Architecture des Ordinateurs

1ère partie

Olivier Temam

Table des Matières

1 REPRESENTATION DES NOMBRES 3

1.1 REPRESENTATION DES NOMBRES ENTIERS 3
1.2 REPRESENTATION DES NOMBRES REELS 5

2 CIRCUITS LOGIQUES 7

2.1 TRANSFORMER UNE EXPRESSION LOGIQUE EN CIRCUIT 7
2.2 SIMPLIFICATION DES EXPRESSIONS LOGIQUES 10

3 UNITE ARITHMETIQUE ET LOGIQUE (UAL) 13

3.1 ADDITIONNEUR 13
3.2 UNITE ARITHMETIQUE ET LOGIQUE 15

4 CIRCUITS SEQUENTIELS 17

4.1 NOTIONS DE TEMPS ET DE MEMORISATION 17
4.2 LATCHS ET FLIP-FLOPS (BASCULES) 20

5 CONTROLE 23

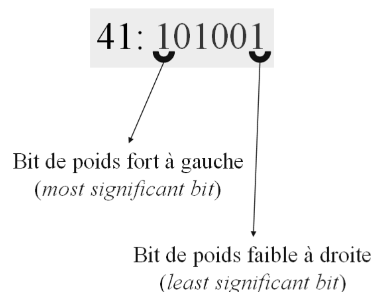
5.1 LE MULTIPLIEUR SEQUENTIEL 24
5.2 CONVERSION D'UN ORGANIGRAMME EN CIRCUIT DE CONTROLE 25
5.3 METHODE SYSTEMATIQUE DE CONCEPTION D'UN CIRCUIT SEQUENTIEL 27

6 LES CHEMINS DE DONNEES 30

BIBLIOGRAPHIE 31

1 Représentation des Nombres

Dans un circuit, on dispose de deux niveaux de voltage (parfois notés V_{SS} et V_{DD}) pour représenter toute information, qu'elle soit de nature logique ou numérique. On représente donc les nombres en base 2 en associant par exemple la valeur binaire 0 au voltage V_{SS}, et la valeur binaire 1 au voltage V_{DD}. En informatique, un chiffre binaire est appelé un **bit** (**bi**nary *digit*) ; le bit le plus à gauche d'un nombre binaire est appelé **bit de poids fort**, et le bit le plus à droite est appelé **bit de poids faible**.



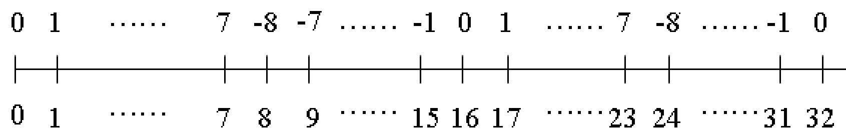
Les circuits arithmétiques opèrent sur des nombres binaires, et la taille des circuits est généralement corrélée à la taille des nombres. La taille des circuits étant bornée par des contraintes matérielles, il est également nécessaire de restreindre la taille des nombres qui peuvent être traités par un opérateur arithmétique. Lorsqu'un processeur comporte des opérateurs qui ne peuvent effectuer des calculs que sur des nombres d'au plus N bits, on dit que le processeur a un **mot** de taille N ou qu'il s'agit d'un processeur N-bits . Le premier microprocesseur (Intel 4004 [1]) était un processeur 4-bits, le Pentium 4 [2] est un processeur 32-bits, et l'Itanium [3] (dernier processeur d'Intel destiné au calcul haute-performance) est un processeur 64-bits.

1.1 Représentation des nombres entiers

La représentation binaire des entiers naturels ne pose pas de difficulté, en revanche, celle des entiers relatifs est plus délicate puisqu'il faut pouvoir représenter et manipuler le signe. La difficulté est de trouver une représentation « transparente », particulièrement pour l'opération la plus fréquente, l'addition. Pour l'addition, par exemple, une représentation est transparente si une opération peut être conduite de la même façon quel que soit le signe des opérandes. Si la représentation n'est pas transparente, il sera nécessaire de détecter le signe des opérandes, puis d'utiliser une addition si les signes sont identiques, ou une soustraction s'ils sont différents. De telles manipulations peuvent avoir un impact significatif sur les performances.

Complément à 2. Le codage utilisé dans la quasi-totalité des processeurs est la représentation en **complément à 2**. Ce codage définit une convention pour la représentation d'un nombre fini d'entiers relatifs à l'aide d'entiers naturels. Dans un processeur fonctionnant avec des mots de *n* bits, on va pouvoir représenter l'intervalle de valeurs $[-2^{n-1};2^{n-1}-1]$ en utilisant l'intervalle $[0;2^n-1]$. Les valeurs 0 à $2^{n-1}-1$ sont représentées directement par les entiers 0 à $2^{n-1}-1$, tandis que les valeurs -2^{n-1} à -1 sont représentées par les entiers 2^{n-1} à 2^n-1. Avec ce codage, on peut remarquer que X, $-2^{n-1} \leq X \leq -1$ est codé par 2^n+X.

Exemple : n=4 bits.



Le principe de la représentation en complément à 2 est de rendre ce codage transparent pour les opérations d'addition en le répétant sur l'intervalle $[2^n ;2^{n+1}-1]$ et en effectuant tous les calculs modulo 2^n. Appliquer modulo 2^n au résultat est une opération transparente en représentation binaire, car elle consiste simplement à conserver les n bits de poids faible. Plusieurs cas peuvent alors se produire lors d'une addition $x + y$, où x et y sont codés en complément à 2 par X et Y :

- $x \geq 0, y \geq 0$ et $x + y < 2^{n-1}$. Il n'y a pas de dépassement de capacité, c'est-à-dire que le résultat peut être codé sur n bits en complément à 2. Le résultat est directement donné par $x + y$.
- $x \geq 0, y \geq 0$ et $x + y \geq 2^{n-1}$. Il y a dépassement de capacité. On peut remarquer que dans ce cas $2^n-1 \geq X + Y \geq 2^{n-1}$, et donc que le résultat correspond au codage d'un nombre négatif.
- $x \geq 0, y \leq 0$ et $x + y \geq 0$. Dans ce cas, l'opération effectuée après codage en complément à 2 est $X + Y = x + 2^n + y$, et le résultat est supérieur à 2^n. En appliquant ensuite le modulo 2^n, on obtient $x + y$. Il ne peut y avoir de dépassement de capacité.
- $x \geq 0, y \leq 0$ et $x + y < 0$. On effectue la même opération, mais le résultat reste inférieur à 2^n. Le résultat $2^n + y + x$ correspond alors au codage du nombre négatif $x + y$. Il ne peut y avoir de dépassement de capacité.
- $x < 0, y < 0$ et $x + y \geq -2^{n-1}$. Il n'y a pas de dépassement de capacité. Le résultat de l'opération en complément à 2 est $2^n + x + 2^n + y$. Comme $0 > x + y \geq -2^{n-1}$, $2^n > 2^n + x + y \geq 2^{n-1}$, et après application du modulo 2^n, le résultat est $2^n + x + y$.
- $x < 0, y < 0$ et $x + y < -2^{n-1}$. Il y a dépassement de capacité. Comme $2^n + x \geq 2^{n-1}$ et $2^n + y \geq 2^{n-1}$ (codage d'un nombre négatif), on peut remarquer que $2^n + 2^{n-1} > 2^n + x + 2^n + y \geq 2^n$, ce qui correspond au codage d'un nombre positif.

Exemple : n=4 bits.

	3		0 0 1 1		*3*		0 0 1 1		*-3*		1 1 0 1
+	*2*		0 0 1 0	+	*-2*		1 1 1 0	+	*-6*		1 0 1 0
(5)	*5*		**0 1 0 1**	**(5)**	*5*	1	**0 0 0 1**	**(7)**	*-9*	1	**0 1 1 1**

Dépassement de capacité

On peut enfin remarquer que le codage des entiers relatifs en complément à 2 est tel que, pour les nombres positifs le bit de poids fort est toujours à 0 (car $x < 2^{n-1}$), et pour les nombres négatifs le bit de poids fort est toujours à 1 (car $x \geq 2^{n-1}$). On peut donc rapidement déterminer le signe d'un nombre à partir de son bit de poids fort.
Enfin, en représentation en complément à 2, l'opposé d'un nombre, $-x$, peut être rapidement calculé avec l'expression $X' + 1$ modulo 2^n, où X' est le complément logique de X.

1.2 Représentation des nombres réels

Le codage en complément à deux sur *n* bits ne permet de représenter qu'un intervalle de 2n valeurs. Pour un grand nombre d'applications, cet intervalle de valeurs est trop restreint. La représentation à virgule flottante (*floating-point*) a été introduite pour répondre à ce besoin. Pour des mots de 32 bits, la représentation en complément à deux permet de coder un intervalle de 2^{32} valeurs tandis que la représentation à virgule flottante permet de coder un intervalle d'environ 2^{255} valeurs.

La représentation en virgule flottante a été normalisée (norme IEEE 754 [7]) afin que les programmes aient un comportement identique d'une machine à l'autre. Plus précisément, les buts de cette norme sont :

- de définir la représentation des nombres,
- de définir le comportement en cas de dépassement de capacité (exceptions),
- de garantir un arrondi exact pour les opérations élémentaires (+,-,*,/,√).

Représentation des nombres. La représentation des nombres en 32 et 64 bits (taille des mots dans les processeurs actuels) est indiquée dans la figure et la table ci-dessous.

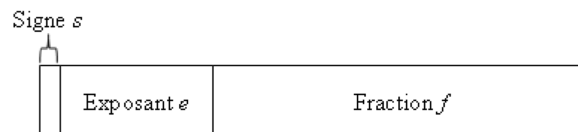


Figure 1. Représentation des nombres à virgule flottante dans la norme IEEE 754

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

Dans cette représentation, la valeur d'un nombre sur 32 bits est donnée par l'expression $(-1)^s \times \left(1+\sum_{i=1}^{23} f_i 2^{-i}\right) \times 2^{e-E_{max}}$ où f_i correspond au $i^{\text{ème}}$ bit de la fraction *f*.

Exemple : *n*=32 bits.

A= 1 10000010 00110000000000000000000

s= 1 (-) , *e*=130-127=3, *f*=$2^{-3}+2^{-4}$=0,125+0,0625

A= -1,1875 x 2^3 = -9,5

Contrairement à la représentation en complément à 2, il n'est pas possible d'effectuer des opérations d'addition (ni d'autres opérations) de façon transparente. En effet, pour effectuer une addition il faut interpréter le signe, aligner les exposants, appliquer l'arrondi choisi puis normaliser le résultat (voir plus loin). En revanche, cette représentation facilite les comparaisons, les champs étant placés dans l'ordre lexicographique (signe, puis exposant, puis fraction/mantisse). C'est d'ailleurs pour cette raison que l'exposant n'est pas signé, la valeur réelle de l'exposant étant obtenue après soustraction de E_{min}.

Exceptions. Afin d'identifier des valeurs extrêmes (valeurs proches de 0, ou ∞) ainsi que des expressions indécidables ($\sqrt{-1}, \frac{0}{0}, \infty - \infty$), la norme introduit des conventions particulières :

e- E_{max}	*f*	Nombre représenté	Cas
E_{min}-1	0	±0	Zéro
E_{min}-1	≠0	$(-1)^s \times 0.f \times 2^{E\min}$	Nombres dénormalisés (petites valeurs)
$E_{min} \leq \; \leq E_{max}$		$(-1)^s \times 1.f \times 2^{e-E\max}$	Nombres normalisés
E_{max}+1	0	±∞	Infini
E_{max}+1	≠0	*NaN*	Expressions indécidables

L'introduction de ces valeurs dans la norme permet de définir une arithmétique étendue :

$$\frac{1}{0^+} = +\infty$$
$$\frac{1}{-\infty} = 0^-$$
$$(+\infty) + (+\infty) = +\infty$$
$$+\infty \times (-\infty) = -\infty$$
$$+\infty - \infty = NaN$$
$$\frac{0}{0} = NaN$$
$$0 \times \infty = NaN$$
$$F(..., NaN, ...) = NaN$$
$$\text{Dépassement de capacité} \rightarrow NaN$$

Ces valeurs sont en général associées à des exceptions : *underflow* (0, $\pm x_{min}$ normalisé, nombres dénormalisés), *overflow* ($\pm\infty, \pm x_{max}$), division par 0 (±∞), résultat *NaN* avec des opérandes ≠*NaN*, et résultat inexact en cas d'application d'arrondi (la plupart des calculs). La norme permet de choisir entre plusieurs comportements en cas d'exception : (1) interrompre le programme, (2) gestion de l'erreur dans le programme, (3) continuer le calcul.

Arrondi. La norme propose quatre types d'arrondi : vers 0, vers +∞, vers -∞, au plus près. En théorie, l'utilisateur a la possibilité de sélectionner le mode d'arrondi, mais en pratique ce choix est souvent imposé dans le langage de programmation.

Exemple : Addition avec arrondi au plus près.

On veut additionner les deux nombres suivants (norme IEEE 754):
R_1 = 0 00000010 0010…10
R_2 = 0 00000010 1010…00
On effectue d'abord l'addition des deux mantisses normalisées 1,0010…01 avec 1,1010…00 et le résultat est : 10,1100…01. Comme la partie entière contient 2 bits (10,…), il faut normaliser en décalant à droite d'un bit (1,01100…01). Le 23ème bit de la mantisse correspond donc au dernier 0 (1,01100…01). Comme le préconise la norme IEEE 754, on a effectué le calcul sur plus de 23 bits, et pour ramener la mantisse à 23 bits, on doit appliquer la politique d'arrondi. On suppose quel'on utilise ici l'arrondi par excès, et donc la présence d'un 1 dans le 24e bit implique que le 23e bit doit devenir un 1. Le résultat final est : 0 00000010 01100…01.
en raison de l'arrondi, il y a eu une perte de précision.

La norme garantit un arrondi exact sur les opérations élémentaires (+,-,×,/,√). Cela signifie que le résultat fourni est le même que si l'opération avait été effectuée avec une précision infinie. Pour ce faire, les circuits compatibles avec la norme IEEE 754 effectuent des calculs en précision étendue, ce degré de précision n'étant ni visible ni accessible à l'utilisateur. Ainsi, en 32 bits, les opérations doivent être effectuées avec $e \geq 11$, $f \geq 32$ (soit sur au moins 6 octets), et en 64 bits avec $e \geq 64$, $f \geq 15$ (soit sur au moins 10 octets). Pour le moment, l'arrondi exact n'est pas encore garanti pour les autres fonctions élémentaires (*sin*, *cos*,…) ce qui peut engendrer des différences importantes de comportement d'une machine à l'autre.
En outre, bien que l'arrondi soit exact, une mauvaise utilisation des calculs en virgule flottante peut engendrer des erreurs numériques très importantes qui peuvent avoir un impact significatif sur les résultats.

Exemple : L'addition n'est pas associative ; *e→3 bits, f→4 bits.*

$$(-1{,}0111 \times 2^4 + 1{,}0111 \times 2^4) + 1{,}1000 \times 2^{-3} = 1{,}1000 \times 2^{-3}$$
$$-1{,}0111 \times 2^4 + (1{,}0111 \times 2^4 + 1{,}1000 \times 2^{-3}) = -1{,}0111 \times 2^4 + 1{,}0111 \times 2^4 = 0$$

2 Circuits logiques

2.1 Transformer une expression logique en circuit

La représentation de l'information logique dans un ordinateur repose sur l'**Algèbre de Boole**. Il existe une relation directe entre les expressions logiques utilisant l'algèbre de Boole et les circuits électroniques. Notamment, la complexité d'une expression logique est corrélée à la complexité et donc la taille du circuit correspondant. Aussi, simplifier les expressions logiques décrivant la tâche à implémenter permet de réduire le coût du circuit associé. Bien que le propos du cours ne soit pas d'étudier de façon théorique l'Algèbre de Boole, nous en rappelons ici les axiomes et les principaux théorèmes qui sont utilisés comme des règles de manipulation et de simplification des expressions logiques.

Axiomes

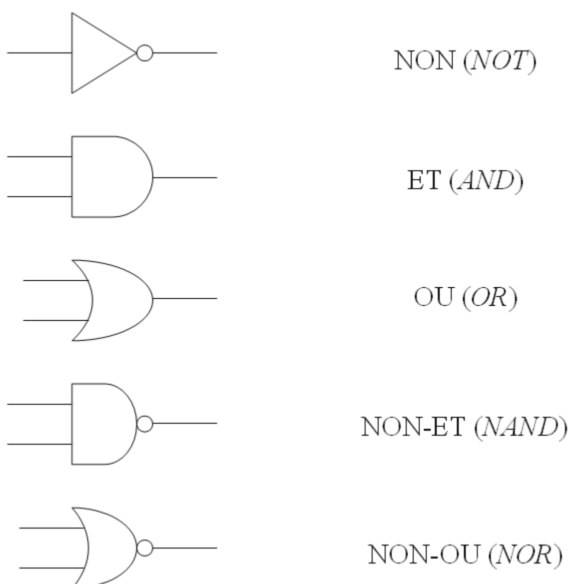
- commutativité: a ET b=b ET a, a OU b=b OU a
- associativité: a ET (b ET c)=(a ET b) ET c, a OU (b OU c)=(a OU b) OU c
- distributivité: a ET (b OU c)=a ET b OU a ET c, a OU (b ET c)=(a OU b) ET (a OU c)
- élément neutre: VRAI ET a=a ET VRAI=a, FAUX OU a=a OU FAUX=a
- complément: a ET a'=FAUX, a OU a'=VRAI

Théorèmes

- élément absorbant: a ET FAUX=FAUX, a OU VRAI=VRAI
- absorption: a ET (a OU b)=a, a OU (a ET b)=a
- idempotence: a ET a=a, a OU a=a
- involution: (a')'=a
- De Morgan: (a ET b)'=a' OU b', (a OU b)'=a' ET b'

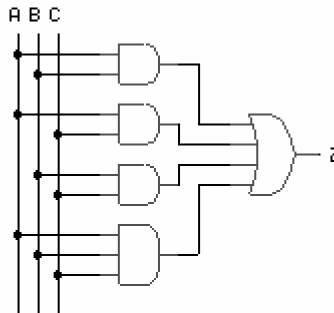
Outre l'Algèbre de Boole, les résultats de Shannon [4] ont permis de passer d'une représentation théorique de l'information logique à son implémentation sous forme de circuits électroniques. Shannon a notamment montré que toute fonction logique peut être exprimée à l'aide des seuls opérateurs ET et NON, ce qui signifie qu'il suffit de disposer de très peu de circuits logiques élémentaires différents pour implémenter toute fonction logique. Dans ce cours, nous allons utiliser les opérateurs élémentaires ET, OU et NON pour des questions de lisibilité des expressions logiques ; en pratique, on utilise surtout les opérateurs NON-ET et NON-OU, dans les circuits.

Notations. De même que l'information numérique n'est représentée qu'avec les deux valeurs binaires 0 et 1 correspondant aux deux niveaux de voltage utilisés dans les transistors, l'information logique est également codée avec ces deux valeurs 0 et 1. 1 est associé à VRAI, et 0 est associé à FAUX.
L'opérateur ET est généralement représenté par «x» ou par «.» ; l'opérateur OU est représenté par «+» ; l'opérateur NON est représenté par «'» (X' pour NON(X)) ou par « ¯ » ($\overline{X}$). Les schémas associés aux principales portes logiques sont représentés ci-dessous. Un rond placé avant une entrée ou une sortie correspond à une négation logique (NON) sur l'entrée ou la sortie.



Exemple. Utilisation de l'algèbre de Boole pour réaliser un circuit combinatoire.

On dispose de trois lampes, et on veut réaliser un circuit qui envoie un signal si au moins deux des lampes sont allumées. On appelle A, B, C les trois lampes avec la convention 0=allumée, 1=éteinte, et Z la sortie du circuit avec la convention 0=pas de signal, 1=signal activé. Le signal doit être activé dans plusieurs cas : si deux des lampes sont allumées (n'importe quelle combinaison de deux lampes), ou si les trois lampes sont allumées. On en déduit que Z=1 si ((A ET B sont allumées) OU (A ET C sont allumées) OU (B ET C sont allumées) OU (A ET B ET C sont allumées)) et Z=0 sinon. On peut traduire cette proposition en l'expression logique suivante: Z = A.B + A.C + B.C + A.B.C. Le circuit correspondant est représenté ci-dessous.



Tables de vérité. Un circuit est une fonction logique des variables d'entrée. Comme chaque variable ne peut prendre que deux valeurs, le nombre de valeurs de la fonction est fini, et on peut donc caractériser une fonction par l'ensemble de ses valeurs. Aussi, lorsque l'on veut réaliser un circuit logique, on commence par déterminer sa **table de vérité**. Cette table indique, pour chaque combinaison des valeurs d'entrée, la valeur de sortie. Par convention, la première ligne correspond au cas où toutes les variables d'entrée ont la valeur 0, et les lignes suivantes sont obtenues en ajoutant 1.
Généralement, on caractérise une fonction logique par les cas où elle vaut 1. À chaque ligne de la table de vérité correspond une expression logique, et l'expression logique de la sortie est simplement le OU des cas (des lignes) où la sortie vaut 1.

Exemple.

La table de vérité de l'exemple ci-dessus est la suivante :

A	B	C	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 exemple, la quatrième ligne de la table de vérité peut être lue ainsi : Z vaut 1 si A=0 ET B=1 ET C=1. L'expression logique associée à A=0 est A', et l'expression logique associée à B=1 est B. On en déduit que l'expression logique associée à cette ligne est A'.B.C. En

l'expression de Z est : Z = A'.B.C + A.B'.C + A.B.C' + A.B.C.

2.2 Simplification des expressions logiques

On peut remarquer que l'expression obtenue dans l'exemple ci-dessus est plus complexe que celle obtenue dans le premier exemple, bien que la fonction réalisée soit exactement la même. Le circuit correspondant à la deuxième expression sera plus coûteux que celui correspondant à la première expression. Pour minimiser le coût d'un circuit, il faut donc simplifier autant que possible l'expression logique associée. Pour ce faire, on peut utiliser les axiomes et théorèmes de l'Algèbre de Boole.

Exemple.

$$
\begin{aligned}
Z &= A'.B.C + A.B'.C + A.B.C' + A.B.C \\
&= A'.B.C + A.B'.C + A.B.C' + A.B.C + A.B.C + A.B.C \\
&= (A'.B.C + A.B.C) + (A.B'.C + A.B.C) + (A.B.C' + A.B.C) \\
&= B.C + A.C + A.B
\end{aligned}
$$

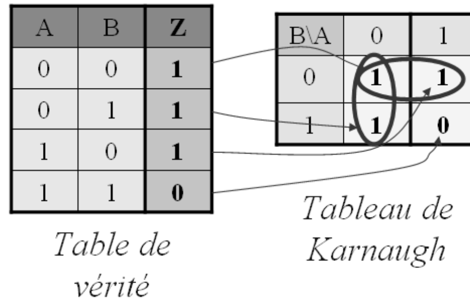
Pour ces simplifications, on a utilisé les propriétés X = X + X et X.Y' + X.Y = X. On remarque que si l'on ajoute quatre termes A.B.C au lieu de trois, on retrouve la première expression :

$$Z = B.C + A.C + A.B + A.B.C$$

Lorsque l'on utilise directement les axiomes et théorèmes de l'Algèbre de Boole pour simplifier une expression logique, il peut être difficile de savoir si l'on a obtenu une expression minimale. Il existe des méthodes plus systématiques pour minimiser les expressions logiques. Pour les fonctions de peu de variables, une méthode à la fois efficace et intuitive est celle des tableaux de Karnaugh. Le principe des tableaux de Karnaugh est de disposer de manière différente les termes de la table de vérité. Cette nouvelle disposition va permettre de réaliser aisément des simplifications. Considérons la table de vérité ci-dessous d'une fonction de deux variables A et B.

A	B	Z
0	0	1
0	1	1
1	0	1
1	1	0

En lisant la table de vérité, on obtient Z = A'.B' + A'.B + A.B'. On remarque que l'on peut utiliser deux fois le terme A'.B' pour réaliser des simplifications du type X.Y + X.Y' = X, et on obtient donc Z = A' + B'. Considérons maintenant le tableau de Karnaugh des deux variables A et B.



A	B	Z
0	0	1
0	1	1
1	0	1
1	1	0

B\A	0	1
0	1	1
1	1	0

On remarque que chacune des cases du tableau correspond à une ligne de la table de vérité. De plus, on peut observer que des cases adjacentes correspondent toujours à des expressions du type X.Y et X.Y'. C'est-à-dire que les expressions de deux cases adjacentes ne diffèrent que d'une variable. Or, si une case contient un 1, la sous-expression correspondante se retrouvera dans l'expression de la fonction de sortie (Z). Par conséquent, si deux cases adjacentes contiennent des 1, les deux sous-expressions correspondantes seront combinées en utilisant la règle X.Y + X.Y' = X, c'est-à-dire en éliminant la seule variable qui les distingue. Le principe du tableau de Karnaugh est d'exprimer intuitivement cette simplification en réunissant les deux cases à 1 dans un « paquet ». On peut remarquer que cette règle d'adjacence reste vraie pour les paquets : les sous-expression associées à deux paquets adjacents ne diffèrent que d'une variable, et peuvent être également combinées en utilisant la règle X.Y + X.Y' = X.

Exemple.

B\A	0	1
0	1	1
1	1	1

Dans le tableau de Karnaugh ci-dessus, on peut constituer deux paquets de deux termes, dont les expressions sont respectivement B' et B. Ces deux paquets étant adjacents, on peut à nouveau les combiner en un paquet de quatre termes dont l'expression est 1.

La taille des paquets varie donc de 1 à 2^n, où *n* est le nombre de variables. Dans une table de vérité, l'expression finale est obtenue par le OU des différentes cases à 1, tandis que dans un tableau de Karnaugh, l'expression finale est obtenue par le OU des différents paquets, l'ensemble des paquets devant recouvrir tous les termes à 1 du tableau.
Dans le tableau ci-dessus, on peut former deux paquets dont les expressions sont *A'* et *B'*. Comme ces deux paquets ne sont pas adjacents, on ne peut les combiner à nouveau entre eux. L'expression finale est Z= A' + B'. Implicitement, le tableau de Karnaugh repose sur deux règles : le dédoublement des expressions logiques (X = X + X), et la règle X.Y + X.Y' = X.

On peut également utiliser des tableaux de 3, 4 (et plus) variables. Le principe est identique, il faut seulement s'assurer que la règle d'adjacence des cases du tableau est respectée. Pour ce faire, il est nécessaire d'inverser deux lignes dans le tableau de 3 variables, et deux lignes et deux colonnes dans le tableau de 4 variables (voir ci-dessous) afin que les expressions logiques associées à n'importe quelle paire de cases adjacentes ne diffèrent que d'une variable.

Exemple. Le tableau de Karnaugh correspondant à l'exemple des trois lampes est le suivant :

BC\A	0	1
00	0	0
01	0	1
11	1	1
10	0	1

B.C
A.C
A.B

Les lignes 10 et 11 ont été inversées pour respecter la règle d'adjacence.
Le recouvrement des termes à 1 est indiqué ci-dessus, et l'expression correspondante est la suivante : Z = A.B + A.C + B.C

Un recouvrement engendre une expression logique minimale si le nombre de paquets et la taille des paquets sont les plus faibles possibles. Bien qu'il existe des méthodes systématiques de recouvrement, on se contentera ici d'utiliser l'heuristique simple suivante :

Choisir un terme à 1 et le recouvrir par le plus grand paquet possible.
Choisir un terme à 1 non encore recouvert, et le recouvrir par le paquet qui le contienne ainsi que le maximum de termes à 1 non encore recouverts.
Itérer à partir de l'étape 2 jusqu'à recouvrement de tous les termes à 1.

Valeurs non assignées. Il arrive parfois que la valeur de sortie de la fonction, pour certaines combinaisons des valeurs d'entrée, ne soit pas spécifiée dans la description du circuit. Plutôt que de placer un 1 ou un 0 dans la table de vérité, on utilise un terme appelé «*don't care*», et noté «*d*» ou «X», qui peut être remplacé indifféremment par un 1 ou un 0. Par exemple, dans la méthode du tableau de Karnaugh, si le fait de considérer ce terme comme un 1 permet de créer des paquets plus grands, et donc des expressions logiques plus simples, on le remplace par un 1, et si ce terme n'appartient à aucun paquet après recouvrement, on le remplace par un 0.

Exemple.

On reprend l'exemple précédent en supposant que la lampe A ne peut jamais être allumée seule ; dans la ligne A=1, B=0, C=0 de la table de vérité, Z vaut alors «d».

A	B	C	Z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	*d*
1	0	1	1
1	1	0	1
1	1	1	1

Le tableau de Karnaugh correspondant est le suivant. On voit que l'on peut créer un paquet à quatre éléments en utilisant le terme don't care.

BC\A	0	1
00	0	d
01	0	1
11	1	1
10	0	1

B.C

A

On en déduit que l'expression de Z devient : Z = A + B.C

3 Unité Arithmétique et Logique (UAL)

L'unité de calcul d'un processeur est appelée Unité Arithmétique et Logique (UAL). Elle effectue toutes les opérations élémentaires sur les entiers. Dans les premières machines,
elle constituait la partie la plus importante du processeur ; aujourd'hui, un processeur contient souvent plusieurs unités arithmétiques et logiques, et bien que leur rôle reste très important, elles ne représentent plus qu'une petite fraction de la surface totale du processeur. Dans cette section, nous allons construire une unité arithmétique et logique simple, en commençant par l'additionneur qui en est l'élément essentiel.

3.1 Additionneur

Comme on l'a vu précédemment, un processeur travaille sur un nombre fini de bits appelés un **mot**. Un processeur n-bits va donc contenir un additionneur n-bits. Cet additionneur est lui-même conçu à partir de n additionneurs 1-bit : tout comme dans une addition effectuée à la main, on additionne les chiffres deux à deux en passant la retenue au couple de chiffres suivant. On va donc commencer par concevoir un additionneur 1-bit, et on en déduira l'additionneur n-bits.

Additionneur 1-bit. Lorsque l'on additionne deux bits x et y, le résultat est la somme S des chiffres, et la retenue R. La table de vérité est la suivante :

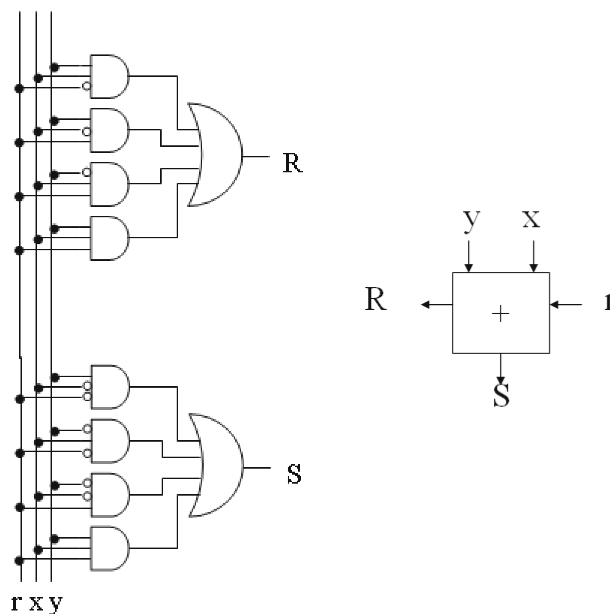
x	y	**S**	**R**
0	0	**0**	**0**
0	1	**1**	**0**
1	0	**1**	**0**
1	1	**0**	**1**

On en déduit que $S = x'.y + x.y' = x \oplus y$, et que $R = x.y$.

Comme ce circuit ne prend pas en compte la retenue d'entrée correspondant au chiffre précédent, on appelle ce circuit un demi-additionneur. L'additionneur 1-bit complet prend donc en compte la retenue d'entrée r ; la table de vérité et le schéma du circuit sont indiqués ci-dessous.

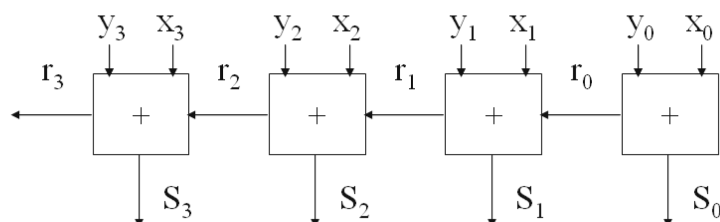
r	x	y	S	R
0	0	0	**0**	**0**
0	0	1	**1**	**0**
0	1	0	**1**	**0**
0	1	1	**0**	**1**
1	0	0	**1**	**0**
1	0	1	**0**	**1**
1	1	0	**0**	**1**
1	1	1	**1**	**1**

$S = r'.(x'.y + x.y') + r.(x.y + x'.y') = r \oplus x \oplus y$ et $R = r'.x.y + r.x'.y + r.x.y' + r.x.y = x.y + r.(x + y)$.



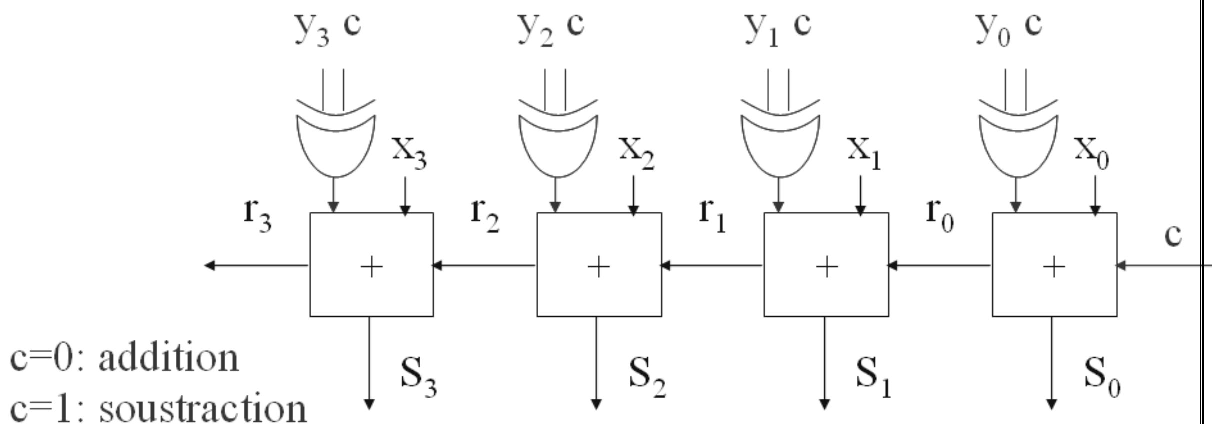
Additionneur *n*-bits. Pour effectuer l'addition de deux nombres de *n* bits, il suffit de chaîner entre eux *n* additionneurs 1-bit complets. La retenue est ainsi propagée d'un additionneur à l'autre. Un tel additionneur est appelé un additionneur série. Bien que tous les chiffres des deux nombres de *n*-bits X et Y soient disponibles simultanément au début du calcul, à t=0, le temps de calcul est déterminé par la propagation de la retenue à travers les *n* additionneurs 1-bit.

<u>Exemple :</u> Additionneur 4-bits.



Soustracteur *n*-bits. Il n'y a pas de circuit soustracteur dans un processeur parce que l'on peut implémenter la soustraction à l'aide de l'additionneur avec des modifications mineures. Pour ce faire, on exploite les propriétés du complément à 2 et le fait que le bit de poids faible de l'additionneur n'a pas de retenue d'entrée. En effet, effectuer $X - Y$ en complément à 2, est équivalent à $X + Y' + 1$. Pour effectuer la deuxième addition (+1), il suffit d'injecter un 1 en guise de retenue dans l'additionneur de poids faible. On peut donc supposer que l'on dispose d'un signal de contrôle *c* qui vaut 0 lorsque l'on veut faire une addition, et 1 lorsque l'on veut faire une soustraction. On utilise ce signal *c* comme retenue du bit de poids faible de l'additionneur. Enfin, pour obtenir *Y'*, il suffit de rajouter un inverseur (une porte XOR) en entrée de chacun des additionneurs 1-bit : $y_i \oplus c$; lorsque *c* vaut 0, la valeur d'entrée de l'additionneur *i* est y_i, et lorsque *c* vaut 1, la valeur d'entrée est y_i'. Donc, lorsque *c* vaut 0, l'opération effectuée par le circuit est $X + Y$, et lorsque *c* vaut 1, l'opération effectuée est $X + Y' + 1$.

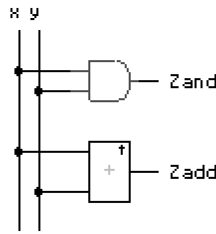
Exemple : Soustracteur 4-bits.



3.2 Unité Arithmétique et Logique

Le nombre d'opérations élémentaires arithmétiques et logiques implémentées dans l'UAL d'un processeur et la façon dont elles y sont implémentées varient considérablement d'un processeur à l'autre. Ici, nous nous contenterons d'implémenter les trois opérations essentielles : l'addition, le ET logique, le NON logique. Le ET et le NON permettent de créer n'importe quelle fonction logique, ainsi que de calculer l'opposé d'un nombre au sens du complément à 2 et donc d'effectuer des soustractions.
Tout comme l'additionneur, on commence par concevoir une UAL 1-bit, et on en déduit une UAL *n*-bits.

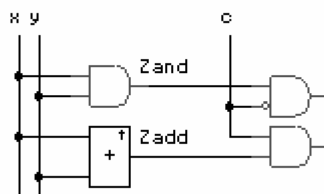
UAL 1-bit. Le circuit de la figure ci-dessous comporte deux sorties Z_{ADD}, Z_{AND} qui correspondent à deux des trois opérations que l'on désire implémenter dans l'UAL.



Cependant, une UAL 1-bit ne doit comporter qu'une seule sortie correspondant à l'opération choisie. Il faut donc ajouter un circuit qui permette d'effectuer un choix entre ces deux sorties. Un tel circuit est appelé un multiplexeur 2x1. Ce circuit doit disposer d'un signal de commande c qui permet de spécifier quelle sortie choisir (par exemple, $c=0$, $Z= Z_{AND}$, et $c=1$, $Z= Z_{ADD}$). La table de vérité et le circuit correspondant sont indiqués ci-dessous.

c	Z_{AND}	Z_{ADD}	Z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

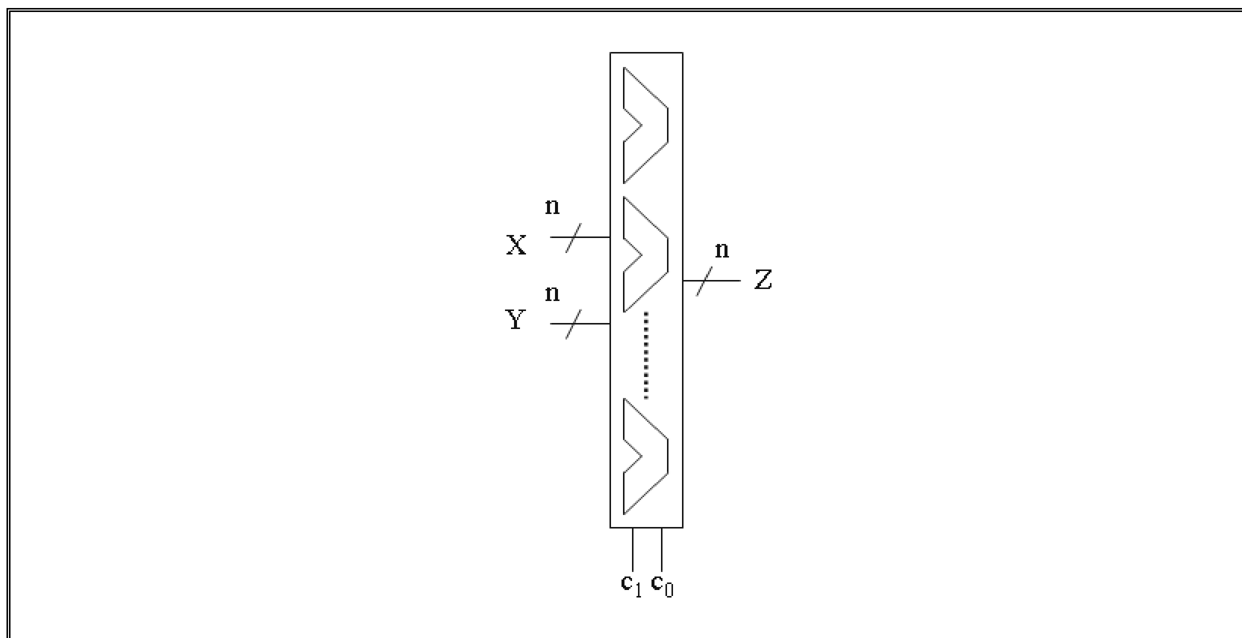
On en déduit que $Z = c.Z_{ADD} + c'.Z_{AND}$, et le circuit correspondant est indiqué ci-dessous.



L'UAL que l'on désire implémenter comporte trois sorties Z_{ADD}, Z_{AND} et Z_{NOT}. Il faudra donc utiliser un multiplexeur 3x1. Le raisonnement est identique, mais il faudra utiliser deux signaux de commande c_0 et c_1 (par exemple, $c_0=0$, $c_1=0$ pour Z_{AND}, $c_0=1$, $c_1=0$ pour Z_{ADD}, $c_0=0$, $c_1=1$ pour Z_{NOT}).

UAL *n*-bits. Tout comme l'additionneur *n*-bits, l'UAL *n*-bits est obtenu en chaînant *n* UAL 1-bits.

Exemple : *UAL 4-bits.*



4 Circuits Séquentiels

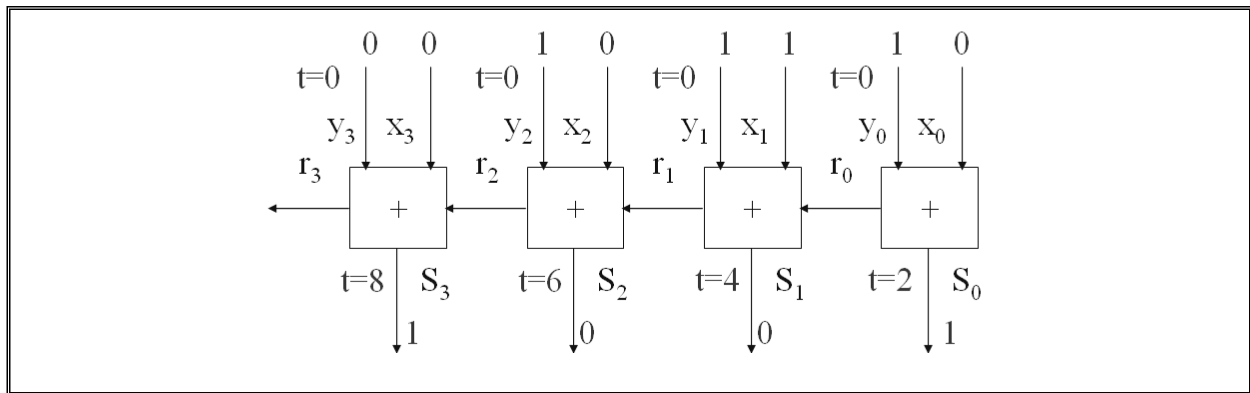
Les circuits logiques combinatoires permettent d'implémenter n'importe quelle fonction, mais ils n'intègrent ni la notion de temps ni la notion de mémorisation. Ces deux propriétés sont nécessaires au fonctionnement de plusieurs composants du processeur.

4.1 Notions de temps et de mémorisation

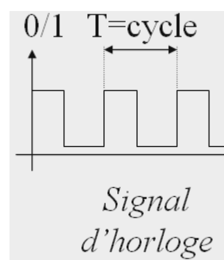
Notion de temps : l'horloge. L'exécution d'une instruction à l'intérieur d'un processeur constitue un enchaînement d'étapes : chaque composant produit un résultat qui est utilisé par le composant suivant. Il est donc nécessaire de synchroniser ces différentes étapes : si un composant C_2 utilise le signal de sortie produit par le composant C_1 d'une étape précédente, avant que le composant C_1 n'ait terminé son exécution, la valeur de ce signal risque d'être incorrecte et la valeur du signal de sortie de C_2 le sera également.

Exemple.

Considérons par exemple un additionneur 4-bits ci-dessous. On suppose que le temps de passage à travers une porte logique élémentaire (ET, OU) requiert une unité de temps, et on négligera le temps de passage à travers une porte NON. D'après la structure de l'additionneur 1-bit vu au chapitre précédent, il faut deux unités de temps pour faire passer la retenue d'un additionneur à l'autre. En conséquence, si une addition commence à t=0, le dernier additionneur dispose de toutes ses opérandes (x_3, y_3, r_2) à t=6, et la dernière sortie S_3 devient donc valide (correspond au résultat final) à t=8. Si un autre composant du processeur exploite la sortie $S = S_3\ S_2\ S_1\ S_0$ avant t=8, le résultat qu'il produira risque d'être incorrect.

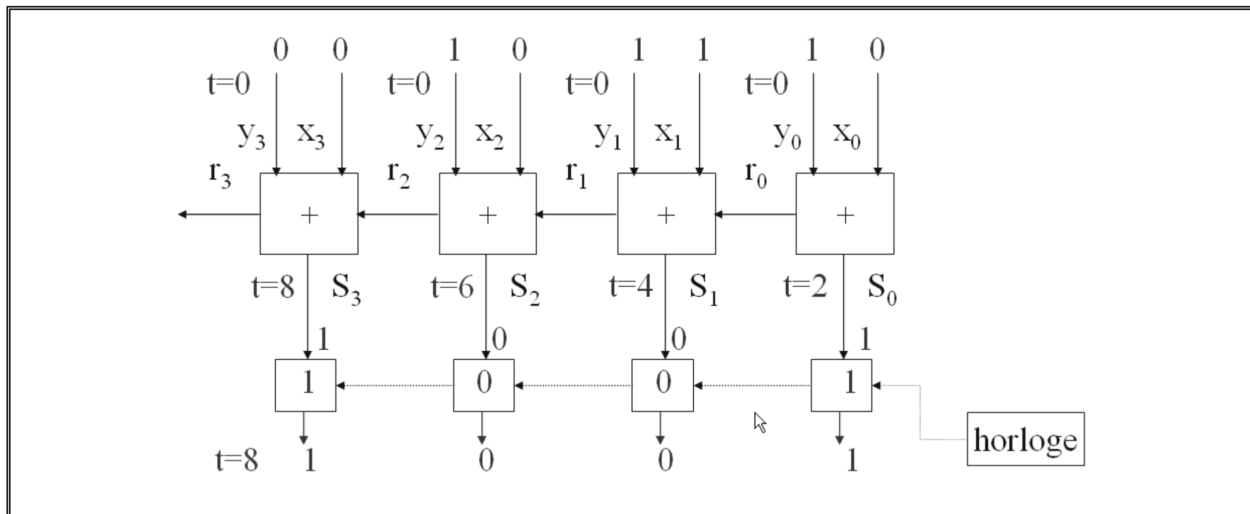


Pour pouvoir chaîner entre eux plusieurs composants, il est donc nécessaire d'estimer le temps de passage à travers chaque composant, et de bloquer la propagation du résultat vers les composants suivants tant que le calcul n'est pas terminé. Pour cela, on utilise des « barrières », des composants tampons placés en entrée et en sortie des composants de calcul, destinées à contrôler la propagation des résultats d'un composant à l'autre. Afin de simplifier la commande de ces barrières, dans la plupart des processeurs, elles s'ouvrent et se ferment à intervalle fixe. Elles sont donc commandées par un signal périodique comportant une phase haute (ouverture) et une phase basse (fermeture), appelé **une horloge**, voir ci-dessous.



Afin de simplifier la synchronisation des différents composants d'un processeur, il n'existe en général qu'une seule horloge à l'intérieur d'un processeur. La durée de la période, également appelée **un cycle**, doit être égale au plus long temps de calcul du plus lent des composants du processeur ; ce composant est en général appelé le facteur limitatif puisqu'il détermine la cadence d'exécution du processeur. Dans un Pentium 4 de fréquence 2 GHz, le temps de cycle est donc de 0.5 nanosecondes. Bien que les temps de cycle soient très bas, le mécanisme de synchronisation par horloge n'est pas toujours efficace puisque le temps de cycle est déterminé par le composant le plus lent ; il existe des travaux de recherche sur des processeurs asynchrones, donc dépourvus de commande par horloge, voir [5].

Additionneur 4-bits avec barrières en sortie.



Notion de mémorisation. Une fois qu'une barrière a été ouverte puis refermée, les valeurs de sortie produites par le composant précédent sont utilisées comme valeurs d'entrée du composant suivant. Comme la tâche effectuée par un composant peut durer jusqu'à un cycle, il faut pouvoir maintenir les valeurs en entrée du composant après la fermeture des barrières. Il est donc nécessaire de **mémoriser** l'information dans ces barrières pour la restituer après fermeture. La propriété de mémorisation d'une information peut être caractérisée de la façon suivante : un circuit a mémorisé une information si la valeur de sa sortie *Z* à l'instant *t* est égale à la valeur de *Z* à l'instant $t+\delta t$.

On dispose maintenant des informations nécessaires pour déterminer la structure d'une barrière ; elle doit avoir les propriétés suivantes :

- lorsque le signal de l'horloge $C=1$, la barrière doit être ouverte, c'est-à-dire qu'elle laisse passer en sortie (*Z*) l'information présente en entrée (*X*),
- lorsque le signal de l'horloge $C=0$, la barrière doit être fermée, c'est-à-dire qu'elle ne laisse pas passer en sortie l'information présente en entrée, et qu'elle fournit en sortie la valeur mémorisée.

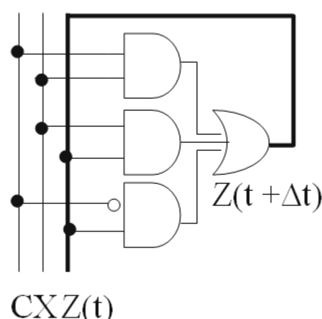
À partir de cette caractérisation du fonctionnement de la barrière, on peut déterminer sa table de vérité :

C(t)	*X(t)*	*Z(t)*	*Z(t+δt)*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$C=0$, la barrière est fermée, et la valeur en sortie est la valeur mémorisée, c'est-à-dire la valeur en sortie à l'instant précédent, $Z(t+\delta t)= Z(t)$

$C=1$, la barrière est ouverte et la valeur en sortie est celle présente en entrée.

Et on en déduit l'expression de $Z(t+\delta t) = C^{'}(t).Z(t) + X(t).Z(t) + C(t).X(t)$. Le circuit correspondant est indiqué ci-dessous. On voit donc que l'on peut implémenter un circuit de mémorisation en utilisant les mêmes portes que pour les circuits combinatoires. La mémorisation est obtenue en renvoyant en entrée le signal de sortie ; cette rétroaction est indiquée en gras dans le schéma ci-dessous.



Les circuits combinatoires intégrant une notion de temps grâce à l'horloge sont appelés **circuits séquentiels**.

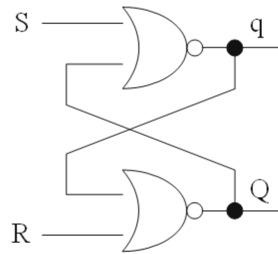
4.2 Latchs et Flip-Flops (Bascules)

Latch SR. En pratique, les circuits séquentiels sont souvent conçus à partir de quelques circuits élémentaires, tout comme pour les circuits combinatoires. Ces circuits élémentaires ont des propriétés similaires au circuit de la section précédente (notions de temps et de mémorisation). Un des principaux circuits élémentaires est le latch SR : il dispose de deux signaux de commande *S* (*Set*) et *R* (*Reset*), et d'une boucle de rétroaction permettant d'assurer la mémorisation. Le rôle des signaux de commande est décrit ci-dessous, ainsi que la table de vérité de ce latch.

S	*R*	*Comportement du latch*
0	0	Sortie inchangée (mémorisation)
0	1	Sortie = 0
1	0	Sortie = 1
1	1	Commande inutilisée

S	*R*	*Z(t)*	*Z(t+δt)*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	*d*
1	1	1	*d*

On en déduit que $Z(t+\delta t) = S + R'Z(t)$; en pratique, on implémente ce latch à l'aide de deux portes NON-OU après avoir remarqué que $Z(t+\delta t) = S + R'Z(t) = (R + (S + Z(t))')'$ si on ignore les termes *d* de la table de vérité, voir figure ci-dessous.



Latch SR

Latch D. Les circuits destinés à la mémorisation sont souvent conçus à partir de latchs D, eux-mêmes conçus à partir de latchs SR. Ces latchs ne disposent que d'un seul signal de commande, *D* (*Delay*), qui est en fait une entrée, et le rôle de ces latchs consiste simplement à mémoriser l'information fournie en entrée. Leur table de commande est indiquée ci-dessous :

D	*Comportement du latch*
0	Sortie = *D* = 0
1	Sortie = *D* = 1

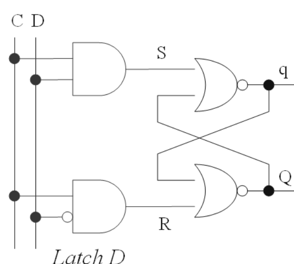
On peut modifier un latch SR pour qu'il se comporte comme un latch D, comme indiqué dans la table ci-dessous.

D	*S*	*R*	*Comportement du latch*
0	0	1	Sortie = *D* = 0
1	1	0	Sortie = *D* = 1

En général, un latch D comporte également un signal d'horloge *C* destiné à inhiber la modification de la sortie, voir ci-dessous.

C	*D*	*S*	*R*	*Comportement du latch*
0	0	0	0	Sortie inchangée (mémorisation)
0	1	0	0	Sortie inchangée (mémorisation)
1	0	0	1	Sortie = 0
1	1	1	0	Sortie = 1

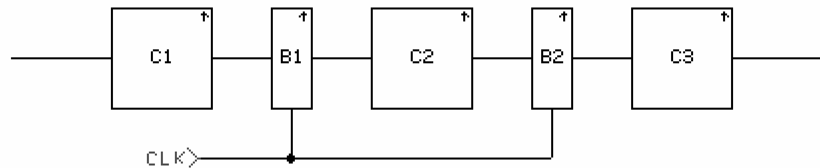
On n'en déduit que *S=C.D* et *R=C.D'*. Le circuit correspondant est le suivant :



Latch D

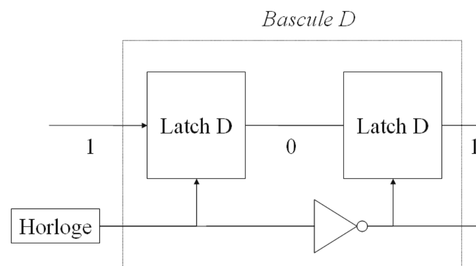
Bascules. Le latch D décrit ci-dessus peut être utilisé pour séparer deux composants, en jouant le rôle des barrières mentionnées à la Section 4.1. Cependant, ce latch ne conviendra pas pour une séquence comportant plus de deux composants. Considérons trois composants C_1, C_2, C_3 correspondant à trois étapes successives de l'exécution d'une instruction dans le

processeur ; la durée de chaque étape est d'un cycle, et la sortie du composant C_i est utilisée comme entrée du composant C_{i+1}. Ces trois composants sont séparés par deux barrières B_1, B_2, comme indiqué ci-dessous. Toutes les barrières sont cadencées par le même signal d'horloge *C* ; elles vont donc toutes s'ouvrir et se fermer en même temps. Si l'on suppose que le calcul commence à t=0 cycle dans le composant C_1, la sortie du composant C_1 est prête à t=1 cycle, celle du composant C_2 à t=2 cycles, et celle du composant C_3 à t=3 cycles.



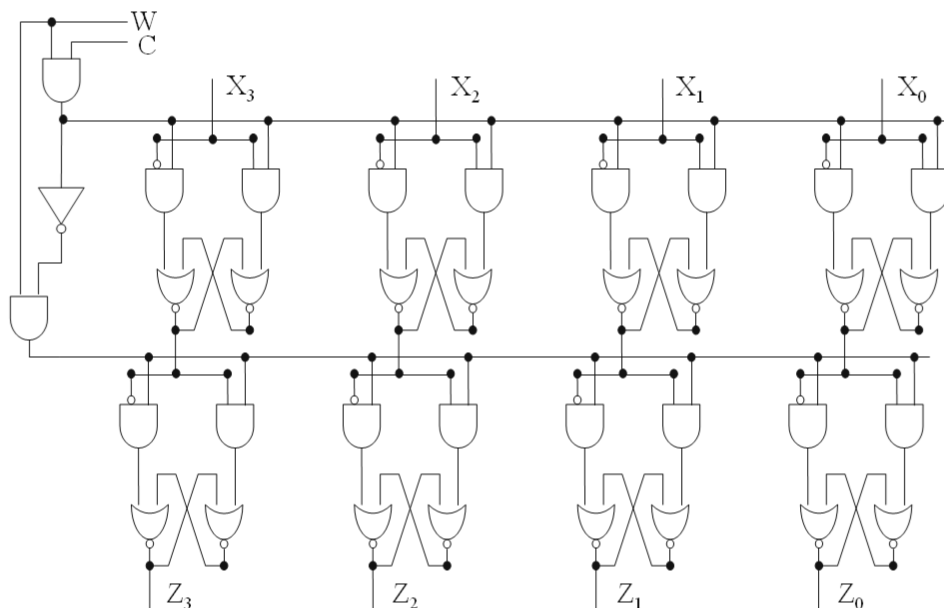
Cependant, avec un latch D nous n'obtiendrons pas le comportement désiré. Si le temps de propagation du signal à travers le circuit combinatoire C_2 est inférieur à ½ cycle, lorsque les barrières s'ouvrent, la sortie de C_1 va traverser C_2, modifier sa sortie, et donc affecter l'entrée de C_3 en ½ cycle, avant que les barrières ne se referment. Le signal peut se propager en un cycle à travers plusieurs composants parce que toutes les barrières s'ouvrent en même temps.

Pour éviter ce phénomène, les barrières doivent être constituées non pas d'un latch, mais de deux latchs commandés par des signaux d'horloge en opposition de phase. Ainsi, lorsqu'un latch est ouvert, le second est fermé et évite la propagation du signal aux composants suivants. Le circuit correspondant est indiqué ci-dessous.

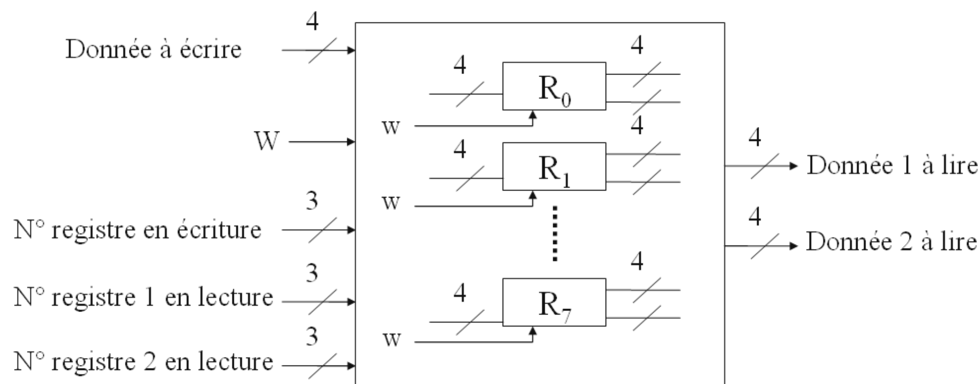


En raison du comportement alterné des latchs, ce circuit est appelé *Flip-Flop* D ou **Bascule** D. Selon les cas, les composants d'un processeur sont séparés par des latchs ou par des bascules. Par la suite, sauf indication contraire, nous supposerons que nous utilisons toujours des bascules.

Registres. Dans un processeur, lorsque l'on veut mémoriser un mot de *n*-bits, on utilise en général un registre. Un registre 1-bit est presque équivalent à une bascule avec un signal supplémentaire *W* qui permet au processeur de contrôler l'écriture dans ce registre. En l'absence de ce signal, une nouvelle donnée serait écrite dans le registre à chaque cycle, tout comme pour les bascules. Un registre *n*-bits est simplement *n* registres 1-bit contrôlés par le même signal d'écriture, comme indiqué dans la figure ci-dessous.



En général, un processeur comporte plusieurs registres regroupés dans un **banc de registres** ; ces registres sont numérotés, et le processeur spécifie quel registre il veut lire ou écrire en indiquant le numéro du registre correspondant.



Le numéro de registre peut être considéré comme une adresse, et le banc de registres comme une petite mémoire.

5 Contrôle

En général, un circuit qui nécessite plusieurs cycles pour réaliser une tâche comporte souvent deux parties : une partie « traitement » et une partie « contrôle ». Le rôle de la partie traitement est de réaliser la tâche proprement dite, par exemple un calcul ; cette partie comporte souvent des circuits combinatoires ou de mémorisation comme ceux vus dans les sections précédentes. Le rôle de la partie contrôle est d'organiser l'enchaînement de toutes les étapes intermédiaires de la tâche à effectuer, ainsi que le transfert des données entre les différentes étapes. Dans cette section, nous allons voir comment réaliser la partie contrôle d'un circuit complexe. Dans un premier temps, nous allons étudier une méthode intuitive de conception d'un circuit de contrôle, puis une méthode systématique de conception d'un circuit séquentiel que nous appliquerons ensuite à la conception de circuits de contrôle. À chaque fois, on utilisera le circuit du multiplieur séquentiel pour illustrer ces différentes méthodes.

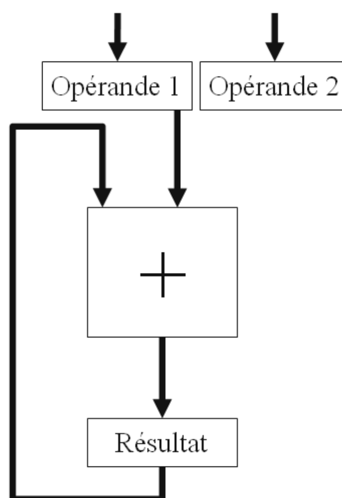
5.1 Le multiplieur séquentiel

Les premiers multiplieurs étaient séquentiels, c'est-à-dire que la multiplication était réalisée par une succession d'additions, comme pour une opération effectuée à la main, voir figure ci-dessous. La seule différence entre le circuit de multiplication et l'opération effectuée à la main est le calcul de résultats intermédiaires. Cela permet de remplacer l'addition de *n* termes par *n* additions successives de deux termes, et donc d'avoir un circuit qui ne comporte qu'un seul additionneur, voir figure ci-dessous.

```
          0 0 1 1
          0 1 1 1
    ---------------
          0 0 1 1
+       0 0 1 1
+     0 0 1 1
+   0 0 0 0
    ---------------
    0 0 1 0 1 0 1
```

```
              0 0 1 1
              0 1 1 1
    -------------------
              0 0 1 1 ─┐
+           0 0 1 1 0  │
    -------------------│
            0 1 0 0 1 ─┤
+         0 0 1 1 0 0  │
    -------------------│
          0 1 0 1 0 1 ─┤
+       0 0 0 0 0 0 0  │
    -------------------│
        0 0 1 0 1 0 1 ─┤
                       │
Résultat intermédiaire ←┘
```

On peut alors réaliser le circuit de multiplication de la façon suivante :



L'additionneur est le circuit combinatoire vu dans les sections précédentes, et *Opérande1*, *Opérande2*, *Résultat* sont des registres ; on suppose que les opérandes comportent *n* bits. L'opération de multiplication va se dérouler de la façon suivante :

1. Il y a *n* itérations.
2. A chaque itération, on doit décaler la première opérande (dans *Opérande1*) vers la gauche, puis la multiplier par un chiffre de la deuxième opérande (dans *Opérande2*).
3. On doit ensuite ajouter ce nombre au résultat intermédiaire ; comme il s'agit de chiffres binaires, on peut remarquer que le nombre à ajouter est soit la première opérande décalée, soit 0. On peut donc formuler différemment l'opération à effectuer :

soit le bit courant de la deuxième opérande vaut 0 et il est inutile de modifier le résultat intermédiaire, soit ce bit vaut 1 et il faut ajouter la première opérande décalée au résultat intermédiaire.
4. On considère le bit suivant de la deuxième opérande et on itère à l'étape 2.

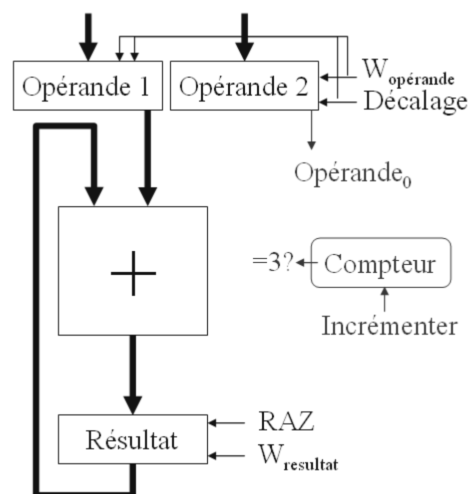
On doit ajouter aux schémas ci-dessus le circuit *Compteur* dont le but est de compter les itérations, et de signaler la fin de l'opération de multiplication. Le résultat de la multiplication de deux nombres de *n* bits peut comporter jusqu'à *2n* bits, le registre *Résultat* doit donc comporter *2n* bits. De même, comme on va décaler la première opérande *n* fois vers la gauche, *Opérande1* doit être un registre de *2n* bits. Contrairement au registre *Résultat*, ce registre doit être **à décalage** ; bien que ces registres n'aient pas été vus en cours, ils sont abordés en TD, et leur structure est très proche de celle des registres classiques. Enfin, une façon simple d'accéder au bit courant de la deuxième opérande est de décaler le registre correspondant vers la droite à chaque itération ; ainsi, le bit courant correspond toujours au bit de poids faible du registre.

Pour commander l'exécution de la multiplication séquentielle, on doit donc disposer des signaux de contrôle suivants :

- un signal *Décalage* pour commander le décalage à gauche du registre *Opérande1* et le décalage à droite du registre *Opérande2*,
- un signal *Wrésultat* pour commander l'écriture dans le registre *Résultat*,
- deux signaux utilisés pour initialiser le circuit : *RAZ* pour remettre à zéro le registre *Résultat*, et *Wopérande* pour écrire les deux opérandes dans les registres *Opérande1* et *Opérande2*,
- un signal *Incrémenter* pour modifier le compteur d'itérations.

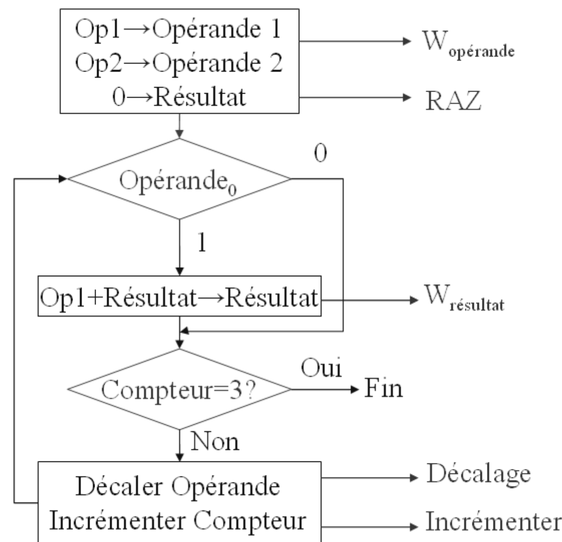
Ces signaux de contrôle sont des commandes, ils vont donc constituer les sorties d'un circuit de contrôle. Il y a deux autres signaux de contrôle qui déterminent le comportement du circuit de contrôle et constituent donc des entrées du circuit de contrôle :

- le signal *Opérande0* qui correspond au bit de poids faible du registre *Opérande2*,
- le signal *Compteur=n-1?* qui indique la fin de l'opération de multiplication.



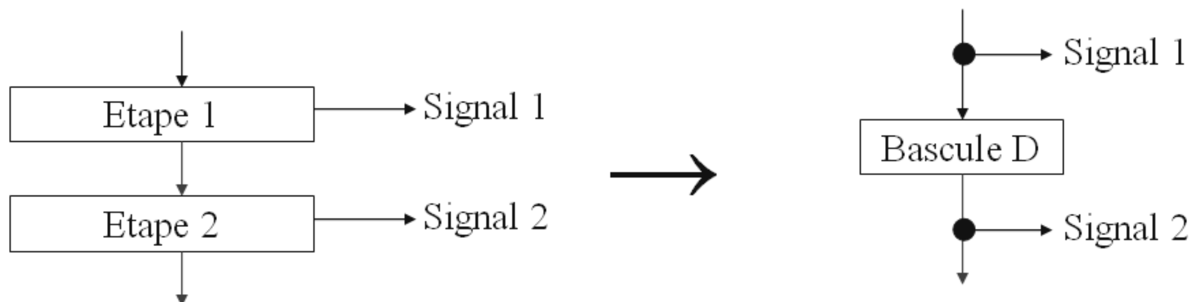
5.2 Conversion d'un organigramme en circuit de contrôle

Cette première méthode consiste à traduire directement l'organigramme du circuit de contrôle en composants combinatoires et séquentiels. L'organigramme correspondant au multiplieur séquentiel est le suivant :



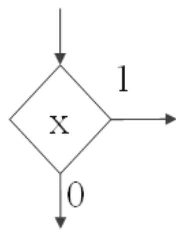
On peut remarquer que chaque élément d'un organigramme peut être traduit en un circuit combinatoire ou séquentiel.

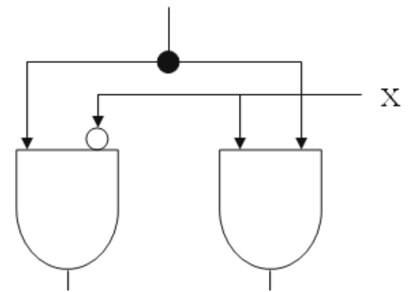
Etape. Considérons d'abord une étape simple de l'organigramme d'un circuit de contrôle. Le rôle d'une telle étape est d'activer un ou plusieurs signaux de contrôle. En termes de circuit, cela peut se traduire simplement par une connexion vers le signal à activer, voir figure ci-dessous, à droite. Le déroulement de l'organigramme d'un circuit de contrôle peut être matérialisé comme le passage d'un jeton à travers les différentes étapes de l'organigramme ; de même, l'exécution du circuit de contrôle lui-même peut être matérialisé par le passage d'un signal à 1 (le jeton) à travers les différentes parties du circuit ; au passage de ce signal, les signaux de contrôle correspondants sont activés.



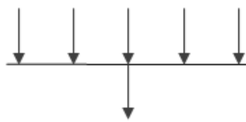
Étapes successives. Considérons maintenant deux étapes successives d'un organigramme : si elles ne sont pas réunies au sein de la même étape, c'est parce qu'elles ne doivent pas être effectuées en même temps. Au niveau du circuit, on matérialise cette propriété en séparant l'activation des signaux de contrôle de chaque étape par une bascule. De cette façon, les signaux de chaque étape seront activés à des cycles différents, voir figure ci-dessus.

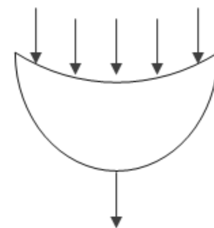
Choix. Dans un organigramme, une étape de choix permet d'orienter le jeton vers deux étapes différentes selon la valeur d'une condition. De même, dans un circuit de contrôle, on peut utiliser des portes ET pour faire passer le signal à 1 dans deux chemins différents, voir figure ci-dessous où *x* désigne le signal de condition.



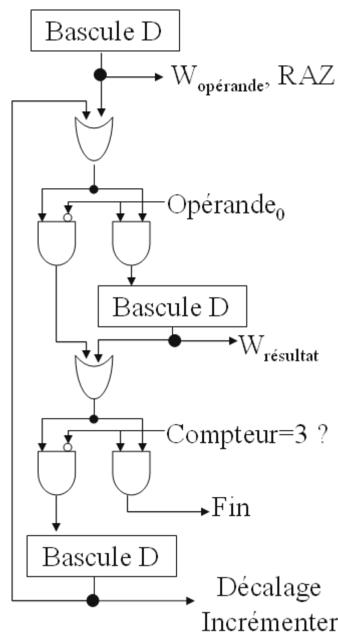


Jonction. Inversement, dans un organigramme plusieurs chemins peuvent aboutir à la même étape. De même, dans un circuit de contrôle, on peut utiliser une porte OU pour faire passer le signal à 1, pouvant provenir de plusieurs chemins différents, vers un seul chemin, voir figure ci-dessous.





En utilisant ces règles de conversion d'un organigramme en circuit, le circuit de contrôle du multiplieur séquentiel est le suivant :



5.3 Méthode systématique de conception d'un circuit séquentiel

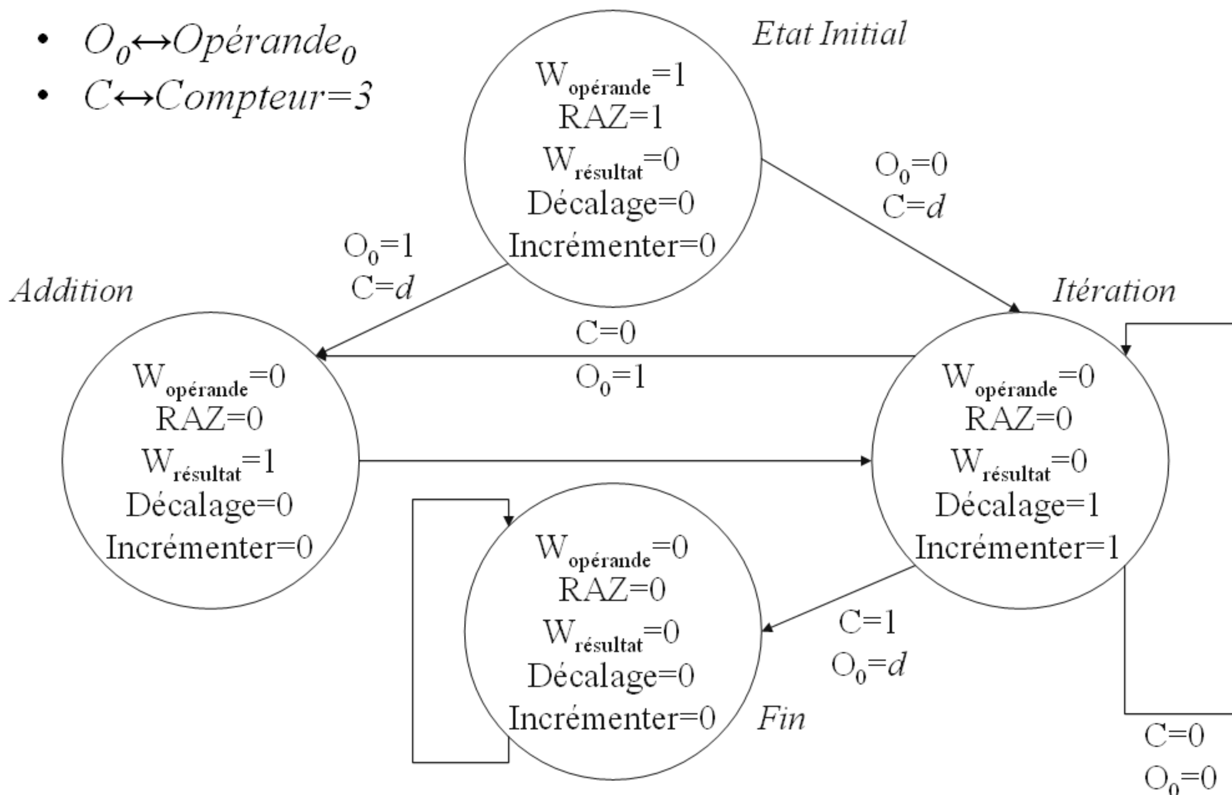
Il existe des méthodes systématiques de conception des circuits séquentiels à l'aide de bascules et de composants combinatoires. Un circuit de contrôle est simplement un type particulier de circuit séquentiel. Les étapes de conception d'un circuit séquentiel sont les suivantes :

1. Définir l'automate du circuit.

2. À partir du nombre d'états de l'automate, déterminer le nombre de bascules nécessaires ; une bascule 1-bit ayant deux états possibles (0 et 1), *n* bascules peuvent représenter et stocker 2^n états. On assigne ensuite les états aux valeurs des bascules.
3. À partir de l'automate du circuit, déterminer toutes les transitions possibles. En déduire toutes les transitions possibles des variables de contrôle des bascules (*D* pour des bascules D), et les valeurs des sorties à partir de la valeur de l'état courant et des valeurs des entrées.
4. En déduire l'expression combinatoire des variables de contrôle des bascules et des sorties.
5. Réaliser un circuit comportant *n* bascules et les circuits combinatoires ci-dessus.

On va appliquer cette méthode de conception au multiplieur séquentiel. À partir de l'organigramme du multiplieur, on peut identifier quatre états différents :
- état initial,
- addition de la première opérande au résultat intermédiaire,
- itération (décalage des opérandes, incrémentation du compteur),
- fin de la multiplication.

L'automate correspondant est le suivant :



Comme l'automate comporte quatre états, il faudra utiliser deux bascules. On peut utiliser des bascules de n'importe quel type, dans notre cas nous choisirons des bascules D. On peut associer arbitrairement les états aux valeurs des bascules, voir un exemple ci-dessous (Q_i dénote la sortie de la bascule *i*).

Q_1	Q_0	Etat
0	0	Initial
0	1	Addition
1	0	Itération
1	1	Fin

A partir de l'automate, on peut déterminer toutes les transitions que doivent effectuer les bascules, et en déduire comment les commander. Plus précisément, cela signifie déterminer quel circuit placer en entrée des variables de contrôle de ces bascules ; par exemple, pour des bascules D, cela signifie déterminer l'expression de D en fonction des variables d'entrée et de l'état courant des bascules. La table ci-dessous récapitule l'ensemble des transitions possibles pour les variables des bascules; notez que si l'état courant d'une bascule est Q, son état suivant est indiqué par Q^+. On indique également sur cette table la valeur des sorties correspondant à l'état courant. Par exemple, on peut lire de la façon suivante la deuxième ligne de la table : lorsque l'état est «addition» (Q_1 Q_0=01) et que les entrées sont O_0=0 et C=0, l'automate indique que l'on passe dans l'état «itération» (Q_1 Q_0=10), et que les valeurs des sorties pour l'état « addition » sont toutes égales à 0, sauf *Wrésultat*.

O_0	C	Q_1	Q_0	Q_1^+	Q_0^+	$W_{op.}$	RAZ	$W_{rés}$	Déc.	Inc.	Fin
0	0	0	0	1	0	1	1	0	0	0	0
0	0	0	1	1	0	0	0	1	0	0	0
0	0	1	0	1	0	0	0	0	1	1	0
0	0	1	1	1	1	0	0	0	0	0	1
0	1	0	0	1	0	1	1	0	0	0	0
0	1	0	1	1	0	0	0	1	0	0	0
0	1	1	0	1	1	0	0	0	1	1	0
0	1	1	1	1	1	0	0	0	0	0	1
1	0	0	0	0	1	1	1	0	0	0	0
1	0	0	1	1	0	0	0	1	0	0	0
1	0	1	0	0	1	0	0	0	1	1	0
1	0	1	1	1	1	0	0	0	0	0	1
1	1	0	0	0	1	1	1	0	0	0	0
1	1	0	1	1	0	0	0	1	0	0	0
1	1	1	0	1	1	0	0	0	1	1	0
1	1	1	1	1	1	0	0	0	0	0	1

On peut remarquer que cette table constitue une table de vérité pour les sorties et pour les variables Q^+ en fonction des entrées O_0, C, Q_0 et Q_1. Comme $D= Q^+$ pour les bascules D, elle constitue également une table de vérité pour les variables de contrôle des bascules D_1, D_0. On peut donc en déduire l'expression des fonctions combinatoires associées aux sorties et aux variables de contrôle des bascules :

$$D_1 = \overline{O_0} + Q_0 + C.Q_1$$

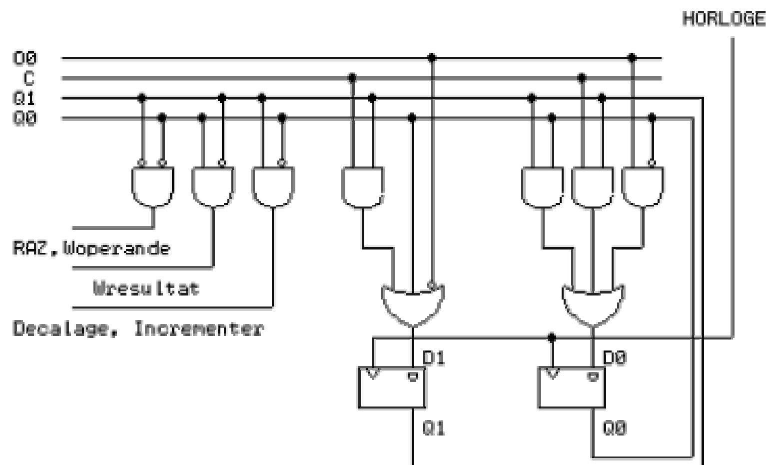
$$D_0 = O_0.\overline{Q_0} + Q_1.Q_0 + C.Q_1$$

$$W_{opérande} = RAZ = \overline{Q_1}.\overline{Q_0}$$

$$W_{rés} = \overline{Q_1}.Q_0$$

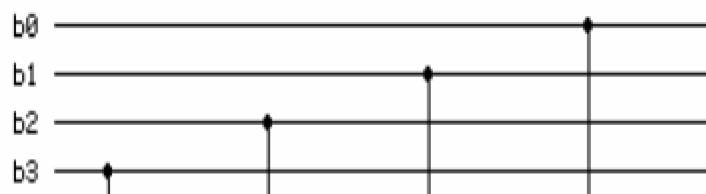
$$Décalage = Incrémenter = Q_1.\overline{Q_0}$$

Le circuit correspondant est le suivant :



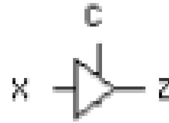
6 Les chemins de données

A l'intérieur d'un processeur, les connexions entre composants peuvent être complexes. Par exemple des données provenant de l'UAL et la mémoire peuvent être stockées dans un registre, et ces deux composants doivent donc être reliés au banc de registres. Une première solution consiste à utiliser de grands multiplexeurs en entrée des unités partagées. La communication est alors rapide, bien que le temps de traversée d'un multiplexeur augmente avec sa taille. En revanche, le coût d'un tel multiplexeur peut être important. Une solution moins coûteuse, mais éventuellement moins rapide, est d'utiliser un **bus**.

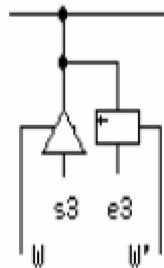


Un bus capable de transférer un mot de *n* bits d'une source à un récepteur est essentiellement un ensemble de *n* lignes. Les *n* bits de sortie de la source sont mis en contact avec le bus dont les *n* lignes portent alors la valeur fournie par la source. Le récepteur est également connecté au bus et ses *n* lignes d'entrée portent alors la valeur des lignes du bus, il y a eu transmission de l'information. Le principal problème est de pouvoir *connecter* et *déconnecter* les sources et les récepteurs du bus. Pour ce faire on utilise un composant spécial, un **tristate**. Il s'agit d'un circuit comportant une entrée *X*, une sortie *Z*, et un signal de contrôle *C*. Le schéma du circuit et la table de vérité sont indiqués ci-dessous.

X	C	Z
0	0	∞
0	1	0
1	0	∞
1	1	1



Lorsque le signal de contrôle C vaut 0, le circuit se comporte comme un interrupteur ouvert. Lorsque C vaut 1, le circuit se comporte comme une simple ligne. Ce circuit permet donc d'assurer la connexion et la déconnexion entre deux composants. Il arrive souvent qu'un bus permette d'envoyer des données à une source et de recevoir des données depuis cette source. Dans ce cas, on place une dérivation en amont du tristate qui alimente le bus, la liaison d'entrée pouvant éventuellement contenir un latch, voir figure ci-dessous.



Charger les lignes du bus avec un signal électrique n'est pas instantané. Ce temps augmente avec la taille du bus, i.e., si un bus est long il faut allouer plus de temps à la propagation du signal d'une unité à l'autre. C'est la première limitation des bus. La deuxième limitation des bus est qu'une seule donnée peut transiter à la fois (les bus les plus récents permettent de s'affranchir partiellement de cette contrainte). Dans le cas où deux unités distinctes désirent utiliser le bus pour communiquer, elles doivent saisir le bus à tour de rôle, en général à des cycles d'horloge distincts. Pour arbitrer ce type de conflits, un bus comporte généralement un contrôleur (un circuit de contrôle) ; ce contrôleur sert à implémenter un protocole de communication qui réglemente les échanges entre unités connectées au bus.
Outre les lignes servant à la transmission des données, les bus comportent souvent des lignes d'adresses permettant d'indiquer où la donnée doit être envoyée ou stockée, et des lignes de commande permettant de spécifier la nature de l'opération (lecture, écriture,...).

Bibliographie

[1] Intel 4004,
http://www.intel.com/intel/intelis/museum/exhibit/hist_micro/hof/hof_main.htm.
[2] Intel Pentium 4, http://www.intel.com/home/desktop/pentium4.
[3] Intel Itanium, http://www.intel.com/itanium.

[4] C.E. Shannon, *The synthesis of two-terminal switching circuits*, Transactions of the American Institute of Electrical Engineers, 28, 1, 59-98, 1949.
[5] S. B. Furber, D. A. Edwards and J. D. Garside, *AMULET3: a 100 MIPS Asynchronous Embedded Processor*, Proceedings of ICCD'00, Austin, Texas.
[6] Yale N. Patt, Sanjay J. Patel, *Introduction to Computing Systems, from bits and gates to C and beyond*, McGraw Hill International Editions, 2001.
[7] Jean-Michel Muller, *Arithmétique des ordinateurs. Opérateurs et fonctions élémentaires*, Masson, 1989.
[8] David A. Patterson et John L. Hennessy, *Organisation et Conception des Ordinateurs: L'interface Materiel/Logiciel*, chez Dunod.
[9] John L. Hennessy et David A. Patterson, *Architecture des Ordinateurs, une Approche Quantitative*, 2ème édition, chez Morgan Kaufmann.