pour le codage des caractères.

Le code ASCII (American Standard Code for Information Interchange) créé en 1961 associe un nombre à un caractère. Chaque caractère est codé sur 7 bits. Il ne contient donc que 128 caractères différents (de 00 à 7F en hexadécimal).

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

Bien qu'étendu par la suite à un code sur 8 bits, beaucoup de caractères utilisés dans certaines langues ne sont pas représentés. L'organisme de normalisation OSI à donc défini différents standards plus adaptés aux besoins des différentes langues occidentales. Le français utilise le plus souvent ISO 8859-1, aussi nommé latin1, ou ISO 8859-15 (latin9).

Enfin, Unicode créé en 1991 a pour vocation de rassembler tous ces codes afin d'avoir un code commun pour toutes les langues. Il contient plus d'un million de caractères.

Chapitre 2

Algèbre de Boole

Le fonctionnement des circuits est décrit en utilisant l'algèbre binaire (algèbre de Boole à deux valeur). Nous en donnons les bases dans cette section.

L'algèbre de Boole est une structure algébrique

— donnée par un ensemble contenant au moins deux valeurs $\{0, 1\}$, et les trois 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.
— et satisfaisant les axiomes suivants (par convention, la conjonction est prioritaire sur la disjonction)
  — commutativité : pour tous $a, b \in \{0, 1\}$,

$$a \cdot b = b \cdot a \qquad a + b = b + a$$

  — associativité : pour tous $a, b, c \in \{0, 1\}$,

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c \qquad a + (b + c) = (a + b) + c$$

  — distributivité : pour tous $a, b, c \in \{0, 1\}$,

$$a \cdot (b + c) = a \cdot b + a \cdot c \qquad a + (b \cdot c) = (a + b) \cdot (a + c)$$

  — élements neutres : pour tout $a \in \{0, 1\}$,

$$1 \cdot a = a \cdot 1 = a \qquad 0 + a = a + 0 = a$$

  — complément : pour tout $a \in \{0, 1\}$,

$$a \cdot \bar{a} = \bar{a} \cdot a = 0 \qquad a + \bar{a} = \bar{a} + a = 1$$

2.1 Algèbre binaire

Nous allons nous intéresser à l'algèbre de Boole binaire, c'est à dire que l'ensemble des éléments de l'algèbre est $\{0, 1\}$ (ou bien $\{Vrai, Faux\}$).

Le définition suivante des opérateurs satisfait l'ensemble des axiomes. C'est celle que nous utiliserons.

— le complément

a	$\bar{a}$
0	1
1	0

— la conjonction

a	b	$a.b$
0	0	0
0	1	0
1	0	0
1	1	1

— la disjonction

a	b	$a+b$
0	0	0
0	1	1
1	0	1
1	1	1

On dit que les opérateurs ainsi définis sont **un modèle des axiomes** car ils vérifient chacun des axiomes.

Par exemple pour $1.a = a$, on a effectivement $1.1 = 1$ et $1.0 = 0$.

2.1.1 Propriétés

Les propriétés suivantes peuvent être déduites des axiomes

— Elément absorbant :

$$a.0 = 0.a = 0 \qquad a + 1 = 1 + a = 1$$

— Absorption

$$a.(a + b) = a \qquad a + (a.b) = a$$

— Idempotence

$$a.a = a \qquad a + a = a$$

— Involution

$$\bar{\bar{a}} = a$$

— Lois de De Morgan

$$\overline{a.b} = \bar{a} + \bar{b} \qquad \overline{a + b} = \bar{a}.\bar{b}$$

Ces propriétés sont déduites directement des axiomes (sans utiliser le modèle particulier, i.e., la définition des opérateurs, que nous venons de voir).

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) $f : \{0,1\}^n \to \{0,1\}$ peut être donnée

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

x	y	z	$m = f(x,y,z)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	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 $x_1, \ldots, x_n$. Par exemple :

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

2.2.1 Forme normale disjonctive

Une expression booléenne est en forme normale disjonctive si elle est écrit comme
— une disjonction de monômes
— chaque monône étant une conjonction de littéraux
— un littéral étant soit les constantes $0, 1$, soit une variable x, soit le complément de x, $\bar{x}$.
Exemple :

$$(x.y.\bar{z}) + (\bar{x}.z) + y$$

Théorème 2.1 *Toute expression booléenne est équivalente à une expression en forme normale disjonctive.*

PREUVE. on suppose les variables comme étant $x_1, \ldots, x_n$,

1. On considère la table de vérité associé à l'expression et appelons f la fonction booléenne ainsi représentée
2. supposons que f vaille 1 pour k entrées dans cette table (et donc 0 pour les $2^n - k$ autres entrées). Comme $0 + 1 = 1$ on peut écrire f comme $f_1 + \ldots + f_k$ où $f_i : \{0,1\}^n \to \{0,1\}$ n'est à 1 que pour une seule entrée.
3. Maintenant pour chaque f_i, il est simple de voir que le monôme $y_1 \ldots . y_n$ défini par $y_j = x_j$ si la $i$eme valeur de l'entrée est 1 et $y_j = \bar{x_j}$ représente précisément la fonction f_i.

□

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

Corollaire 2.2 *Toute fonction booléenne peut être représentée comme une expression booléenne.*

PREUVE. Il suffit d'extraire l'expression en forme normale disjonctive de la table de vérité de la fonction booléenne. □

2.2.2 Forme normale conjonctive

Une expression booléenne est en forme normale conjonctive si elle écrit comme
— une conjonction de sommes
— chaque somme étant une disjonction de littéraux
— un littéral étant soit les constantes $0, 1$, soit une variable x, soit le complément de x, $\bar{x}$.

Exemple :

$$(x + y + \bar{z}).(\bar{x} + z).1$$

Théorème 2.3 *Toute expression booléenne est équivalente à une expression en forme normale conjonctive.*

Pour obtenir la forme normale conjonctive d'une expression booléenne à n variables $x_1, \ldots, x_n$, on construit sa table de vérité, et on écrit l'expression composée de la conjonction des sommes s_j pour chaque ligne j valant 0. La somme s_j est obtenue de la façon suivante : $s_j = y_1 + \ldots + y_n$ où

— $y_i = x_j$ si la $i$eme valeur de l'entrée vaut 0

— $y_j = \bar{x_j}$ sinon.

Pour l'exemple précédent, la forme normale conjonctive est donc
$(x + y + z) \cdot (x + \bar{y} + z) \cdot (\bar{x} + y + z) \cdot (\bar{x} + \bar{y} + \bar{z})$

2.2.3 Simplifications de fonctions booléennes : tables de Karnaugh

Nous allons voir que les fonctions booléennes sont en fait implantées à l'aide de portes logiques constituées de transistors. Afin d'économiser de l'espace, de l'énergie et de l'argent, on souhaite utiliser le moins de transistors possibles. On va donc essayer de trouver pour les fonctions booléennes une représentation la plus petite possible (en terme d'expression).

Pour simplifier l'expression d'une fonction booléenne, on peut tout simplement utiliser les axiomes et les propriétés de l'algèbre de Boole, afin de passer d'une expression à une autre plus simple.

Une méthode très efficace est l'utilisation des tableaux de Karnaugh, qui permet une simplification visuelle. Cette méthode de simplification se base sur la propriété suivante :

$$(a.b) + (a.\bar{b}) = a.(b + \bar{b}) = a \tag{2.1}$$

stipulant que si une disjonction contient deux conjonctions qui ne varient que sur une variable, positive (sans symbole non devant, b) dans l'une et négative (avec un symbole non devant, $\bar{b}$) dans l'autre alors on peut éliminer ces deux conjonctions et la remplacer par une conjonction identique mais ne contenant plus b.

La première étape de la méthode consiste à transformer la table de vérité de la fonction à simplifier en un tableau bi-dimensionnel (séparation en deux parties de l'ensemble des variables (une pour les lignes, une pour les colonnes)).

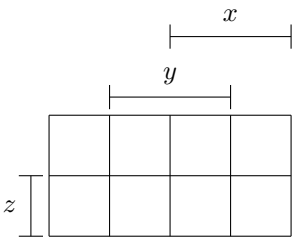
Lignes et colonnes sont indexées par toutes les valuations des variables correspondantes **tel que** entre deux lignes (resp. colonnes) une seule valeur booléenne change.

Par exemple, pour une fonction de trois variables x, y, z, la table à l'allure suivante :

$\frac{xy}{z}$	00	01	11	10
0				
1				

La première ligne (respectivement colonne), est indexée par une valuation booléenne possibles des variables choisies pour les lignes (resp. des colonnes). Dans notre exemple pour les colonnes (ie les variables xy), il y a 4 valuations possibles 00, 01, 10, 11. Les dispositions des valuations ne doivent pas être quelconques : il faut en passant de l'une à sa suivante, qu'un seul bit change de valeur (ainsi, on a côte-à-côte le monomes avec x et celui avec $\bar{x}$). Il convient de noter que pour passer de la dernière à la première colonne (resp. ligne) un seul bit change dans la valuation : le tableau sera considéré de manière sphérique.

Les tables de Karnaugh sont souvent présentées sous la forme équivalente suivante :

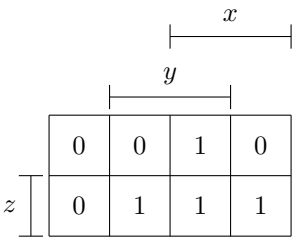


C'est celle-ci que nous allons utiliser par la suite.

Une fois le tableau construit, on le remplit avec les valeurs de la fonction booléenne. Voici le tableau rempli de la fonction g dont la forme normale disjonctive est

$$x.y.z + \bar{x}.y.z + x.\bar{y}.z + x.y.\bar{z}$$

$\frac{xy}{z}$	00	01	11	10
0	0	0	1	0
1	0	1	1	1



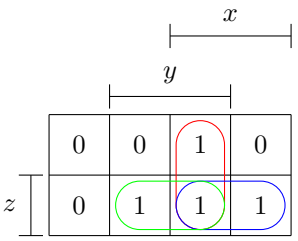
Ensuite, on cherche à recouvrir tous les 1 du tableau en formant des regroupements. Chaque regroupement :

— ne contient que des 1 adjacents,
— doit former un rectangle
— et le nombre d'éléments d'un regroupement doit être une puissance de deux.

On choisit toujours

— les plus grands regroupements possibles,
— et le plus petit nombre possible de regroupements recouvrant les 1 du tableaux.

Dans notre exemple on peut faire trois regroupements de deux éléments, et ils sont tous nécessaires au recouvrement. Chaque regroupement correspond à un monôme obtenu en utilisant la simplification 2.1



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

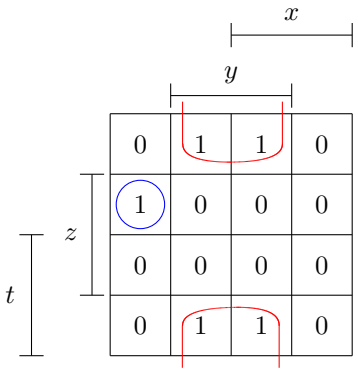
Finalement, on obtient l'expression simplifiée suivante :

$$x.y + x.z + y.z$$

Les règles suivantes doivent toujours être respectées

— Tous les 1 de la table de vérité doivent être considérés : un 1 peut être utilisé dans plusieurs regroupement ; au pire un 1 isolé est un regroupement de taille un.
— Le tableau doit être considéré de façon circulaire (on peut replier le tableau comme une sphère).
— Les regroupements peuvent également être de taille 4,8, ... (toutes puissances de 2)

Voici un dernier exemple dans lequel on fait un regroupement de taille 1 et un regroupement de taille 4, en utilisant la propriété de circularité du tableau.



$y.\bar{z} + \bar{x}.\bar{y}.z.\bar{t}$

Chapitre 3

Circuits combinatoires

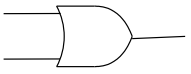
Un circuit combinatoire est un circuit physique élaboré à partir de composants électroniques. Il comporte des entrées et des sorties. Les entrées et sorties sont des valeurs booléennes et chaque sortie est valeur d'une fonction booléenne fonction des entrées. Les circuits combinatoires sont construits à partir de "portes logiques"

3.1 Portes logiques

Les portes logiques sont des 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.

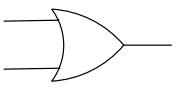
Les tables de vérité donnent la valeur de la sortie pour chacune des portes en fonction de la valeur de entrées.

Porte ET



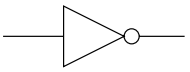
0	0	0
0	1	0
1	0	0
1	1	1

Porte OU



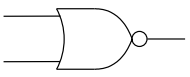
0	0	0
0	1	1
1	0	1
1	1	1

Porte NON



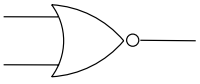
0	1
1	0

Porte NON-ET



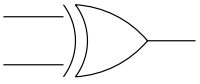
0	0	1
0	1	1
1	0	1
1	1	0

Porte NON-OU



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

Porte OU-X



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 : typiquement, un signal compris entre 0 et 1 volt correspond au 0 et un signal entre 2 et 5 volt correspond à l'état binaire 1. Une porte logique est composée de **transistors**. Un transistor est un composant électronique se comportant comme un interrupteur très rapide. Un transistor correspond en fait à un inverseur, en montant deux transistors en série (ou cascade), on obtient une porte NON-ET, et en montant deux transistors en parallèle, on construit une porte NON-OU.

Toutes les autres portes peuvent ensuite être construites en utilisant ces trois portes.

3.2 Circuits combinatoire

Un circuit combinatoire :

— est défini par un ensemble de portes reliées les unes aux autres.
— les sorties des portes sont reliées aux entrées d'autres portes (définissant une orientation des connexions)
— en suivant l'orientation des connections, il est **impossible** que partant de la sortie d'une porte, on revienne à l'une des ses entrées (graphe acyclique)

Un circuit combinatoire peut être vu comme une porte logique (à plusieurs sorties).

De l'idéal à la réalité... Les circuits combinatoires sont **une idéalisation** dans lesquels

— le temps de propagation n'est pas pris en compte
— la sortie est disponible dès que les entrées sont présentes

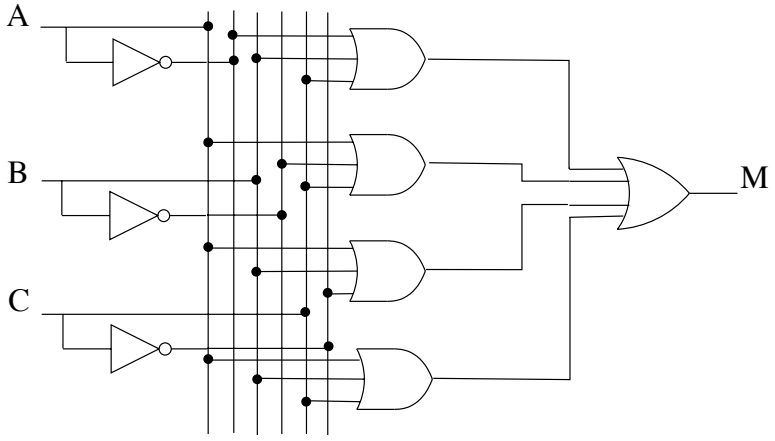
En réalité le temps de passage de 0 à 1 **n'est pas**

— immédiat (temps de parcours du courant électrique)
— instantané (temps de réponse d'une porte)

3.2.1 Le circuit "Majorité"

Le circuit "majorité" comporte 3 entrées A, B, C et une sortie M.

La sortie M vaut 1 si une majorité des entrées valent 1. Voici la table de vérité de la fonction majorité et le circuit correspondant



A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

La forme normale disjonctive de cette fonction est :

$$\bar{A}.B.C + A.\bar{B}.C + A.B.\bar{C} + A.B.C$$

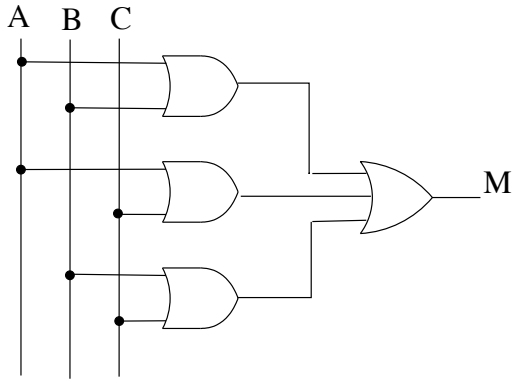
Les sorties des portes sont connectées aux entrées d'autres portes par des fils. Les fils se croisent sans contact sauf s'ils sont explicitement connectés par un $\bullet$.

Ce circuit implémente précisément la fonction booléenne donnée au dessous. On note cependant qu'on utilise des portes OU et ET à plus de deux entrées. Cela est possible car ces opérations ET et OU sont associatives.

En utilisant un tableau de Karnaugh, on obtient la forme simplifiée suivante :

$$A \cdot B + B \cdot C + C \cdot A$$

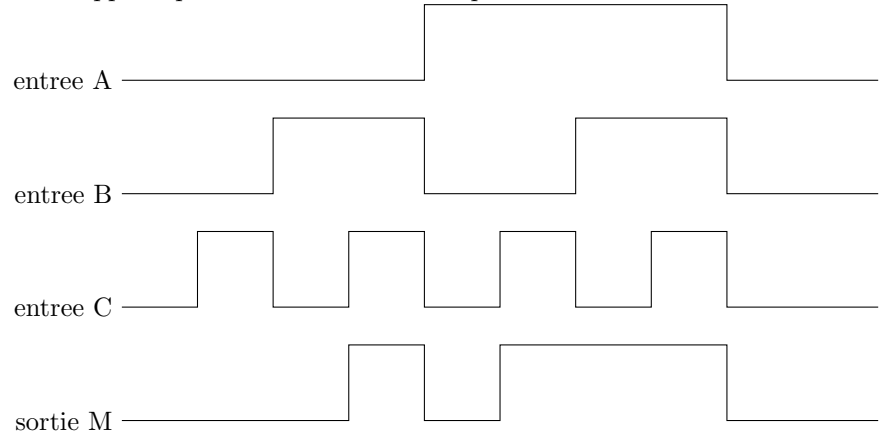
Le circuit correspondant est le suivant :



Chronogramme du circuit "Majorité" : Un chronogramme est une représentation visuelle des valeurs des entrées et sorties au cours du temps. On modifie les entrées, ce qui modifie également les sorties. On a le plus à gauche, les valeurs des entrées à $t = 0$ et les t croissent le long de l'axe des abscisses. Une valeur haute sur l'axe des ordonnées correspond à un signal à 1, et une valeur base correspond à un signal à 0.

On remarque que les entrées ne dépendent que des sorties : pour deux valeurs identiques des entrées (à des endroits différents du chronogramme), les valeurs de sortie sont identiques. C'est une propriété des *circuits combinatoires*.

Le chronogramme représenté ici est une version "idéalisée" du chronogramme réel du circuit car on a supposé que le franchissement des portes était instantané.

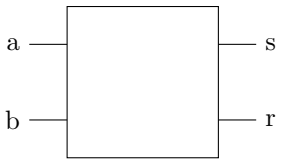


3.2.2 Les additionneurs

3.2.2.1 Le demi-additionneur

Un demi-additioneur est un circuit qui prend en entrée deux bits (a et b) et qui produit la somme (addition) de ces deux nombres s et la retenue éventuelle r.

— entrées : a et b
— sorties : s la somme et r la retenue

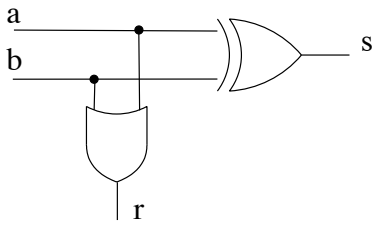


a	b	s	r
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$s = a \oplus b$
$r = a.b$

$\oplus$ est l'opérateur de ou-exclusif (XOR)

On parle de **demi-additioneur** (additioneur 1 bit) : ce circuit ne peut pas être étendu en un additioneur n bits. Voici le circuit correspondant au demi-additionneur :

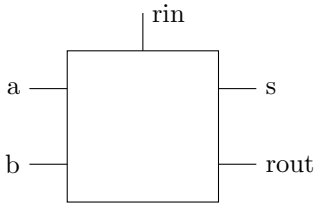


Il est réalisé avec une porte XOR. Vous pouvez, comme exercice, essayer de réaliser un circuit pour le demi-additioneur en n'utilisant que des portes OU et ET.

3.2.2.2 L'additionneur

Un additioneur est un circuit qui prend en entrée trois bits a,b (les chiffres à additionner) et rin une retenue d'entrée (qui peut provenir par exemple de l'addition d'autres chiffres) et qui produit la somme (addition) de ces trois nombres s et la retenue éventuelle $rout$.

— entrées : a, b et rin la retenue d'entrée
— sorties : s la somme et $rout$ la retenue de sortie



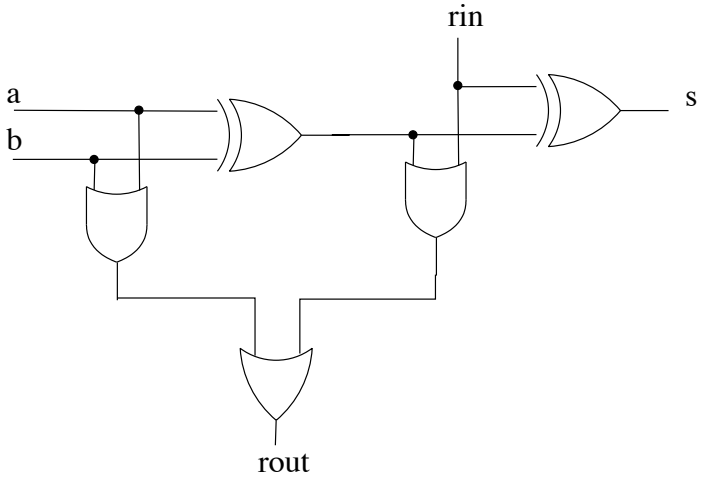
a	b	rin	s	$rout$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s = a \oplus b \oplus rin$$

$$rout = majorite(a, b, rin)$$

Voici un circuit de l'additionneur : on utilise une porte XOR pour calculer la somme de a et b dont le résultat est sommé avec rin toujours grâce à une porte XOR produisant la sortie s.

La sortie $rout$ est produite comme la "majorité" des entrées : en effet dès que au moins deux entrées valent 1, alors une retenue doit être produite.



3.2.2.3 Addtionneur 4 bits

On ajoute les nombres $(b_3b_2b_1b_0)_2$ et $(a_3a_2a_1a_0)_2$, on obtient un résultat $(s_3s_2s_1s_0)_2$ et un bit de débordement c (correspondant à une retenue éventuelle).

On pourrait bien-sûr construire un additionneur 4 bits à partir de sa table de vérité. Cependant, celle-ci aurait 256 lignes. Il parait difficile de construire à la main une telle table. Nous allons plutôt appliquer une approche modulaire : on utilise des (demi-)additionneurs 1 bits pour construire un additionneur 4 bits.

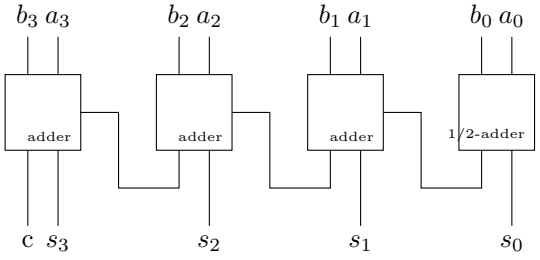
L'additionneur 4 bits va fonctionner selon la méthode suivante :

— on ajoute les chiffres des unités produisant le chiffre des unités du résultat et une retenue éventuelle,
— on ajoute les chiffres des "dizaines" et la retenue des unités produisant le chiffre des "dizaines"du résultat et une retenue éventuelle,
— on ajoute les chiffres des "centaines" et la retenue des dizaines produisant le chiffre des "centaines" du résultat et une retenue éventuelle,
— ...

La première étape est réalisée avec un demi-additionneur et les suivantes avec des additioneurs.

Voici une réalisation d'un additionneur 4 bits selon cette méthode. Son inconvénient est que les retenues se propagent d'un additionneur à l'autre en prenant du temps.

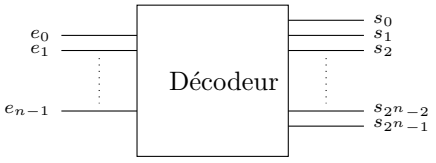
Cela est dû au fait que l'implantation d'un circuit est un dispositif physique.



3.2.3 Le décodeur

Nous allons maintenant voir quelques circuits combinatoires qui peuvent servir de briques pour construire des circuits plus complexes.

Un décodeur décode un nombre codé en binaire en activant la ligne correspondant à ce nombre. Il comprend n entrées et 2^n sorties. La $i$ème sortie de décodeur vaut 1 si les n entrées forment l'entier i, *ie* $(e_n e_{n-1} \dots e_1 e_0)_2 = (i)_{10}$.



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

e_1	e_0	s_3	s_2	s_1	s_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

$s_0 = \overline{e_0}.\overline{e_1}$
$s_1 = \overline{e_1}.e_0$
$s_2 = e_1.\overline{e_0}$
$s_3 = e_0.e_1$

Utilisation du décodeur

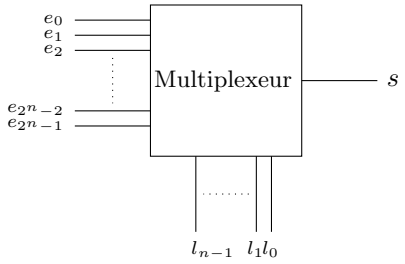
— Dans une UAL : supposons que nous ayons une puce capable de quatre opérations élémentaires (appelons-les, pour faire simple, opérations A, B, C et D). En attribuant un "code opérationnel" binaire à chacune de ces opérations, comme ceci :
  — code de l'opération A : 00
  — code de l'opération B : 01

— code de l'opération C : 10
— code de l'opération D : 11
Un décodeur judicieusement placé peut servir à activer le circuit correspondant à l'opération demandée. Dès lors, ces deux lignes d'entrée agissent comme des "lignes de commande" et le décodeur mérite vraiment son nom puisqu'il se comporte ici comme un véritable décodeur d'instruction.
— Gestion de la mémoire : la mémoire composée de circuits électroniques, est organisée comme une gigantesque matrice de petites cases numérotées, chaque case étant capable d'abriter un nombre binaire de taille fixe. Un décodeur permet d'accéder à cette mémoire puisque il suffira de transmettre l'adresse de la case mémoire réclamée aux lignes d'entrée du décodeur pour que celui-ci active la cellule mémoire correspondante.

3.2.4 Le multiplexeur

Un multiplexeur comporte 2^n entrées, 1 sortie et n lignes de sélection (entrées). Il recopie sur sa sortie la valeur de la ligne d'entrée dont le numéro est codé en binaire sur les lignes d'entrées.

La sortie du multiplexeur vaut la valeur de la $i$ème entrée si l'entier i est codé sur les lignes de sélection, *ie* $(l_n l_{n-1} \dots l_1 l_0)_2 = (i)_{10}$.



3.3 Unité arithmétique et logique

L'unité arithmétique est logique (UAL ou ALU) est le composant d'un ordinateur chargé d'effectuer les calcul. 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, et leur équivalents « ou égal »,
— 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,
— modulos

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 transcendantales (sin x, cos x, etc.), opération vectorielle (produit scalaire, vectoriel, etc.), etc.

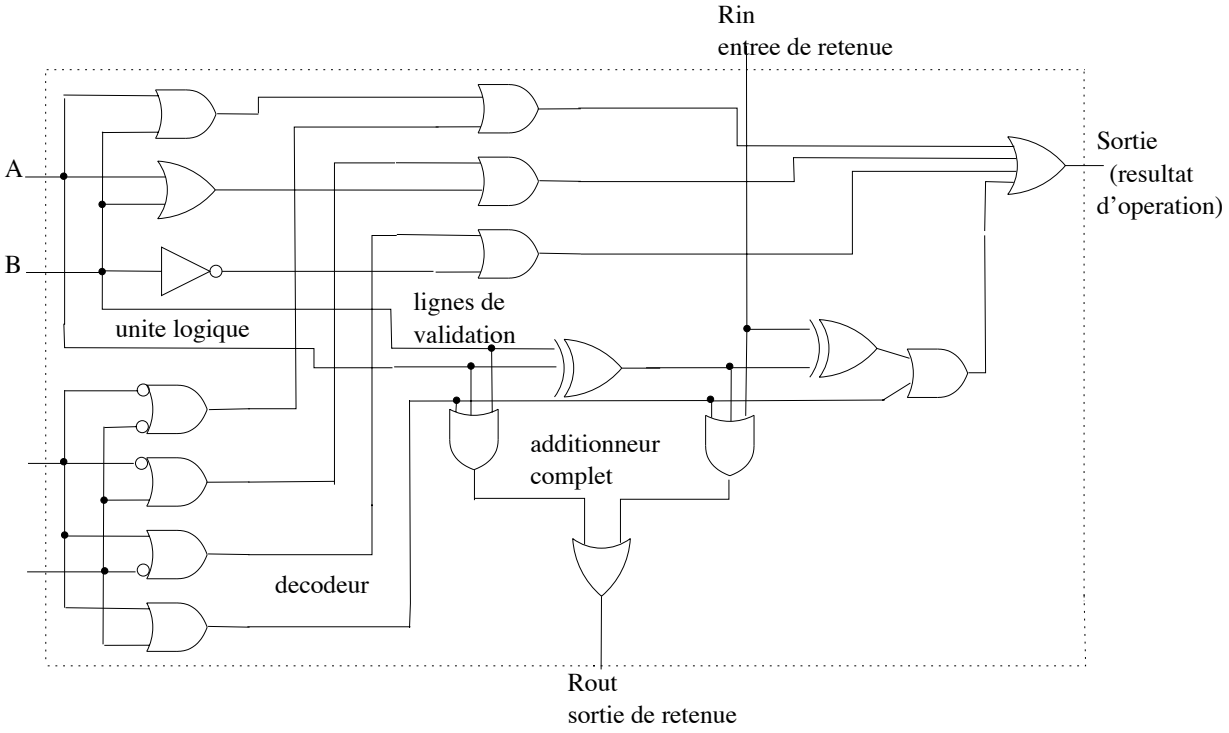
Un processeur fait appel à plusieurs UAL.

Détaillons une UAL 1 bit (les calculs sont fait sur des mots de 1 bit) : elle est capable de réaliser 4 opérations pour ses entrées A et B, à savoir, $\bar{B}$, A et B, A ou B et $(A + B) + Rin$; le résultat est alors présent sur la sortie et lorsqu'il s'agit d'une addition, la retenue de sortie est sur $Rout$ (qui vaut 0 pour les autres opérations).

Le décodeur est utilisé pour sélectionner l'opération à réaliser : on remarque que les sorties du décodeur vont activées ou désactivées au travers de portes ET (lignes de validation) les résultats

des différentes opérations, ne laissant passer que l'un d'entre eux. On remarque que pour les entrées 00 du décodeur, c'est A et B qui est calculé et pour 11, $(A + B) + Rin$.

ALU 1 bit (et,ou,non,addition)



Chapitre 4

Complément sur les tables de Karnaugh

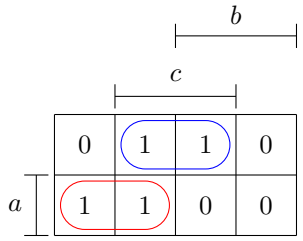
4.1 Les d'aléas

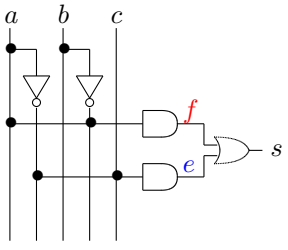
Les aléas dans les circuits combinatoires sont des comportements non conformes à la table de vérité correspondant au circuit. Ils sont dûs principalement au temps de franchissement des portes.

4.1.1 Dissection d'un aleas

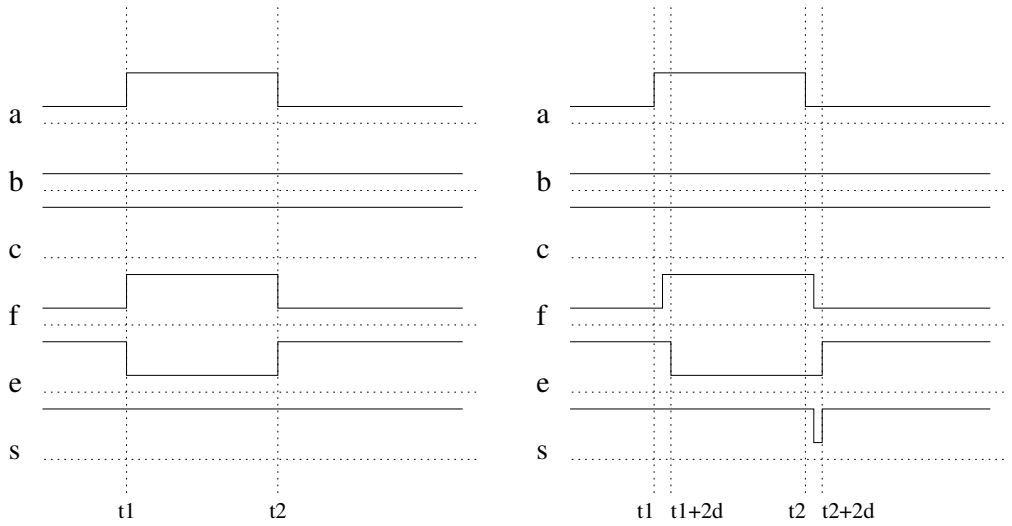
Imaginons que l'on doive construire un circuit réalisant la fonction booléenne suivante. On construit sa table de Karnaugh et on décide de choisir les deux regroupements rouge et bleu, ce qui nous donne le circuit suivant :

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





Observons maintenant le chronogramme du circuit lorsque $c = 1$, $b = 0$ et que a passe de 1 à 0 : Idéalement, le chronogramme serait celui de gauche, or, compte tenu du temps de franchissement des portes (supposons pour simplifier que le délai de franchissement d'une porte quelconque est d), on obtient en réalité le chronogramme de droite.



On voit en sortie un comportement qui n'est pas conforme à celui attendu : pendant un très bref moment (le temps de passage d'une porte), la valeur de la sortie passe à 0 alors qu'elle devrait rester toujours à 1.

4.1.2 Prévoir les aléas

Comment aurait-on pu prévoir cet aléa ? En regardant la table de Karnaugh. Elle montre un problème d'aléa à chaque fois qu'elle contient deux blocs adjacents sans intersection. Il y aura donc forcément un saut d'un bloc à l'autre lors de l'inversion d'une certaine valeur. Si on regarde la table de notre exemple, le 1 à droite du bloc rouge est adjacent au 1 à gauche du bloc bleu, et ces deux 1 n'appartiennent pas à un troisième bloc. On peut sauter d'un bloc à l'autre en fixant $c = 1$ et $b = 0$ et en modifiant la valeur de a : si a passe de 1 à 0, on saute du bloc rouge au bloc bleu et lorsque a passe de 0 à 1, on saute du bloc bleu au bloc rouge.

Ce sont ces sauts qui provoquent les aléas.

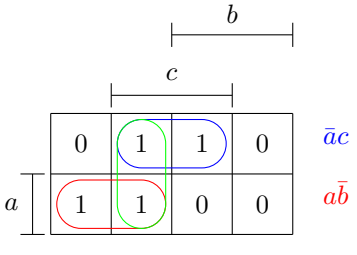
— Lorsque a passe de 0 à 1 au temps t_1, on est alors dans le bloc bleu ($c\bar{a}$) dont on sort après un temps $2d$, et on passe au bloc rouge ($a\bar{b}$) en un temps d, on est donc à tout moment dans un bloc et la sortie reste ainsi à 1.

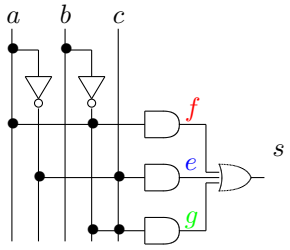
— Lorsque a passe de 1 à 0 au temps t_2, on est alors dans le bloc rouge dont on sort (i.e., f passe à 0) après un délai d. On entre par contre dans le bloc bleu (i.e., e passe à 1) qu'après un délai $2d$ correspondant au temps nécessaire au signal pour franchir la porte NON*puislaporte*ET. Donc entre le temps $t_2 + d$ et $t_2 + 2d$, la sortie passe à 0. Il se produit un aléa.

En conclusion : dès qu'on passe d'un bloc b_1 à un bloc b_2 par la modification d'une valeur x, et sans rester dans un troisième bloc, on est susceptible d'avoir un aléa.

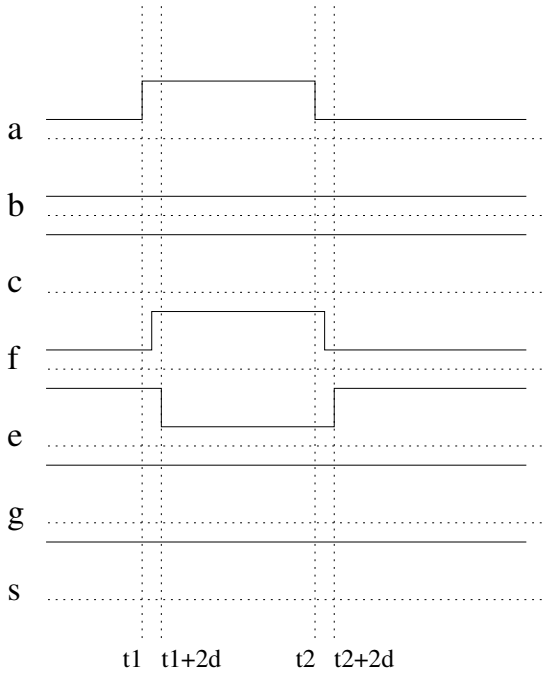
4.1.3 Eviter les aléas

Pour éviter les aléas, il suffit donc de ne pas avoir de saut brutal d'un bloc à l'autre de la table. On ajoute donc les blocs nécessaires. Dans notre exemple, on ajoute le bloc vert :





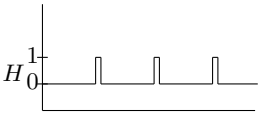
Que se passe t-il maintenant lorsque $c = 1$, $b = 0$ et que a passe de 1 à 0 ? La valeur de la sortie reste à 1 grâce au bloc vert $\bar{b}c$. On a supprimé l'aléa en ajoutant une porte.



4.1.4 Conclusion

Les tables de Karnaugh permettent donc d'éviter certains problèmes d'aléas (pas tous). Les aléas sont souvent très difficiles à détecter car ils se produisent sur des durées souvent très courtes, les rendant impossibles à visualiser sur un oscilloscope. Il n'est pas toujours indispensable de se préoccuper des aléas. La plupart du temps, une variation extrèment brève d'une sortie n'entraînera pas de dysfonctionnement du circuit. Par exemple, pour un système d'affichage, il n'est pas gênant que l'affichage soit perturbé pendant un temps très court. Enfin, ce phénomène d'aléas peut être utilisé pour créer des horloges ayant des durées à 1 ou à 0 très brèves.

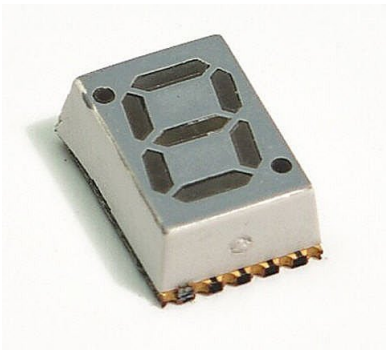
Voici le chronogramme d'une horloge de ce type :

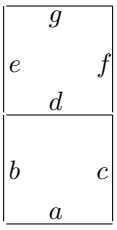


4.2 Avantages des tables de Karnaugh

Les tables de Karnaugh s'avèrent très utiles lorsque certaines valeurs de sortie du systèmes peuvent être indifféremment 0 ou 1. Ce cas arrive lorsque certaines entrées du système sont impossibles.

Prenons le cas d'un afficheur 7 segments, c'est celui qu'on trouve par exemple sur les montres à affichage digital.

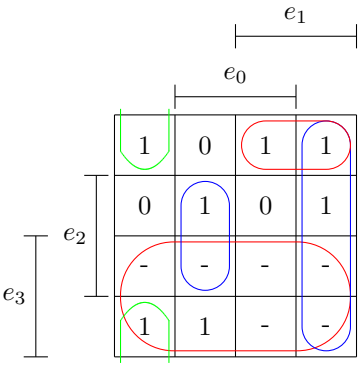




Le système reçoit en entrée un nombre compris entre 0 et 9 codé sur 4 bits $e_3e_2e_1e_0$ et en sortie illumine certains des 7 segments a, b, c, d, e, f, g de façon à afficher le nombre reçu en entrée. Il y a donc seulement 10 entrées utilisées parmi les 16 possibles.

Ecrivons par exemple la table de vérité de la diode a. Les 6 dernières valeurs de a nous importent peu car les entrées correspondantes ne peuvent pas se présenter. Dans la table de Karnaugh correspondante, on choisira donc de placer des 1 ou des 0 à la place des points d'interrogation, de la façon qui nous conviendra le plus. Dans cet exemple, ils sont tous mis à 1 afin de former le gros bloc rouge.

e_3	e_2	e_1	e_0	a
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	?
1	0	1	1	?
1	1	0	0	?
1	1	0	1	?
1	1	1	0	?
1	1	1	1	?

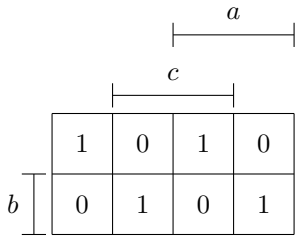


Remarquez que nous somme typiquement dans le cas où les aléas nous importent peu, car un changement d'affichage d'une micro seconde sera invisible pour l'oeil humain.

4.3 Inconvénients des tables de Karnaugh

L'inconvénient majeur des tables de Karnaugh est que l'on rate certaines simplifications lorsque le circuit aurait pu se construire plus facilement avec des portes NON-ET ou des portes OU-X, cela ne se voit pas directement avec les tables. Il est donc important de bien connaître les règles de simplification comme les lois de Morgan par exemple, et les propriétés du $\oplus$.

Par exemple, un des pires cas est celui où on ne fait que de regroupements de taille 1 comme dans l'exemple suivant :



On obtient l'équation $s = \bar{b}\bar{a}\bar{c} + bc\bar{a} + \bar{b}ca + b\bar{c}a$
qui se simplifie de la façon suivante :

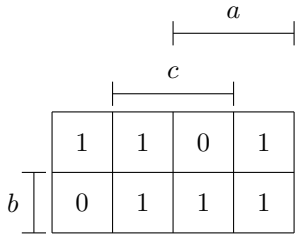
$$\begin{aligned} s &= \bar{b}(\bar{a}\bar{c} + ca) + b(c\bar{a} + \bar{c}a) \\ &= \bar{b}\overline{(c\bar{a} + \bar{c}a)} + b(c\bar{a} + \bar{c}a) \\ &= \bar{b} \oplus (c\bar{a} + \bar{c}a) \\ &= \bar{b} \oplus (a \oplus c) \end{aligned}$$

Il suffit donc de 2 portes OU-X pour réaliser le circuit.

4.4 Regroupement de 0 dans les tables de Karnaugh

Pour terminer, remarquons que dans certains cas, il est plus judicieux d'effectuer des regroupements de 0, puis d'inverser la sortie du circuit.

Par exemple, dans la table suivante il faut au moins 3 regroupements de 1, et 6 si on veut éviter les aléas, alors qu'il suffit de 2 regroupements de 0.



Il sera donc plus judicieux d'écrire la sortie $s = \overline{b\bar{c}\bar{a} + \bar{b}ca}$.

Chapitre 5

Logique électronique, CMOS

Cette section a un double but : expliquer les circuits au niveau des transistors, éléments de base de la réalisation électronique contemporaine des circuits ; puis, introduire un élément essentiel de circuits complexes, les « buffers Tri-State », qui s'exprime avec des réseaux de transistors, mais pas avec des portes logiques.

5.1 Codage par tension

La réalisation électronique de la logique se base sur la représentation des valeurs 0 et 1 par des tensions relatives à une référence, la « terre ». Par exemple, en logique TTL, un 1 est idéalement représenté par $5V$ et un 0 est idéalement représenté par $0V$. En pratique, il y a des seuils de tolérance, par exemple une tension inférieure à $1,5V$ serait toujours correctement interprétée comme 0 et une tension supérieure à $3,5V$ serait interprétée comme 1.

D'autres logiques, notamment celles utilisées dans des microprocesseurs modernes, travaillent sur le même principe, mais avec des tensions plus basses.

En électronique, on trouve souvent la désignation « VSS » ou « Gnd » (« ground » comme terre) pour la tension correspondante à 0 et « VDD » ou « VCC » pour la tension correspondante à 1.

L'objectif de la réalisation électronique de portes logiques consiste alors en la transformation d'entrées (codées en tensions) en sorties correspondantes (codées également en tensions).

5.2 Logique électro-mécanique

La logique électromécanique est obsolète, mais nous l'introduisons pour préparer la compréhension des réalisations en semi-conducteurs.

Un relais est un interrupteur contrôlé par un aimant électrique : Lorsqu'il y a un courant dans la bobine, un champs magnétique est crée qui bouge un interrupteur d'une position à l'autre. Le courant est crée grâce à une tension entre les deux contacts de la bobine.

La Figure 5.2 montre la réalisation d'un inverseur en logique électro-mécanique : lorsqu'il n'y a pas de tension suffisante à l'entrée (donc notamment pour une entrée proche de $0V$ ou de 0), l'aimant n'opère pas et la sortie est reliée directement à VDD, la tension représentant le 1 ; en revanche, avec une tension suffisante (donc pour un 1 logique) à l'entrée, l'aimant tire l'interrupteur en position haute et relie la sortie à la terre, donc 0. Ce circuit réalise non seulement la fonction logique d'un

FIGURE 5.1 – Terre et tension « 1 » comme symboles électroniques.

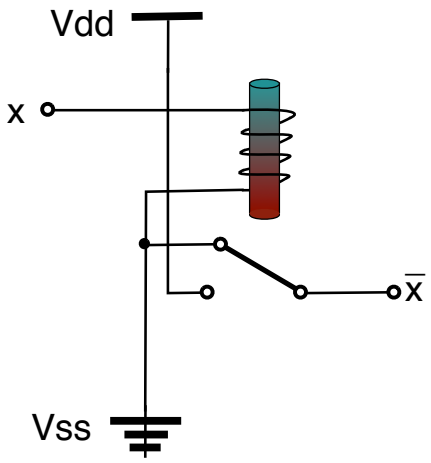


FIGURE 5.2 – Inverseur avec relais

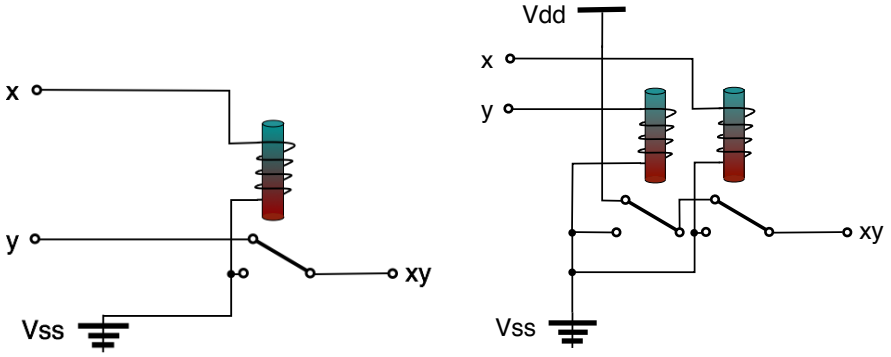


FIGURE 5.3 – « AND » avec un et avec deux relais

inverseur, mais peut également améliorer la qualité du signal à la sortie puisque le 1 produit est obtenu directement de VDD. Cet aspect de sa fonction est appellée « buffer » en électronique, à ne pas confondre avec la notion d'un « tampon » en informatique !

Partant de ce principe, on peut réaliser des portes plus complexes. Dans Figure 5.3 on voit deux réalisations pour un « AND », la première utilisant un seul relais en codant `xy = (x ? y : 0)`, la seconde en codant `xy= (x?(y?1:0):0)`. La différence fondamentale est que la deuxième construction permet de produire de bonnes sorties même si les entrées sont médiocres, alors que pour le premier cas, la tension sortante ne peut jamais dépasser celle de y. Bien qu'utilisant deux fois le nombre de relais, la deuxième solution représente le coeur de la technologie numérique, la possibilité de copier des données sans dégradation.

Le fonctionnement avec relais a été utilisé dans plusieurs ordinateurs historiques, notamment l'ordinateur Z3 [1] de Konrad Zuse (1941 à Berlin) et le « Mark I » developpé par Howard Aiken (1944 IBM/Harvard). Ils étaient géants, brouillants et très gourmands (courants dans les relais !).

1. Ce peut-être premier ordinateur de l'histoire a été détruite pendant la guerre, et a été reconstruit dans les années 1960 par Konrad Zuse lui-même et est exposé dans le Deutsches Museum à Munich. Une démonstration de la machine sur youtube existe ici : https://www.youtube.com/watch?v=aUXnhVrT4CI. . L'original avait environs 2500 relais et consommait 4kW pour une vitesse assez limitée : une multiplication de deux nombres flottants avec exposant 7 bit et mantise 14 bit prennait environs 3s.

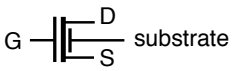


FIGURE 5.4 – schéma 4 pin d'un transistor à effet de champ

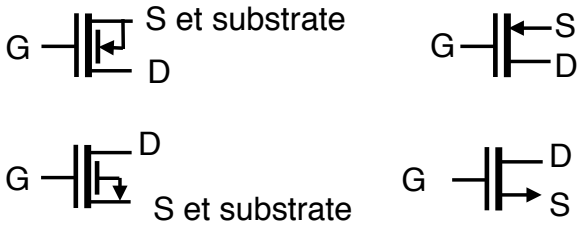


FIGURE 5.5 – Symboles pour transistors p-MOS (en haut) et n-MOS (en bas).

5.3 Transistors comme interrupteurs

Les relais ont été rapidement remplacés par des tubes électroniques pour des raisons de leur manque de vitesse et de fiabilité [2]. A partir de 1954, les tubes, utilisés comme amplificateurs et interrupteurs électroniques, ont été remplacés par les transistors bipolaires, des semi-conducteurs qui ont rendus possibles la révolution électronique et finalement informatique. Les années 1960 ont vu l'invention des circuits intégrés (puces) et le premier microprocesseur, le Intel 4004 [3], a vu le jour en 1971.

Ici, on discute seulement des transistors à effet de champ (FET, field effect transistor) et sans entrer dans la discussion du fonctionnement physique [4]. Le schéma général des transistors à effet de champs est donné dans Figure 5.4 : le principe est que la tension (et donc le *champ* électrique) entre Gate et Substrate contrôle les propriétés de conductivité (semi-conducteur !) entre Source et Drain. Deux types de transistors sont utilisés dans la logique CMOS : le n-MOS et le p-MOS.

Pour nos fins, un modèle simplifié suffit : Pour un transistor n-MOS, une tension positive entre G (gate/grille) et Substrat(e) crée une connexion entre D (Drain) et S (Source) (en analogie au relais qui ferme un interrupteur). En absence de tension positive (de niveau suffisant), la liaison S-D est ouverte, coupée. Pour un transistor p-MOS, il faut par contre une tension négative entre G et Substrate pour établir la liaison S-D.

En pratique, dans des transistors de type n-MOSFET, Source et Substrat sont liés et devraient être reliés à la terre (ou à rien, en tout cas jamais à VDD), et en dualité pour les transistors de type p-MOS, Source et Substrate sont liés et devraient être connectés à VDD (ou à rien, en tout cas jamais à VSS/terre). Ainsi, une tension de niveau suffisante (un «1») à G relie D à S pour un transistor n-MOS, alors qu'un signal proche de 0 à G fait que D est déconnecté de S. Pour un transistor p-MOS, c'est le 0 à l'entrée Gate qui lie D à S et c'est le 1 qui les découple.

Cet usage fait que les transistors p-MOS et n-MOS sont représentés avec trois contacts et avec une flèche qui indique l'orientation de la tension pour le fermer, voir Figure 5.5. A droite, un symbole plus simple fréquemment utilisé est montré qui est également utilisé dans TkGate.

Le fonctionnement peut donc se résumer de la façon suivante :

— pour un transistor p-MOS : si S est lié à VDD, alors une entrée à 0 en G engendre un 1 en sortie D, alors qu'une entrée à 1 en G découple D d'un point de vue électrique : il n'y a aucune sortie, ni 0 ni 1.

2. Pour l'anecdote, l'expression « bug » trouve son origine avec Mark I : un papillon de nuit (« moth ») s'est introduit dans un relais et a causé un dysfonctionnement. Les ingénieurs chez IBM auraient par la suite mis la responsabilité pour tout problème sur des « bugs » (« insectes » en américain).

3. Ce fut un microprocesseur travaillant sur des mots de 4bits destiné à réaliser une calculatrice et de faire du calcul décimal. Il utilisait environs 2300 transistors, curieusement proche des 2500 relais de la Z3

4. Le lecteur curieux consultera par exemple l'article Wikipedia au sujet « MOSFET » (Metal Oxyde Semiconductor Field Effect Transistor. Différemment des tubes et des transistors bipolaires qui servent comme amplificateur de courant, les transistors à effet de champ ne nécessitent pas de courant mais juste d'une tension pour le contrôle.

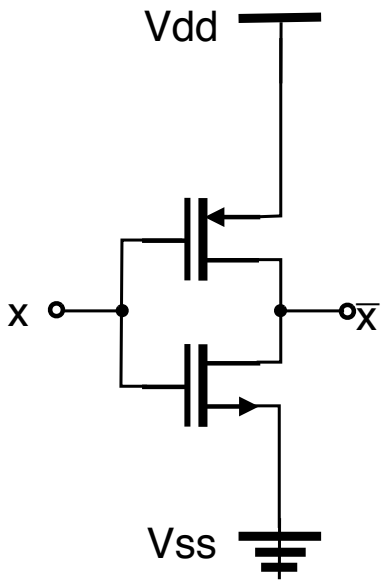


FIGURE 5.6 – inverteur CMOS

— pour un transistor n-MOS : si S est lié à VSS, alors une entrée à 1 en G engendre un 0 en sortie D, alors qu'une entrée à 0 en G découple G.

La combinaison des deux transistors réalise donc un inverseur, voir Figure 5.6.

Plus généralement, en logique C-MOS (C pour complementary), on combine deux circuits, pour la réalisation d'une fonction logique :

— un qui réalise les sorties 1 réalisé avec des transistors p-MOS, et qui découple la sortie pour les cas de 0.
— un qui réalise les sorties 0 avec des transistors n-MOS et qui découple la sortie pour les cas 1.

Ces deux circuits sont combinés par une liaison électrique des sorties qui donne donc toujours 1 ou 0. Attention, une combinaison de deux tels circuits peut être partielle (découplage pour les deux demi-circuits), mais ne doit pas lier à 1 (pour la partie p-MOS) et 0 (pour la partie n-MOS) : un tel cas représente un court-circuit et peut engendrer un courant conduisant à la destruction physique du circuit.

Il est possible de mettre en parallèle ou en série des transistors du même type. Plusieurs transistors en ligne (D de l'un lié à S de l'autre) deviennent conducteurs s'ils sont tous conducteurs, donc si tous les G sont à 0 pour des transistors type p ou tous les G à 1 pour des types n. En revanche, lors de la mise en parallèle de plusieurs transistors, il suffit qu'un d'entre eux soit conducteur.

Un exemple d'une porte avec deux entrées est un NOR réalisé en CMOS, voir figure 5.3. Ce schéma utilise une mise en série de transistors dans la partie p-MOS (1 si les deux entrées sont 0 !) et une mise en parallèle dans la partie n-MOS (0 si au moins une entrée est 1).

Dans la conception de circuits plus complexes, il est utile de prévoir pour chaque variable a un signal a et son complément $\overline{a}$. Ainsi, on peut obtenir ab en tant que $\overline{\overline{a} + \overline{b}}$ (donc en appliquant une porte NOR aux entrées en complément) et le complément $\overline{ab}$ directement par une porte NAND. A la fin, on peut éliminer les signaux et circuits non-utilisés.

Combien de transistors peut-on mettre en série ou en parallèle ? En parallèle, il n'y sur le principe pas de limite, mais en série, les choses sont plus compliqués : la tension GD (ou GS) dépend de l'état ouvert/fermé des transistors sur le chemin vers 1 (ou 0). Si un des transistors sur le chemin est ouvert, alors il n'y a pas de tension, si tous les transistors sont fermées, alors il peut y avoir tension, mais cette tension est réduite par les chutes de tension DS sur chaque transistor sur le chemin. Deux conséquences : (a) une limite sur le nombre de transistors qu'on peut mettre en série en garantissant le fonctionnement correct pour une tension entre 1 et 0 donnée et (b) un temps de stabilisation qui dépend du nombre d'entrées. Au delà de la limite du nombre d'entrées

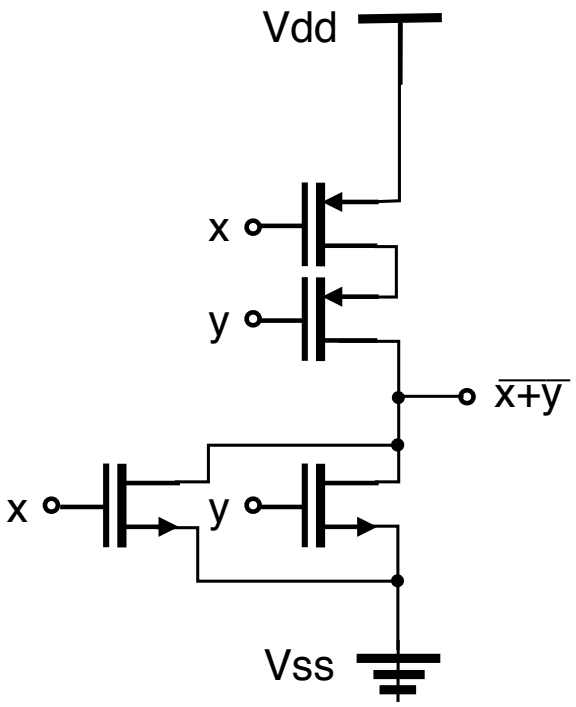
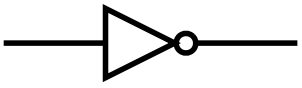


FIGURE 5.7 – porte NOR en CMOS

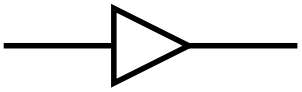
techniquement possibles, seulement une cascade arborescente permet des portes de plus grande taille – au prix d'un ralentissement également.

5.4 Amplification

Selon les caractéristiques des transistors utilisés, les sources 1 et 0 liées à la sortie permettent un courant plus fort que celui utilisé pour le changement d'état de l'inverseur et permettent au besoin d'alimenter des entrées de plusieurs portes ou même de consommateurs électriques. On appelle de ce fait cette porte un buffer [5] inverter, désigné par un petit triangle avec un rond de complémentation.



Le triangle est également utilisé sans négation, un buffer :



D'un point de vue logique, un buffer (qui peut être réalisé en enchainant deux buffers inverters) donne le même signal logique à la sortie qu'il a à l'entrée, mais le courant sortant ne provient pas du courant entrant mais de son alimentation VDD et VSS.

5.5 Logique à trois états

Le principe des circuits CMOS composé de deux moitiés, une qui lie la sortie à 0 (ou pas), l'autre qui lie la sortie à 1 (ou pas), peut être compris comme une logique à trois valeurs, 0, 1 et Z (ou « rien »). Z correspond à une sortie déconnectée.

5. à ne pas confondre avec l'usage du terme « buffer / tampon » en informatique !

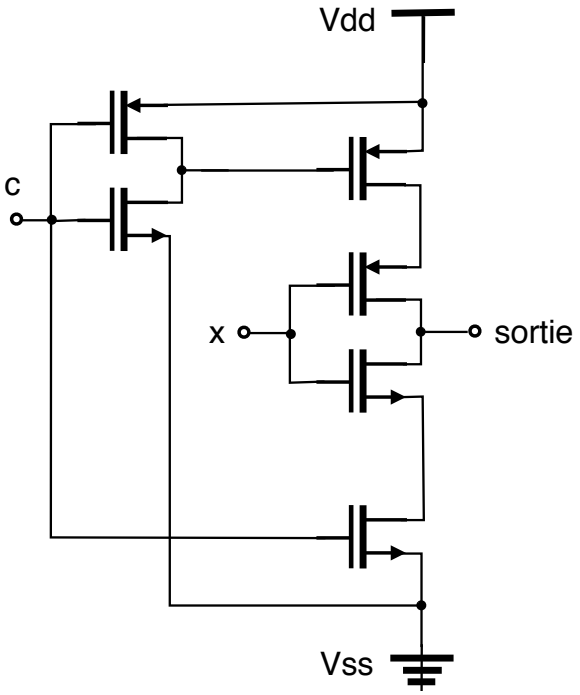


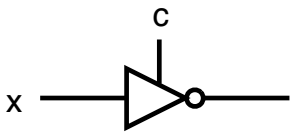
FIGURE 5.8 – Réalisation CMOS d'un buffer inverter tri-state

Si deux câbles qui portent des signaux 1 et 0 respectivement sont liées, cela crée un court circuit, en pratique un courant très fort et destructeur. En revanche, une combinaison de 0 et 0 est toujours 0, 1 et 1 est toujours 1 et une combinaison Z et 1 est 1 et une combinaison de Z et 0 donne 0.

Dans la logique CMOS, on combine deux moitiés de portes, une avec sortie 1 ou Z, l'autre avec sortie 0 ou Z, mais avec la garantie de complémentarité de couples $(1, Z)$ ou $(Z, 0)$ et jamais $(1, 0)$ ou (Z, Z), donc avec la garantie que la combinaison donne toujours 1 ou 0 (du moins après une intervalle de stabilisation).

Au delà de cette application, il existent des circuits où Z est utilisé pour coder véritablement une valeur, par exemple une logique avec des valeurs 0 et Z. Cette logique généralise de circuits logiques standard dans la mesure où plusieurs sous-circuits peuvent émettre une sortie sur le même câble.

Un composant fortement utile dans la conception de circuits complexes est le buffer « tri-state » (trois états, trois valeurs logiques) : il a les mêmes entrées et sorties qu'un buffer plus une entrée de contrôle. Voici son symbole pour le cas d'une sortie inversée :



Les entrées x et S doivent être des valeurs logiques (donc 0 ou 1) mais la sortie peut admettre les trois valeurs 0, 1, ou Z.

Une réalisation d'un buffer inverteur tri-state en CMOS peut se faire comme indiquée dans Figure 5.8

Le buffer tri-state permet notamment une réalisation alternative au multiplexeur qui se passe de la porte OR pour la combinaison, voir Figure 5.9. L'entrée `ch` est décodée et seulement un des buffers tri-state laissera passer une des quatres entrées sur la sortie.

Par rapport à la réalisation avec une grande porte OR dont la vitesse dépend du nombre d'entrées, la réalisation avec une logique tri-state permet deux choses : (a) un circuit plus rapide et (b) un cablage plus simple.

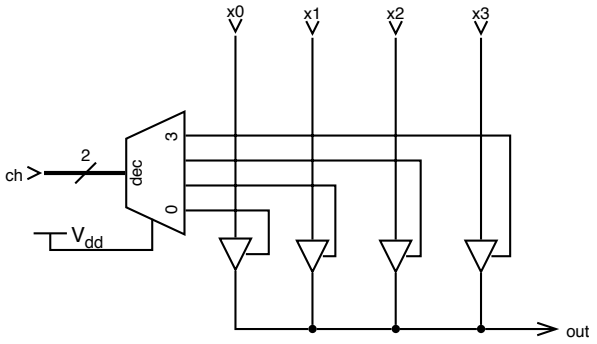


FIGURE 5.9 – Réalisation d'un multiplexeur avec des buffers tri-state

5.6 Calcul et Energie

La particularité des transistors FET est qu'une tension/charge suffit pour garder leur état et qu'il n'y a pas de courant entre G et D ou S. Un changement à l'entrée d'un inverseur de 0 à 1 par exemple engendre un chargement positif du transistor type n et un déchargement négatif (donc chargement positif) du type p et pour le passage de 1 à 0, c'est le contraire. Sans changement à l'entrée, il n'y a aucun courant sur G. En résumé, une porte CMOS engendre des courants lors d'un changement d'entrées plus les courants tirées aux sorties. Si les sorties sont des entrées d'autres portes d'un circuit combinatoire, on voit que ce circuit consomme de l'énergie pour l'essentiel seulement pendant des changements des entrées et des propagations de ces changements, donc en calculant de nouvelles sorties !

Dans les technologies obsolètes (comme les relais, mais aussi des logiques plus gourmandes comme TTL), il faut en revanche de l'énergie pour maintenir l'état du circuit, même sans calcul. La logique TTL dominait sur CMOS durant les années 1970 et 1980 à cause des contraintes d'espaces (nombre de composants sur une puce), CMOS nécessitant jusqu'à 2 fois plus de transistors. Mais les progrès en intégration ont inversé la tendance et la grande majorité des circuits intégrés utilise CMOS aujourd'hui.

Il est à noter aussi que la disponibilité de plus de composants permet une conception de circuits qui optimise la consommation énergétique en figeant l'état de sous-systèmes où cela est possible. Par exemple la conception logique d'une ALU consiste à router des arguments à différents circuits de calcul arithmétique et logique et d'extraire de l'ensemble des résultats calculés celui désiré par un multiplexeur. Fonctionnellement, cette conception est correcte mais elle engendre un fonctionnement gaspilleur parce que la plupart des résultats de ces calculs sont jetés. On peut optimiser l'énergie consommé au prix d'un petit délai en ajoutant un filtre à l'entrée de ces circuits calculateurs qui laisse passer d'autres valeurs que 0 seulement lorsque l'opération est nécessaire. Par conséquent, le circuit change d'état seulement quand l'opération est sollicitée.

5.7 Vitesse et Energie

Le changement d'entrée G, correspondant à la charge ou décharge d'un condensateur au point d'un effet fiable (liaison ou pas entre D et S) prend du temps, le délai du transistor. Les mêmes transistors peuvent être utilisés à différents tensions, par exemple 5V, 3.3V, 1V, 0.8V sont utilisés de nos jours avec une tendance vers des tensions de plus en plus bas. Comme pour un condensateur, plus qu'il y a tension, plus il y a de charge et donc de courant pour le changement d'état. En revanche, une tension plus élevé engendre un changement d'état fiable plus rapide. En résultat, une plus haute tension permet un calcul plus rapide mais conduit à plus de courants, plus de consommation d'énergie et donc aussi plus de chaleur à dissiper.

L'antagonisme entre l'économie d'énergie (et si ce n'est pour le problème de chaleur qui peut endommager le circuit ou son environnement) et de vitesse est le challenge de l'informatique de notre époque. La miniaturisation des transistors réduit également leur « capacité » et augmente de

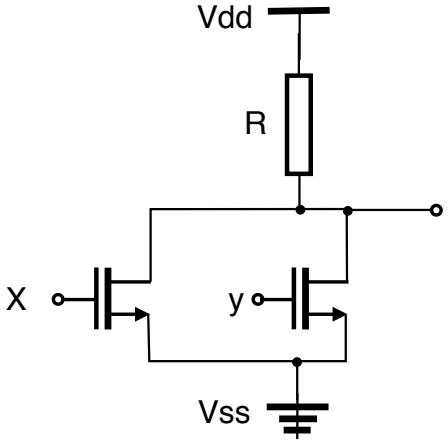
ce fait leur vitesse tout en réduisant leur consommation d'énergie. Alors que cette tendance à été utilisé dans les années 90 d'augmenter la cadence des processeurs, on a atteint un seuil technique pour cette façon d'accélération de calcul. Aujourd'hui, l'objectif premier est de rendre le calcul plus économe en énergie. Deux circuits opérant à la moitié de vitesse et en revanche à une tension plus basse consomment moins d'énergie. La parallélisation massive et sur tous les niveaux (au delà du « multi core » est la conséquence pour les années à venir.

Le jeune informaticien fait bien de s'approprier des connaissances en programmation parallèle, concrètement dans la programmation multi-thread par la programmation de cartes graphiques (Cuda, OpenCL !), mais surtout comme façon de penser les algorithmes et données.

5.8 D'autres logiques, l'exemple RTL

Il est possible de créer une logique à 2 valeurs, par exemple 0 et Z (rien) et d'utiliser un seul type de transistors (par exemple n-MOS) en utilisant des résistances pour la transformation de Z en 1 : une réalisation d'un circuit qui tire vers le bas (la partie n-MOS dans un circuit CMOS) est complétée par une résistance R reliée à VDD qui tire vers le haut (pullup).

Par exemple, on peut réaliser une porte NOR comme suit :



Dans cette réalisation, en cas de deux entrées 0, la sortie est simplement liée à VDD (1), certes passant par une résistance. La tension de sortie baisse en cas de courant (loi de Ohm, un courant sur R a pour conséquence une chute de tension), mais puisque les transistors n-MOS ne necéssitent pas de courant pour maintenir un état, cette chute s'achève une fois qu'un état stable est atteint.

En revanche, si une entrée est 1, la sortie est directement liée à VSS (0), ce qui entraîne un courant sur la résistance, différemment de CMOS où il n'y a jamais de liaison directe entre VDD et VSS dans une porte.

Cette logique s'appelle RTL (resistor-transistor logic) et a été utilisé dans les années 60 avec des transistors bipolaires (c'est un peu différent). Elle est plus gourmande en énergie que CMOS, mais elle est toujours utilisé pour certaines interfaces. Notamment, son intérêt consiste en la possibilité de mettre une grande nombre d'entrées en parallèle !

Par exemple, le bus I2C de Philips permet à plusieurs composants d'écrire et de lire sur le même bus : si aucun composant écrit, alors la valeur est Z, l'écriture se réduit alors à l'écriture de 0. Une résistance tire une valeur Z à 1. Un bus I2C représente à tout moment le NOR de toutes ses entrées. Il est prévu pour des centaines de noeuds.

5.9 CMOS et Verilog

Dans Verilog, les valeurs logiques 0, 1 et z (flottant) sont complémentées par une valeur x, la « valeur inconnue ». Cette valeur est une abstraction notamment pour des fins de simulation et est utilisé dans différents cas :

— Dans le cas d'une collision 0-1 sur le même câble (court-circuit).
— En tant que valeur initiale dans le cas de circuits circulaires (tels que les bascules, à venir).

La modélisation Verilog de portes logique tient compte de leur propriétés physiques (portes CMOS) et on obtient ainsi des tables de vérité avec $0, 1, X, Z$ comme valeurs, par exemple pour la porte AND :

a \ b	0	1	x	z
0	0	0	0	0
1	0	1	x	x
X	0	x	x	x
Z	0	x	x	x

Les transistors n-mos et p-mos sont modélisés en supposant le substrate lié à soit 1 (pour p-mos), soit 0 (pour n-mos). Ainsi, un transistor p-mos est déclaré comme suit avec deux entrées `source` et `gate` et une sortie `drain` :

```
pmos p (d, s, g);
```

La table de vérité pour `d` en fonction de `s` et de `g` est comme suit :

s\ g	0	1	X	Z
0	0	z	x	x
1	1	z	x	x
x	x	z	x	x
z	z	z	z	z

5.9.1 Retards en Verilog

Les retards en Verilog sont déterministes et sont spécifiés de différentes manières :

— En cas d'une spécification par équation d'une porte, une spécification `# n` signifie un délai de n unités de temps jusqu'à l'apparition d'un changement à la sortie en fonction d'un changement à l'entrée. Par exemple, la déclaration `assign #3 a = b + c;` signifie que a est calculé en tant que disjonction de b et de c avec un retard de 3 unités de temps.
— Au cas d'une utilisation de primitifs, le même cas peut être décrit ainsi : `or #3 p1 (a,b,c);`.

Chapitre 6

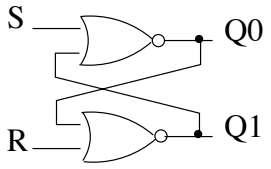
Circuits séquentiels

Les circuits séquentiels sont des circuits dont les sorties dépendent des entrées mais également **des valeurs antérieures des sorties**. 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.

La majorité des circuits séquentiels sont réalisés à partir de circuits séquentiels appelés **bascules**. Les bascules (latch) sont des dispositifs permettant de mémoriser un bit.

6.1 La bascule RS

La bascule RS est composée de deux portes NON-OU. Elle comporte deux entrées R et S et deux sorties $Q0$ et $Q1$. L'entrée S (pour Set) est utilisée pour la mise à l'état 1, et l'entrée R (pour Reset) pour la mise à l'état 0.



Ce circuit n'est pas un circuit combinatoire puisqu'il comporte un cycle. Clairement, les valeurs de Q0 et Q1 vont dépendre des valeurs antérieures de Q0 et Q1. Supposons que le temps de passage d'une porte NON-OU est ϵ, si $Q0$ et $Q1$ sont les valeurs des sorties à l'instant t, on note $Q0'$ et $Q1'$ les valeurs des sorties à l'instant $t + \epsilon$.

On a alors l'équation suivante :

$$Q0' = \overline{S + Q1} \text{ et } Q1' = \overline{R + Q0}.$$

Voici la table de vérité de ce circuit :

S	R	$Q0$	$Q1$	$Q0'$	$Q1'$
0	0	0	0	1	1
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	0	0
0	1	0	0	1	0
0	1	0	1	0	0
0	1	1	0	1	0
0	1	1	1	0	0
1	0	0	0	0	1
1	0	0	1	0	1
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	0	0
1	1	1	0	0	0
1	1	1	1	0	0

6.1.1 Etats de la bascule RS

Le circuit est dans un **état stable** si en laissant inchangées les entrées R et S, les sorties restent inchangées. Lorsqu'un état n'est pas stable, les valeurs des sorties oscillent au cours du temps. On est donc dans un état indésirable puisque le but est de construire un circuit capable de mémoriser une valeur.

Examinons les états que peut prendre la bascule RS en fonction des entrées R et S.

Si $R = 0$ **et** $S = 1$ Voici l'évolution du système au cours du temps :

t	$t+\varepsilon$	$t+2\varepsilon$	$t+3\varepsilon$	$t+4\varepsilon$	$\cdots$
$Q0$	0	0	0	0	$\cdots$
$Q1$	$\overline{Q0}$	1	1	1	$\cdots$

Conclusion : On est dans un état stable où on mémorise 1 dans $Q1$ et 0 dans $Q0$ (Set). Remarquons que $Q0 = \bar{Q}1$.

Si $R = 1$ **et** $S = 0$ Voici l'évolution du système au cours du temps :

t	$t+\varepsilon$	$t+2\varepsilon$	$t+3\varepsilon$	$t+4\varepsilon$	$\cdots$
$Q0$	$\overline{Q1}$	1	1	1	$\cdots$
$Q1$	0	0	0	0	$\cdots$

Conclusion : On est dans un état stable où on mémorise 0 dans $Q1$ et 1 dans $Q0$ (Reset). Remarquons que $Q0 = \bar{Q}1$.

Si $R = 0$ **et** $S = 0$ Voici l'évolution du système au cours du temps :

t	$t+\varepsilon$	$t+2\varepsilon$	$t+3\varepsilon$	$t+4\varepsilon$	$\cdots$
$Q0$	$\overline{Q1}$	$Q0$	$\overline{Q1}$	$Q0$	$\cdots$
$Q1$	$\overline{Q0}$	$Q1$	$\overline{Q0}$	$Q1$	$\cdots$

Si $Q_1 = Q_2$ les valeurs vont osciller, on est dans un état instable. Par contre, lorsque $Q1 \neq Q0$, on est dans un état stable puisque l'évolution est la suivante :

t	$t+\varepsilon$	$t+2\varepsilon$	$t+3\varepsilon$	$t+4\varepsilon$	$\cdots$
$Q0$	$Q0$	$Q0$	$Q0$	$Q0$	$\cdots$
$Q1$	$Q1$	$Q1$	$Q1$	$Q1$	$\cdots$

Conclusion : on est dans un état stable ssi $Q1 \neq Q0$. Dans ce cas les valeurs de $Q1$ et $Q0$ restent inchangés, on est donc dans un état de mémorisation.

Si $R = 1$ et $S = 1$ Voici l'évolution du système au cours du temps :

t	$t+\varepsilon$	$t+2\varepsilon$	$t+3\varepsilon$	$t+4\varepsilon$	$\cdots$
$Q0$	0	0	0	0	$\cdots$
$Q1$	0	0	0	0	$\cdots$

L'état est donc stable, les sorties sont stabilisées à 0. Toutefois, si on passe maintenant à $R = S = 0$, on obtient un état instable puisque $Q0 = Q1$:

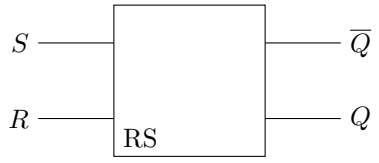
t	$t+\varepsilon$	$t+2\varepsilon$	$t+3\varepsilon$	$t+4\varepsilon$	$\cdots$
$Q0$	0	1	0	1	$\cdots$
$Q1$	0	1	0	1	$\cdots$

Cet état est donc indésirable, car un comportement normal peut faire basculer le système dans un état instable. L'état $R = 1$ et $S = 1$ est appelé **état indéfini**.

Conclusion : on est dans un état indéfini. On évitera donc soigneusement cet état.

6.1.2 Bascule RS : le circuit

Dans un fonctionnement normal de la bascule RS, Q_0 et Q_1 sont complémentaires. Ainsi, on note Q_1, Q et Q_0, $\overline{Q}$.



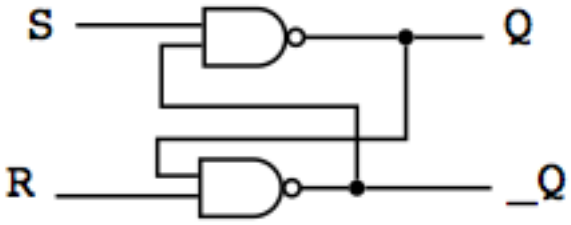
On considére donc le fonctionnement du circuit uniquement pour une sortie Q.

S	R	Q'
0	0	Q
0	1	0
1	0	1
1	1	indéfini

Le fonctionnement de la bascule RS peut être résumé comme suit : mettre S à 1 (et R à 0) met la sortie à 1 (**set**) tandis que mettre R à 1 (et S à 0) met la sortie à 0 (**reset**). Lorsque les deux entrées sont à 0, la bascule restitue en sortie la dernière action mémorisée sur la sortie (set ou reset).

6.1.3 Bascule RS : un autre circuit

Il est possible de construire une bascule RS en utilisant des portes NON-ET plutôt que des portes NON-OU.

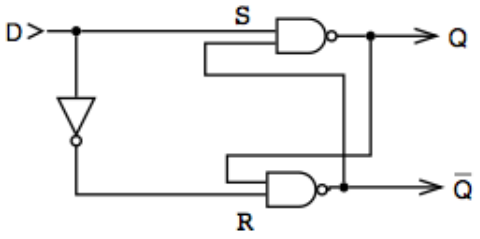


Il est à remarquer que dans ce circuit, l'état interdit est $R = S = 0$ et l'état de mémorisation est $R = S = 1$.

6.1.4 Bascule D

Une bonne façon de résoudre l'ambiguité propre à la bascule RS consiste à faire en sorte que l'état indéfini ne soit jamais présenté à l'entrée de la bascule. C'est l'idée de la bascule D qui ne dispose que d'une seule entrée :

Voici une réalisation d'une bascule D :



On a $S = D$ et $R = \bar{D}$, ainsi les valeurs de S et R sont toujours complémentaires.

Son équation est la suivante : $Q' = D$

6.2 Bascules synchrones

Le circuit précédent est dit **asynchrone** : les sorties évoluent dès que les entrées changent. Il n'y a pas de contrôle sur les instants où entrées et sorties changent.

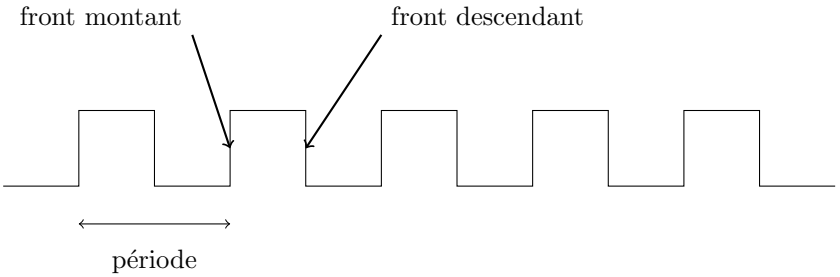
Ceci est bien sûr problématique si les différentes valeurs d'entrée ne sont pas toutes disponibles en même temps.

On opère alors à une **synchronisation** qui s'effectue à l'aide d'un signal impulsionnel de fréquence fixe appelé **signal d'horloge**.

Pour les circuits synchrones, les sorties évoluent seulement au signal de l'**horloge**.

6.2.1 Horloge

Voici le chronogramme d'une horloge, c'est à dire le graphe de la valeur de sortie de l'horloge en fonction du temps.



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

On considère un signal passant 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 appellés **front**). De plus, on suppose généralement que les temps passés au niveau haut et au niveau bas sont égaux.

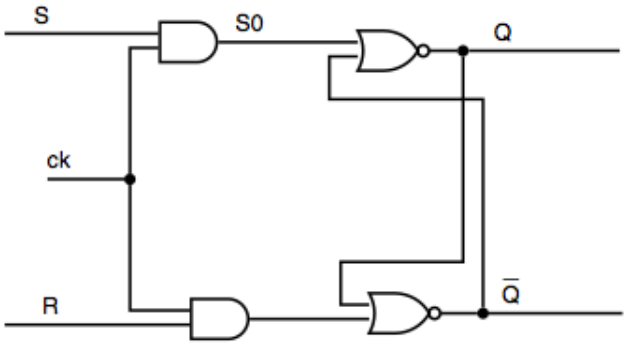
6.2.2 Modes de synchronisation

Il y a deux façons de synchroniser une bascule sur une horloge.

6.2.2.1 Bascules déclenchées sur niveau d'horloge :

Pour ce type de bascules, les entrées sont prises en compte pendant un niveau fixé de l'horloge ($CK = 1$ ou $CK = 0$). Si par exemple on choisi un déclenchement sur niveau haut, les modifications des entrées ne seront prises en compte que pendant les moments où $CK = 1$. Lorsque $CK = 0$, la bascule n'est pas déclenchée, on dit qu'elle est **verrouillée**.

Voici par exemple un circuit de bascule RS synchronisée sur niveau d'horloge.



Lorsque l'horloge est en état bas ($CK = 0$), les signaux S' et R' sont à 0, et les sorties Q et $\bar{Q}$ restent inchangées. Au moment où CK passe à 1, les portes ET n'ont plus d'effet et donc $S' = S$ et $R' = R$. Les entrées R et S reprennent donc le contrôle du système. Cette bascule se déclenche donc sur niveau haut.

Son équation est la suivante : $Q' = Q \cdot \bar{CK} + CK \cdot S \cdot \bar{R}$

6.2.2.2 Bascules déclenchées sur front d'horloge :

Pour certaines bascules, ce n'est pas la **position haute** (ou basse) de l'horloge (niveau d'horloge) qui les active mais le passage du **niveau bas au niveau haut** du signal d'horloge (ou inversement).

On parle alors de bascules **déclenchés par front d'horloge** :

— sur **front montant** quand l'horloge est mise de 0 à 1.

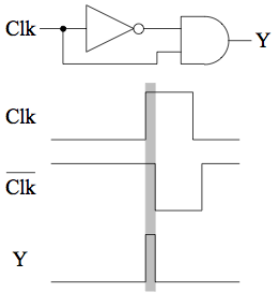
— sur **front descendant** quand l'horloge est mise de 1 à 0.

Le front **active** la bascule et les entrées sont prises en compte, le reste de la période la bascule est **verrouillé**.

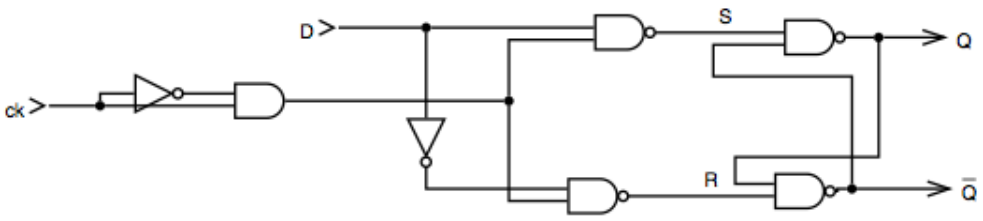
Le déclenchement sur front permet de mieux contrôler l'instant des actions.

Pour construire de telles bascules, on utilise le retard induit par le passage d'un inverseur.

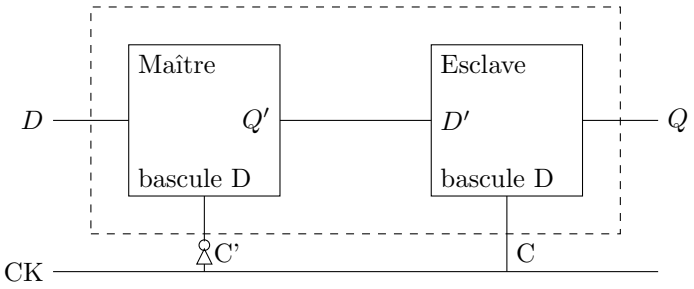
6.2.2.2.1 Circuit détecteur de front Le montage suivant permet de produire une brève sortie à 1, pendant le front montant de l'horloge :



Il permet donc d'isoler le front montant de l'horloge. Il suffit alors d'insérer un tel détecteur de front entre une horloge une bascule sur niveau haut pour obtenir une bascule déclenchée par le front montant ou le front descendant du signal d'horloge.



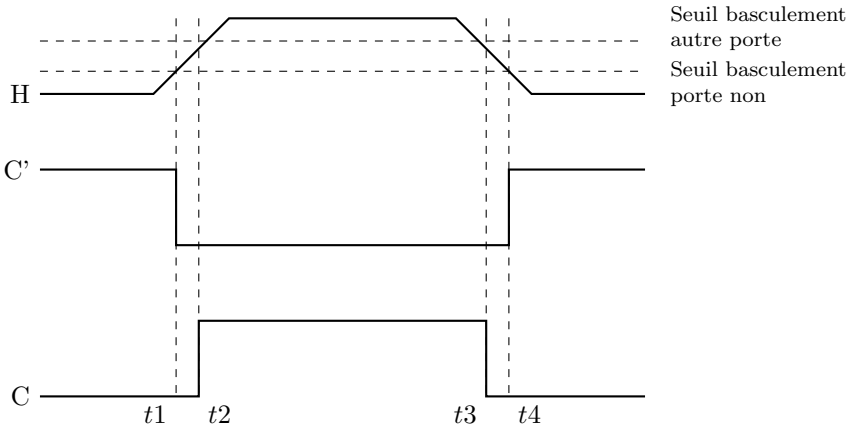
6.2.2.2.2 Montage maître-esclave Ce montage permet d'obtenir un circuit plus stable et insensible aux parasites.



L'entrée de la bascule "Esclave" est reliée à la sortie Q' de la bascule "Maître".

Toute modification sur la sortie du Maître se répercute sur l'Esclave : le Maître asservie l'Esclave.

Tout le fonctionnement de cette bascule se joue sur le fait que que les entrées des horloges des bascules maître et esclave sont inversées ($\bar{C} = C'$), et sur le temps de basculement d'une porte NON :



On voit que C et C' ne sont jamais en même temps à 1, ce qui fait que les bascules Maître et Esclave ne sont jamais acives en même temps. Ceci permet que la modification de l'entrée D n'ai aucune incidence sur la sortie Q lors d'une phase stable de l'horloge (0 ou 1).

Etude du fonctionnement : Examinons le comportement de la bascule :

temps	avant $t1$	entre $t1$ et $t2$	juste après $t2$	entre $t2$ et $t3$	entre $t3$ et $t4$	après $t4$
Maître	actif	inactif	inactif	inactif	inactif	actif
Esclave	inactif	inactif	actif	actif	actif	inactif
D	D	D	D	D	D	D"
Q'	D	D	D	D	D	D"
D'	D'	D'	D	D	D	D
Q	Q	Q	D	D	D	D

Avant $t1$ **:** $C' = 1$ et $C = 0$, donc le maître est actif et l'esclave est inactif :

la donnée D est transférée par le Maître sur sa sortie Q'. L'esclave est inactif donc la sortie Q reste inchangée

Entre $t1$ **et** $t2$ $C' = C = 0$, les deux bascules sont donc verrouillées.

L'information presente sur l'entrée D juste avant $t1$ est présente en Q' et mémorisée par la bascule (puisque C' est passé de 1 à 0). La sortie Q reste donc inchangée.

Juste après $t2$: $C' = 0$ et $C = 1$:

le Maître est inactif et l'Esclave devient actif. L'information sur l'entrée D' (donc celle de Q') passe sur la sortie Q.

Le transfert de la donnée en D vers la sortie Q a bien été effectué sur le front de l'horloge (en $t2$)

De l'instant $t2$ **à** $t3$ **:** Rien ne change pour Q puisque le maitre est inactif et D' n'est donc jamais modifié.

Entre $t3$ **et** $t4$ **:** $C' = C = 0$, l'Esclave devient inactif et le Maître le reste. Les sorties Q et Q' sont inchangées

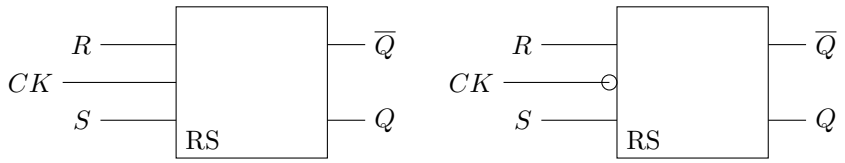
Après l'instant $t4$ **:** $C' = 1$ et $C = 0$:

le Maître devient actif, l'Esclave reste inactif. L'entrée D est recopiée sur la sortie Q' mais la sortie Q reste identique.

6.3 Les différents types de bascules et leur représentation symbolique

6.3.1 Bascule RS synchrone

Voici la représentation symbolique des bascule RS déclenchées sur niveau d'horloge

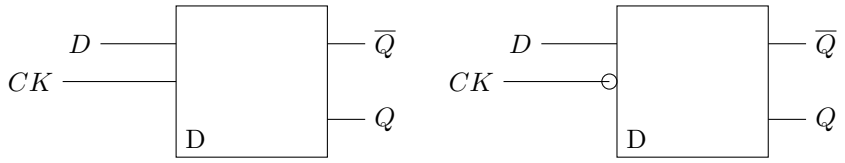


La bascule de gauche se déclenche sur le **niveau haut** de l'horloge tandis que celle de droite se déclenche sur le **niveau bas**.

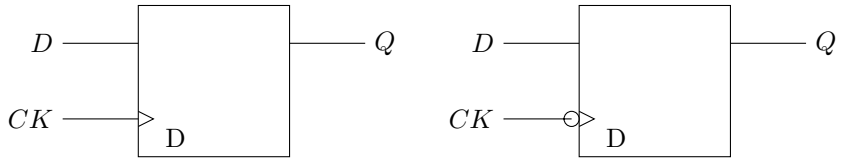
6.3.2 Bascule D

L'équation de la bascule D synchronisée sur niveau haut est la suivante : $Q' = Q \cdot \bar{CK} + CK \cdot D$

Voici la représentation symbolique de la bascule D à déclenchement sur niveau haut et niveau bas :



Voici les représentation des Bascules D à déclenchement sur front d'horloge. La bascule de gauche se déclenche sur **front montant** (FM) et celle de droite sur **front descendant**. (FD).



Pour la bascule à déclenchement sur front montant :

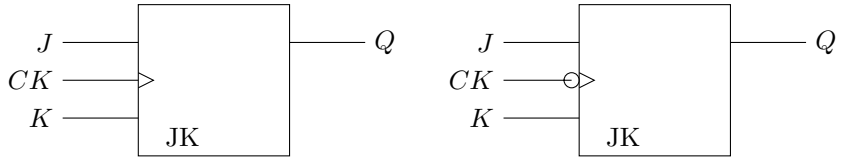
CK	D	Q'
0	_	Q
1	_	Q
FM	0	0
FM	1	1

En fait, pour les bascules à déclenchement sur front d'horloge, on ne met jamais l'horloge dans la table de vérité, on donne directement la table indiquant le comportement du circuit pendant le front déclencheur. La table correcte est donc la suivante

D	Q'
0	0
1	1

6.3.3 Bascule JK

Voici la représentation symbolique de la bascule JK à déclenchement sur niveau d'horloge :



La bascule de gauche se déclenche sur front montant et celle de droite sur front descendant.

L'équation de la bascule est la suivante : $Q' = J\bar{Q} + \bar{K}Q$.

Voici la table de vérité :

J	K	Q'
0	0	Q
0	1	0
1	0	1
1	1	$\overline{Q}$

6.3.4 La bascule T

La bascule T (pour Toggle) change d'état à chaque front montant (ou descendant selon sa conception). Ainsi, lorsque l'entrée T est à 1, cela a pour effet de d'inverser la valeur de Q (Q' est donc l'inverse de Q). Pour toute les autres configurations Q gardera la même configuration.

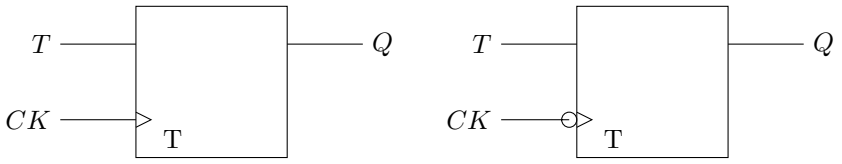
Table de vérité :

T	Q'
0	Q
1	$\bar{Q}$

On peut remarquer que, lorsque T=1, la sortie Q est de période 2 fois plus longue que celle de H, en effet, il faut 2 fronts montant de H pour obtenir celui que Q.

C'est pour cela que l'on lui donne aussi le nom de diviseur de fréquence par 2

Voici les représentation des Bascules T à déclenchement sur front d'horloge. La bascule de gauche se déclenche sur front montant et celle de droite sur front descendant.



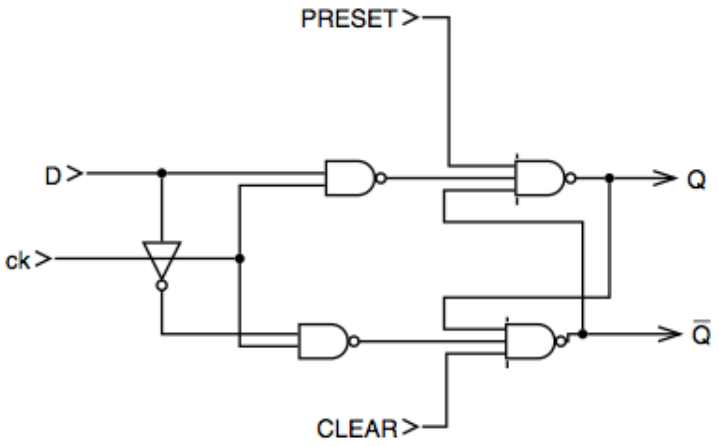
6.4 Forçage des bascules

Sur les bascules (synchrones), il existe généralement une ou deux entrées supplémentaires `PRESET`, `CLEAR` qui indépendamment de l'horloge

— `PRESET` : force la sortie à 1
— `CLEAR` : force la sortie à 0.

Ceci est utilisé notamment pour l'initialisation de la bascule lors de sa mise sous tension (garantie d'un état initialement stable).

Elles agissent sur l'esclave des bascules. Par exemple, pour construire une bascule où les signaux `PRESET` et `CLEAR` sont actifs sur niveau bas, il suffit de compléter la bascule esclave de la façon suivante :

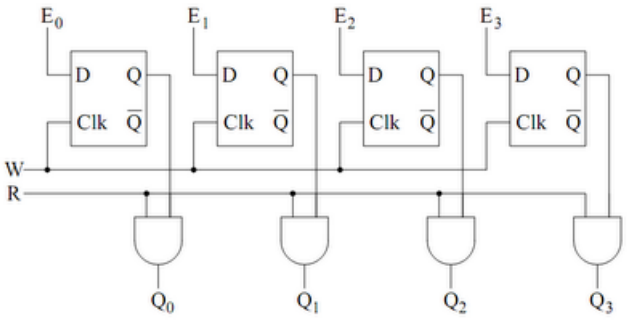


6.5 Les registres

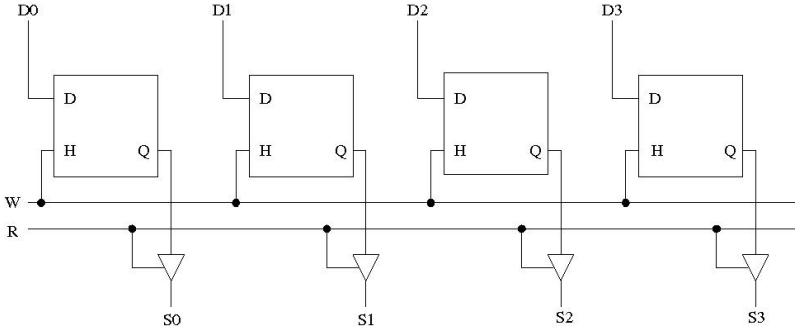
Puisque les bascules permettent la mémorisation de bits, elles sont le composant principal des registres.

6.5.1 Registre élémentaire

Voici par exemple un registre 4 bits. L'entrée W ordonne l'écriture des entrées sur les registres, l'entrée R ordonne la lecture des valeurs mémorisées.



En synchronisme avec le signal d'écriture W le registre mémorise les données présentent sur les entrées E0, E1, E2 et E3. Elles sont conservées jusqu'au prochain signal de commande W. Dans cet exemple les états mémorisés peuvent être lus sur les sorties Q0, Q1, Q2 et Q3 en coïncidence avec un signal de validation R. Lorsque ces sorties sont connectées à un bus, les portes ET en coïncidence avec ce signal de lecture sont remplacées par des portes à trois états comme sur le schéma ci-dessous.



6.5.2 Registre à décalage

Le registre à décalage permet de mémoriser un mot binaire donné en entrée, ou bien de décaler le mot stocké en mémoire. Un registre à décalage à droite peut-être utilisé comme un diviseur pas 2, alors qu'un registre à décalage à gauche agit comme un multiplieur par 2.

Chapitre 7

Les mémoires

7.1 Généralités

Une **mémoire** est un dispositif permettant d'enregistrer, de conserver et de restituer de l'information.

Voici les unités utilisées pour désigner des quantités de mémoire. On parle en fait de **quantité adressable** dans le sens où généralement on ne peut accéder à l'information bit par bit : on lit ou on écrit en mémoire un groupe de bits. Historiquement, ce groupe est de taille 8 et on parle d'un octet. Maintenant, on parle de mot mémoire et ces mots sont de taille 16, 32 ou 64 bits.

La taille de ces mots est à distinguer de celle des mots traités par le processeur (16 bits, 32 bits, 64 bits). La taille de ces deux types de mots n'étant pas nécessairement (ou théoriquement) liée.

- unité de base : 1 **bit** (0 ou 1)
- **octet** (byte) = groupe de 8 bits
- **mot** = regroupement d'octets (8 bits, 16 bits, 32 bits, ...)
  $\Rightarrow$ unité d'information adressable en mémoire.
- un **KiloOctet** = 2^{10} octets = 1024 octets = 1 Ko
- un **MegaOctet** = 2^{10} Ko = 1 Mo
- un **GigaOctet** = 2^{10} Mo = 1 Go
- un **TeraOctet** = 2^{10} Go = 1 To

La notion de **mémoire** pour un ordinateur regroupe différents matériels et ne correspond pas qu'à un seul dispositif permettant de stocker de l'information.

Ces dispositifs peuvent être de différentes natures, de différentes technologies, ceci leur donnant des caractéristiques différentes.

- différentes technologies :
  Electronique - Magnétique - Optique
- différentes caractéristiques :
  - capacité
  - temps d'accès
  - débit
  - volatilité

7.1.1 Performances d'une mémoire

Voici quelques unes des grandeurs caractérisant les mémoires. Le temps d'accès est ce qui mesure la latence entre l'ordre donné à la mémoire (écriture/lecture) et l'exécution effective de cette ordre.

- **temps d'accès :** temps qui sépare une demande de lecture/écriture et sa réalisation t_a
- **temps de cycle :** temps minimum entre deux accès à la mémoire t_c
  On a $t_a < t_c$ (stabilisation des signaux, synchronisation, ...)
- **débit** (ou **bande passante**) : nombre de bits maximum transmis par seconde

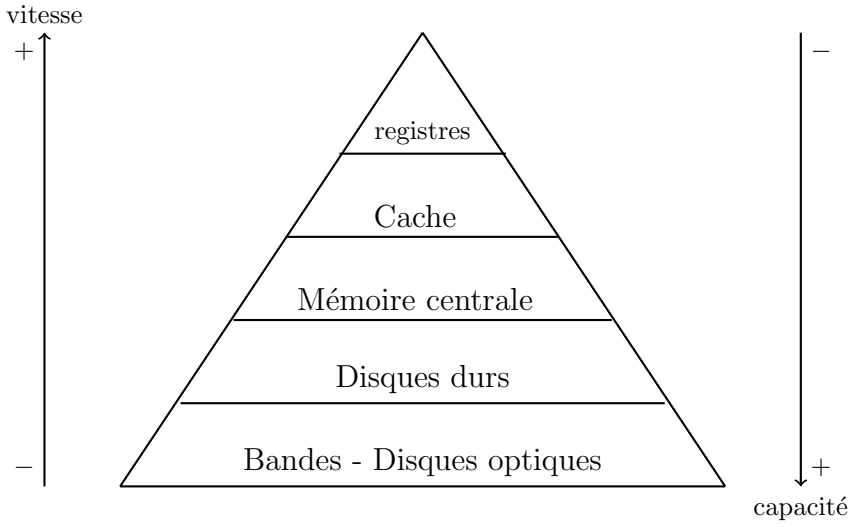
en cas d'accès en temps uniforme au données

$$B = \frac{n}{t_c} \qquad n \text{ est le nombre de bits transférés par cycle}$$

Le temps de cycle est le temps minimum séparant la réalisation de deux opérations succéssives.

7.1.2 Types de mémoires

Pour des raisons technologiques (l'augmentation de la taille d'une mémoire s'accompagne toujours de l'augmentation du temps d'accès) et des raisons économiques (plus vite = plus cher), on utilise différents types de mémoires. La vitesse d'une mémoire (temps d'accès et débit) est inversement proportionnel à sa taille "usuelle". On a le schéma suivant :

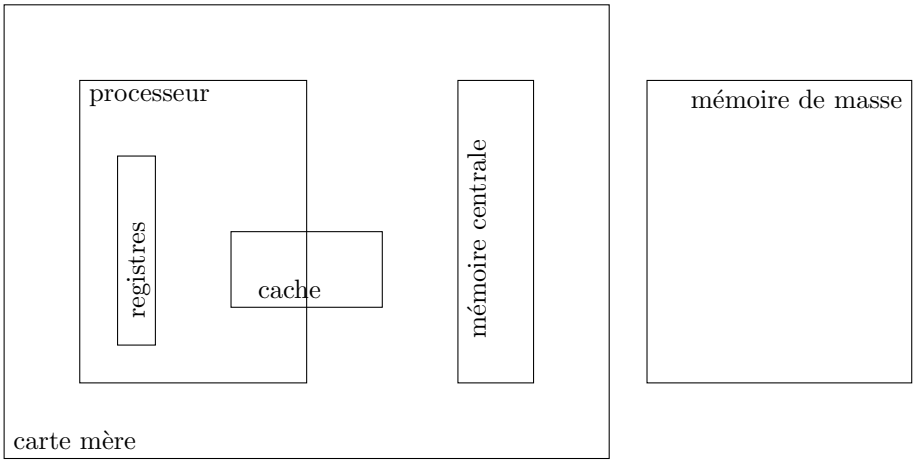


	vitesse (temps d'accès)	vitesse (débit)	capacité
registres	< 1 ns	> 50 Go/s	< 100 octets
cache	2 - 5 ns	5 - 20 Go/s	100 Ko - 1 Mo
mémoire centrale	20 ns	1 Go/s	256 Mo - 4 Go
disque dur	1-10 ms	300 Mo/s	50 Go - 500 Go

(Ce tableau date que 4-5 ans maintenant, il faudrait l'actualiser)

7.1.3 Localisations

Voici la disposition des différentes mémoires dans un ordinateur. Les registes, qui constituent la "mémoire de travail" du processeur, se trouvent en son sein. La mémoire cache (copie rapide de la mémoire centrale) est souvent décomposée en plusieurs parties, l'une collée sur le processeur et l'autre toute proche mais sur la carte mère. Toujours sur la carte mère, la mémoire centrale (appelée mémoire vive ou RAM) qui stocke données et programmes. Finalement, hors de la carte mère, la mémoire de masse (appelée mémoire morte ou ROM) stockant les informations, généralement sous forme de fichiers.



7.1.4 Méthodes d'accès

La méthode d'accès décrit comment accéder à une information en connaissant sa position.

L'accès direct est similaire à l'accès à une case d'un tableau, on accède directement à n'importe quelle case directement par son indice.

L'accès séquentiel est similaire à l'accès d'une information dans une liste chaînée, il faut parcourir toutes les cellules précédant l'information d'intérêt.

- Accès séquentiel
  - pour accéder à une information, il faut parcourir toutes les informations qui la précède
  - exemple : bandes magnétiques
- Accès direct
  - chaque information possède une adresse propre, à laquelle on peut accéder directement
  - exemple : mémoire centrale de l'ordinateur
- Accès semi-séquentiel
  - intermédiaire entre séquentiel et direct
  - exemple : disque dur
    - accès direct au cylindre
    - accès séquentiel au secteur sur un cylindre
- Accès associatif
  - une information est identifiée par sa clé
  - on accède à une information via sa clé
  - exemple : mémoire cache

7.2 Types de mémoire

Sur la carte mère cohabitent

- Mémoires persistantes (mémoires mortes - ROM)
  - leur contenu est fixé (ou presque)
  - et conservé en permanence même hors alimentation électrique
- Mémoires volatiles (mémoires vives - RAM)
  - leur contenu est modifiable
  - et perdu hors alimentation électrique

ROM = Read Only Memory RAM = Random Access Memory

7.2.1 Mémoires mortes (ROM)

Les différents types de ROM

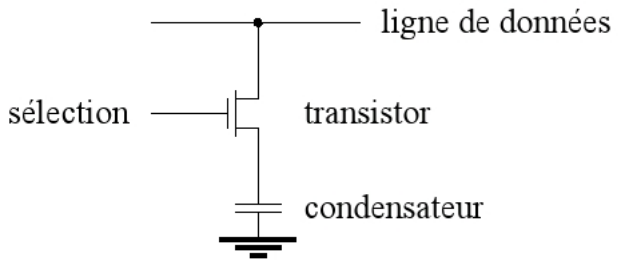
- **ROM** (**R**ead **O**nly **M**emory) : information stockée au moment de la conception du circuit.

— **PROM** (**P**rogrammable **R**ead **O**nly **M**emory) : mémoire programmble par un utilisateur final mais une seule fois
— **EPROM** (**E**rasable **P**rogrammable **R**ead **O**nly **M**emory) : mémoire (re)programmable et effaçable par ultraviolet.
— **EEPROM** (**E**lectrically **E**rasable **P**rogrammable **R**ead **O**nly **M**emory) : mémoire (re)programmable et effaçable électriquement.
— exemple : Bios (“flashable”) - lecteur MP3

7.2.2 Mémoires volatiles (RAM)

On a deux grands types de RAM

DRAM : Dynamic RAM Mémoire électronique à réalisation très simple : 1 bit = 1 transistor + 1 condensateur, le condensateur stocke l'information. Le problème est que les condensateurs ont le défaut de se décharger (perdre lentement sa charge) et ils doivent être rechargés fréquemment (rafraichissement). Durant ces temps de rechargement, la mémoire ne peut être ni lue, ni écrite, ralentissant donc son fonctionnement (d'où le terme de Dynamique). C'est une mémoire volatile car sans alimentation, les données sont perdues. Peu coûteuse elle est principalement utilisée pour la mémoire centrale de l'ordinateur.



SRAM : Static RAM Les SRAM n'ont pas besoin de rafraichissement car un bit est stocké par une bascule : 1 bit = 4 transistors = 2 portes NOR. Elle est donc également volatile. Elle est plus coûteuse qu'une DRAM et est notamment utilsée pour les mémoires caches du processeur

Ceci correspond à la notion de bascule utilisée pour stocker une information.
— Statique : l'information n'a pas besoin d'être rafraîchie
— réalisation :
—
— Bascule RS (ou D) qui stocke l'information
— beaucoup plus **rapide** que la DRAM
— beaucoup plus **cher** que la DRAM

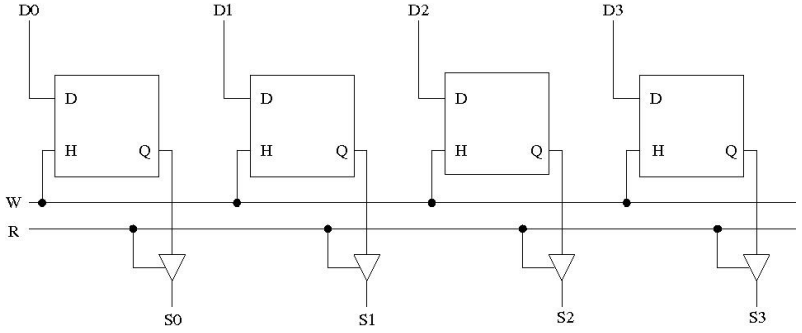
7.3 Registres

Le processeur traite des informations ; les registres sont utilisés pour stoker l'information qui va être traitées ou qui vient d'être traitées. Ils stockent les informations relatives à une instruction : les opérandes nécessaires à l'instruction, les résultats produits par l'instruction.

Les registres sont au coeur du processeur, on peut donc en mettre peu (¡ 20). Ils doivent être très rapide (cadencés à la vitesse du processeur). Ce sont en fait les mémoires les plus rapides et les plus chères.

Réalisation :
— registre 1-bit = 1 bascule RS (ou D)
— registre n-bits = n bascules RS (ou D) en parallèle

Registre 4 bits :
On appelle le registre 4 bits un registre **parallèle-parallèle** car on écrit et on lit en parallèle 4 bits (4 entrées et 4 sorties).

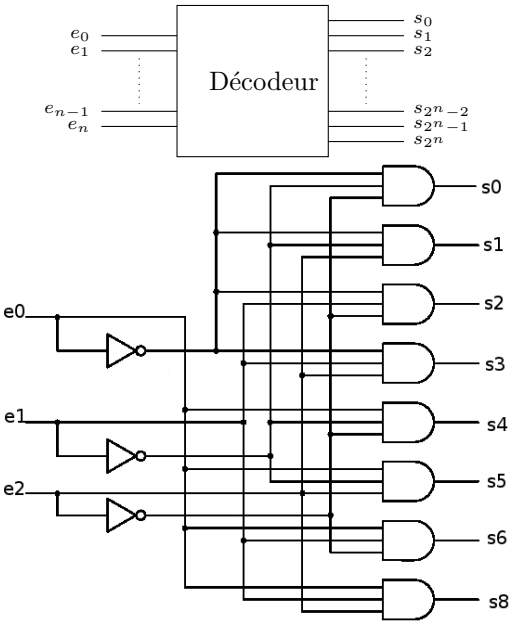


7.4 Bancs de registres

7.4.1 Décodeurs - multiplexeurs : rappels

On rappelle d'un décodeur sert à décoder un nombre : il active la ligne i si i est le nombre binaire codé sur ses entrées.

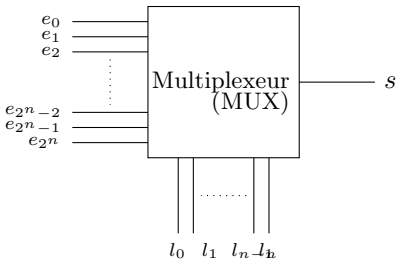
— Un **décodeur** permet de **décoder un mot binaire** : il comprend n entrées et 2^n sorties.
— la $i$ème sortie de décodeur vaut 1 si les n entrées forment l'entier binaire i.

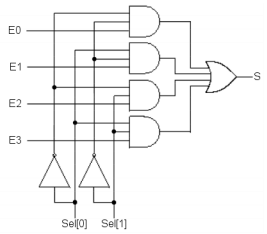


décodeur 3 vers 8

Le multiplexeur est une sorte d'aiguillage. Une des entrées est aiguillée vers la sortie selon la valeur des lignes de sélection.

— Un **multiplexeur** comporte 2^n entrées, 1 sortie et n lignes de sélection (entrées).
— la sortie du multiplexeur vaut la valeur de la $i$ème entrée si l'entier i est codé sur les lignes de sélection.

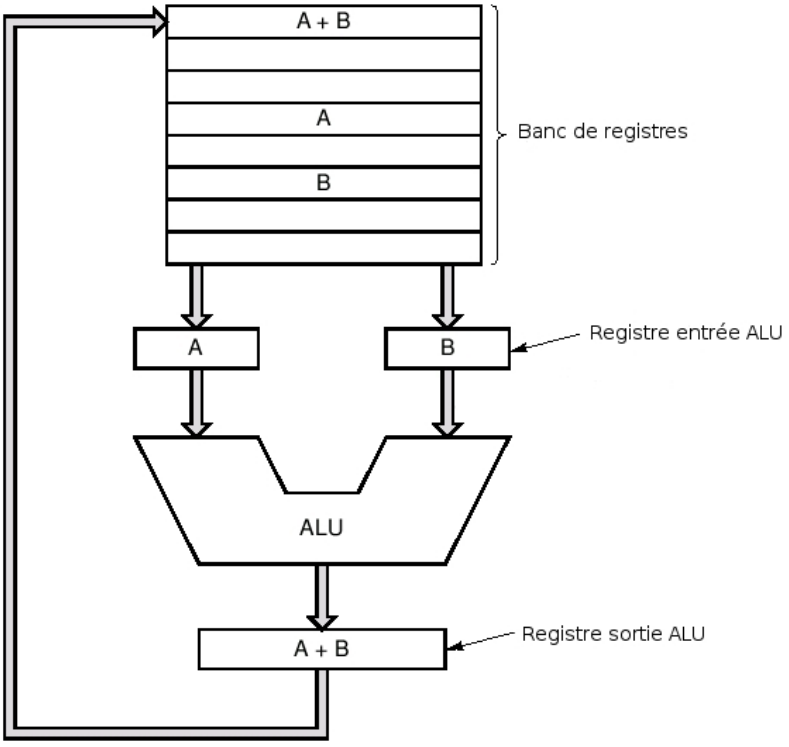




multiplexeur 4 vers 1

7.4.2 Bancs de registres

Un **banc de registres** de hauteur n et de largeur k est
— un ensemble de n registres de k bits
— une mémoire de n adresses
Exemple :

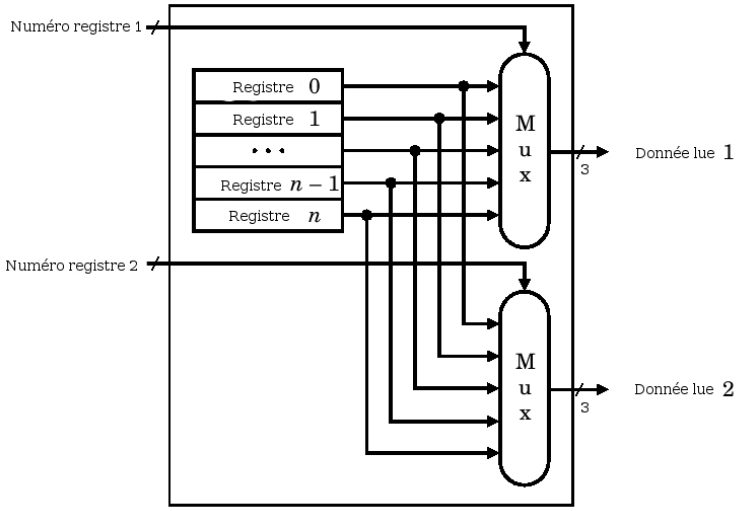


Un banc de n registres de 3 bits avec
— un port d'écriture
— deux ports de lecture (lors d'une lecture le contenu de deux registres est lu même temps.)

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.

Fonctionnement en lecture : (2 registres en parallèle)

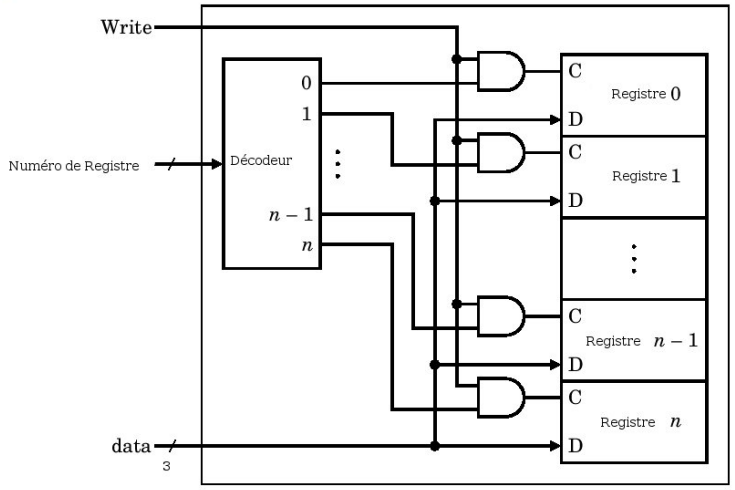


Nous avons vu tout à l'heure une façon explicite de sélectionner les sorties d'une ligne de bascule D dans un banc mémoire ; on fait de même ici mais en utilisant un circuit "haut niveau", un multiplexeur. Selon la valeur de l'entrée *Numero registre* 1, le multiplexeur sélectionne une de ses entrées 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, *Numero 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).

Fontionnement en écriture :

Voici une réalisation d'un banc de registres de n mots de 3 bits avec un port d'écriture. Lorsque l'entrée `Write` est activée, le mot de l'entrée data en mémorisé sur le registre `numéro de registre` qui est identifié par le décodeur.



Exemple complet :

Voici une réalisation d'un banc de registres de 4 mots de 3 bits avec un port de lecture et un port d'écriture. $I_0I_1I_2$ sont les entrées, on y place donc les données qu'on veut écrire dans le banc de registres. $D_0D_1D_2$ sont les sorties, on y trouve donc les données lu dans le banc de registres.

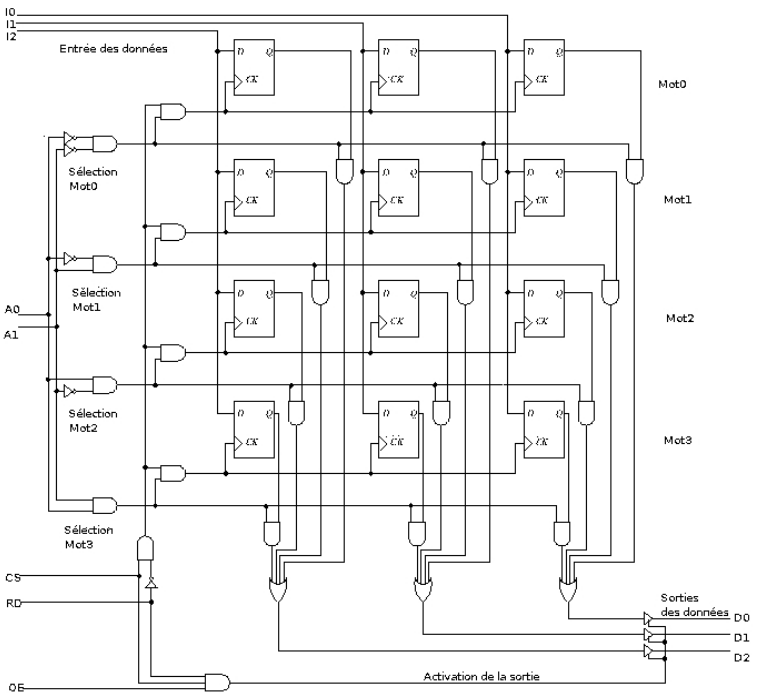
$I_0I_1I_2$ sont les entrées, on y place donc les données qu'on veut écrire dans le banc de registres. $D_0D_1D_2$ so

Pour sélectionner le registre du banc concerné par la lecture ou par l'écriture, on utilise les entrées A_0A_1 pour selectionner celui-ci. Il est simple de voir que parmi les portes portant la mention "Sélection Mot X", une et une seule verra sa sortie mise à 1. Ainsi, ces sorties étant reliées chacune

à une ligne de bascules D, une seule ligne de 3 bascules D sera active (1 sur leur entrée CK) permettant une écriture si CS = 1 et RD = 0.

L'entrée CS est un interrupteur du banc de registres qui ne fonctionne que si celui-ci vaut 1 tandis que RD est la commande RD = 0 on réalise une écriture et RD = 1 une lecture dans le banc.

Si CS = 1 et RD = 0 alors les entrées $I_0I_1I_2$ sont mémorisées sur une ligne de bascules D, celle sélectionnée selon les valeurs de A_0A_1.



7.5 Mémoire centrale

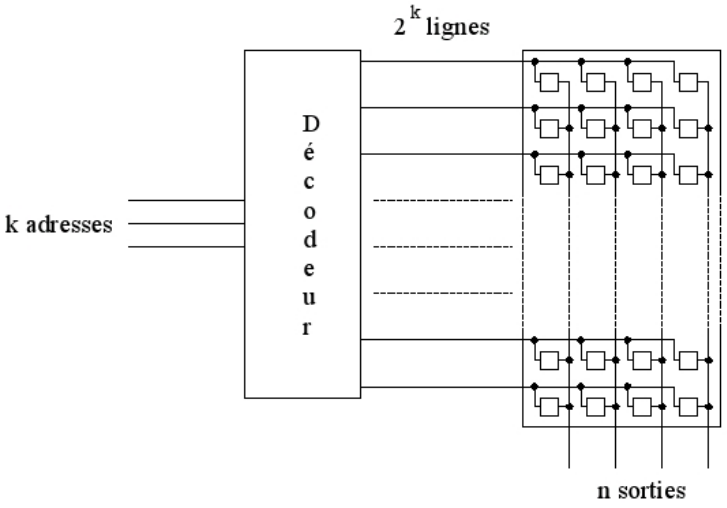
Rappels :

— Mémoire de type DRAM
— l'information y est stockée comme des mots (mémoire) d'un certain nombre de bits (8, 16, 32, 64 bits) : la longueur des mots est le **format** de la mémoire.
— vitesse relativement lente (comparée à celle du processeur)

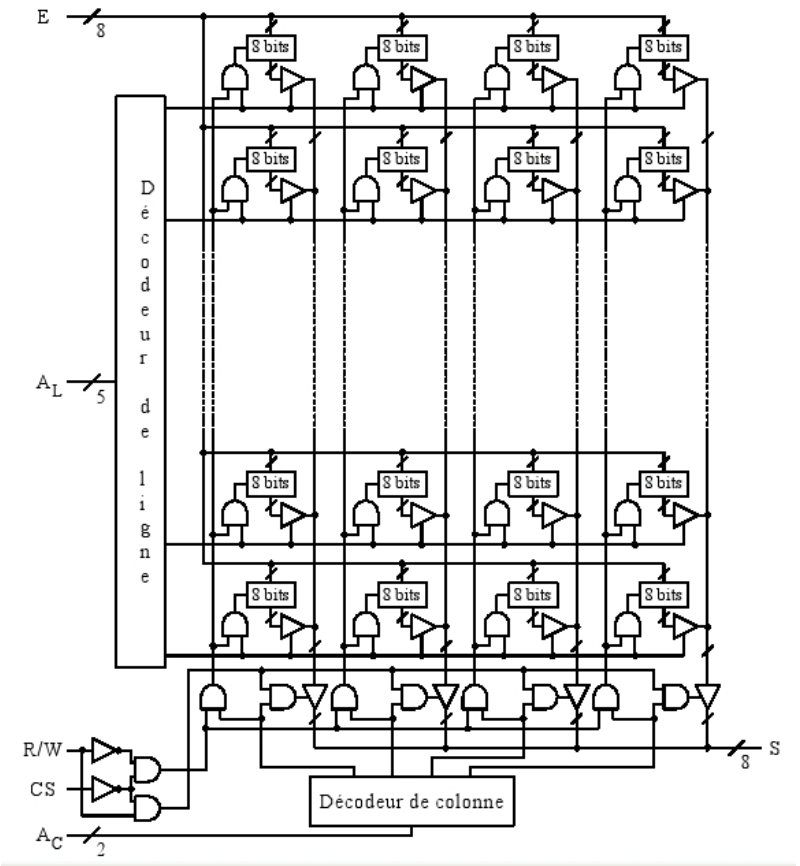
7.5.1 Organisation de la Mémoire centrale

Voici une première organisation possible de la mémoire utilisant le même principe que le banc mémoire vu précédemment. Malheureusement, alors que le banc mémoire ne contient de quelques dizaines de registres tout au plus, l'adressage de la mémoire centrale est de plusieurs milliards de mots. Le décodeur nécessaire n'est alors simplement plus réalisable.

Mémoire unidimensionnelle



Le nombre de portes dans le décodeur est trop important.
Mémoire bidimensionnelle : utilisation d'un décodeur pour les lignes et un pour les colonnes.



Les mots mémoires (ici, des mots de 8 bits) sont organisées selon un tableau bi-dimensionnel : entrées E et sorties S sont reliées à tous les mots mémoires. Un seul mot est actif pour l'écriture ($R/W = 0$ et $CS = 1$) car un seul couple ligne-colonne est à 1. Ceci est assuré par les décodeurs de ligne et de colonne d'entrées respectivement A_L et A_C.

En ce qui concerne la sortie, on remarque que la sortie de chaque mot mémoire d'une même colonne est réliée à un même bus via une porte trois-états (Ceci ne serait pas possible sans une porte trois-états). Un porte trois-états est une porte qui en plus des états 0 et 1 peut prendre une

troisième valeur appelée “haute impédence”. Une seule porte ne sera pas en haute impédance, les autres seront donc déconnectées, comme si la sortie de leur mot mémoire n'était pas relié au bus.

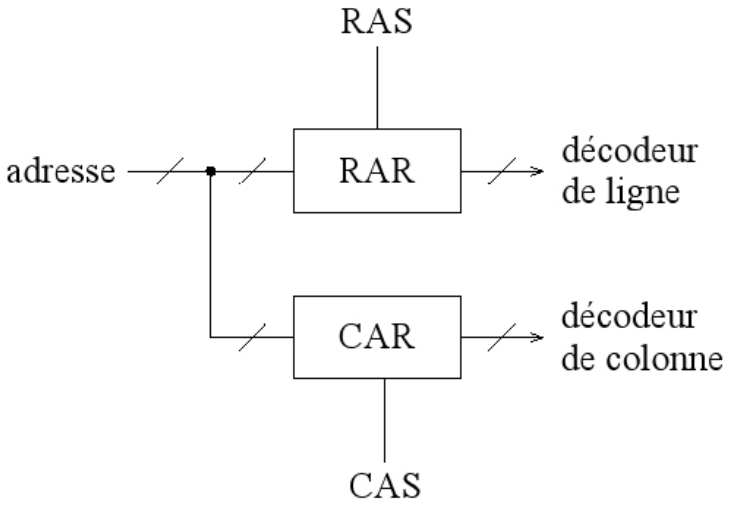
Pour la lecture ($R/W = 1$ et $CS = 1$), le principe est alors le même que pour les sorties des mots mémoire par colonne, le décodeur de ligne n'activant qu'une seule porte trois-états par colonne.

Mémoire matricielle (organisée comme une matrice carrée)

Ceci n'est qu'une variante mineure de ce que nous avons vu précédemment sur l'organisation bidimensionnelle. Une seule ligne est utilisée pour alimenter deux registres qui vont stockés durant l'opération sur la mémoire respectivement le numéro de ligne et de colonne. Le chargement de la valeur d'adresse dans ces registres se fait sur les signaux RAS et CAS

Utilisation des mêmes lignes d'adresses utilisées pour les lignes et les colonnes.

Chargement de l'adresse en deux temps et stockée dans deux registres

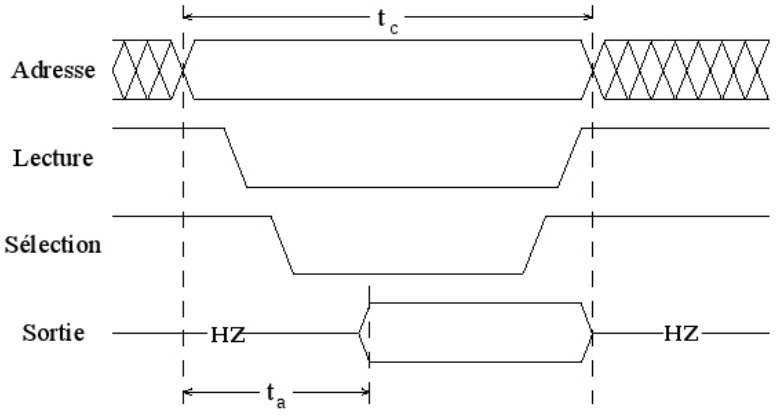


— **CAS** : **C**olumn **A**ddress **S**trobe
— **RAS** : **R**ow **A**ddress **S**trobe

7.5.2 Fonctionnement de la mémoire centrale

Cycle de lecture

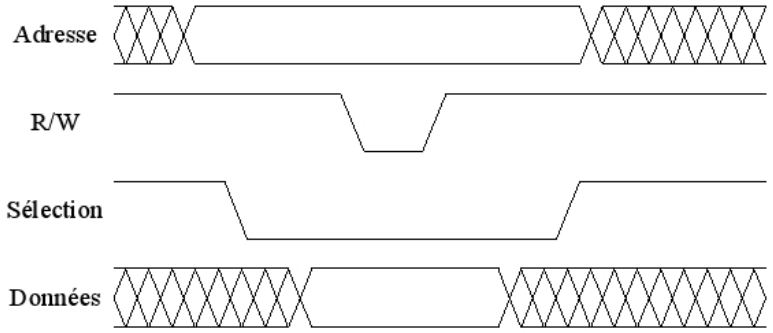
— établissement de l'adresse
— signal de lecture (R/W=0 par exemple)
— sélection du boitier (CS=0)
— Après un certain temps, l'information apparaît sur la sortie et reste présente jusqu'à la fin du cycle.



On voit qu'entre deux établissements de l'adresse, on trouve t_c, le temps de cycle. Entre l'établissement de l'adresse et l'apparition de la donnée lue sur la sortie, on a t_a, le temps d'accès. Le reste du temps, la sortie est en haute impédance

Cycle d'écriture

— établissement de l'adresse
— sélection du boitier (CS=0)
— établissement de la donnée sur l'entrée
— signal d'écriture (R/W=0 ci-dessous)



Protocoles échanges processeur-mémoire

— **Synchrone :** au bout de k unités de temps, le processeur suppose que l'opération sur la mémoire a été réalisée (mot écrit en mémoire, mot lu disponible sur la sortie)
— **Asynchrone (handshaking) :** processeur et mémoire s'échangent des informations de contrôle (**request**/ **acknowledgment**)

Optimisations La mémoire synchrone est synchronisée avec le **bus** qui achemine les informations entre la mémoire centrale et le processeur. Le temps de cycle de la mémoire et celui du bus se superposent.

Le **mode page** permet de charger une page de mémoire, chaque numéro de ligne étant un début de page. Ceci est proche de la notion de segmentation de la mémoire qu'on verra plus loin (mais attention ici, il s'agit de chargement de données et non de découpage de la mémoire).

Dans les modes **page** et **rafale**, on se base sur la localité des données : si on utilise une donnée, il y a une forte chance d'utiliser en suivant une donnée voisine (parcours de tableau).

— mémoire synchrone (synchronisée avec le bus) : **SDRAM**
— Pour les mémoires matriciels, **accès en mode page** : on charge ligne et colonne, puis on ne change que les colonnes pour les accès suivants (localité des données) : **DRAM FPM**
— Pour les mémoires matriciels, **accès en rafale (burst)** : on charge ligne et colonne ainsi que le nombre de données à lire ; incrémentation dans la mémoire des colonnes pour les accès suivants (localité des données). **DDR-SDRAM**

7.6 Assemblage de boitiers mémoire

Les mémoires sont regroupées dans des **boitiers mémoire**.

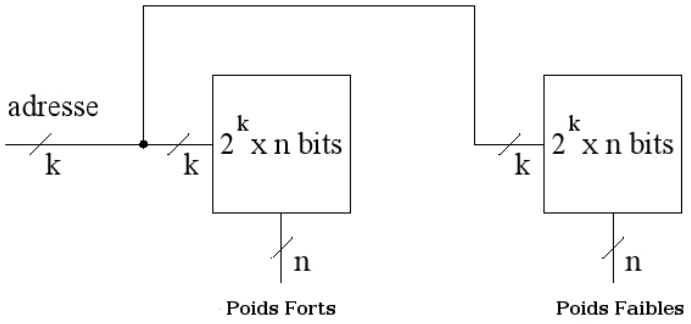
Dû aux limites technologiques d'intégration, pour obtenir des mémoires de grandes tailles, on associe plusieurs boitiers mémoires.

Ces blocs sont assemblés :

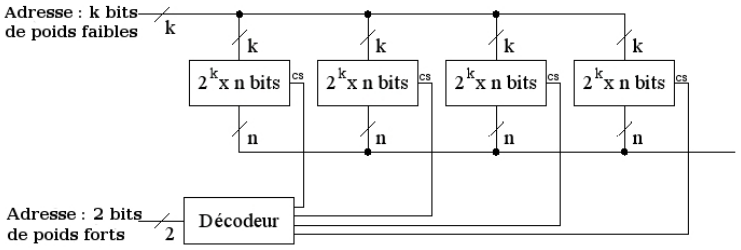
— pour augmenter la taille des mots de la mémoire
— pour augmenter le nombre de mots dans la mémoire

Augmentation de la taille des mots

2 boitiers de 2^k mots de n bits $\Longrightarrow$ un bloc de 2^k mots de $2n$ bits



On obtient donc des mots plus longs en mettant en parallèle plusieurs boitiers mémoire.
Augmentation du nombres de mots
4 boitiers mémoires de 2^k mots de n bits $\Longrightarrow$ un bloc de $4 * 2^k$ mots de n bits
adresse pour le nouveau bloc sur $k + 2$ bits



On obtient donc des mémoires de taille plus grande en mettant en série plusieurs boitiers mémoire.

En décodant les bits de poids forts de l'adresse, on obtient une ligne à 1 qui va sélectionner (avec $CS = 1$) un seul boitier.

7.7 Mémoire et erreurs

Du fait de sa nature "physique", les informations en mémoire peuvent comporter une ou des erreurs.

Pour détecter et corriger, on ajoute des **bits de contrôle**.

— **bit de parité :** 1 bit supplémentaire (en plus des bits de données) tel que le nombre de bits à 1 est pair
— **mémoire ECC** (Error Correction Coding) possède des bits supplémentaires pour détecter et corriger le(s) bit(s) erroné(s).

7.8 Mémoire Logique

Nous avons vu que la **mémoire physique** regroupait des mots de plusieurs bits en assemblant plusieurs bascules D, par exemple. Néanmoins, le programmeur peut se voir offrir une vue différente de la mémoire (comme si les assemblages étaient par exemple plus grand). On parle alors de **mémoire logique**. La mémoire logique est la façon dont le processeur (ou le programmeur) voit la mémoire (physique).

La mémoire est définie comme un ensemble de N octets consécutifs dont
— la première adresse est 0
— la dernière adresse est $N - 1$

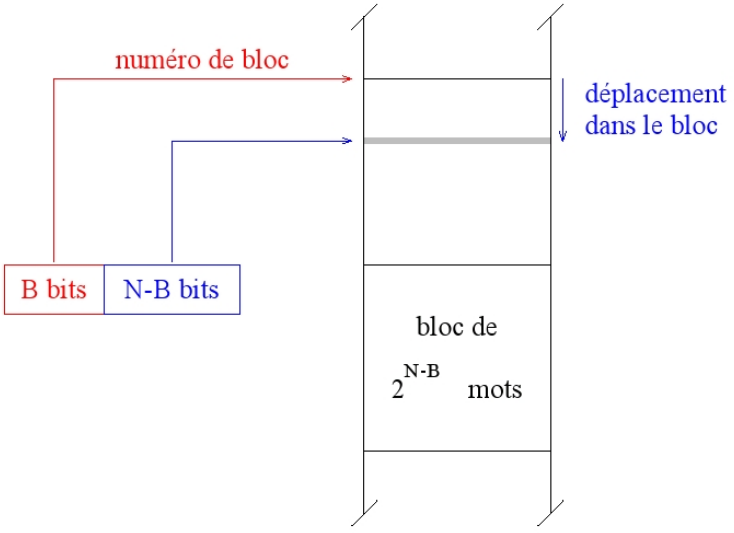
Adressage de la mémoire par des mots de : 8 (octet), 16, 32, 64, .. bits.
Un mot de 32 bits est constitué de 4 octets consécutifs.
Pour un mot mémoire de 32 bits, il existe 2 façons de ranger les octets qui le compose :
— Le mot de poids fort est stocké en premier : **big-endian**

i	i+1	i+2	i+3
Octet 3	Octet 2	Octet 1	Octet 0

— Le mot de poids fort est stocké en dernier : **little-endian**

i	i+1	i+2	i+3
Octet 0	Octet 1	Octet 2	Octet 3

Un mot mémoire ne peut commencer n'importe où
— les mots de 16 bits commencent sur des adresses paires
— les mots de 32 bits commencent sur des adresses multiples de 4
Si on considère le nombre (32 bits) FFA0 alors
— en big-endian, il est codé en mémoire comme FF AO
— en little-endian, il est codé en mémoire comme A0 FF
Segmentation de la mémoire : découpage logique de la mémoire en un certain certains nombres de blocs (ou segments)
Une adresse est codée comme
— un numéro de blocs
— un déplacement dans le bloc (**offset**)
N bits d'adresses $\Longrightarrow 2^N$ cases mémoire.
Les N bits sont séparés en deux :
— B bits (de poids fort) pour le numéro de bloc
— $N - B$ bits (de poids faible) pour le déplacement dans le bloc
On retrouve ces notions chez certains processeurs INTEL ; cette segmentation influe sur le modèle mémoire présent dans les langages assembleur ou C (taille des adresses de saut et des pointeurs). Les pointeurs courts et les adresses courtes sont alors simplement l'offset, le numéro de bloc étant commun à toutes ces adresses courtes.
On a donc 2^B blocs ayant chacun 2^{N-B} cases.



7.9 Mémoire Virtuelle

La mémoire physique n'est qu'un partie de la mémoire disponible : le système permet l'utilisation de la mémoire de masse (disque durs) comme de la **mémoire virtuelle**.
Pagination de la mémoire virtuelle

— Les adresses mémoires émises par le processeur sont des adresses virtuelles, indiquant la position d'un mot dans la mémoire virtuelle.
— Cette mémoire virtuelle est formée de zones de même taille, appelées pages. Une adresse virtuelle est donc un couple (numéro de page, déplacement dans la page). La taille des pages est une puissance de deux, de façon à déterminer sans calcul le déplacement (10 bits de poids faible de l'adresse virtuelle pour des pages de 1 024 mots), et le numéro de page (les autres bits).
— La mémoire vive est également composée de zones de même taille, appelées « cadres » (frames en anglais), dans lesquelles prennent place les pages.
— Un mécanisme de traduction (translation, ou génération d'adresse) assure la conversion des adresses virtuelles en adresses physiques, en consultant une table des pages » (page table en anglais) pour connaitre le numéro du cadre qui contient la page recherchée. L'adresse physique obtenue est le couple (numéro de cadre, déplacement).
— Il peut y avoir plus de pages que de cadres (c'est là tout l'intérêt) : les pages qui ne sont pas en mémoire sont stockées sur un autre support (disque), elles seront ramenées dans un cadre quand on en aura besoin.

Mécanisme de **swap** :

Il est parfois nécessaire de supprimer toutes les pages ou segments d'un processus de la mémoire centrale. Dans ce cas le processus sera dit swappé, et toutes les données lui appartenant seront stockées en mémoire de masse. Cela peut survenir pour des processus dormant depuis longtemps, alors que le système d'exploitation a besoin d'allouer de la mémoire aux processus actifs. Les pages ou segments de code (programme) ne seront jamais swappés, mais tout simplement réassignés, car on peut les retrouver dans le fichier correspondant au programme (le fichier de l'exécutable). Pour cette raison, le système d'exploitation interdit l'accès en écriture à un fichier exécutable en cours d'utilisation ; symétriquement, il est impossible de lancer l'exécution d'un fichier tant qu'il est tenu ouvert pour un accès en écriture par un autre processus.

Je n'irai pas plus loin ici puisque nous arrivons à la gestion mémoire qui est du domaine des **systèmes d'exploitation**.

Chapitre 8

Machines de Mealy - Machines de Moore

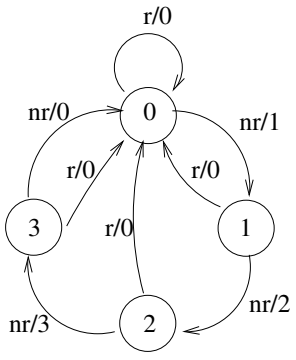
Il n'est pas toujours facile de réaliser un circuit séquentiel, nous présentons ici une méthode permettant de ramener la construction d'un tel circuit à l'élaboration d'un circuit combinatoire. La méthode repose sur l'abstraction du circuit par des machines proches des automates à états finis.

8.1 Introduction - Un exemple simple

On souhaite réaliser une compteur synchrone modulo 4 avec une entrée R de remise à 0.

On commence par modéliser le circuit par une machine à états. Chaque état modélise l'état du système à un instant donné, une transition correspond au changements du systèmes au prochain top d'horloge : elle donne le nouvel état du circuit, et sa sortie, pour une entrée donnée. Pour notre compteur, l'état du système est le contenu de sa mémoire : la valeur stockée soit un entier entre 0 et 3. Il y a deux entrées correspondant à l'envoie ou non du signal de remise à 0, notons les r et nr et il y a quatre sorties possibles 0, 1, 2, 3.

Le circuit peut donc être modélisé par le graphe de transitions suivant, ou de façon équivalente par la table suivante

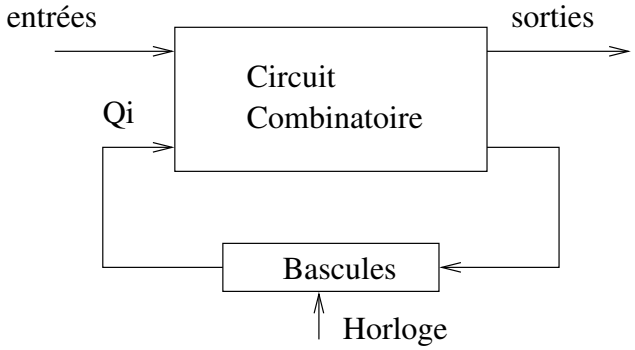


entrée	état courant	sortie	état suivant
nr	0	1	1
nr	1	2	2
nr	2	3	3
nr	3	0	0
r	0	0	0
r	1	0	0
r	2	0	0
r	3	0	0

Les arêtes du graphe de transition sont étiquetées par un couple e/s ou e est une valeur d'entrée et s une valeur de sortie. La signification d'une telle arête entre deux états q et q' est : si le système est dans l'état q, et que le système reçoit e en entrée, alors le système passe dans l'état q' et envoie en sortie la valeur s.

Ici, on peut remarquer que la sortie dépend de l'état courant et de l'entrée. Ce type de système de transition est appelé **Machine de Mealy**.

La construction du circuit séquentiel consiste maintenant en la concrétisation de notre abstraction en un circuit de la forme suivante :



Pour réaliser ce circuit, nous avons besoin de pouvoir mémoriser les états. Pour cela, nous allons utiliser des bascules. Il y a 4 états, nous pouvons coder ces quatre états sur 2 bits, donc nous allons utiliser 2 bascules en suivant le codage suivant (chaque Q_i est la sortie d'une bascule) :

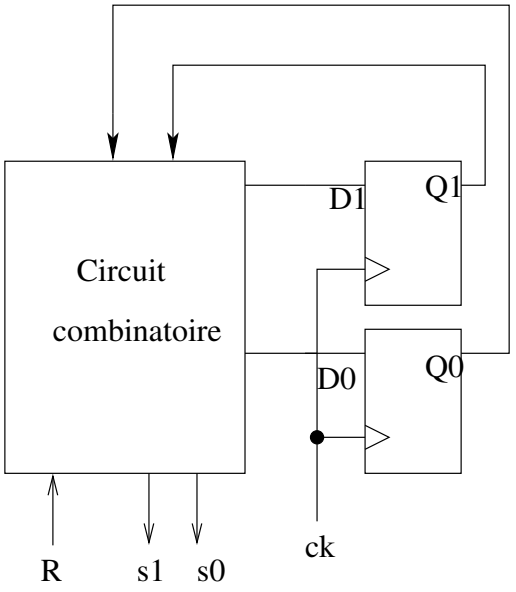
état	Q_1	Q_0
0	0	0
1	0	1
2	1	1
3	1	0

(Ce codage a été choisi arbitrairement, tout codage sur 2 bits aurait été aussi valable).

De même les deux entrées r et nr sont concrétisées par une entrée binaire R et les quatres sorties 0,1,2,3 par deux sorties binaires s_1 et s_0. En exprimant ces codages à l'intérieur de la table de l'abstraction, nous obtenons la table de vérité suivante :

entrée	état courant		sortie		état suivant	
R	Q_1	Q_0	s_1	s_0	Q'_1	Q'_0
0	0	0	0	1	0	1
0	0	1	1	0	1	1
0	1	1	1	1	1	0
0	1	0	0	0	0	0
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	1	0	0	0	0
1	1	0	0	0	0	0

Il faut maintenant en déduire un circuit. Nous choisissons pour le moment de mémoriser Q_1 et Q_0 à l'aide de bascules D, le circuit sera de la forme suivante :



.

Puisque pour une bascule D, la sortie Q correspond à l'entrée D, les sorties D_1 et D_0 du circuit combinatoire doivent êtres respectivement Q'_1 et Q'_0.

On utilise des tables de Karnaugh pour obtenir le circuit combinatoire :

Table de D_1 :

Q_1 \ R, Q_0	00	01	11	10
0	0	1	0	0
1	0	1	0	0

Table de D_0 :

Q_1 \ R, Q_0	00	01	11	10
0	1	1	0	0
1	0	0	0	0

Nous obtenons les équations suivantes :

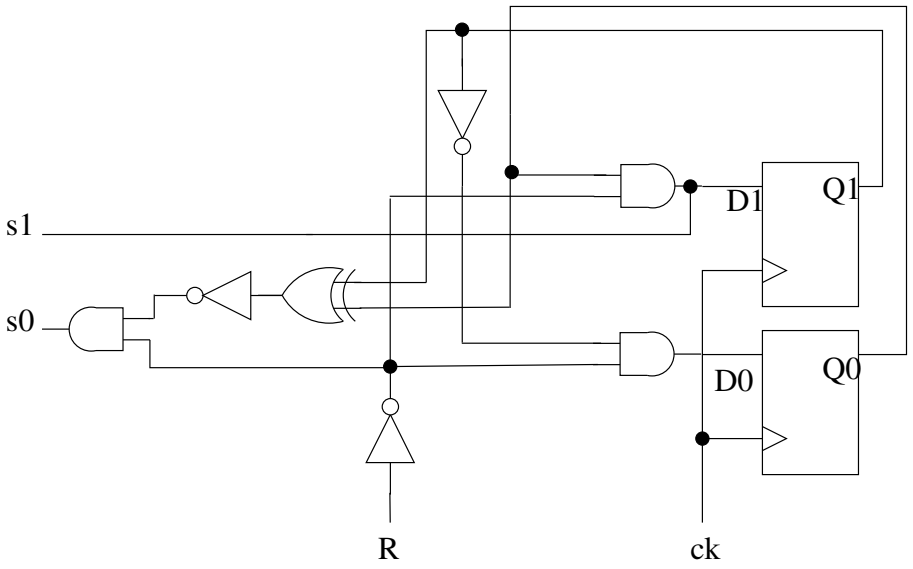
$D_1 = Q_0 \cdot \bar{R}$

$D_0 = \bar{Q_1} \cdot \bar{R}$

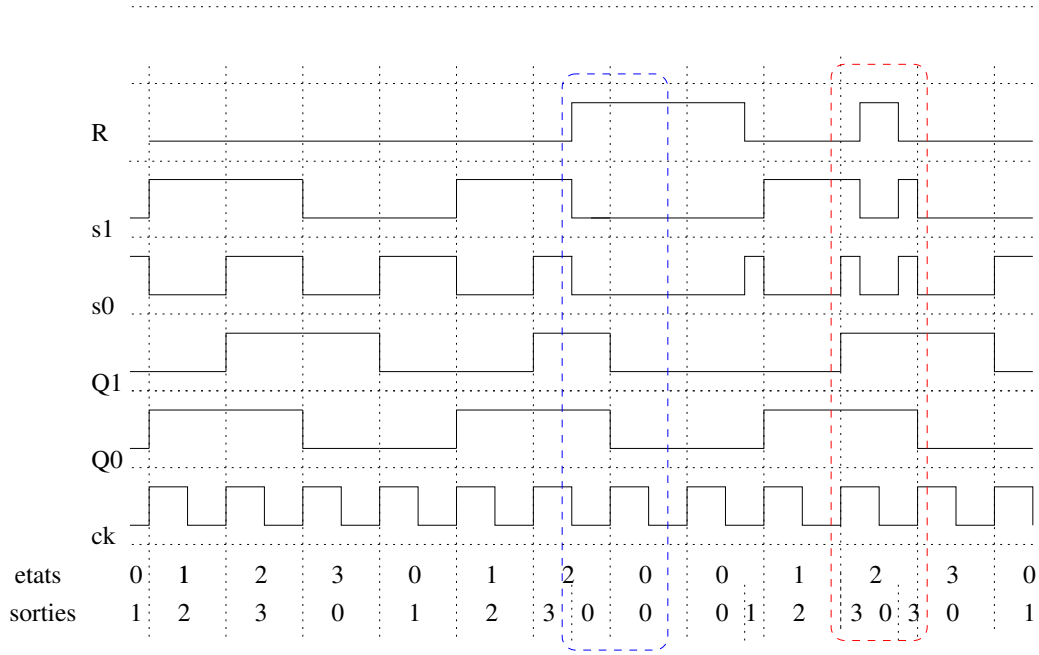
$s_1 = Q_0 \cdot \bar{R}$

$s_0 = \overline{Q_0 \oplus Q_1} \cdot \bar{R}$

Le circuit est donc le suivant :



Voici le chronogramme du circuit :

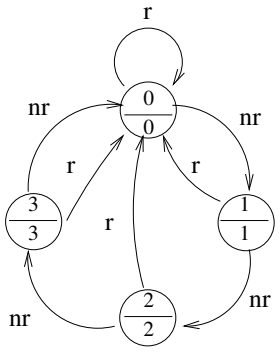


On remarque que :

1. les sorties ne durent pas forcément une période horloge (zone bleue),
2. si les signaux en entrée sont trop courts (zone rouge), il peuvent ne pas avoir d'effet sur la mémoire, et des sorties parasites apparaissent. Pour un fonctionnement normal, les changements de valeur des entrées doivent avoir lieu à cheval sur un front montant de l'horloge.

On aurait pu éviter ces problèmes en utilisant à la place d'une machine de Mealy, une **machine de Moore**. Une machine de Moore est un système de transition dont les sorties ne dépendent que de l'état. Dans ce cas, les sorties durent forcément un temps proportionnel à une période de l'horloge, et il n'y a plus de sorties parasites. Toutefois, ce circuit obtenu est moins réactif puisqu'il a un délai entre la variation d'une entrée et la variation de la sortie, de plus il comporte souvent plus d'états et il est donc plus coûteux.

Voici une machine de Moore abstrayant notre circuit :



entrée	état courant	sortie	état suivant
nr	0	0	1
nr	1	1	2
nr	2	2	3
nr	3	3	0
r	0	0	0
r	1	1	0
r	2	2	0
r	3	3	0

Puisque les sorties ne dépendent plus que des états, elles sont maintenant représentées dans les états de la machine

Après codage des états, entrées et sorties, nous obtenons la table de vérité suivante :

entrée	état courant		sortie		état suivant	
R	Q_1	Q_0	s_1	s_0	Q'_1	Q'_0
0	0	0	0	0	0	1
0	0	1	0	1	1	1
0	1	1	1	0	1	0
0	1	0	1	1	0	0
1	0	0	0	0	0	0
1	0	1	0	1	0	0
1	1	1	1	0	0	0
1	1	0	1	1	0	0

Les équations sont donc :

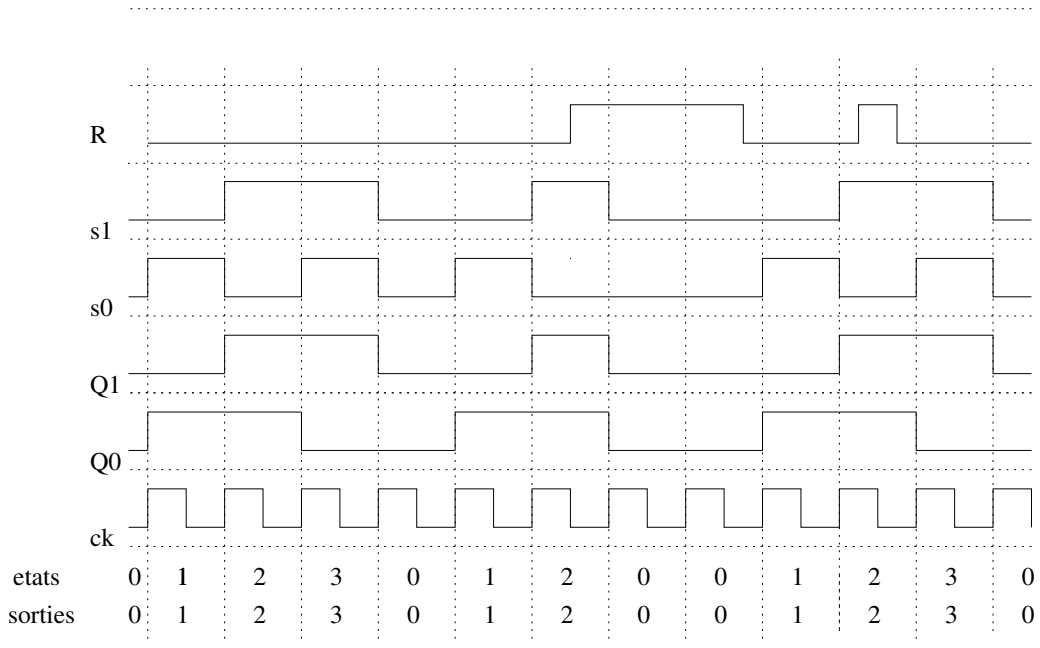
$D_1 = Q_0 \cdot \bar{R}$

$D_0 = \bar{Q_1} \cdot \bar{R}$

$s_1 = Q_1$

$s_0 = Q_0 \oplus Q_1$

Voici le chronogramme du circuit :



Il n'y a plus de sorties parasites, mais lorsque l'entrée passe à 1, il y a un temps de latence avant que la sortie passe à 0.

Nous avons choisi d'utiliser des bascules D pour nos circuits (ce sont les moins coûteuses) mais nous aurions pu choisir n'importe qu'elle autre bascule. Par exemple des bascules T. On rappelle que pour une telle bascule, si l'entrée de la bascule est T=0, la sortie ne change pas et si T=1, la sortie est inversée.

Pour construire ce circuit on considère donc la table suivante :

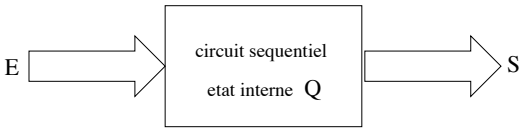
entrée	état courant		sortie		état suivant		entrées bascules	
R	Q_1	Q_0	s_1	s_0	Q'_1	Q'_0	T_1	T_0
0	0	0	0	1	0	1	0	1
0	0	1	0	0	1	1	1	0
0	1	1	1	1	1	0	0	1
0	1	0	1	0	0	0	1	0
1	0	0	0	1	0	0	0	0
1	0	1	0	0	0	0	0	1
1	1	1	1	1	0	0	1	1
1	1	0	1	0	0	0	1	0

Ici, on voit très bien qu'exprimer T_1 et T_0 à la place de D_1 et D_0 n'est pas très rentable, le circuit va être plus compliqué. Par contre on remarque que changer le type des bascules change énormément le circuit combinatoire, il peut donc être intéressant d'envisager les différentes possibilités.

8.2 Abstraction de circuits séquentiels

Afin de faciliter l'implantation de circuits séquentiels, on modélise (abstrait) un circuit par une machine à états.

Les circuits séquentiels ayant une mémoire du passé, ils possèdent un **état (interne)** représentant l'état de la mémoire à un instant donné.



— la sortie S du circuit est fonction des entrées E et de l'état du circuit Q

$$S = f(E, Q)$$

— le nouvel état du circuit Q' est fonction des entrées E et de l'état précédent Q

$$Q' = g(E, Q)$$

Exemple 8.1 (La bascule RS)

S	R	Q^{t+1}
0	0	Q^t
0	1	0
1	0	1
1	1	*indéfini*

— *La bascule RS a deux états internes correspondant à la valeur mémorisée :* $\{0, 1\}$
— *Elle possède trois entrées correspondant aux valeurs que peuvent prendre* R *et* S *:* $\{\bar{R}\bar{S}, R\bar{S}, \bar{R}S\}$ *(avec* $\bar{R}$ *pour* $R = 0$ *et* R *pour* $R = 1$*, de même pour* S*).*
— *Elle possède deux sorties :* $\{0, 1\}$
— *les fonctions* f *et* g *calculant la sortie et le nouvel état sont les suivantes :*

E	Q	$S = f(E,Q)$	$Q' = g(E,Q)$
$\bar{R}\bar{S}$	0	0	0
$\bar{R}\bar{S}$	1	1	1
$R\bar{S}$	0	0	0
$R\bar{S}$	1	0	0
$\bar{R}S$	0	1	1
$\bar{R}S$	1	1	1

Exemple 8.2 (Un 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 : $E = \{r_0 r_1, \bar{r_0} r_1, r_0 \bar{r_1}, \bar{r_0}\bar{r_1}\}$,*
- *sorties : $S = \{s_0 s_1, \bar{s_0} s_1, s_0 \bar{s_1}, \bar{s_0}\bar{s_1}\}$,*
- *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 : $Q = \{0, 1\}$,*
- *fonction de sortie f et fonction d'entrée g donnés par :*

E	Q	$S = f(E,Q)$	$Q' = g(E,Q)$
$\bar{r_0}\bar{r_1}$	0	$\bar{s_0}\bar{s_1}$	0
$\bar{r_0}\bar{r_1}$	1	$\bar{s_0}\bar{s_1}$	1
$r_0\bar{r_1}$	0	$s_0\bar{s_1}$	0
$r_0\bar{r_1}$	1	$s_0\bar{s_1}$	0
$\bar{r_0}r_1$	0	$\bar{s_0}s_1$	1
$\bar{r_0}r_1$	1	$\bar{s_0}s_1$	1
$r_0 r_1$	0	$s_0\bar{s_1}$	0
$r_0 r_1$	1	$\bar{s_0}s_1$	1

8.3 Machine de Mealy

A partir d'une abstraction d'un circuit séquentiel, on construit une **machine de Mealy**.

8.3.1 Définition

Une **machine de Mealy** est définie par un quintuplet $(\mathcal{Q}, \iota, E, S, \mathcal{T})$ où

- $\mathcal{Q}$, un ensemble fini d'états
- E est l'alphabet des valeurs d'entrées
- S est l'alphabet des valeurs de sorties
- $\mathcal{T}$ est l'ensemble des transitions : $\mathcal{T} \subseteq \mathcal{Q} \times E \times S \times \mathcal{Q}$

Une transition $(q, e, s, q') \in \mathcal{T}$ sera notée $q \xrightarrow{e/s} q'$.

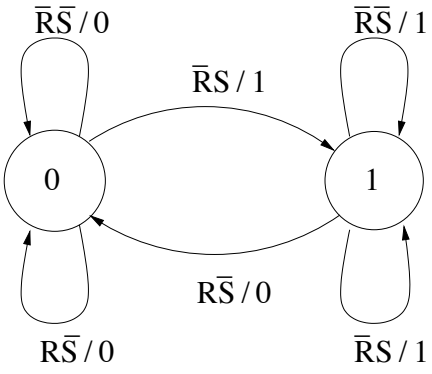
Chaque machine de Mealy $\mathcal{M}$ peut être représentée par un graphe dirigé : les sommets du graphe sont les états de la machine, et on dessine une flèche de l'état q à l'état q' étiquetée par e/s ssi il existe une transition $q \xrightarrow{e/s} q'$ dans $\mathcal{T}$

8.3.2 De l'abstraction à la machine de Mealy

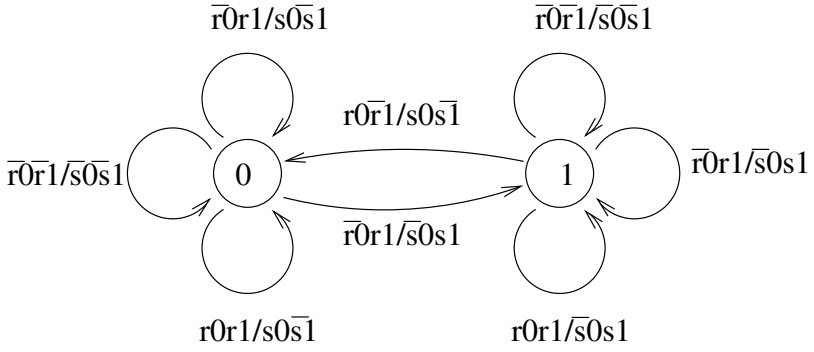
On considère une abstraction d'un circuit séquentiel donnée par
— ses entrées E,
— ses sorties S,
— ses états internes Q
— sa fonction de sortie f
— sa fonction d'états g.
On associe à cette abstraction une machine de Mealy donnée par :
— $\mathcal{Q} = Q$
— ι est un état initial quelconque ($\iota \in \mathcal{Q}$)
— E est l'alphabet des valeurs d'entrées
— S est l'alphabet des valeurs de sorties
— l'ensemble $\mathcal{T}$ des transitions est donné par
$(p, e, s, q) \in \mathcal{T}$ ssi $s = f(e, p)$ et $q = g(e, p)$.

Exemple 8.3 (Bascule RS (suite)) *La figure 8.1 représente une machine de Mealy correspondant à la bascule RS de l'exemple 8.1.*

FIGURE 8.1 – La bascule RS



Exemple 8.4 (Circuit d'arbitrage (suite)) *La machine de Mealy associée au circuit d'arbitrage dont l'abstraction est donnée dans l'exemple 8.2 est la suivante :*



8.3.3 Fonctionnement

Soit $\mathcal{M}$ une machine de Mealy $(\mathcal{Q}, \iota, E, S, \mathcal{T})$, et $\mathcal{A}_\mathcal{M}$ l'automate fini associé.

Un **fonctionnement** de la machine $\mathcal{M}$ est un couple de mots $(e_1e_2 \dots e_n, s_1s_2 \dots s_n)$ avec $e_i \in E$ et $s_i \in S$ tel que le mot $(e_1/s_1) \cdot (e_2/s_2) \cdots (e_n/s_n)$ est reconnu par l'automate $\mathcal{A}_\mathcal{M}$.

Voici un fonctionnement de la machine de la bascule RS partant de l'état 0 :

$$(\bar{R}\bar{S}R\bar{S}\bar{R}SR\bar{S}\bar{R}\bar{S}R\bar{S}, 001110)$$

8.4 Machine de Moore

Les machines de Moore s'utilisent dans le cas particulier où la sortie du système ne dépend que de l'état. On appelle ces systèmes strictement synchrones car le changement des sorties ne se fait qu'avec le changement d'état. Les machines de Moore s'opposent aux machines de Mealy pour lesquelles les sorties dépendent à la fois de l'état courant et des variables d'entrée.

Reprenons le cas de la bascule RS, la sortie **ne dépend que de l'état** dans lequel passe (ou reste) la bascule.

E	Q	$S = f(E, Q)$	$Q' = g(E, Q)$
$\bar{R}\bar{S}$	0	0	0
$\bar{R}\bar{S}$	1	1	1
$R\bar{S}$	0	0	0
$R\bar{S}$	1	0	0
$\bar{R}S$	0	1	1
$\bar{R}S$	1	1	1

La sortie n'est alors fonction que de l'état dans lequel passe le circuit

$$S = f(Q')$$

Pour la bascule RS, la fonction f est l'identité.

8.4.1 Machine de Moore : définition

Une **machine de Moore** est définie par un sextuplet $(\mathcal{Q}, \iota, E, S, \mathcal{T}, f)$ où

— $\mathcal{Q}$, un ensemble fini d'états

— E est l'alphabet des valeurs d'entrées

— S est l'alphabet des valeurs de sorties

— $\mathcal{T}$ est l'ensemble des transitions

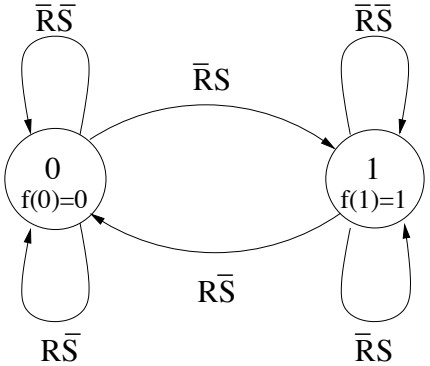
$$\mathcal{T} \subseteq \mathcal{Q} \times E \times \mathcal{Q}$$

— $f : \mathcal{Q} \mapsto S$

On note (q, e, q') par $q \xrightarrow{e} q'$

Comme pour les machines de Mealy, on peut donner une représentation une machine de Moore par un graphe. Cette fois les arêtes ne sont étiquetées que par des entrées, la fonction de sortie, qui ne dépend que de l'état est donnée dans les sommets.

Exemple 8.5 (Bascule RS (suite)) *Voici la machine de Moore décrivant le comportement de la bascule RS :*



8.4.2 Fonctionnement

Soit $\mathcal{M}$ une machine de Moore $(\mathcal{Q}, \iota, E, S, \mathcal{T}, f)$, et $\mathcal{A}_\mathcal{M}$ l'automate associé,
Un fonctionnement de la machine $\mathcal{M}$ est un couple de mots $(e_1e_2\dots e_n, f(q_1)f(q_2)\dots f(q_n))$ tel que :
$(\iota, \varepsilon) \to (q_1, e_1) \to (q_2, e_2) \to (q_n, e_n)$.

8.5 Comparaison de Modèles

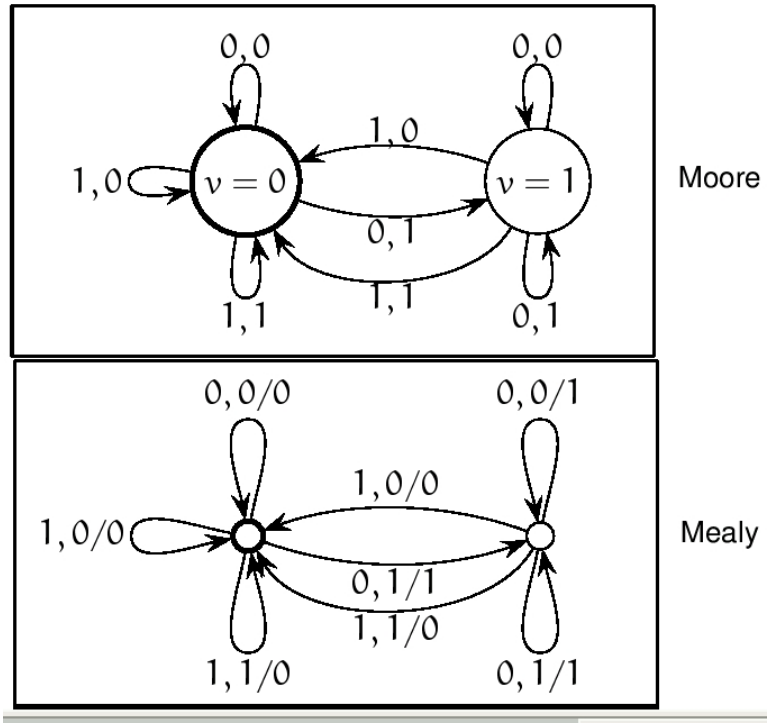
Théorème 8.6 *Pour toute machine de Moore $\mathcal{M} = (\mathcal{Q}, \iota, E, S, \mathcal{T}, f)$, il existe une machine de Mealy $\mathcal{M}' = (\mathcal{Q}', \iota', E', S', \mathcal{T}')$ qui possède les mêmes fonctionnements.*

Preuve. On pose :
— $E' = E$ et $S' = S$,
— $\mathcal{Q}' = \mathcal{Q}$ et $\iota' = \iota$,
et $\mathcal{T}'$ vérifie :

$$(q, e, s, q') \in \mathcal{T}' \text{ iff } (q, e, q') \in \mathcal{T} \text{ et } s = f(q')$$

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

□



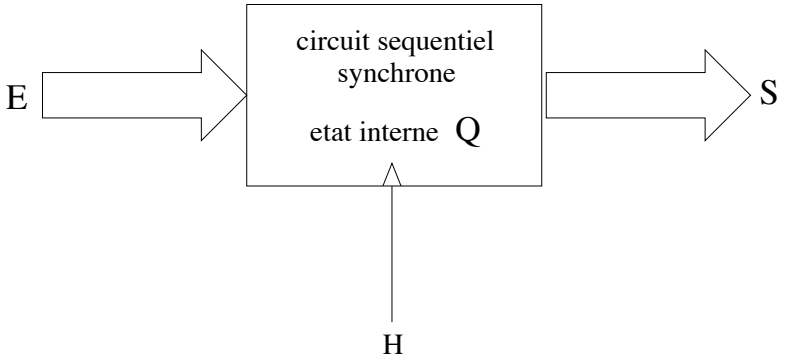
Théorème 8.7 *Pour toute machine de Mealy $\mathcal{M} = (\mathcal{Q}, \iota, E, S, \mathcal{T})$, il existe une machine de Moore $\mathcal{M}' = (\mathcal{Q}', \iota', E', S', \mathcal{T}', f')$ qui possède les mêmes fonctionnements.*

Preuve. On pose :
— $E' = E$ et $S' = S$,
— $\mathcal{Q}' = \mathcal{Q} \times S$
— $\iota' = (\iota, s_0)$,
$\mathcal{T}'$ vérifie : pour tout s, $((q, s), e, (q', s')) \in \mathcal{T}'$ *ssi* $(q, e, s', q') \in \mathcal{T}$
et f vérifie : $f((q, s)) = s$ *pour tout* (q, s)
On prouve que tout fonctionnement de l'une est un fonctionnement de l'autre par récurrence sur la longueur des fonctionnements. □

8.6 Réalisation de circuits séquentiels synchrones

Les circuits séquentiels synchrones ont un fonctionnement synchronisé par une horloge.

L'état interne et la sortie du circuit ne sont modifiés qu'à des intervalles de temps (périodes) définis par l'horloge.



L'utilisation d'une abstraction du circuit à réaliser permet de simplifier grandement la conception qui à ramenée (dans le cas d'une réalisation cablée) à l'élaboration de deux circuits combinatoires.

On distingue deux types de réalisations à partir d'une abstraction. La réalisation cablée qui consiste en la création d'un circuit séquentiel, et la réalisation microprogrammée.

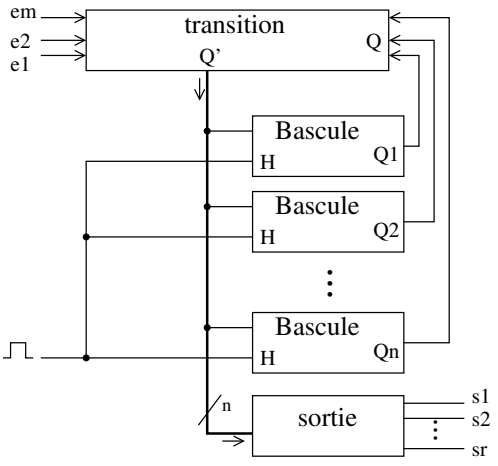
8.6.1 Réalisation cablée

8.6.1.1 À partir d'une machine de Moore

Supposons qu'on dispose d'une machine de Moore à 2^m entrées, 2^n états et 2^r sorties et dont la fonction de sortie est f et la fonction d'états est g. Par définition, la fonction f ne dépend que de l'état courant.

On réalise un circuit reproduisant le fonctionnement de la machine. Ce circuit possède m entrées (chaque entrée de la machine est codée par un vecteur de m booléens), r sorties (chaque sortie de la machine est codée par un vecteur de r booléens), et utilise n bascules (souvent des bascules D) qui serviront à mémoriser les états courants de la machine (chaque état est codé par un vecteur de n booléens).

Le circuit est le suivant :



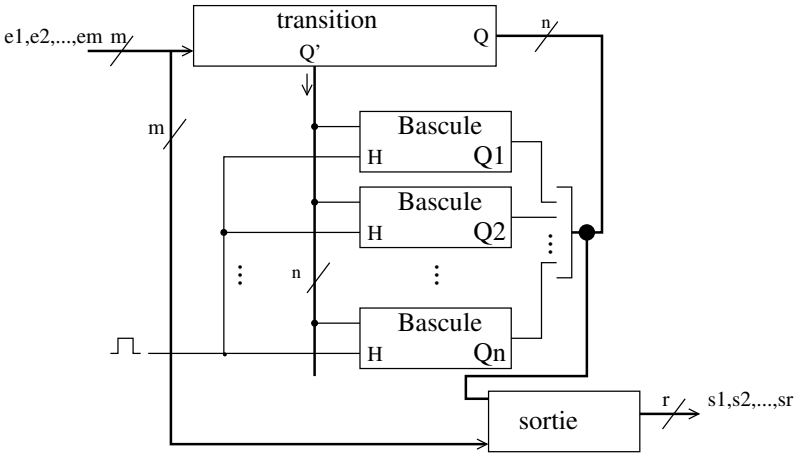
La fonction de transition et la fonction de sortie sont des circuits combinatoires et sont respectivement une implantation de la fonction d'états g et une implantation de la fonction de sortie f.

8.6.1.2 À partir d'une machine de Mealy

Supposons qu'on dispose d'une machine de Mealy à 2^m entrées, 2^n états et 2^r sorties et dont la fonction de sortie est f et la fonction d'états est g. Cette fois, la fonction f dépend de l'entrée et de l'état courant.

On réalise un circuit reproduisant le fonctionnement de la machine. Ce circuit possède m entrées (chaque entrée de la machine est codée par un vecteur de m booléens), r sorties (chaque sortie de la machine est codée par un vecteur de r booléens), et utilise n bascules qui serviront à mémoriser les états courants de la machine (chaque état est codé par un vecteur de n booléens).

Le circuit est le suivant :



La fonction de transition et la fonction de sortie sont des circuits combinatoires et sont respectivement une implantation de la fonction d'états g et une implantation de la fonction de sortie f.

8.6.1.3 Exemple : le distributeur

On considère le distributeur suivant :

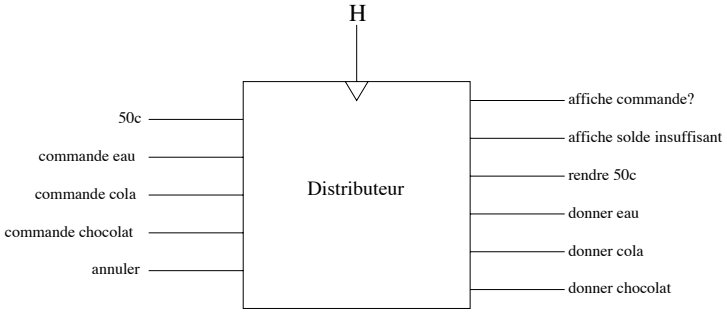
— le distributeur délivre des bouteilles d'eau à 50c, des canettes de cola à 50c et des barres chocolatées à 1 euro.

— le distributeur n'accepte que des pièces de 50c.

— la machine fonctionne selon le principe suivant :

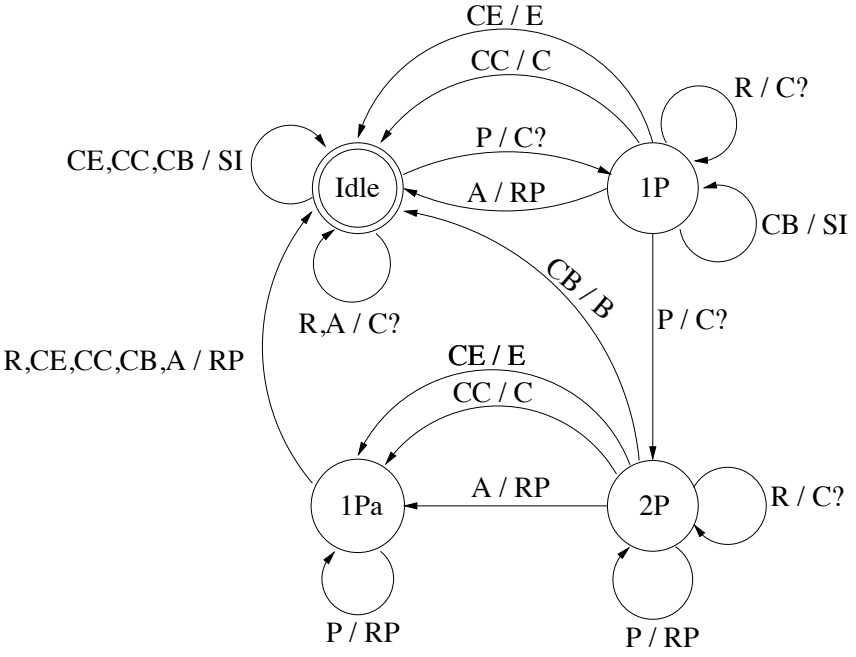
1. on introduit un certain nombre de pièces ; si ce nombre est strictement plus grand que 2 alors la pièce introduite est immédiatement rendue.
2. on peut (tenter de) commander dès qu'on a introduit une pièce ; si le solde est insuffisant, la machine le signale. Dans le cas contraire, la commande est honorée et la monnaie éventuelle est rendue.
3. tant que la commande n'est pas passée, on peut toujours annuler et les pièces introduites sont rendues.

Voici le circuit correspondant :



On suppose qu'au plus une ligne d'entrée est à 1.

Construisons la machine de Mealy correspondante.
Alphabet d'entrée E
— R : tous les entrées du circuit sont à 0
— P : l'entrée "50" du circuit est à 1, les autres sont à 0
— CE : l'entrée "eau" du circuit est à 1, les autres sont à 0
— CC : l'entrée "cola" du circuit est à 1, les autres sont à 0
— CB : l'entrée "chocolat" du circuit est à 1, les autres sont à 0
— A : l'entrée "annuler" du circuit est à 1, les autres sont à 0
Alphabet de sortie S
— C ? : la sortie "affiche commande ?" est à 1, les autres à 0
— SI : la sortie "affiche solde insuffisant" est à 1, les autres à 0
— RP : la sortie "rendre 50c" est à 1, les autres à 0
— E : la sortie "donner eau" est à 1, les autres à 0
— C : la sortie "donner cola" est à 1, les autres à 0
— B : la sortie "donner chocolat" est à 1, les autres à 0
Etats La machine peut prendre 4 états :
— Idle : la machine attend que quelqu'un vienne l'utiliser
— 1P : quelqu'un a introduit 50 centimes et n'a pas encore commandé
— 2P : quelqu'un a introduit 1 euro et n'a pas commandé
— 1Pa : une commande à 50 centimes vient d'être fournie, et il reste 50 centimes à rendre.
On obtient la machine suivante :



On peut maintenant facilement donner la réalisation cablée du distributeur qui comportera deux bascules D, 3 entrées et 3 sorties.

Nous commençons par donner le codage des entrées et sorties du circuit, et des états.

entrée machine	entrée circuit$(e1, e2, e3)$
R	$(0, 0, 0)$
P	$(0, 0, 1)$
CE	$(0, 1, 0)$
CC	$(1, 0, 0)$
CB	$(1, 1, 0)$
A	$(1, 1, 1)$

sortie machine	sortie circuit$(s1, s2, s3)$
$C?$	$(0, 0, 0)$
SI	$(0, 0, 1)$
RP	$(0, 1, 1)$
E	$(0, 1, 0)$
C	$(1, 0, 0)$
B	$(1, 1, 0)$

état machine	bascules(Q_1, Q_2)
$Idle$	$(0,0)$
$1P$	$(0,1)$
$2P$	$(1,0)$
$1Pa$	$(1,1)$

Voici les tables de vérité des circuits combinatoires $fct\_de\_transition$ et fct_de_sortie.

entrées			états courants		sorties			nouveaux états	
$e1$	$e2$	$e3$	Q_1	Q_2	$s1$	$s2$	$s3$	Q'_1	Q'_2
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	1
0	1	0	0	0	0	0	1	0	0
1	0	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	1	0	0
1	1	1	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1
0	0	1	0	1	0	0	0	1	0
0	1	0	0	1	0	1	0	0	0
1	0	0	0	1	1	0	0	0	0
1	1	0	0	1	0	0	1	0	1
1	1	1	0	1	0	1	1	0	0
0	0	0	1	0	0	0	0	1	0
0	0	1	1	0	0	1	1	1	0
0	1	0	1	0	0	1	0	1	1
1	0	0	1	0	1	0	0	1	1
1	1	0	1	0	1	1	0	0	0
1	1	1	1	0	0	1	1	1	1
0	0	0	1	1	0	1	1	0	0
0	0	1	1	1	0	1	1	1	1
0	1	0	1	1	0	1	1	0	0
1	0	0	1	1	0	1	1	0	0
1	1	0	1	1	0	1	1	0	0
1	1	1	1	1	0	1	1	0	0

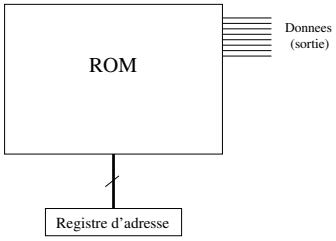
On peut alors facilement construire les circuits combinatoires correspondants.

8.6.2 Microprogrammation

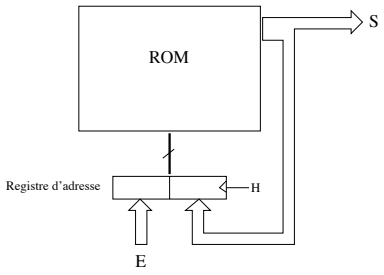
La microprogrammation est une alternative à la solution cablée pour l'implantation de machines de Mealy/Moore inventée par Maurice Wilkes en 1951. L'idée est de remplacer un circuit combinatoire par une ROM contenant la table de vérité du circuit.

On distingue 2 types de microprogrammation : la microprogrammation horizontale et la microprogrammation verticale. Nous considérons ici la microprogrammation horizontale.

On dispose d'une mémoire ROM contenant des données. Ces données sont accessibles via des adresse présente dans un "Registre d'adresse". Le circuit suivant place donc sur la sortie "Données" les données présentes dans la ROM à l'adresse se trouvant dans le "Registre d'adresse".



Pour la programmation d'une machine de Mealy/Moore, les adresses contenues dans le Registre d'adresse sont composée d'une entrée et d'un état de la machine. A l'adresse correspondante dans la ROM sont stockés l'état suivant et la sortie Le chargement dans le registre d'adresse est piloté par l'horloge H.



Voici une réalisation microprogrammée du distributeur :

Registre d'adresses					données				
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	1
0	1	0	0	0	0	0	1	0	0
1	0	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	1	0	0
1	1	1	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1
0	0	1	0	1	0	0	0	1	0
0	1	0	0	1	0	1	0	0	0
1	0	0	0	1	1	0	0	0	0
1	1	0	0	1	0	0	1	0	1
1	1	1	0	1	0	1	1	0	0
0	0	0	1	0	0	0	0	1	0
0	0	1	1	0	0	1	1	1	0
0	1	0	1	0	0	1	0	1	1
1	0	0	1	0	1	0	0	1	1
1	1	0	1	0	1	1	0	0	0
1	1	1	1	0	0	1	1	1	1
0	0	0	1	1	0	1	1	0	0
0	0	1	1	1	0	1	1	1	1
0	1	0	1	1	0	1	1	0	0
1	0	0	1	1	0	1	1	0	0
1	1	0	1	1	0	1	1	0	0
1	1	1	1	1	0	1	1	0	0

8.7 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. Normalement, pour un même problème, une machine de Moore demande plus d'états que la machine de Mealy équivalente Toutefois, assez souvent on préfère la solution de type machine de Moore. En effet, les sorties d'une machine Mealy ne sont pas synchronisées avec l'horloge, pouvant changer en même temps qu'une entrée externe. Tandis que les états changent seulement avec le flanc de l'horloge, les sorties peuvent changer à tout moment, avec le changement d'un état ou d'une entrée (dans une machine de Moore, les sorties changent uniquement avec le changement d'un état). Cet asynchronisme amène comme conséquence des durées quelconques pour les sorties, souvent inférieures à une période d'horloge. Si la sortie d'une machine séquentielle sert à contrôler une action externe, la durée du signal est un facteur très important. Comme les sorties des machines de type Moore sont synchronisées, leur durée est toujours un multiple de la période du signal d'horloge.

Chapitre 9

Assembleur

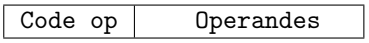
9.1 Langage d'assemblage

Un langage d'assemblage ou langage assembleur 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 dits " mnémoniques " (...), c'est-à-dire faciles à retenir. Le programme assembleur convertit ces mnémoniques en langage machine en vue de créer par exemple un fichier exécutable. Le langage machine est le seul langage qu'un processeur puisse exécuter. Chaque famille de processeur utilise un jeu d'instructions différent. Ainsi le langage assembleur, représentation exacte du langage machine, est spécifique à chaque architecture de processeur. (Citation Wikipédia.org)

Le **langage machine** est le langage directement interprétable par le processeur. Il est défini par un ensemble d'**instructions** que le processeur exécute directement.

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



Un programme en langage machine est une suite de mots codant opérations et opérandes

Chaque processeur possède son propre langage machine.

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
— un certain nombre de manière d'accéder à la mémoire : **modes d'adressage**

9.1.1 Jeu d'instructions

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
— CISC (Complex Instruction Set Computer)

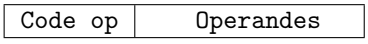
  Pentium

Leurs caractéristiques sont les suivantes :
- RISC :
  - jeu d'instructions de taille limitée
  - instructions simples
  - format des instructions petit et fixé
  - modes d'adressage réduits
- CISC :
  - 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 types ont leurs avantages et leurs inconvénients :
- CISC
  - ⊕ programmation de plus haut niveau
  - ⊕ programmation plus compacte (écriture plus rapide et plus élégante des applications)
  - ⊕ 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
  - ⊖ compilation plus compliquée

9.1.2 Modes d'adressage

Les instructions du langage machine manipulent des données. Selon où ces données se trouvent, on parle de différents **modes d'adressage**.



Comment interpréter `Operandes` pour trouver les données de l'instruction `Code op` ?
- *adressage implicite* : l'instruction opère sur une donnée qui se trouve à un emplacement précis et déterminé du processeur (par exemple l'accumulateur). Dans ce cas, il n'est pas nécessaire de spécifier l'adresse du registre en question.
- *adressage par registres* : `Operandes` contient le(s) numéro(s) du (des) registre(s) où se trouvent les données manipulées par l'instruction.
- *adressage direct (ou direct restreint)* : `Operandes` est l'adresse (ou un fragment de l'adresse) où se trouve la donnée en mémoire.
- *adressage relatif* : `Operandes` contient un déplacement relatif par rapport à une adresse qui se trouve dans un registre précis (par exemple, le compteur ordinal PC).
- *adressage indirect* : `Operandes` contient le numéro d'un registre dont le contenu est l'adresse où se trouve la donnée en mémoire.
- *adressage (indirect) indexé* : `Operandes` contient le numéro d'un registre contenant une adresse a. La donnée est en mémoire à l'adresse $a+i$, où i est le contenu d'un autre registre dans `Operandes` ou d'un registre spécifique, appellé *registre d'index*
- *adressage immédiat* : `Operandes` est la valeur utilisée par l'instruction

9.1.3 Cycle d'exécution d'une instruction

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

1. **Récupérer (en mémoire) l'instruction à exécuter** : $RI \leftarrow Mémoire[PC]$
   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)

9.2 Assembleur MIPS

Nous allons travailler avec le langage assembleur MIPS, qui est l'assembleur du processeur MIPS R2000 (processeur de type RISC)

Ces processeurs sont utilisés par exemple par NEC, SGI, Sony PSP, PS2. Le langage assembleur MIPS est proche des autres assembleurs RISC.

9.2.1 Processeur MIPS

Processeur 32 bits 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

NB : une instruction est donc codée sur 32 bits.

9.2.2 Mémoire

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

9.2.3 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'inter-opérabilité entre pro- grammes 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

9.2.4 Instructions MIPS

Nous donnons ici quelques exemples d'instructions. Vous pouvez vous référer à la documentation MIPS pour avoir l'ensemble des instructions.

9.2.4.1 Arithmétique

Code C	Assembleur
`A = B + C`	`add $s0, $s1, $s2`

Toutes les opérandes se trouvent dans des registres. Ici, A↦$s0, B↦$s1, C↦$s2.
Pour les instructions arithmétiques, le résultat est toujours placé dans la première opérande.

Code C	Assembleur

`A = B + C + D`
`E = F - A`
	`add $t0, $s1, $s2`
	`add $s0, $t0, $s3`
	`sub $s4, $s5, $s0`

Ici, A↦$s0, B↦$s1, C↦$s2, D↦$s3, E↦$s4, F↦$s5.
Le jeu d'instruction MIPS contient toutes les opérations arithmétique et booléennes de base.

9.2.5 Pseudo-instruction `move`

Comment traduire `A=B` ?
Sachant que A↦$s0, B↦$s1 et que le registre $0 vaut toujours 0 on peut écrire :

```
add $s0, $0, $s1
```

Il vaut mieux utiliser l'instruction `move` :

Code C	Assembleur
`A = B`	`move $s0, $s1`

`move` est une pseudo-instruction : sa traduction en langage machine est celle de `add $s0, $0, $s1`.

9.2.6 Pseudo-instruction `li`

`li r, imm` (load immediate) charge la valeur `imm` (sur 32 bits) dans le registre `r`.

9.2.7 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 m obtenue en ajoutant 10 au nombre stocké dans la registre $3.

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

9.2.8 Branchements conditionnels

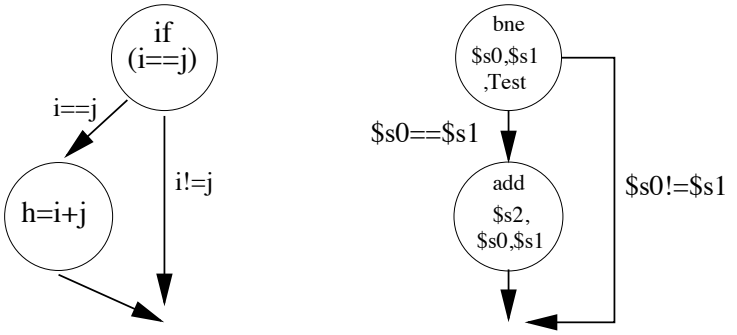
Syntaxe

`bne $t0, $t1, Label`

Si la valeur contenue dans le registre $t0 n'est pas égale à celle stockée dans le registre $1 alors la prochaine instruction à exécuter est celle placée après l'étiquette `Label`

`beq $t0, $t1, Label`

Si la valeur contenue dans le registre $t0 est égale à celle stockée dans le registre $1 alors la prochaine instruction à exécuter est celle placée après l'étiquette `Label`



Code C	Assembleur

`if (i==j) h =i+j;`	`bne $s0, $s1, Test`
	`add $s2, $s0, $s1`
	`Test:`

Ici, i↦$s0, j↦$s1, h↦$s2.

9.2.9 Branchements inconditionnels

Syntaxe

`j Label`

La prochaine instruction à exécuter est celle placée après l'étiquette `Label` : $PC \leftarrow$ `Label`.

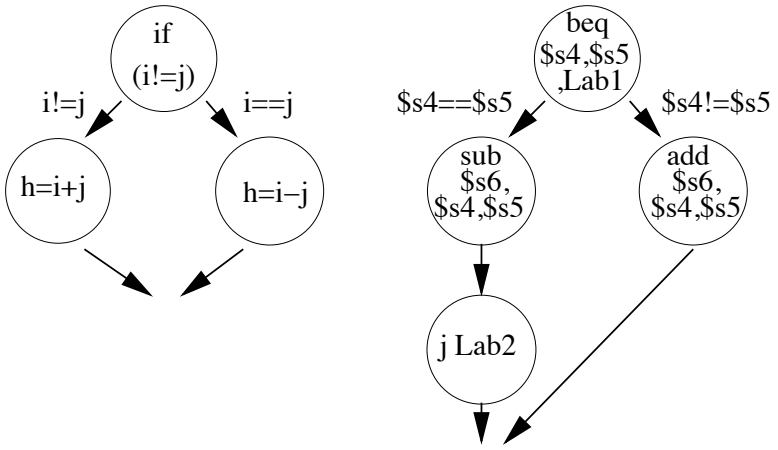
`jr registre`

La prochaine instruction à exécuter est celle à l'adresse se trouvant dans le registre `registre` : $PC \leftarrow$ `registre`.

`jal Label`

La prochaine instruction à exécuter est celle placée après l'étiquette `Label` et l'adresse de l'instruction suivant l'instruction courante (adresse de retour est stockée dans $ra : $\$ra \leftarrow PC + 4$, $PC \leftarrow$ `Label`.

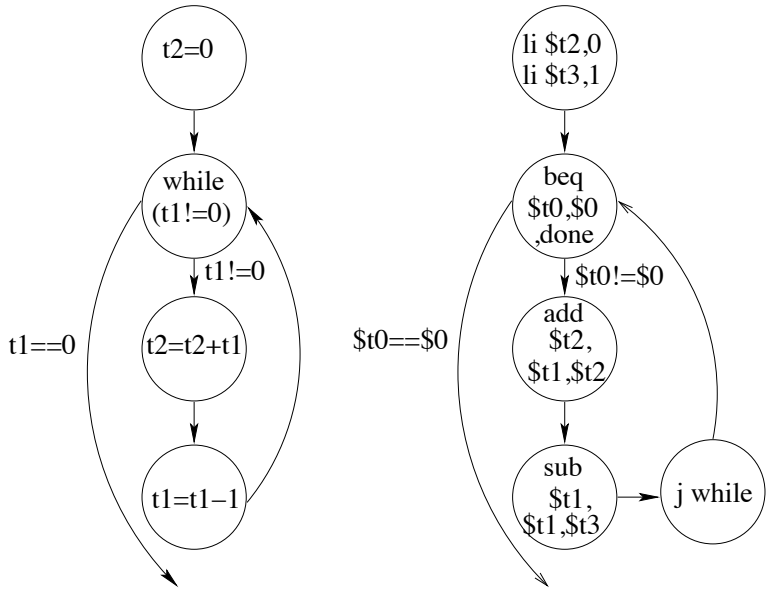
En utilisant ces instructions, on peut effectuer des branchements plus complexes que le `if ... then`. Voici par exemple comment programmer un `if ... then ... else`.



Code C	Assembleur

`if (i!=j) h =i+j`
`else h =i-j`
	`beq $s4, $s5, Lab1`
	`add $s6, $s4, $s5`
	`j Lab2`
	`Lab1:sub $s6, $s4, $s5`
	`Lab2:`

Ici, i↦$s4, j↦$s5, h↦$s6.
On peut également programmer des boucles. Voici l'exemple d'une boucle `while`.



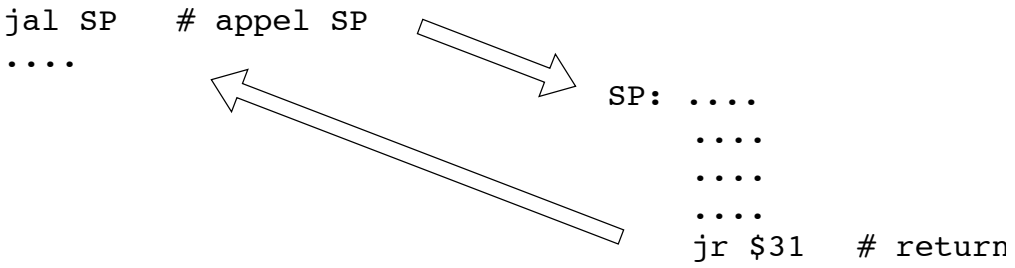
Code C	Assembleur

`t2=0`
`while (t1 != 0){`
`t2 = t2 + t1`
`t1=t1-1 }`
	`li $t2, 0`
	`li $t3, 1`
	`while:beq $t1, $0, done`
	`add $t2, $t1, $t2`
	`sub $t1, $t1, $t3`
	`j while`
	`done:`

`li registre, valeur` charge la valeur `valeur` dans le registre `registre`

9.2.10 Appel de sous-programmes

L'instruction `jal SP` permet d'exécuter le sous-programme de label `SP`, la sauvegarde de l'adresse de retour étant réalisée par cette instruction (dans le registre $31 ou $ra).



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'enchainement 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**.

Les registres $a0-$a3 sont ceux qui ne sont pas sauvegardés car ils contiennent lors de l'appel la valeur des paramètres effectifs et au retour les valeurs retournés par le sous-programme.

Appel de sous-programmes : 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 élement (et le récuperer dans un registre)

Ces deux instructions n'existent pas en assembleur MIPS, mais elles peuvent être "simulées"
— en utilisant les intructions `sw` et `lw`
— en stockant l'adresse du sommet de pile dans le registre $sp (le pointeur de pile)

Traditionnellement, la pile croît vers les adresses les moins élevées.

Appel de sous-programmes : politique de gestion de la pile Deux politiques de sauvegarde des registres :
— *sauvergarde par l'appelant* : le programme appelant sauvegarde tous les registres sur la pile (avant l'appel).
— *sauvergarde 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

9.2.11 Appel de sous-programmes : exemple

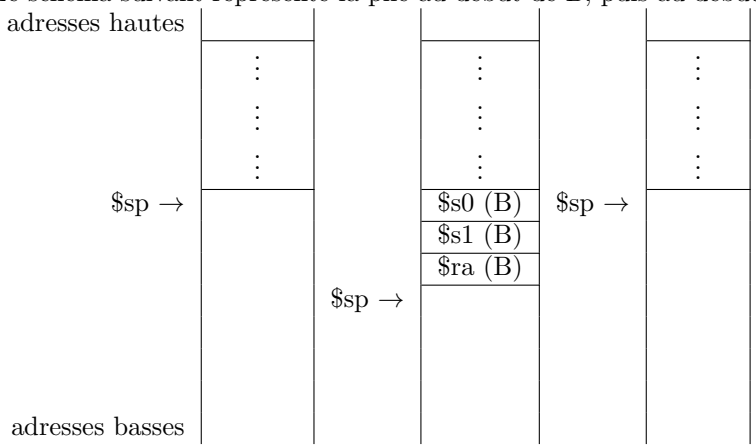
On a deux sous-programmes B et C, le programme B appelle le programme C. Avant d'appeler C on souhaite sauvegarder les valeurs des registres `$s0`, `$s1`, et l'emplacement où se fera le retour de B dans le programme principal, stocké dans le registre `$ra`. Initialement, le sommet de la pile est pointé par `$sp`. On sauvegarde d'abord `$s0` à cet emplacement de la pile, puis `$s1` 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
    ...
    sw $s0,0($sp)    sauvegarde de $s0 dans la pile
    sw $s1,-4($sp)   sauvegarde de $s1 dans la pile
    sw $ra,-8($sp)   sauvegarde de l'adresse de retour de B
    li $t0,12
    sub $sp,$sp,12   ajustement du sommet de pile
    jal C            appel du sous-programme C
    lw $ra,4($sp)    restauration de l'adresse de retour de B
    lw $s1,8($sp)    restauration de $s1
    sw $s0,12($sp)   sauvegarde de $s0
    li $t0,12
    add $sp,$sp,12   ajustement du sommet de pile
    ...
    jr $ra
    ...              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.



9.3 De l'assembleur à l'exécution

9.3.1 Format d'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 instruction 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.

9.3.1.1 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

	op	*rs*	*rt*	*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.

9.3.1.2 Format d'instructions J

6 bits	*26 bits*
Code op	*adresse*

	op	*adresse 26 bits*
`j 1000`	2	1000
`jal 1000`	3	1000

Codage des adresses
Les adresses dans les instructions ne sont **pas sur 32 bits** !
— Pour les instructions de type I : 16 bits
$\Longrightarrow$ Adresse = PC + signé(16 bits) * 4 adressage relatif
— Pour les instructions de type J : 26 bits
$\Longrightarrow$ On obtient l'adresse d'un mot mémoire (de 32 bits) en ajoutant devant les 26 bits les 4 bits de poids fort de PC (Il faut multiplier par 4 pour l'adresse d'un octet)

9.3.1.3 Format d'instructions R

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

— *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*
`add $1,$2,$3`	0	2	3	1	0	32
`sub $1,$2,$3`	0	2	3	1	0	34
`slt $1,$2,$3`	0	2	3	1	0	42
`jr $31`	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.

9.3.2 Exemple

Code C	Assembleur MIPS

`while (tab[i] == k)`
`  i = i+j;`
	`Loop :  mul $9, $19, $10`
	`  lw $8 , Tstart($9)`
	`  bne $8 , $21, Exit`
	`  add $19, $19, $20`
	`  j Loop`
	`Exit :`

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

— $\overbrace{80008+4}^{PC}+2*4$
— $20000*4=80000$

Chapitre 10

Processeur

10.1 Description d'un processeur

On trouve au sein d'un processeur :

— des éléments de mémorisation : (banc de) registres - cache
— des éléments de calcul : unité arithmétique et logique (UAL-ALU) - unités de calcul flottant (FPU - Floating Point Unit)
— des éléments de commandes : unité de contrôle/commande

Nous détaillons ici ces différents éléments.

10.1.1 Unité de calcul

L'unité de calcul est constituée de plusieurs unités :

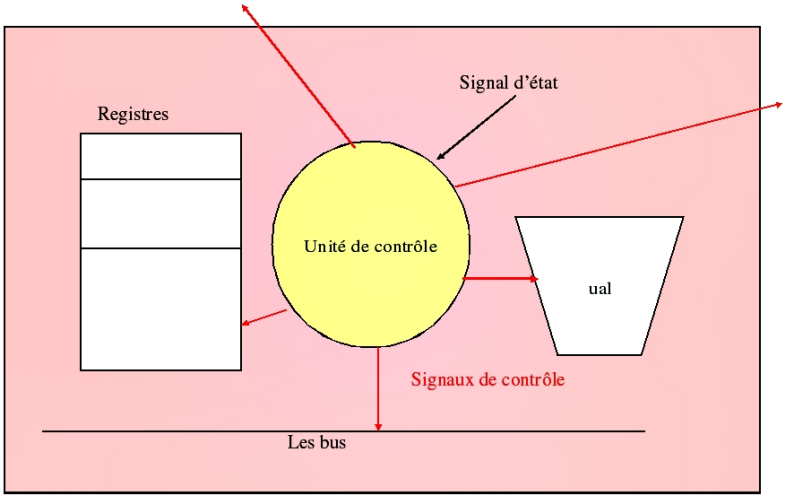
— l'Unité Arithmétique et Logique (ALU), qui gère les calculs sur les entiers et opérations booléennes
— l' Unité de calcul flottant : (FPU - Floating Point Unit), qui effectue les calculs les flottants : sqrt, sin, ...
— l'unité multimédia, qui gère le calcul vectoriel (même instruction sur plusieurs donnée en parallèle). Par exemple dans les processeurs Intel MMX et SSE, AMD 3DNow !

On peut trouver plusieurs unités de calculs au sein d'un processeur : par exemple 3 ALU pour Pentium

10.1.2 Unité de contrôle/commande

Cette unité coordonne le fonctionnement des autres éléments pour exécuter la séquence d'instructions constituant le programme. Pour exécuter une instruction, deux cycles se succèdent

— recherche de l'instruction à exécuter
— exécution de l'instruction



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 : contient l'adresse de la donnée à lire ou à écrire en mémoire.
  - registres de données : 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 évènements
- **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

10.1.2.1 Cycle d'exécution d'une instruction

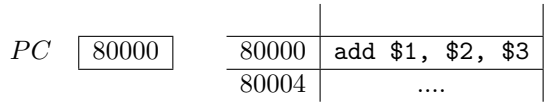
Nous détaillons ici les deux cycles formant le cycle d'exécution d'une instruction :
Le cycle de recherche :
- On récupère dans RI l'instruction à exécuter (celle à l'adresse contenue dans PC)
- On incrémente de compteur ordinal PC

Plus finement, utilisation des registres d'adresses et de données

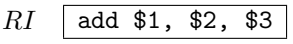
Le cycle d'exécution :
- On décode l'instruction
- Lire les adresses et les registres nécessaires à l'instruction
- Déterminer que faire pour cette instruction
- Le faire (ou le faire faire) (**utilisation d'une unité de calcul**)

Illustrons ces notions par un exemple :

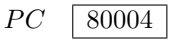
Cycle de recherche



1 On récupère l'instruction à exécuter
   — On met PC dans RA (le registre d'adresse)
   — On envoie un ordre de lecture à la mémoire
   — On place le contenue de RD (le registre de donnée) dans RI



2 On incrémente le compteur ordinal PC
   — Soit PC est muni d'un dispositif d'incrémentation
   — Soit on utilise l'ALU



Cycle de recherche

3 Décodage de l'instruction (**Décodeur**)
   — identification d'une addition entre deux registres avec placement du résultat dans un registre

4 Préparation des données (**Séquenceur**)
   — On place les contenus des registres $2 et $3 dans les deux registres d'entrée de l'ALU

5 Déterminer ce qu'il faut faire (**Séquenceur**)
   — Envoi du signal de l'opération d'addition à l'ALU

6 Le faire (**Séquenceur**)
   — 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

10.1.2.2 Horloge

L'horloge définit le cycle de base appelé cycle machine. Elle est utilisée pour synchroniser chaque étape des cycles de recherche et d'exécution.

L'exécution du cycle de recherche ou d'exécution **prend un certain nombre** de cycles de base (dépendant de l'instruction).

On définit le cycle CPU comme le temps d'exécution minimal d'une instruction (recherche + exécution)

10.1.2.3 Séquenceur

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

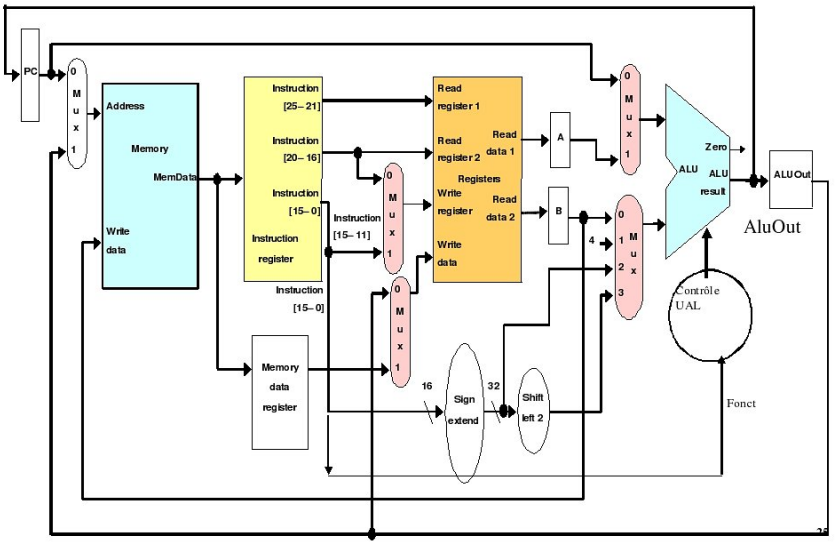
10.1.3 Chemin de données

Un **chemin de données** est défini par

— l'ensemble des composants requis pour l'exécution d'une instruction : PC, UAL, banc de registres, mémoire, ...

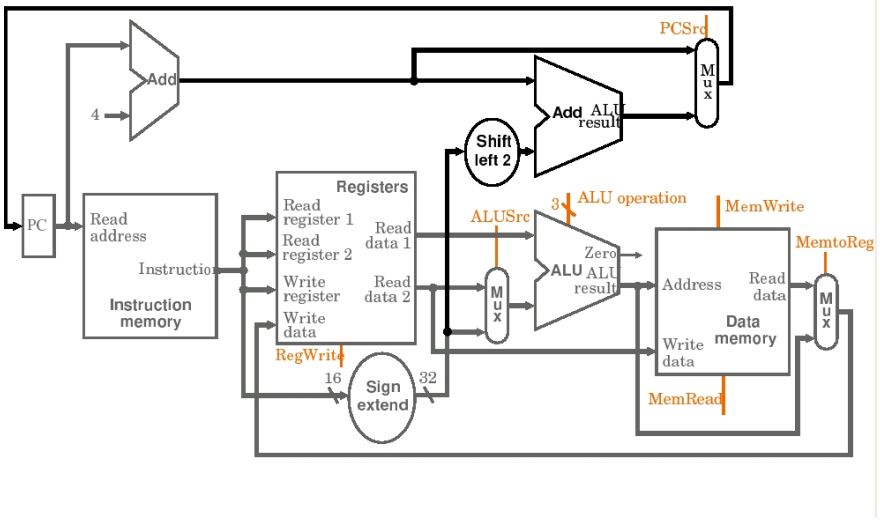
— les liens entre ces composants : flux de données, signaux de lecture/écriture, multiplexage des unités partagées, ...

Selon les instructions, les composants requis et les liens existants entre eux varient

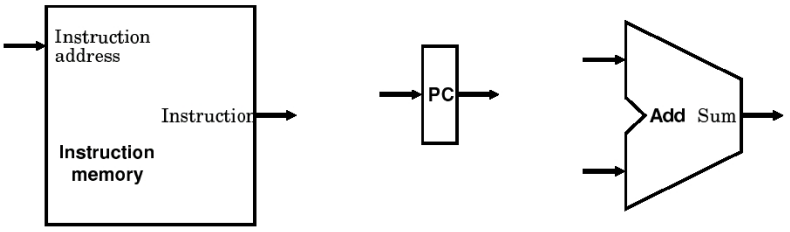


10.2 Processeur : fonctionnement

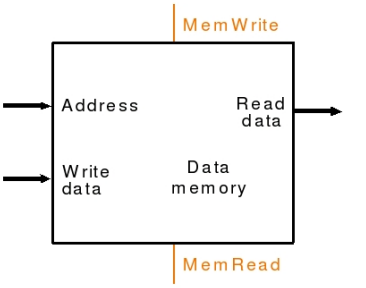
Nous illustrons le fonctionnement d'un processeur en prenant pour exemple d'étude un processeur MIPS dont voici l'architecture.



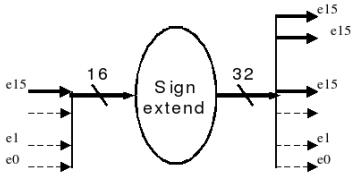
Les différents composant du processeur MIPS sont détaillés ci-dessous :



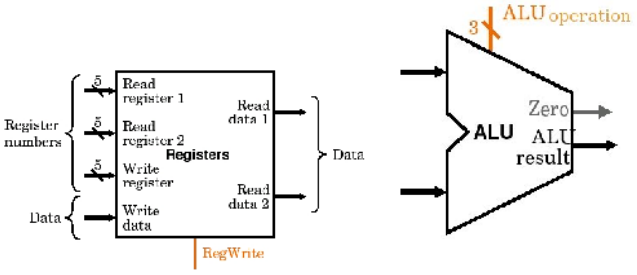
Mémoire d'instructions | Compteur programme | Additionneur



Mémoire de données



Extension signée



Registres ALU

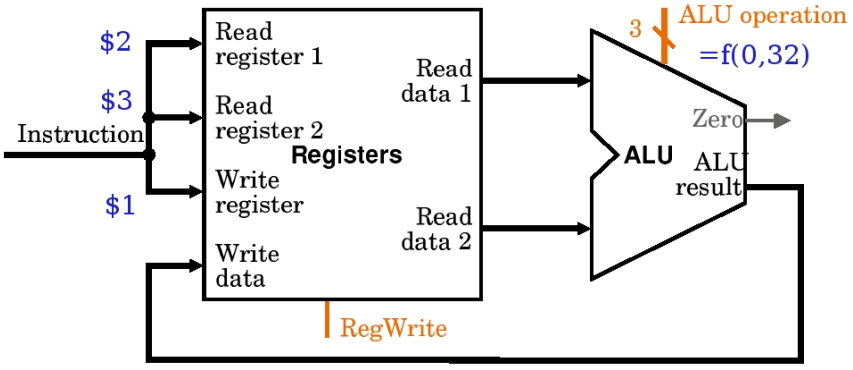
10.2.1 Exécution d'une instruction

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	adresse sur 16 bits		
Format J	Code op	adresse sur 26 bits				

Exécution de : `add $1,$2, $3`

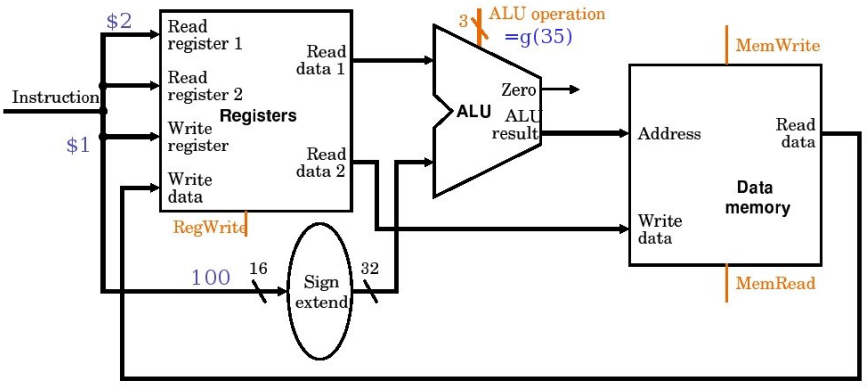
Codeop	rs	rt	rd	sa	funct
0	2	3	1	0	32



— 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	adresse sur 16 bits
35	2	1	100

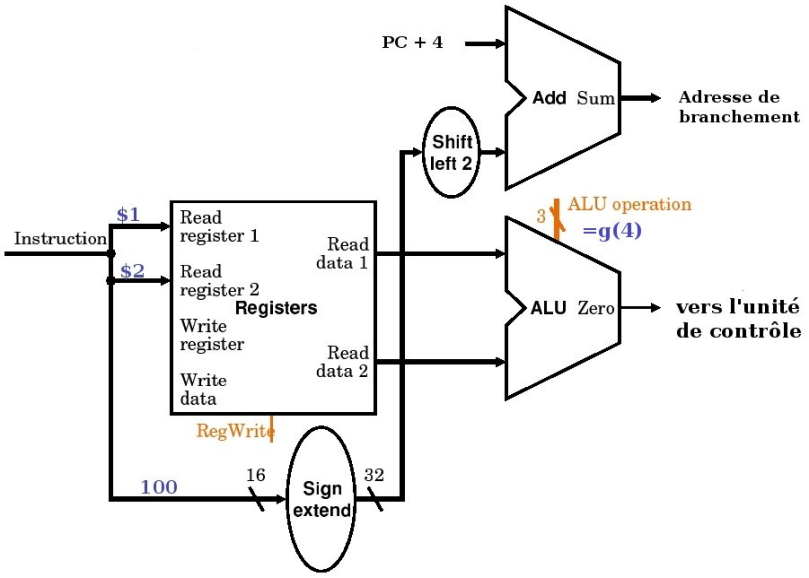


Le signal MemRead est activité.

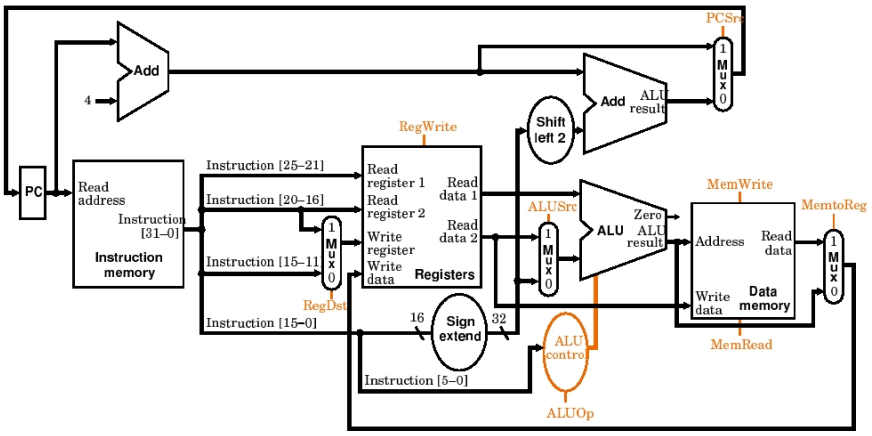
— "adresse 16 bits" est un déplacement relatif signé
— les signaux MemWrite et MemRead contrôlent respectivement l'écriture et la lecture dans la mémoire.

Exécution de : `beq $1,$2, 100`

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



10.2.2 Contrôle de l'ALU



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

10.2.3 L'unité de contrôle