ELECTRONIQUE NUMERIQUE

+

ARCHITECTURE DES ORDINATEURS

ELECTRONIQUE

NUMERIQUE

0. Préambule

ELECTRONIQUE NUMERIQUE

Différentes solutions électroniques :

. câblée :	- *avantage :* rapidité	- *inconvénient :* solution figée
. microprocesseur/microcontrôleur :	- *avantage :* souplesse	- *inconvénient :* lenteur/coût
. ordinateur embarqué :	- *avantage :* souplesse	- *inconvénient :* lenteur/coût
. FPGA (VHDL) :	- *avantage :* souplesse	- *inconvénient :* coût

Plan du cours

1. **Logique combinatoire 1. Fonctions logiques combinatoires basiques. Technologie** *(1hC + 1h30TD + 1h30TPx2-Tuto CM)*
   - Algèbre de Boole
   - Représentation des fonctions logiques
   - Minimisation des fonctions logiques
   - Matérialisation des fonctions logiques - Technologie
2. **Logique combinatoire 2. Applications** *(1hC + 1h30TD + 1h30TP)*
   - Les Applications directes (Codeur / Décodeur / Transcodeur / Multiplexeur / Démultiplexeur / Circuits arithmétiques)
   - Les Réseaux Logiques Programmables (ROM / PROM / PAL-GAL / PLA-PLD / FGPA / RAM / ASIC)
3. **Logique séquentielle 1. Fonctions logiques séquentielles basiques** *(1hC + 1h30TD + 1h30TP)*
   - Les bascules synchrones (RST /D /T / JK)
4. **Logique séquentielle 2. Applications** *(1hC + 1h30TD + 1h30TP)*
   - Registre
   - Compteur
   - Séquenceur
5. **Le langage VHDL** *(1hC + 1h30TD + 1h30TP)*

Le langage

- Eléments du langage / Unités de conception / Sous-programmes / Types de données / Déclarations & Spécifications / Instructions séquentielles / Instructions concurrentes / Généricité / Attributs / Paquetage

Modélisation / Synthèse

- Modélisation (Registre & Additionneur / Cicuits linéaires / Automate d'étas finis / ALU / RAM / ROM)
- Synthèse (Circuits combinatoires / Circuits synchrones)

Annexe

6. **Circuits Programmables.**
7. **Microprocesseur**
   - Matériel (Achitecture / Séquencement des instructions / ALU / Registres / Pile)
   - Logiciel (Assembleur / Jeu d'instructions / Modes d'adressage / Adressage des périphériques / Interruptions)
   - Interfaces
   - Microcontrôleur (Achitecture / Système de développement / Familles de microcontrôleurs)

Bibliographie

[1]	**R. Airiau & al.**	« VHDL langage, modélisation, synthèse »	*PPUR*
[2]	**J. Auvray**	« Electronique des signaux échantillonnés et numériques »	*Dunod*
[3]	**G. Baudouin/F. Virolleau**	« Les processeurs de traitement de signal: famille 320C5X »	*Dunod*
[4]	**B. Beghyn**	« Le microcontrôleur 68HC11 »	*Hermès*
[5]	**Dietsche/Ohsmann**	« Manuel des microcontrôleurs 8032/805/80535 »	*Publitronic*
[6]	**M. Gindre / D. Roux**	« Electronique numérique »	*McGraw-Hill*
[7]	**R.D. Hersch**	« Informatique industrielle: microprocesseurs/temps réel »	*PPUR*
[8]	**H. Lilen**	« Microprocesseurs : du CISC au RISC »	*Dunod*
[9]	**E. Martin/J.L. Philippe**	« Ingénierie des systèmes à microprocesseurs »	*Masson*
[10]	**M. Meaudre J. Weber**	« VHDL du langage au circuit, du circuit au langage »	*Masson*
[11]	**B. Mitton**	« Interfaces et bus microprocesseurs 68000/68070 »	*Mentor sciences*
[12]	**B. Mitton**	« Microprocesseur 68000 »	*Mentor sciences*
[13]	**B. Odant**	« Microcontrôleurs 8051/8052»	*Dunod Tech*
[14]	**C. Tavernier**	« Microcontrôleurs »	*Radio*
[15]	**R.L. Tokheim**	« Les microprocesseurs »	*Série Schaum*

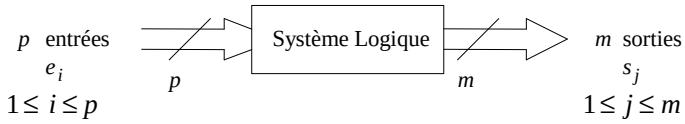
PPUR : Presses Polytechniques et Universitaires Romandes

1. LOGIQUE COMBINATOIRE 1 - LES BASES

0. Introduction

L'électronique logique (≡ électronique numérique) met en jeu des signaux binaires (n'ayant que 2 états possibles) appelés bits et notés 0 (état logique bas) et 1 (état logique haut).

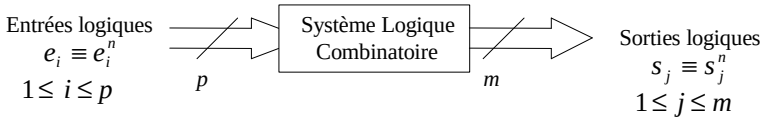
Système logique



Système logique combinatoire

A l'instant discret n, une sortie s_j, notée s_j^n , d'un système logique combinatoire ne dépend que de ses entrées $e_1^n,\ldots,e_p^n$ au même instant : (la seule connaissance des entrées suffit à déterminer les sorties)

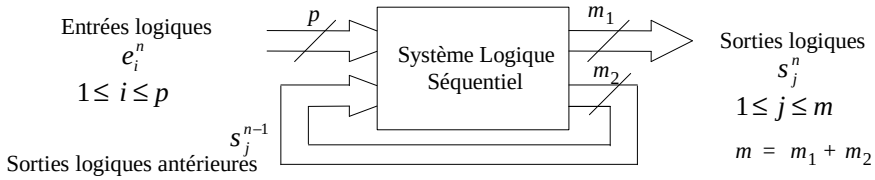
$$s_j^n = f(e_1^n,\ldots,e_p^n) \quad (1 \le j \le m)$$



Système logique séquentiel

A l'instant discret n, une sortie s_j^n d'un système logique séquentiel dépend de ses entrées $e_1^n,\ldots,e_p^n$ mais aussi de l'état antérieur des sorties ($s_1^{n-1},\ldots,s_m^{n-1}$) qui peuvent être considérées comme des entrées secondaires, alors que les entrées $e_1^n,\ldots,e_p^n$ sont appelées primaires. (Notion de mémoire, car les systèmes séquentiels sont bouclés, ou encore récursifs) : (la seule connaissance des entrées (primaires) ne suffit pas à déterminer l'état des sorties)

$$s_j^n = f(e_1^n,\ldots,e_p^n,s_1^{n-1},\ldots,s_m^{n-1}) \quad (1 \le j \le m)$$



Logique combinatoire

1. Algèbre de Boole

1.1. Opérateurs Fondamentaux (ET, OU, NON)

En algèbre de Boole, une variable, ou une fonction, ne peut prendre que deux valeurs binaires que l'on note symboliquement 0 et 1.

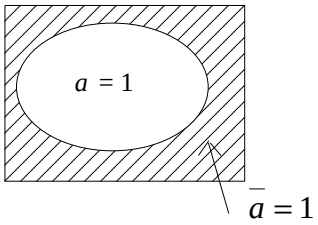
A l'aide de variables binaires (donc à 2 états) on peut néanmoins décrire des variables ayant un plus grand nombre d'états, en constituant ces dernières comme des mots binaires. Un mot binaire est une association de variables binaires. Ainsi un mot constitué de m bits ou variables binaires peut décrire 2^m combinaisons logiques.

Ex: Mot binaire *M* formé de m = 3 variables logiques binaires $a_2, a_1, a_0 \rightarrow M = a_2 a_1 a_0$ peut prendre 2^3 = 8 valeurs.

On définit les opérateurs fondamentaux : (dans tout ce qui suit, *x*, *a*, *b*, *c* ... représentent des variables binaires (bits))

a) Le *complément* (≡ opérateur *NON (NOT)*) $\overline{x}$ de x $\begin{cases}\text{qui vaut 0 si } x = 1\\ \text{qui vaut 1 si } x = 0\end{cases}$

Un opérateur peut être associée à une représentation géométrique issue de la théorie des ensembles que l'on appelle diagramme de Venn (ou d'Euler) :



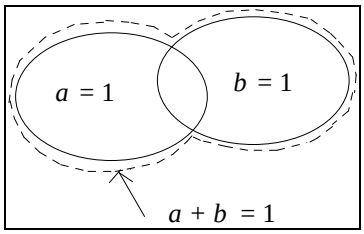
b) La *somme logique* (≡ opérateur *OU (OR)*) de deux variables booléennes.

Cet opérateur est noté par le signe + (mais aussi par le signe $\vee$ ou encore $\cup$)

$a + b = 1$ si a ou b (ou les deux) vaut 1 et $a + b = 0$ sinon . $a + b \equiv a \vee b \equiv a \cup b$

Le signe + n'a pas ici la signification habituelle, il est évident en effet qu'en algèbre de Boole : 1 + 1 = 1
Si une ambiguité peut exister, il vaut mieux alors utiliser la notation usuelle de la théorie des ensembles : $a \cup b$.

On considère un plan dans lequel on délimite une région où la variable a vaut 1 et une autre région dans laquelle b vaut 1. La somme logique a pour valeur 1 dans la surface formée par la réunion des deux régions précédentes :

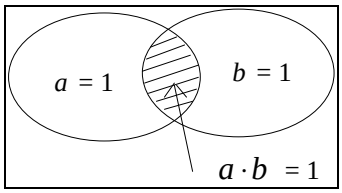


c) Le *produit logique* (≡ opérateur *ET (AND)*) de deux variables booléennes :

Cet opérateur est noté par le signe • (mais aussi par le signe $\wedge$ ou encore $\cap$ ou même pas de signe)

$a \cdot b = 1$ si a et b valent 1 et $a \cdot b = 0$ sinon . $a \cdot b \equiv a \wedge b \equiv a \cap b \equiv ab$

Le diagramme de Venn correspondant conduit à écrire $a \cap b$:



1.2. Propriétés des opérations logiques élémentaires ET, OU, NON

a) Elément neutre :

$$\begin{cases} a+0=a \\ a\cdot 1=a \end{cases}$$

b) Elément absorbant :

$$\begin{cases} a+1=1 \\ a\cdot 0=0 \end{cases}$$

c) Complément :

$$\begin{cases} \overline{(\overline{a})}=a \\ a+\overline{a}=1 \\ a\cdot\overline{a}=0 \end{cases}$$

d) Idempotence :

$$\begin{cases} a+a=a \\ a\cdot a=a \end{cases}$$

Les opérations d'addition et de multiplication logiques ont les propriétés des opérations de même nom en arithmétique classique : commutativité, associativité, distributivité :

e) Associativité :

$$\begin{cases} (a+b)+c=a+(b+c)=a+b+c \\ (a\cdot b)\cdot c=a\cdot(b\cdot c)=a\cdot b\cdot c \end{cases}$$

f) Commutativité :

$$\begin{cases} a+b=b+a \\ a\cdot b=b\cdot a \end{cases}$$

g) Double distributivité :

$$\begin{cases} a\cdot(b+c)=a\cdot b+a\cdot c & \text{(Distributivite de } \cdot \text{ par rapport a } +) \\ a+(b\cdot c)=(a+b)\cdot(a+c) & \text{(Distributivite de } + \text{ par rapport a } \cdot) \end{cases}$$

(en arithmétique classique, on n'a que la distributivité de $\cdot$ par rapport à $+$)

Parmi les relations simples les plus utilisées nous citerons les relations suivantes :

h) Absorption :

$$\begin{cases} a+a\cdot b=a & (1) \\ a\cdot(a+b)=a & (2) \\ a+\overline{a}\cdot b=a+b & (3) \\ x\cdot a+x\cdot\overline{a}=x & (4) \end{cases}$$

Explications :

(1) : $a+a\cdot b=a\cdot(1+b)=a\cdot 1=a$ car : $(1+b=1)$

(2) : $a\cdot(a+b)=a\cdot a+a\cdot b=a+a\cdot b$ (et relation précédente)

(3) : $a+(\overline{a}\cdot b)=(a+\overline{a})\cdot(a+b)$ (double distributivité) $\rightarrow a+(\overline{a}\cdot b)=1\cdot(a+b)=a+b$.

(4) : $x\cdot a+x\cdot\overline{a}=x\cdot(a+\overline{a})=x\cdot 1=x$: Cette relation est importante car elle permet l'élimination d'une variable.

1.3. Les théorèmes de De Morgan

Ils définissent les relations entre l'opération complément et les deux autres opérations de base (OU et ET).

$$\begin{cases} \overline{a+b} = \bar{a}\cdot\bar{b} \\ \overline{a\cdot b} = \bar{a}+\bar{b} \end{cases}$$ généralisable à *n* variables

Le complément d'une somme est égal au produit des compléments des termes.
Le complément d'un produit est égal à la somme des compléments des termes.

Ces théorèmes peuvent être montrés à l'aide des diagrammes de Venn ou encore des tables de vérité (cf. plus loin).

1.4. Principe de dualité

Une équation logique reste vraie si on remplace + par $\cdot$ et 0 par 1 et réciproquement.

Ex. : $a+b=b+a \quad\leftrightarrow\quad \bar{a}\cdot\bar{b}=\bar{b}\cdot\bar{a}$ *Autre exemple :* $a+1=1 \quad\leftrightarrow\quad \bar{a}\cdot 0=0$

1.5. Autres fonctions élémentaires de deux variables

Les trois fonctions précédentes suffisent à elles seules à effectuer toutes les opérations logiques. On définit cependant quelques fonctions annexes.

a) Le *OU exclusif* (XOR)

La fonction $a\ XOR\ b$ est notée $a\oplus b$.

$a\oplus b$ vaut 1 si a ou b vaut 1, mais pas les deux à la fois: $a\oplus b=1$ si $\begin{cases}\text{si } a=1 \text{ et } b=0\\ \text{ou si } a=0 \text{ et } b=1\end{cases}$ et $a\oplus b=0$ sinon.

On a bien évidemment la relation : $a\oplus b=\bar{a}\oplus\bar{b}$.

$a\oplus b$ peut évidemment s'exprimer à partir des opérations élémentaires : $a\oplus b=a\cdot\bar{b}+\bar{a}\cdot b$.

Application : cryptographie

On a les propriétés : $x\oplus\bar{x}=1$
$x\oplus x=0$
$x\oplus 0=x$
$x\oplus 1=\bar{x}$
$\rightarrow \quad x\oplus a\oplus a=x\oplus 0=x$

Soit à crypter la donnée x. La clé de cryptage et de décryptage est a. L'algorithme de cryptage/décryptage est le OU exclusif entre la donnée et la clé de cryptage/décryptage (Cette méthode bien connue présente la particularité d'utiliser la même clé ainsi que le même algorithme pour le cryptage et le décryptage).

à crypter x	clé a	cryptage $x\oplus a$	décryptage $(x\oplus a)\oplus a=x$
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

Le *OU exclusif* de deux variables a, b traduit *l'inégalité* de ces deux variables.

La fonction inverse, traduisant l'égalité, est la **coïncidence**, notée $\odot$: $a\odot b=\overline{a\oplus b}$

$a\odot b=a\cdot b+\bar{a}\cdot\bar{b}$ car : $\overline{a\oplus b}=\overline{a\cdot\bar{b}+\bar{a}\cdot b}=(\bar{a}+b)\cdot(a+\bar{b})=a\cdot\bar{a}+\bar{a}\cdot\bar{b}+a\cdot b+b\cdot\bar{b}=a\cdot b+\bar{a}\cdot\bar{b}$

On a bien évidemment la relation : $a\odot b=\bar{a}\odot\bar{b}$.

Le symbole utilisé pour le OU exclusif est identique à celui désignant en arithmétique une addition modulo 2, il s'agit en effet de la même opération : $1\oplus 1=0 \qquad 1\oplus 0=0\oplus 1=1 \qquad 0\oplus 0=0$

b) Les fonctions NOR et NAND

La fonction NOR est le complément de OU (NOR ≡ NO OR) : a NOR $b = \overline{a+b}$ (entre 2 variables a et b)
On la note souvent simplement : a NOR b
On trouve parfois la notation suivante introduite par PIERCE : $a \downarrow b$

La fonction NAND (NO AND) est le « ET complémenté » : a NAND $b = \overline{a \cdot b}$ (entre 2 variables a et b)
On la note souvent simplement : a NAND b
On trouve parfois la notation introduite par PIERCE : $a \uparrow b$ ou encore la notation : a / b

La combinaison de ces diverses opérations logiques entre plusieurs variables constitue ce que l'on appelle les *fonctions logiques*.

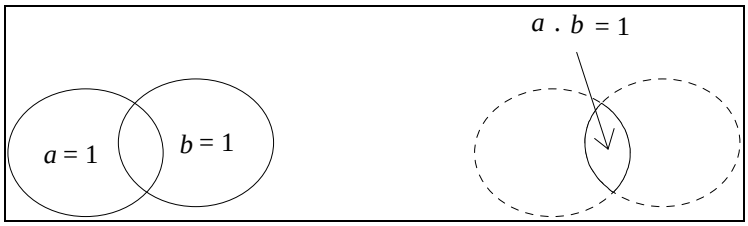
Par exemple, la fonction logique de 3 variables : $f = f(a,b,c) = a + b \cdot \bar{c} + \bar{a} \cdot b \cdot c$

1.6. Représentation des fonctions logiques

a) Diagramme de Venn

C'est la représentation issue de la théorie des ensembles.

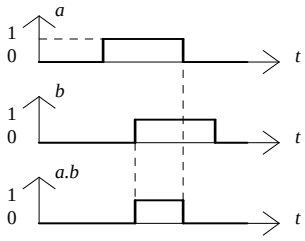
Ex. :



b) Chronogramme

C'est la représentation de la fonction logique en fonction du temps pour diverses valeurs des variables d'entrée.

Ex. :



c) Table de vérité

C'est le tableau des valeurs de la fonction pour toutes les valeurs possibles des variables d'entrée.

Ex. :

Fonction NOT

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

Fonction OR

a	*b*	*a* + *b*
0	0	0
0	1	1
1	0	1
1	1	1

Fonction AND

a	*b*	*a.b*
0	0	0
0	1	0
1	0	0
1	1	1

Fonction NOR

a	*b*	*a* NOR *b*
0	0	1
0	1	0
1	0	0
1	1	0

Fonction NAND

a	*b*	*a* NAND *b*
0	0	1
0	1	1
1	0	1
1	1	0

Fonction OU exclusif

a	*b*	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

Fonction Coïncidence

a	*b*	$a \odot b$
0	0	1
0	1	0
1	0	0
1	1	1

d) Diagramme de Karnaugh

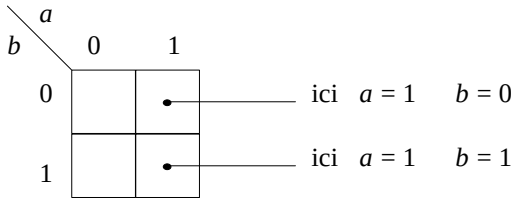
C'est une forme particulière de la table de vérité.

Le diagramme de Karnaugh se compose d'un rectangle divisé en 2^n cases, *n* étant le nombre de variables de la fonction considérée. Dans chacune de ces cases les variables ont une valeur déterminée et on y place un 0 ou un 1 suivant la valeur correspondante de la fonction.

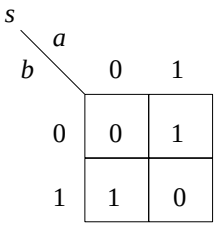
L'ordre des variables en abscisse et ordonnée est tel que lorsque l'on passe d'une case à la case adjacente *une seule variable est modifiée*.

2 cases sont adjacentes si elles sont voisines verticalement, horizontalement ou en coin, mais toujours de telle sorte qu'une seule variable d'entrée est modifiée lorsque l'on passe d'une case à une case adjacente.

Diagramme de Karnaugh à 2 variables a, b. (2^2 = 4 cases)

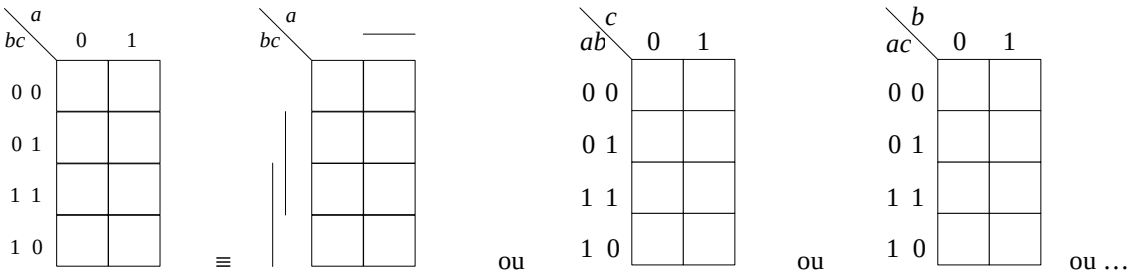


Représentons la fonction OU exclusif : $s = a \oplus b$ sur ce diagramme :



$s = a \oplus b = 1$ si $\begin{cases} \text{si } a = 1 \text{ et } b = 0 \\ \text{ou si } a = 0 \text{ et } b = 1 \end{cases}$ Ce sont les 2 cases suivant la 2ème diagonale.

Diagramme de Karnaugh à 3 variables a, b, c



2^3 = 8 cases, il faut utiliser un rectangle ayant par exemple 4 lignes et 2 colonnes, mais on peut prendre aussi 2 lignes et 4 colonnes. On remarquera que de la deuxième à la troisième ligne on passe de (0 1) à (1 1) et non de (0 1) à (1 0) de façon à ne modifier qu' une variable à la fois (code Gray).

La fonction $s = a\bar{b} + c\bar{a}$ dont la table de vérité est :

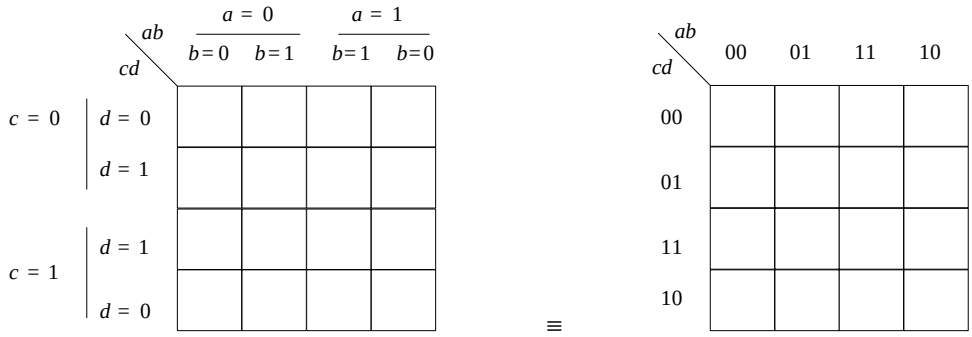
a	*b*	*c*	$a\bar{b}$	$c\bar{a}$	*s*
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	1	0	1
1	0	1	1	0	1
1	1	0	0	0	0
1	1	1	0	0	0

a pour diagramme de Karnaugh :

s : *bc* \ *a*	0	1
0 0	0	1
0 1	1	1
1 1	1	0
1 0	0	0

Diagramme de Karnaugh à 4 variables a, b, c, d

(2^4 = 16 cases). On retombe sur un carré.



L'ordre des variables est le même que le précédent : 0 0 - 0 1 - 1 1 - 1 0

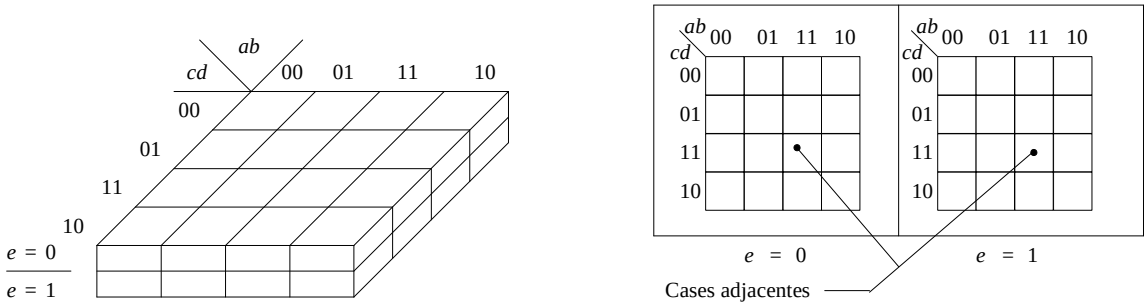
Exemple : $s = a\bar{b}cd + \bar{a}bc$

On pourra, en établissant la table de vérité, montrer que le diagramme de Karnaugh est celui ci :

s : *cd* \ *ab*	00	01	11	10
0 0	0	0	0	0
0 1	0	0	0	0
1 1	0	1	0	1
1 0	0	1	0	0

Diagramme de Karnaugh à 5 variables a, b, c, d, e

Lorsque l'on passe d'une case d'un diagramme de Karnaugh à la case adjacente, une seule variable est modifiée. Avec 5 variables, il faut que chaque case soit adjacente à 5 cases ce qui n'est pas possible dans une représentation plane. Il faut faire appel à un volume à 2^5 = 32 cases cubiques que l'on peut remplacer par un double tableau carré :



La commodité d'emploi est alors très réduite car il est plus délicat de repérer deux cases adjacentes.
La situation est encore pire avec 6 variables où il faut travailler avec un cube ou 4 tableaux carrés.
Au delà de 6 variables aucune représentation n'est possible et il faudra faire appel à d'autres procédés (logiciel de minimisation).

Les diagrammes de Karnaugh permettent comme nous allons le voir de simplifier très facilement des fonctions booléennes complexes. Au delà de 5 variables des méthodes algébriques peuvent toujours les remplacer mais sont beaucoup moins souples et d'un intérêt contestable.

e) Forme canonique

Toute fonction logique peut être mise sous forme canonique :

- comme une somme de *mintermes* ou encore,
- comme un produit de *maxtermes*.

On appelle *minterme* ou *fonction unité* de *n* variables un produit de ces *n* variables ou de leur complément.
Par exemple $A\,B\,\overline{C}\,D$ est un minterme des 4 variables *A, B, C, D* mais $A\,\overline{C}\,D$ n'en est pas un, car il manque la variable *B*.

Une fonction boolénne de *p* variables est décrite comme une *somme canonique* si elle est mise sous la forme d'une somme de mintermes de ces *p* variables.

On définit également un *produit canonique* qui est le produit de sommes contenant chacune toutes les variables.

$\left(A+\overline{B}+C+D\right)\left(\overline{A}+B+C+\overline{D}\right)\left(A+B+\overline{C}+D\right)$ est un produit canonique des 4 variables A, B, C, D.

On remarque que chaque parenthèse est un maxterme (complément d'un minterme). *Ex :* $A+\overline{B}+C+D=\overline{\overline{A}B\overline{C}\,\overline{D}}$

Les diagrammes de Karnaugh permettent très facilement de mettre une fonction logique sous forme d'une somme canonique car chaque minterme correspond à un 1 dans une seule case du diagramme.

1.7. Minimisation (≡ simplification) des fonctions logiques

Pourquoi simplifier une fonction logique ? Pour donner lieu à une réalisation matérielle la plus simple possible mettant en jeu un nombre minimal de circuits logiques et de signaux logiques.

1.7.1. Méthode algébrique

→ Utilisation des propriétés des opérations logiques élémentaires et des théorèmes de De Morgan.

La mise en équation d'un problème de logique peut conduire à une fonction boolénne assez complexe pouvant, par des opérations algébriques simples, se mettre sous une forme beaucoup plus condensée.

Soit par exemple, la fonction logique : $S = \underbrace{A\,C}_{1} + \underbrace{A\,\overline{B}}_{2} + \underbrace{B}_{3} + \underbrace{A\,\overline{D}}_{4} + \underbrace{A\,B\,D}_{5} + \underbrace{A\,\overline{C}}_{6} + \underbrace{A\,B}_{7}$

En groupant les termes 2 et 4: $A\,\overline{B} + A\,\overline{D} = A\left(\overline{B} + \overline{D}\right) = A\,\overline{B\,D}$

en ajoutant 5: $A\,\overline{B\,D} + A\,B\,D = A\left(\overline{B\,D} + B\,D\right) = A$ (8)

il reste: $S = \underbrace{A\,C}_{1} + \underbrace{B}_{3} + \underbrace{A\,\overline{C}}_{6} + \underbrace{A\,B}_{7} + \underbrace{A}_{8}$

mais: $1+6 \rightarrow A\,C + A\,\overline{C} = A\left(C + \overline{C}\right) = A$, terme identique à 8 donc inutile $\left(A + A = A\right)$

il reste: $S = \underbrace{A}_{8} + \underbrace{B}_{3} + \underbrace{A\,B}_{7}$

mais encore: $8 + 7 = A(1 + B) = A$

donc finalement : $S = A + B$ (résultat qui serait obtenu aussi en faisant $3 + 7 = B$)

De façon générale de très nombreuses fonctions logiques sont susceptibles d'être simplifiées mais la forme la plus compacte n'est pas toujours trouvée immédiatement car la voie de simplification algébrique la plus rapide n'est pas évidente. Dans le cas de 5 variables au plus, les diagrammes de Karnaugh permettent d'effectuer cette simplification « automatiquement ».

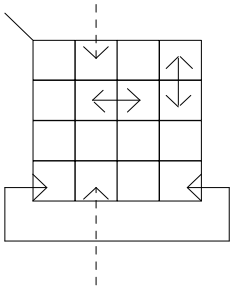
1.7.2. Méthode graphique : Diagramme de Karnaugh

La méthode de simplification de Karnaugh consiste à mettre à profit la relation : $x\,A + x\,\overline{A} = x$
pour donner une expression simplifiée de la fonction à l'aide des opérateurs fondamentaux.

On recherche les termes ne différant que par un seul facteur qui apparait complémenté dans le premier et non complémenté dans le second. Ces deux termes s'ils font partie d'une somme canonique correspondent dans le diagramme de Karnaugh à deux 1 placés dans des cases adjacentes. On forme alors ce que l'on appelle une boucle d'ordre 2. Remarquons que deux cases doivent être considérées comme adjacentes si l'on passe de l'une à l'autre en ne modifiant qu'une seule variable, ce qui est le cas de deux cases placées réellement côte à côte, mais aussi aux deux extrémités d'une ligne.

La simplification se fera en regroupant les 1 en boucles d'ordre 2, 4, 8, etc ... ordre 2^n .

Cases adjacentes

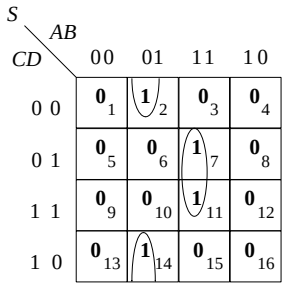


Soit la fonction logique : $S = AB\overline{C}D + ABCD + \overline{A}B\overline{C}\,\overline{D} + \overline{A}BC\overline{D}$

case 7 11 2 14

$S = A\,B\,D + \overline{A}\,B\,\overline{D}$ par la méthode algébrique.

a) Boucles d'ordre 2



On peut former deux boucles aves les cases adjacentes 7 - 11 et 2 - 14.

La boucle 7 - 11 donne un terme *ABD* : en effet la variable *C* qui change en passant d'une case à l'autre s'élimine :

$$AB\overline{C}D + ABCD = ABD(C + \overline{C}) = ABD$$

La boucle 2 - 14 donne $\overline{A}\ B\ \overline{D}$, donc :

$$S = \overline{A}\ B\ \overline{D} + A\ B\ D \qquad \text{ou :} \qquad S = B\left(\overline{A}\ \overline{D} + A\ D\ \right) = B\left(\overline{A \oplus D}\right)$$

Les boucles d'ordre 2 font disparaître 1 variable dans les mintermes de la fonction. Cette variable est celle qui varie dans ces boucles $\rightarrow$ les variables restantes sont celles qui sont constantes dans ces boucles.

Si on veut écrire et simplifier *S*, tous les 1 du tableau doivent être groupés en boucles (avec des boucles comptant le plus grand nombre de termes possibles, les boucles pouvant éventuellement se recouper, du fait de la propriété : $x + x = x$) ou si ce n'est pas possible, comptés individuellement.

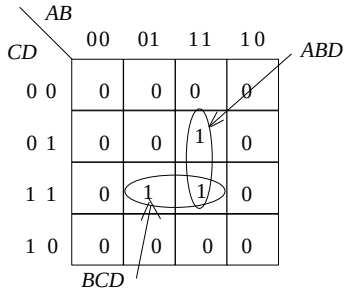
b) Boucles imbriquées

Soit la fonction logique : $S = A\ B\ \overline{C}\ D + A\ B\ C\ D + \overline{A}\ B\ C\ D$

case ↓ 7 ↓ 11 ↓ 10

Deux boucles sont possibles 7 - 11 ou 10 - 11, elles ont la case 11 en commun mais on peut appliquer la règle précédente comme si ces boucles étaient disjointes. En effet *S* ne change pas si on dédouble un de ses termes.

$$S = \underbrace{A\ B\ \overline{C}\ D + A\ B\ C\ D}_{\text{boucle 7 - 11}} + \underbrace{A\ B\ C\ D + \overline{A}\ B\ C\ D}_{\text{boucle 10 - 11}}$$



$$S = A\ B\ D + B\ C\ D = BD(A + C)$$

c) Boucles d'ordre 4

Supposons que deux boucles d'ordre 2 soient adjacentes.

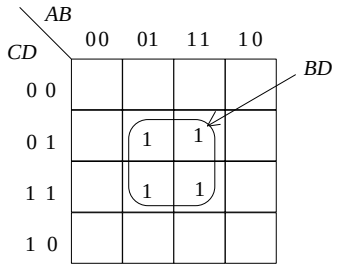
Par exemple : 6 - 7 et 10- 11 soit : $\overline{A}\,B\,\overline{C}\,D + A\,B\,\overline{C}\,D + \overline{A}\,B\,C\,D + A\,B\,C\,D$

(6) (7) (10) (11)

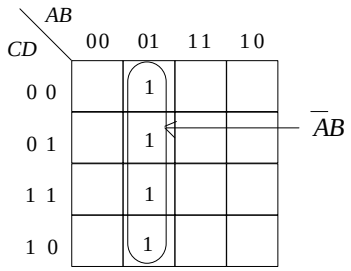
$B\,\overline{C}\,D \quad + \quad B\,C\,D$

Les deux résultats peuvent de nouveau se combiner et *C* disparaît : $B\,\overline{C}\,D + B\,C\,D = B\,D$

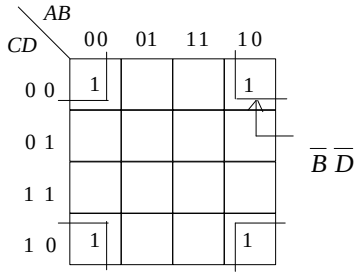
Les quatre cases ainsi groupées forment une boucle d'ordre 4 :



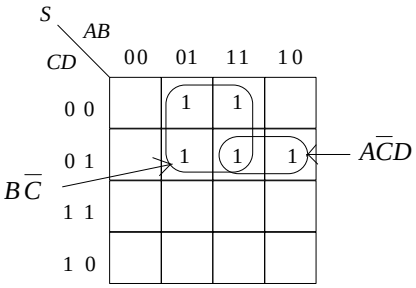
Une boucle d'ordre 4 n'est pas forcément carrée. *Par exemple :*



Elle peut être écartelée entre les deux bords ou même entre les quatre coins :



Elle peut être en partie commune avec une boucle du deuxième ordre comme ci-dessous :



qui pourrait s'écrire : $S = \overline{A}\,B\,\overline{C}\,\overline{D} + A\,B\,\overline{C}\,\overline{D} + \overline{A}\,B\,\overline{C}\,D + A\,B\,\overline{C}\,D + A\,\overline{B}\,\overline{C}\,D = A\,\overline{C}\,D + C\,B$

Les boucles d'ordre 4 font disparaître 2 variables dans les mintermes.

d) Boucles d'ordre 8

Si deux boucles d'ordre 4 sont adjacentes, on peut former une boucle d'ordre 8 pour laquelle trois variables disparaissent.

Les deux boucles d'ordre 4 : (1 - 5 - 9 - 13) et (4 - 8 - 12 - 16) donnent : $\overline{A}\,\overline{B} + A\overline{B} = \overline{B}$:

CD \ *AB*	00	01	11	10
0 0	1			1
0 1	1			1
1 1	1			1
1 0	1			1

Elles sont adjacentes et forment une boucle d'ordre 8 où seule la variable $\overline{B}$ est conservée.

Les boucles d'ordre 8 font disparaître 3 variables dans les mintermes.

e) Boucles d'ordre 2^n

Les boucles d'ordre 2^n regroupent 2^n variables et font disparaître n variables dans les mintermes de la fonction.

De façon générale, on a intérêt à effectuer les plus grands regroupements possibles (≡ boucles d'ordre le plus élevé) pour simplifier au maximum la fonction.

f) Fonctions Booléennes X (ou ϕ*)*

Il existe des cas où toutes les combinaisons possibles des n variables ne sont pas utilisées, c'est la cas par exemple des 4 variables constituant une tétrade en code *DCB (Décimal Codé Binaire* ou encore *Binaire pur),* les six pseudo tétrades (10 à 15) sont exclues :

Chiffre de 0 à 9	Code *DCB : A B C D*
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1
	6 codes restants non utilisés

Dans ces conditions pour simplifier une fonction booléenne de quatre variables dont le champ de variation est limité, une valeur quelconque peut être donnée à la fonction dans les cases interdites de façon à constituer des boucles d'ordre le plus élevé possible sur le diagramme de Karnaugh. Le signe X (ou ϕ) étant utilisé pour indiquer qu'un 1 aussi bien qu'un 0 convient. On parle souvent de fonctions ϕ booléennes.

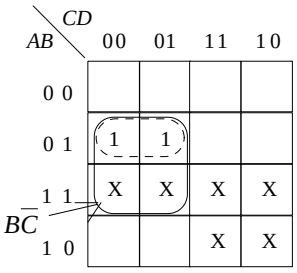
Soit par exemple à commander un voyant qui doit être allumé lorsque les chiffres 4 ou 5 apparaissent et seulement dans ces cas. Les chiffres sont codés en *DCB* sur quatre fils *A, B, C* et *D*. La fonction booléenne L à créer ne doit valoir 1 que lorsque les configurations 4 (0100) ou 5 (0101) apparaissent.

En toute rigueur : $L = \overline{A}B\overline{C}\,\overline{D} + \overline{A}B\overline{C}\,D$ correspondant au diagramme de Karnaugh ci-dessous sur lequel apparaît une boucle d'ordre 2 amenant ainsi la simplification : $L = \overline{A}B\overline{C}$.

Mais les cases marquées d'une croix (X ou ϕ) correspondent à des situations interdites qui ne se présenterons jamais sur les quatre fils (pseudo tétrades) et peuvent être choisies comme on veut (X ≡ *Don't care* (sans effet)).

Les X utilisés dans les regroupements sont mis à 1, les X non utilisés sont mis à 0.

Ici on peut placer des 1 dans les deux cases correspondant aux états 1100 (12) et 1101 (13) de façon à former une boucle d'ordre 4 :



Amenant ainsi une simplification optimale de L : $L = B\,\overline{C}$

La lampe L s'allume dans les états 4 et 5 demandés et aussi pour 12 et 13 ce qui n'a pas d'importance puisque ces deux derniers états ne se présentent jamais.

. Ne jamais regrouper uniquement des X ensemble : ça ne simplifie pas la fonction mais la complique en ajoutant un terme inutile à la fonction.

. Chaque X a une valeur indépendamment des autres X.

. Ne jamais affecter des valeurs différentes à un même X, lors de différents regroupements le faisant intervenir.

Complément sur la simplification :

Si le tableau de Karnaugh comporte plus de 0 que de 1, on a alors intérêt à regrouper les 0 et non les 1 pour exprimer $\overline{S}$ plutôt que *S*.

De même pour une simplification algébrique, la simplification peut se faire plus simplement sur $\overline{S}$ plutôt que *S*. Lorsque $\overline{S}$ est simplifiée au maximum, on obtient alors simplement *S* par complémentation de $\overline{S}$.

2. Matérialisation des fonctions logiques

2.1. Logique positive et logique négative

A toute grandeur physique ayant seulement deux valeurs possibles on peut associer une variable booléenne. En électronique, les grandeurs considérées sont essentiellement le courant et la tension. Par exemple, la tension collecteur d'un transistor NPN (*T*) alimenté sous 5 Volts (cas de la famille logique TTL) et fonctionnant en régime de commutation (régime de fonctionnement en logique) peut valoir :

$$\begin{cases} V_c = 0 & \text{si } T \text{ est saturé} \\ V_c = +5 \text{ Volts} & \text{si } T \text{ est bloqué} \end{cases}$$

On peut par convention admettre que la variable booléenne *C* associée à la tension V_c vaut 1 si V_c = + 5 V, 0 si V_c = 0. La valeur 1 est associée à la valeur la plus élevée de V_c ; on dit alors que l'on a défini une *logique positive* :

$$\begin{cases} C = 0 & \text{pour } V_c = 0 \\ C = 1 & \text{pour } V_c = +5 \text{ Volts} \end{cases}$$

Le contraire, bien que moins courant, est également possible. la logique est qualifiée alors de *négative* :

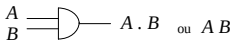
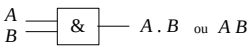
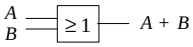
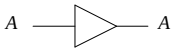
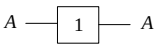
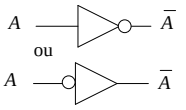
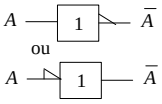
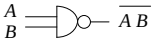
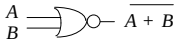
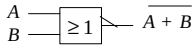
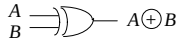
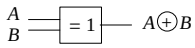
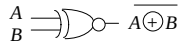
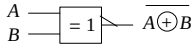
$$\begin{cases} C = 1 & \text{pour } V_c = 0 \\ C = 0 & \text{pour } V_c = +5 \text{ Volts} \end{cases}$$

Dans ce qui suit nous travaillerons toujours en logique positive.

2.2. Symboles logiques

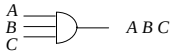
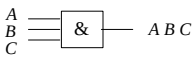
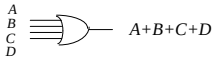
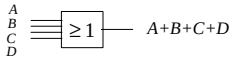
Une fonction (ou opérateur) logique élémentaire (≡ fondamental) est matérialisée par un circuit logique appelé porte logique.

Symboles logiques fondamentaux *(ancien ≡ américain ; nouveau ≡ européen)*

Opérateur	*Ancien symbole*	*Nouveau symbole*
AND		
OR		
BUFFER ou **DRIVER**	 (Un Buffer réhausse au niveau haut une tension de niveau haut diminuée)	
NOT		
NAND		
NOR		
XOR		
XNOR		

Note :
Le petit cercle représente la complémentation
La flêche représente la complémentation en plus du sens de l'information
En l'absence de flêche, le signal circule implicitement de la gauche vers la droite.

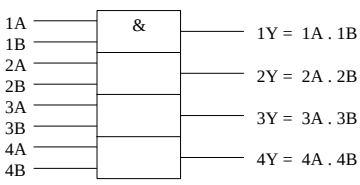
Symboles logiques de portes élémentaires à 3 et 4 entrées

Opérateur	*Ancien symbole*	*Nouveau symbole*
AND 3 entrées		
OR 4 entrées		

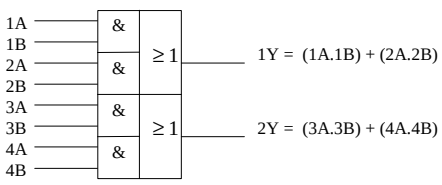
Symboles logiques de circuits multiples d'une même porte élémentaire à 2 entrées

Nouveaux symboles

Opérateurs ET
Circuit 74XX00 4 portes AND 2 entrées

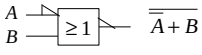


Opérateurs ET / OU
Circuit 74XX51



Symbole d'une fonction logique à l'aide des opérateurs fondamentaux

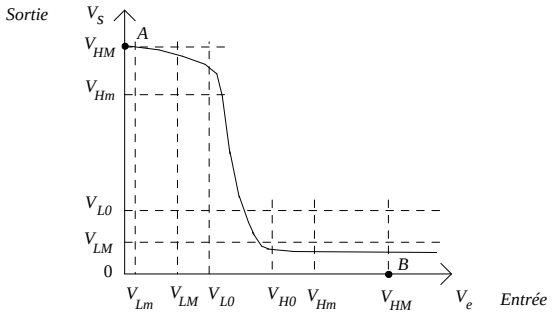
Exemple : Représentation de la fonction logique : $\overline{\overline{A} + B}$:



2.3. Caractéristiques fondamentales d'une porte logique

2.3.1. Définition des niveaux logiques : immunité au bruit

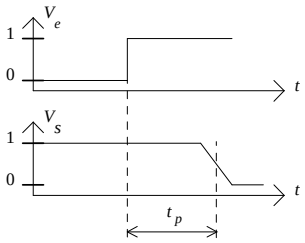
Considérons le cas simple d'un inverseur en logique 5 volts, c'est un circuit dont la sortie est à 5 volts si l'entrée est au zéro et réciproquement, ce qui ne définit sur la caractéristique de transfert que les deux points *A* et *B* de la figure suivante. En réalité par suite de l'influence des autres circuits qui lui sont connectés, les niveaux d'entrée et de sortie d'un tel inverseur n'ont jamais ces valeurs idéales et il y a lieu de considérer la courbe de transfert complète suivante :



En régime normal, la tension d'entrée au niveau zéro se trouve entre V_{Lm} et $V_{LM'}$, la tension de sortie étant alors au niveau haut (1) entre V_{Hm} et V_{HM} .
Si une impulsion parasite vient se superposer à la tension d'entrée, la tension de sortie restera compatible avec le niveau 1 si le niveau V_{L0} n'est pas dépassé.
$M_i = V_{L0} - V_{LM}$ est la marge de bruit admissible à l'entrée au niveau bas. De même, si l'entrée est au niveau 1, une impulsion parasite ne doit pas faire tomber V_e en dessous de V_{H0} .
$M_0 = V_{Hm} - V_{H0}$ est la marge de bruit admissible au niveau haut.
Ces deux marges définissent ce que l'on appelle *l'immunité au bruit* du circuit.

2.3.2. Temps de propagation

Si le niveau d'entrée d'un circuit change brutalement, son niveau de sortie ne varie qu'avec un certain retard appelé *temps de propagation* t_p .
La figure suivante illustre le cas d'un inverseur. Les temps de propagation sont couramment de 30 ns et peuvent, pour les circuits les plus rapides, être inférieurs à 1 ns.
La fréquence maximale d'utilisation au-delà de laquelle les signaux ne sont plus restitués par les circuits (limite haute de la Bande Passante) est lié au temps de propagation.



2.3.3. Tension d'alimentation

Par exemple pour les familles TTL et CMOS :

Tolérance sur les niveaux TTL

Tension d'alimentation	*Vcc*	5 V ± 0.5 V	
Tension maxi d'entrée pour un niveau bas	V_{IL}	0.8 V	L : Low
Tension mini d'entrée pour un niveau haut	V_{IH}	2 V	H : High
Tension maxi de sortie pour un niveau bas	V_{OL}	0.4 V	I : Input
Tension mini de sortie pour un niveau haut	V_{OH}	2.4 V	O : Output

Puissance moyenne absorbée par porte : ~ 10 mW
Courant moyen par porte : ~ qq. mA

CMOS

Tension d'alimentation : ≠ à TTL elle peut être de 3 à 18 Volts (série 4000)
(les nouvelles générations plus performantes n'autorisent que 2 à 6 Volts).
La puissance consommée est << TTL : de l'ordre de 0.1 mW → courant très faible < 1 mA.
Les tolérances sur les niveaux logiques sont du même ordre qu'en TTL.

2.3.4. Entrance (Fan in) et Sortance (Fan out)

La source qui impose à l'entrée d'un circuit logique un niveau 0 ou 1 doit fournir un certain courant. Ce courant est différent suivant l'état. Il peut être suivant le cas, maximal pour l'état 1 ou l'état 0. Dans une même famille de circuits, ces valeurs sont des constantes, sauf pour certains circuits particuliers dont les exigences peuvent être plus importantes.

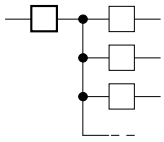
Entrance

On appelle *entrance* d'un circuit (ou *fan in*) la valeur du courant de commande d'une entrée de ce circuit exprimée en une unité qui est le courant de comande typique de la famille (appelé *charge*).

Ex. : un circuit ayant une entrance de 2 consomme (ou fournit) un courant d'entrée double de celui d'un circuit ordinaire de la même famille. Le courant unité correspond à ce qu'on appelle une « charge ».

Sortance

Il est clair qu'un circuit logique ne peut garantir sa tension de sortie que si le nombre de charges qui lui sont connectées est limité (un niveau logique 1 de sortie chute à 0 si le nombre de charges est trop élevé) :

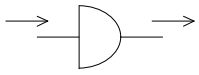


□ : circuit logique

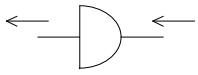
Un circuit logique peut d'autre part, sans que le niveau logique de sortie ne sorte des limites permises, fournir un courant maximal $I_{S\,max}$. Le rapport entre ce courant maximal et celui correspondant à une charge est appelé *sortance* du circuit (ou *fan out*, ou *facteur pyramidal de sortie*) : c'est le nombre maximal de charges que peut commander une sortie (à entrée unitaire) en garantissant les niveaux logiques.

Ex. : à un circuit ayant une sortance de 10, on peut connecter 10 charges tout en garantissant les niveaux de sortie de cette porte.

Le sens des courants est également très important. Une famille logique dont les circuits doivent être pilotés par un courant entrant est dite à *injection de courant*. Dans le cas contraire, on parle de logique à *extraction de courant* :



Injection de courant



Extraction de courant

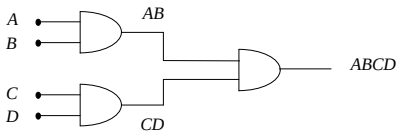
De façon à éviter l'action des signaux parasites (fil ≡ antenne !) les entrées non utilisées d'un circuit à entrées multiples doivent être polarisées, soit en les reliant aux autres, soit en les connectant à la source d'alimentation ou à la masse suivant le cas. Ceci est particulièrement important dans le cas des circuits ayant des courant d'entrée très faible comme les circuits MOS. (Une entrée enl'air a un état indéterminé qui prend en général la valeur 1 par effet d'antenne).

2.3.5. Circuits expansibles, ET et OU câblés

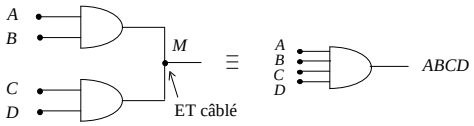
Soit un circuit ET à 2 entrées effectuant l'opération : $S = A\,B$

Pour réaliser un circuit ET à 4 entrées qui réaliserait : $S = A\,B\,C\,D$

on peut songer à utiliser 3 circuits ET à 2 entrées en faisant le produit des 2 produits partiels *AB* et *CD* :.



Dans certains cas on peut associer plus directement les sorties des 2 circuits ET sans dommage pour les circuits. Si ceci est possible, les circuits sont qualifiés *d'expansibles* :



On a bien un ET à 4 entrées :

. si $AB = 0$ et $CD = 0 \rightarrow M = 0$ (pas de court-circuit)
. si $AB = 1$ et $CD = 1 \rightarrow M = 1$ (pas de court-circuit)
. si $AB = 0$ et $CD = 1 \rightarrow M = 0$ (court-circuit entre 0 Volt et 5 Volts (en TTL) → le résultat est 0 Volt, soit 0 logique)
. si $AB = 1$ et $CD = 0 \rightarrow M = 0$ (court-circuit entre 0 Volt et 5 Volts (en TTL) → le résultat est 0 Volt, soit 0 logique)

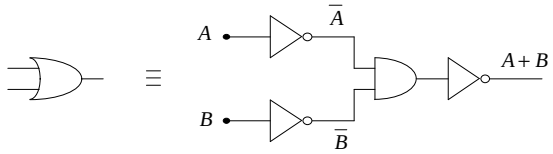
La jonction au point M est appelée « ET câblé ».

2.4. Les familles logiques

Toutes les fonctions booléennes peuvent être construites à l'aide des trois opérateurs fondamentaux ET, OU et complément. Ce groupe de trois opérateurs forme ce que l'on appelle un *système logique complet*. La matérialisation des fonctions logiques nécessite donc de pouvoir réaliser des systèmes physiques remplissant ces trois fonctions.
Un système logique complet permettant la construction de toute fonction peut cependant être réalisé en utilisant un nombre plus faible de structures de base. Par exemple, le groupe des deux fonctions ET et complément constitue un système logique complet. En effet, la fonction OU peut être reconstituée à partir de ces deux fonctions seulement comme le montrent les théorèmes de De Morgan :

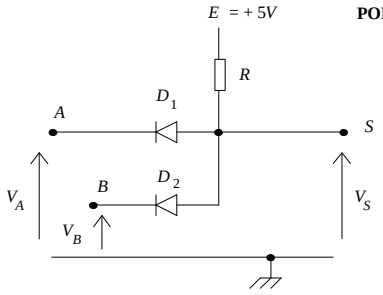
$$A + B = \overline{\left(\overline{A + B}\right)} = \overline{\overline{A} \cdot \overline{B}}$$

soit :



2.4.1. Circuits logiques à diodes

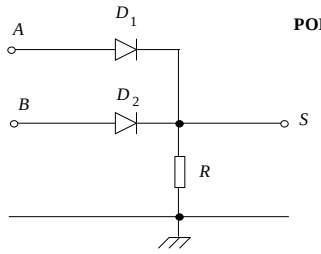
Soient, deux diodes et une résistance connectées comme le montre la figure ci-après. Admettons d'abord que les diodes sont parfaites, de résistance nulle dans le sens passant : $(R_d = 0 \text{ pour } V > 0)$.
Si l'une ou l'autre des entrées *A* ou *B* est reliée à la masse $(V = 0)$, la sortie $V_S = 0$.
La sortie V_S n'est au potentiel haut (*S* = 1 en logique positive) que si V_A et V_B sont au potentiel haut. On a réalisé le produit logique *S* = *A* . *B* :



PORTE ET (Diodes)

V_A	V_B	V_S	
0	+ *E*	0	D_2 bloquée mais D_1 passante → *S* à la masse
+ *E*	0	0	D_1 bloquée D_2 passante → *S* à la masse
0	0	0	D_1 D_2 passantes → *S* à la masse
+ *E*	+ *E*	+ *E*	D_1 D_2 bloquées → V_S = + *E* (pas de courant dans *R*)

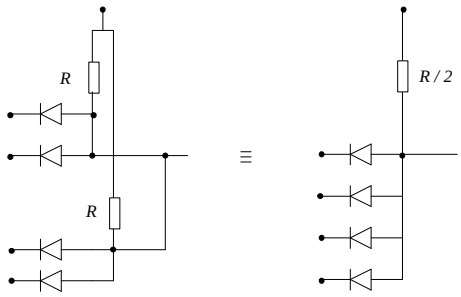
De même avec le circuit de la figure ci-dessous, la sortie ne vaut + *E* que si *A* ou *B* valent 1. (Somme logique *A* + *B*) :



PORTE OU (Diodes)

R: résistance de tirage (pull-down/pull-up) de l'ordre de 10 kΩ

Les portes à diodes *sont expansibles*, en effet :

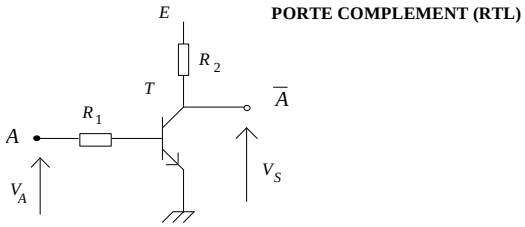


Compatiblité des portes ET *et* OU *à diodes*

La mise en série de portes de type différent pose un certain nombre de difficultés liées au fait que les portes ET sont à extraction de courant alors que les portes OU sont à injection de courant. Il faut leur adjoindre un élément actif du type transistor qui peut par contre à lui seul constituer un système complet comme dans la famille RTL.

2.4.2. La famille RTL (Résistance Transistor Logic)

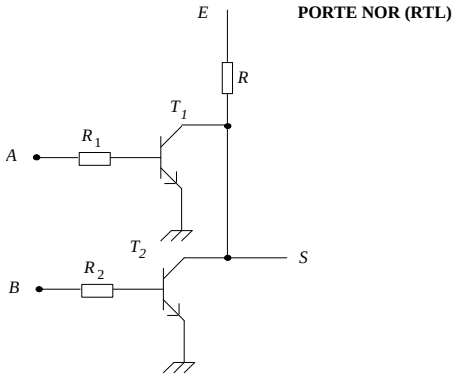
Le transistor permet très simplement d'obtenir le complément d'une variable logique.



Si $V_A = 0$ $(A = 0)$ le transistor *T* est bloqué et $V_S = +E$ $(S = 1)$.

Si $V_A = E$, sous réserve que la condition de saturation : $R_2 > \frac{R_1}{\beta}$ soit satisfaite, *T* est saturé : $V_S = 0,\ S = 0$.

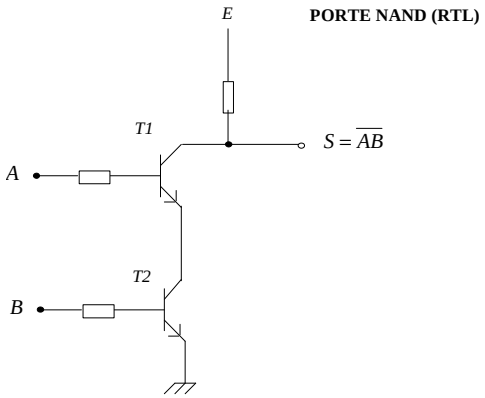
On a donc réalisé le complément $S = \overline{A}$. Un transistor associé à des résistances (d'où le nom de ce type de circuits) permet de réaliser des opérations ET et OU (ou plus exactement des ET et OU complémentés soit des NOR et NAND):



Si $V_A = V_B = 0$ les deux transistors sont bloqués. Si l'une des tensions d'entrée vaut *+E*, le transistor correspondant se sature : $V_S = 0$. D'où la table de vérité, correspondant à la fonction NOR : $S = \overline{A + B}$:

A	B	S
0	0	1
0	1	0
1	0	0
1	1	0

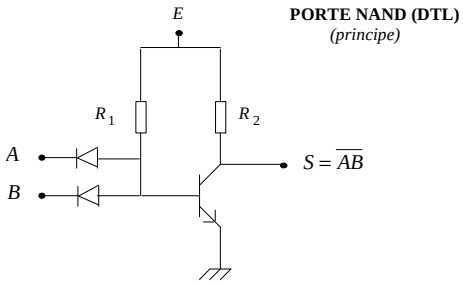
Pour réaliser la porte ET, on peut utiliser deux inverseurs et un NOR conformément à l'expression $AB = \overline{\overline{A} + \overline{B}}$.
Pour la porte NAND, on peut faire appel au montage direct de la figure suivante, mais qui est peu utilisé car les entrées sont mal découplées entre elles :



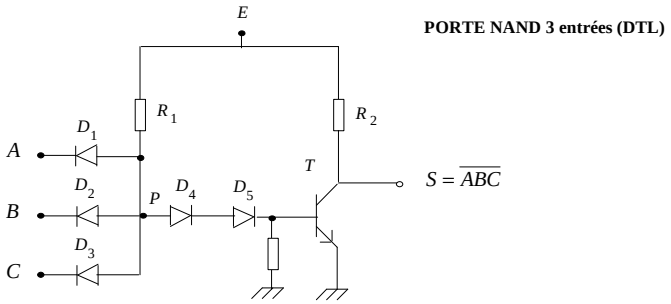
La structure de base en RTL est la porte NOR qui constitue à elle seule, comme on l'a vu plus haut, un système logique complet. On remarque enfin que la RTL est une logique à injection de courant.

2.4.3. Logique DTL (Diode Transistor Logic)

Elle peut être considérée comme l'association d'une logique à diodes et d'un transistor inverseur. L'ensemble constitue alors un circuit NAND qui à lui seul forme un système logique complet. Le schéma de principe est présenté sur la figure suivante :



Si $V_A = V_B = +E$, les deux diodes sont bloquées et *T* est saturé par le courant base traversant $R_1 \cdot (V_S = 0)$. Si $V_A = 0$ la diode d'entrée parfaite bloque le transistor $(V_S = E)$. En réalité si le point *A* est à la masse, l'anode de la diode correspondante est un potentiel voisin de 0.6 Volt qui est aussi le seuil de conduction du transistor.
Le montage ne peut fonctionner que si le V_{BE} limite de conduction du transistor est plus élevé que la tension de conduction de la diode. Cela pourrait se faire avec des diodes au germanium associé à un transistor silicium (solution incompatible avec l'intégration du circuit). Une solution plus efficace consiste à utiliser des diodes remontant le seuil de conduction du transistor. Le circuit réel est représenté sur la figure suivante. Les deux diodes D_4 et D_5 remontent au voisinage de 1.8 Volts la tension en P nécessaire à la conduction de *T*. Alors : si $V_A = 0$, $V_P = 0.6$ Volt $\rightarrow T$ est bloqué, $V_S = +E$:



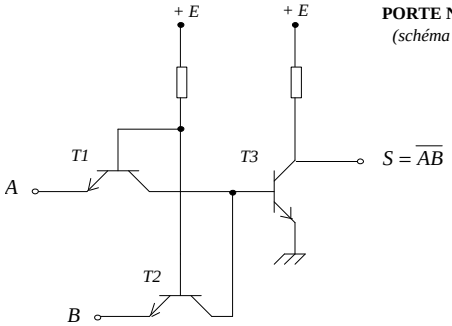
Il n'existe pas de circuit spécifiquement NOR en DTL.

La logique DTL est une *logique à extraction de courant* qui n'est donc pas compatible avec la RTL.
Comme pour la logique à diodes les portes sont expansibles (on diminue la résistance de charge du transistor).

2.4.4. La logique TTL (Transistor Transistor Logic) (famille la plus répandue)

Elle ne diffère de la DTL que par le remplacement du réseau de diodes d'entrée par un transistor spécial multi-émetteurs. Cest une logique qui ne se conçoit qu'en circuit intégrés, elle est de loin la plus courante actuellement (série 54/74). Comme en DTL, le circuit de base est une porte NAND.

La figure suivante représente le montage de base du circuit d'entrée, les diodes sont remplacées par les jonctions EB des transistors. Les deux transistors d'entrée ont leurs bases et collecteurs reliés.



PORTE NAND (TTL)
(schéma de base)

T1 et T2 sont toujours saturés

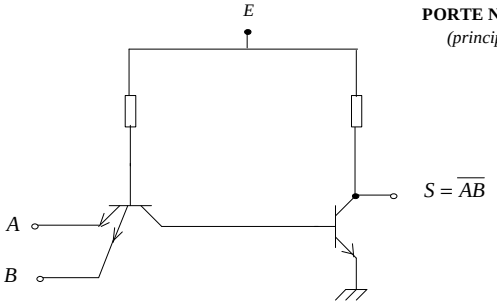
A=0 B=0 -> 0 sur la base de T3 -> T3 bloqué -> S=1

A=0 B=1 -> court-circuit 0-1 -> 0 sur la base de T3 -> T3 bloqué -> S=1

A=1 B=0 -> court-circuit 0-1 -> 0 sur la base de T3 -> T3 bloqué -> S=1

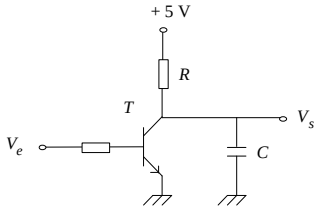
A=1 B=1 -> 1 sur la base de T3 -> T3 saturé -> S=0

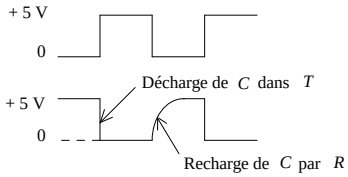
Lors de leur fabrication cettre liaison peut aller jusqu'à la fusion totale conduisant à un transistor multi-émetteur qui réalise la fonction ET. D'où le circuit d'entrée de la figure suivante :



PORTE NAND (TTL)
(principe)

Pour augmenter les performances, le circuit de sortie n'est pas un simple transistor. En effet un tel montage est très mal adapté à l'attaque de charges capacitives. Considérons en effet la figure suivante :

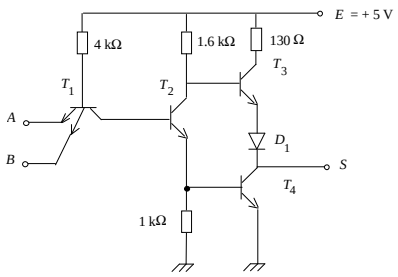


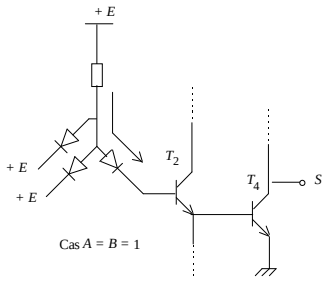


Lorsque le niveau de sortie passe de + *E* à 0, le courant de décharge de *C* traverse le transistor qui en régime de saturation se comporte presque comme un commutateur parfait. Lorsque le niveau de sortie passe de 0 à + *E* ce qui correspond au bloquage du transistor, la charge de *C* doit se faire grâce à un courant traversant R, donc avec une constante de temps *RC* appréciable. Pour diminuer le temps de montée il faut diminuer *R* ce qui augmente proportionnellement la consommation du circuit.

Pour augmenter la vitesse on fait appel à un deuxième transistor monté à la place de *R* qui en se saturant branche directement *C* à + *E*. Le montage présente quelques analogies avec le push-pull :

Porte NAND (TTL) 7400





Si $V_A = V_B = +5\,\text{V}$, les diodes émetteur-base de T_1 sont bloquées, par contre la diode base-collecteur est conductrice. Un courant circule et T_2 et T_4 sont saturés. T_2 étant saturé, sa tension collecteur est égale à sa tension émetteur soit environ 0.6 Volt. Or T_3 ne peut conduire (à cause de D_1) que si sa base est portée à environ 1.2 V; il est donc bloqué. Alors $V_S = 0,\ S = 0$.
Annulons l'une des tensions *A* ou *B* (ou les deux). La résistance de 4 kΩ assure la saturation de T_1 ce qui amène à zéro le potentiel base de T_2 donc bloque T_2 et aussi T_4 . T_3 se trouve alors saturé grâce à la résistance de 1.6 kΩ reliant sa base à + *E*, la sortie est alors au niveau haut. Ce circuit réalise donc bien la fonction NAND : $S = \overline{AB}$.

Le montage constitué par les deux transistors de sortie T_3 , T_4 est appelé « *totem pole* », (pour chacun des 2 états logiques on a : T_3 bloqué et T_4 saturé, ou l'inverse). Il permet des transitions rapides du niveau de sortie même sur charge capacitive. Le temps de transit est couramment de 10 ns.

Comme la logique DTL, la logique TTL est une *logique à extraction de courant*.

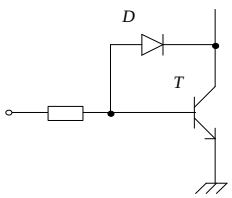
Tolérance sur les niveaux TTL

Tension d'alimentation	*Vcc*	5 V ± 0.5 V	
Tension maxi d'entrée pour un niveau bas	V_{IL}	0.8 V	L : Low
Tension mini d'entrée pour un niveau haut	V_{IH}	2 V	H : High
Tension maxi de sortie pour un niveau bas	V_{OL}	0.4 V	I : Input
Tension mini de sortie pour un niveau haut	V_{OH}	2.4 V	O : Output

Puissance moyenne absorbée par porte :	~ 10 mW
Courant moyen par porte :	~ qq. mA

TTL Schottky

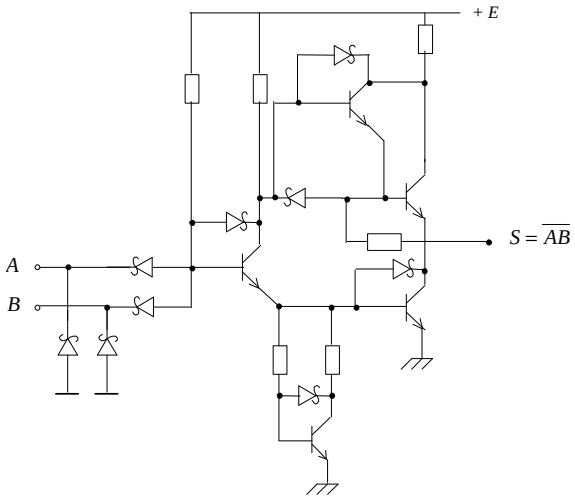
Dans la famille précédente les transistors travaillent en commutation c'est à dire qu'ils sont parfois saturés. Or un transistor saturé stocke des charges dans sa base qui doivent ensuite être évacuées. Ceci limite fortement la vitesse de commutation.
Pour augmenter la vitesse, il faut éviter la saturation, ceci peut se faire en plaçant une diode en parallèle sur l'espace base-collecteur, de façon à maintenir le collecteur à un potentiel très légèrement inférieur à celui de la base :
(*T* ne peut pas se saturer car *D* conduirait, ce qui amènerait la base à +0.15 volt bloquant *T*)



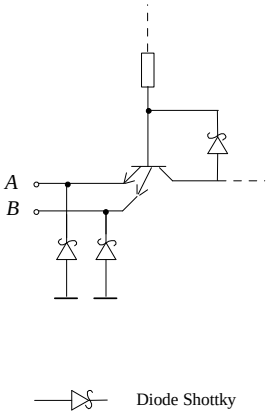
En réalité le gain de vitesse n'est pas grand car la diode elle même stocke des charges. La solution est trouvée en remplaçant *D* par une diode Schottky. Une telle diode est constituée par un contatct métal semi-conducteur. Sa tension de conduction est de l'ordre de 0.4 Volt et elle est très rapide car le phénomène de stockage est très réduit.

Dans un circuit TTL le transistor multi-émetteur d'entrée peut être également de type Schottky (c'est à dire avec une diode Schottky en parallèle) ou remplacé par des diodes Schottky comme en DTL :

PORTE NAND (TTL S)



ou avec entrée multi-émetteur :



Le gain en vitesse est important, les temps de transit étant de quelques nanosecondes seulement.

Variantes du circuit de sortie

a) La sortie totem-pole permet d'intéressantes performances en vitesse mais interdit le ET et le OU câblés, l'interconnexion directe des sorties peut en effet conduire à la destruction des circuits. Pour remédier à cet inconvénient, deux solutions ont été retenues : les sorties *open collector* et *tri-state*.
En sortie totem pole, la charge est fixée par construction. 2 transitors en alternat de commutation (un est bloqué quand l'autre est saturé) en sortie augmentent la rapidité du circuit.

Sortie TOTEM POLE

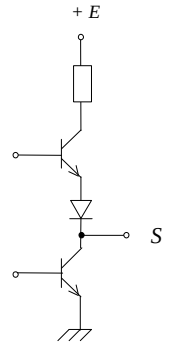
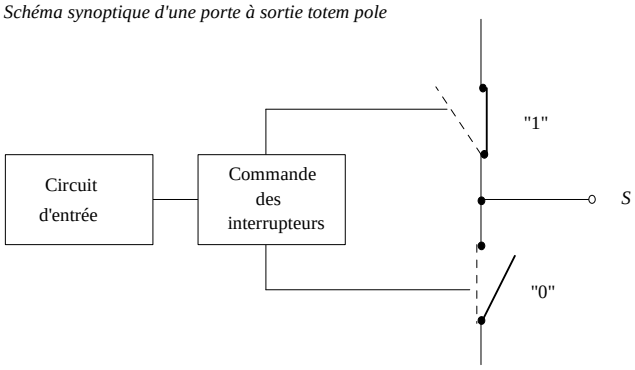
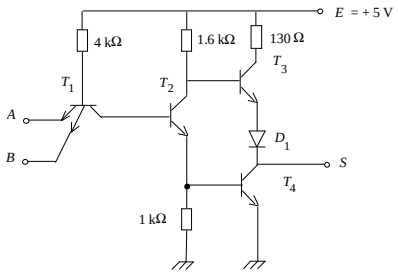
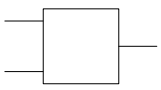


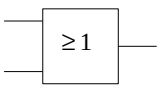
Schéma synoptique d'une porte à sortie totem pole



Porte NAND (TTL) 7400



Symbole : (c'est le symbole par défaut) 

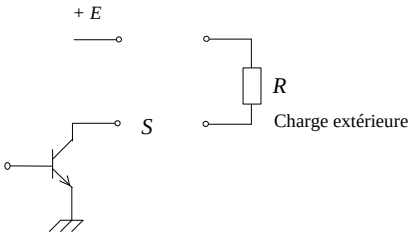
Exemple: Porte OU totem pole 

b) Sortie open collector (collecteur ouvert)

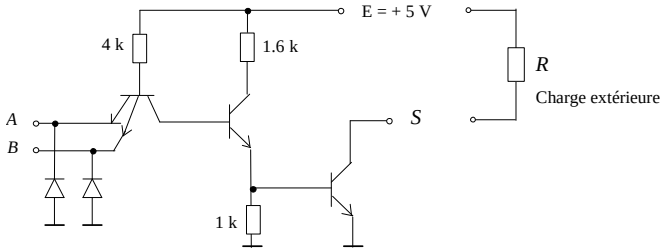
Le totem pole est supprimé et remplacé par un seul transistor dont la résistance de collecteur n'est pas intégrée. Elle doit être mise en place par l'utilisateur (en fonction de son problème).
Le collecteur du transistor de sortie du circuit logique n'est pas connecté à une alimentation dans le circuit. C'est à l'utilisateur de placer la charge la mieux adaptée selon la sortance désirée.
Le OU et le ET câblés deviennent ainsi possible. De plus certains circuits sont prévus avec un transistor de sortie pouvant supporter une tension de plusieurs dizaines de volts et sont précieux comme générateurs d'impulsions de grande amplitude.

La sortie *S* peut avoir 2 états (0 ou 1) selon que le transistor de sortie est respectivement saturé ou bloqué.

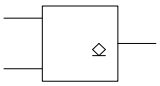
Sortie OPEN COLLECTOR



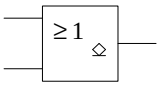
Circuit "open collector" (7426) NAND 2 entrées



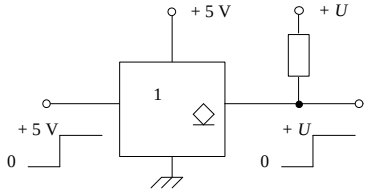
Symbole :



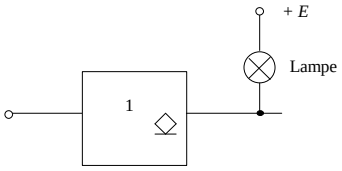
Exemple: Porte OU open collector



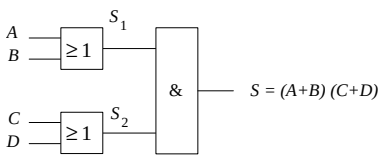
Applications : *- Changement de niveau logique (de TTL +5 Volts à + U Volts)*

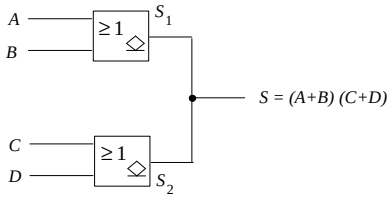


- Commande de charge importante

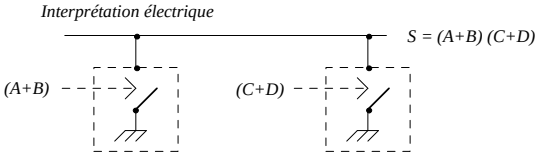


- Réalisation d'une fonction câblée



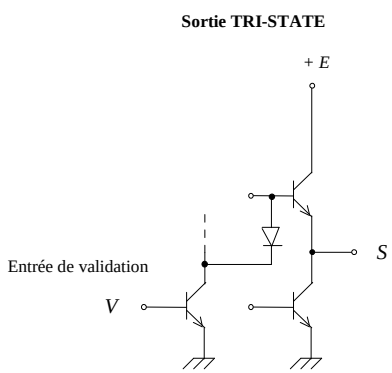


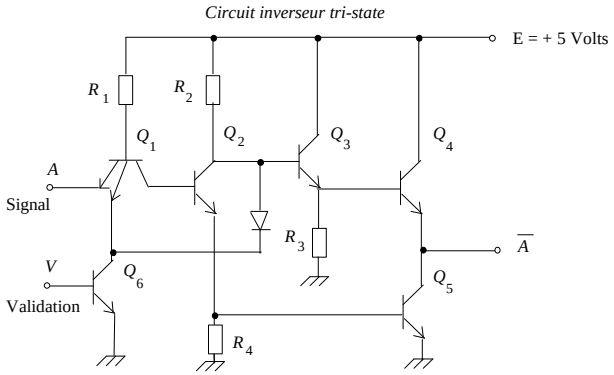
Interprétation électrique



c) Sortie tri-state (3 états)

La charge est fixée par construction. Les 2 transitors en alternat de commutation du totem pole sont désolidarisés pour donner en sortie 3 états possibles : les 2 états logiques 0 et 1, et le 3ème état (haute impédance) obtenu lorsque les 2 transistors de sortie sont bloqués.
La sortie peut donc se présenter sous les 3 états : 0, 1 et l'état haute impédance (circuit déconnecté).





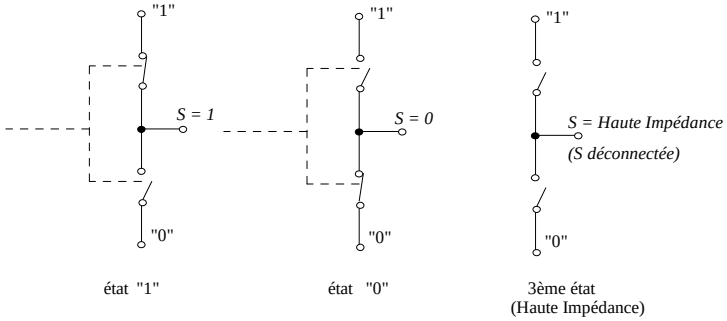
Sur le schéma de l'inverseur 3 états, on peut commenter le fonctionnement :
Si $V = 0$, Q_6 est bloqué, le système fonctionne comme un circuit TTL classique :

Si $A = 0$, Q_1 conduit, la base de Q_2 est au zéro donc Q_2 est bloqué ainsi que Q_5, pendant que Q_3 est conducteur ainsi que Q_1, donc $S = \overline{A} = 1$.

Si $V = 1$, Q_6 est saturé donc Q_1 également et comme plus haut Q_2 et Q_6 sont bloqués, mais par la diode *D* reliée à la masse Q_3 est maintenu bloqué malgré le courant dans R_2, Q_4 se trouve donc également bloqué. N'importe quel potentiel peut être imposé en *S* par un circuit extérieur sans détériorer le circuit.

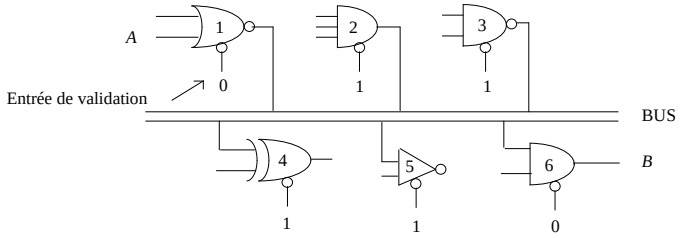
Il est ainsi possible de relier plusieurs sorties *à condition qu'un seul circuit soit validé à la fois*.
(OU et ET câblés possibles à condition qu'un seul circuit soit validé à la fois).

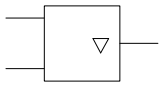
Dans la famille TTL, une sortie passe dans l'état haute impédance en désolidarisant les 2 interrupteurs du *totem pole* :

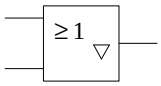


Ce type de circuit est très utilisé dans les systèmes logiques complexes dans lesquels les informations circulent sur des lignes communes auxquelles sont reliées de nombreux circuits. Ce sont des *BUS*.

On voit sur l'exemple de la figure suivante que pour faire circuler l'information de *A* à *B* il suffit de valider seulement les portes 1 et 6, toutes les autres portes devant être « *inhibées* » (déconnectées par état haute impédance) sous peine de court-circuit destructeur: (les portes 1 à 6 pouvant être des circuits drivers d'entités informatiques par ex., mémoire, périphérique d'ordinateur ...).
(driver ≡ pilote ≡ circuit de commande, de contrôle, d'interface)



Symbole : 

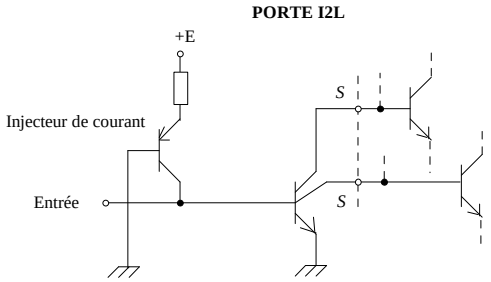
Exemple: Porte OU tri-state 

Remarque :

Pour éviter que plus de 2 entités soient connectées simultanément au bus (ce qui entraînerait des courts-circuits), un circuit programmable spécialisé (contrôleur de bus) gère ces signaux de validation.

2.4.5. La logique I^2L

L' I^2L est une technologie bipolaire rapide utilisée exclusivement dans les circuit intégrés très complexes du type microprocesseurs. La structure fondamentale est représentée sur la figure suivante (avec un transistor multi-collecteur) :



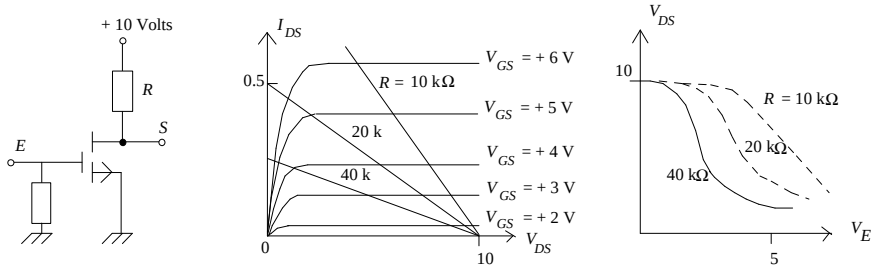
2.4.6. Les familles ECL

Dans les montages précédents les transistors fonctionnent au blocage et à la saturation, or on sait qu'un transistor saturé accumule dans sa base une charge qui doit être éliminée pour obtenir le blocage, ce qui prend un certain temps. Pour augmenter la vitesse de fonctionnement, des familles logiques où les transistors ne sont jamais saturés ont été développées, c'est la cas de l'ECL (logique à émetteurs couplés) de *Motorola*.
Pour permettre la mise à la masse des collecteurs de l'étage de sortie, l'alimentation est négative (- 5 Volts) mais pour faciliter la liaison avec d'autres logiques une alimentation positive est possible. Les niveaux logiques sont : niveau haut (1) codé par -0.8 Volt et niveau bas (0) codé par -1.8 Volts.

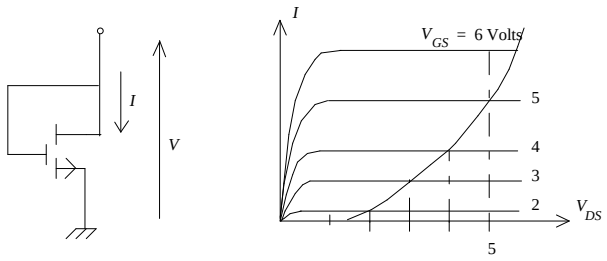
2.4.7. Les familles MOS

Le Transistor à Effet de Champ à jontion (FET) n'est pas utilisé pour construire des circuits logiques, il n'en est pas de même du transistor MOS, Transistor à Effet de Champ à grille isolée. Pour les transistors MOS construits actuellement, la tension de seuil peut être inférieure à 2 Volts, ce qui permet d'utiliser ces composants avec des tensions d'alimentation de 3 Volts seulement.

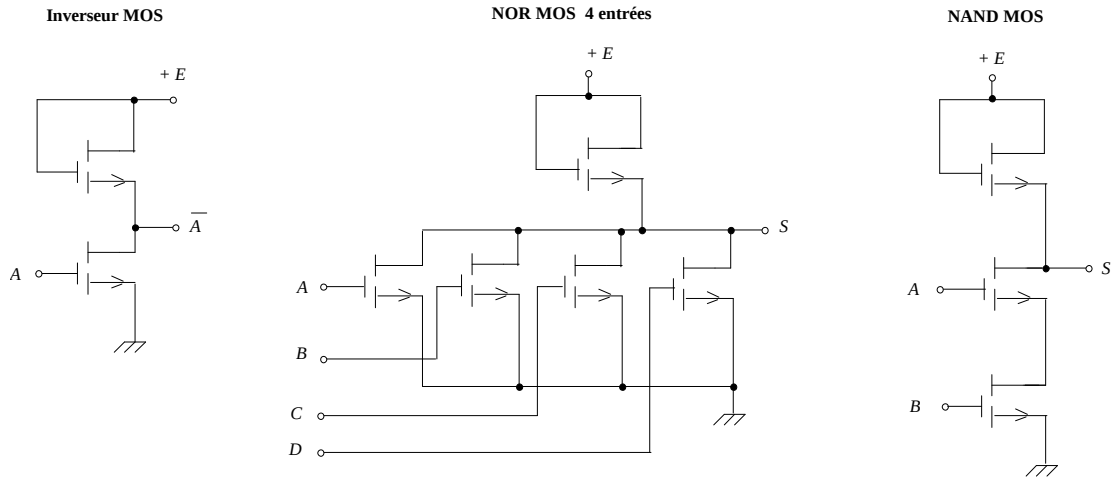
La figure suivante représente un inverseur MOS. Sa structure est analogue à celle de l'inverseur à transistor bipolaire, sa caractéristique de transfert peut être tracée à partir du réseau de caractéristiques du MOS pour une valeur donnée de la résistance de charge. La transmission est d'autant plus brutale que la résistance est élevée. On voit que le système est compatible avec un niveau logique bas inférieur à 3 Volts, et haut supérieur à 7 Volts (pour 10 Volts d'alimentation).



Or en circuit intégré, une résistance occupe d'autant plus de surface sur la "*puce*" que sa valeur est importante. On a donc cherché à remplacer la résistance de charge par un second transistor MOS. Considérons un MOS canal N dont la grille est reliée au drain, il constitue un dipôle dont la caractéristique est tracée sur la figure suivante. C'est à un décalage de tension près, celle d'une résistance qui peut être utilisée come charge dans le montage inverseur.



La figure suivante représente le montage fondamental de l'inverseur MOS dans lequel toute résistance a été bannie. Nous ne détaillons pas ici les nombreuses variantes technologiques mises au point depuis quelques années et qui sont en constante évolution, (grille en aluminium et isolement par de la silice, grille en silicium poly ou monocristallin, isolement par du nitrure de silicium ayant une constante diélectrique élevée etc...).

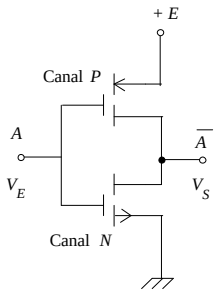


Une particularité des MOS liée à leur impédance d'entrée très élevée est la présence d'une capacité grille-substrat qui peut être utilisée comme élément de mémoire et limite les performances en vitesse.

2.4.8. La famille CMOS (autre grande famille avec la famille TTL)
CMOS pour MOS Complémentaires : MOS utilisés par paires : MOS canal *N*; MOS canal *P*.

L'emploi simultané de MOS complémentaires permet de réaliser des circuits dont la consommation au repos est particulièrement basse. La firme américaine RCA s'est spécialisée dans cette technique et commercialise ces circuits logiques (série 4000/40000).

Le circuit fondamental est l'inverseur ci-dessous.



Lorsque V_E # +*E*, niveau haut, le MOS-*N* ayant sa grille positive est conducteur. Par contre le MOS-*P* est bloqué. Donc V_S est petit (V_S ~ 0) mais le courant consommé est nul, M_2 étant bloqué.

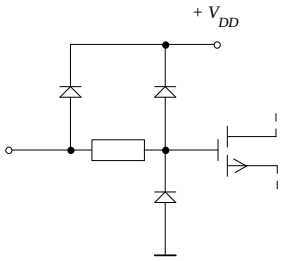
Lorsque V_E # 0 niveau bas, le MOS-*N* est bloqué (il s'agit toujours de MOS à enrichissement ayant un I_{DSS} nul). Par contre, M_2 de type *P* est conducteur et V_S ~ *E*. Là encore M_1 étant bloqué, le courant consommé par la cellule est nul.

Les deux transistors ne sont pas simultanément conducteurs, le circuit ne consomme donc rien à l'état stable. Une consommation apparaît seulement en régime transitoire car il faut charger et décharger les capacités de structure. La consommation typique à vitesse moyenne peut être cent fois inférieure à celle de la cellule identique à transistors à jonctions mais la vitesse limite est actuellement plus faible, typiquement 10 MHz contre plus de 500 MHz pour des ECL.

Tension d'alimentation : ≠ à TTL elle peut être de 3 à 18 Volts (série 4000)
(les nouvelles générations plus performantes n'autorisent que 2 à 6 Volts).
La puissance consommée est << TTL : de l'ordre de 0.1 mW → courant très faible < 1 mA.
Les tolérances sur les niveaux logiques sont du même ordre qu'en TTL.

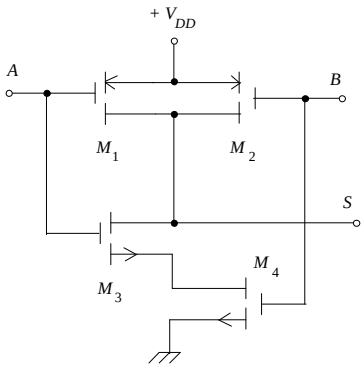
Un des problèmes à maîtriser a été la protection des entrées contre les surtensions d'origine statique, la couche d'oxyde des grilles est en effet très fragile. La solution a été trouvée en intégrant des diodes au niveau des entrées. Actuellement ce système fonctionne bien et enlève tout souci à l'utilisateur concernant des manipulations destructrices.

Circuit de protection d'entrée



La figure suivante représente un NAND à deux entrées.

PORTE NAND (CMOS)



Si l'une des entrées est au zéro le MOS correspondant M_1 ou M_2 de type *P* est conducteur amenant *S* au $+V_{DD}$.
Si au contraire *A* et *B* sont à $+V_{DD}$, M_1 et M_2 sont bloqués mais M_3, M_4 conducteurs, fixent *S* au zéro.

Un MOS n'ayant pas de tension d'offset les niveaux de sortie (sans charge) sont rigoureusement $+V_{DD}$ et zéro, les impédances de sortie étant les résistances des canaux, ces résistances sont de l'ordre du kΩ.

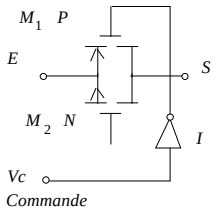
Le courant d'entrée est toujours très faible, typiquement 10 pA, le courant susceptible d'être délivré en sortie est au maximum de l'ordre du milliampère. Au moins en fonctionnement lent, la sortance est donc très grande. Les constructeurs l'annoncent supérieure à 50.

Comme pour la famille TTL, il y a 3 variantes pour le circuit de sortie : totem pole, open drain, tri-state.

Les principaux circuits de la série 4000 sont les suivants :

4001 Quadruple NOR à deux entrées,
4011 Quadruple NAND à deux entrées,
4009 Six inverseurs (Buffer inverseur),
4010 Six amplis non inverseurs (Buffer),
4013 Double bascule *D*,
4027 Double bascule *JK*,
4042 Quadruple bascule *D* (latche),
4017 Décade DCB à 10 sorties décodées,
etc ...

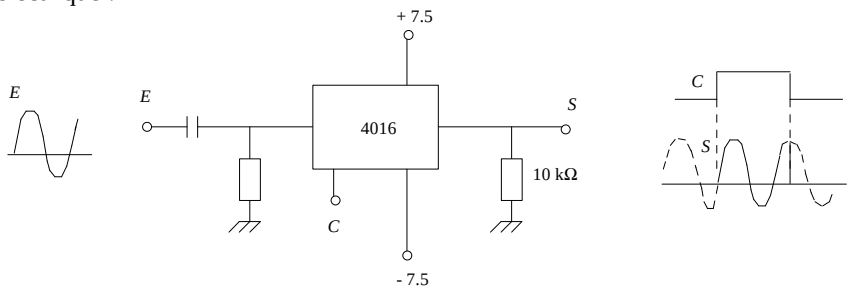
Soit la porte analogique (4016) représentée ci dessous :



Si $V_C = 1$, M_2 canal *N* est conducteur ainsi que M_1 qui, grâce à l'inverseur *I* voit sa grille portée au 0 ; les deux MOS se comportent alors comme leur résistance de conduction ~ 200 Ω et $V_{out} = V_{in}$.
Si au contraire $V_C = 0$, M_1 et M_2 sont bloqués, leur résistance de fuite étant supérieure à $10^{11}\Omega$.

En plaçant un système de ce type en sortie d'un circuit logique, on obtient le même résultat qu'avec le montage tri-state de la TTL.

Le 4016 peut être utilisé avec des signaux d'entrée analogiques, son comportement est celui d'un interrupteur mécanique à commande électrique :



Les circuits CMOS sont de plus en plus utilisés grâce à leur souplesse d'emploi :

- niveaux de sortie très bien définis.
- grande immunité au bruit
  (mais en contre partie les impédances d'entrée favorisent la réception de signaux parasites rayonnés).
- possibilité de fonctionnement dans une large plage de tensions d'alimentation.
- très faible consommation.
- Une dernière propriété des CMOS, liée à leur impédance d'entrée et comportement en sortie et la possibilité de les utiliser dans les configurations où ils fonctionnent de façon pseudo-linéaire : amplificateur, oscillateurs, etc ...

Avec les séries 4000 et 40000, on trouve aussi la série 74C dont les circuts sont compatibles broche à broche avec ceux portant le numéro correspondant en logique TTL.

2.4.9. Les interfaces entre familles

Pour des raisons d'incompatibilité entre les familles logiques, tous les circuits logiques connectés d'un montage doivent être de la même famille; dans le cas contraire, il faut en outre prévoir des circuits d'interfaçage.

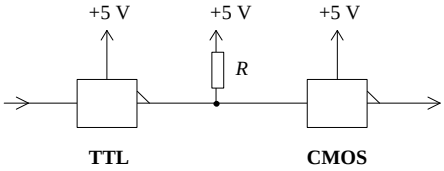
Ce sont des circuits permettant l'association de circuits logiques appartenant à des familles logiques différentes. Les cas les plus souvent rencontrés sont :

- l'attaque d'une logique lente, le plus souvent TTL, par une logique ultra rapide (ECL). L'inverse étant sans intérêt.
- une association de circuits TTL et CMOS.

2.4.9.1. Interface CMOS - TTL

Il est évident que l'alimentation doit se faire en + 5 Volts à cause de la TTL.

2.4.9.1.1. Attaque d'un circuit CMOS par un circuit TTL



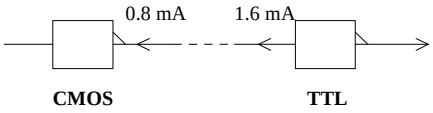
Une porte TTL totem pole fournit en sortie :

- au plus 0.4 Volt au niveau 0
- au moins 3.6 Volts au niveau 1

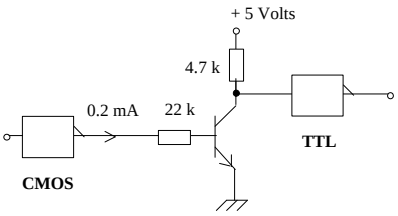
Or sous 5 Volts, il faut pour le CMOS au plus 1.5 Volts au niveau 0 et au moins 3.5 Volts au niveau 1.
En conséquence, la TTL pilote sans problème le CMOS au niveau 0. C'est par contre un peu juste au niveau 1.
On utilise alors une résistance *R* dite de *pull up* (*R* de l'odre de 10 kΩ) qui remonte le niveau haut de la TTL.

2.4.9.1.2. Attaque d'une entrée TTL par une sortie CMOS

Au niveau 1 il n'y a pas de problème car l'entrée TTL se contente d'un courant faible.
Il n'en est pas de même au niveau 0 : pour une tension de 0.8 Volt max, il faut extraire d'une entrée TTL un courant de 1.6 mA. Or une porte CMOS peut tout juste accepter 0.8 mA pour cette valeur de tension. La liaison directe est donc impossible.



On peut alors utiliser un circuit d'interface spécialisé ou même un transistor intermédiaire (impliquant alors une inversion).

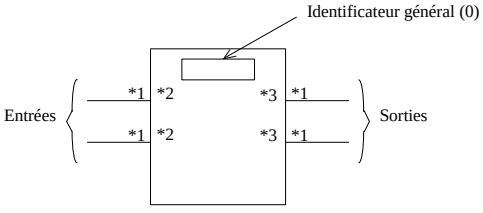


2.4.9.2. Interface ECL - TTL

La difficulté est différente suivant que l'ECL est alimenté entre 0 et + 5 Volts ou entre - 5 Volts et 0. Dans le cas où une association avec des circuits TTL est prévue, la première solution est généralement retenue.
Des circuits spécialisés ou des montages adaptateurs sont proposés par le constructeur.

2.5. Symbolisme des opérateurs logiques

2.5.1. Forme des symboles

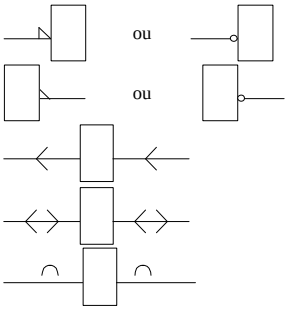


*x : positions possibles pour les spécifications x particulières à chaque entrée ou sortie.

2.5.2. Identificateur général (0)

Symbole	*Description*
&	Fonction ET
≥1	Fonction OU
=1	Fonction OU Exclusif
=	Fonction Coïncidence
2k	Le nombre d'entrées activées doit être pair (pour activer la sortie)
2k+1	Le nombre d'entrées activées doit être impair (pour activer la sortie)
1	L'entrée unique doit être active (pour activer la sortie)
▷	Tampon (amplificateur). Le sens du triangle indique le sens de propagation du signal
⊓	Elément présentant un hystérésis (trigger de Schmitt)
X/Y	Codeur, convertisseur (DEC/BCD, BIN/7 segments)
MUX	Multiplexeur
MUX ou DX	Démultiplexeur
Σ	Additionneur
P-Q	Soustracteur (Comparateur numérique)
CPG	Générateur de retenue anticipée
π	Multiplieur
COMP	Comparateur (amplitude analogique)
ALU	Unité arithmétique et logique
⊓	Monostable programmable
1⊓	Monostable
G⊓⊓	Elément astable (la forme d'onde est optionnelle)
!G⊓⊓	Oscillateur à démarrage synchrone
G!⊓⊓	Oscillateur astable à arrêt commandé
SRG m	Registre à décalage (m = nombre de bits)
CTR m	Compteur m bits (cycle de 2^m états)
CTR DIV m	Compteur de cycle = m
ROM	Mémoire morte *(Read Only Memory)*
RAM	Mémoire vive *(Random Access Memory)*
FIFO	Mémoire vive à rangement séquentiel file d'attente *(First In First Out)*

2.5.3. Symbole externe au contour du circuit (1)



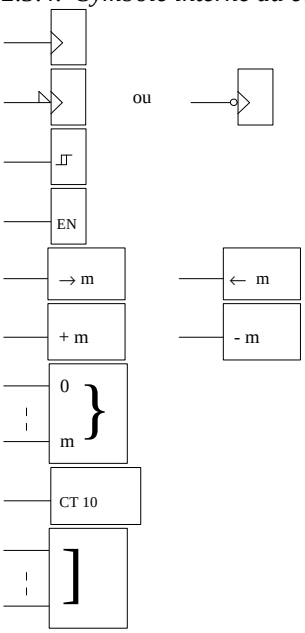
Complément logique en entrée (la flêche indique aussi le sens du signal)

Complément logique en sortie (la flêche indique aussi le sens du signal)

Flux de données de la droite vers la gauche
(implicitement, il est de la gauche vers la droite)
Flux bidirectionnel

Entrée (sortie) analogique

2.5.4. Symbole interne au contour du circuit pour les entrées (2)



Entrée active sur front montant
(en l'absence du triangle interne au circuit l'entrée est active sur niveau haut)

Entrée active sur front descendant)
(en l'absence du triangle interne au circuit l'entrée est active sur niveau bas)

Entrée possédant un hystérésis électrique

Entrée de validation (ENABLE)

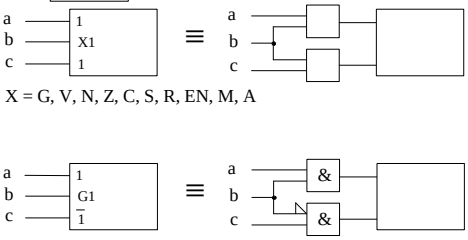
Entrée de décalage d'un registre; La flêche indique le sens
m indique le nombre de décalage effectués

Entrée de comptage (+) ou de décomptage (-); m précise l'incrément

Entrées groupées, pondérées de 0 à m
Les combinaisons sont repérées de 0 à 2^m -1

Entrée de chargement d'un compteur à la valeur indiquée (ici 10)

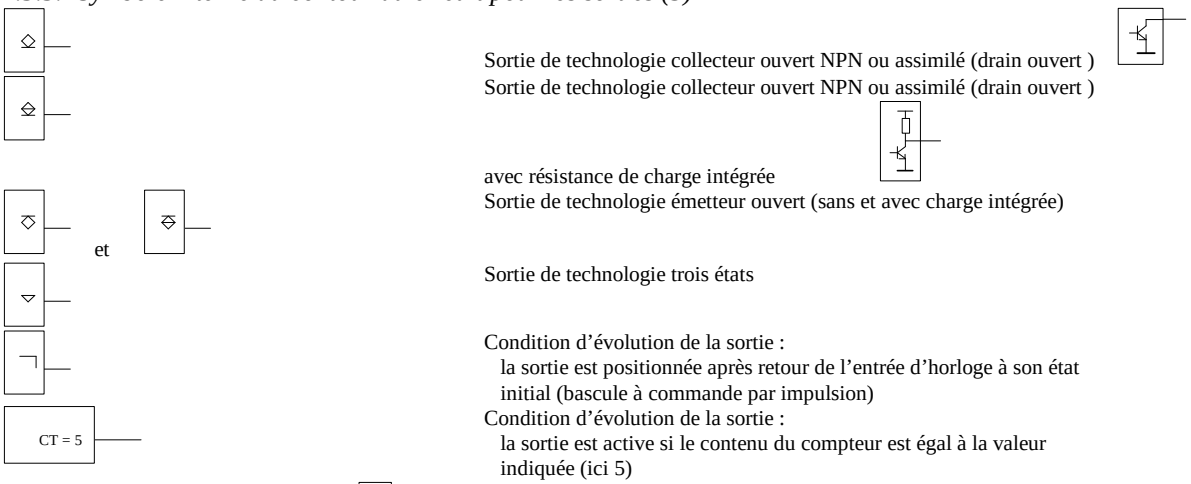
Entrées groupées : elles réalisent indépendamment la même fonction



X = G, V, N, Z, C, S, R, EN, M, A

Dépendance d'une entrée envers une autre entrée. Ici, les entrées a et c sont dépendantes de l'entrée b (même numéro) par la fonction X :
G : ET
V : OU
N : OU Exclusif
Z : relation d'interconnexion
C : relation de contrôle (horloge)
S : SET (Mise à 1)
R : RESET (Mise à 0)
EN : ENABLE (autorisation)
M : Sélection d'un mode de fonctionnement
A : Sélection d'une adresse

2.5.5. Symbole interne au contour du circuit pour les sorties (3)



Sortie de technologie collecteur ouvert NPN ou assimilé (drain ouvert)
Sortie de technologie collecteur ouvert NPN ou assimilé (drain ouvert)
avec résistance de charge intégrée
Sortie de technologie émetteur ouvert (sans et avec charge intégrée)

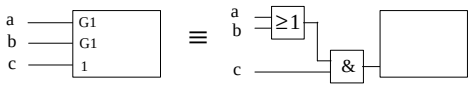
Sortie de technologie trois états

Condition d'évolution de la sortie :
la sortie est positionnée après retour de l'entrée d'horloge à son état initial (bascule à commande par impulsion)
Condition d'évolution de la sortie :
la sortie est active si le contenu du compteur est égal à la valeur indiquée (ici 5)

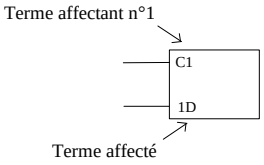
Dépendance d'une sortie envers une autre sortie
Ici, les sorties a et c sont dépendantes de la sortie b (même numéro) par la fonction X:
G : ET
V : OU
N : OU Exclusif
Z : relation d'interconnexion
C : relation de contrôle (horloge)
S : SET (Mise à 1)
R : RESET (Mise à 0)
EN : ENABLE (autorisation)
M : Sélection d'un mode de fonctionnement
A : Sélection d'une adresse

Compléments sur la dépendance

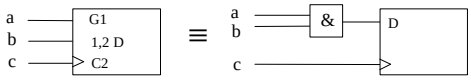
- Si plusieurs termes affectants portent le même numéro, les termes sont implicitement liés par un OU :



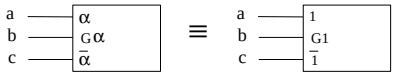
- Un symbole précisant la fonction d'un terme affecté est placé à droite du numéro du terme affectant :



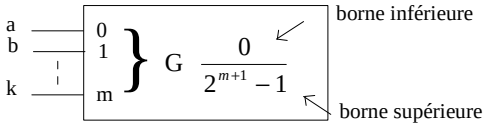
- Si une entrée (ou une sortie) est affectée par plusieurs termes affectants, les numéros d'identification de chacun de ces termes sont écrits séparés par des virgules et dans le même ordre que celui des relations logiques affectantes :



- Si l'écriture d'un numéro d'identification risque d'introduire une confusion, il peut être remplacé par un autre caractère, par ex. une lettre grecque :

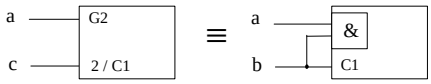


- Il est plus simple, lorsque des entrées de dépendance peuvent être groupées, d'affecter à chacune d'elles un poids, puis d'identifier une combinaison donnée par le nombre binaire correspondant :



Le nombre binaire $(k...ba)_2$ où (a) a le poids 2^0, (b) a le poids 2^1 etc ... est alors compris entre 0 et 2^{m+1}. On précise alors les bornes d'utilisation effective.

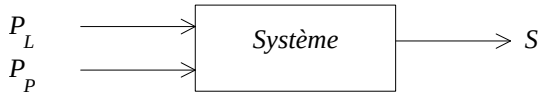
- Lorsqu'une entrée a plusieurs fonctions, il est possible de préciser ces fonctions sur une même figure, en séparant les termes respectifs par des barres obliques (/) :



TD 1. LOGIQUE COMBINATOIRE 1

1. Modélisation d'une fonction logique - Simplification - Réalisation

Soit le système logique d'entrées P_L et P_P et de sortie S :



$P_L = 1$ s'il pleut; $P_L = 0$ s'il ne pleut pas.
$P_P = 1$ si présence de parapluie; $P_P = 0$ si absence de parapluie.

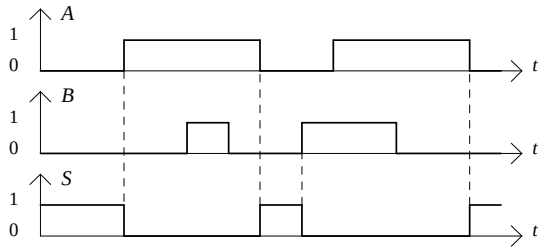
a) Construire le système logique pour permettre de sortir ($S = 1$) quand il ne pleut pas, ou quand il pleut et que l'on est muni d'un parapluie. Construire également $\overline{S}$ directement.

b) Simplifier la fonction logique S :
   b1) de façon algébrique
   b2) de façon graphique

c) Donner le schéma électrique symbolique de la fonction logique *S*.

2. Reconnaissance d'une fonction logique d'après un chronogramme

On donne le chronogramme suivant décrivant les signaux A, B et S en fonction du temps :



a) Quelle est la fonction logique f reliant les variables logiques A, B et S : $S = f(A, B)$

b) Donner le symbole électrique de f.

3. Simplification d'une fonction logique

Soit la fonction logique à 4 variables A, B, C, D :

$$S = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,\overline{B}\,C\,D + \overline{A}\,B\overline{C}\,\overline{D} + \overline{A}\,BC\,D + A\overline{B}\,\overline{C}\,\overline{D}$$

Les mots suivants (≡ états logiques du mot $ABCD$) étant indéfinis (≡ indifférents pour la sortie *S*) car ces combinaisons d'entrée n'apparaîtront jamais :

$$A\,B\overline{C}\,\overline{D},\ A\,B\overline{C}\,D,\ A\,BC\,D,\ A\,BC\overline{D},\ A\overline{B}\,C\,D \text{ et } A\overline{B}\,C\overline{D}$$

a) Simplifier la fonction S en regroupant les 1 du diagramme de Karnaugh. En déduire la fonction logique S .

b) Facultatif : Simplifier la fonction $\overline{S}$ en regroupant les 0. Retrouver le même résultat qu'en a) pour la fonction S .

4. Synthèse d'un système logique combinatoire

La commande des essuie-glace d'un véhicule est assurée par un bouton de mise en marche *M* et un contact de fin de course *F*. La mise en rotation est provoquée par la mise à 1 du bouton *M*, quelle que soit la position des essuie-glace. L'arrêt est obtenu si *M* = 0 et si les essuie-glace sont dans leur position de repos *F* = 1.

1. Donner l'expression de la variable logique *C*.
2. Proposer un montage chargé de générer la variable logique *C* de commande des essuie-glace.

5. Synthèse d'une Fonction logique

Soit la fonction logique s de 3 variables a, b, c : $s = MAX(a,b,c)$.

On définit le MAX de 3 variables a, b, c comme égal à :

0 si parmi a, b, c le nombre de variables à l'état 0 est plus élevé que celui à l'état 1
1 sinon.

. Donner l'expression de la variable logique s en fonction des variables a, b et c.

A préparer pour les Travaux Pratiques :

6. Réalisation d'une fonction logique à l'aide de portes NAND exclusivement

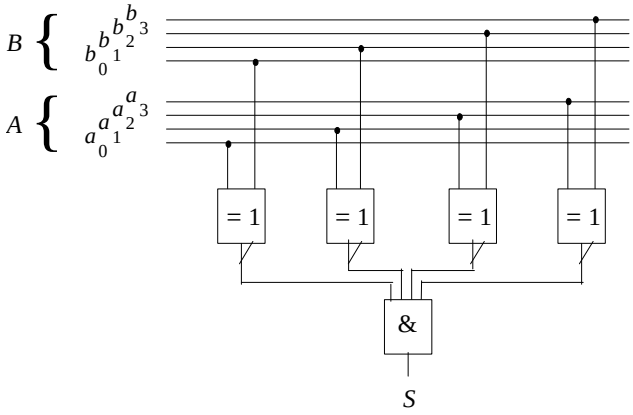
En utilisant exclusivement des portes logiques NAND, réaliser les fonctions suivantes :

a) NON (*NOT*) (1 entrée)
b) OU (*OR*) (2 entrées)
c) ET (*AND*) (2 entrées)
d) NOU (*NOR*) (2 entrées)
e) OUX ou XOU (*XOR*) (2 entrées)

TD 1 ANNEXE. LOGIQUE COMBINATOIRE 1

1. Analyse d'un système logique combinatoire

- Donner l'expression de la variable logique *S*. - Indiquer le rôle du montage.



2. Simplification d'une fonction logique

Le verrou *S* d'une serrure électronique sans mémoire (l'ordre n'importe pas) doit s'ouvrir dans les configurations suivantes des clés logiques *A, B, C, D* :

$$\overline{A}\,\overline{B}CD,\ \overline{A}\,\overline{B}C\overline{D},\ \overline{A}B\overline{C}\,\overline{D},\ \overline{A}B\overline{C}D,$$

les conditions suivantes étant indifférentes (ces combinaisons d'entrée ne peuvent se produire du fait de la mécanique de la serrure) :

$$AB\overline{C}\,\overline{D},\ AB\overline{C}D,\ ABCD,\ ABC\overline{D},\ A\overline{B}CD \text{ et } A\overline{B}C\overline{D}.$$

- Donner, après simplification par exemple par la méthode de Karnaugh, l'expression de la fonction logique *S* sous une forme autorisant une réalisation à l'aide d'une seule porte logique.

Tutorial 1. Tutorial Circuit Maker Numérique

0. Circuit Maker tour - Modes de simulation

Mode de simulation numérique : 

Vitesse de simulation : *Simulation* ⟶ *Digital Options* ⟶ *Simulation Speed*

Instrument Oscilloscope : *Device* ⟶ *Instruments* ⟶ *Digital* ⟶ *SCOPE (sonde)* +  *(Waveform)*

Instrument Horloge : *Device* ⟶ *Instruments* ⟶ *Digital* ⟶ *Pulser*

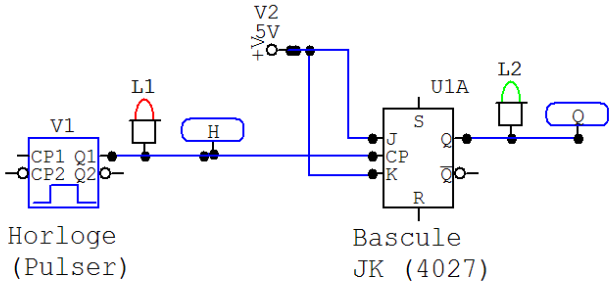
Instrument Data Sequenceur : *Device* ⟶ *Instruments* ⟶ *Digital* ⟶ *Data Seq*

Instrument +5 Volts : *Device* ⟶ *Digital* ⟶ *Power* ⟶ *Logic Switch*

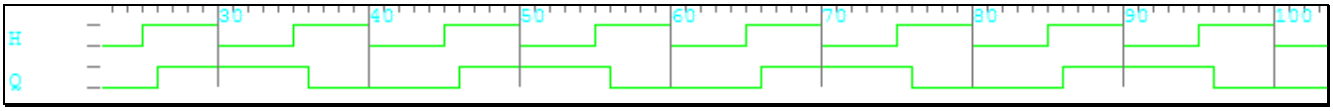
Composant LED : *Device* ⟶ *Digital Animated* ⟶ *Displays* ⟶ *Logic Display*

Instrument Roue codeuse : *Device* ⟶ *Switches* ⟶ *Digital* ⟶ *Hex Key*

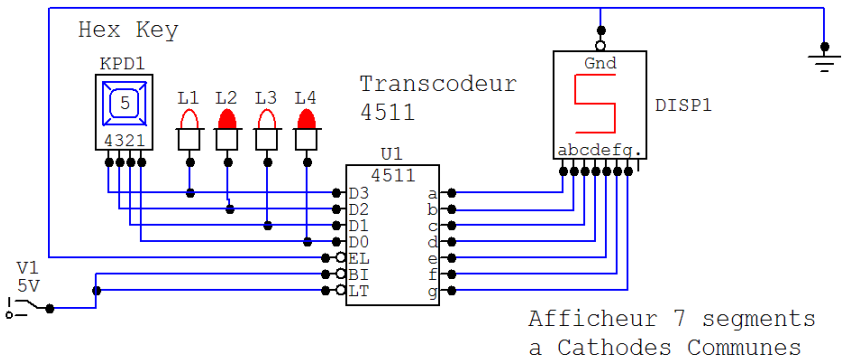
1. Diviseur de frequence par 2



Visualiser les chronogrammes de H et Q simultanement

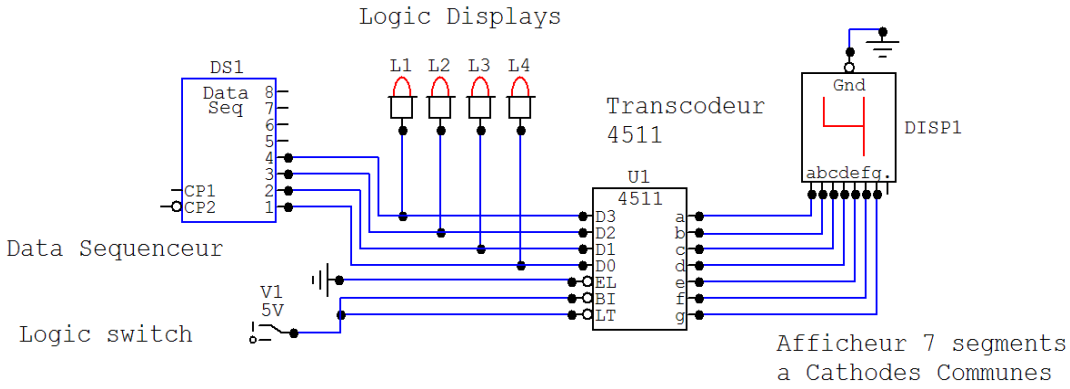


2. Transcodeur



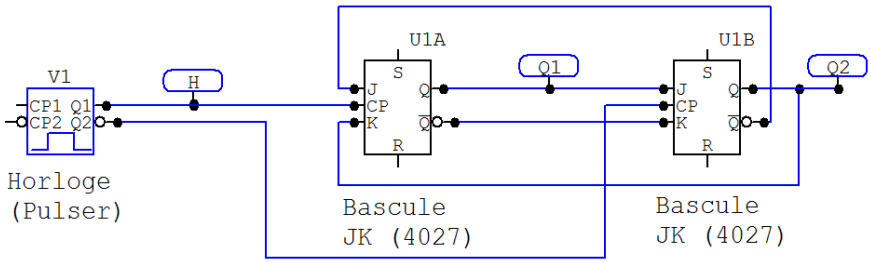
Verifier l'affichage du digit
selectionne par Hex Key

3. Data Sequenceur

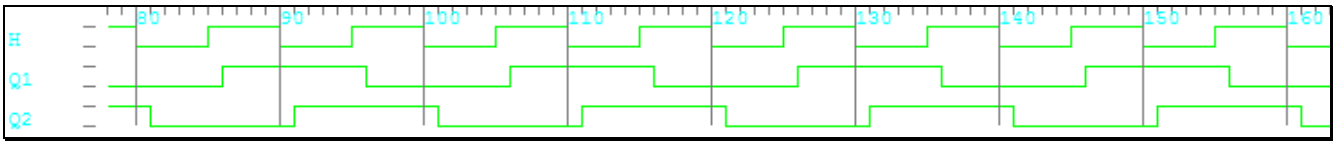


Verifier la séquence d'affichage du
digit engendré par le Data Sequenceur

4. Compteur



- Chronogrammes simultanés de H, Q1, Q2 (temps 0-50 Ticks)



TP 1. LOGIQUE COMBINATOIRE 1

1. Matériel nécessaire

- Alimentation stabilisée (2x[0-30 V] ⎓ + 1x[5 V] ⎓)
- Multimètre
- Moniteur MS05 (plaquette de câblage)
- Câbles : 6 fils Banane, petits fils.
- **Composants :** - 4 Résistances 1 kΩ (1/4 Watt)
  - 4 LEDs rectangulaires (3 Vertes, 1 Rouge)
  - 3 mini-interrupteurs

  Circuits logiques de la famille CMOS 4000 :
  - 1 *4071* : 4 OR à 2 entrées
  - 1 *4081* : 4 AND à 2 entrées
- Logiciel de simulation Circuit Maker

Simulation (& Câblage) :

Pour des raisons de compatibilité, n'utiliser que des circuits de la même famille (famille CMOS 4000 à ne pas mélanger avec la famille TTL 74xxx).

2. Notation du TP

Faire examiner par le professeur en fin de séance, les différentes parties du TP.

3. Etude Théorique

On reprend l'exercice du TD :

3.1. Synthèse de la Fonction logique MAX

Soit la fonction logique s de 3 variables a, b, c : $s = MAX(a,b,c)$.

On définit le MAX de 3 variables a, b, c comme égal à :

0 si parmi a, b, c le nombre de variables à l'état 0 est plus élevé que celui à l'état 1

1 sinon.

. Donner l'expression de la variable logique s en fonction des variables a, b et c.

Synthèse de la Fonction logique MAX - Corrigé

a	b	c	s
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

d'où : $s = \bar{a}bc + a\bar{b}c + ab\bar{c} + abc$

Simplification de s :

- algébrique : $s = c\left(\bar{a}b + a\bar{b}\right) + ab\left(c + \bar{c}\right) = c\left(a \oplus b\right) + ab$
- graphique (Karnaugh) : $\boxed{s = ab + bc + ac}$

c \ ab	00	01	11	10
0	0	0	1	0
1	0	1	1	1

3.2. Réalisation d'une fonction logique à l'aide de portes NAND exclusivement

En utilisant exclusivement des portes logiques NAND, réaliser les fonctions suivantes :

a) NON (*NOT*) (1 entrée)
b) OU (*OR*) (2 entrées)
c) ET (*AND*) (2 entrées)
d) NOU (*NOR*) (2 entrées)
e) OUX ou XOU (*XOR*) (2 entrées)

4. Etude Expérimentale

4.0. Logiciel

- Portes logiques : pour des raisons de compatibilité, prendre uniquement des circuits de la même famille, par exemple CMOS (série 40xx) conseillé, ou à la rigueur la famille TTL, série 74xx) :
  Library : Digital → Digital by function
- Source binaire 0-5V : *Analog → Power → Logic switch*
- LED : *Digital animated → Display → Logic display*
- Simulation : Simuler en *Digital*
- Placer des **«** *Logic display* **»** en entrée et en sortie pour vérifier le fonctionnement des montages.

4.1. Synthèse de la Fonction logique MAX

Simuler (avec le logiciel de simulation) la fonction simplifiée : $\boxed{s = ab + bc + ac}$ avec les circuits de la bibliothèque du logiciel de simulation. Vérifier avec les LEDs la table de vérité établie.

4.2. Réalisation d'une fonction logique à l'aide de portes NAND exclusivement

En utilisant exclusivement des portes logiques NAND, simuler les fonctions suivantes (vérifier avec le logiciel de simulation) :

a) NON (*NOT*) (1 entrée)
b) OU (*OR*) (2 entrées)
c) ET (*AND*) (2 entrées)
d) NOU (*NOR*) (2 entrées)
e) OUX ou XOU (*XOR*) (2 entrées)

4.3. Réalisation d'une fonction logique à l'aide de portes NOR exclusivement

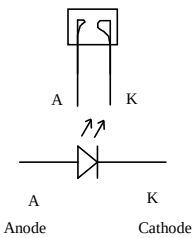
En utilisant exclusivement des portes logiques NOR, simuler les fonctions suivantes (vérifier avec le logiciel de simulation) :

a) NON (*NOT*) (1 entrée)
b) OU (*OR*) (2 entrées)
c) ET (*AND*) (2 entrées)
d) NET (*NAND*) (2 entrées)
e) OUX ou XOU (*XOR*) (2 entrées)

Rangement du poste de travail
Examen des différentes parties du TP et rangement (0 pour tout le TP sinon).

ANNEXE : DOCUMENTATION DES COMPOSANTS

- LED



2. LOGIQUE COMBINATOIRE 2 - APPLICATIONS

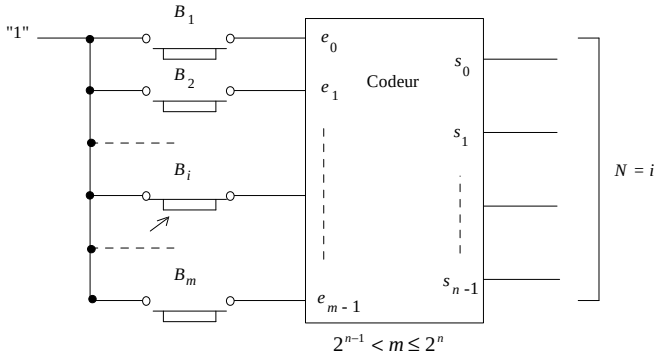
1. Les Applications directes

1.1. Codeur

1.1.1. Définition

Un codeur est un dispositif qui traduit les valeurs d'une entrée dans un code choisi.

Par exemple, un clavier de console ou de machine à écrire comporte *m* touches. Chaque touche, représentative d'un caractère, est affectée d'un numéro. L'opération de codage consiste à donner à chaque numéro (donc à chaque caractère) un équivalent binaire, c'est à dire un mot composé d'éléments binaires.



(Dans la symbolique de ce schéma et contrairement à la majorité des technologies, une entrée « en l'air » est au niveau logique 0).

Si $i = 4 \Rightarrow$ Soit $N = S_3S_2S_1S_0$ on a: $S_3 = 0$, $S_2 = 1$, $S_1 = 0$, $S_0 = 0$ pour un codeur binaire (pur).
Si seul le bouton numéro *i* est actionné, le nombre binaire à 4 éléments $N = S_3S_2S_1S_0$ est égal à *i*, dans le code choisi.

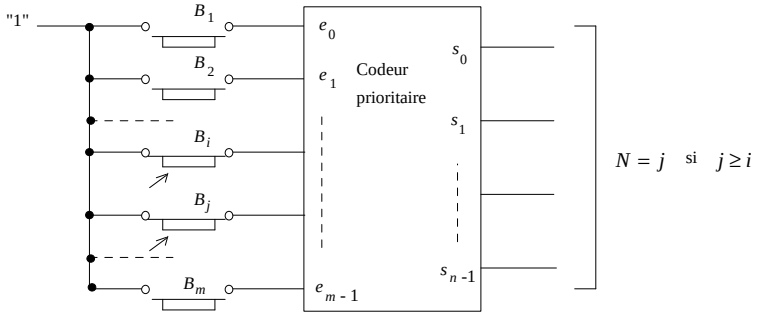
Les tables de Karnaugh (1 table pour chaque sortie) ne sont pas utilisées car le nombre d'éventualités est réduit (1 seule entrée activée).

1.1.2 Intérêt du codage

Si le nombre de boutons est de 10, codé en binaire pur, 4 variables suffisent. Pour un clavier classique, la centaine de touches se codent facilement avec 7 variables binaires (en supposant toujours 1 seule touche active à la fois).
Le codage des informations apporte une réduction non négligeable du nombre de variables à traiter.

1.1.3. Application : codeur prioritaire

Si maladroitement plusieurs boutons sont enfoncés simultanément, le codeur classique donne un résultat erroné car il ne sait plus quel numéro doit être codé.
Un codeur prioritaire est un dispositif réalisant le codage du numéro le plus élevé dans le cas où plusieurs boutons seraient actionnés simultanément. Si une seule commande est envoyée sur le codeur prioritaire, celui-ci fonctionne comme un codeur classique.



Si 2 entrées ou plus sont activées simultanément l'entrée sélectionnée pour le codage est celle ayant le numéro d'entrée le plus élevé. Sinon le codeur prioritaire se comporte comme un codeur classique.

1.1.4. Réalisation pratique des codeurs

Dans sa version la plus générale, un codeur est un ensemble de circuits OU.

Exemple : Codeur Décimal → DCB (en anglais BCD)

Soit la table de codage suivante pour des entrées de e_0 à e_9 : (code DCB) (clavier à 10 touches d'un digicode)

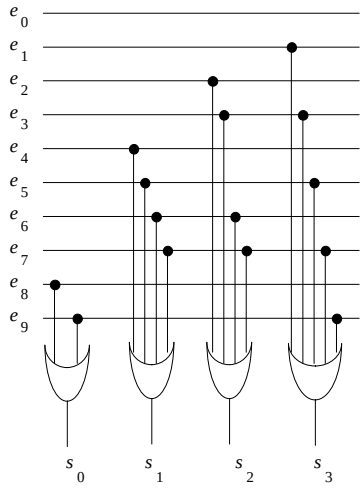
e_i est traduite par $s_0 s_1 s_2 s_3$:

e_0 est traduite par $s_0 s_1 s_2 s_3 = 0\ 0\ 0\ 0$
e_1 est traduite par $s_0 s_1 s_2 s_3 = 0\ 0\ 0\ 1$
e_2 est traduite par $s_0 s_1 s_2 s_3 = 0\ 0\ 1\ 0$
e_3 est traduite par $s_0 s_1 s_2 s_3 = 0\ 0\ 1\ 1$
e_4 est traduite par $s_0 s_1 s_2 s_3 = 0\ 1\ 0\ 0$
e_5 est traduite par $s_0 s_1 s_2 s_3 = 0\ 1\ 0\ 1$
e_6 est traduite par $s_0 s_1 s_2 s_3 = 0\ 1\ 1\ 0$
e_7 est traduite par $s_0 s_1 s_2 s_3 = 0\ 1\ 1\ 1$
e_8 est traduite par $s_0 s_1 s_2 s_3 = 1\ 0\ 0\ 0$
e_9 est traduite par $s_0 s_1 s_2 s_3 = 1\ 0\ 0\ 1$

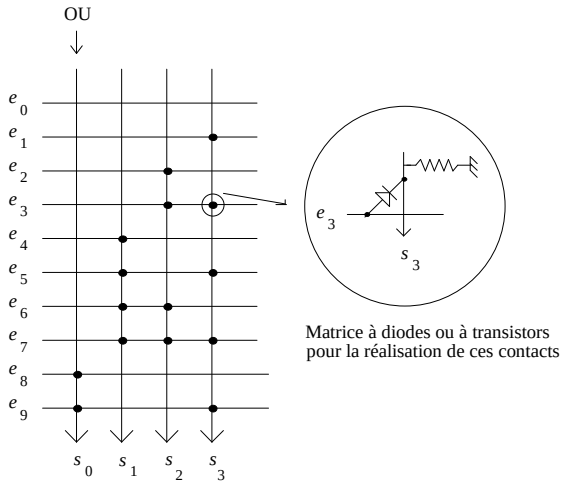
et le codage des sorties :

$$\begin{cases} s_0 = e_8 + e_9 \\ s_1 = e_4 + e_5 + e_6 + e_7 \\ s_2 = e_2 + e_3 + e_6 + e_7 \\ s_3 = e_1 + e_3 + e_5 + e_7 + e_9 \end{cases}$$

Réalisation



Représentation symbolique



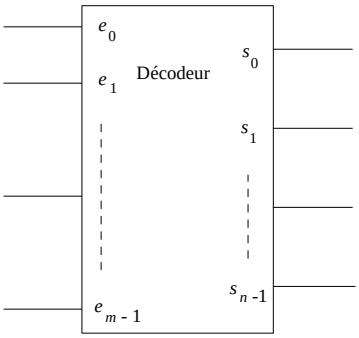
Matrice à diodes ou à transistors pour la réalisation de ces contacts

1.2. Décodeur

1.2.1. Définition

Un décodeur est un circuit qui délivre une (ou des) information(s) lorsque la combinaison des variables binaires d'entrée est représentative du (ou des) mot(s)-code choisi(s).

Un décodeur réalise la fonction inverse (≡ duale) du codeur.

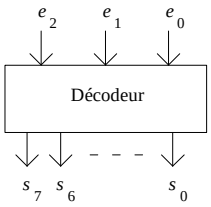


$n \leq 2^m$

m entrées

n sorties
1 seule sortie à la fois est activée

Exemple : Décodeur 3 entrées / 8 sorties



1.2.2. Réalisation des décodeurs

La réalisation des décodeurs se fait à partir d'une matrice ET.

L'expression d'une sortie s_i d'un décodeur est un *minterme* sur les entrées e_i : $s_i = m_i$

Exemple : Décodeur 3 entrées / 8 sorties défini par (code DCB des entrées : DCB → Décimal) :

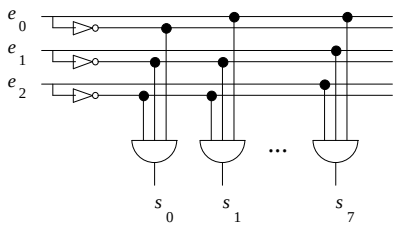
$s_0 = \overline{e}_2\ \overline{e}_1\ \overline{e}_0$

$s_1 = \overline{e}_2\ \overline{e}_1\ e_0$

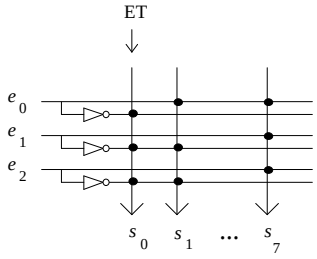
$\vdots$

$s_7 = e_2\ e_1\ e_0$

Réalisation



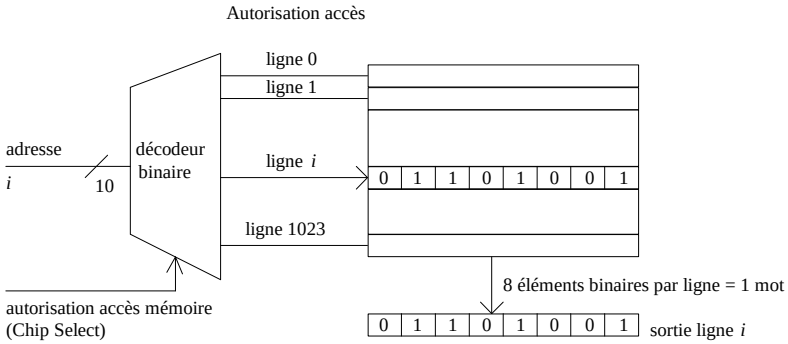
Représentation symbolique



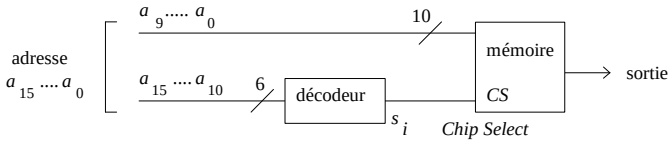
1.2.3. Applications des décodeurs

1.2.3.1. Adressage d'une mémoire (décodage d'adresses) (mémoire paginée)

Représentons une mémoire comme un tableau d'éléments binaires. Ce tableau est divisé en lignes et en colonnes. Pour lire un mot mémoire, il faut lui envoyer le numéro de la ligne souhaitée : c'est son *adresse*. Une mémoire ayant 1 024 lignes, par exemple, nécessite 10 bits d'adresse. Un décodeur interne à la mémoire permet la sélection d'une ligne et d'une seule à un instant donné.



Si un microprocesseur délivre une adresse sur 16 fils, il possède une capacité d'adressage (espace adressable) de 2^{16} soit 65 536 mots. Les 1 024 mots de la mémoire précédente occupent donc une faible partie de cet espace.
Il est alors commode de partager celui-ci en 64 pages de 1 024 mots, chaque page pouvant correspondre à un boîtier mémoire. La sélection du numéro de page, donc du boîtier correspondant (*Chip Select*), est effectué par le décodage de 6 bits parmi les 16 (en général les poids forts). Les bits restants permettent la sélection interne d'un mot mémoire.



La sortie s_i du décodeur est connectée à l'entrée Chip Select (*CS*). Si le nombre décimal équivalent à $(a_{15} \cdots a_{10})_2$ est différent de *i*, la mémoire ne délivre aucune information en sortie. Dans le cas contraire, la sortie de la mémoire est le contenu de la ligne dont le numéro est fixé par les adresses $(a_9 \cdots a_0)$ avec $0 \le i \le 63$.
Avec *i* = 3, pour accéder à la mémoire, le microprocesseur doit envoyer une adresse $a_{15} \cdots a_{10}$ telle que :
$(a_{15} \cdots a_{10})_2 = 3_{10}$ ce qui correspond à des adresses :

$(\overline{0}C00\ )_H \leqslant (\overline{0\,0\,0\,0\,1\,1}\ A_9 \ldots A_0)_2 \leqslant (0FFF)_H$ (1 caractère Hexa ≡ 4 bits car $2^4 = 16$)

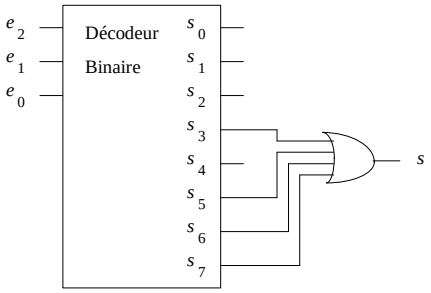
ou bien : $(3\,072)_{10} \le$ Adresse $\le (4\,095)_{10}$

1.2.3.2. Génération de fonction

Comme toute fonction logique s peut s'exprimer comme une somme de mintermes : $s = \sum_i m_i$, il suffit, pour engendrer s , de faire un OU avec les sorties $s_i = m_i$ d'un décodeur $s = \sum_i s_i$ (une sortie de décodeur est un minterme).

Exemple :

N	e_2	e_1	e_0	$s = f(e_2, e_1, e_0)$
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1



1.3. Transcodeur

Un transcodeur est un dispositif permettant de passer du nombre *N* écrit dans un code C_1 au même nombre *N* écrit dand le code C_2.

1.3.1. Synthèse d'un transcodeur

Le nombre *N* dans le code C_1 s'exprime à l'aide des variables *A, B, C, D* par exemple, et dans le code C_2 avec les variables *X, Y, Z.* (Le nombre des variables dans chaque code n'est pas forcément identique).
Le problème de la synthèse d'un transcodeur revient à calculer chacune des sorties, c'est à dire les variables de C_2 (ici *X, Y, Z*) en fonction des entrées ou variables du code C_1 (ici *A, B, C, D*), soit :

$$X = f_1(A, B, C, D)$$
$$Y = f_2(A, B, C, D)$$
$$Z = f_3(A, B, C, D)$$

Exemple : Transcodage d'un nombre *N* (code Gray) en code binaire pur de la moitié entière du nombre *N* :
(le code de Gray est défini tel qu'1 seul bit change au passage d'un mot au suivant)

N	$ABCD$ *Code Gray*	$\rightarrow$	XYZ *Moitié entière en binaire pur*
0	0000		000
1	0001		000
2	0011		001
3	0010		001
4	0110		010
5	0111		010
6	0101		011
7	0100		011
8	1100		100
9	1101		100
10	1111		101
11	1110		101
12	1010		110
13	1011		110
14	1001		111
15	1000		111

X est égale à 1 si les variables *ABCD* prennent les valeurs 1 1 0 0, 1 1 0 1, 1 1 1 1, 1 1 1 0, 1 0 1 0, 1 0 1 1, 1 0 0 1, ou 1 0 0 0, ce que l'on peut reporter dans un tableau de Karnaugh pour obtenir l'expression la plus simple de la fonction f_1.
On procède de la même façon pour *Y, Z* ce qui donne :

X

AB \ CD	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	1	1
10	1	1	1	1

Y

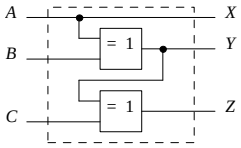
AB \ CD	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	0	0	0	0
10	1	1	1	1

Z

AB \ CD	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	0	0	1	1
10	1	1	0	0

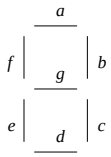
$X = A$ $\qquad Y = \overline{A}B + A\overline{B} = A \oplus B$ $\qquad Z = A \oplus B \oplus C$

D'où le schéma :

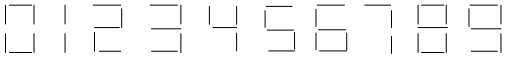


1.3.2. Application : transcodeur BCD / 7 segments

On appelle transcodeur BCD / 7 segments le dispositif de transcodage permettant de passer du code BCD (Décimal Codé Binaire encore appelé binaire pur) ou du code Hexadécimal Codé Binaire au code d'affichage du chiffre sur un afficheur 7 segments. L'opération de décodage du chiffre est réalisé visuellement (interprétation visuelle de la forme du chiffre formé par l'allumage des segments).
Soient *a, b, c, d, e, f* et *g* les variables correspondant aux 7 segments. Si une variable est au niveau actif (par exemple 1), le segment correspondant est allumé. Les segments sont répartis comme l'indique la figure ci-après :



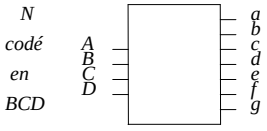
Les chiffres étant formés de la façon suivante :



Le code à 7 segments correspondant est donné par la table :

N	code *BCD* *ABCD*	code 7 segments *abcdefg*
0	0000	1111110
1	0001	0110000
2	0010	1101101
3	0011	1111001
4	0100	0110011
5	0101	1011011
6	0110	1011111
7	0111	1110000
8	1000	1111111
9	1001	1111011

La synthèse d'un décodeur 7 segments s'effectue comme pour un transcodeur classique. Le code C_2 de représentation du nombre *N* ayant 7 variables, il y a 7 fonctions à calculer (*a,b,c,d,e,f,g*) en fonction des variables du code C_1.

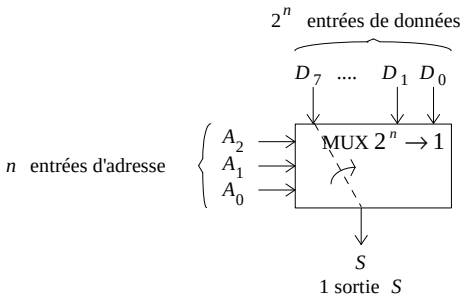


Transcodeur BCD - 7 segments

1.4. **Multiplexeur** (ou **sélecteur** ou **aiguilleur**)

1.4.1. Définition

Un multiplexeur est un circuit réalisant un aiguillage (recopie) de l'une des entrées de données (par la commande des entrées d'adresse) vers une sortie unique. Il y a sélection d'une donnée parmi 2^n (n entrées d'adresses).



Exemples : *Multiplexeur 2 → 1 :*

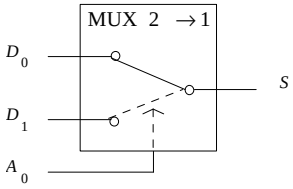


Table de vérité :

A_0	S
0	D_0
1	D_1

Equation de S :

$S = D_0 \quad \text{si} \quad A_0 = 0$

$S = D_1 \quad \text{si} \quad A_0 = 1$

soit : $S = D_0 \overline{A}_0 + D_1 A_0$

Multiplexeur 4 → 1 :

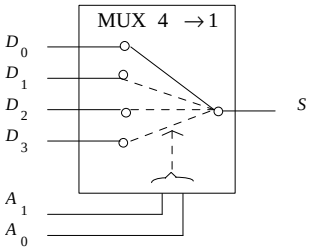


Table de vérité :

A_1	A_0	S
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

Equation de S :

$S = D_0 \quad \text{si} \quad A_1 A_0 = 00$

$S = D_1 \quad \text{si} \quad A_1 A_0 = 01$

$S = D_2 \quad \text{si} \quad A_1 A_0 = 10$

$S = D_3 \quad \text{si} \quad A_1 A_0 = 11$

soit : $S = D_0 \overline{A}_1 \overline{A}_0 + D_1 \overline{A}_1 A_0 + D_2 A_1 \overline{A}_0 + D_3 A_1 A_0$

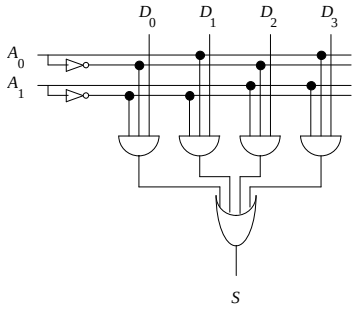
De façon générale, la sortie d'un multiplexeur à *n* entrées d'adresses s'exprime en fonction des entrées de données D_i et des mintermes m_i sur les entrées d'adresses :

$$S = \sum_{i=1}^{2^n} D_i \, m_i$$

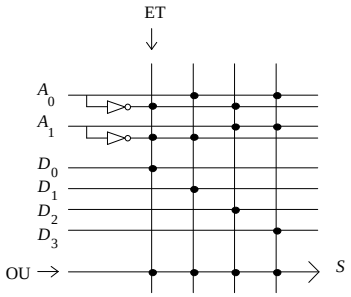
1.4.2. Réalisation

Exemple : Multiplexeur 4 → 1 :

Réalisation



Représentation symbolique



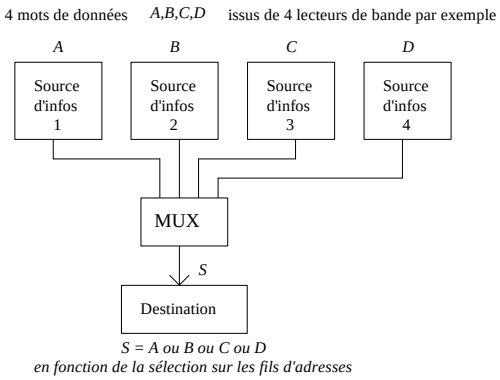
1.4.3. Applications

1.4.3.1. Sélection d'un bit parmi plusieurs bits & sélection d'un mot binaire parmi plusieurs mots binaires

Exemple: Sélection d'un mot de 3 bits parmi les 4 mots de 3 bits :

→ il faut autant de multiplexeurs qu'il y a de bits dans le mot (ici 3 multiplexeurs) :

Principe

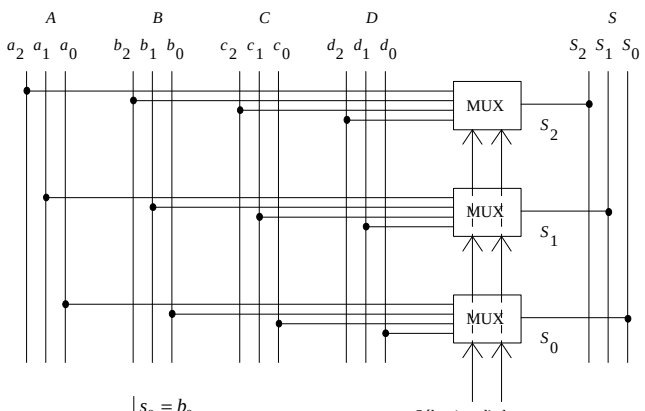


S = A ou B ou C ou D
en fonction de la sélection sur les fils d'adresses

$$e_1e_0 = 00 \rightarrow \begin{vmatrix} s_2 = a_2 \\ s_1 = a_1 \\ s_0 = a_0 \end{vmatrix} \rightarrow S = A$$

$$e_1e_0 = 10 \rightarrow \begin{vmatrix} s_2 = c_2 \\ s_1 = c_1 \\ s_0 = c_0 \end{vmatrix} \rightarrow S = C$$

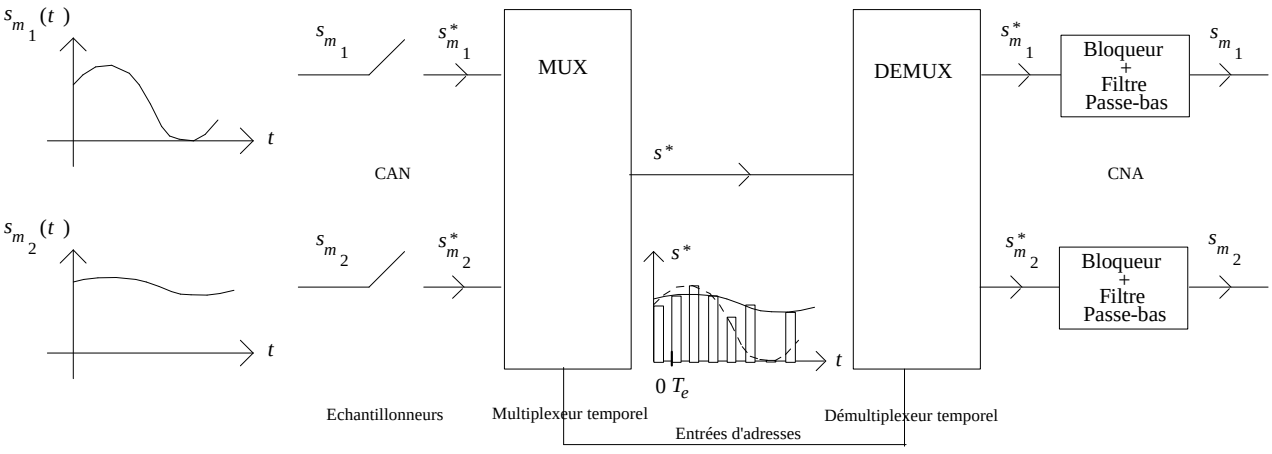
Réalisation



$$e_1e_0 = 01 \rightarrow \begin{vmatrix} s_2 = b_2 \\ s_1 = b_1 \\ s_0 = b_0 \end{vmatrix} \rightarrow S = B$$

$$e_1e_0 = 11 \rightarrow \begin{vmatrix} s_2 = d_2 \\ s_1 = d_1 \\ s_0 = d_0 \end{vmatrix} \rightarrow S = D$$

1.4.3.2. Transmission de plusieurs conversations sur une seule ligne téléphonique (numérique)



1.4.3.3. Génération (matérialisation) d'une fonction logique

La sortie d'un multiplexeur s'exprimant comme une somme de mintermes (forme canonique), et comme toute fonction logique peut se mettre sous forme canonique, elle peut donc s'exprimer comme la sortie d'un multiplexeur.

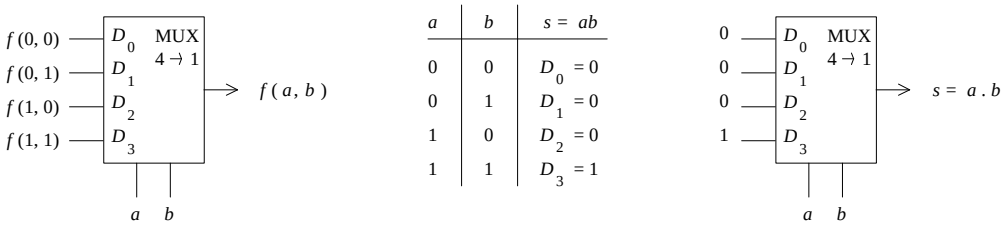
Exemple : *fonction f* de 2 *variables a et b exprimée sous forme canonique (somme de mintermes) :*

$$f(a,b) = \bar{a}\,\bar{b}\cdot f(0,0) + \bar{a}\,b\cdot f(0,1) + a\,\bar{b}\cdot f(1,0) + a\,b\cdot f(1,1)$$

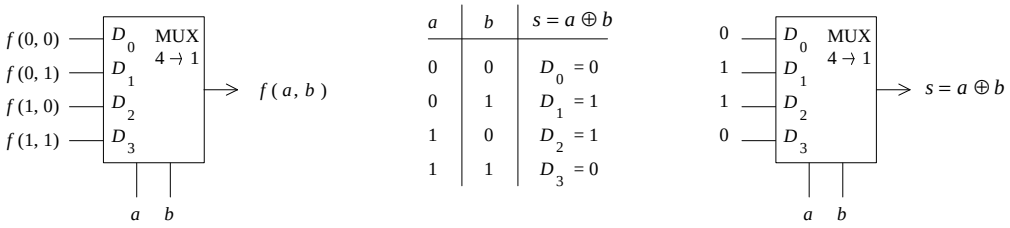
avec : $f(i,j)$ = valeur particulière fonction logique *f* lorsque $a = i$ et $b = j$

$\rightarrow$ utilisation d'un multiplexeur à 4 entrées de données donc 2 entrées d'adresses :

Exemple 1 : opérateur ET : $ab = \bar{a}\bar{b}\cdot 0 + \bar{a}b\cdot 0 + a\bar{b}\cdot 0 + ab\cdot 1 = \bar{a}\bar{b}\cdot D_0 + \bar{a}b\cdot D_1 + a\bar{b}\cdot D_2 + ab\cdot D_3$



Exemple 2 : opérateur OU exclusif : $a \oplus b = a\bar{b} + \bar{a}b = \bar{a}\bar{b}\cdot 0 + \bar{a}b\cdot 1 + a\bar{b}\cdot 1 + ab\cdot 0 = \bar{a}\bar{b}\cdot D_0 + \bar{a}b\cdot D_1 + a\bar{b}\cdot D_2 + ab\cdot D_3$



$\rightarrow$ Le multiplexeur est un opérateur programmable.

Pour matérialiser par un multiplexeur une fonction de *n* variables, il faut un multiplexeur à 2^n entrées de données donc *n* entrées d'adresses : MUX $2^n \rightarrow 1$.

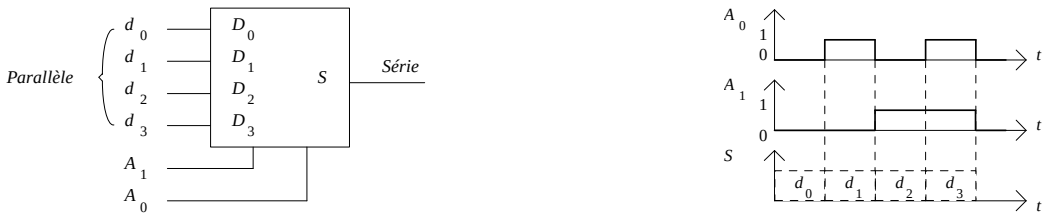
1.4.3.4. Conversion parallèle-série (registre à décalage)

Soit un mot binaire $D = d_3\,d_2\,d_1\,d_0$ disponible en mode parallèle, c'est à dire sur quatre fils, chaque fil étant affecté à un élément binaire (bit) du mot.

Pour transmettre les éléments binaires en série, c'est à dire les uns à la suite des autres sur un seul fil, il faut d'abord (en commençant par exemple par le LSB *(Less Significant Bit)*, bit de plus faible poids) transmettre d_0, puis d_1, puis d_2 et enfin d_3. Ceci revient à sélectionner (ou aiguiller) l'un des éléments binaires de *D* sur le fil unique de sortie série. Le multiplexeur est capable d'effectuer cette tâche si les combinaisons correspondantes sont placées successivement sur les commandes de sélection.

Comme le montre le le chronogrammme ci-dessous :

Dans le premier temps il faut que $A_1 = A_0 = 0$ pour que $S = D_0 = d_0$.
Ensuite A_0 passe à 1 ce qui impose $S = D_1 = d_1$.
Puis $A_1 = 1$ et $A_0 = 0$ d'où $S = D_2 = d_2$.
Et enfin $A_1 = A_0 = 1$ alors $S = D_3 = d_3$.

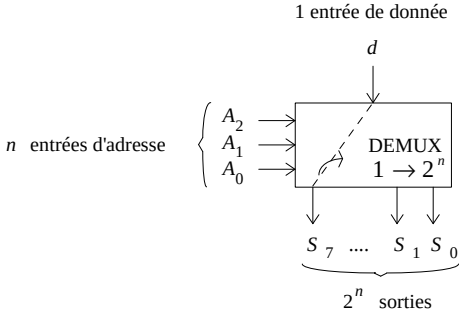


Le mécanisme de changement simultané d'état pour A_0 et A_1 doit être fait à l'aide d'une synchronisation (horloge).

1.5. Démultiplexeur

1.5.1. Définition

Un démultiplexeur réalise l'opération duale du multiplexeur : il aiguille 1 donnée sur 1 parmi 2^n sorties (n entrées d'adresses).



Les sorties non aiguillées sont non activées.
Par convention on les suppose au niveau 0 (en fait, cela peut être 0 ou 1 ou *z* (haute impédance) suivant la technologie utilisée par le constructeur).

Exemple : *Démultiplexeur 1 → 8 :*

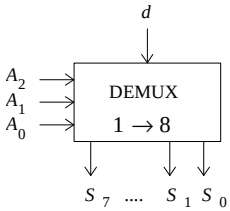


Table de vérité :

$A_2 A_1 A_0$	S_i
000	$S_0 = d$
001	$S_1 = d$
...	...
111	$S_7 = d$

Equation de S_i : (exemple : S_3) $S_3 = \overline{A}_2 \cdot A_1 \cdot A_0 \cdot d$

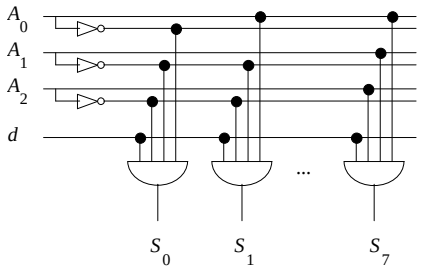
De façon générale, la sortie S_i d'un démultiplexeur à *n* entrées d'adresses s'exprime en fonction de l'entrée de donnée d et d'un minterme m_i sur les entrées d'adresses :

$$S_i = m_i \, d$$

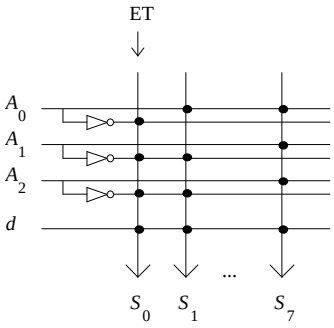
1.5.2. Réalisation

Exemple : Démultiplexeur 1→8 :

Réalisation

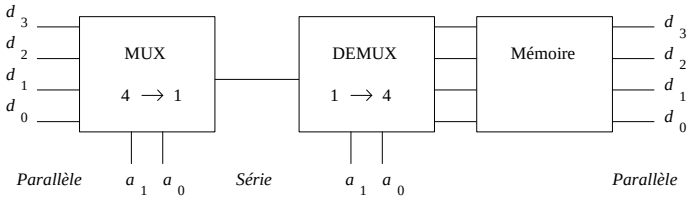


Représentation symbolique



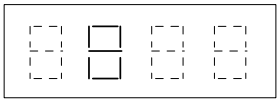
1.5.3. Applications

1.5.3.1. Conversion série/parallèle (registre à décalage)



1.5.3.2. Affichage multiplexé

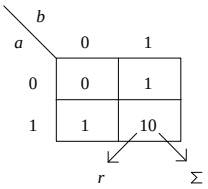
Soit 4 chiffres à afficher, on peut afficher les chiffres l'un après l'autre très vite pour donner l'impression de simultanéité à l'oeil.



1.6. Circuits intégrés arithmétiques

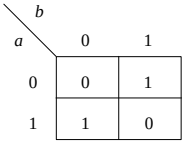
1.6.1. Additionneur

C'est un circuit réalisant l'addition de deux nombres binaires. La table d'addition de deux nombres à un élément binaire est la suivante :



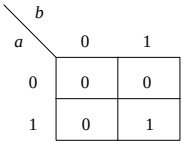
Le résultat de l'opération comporte deux parties :

- la somme $\sum$:



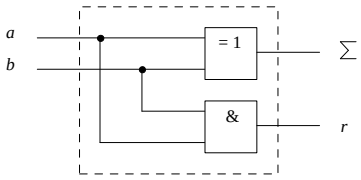
$\sum = a \oplus b$

- et la retenue générée r :



$r = a\,b$

Le circuit élémentaire réalisant cette opération est le **demi-additionneur** :



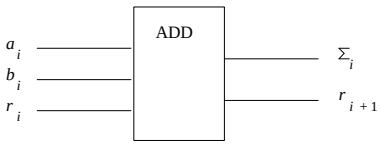
La structure de l'additionneur de deux mots est alors répétitive. Une cellule élémentaire peut donc être utilisée pour chaque poids. Elle est appelée **additionneur complet**. L'addition globale est réalisée par la mise en cascade des cellules au sens des retenues.

L'additionneur complet est défini par la table de vérité ci-après :
(r_i : retenue propagée de l'étage précédent du mot; r_{i+1} : retenue générée)

plus arithmétique

Somme (r_{i+1} , Σ_i) = $a_i + b_i + r_i$

a_i	b_i	r_i	r_{i+1}	Σ_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Σ_i

r_i \ $a_i b_i$	00	01	11	10
0	0	1	0	1
1	1	0	1	0

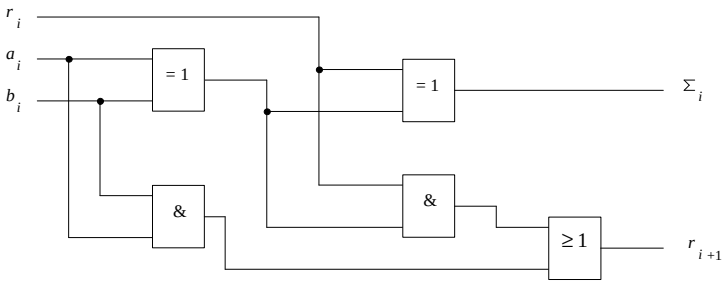
r_{i+1}

r_i \ $a_i b_i$	00	01	11	10
0	0	0	1	0
1	0	1	1	1

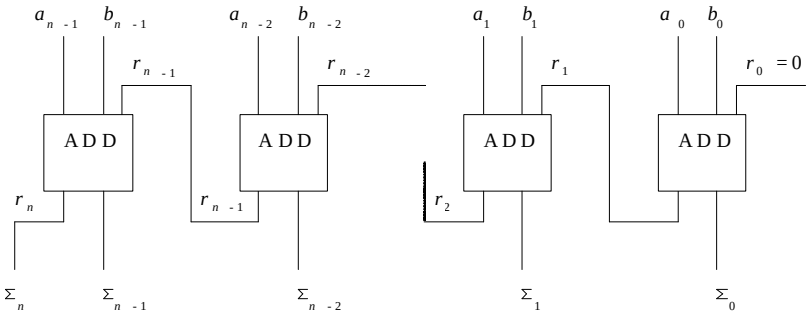
$$\sum_i = a_i \oplus b_i \oplus r_i \qquad r_{i+1} = a_i\, b_i + r_i \left(a_i \oplus b_i\right)$$

(diagonales de 1 significatives de fonctions OUX)

Ce qui donne le schéma de réalisation :



L'addition de deux mots de *n* bits nécessite *n* additionneurs complets, la retenue appliquée sur les plus faibles poids est nulle et chaque retenue calculée est appliquée au chiffre de poids immédiatement supérieur.

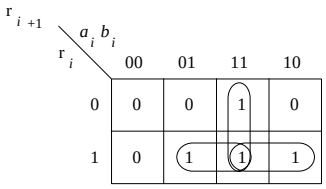


r_i r_1 $r_{0=0}$
a_{n-1} ... a_i .. $a_1 a_0$
\+ b_{n-1} ... b_i .. $b_1 b_0$
Σ_n Σ_{n-1} ..... Σ_0 (+ arithmétique)

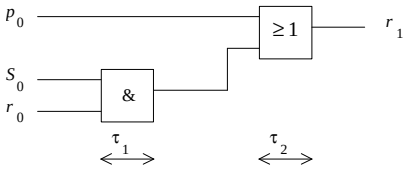
Cette solution est intéressante d'un point de vue du matériel parce que répétitive. Par contre, comme le résultat d'une addition ne peut pas être obtenu instantanément, le temps maximum mis pour obtenir le résultat est directement proportionnel au nombre d'additionneurs. En effet, après le premier temps de calcul la retenue r_1 est appliquée au second additionneur. Ce n'est qu'après le second temps de calcul que la retenue r_2 est délivrée et ainsi de suite, jusqu'au dernier additionneur. Pour cette raison, l'additionneur ainsi réalisée porte le nom d'« **additionneur à propagation de la retenue »** ou « **additionneur à retenue série** ».

Pour éliminer cet inconvénient, la seconde technique consiste à calculer toutes les retenues en parallèle, directement à partir des données sans même calculer les sommes partielles. Le circuit ainsi réalisé est alors appelé « **additionneur à retenue anticipée** ».

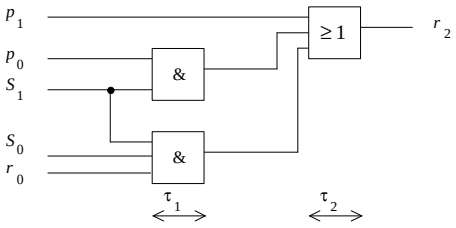
En reprenant le tableau de Karnaugh relatif au calcul de la retenue il vient : $r_{i+1} = a_i b_i + r_i (a_i + b_i)$



Afin d'éviter des temps de calcul cumulatifs, il ne faut pas utiliser la relation en tant que relation de récurrence, c'est à dire qu'il ne faut pas utiliser un résultat de calcul pour le calcul suivant. Il faut systématiquement recalculer chaque terme, ce qui donne, en posant $S_i = a_i + b_i$ et $p_i = a_i b_i$: $r_1 = p_0 + r_0 S_0$



De même : $r_2 = p_1 + r_1 S_1 = p_1 + (p_0 + r_0 S_0) S_1 = p_1 + p_0 S_1 + r_0 S_0 S_1$

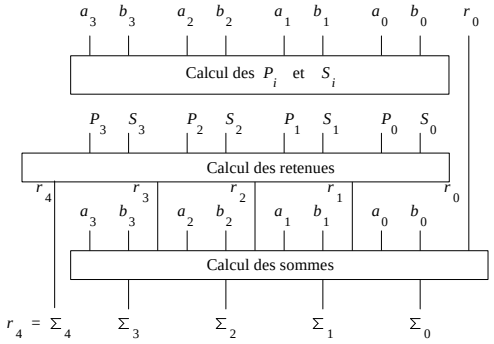


Et ainsi de suite : $r_3 = p_2 + r_2 S_2 = p_2 + (p_1 + p_0 S_1 + r_0 S_0 S_1) S_2 = p_2 + p_1 S_2 + p_0 S_1 S_2 + r_0 S_0 S_1 S_2$

et : $r_4 = p_3 + r_3 S_3 = p_3 + p_2 S_3 + p_1 S_2 S_3 + p_0 S_1 S_2 S_3 + r_0 S_0 S_1 S_2 S_3$

On constate que les temps de calcul des retenues sont tous égaux. Ils correspondent au temps de transit de l'information dans une porte ET (τ_1) et une porte OU (τ_2) en cascade (le nombre d'entrée d'une porte n'affectant pas son temps de transit).

La structure d'un additionneur 4 bits utilisant la technique de calcul anticipé des retenues est la suivante :



Comparaison des retenues propagée et anticipée

Format des mots (en bits)	Temps de calcul en ns (logique TTL série N)		
	Propagation de la retenue	Retenue anticipée	
4	24	24	avec additionneur 4 bits intégré
8	36	36	avec utilisation d'un générateur de retenue
12	48	36	
16	60	36	
64	192	60	avec 2 générateurs de retenue en cascade

1.6.2. Soustraction

Pour la soustraction, on se ramène à une addition. Le nombre négatif est codé en code complément à 2 :

$B \rightarrow \overline{B} + 1$

1001 $\rightarrow$ 0110 + 1 = 0111 (+ arithmétique et non logique).

$$A - B = A + (-B) = A + C_2 B = A + (\overline{B} + 1)$$

1.6.3. Codage

Les codes utilisés pour représenter un mot binaire sont très nombreux : code DCB (Décimal Codé Binaire ou Binaire pur), code Grey, code NRZ (Non Retour à Zéro) en Télécommunications …

Pour coder des nombres entiers signés, le code Cà2 (Complément à 2) est le plus utilisé car plus efficace que le code SVA (Signe et Valeur Absolue). En code Cà2, 0 n'a qu'une seule représentation, et il offre de ce fait une éventualité de codage supplémentaire. Le code SVA réserve le bit de plus fort poids (MSB) pour coder le signe (0 si +, 1 si -), le reste du mot codant en DCB la valeur absolue de *N*.
Le code Cà2 s'obtient en complémentant bit à bit le mot binaire (Cà1, Complément à 1) et en ajoutant 1au résultat issu du Cà1. Le code Cà2 garde également la propriété de réserver le bit MSB pour coder le signe (0 si +, 1 si -).

Exemple : Codes SVA et Cà2 sur 3 bits

N	*Code SVA*	*Code Cà2*
+3	011	011
+2	010	010
+1	001	001
+0	000	000
-0	100	-
-1	101	111
-2	110	110
-3	111	101
-4	-	100

1.6.4. Comparateur

Un comparateur est un dispositif capable de détecter l'égalité de deux nombres et éventuellement d'indiquer le nombre le plus grand ou le plus petit.

Principe : Pour effectuer la comparaison de deux nombres *A* et *B*, deux techniques sont couramment utilisées :

- la soustraction des deux nombres. Si le résultat de l'opération *A* - *B* est positif, cela signifie que *A* est supérieur à *B*. Si le résultat est nul, les deux nombres sont égaux.

- une comparaison bit à bit. C'est cette méthode qui est utilisée dans la plupart des circuits intégrés commercialisés. La comparaison s'effectue poids à poids en commençant par le chiffre le plus significatif.

Les nombres *A* et *B* ayant le même format, le nombre *A* est forcément supérieur à *B* si son élément binaire le plus significatif (MBS) est supérieur au MSB de *B*. Si ces deux bits sont égaux, la supériorité (ou l'infériorité) ne peut être déterminée que par l'examen des bits de poids immédiatement inférieur et ainsi de suite. L'examen des poids successifs s'arrête dès que l'un des éléments binaires est supérieur ou inférieur à l'autre. Les deux nombres *A* et *B* sont égaux si, après avoir examiné tous les éléments binaires, il n'a pas été détecté de supériorité ou d'infériorité.

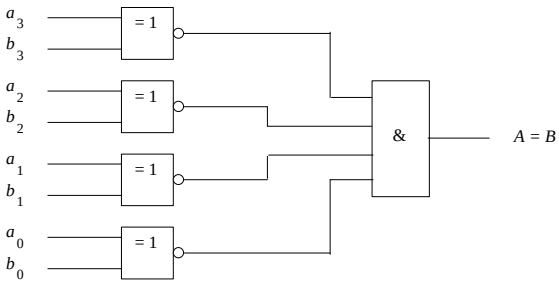
Comparateur donnant l'égalité des deux nombres

C'est le comparateur le plus simple. Deux nombres sont égaux si tous les chiffres sont égaux deux à deux. Pour détecter l'égalité de deux éléments binaires, un opérateur OU exclusif complémenté est indispensable. Un opérateur ET indique la simultanéité de toutes les inégalités partielles.

Soient deux nombres *A* et *B* de quatre éléments binaires chacun, $A = a_3 a_2 a_1 a_0$ et $B = b_3 b_2 b_1 b_0$:

$A = B$ si $(a_3 = b_3)$ ET $(a_2 = b_2)$ ET $(a_1 = b_1)$ ET $(a_0 = b_0)$

Ce qui donne le schéma :



Comparateur complet

Par analogie avec l'additionneur, la conception d'un comparateur complet pour des nombres de quatre éléments binaires peut se faire de deux façons différentes :

- *Première solution* : En cascade, c'est à dire avec propagation des égalités partielles. Les poids de *A* et de *B* sont comparés en commençant par le plus élevé. La comparaison sur les poids faibles ne peut être faite que si tous les bits de poids plus élevés sont égaux deux à deux.

La cellule élémentaire de comparaison comporte trois entrées, les éléments binaires *a* et *b* de même poids de chaque nombre et une entrée *E* pour autoriser la comparaison, ce qui donne la table de vérité ci-après :

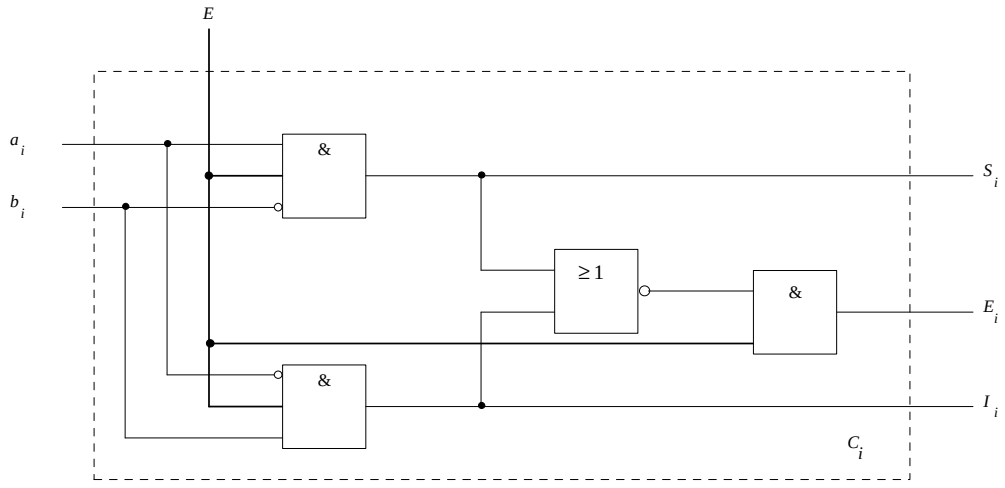
E	a_i	b_i	$a_i = b_i$	$a_i > b_i$	$a_i < b_i$	
			E_i	S_i	I_i	
0	0	0	0	0	0	Pas de comparaison
0	0	1	0	0	0	
0	1	0	0	0	0	
0	1	1	0	0	0	
1	0	0	1	0	0	$S_i = 1$ si $a_i > b_i$
1	0	1	0	0	1	$E_i = 1$ si $a_i = b_i$
1	1	0	0	1	0	$I_i = 1$ si $a_i < b_i$
1	1	1	1	0	0	

$$S_i = E\left(a_i \overline{b}_i\right)$$

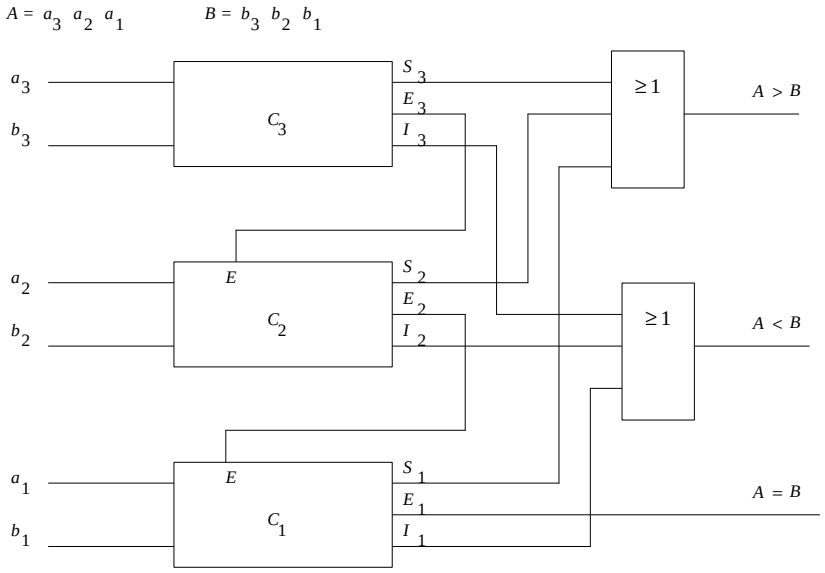
$$I_i = E\left(\overline{a}_i\, b_i\right)$$

$$E_i = E\left(\overline{a_i \oplus b_i}\right) = E\, a_i b_i + E\, \overline{a}_i\, \overline{b}_i = E\left(\overline{S_i + I_i}\right)$$

D'où le schéma d'une cellule de comparaison, notée C_i :



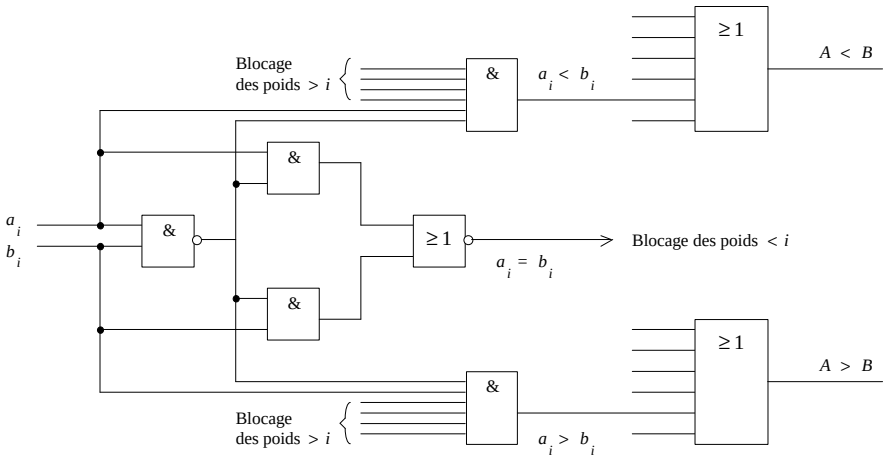
L'entrée d'autorisation *E* est en fait la détection d'égalité des éléments binaires de poids supérieurs; le schéma de l'ensemble est alors le suivant :



Remarque : Comme pour l'additionneur à propagation de la retenue, le résultat de la comparaison apparaît après un temps directement lié au nombres de cellules à traverser à cause de la mise en cascade (calcul série). Pour palier cet inconvénient, c'est une structure parallèle qu'il faut adopter.

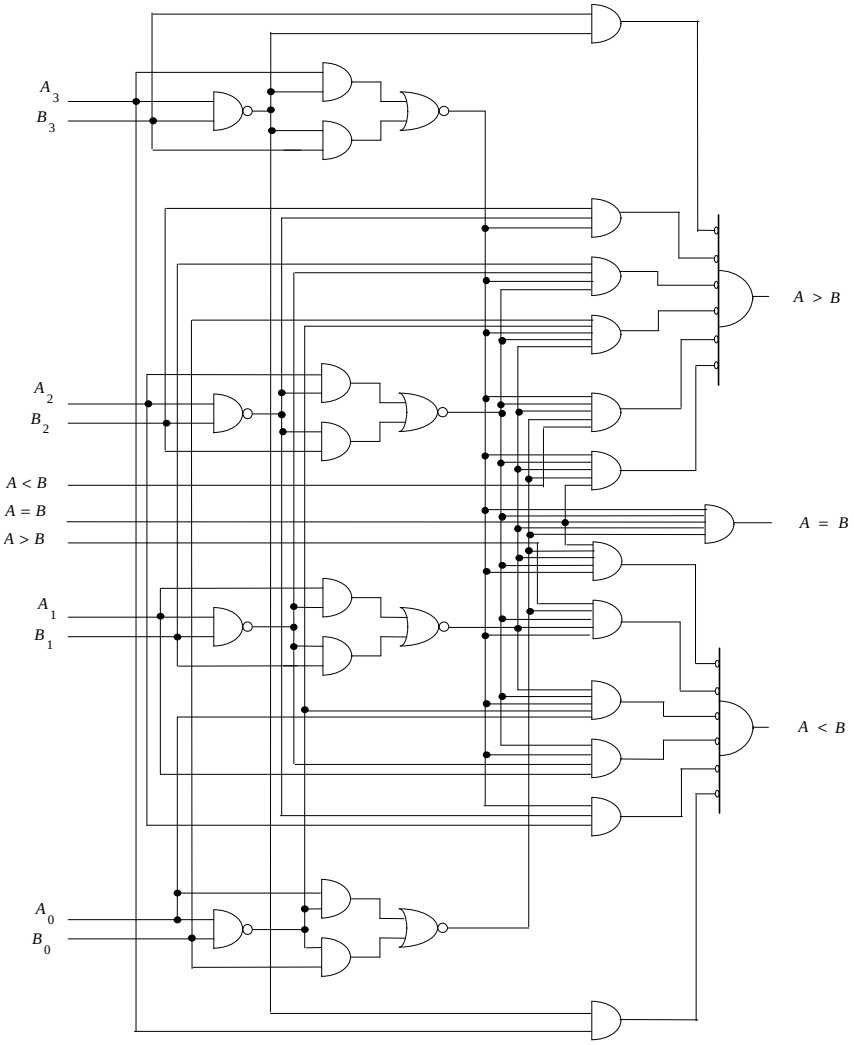
- *Deuxième solution* : Comparaison parallèle. Tous les éléments binaires de même poids sont systématiquement et simultanément comparés. Le blocage s'effectue alors sur les résultats de chaque comparaison.

La cellule élémentaire C_i devient :

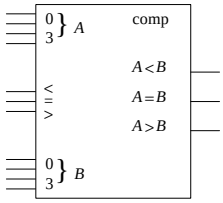


Le blocage des sorties $b_i > a_i$ (ou $b_i < a_i$) se fait par une porte ET recevant toutes les sorties détectant les égalités $b_j = a_j$ des poids supérieurs au rang $i\ (\forall\ j > i)$. Le nombre d'entrées de cette porte augmente donc au fur et à mesure que l'on s'éloigne du MSB.
L'information *A = B* est fournie par une porte ET vérifiant la simultanéité des égalités partielles.

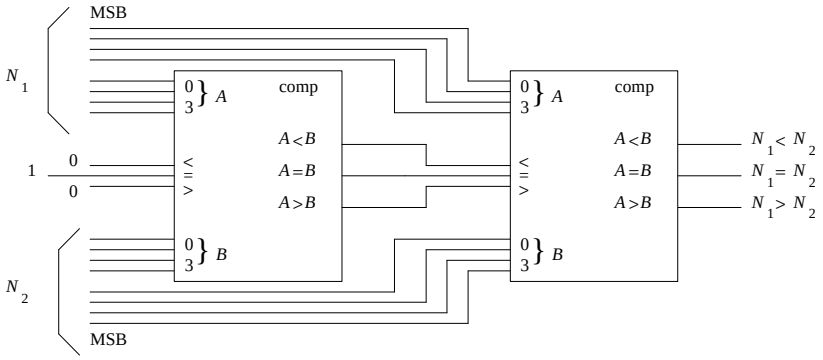
Le schéma du circuit 7485 ci-après montre l'ensemble d'un comparateur 4 bits cascadable.



Symbole

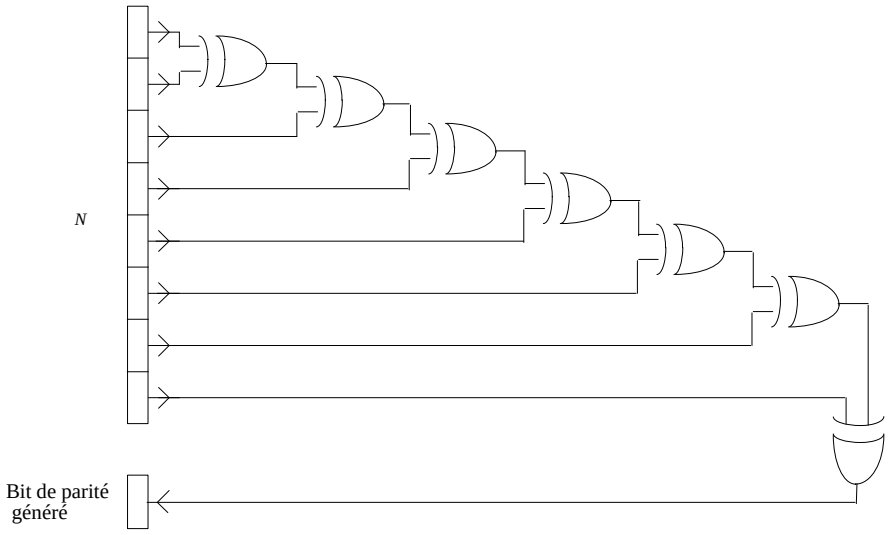


Les entrées $a < b$, $a = b$ et $a > b$, dites entrées de mise en cascade, sont représentatives des résultats des comparaisons sur les éléments binaires d'indice inférieur. Ainsi, pour effectuer la comparaison de deux nombres de huit éléments binaires, on adopte le montage ci-après :



1.6.5. Générateur de parité

On appelle parité d'un mot binaire *N* le nombre de 1 contenus dans ce mot; le mot a une parité paire si ce nombre de 1 est pair. Afin de rendre les transmissions numériques plus robustes au bruit, on adjoint un bit à tous les mots transmis. Ce bit, dit de parité, est choisi de façon à ce que le mot complet formé du mot et du bit de parité ait une parité paire (dans le cas de la parité paire). Le principe utilisé pour engendrer ce bit de parité repose sur la propriété du OU exclusif : $a \oplus b \oplus c \oplus \cdots \oplus m$ vaut 1 si un nombre impair de variables est au niveau 1 :



1.6.6. Unité Arithmétique et Logique (UAL / ALU)

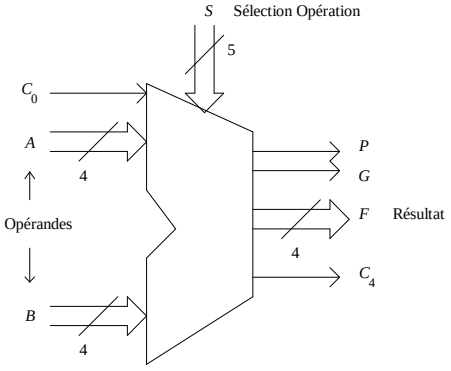
Ce circuit est utilisé dans quasiment tous les processeurs de calcul. C'est un opérateur capable d'effectuer, comme son nom l'indique, un ensemble de traitements arithmétiques (addition, soustraction, multiplication (par 2 par décalage d'1 cran vers la gauche des bits du mot), division (par 2 par décalage d'1 cran vers la droite des bits du mot) etc ...) ou logiques (ET, OU ...) sur des mots binaires de longueur donnée.

Le choix de l'opération est déterminé par des bits de commande. C'est donc un opérateur programmable.

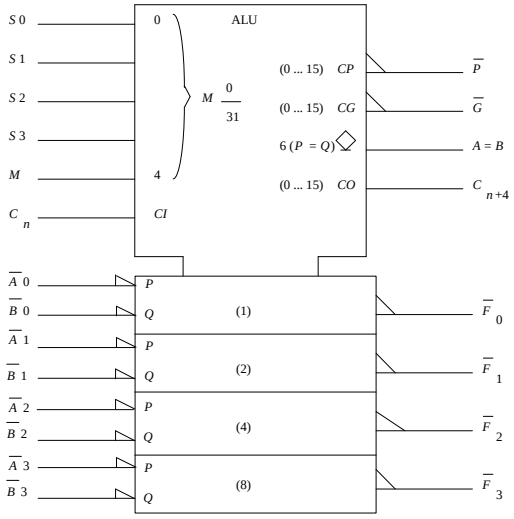
Il n'est pas intéressant de présenter en détail l'architecture interne de l'ALU, qui résulte d'une grande partie des circuits déjà présentés. Par contre, il est important de comprendre l'action de l'unité arithmétique et logique sur les mots binaires (chemin des données ...).

Exemple d'unité logique et arithmétique intégrée 74181

Représentation fonctionnelle

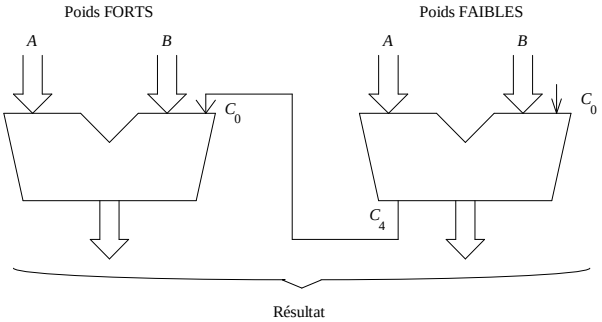


Symbole logique

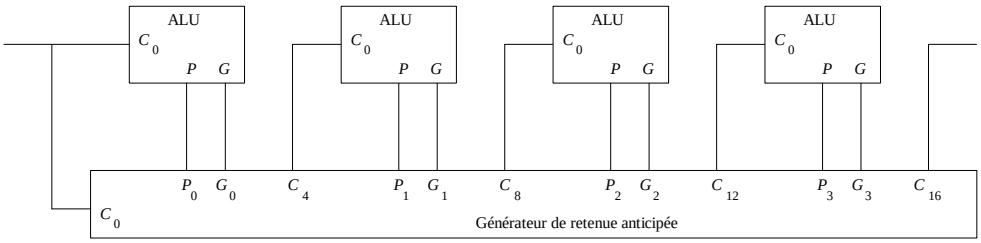


Ce circuit intégré utilise des mots de quatre éléments binaires. Cinq fils de sélection permettent un choix parmi 32 fonctions groupées en 16 opérations arithmétiques et 16 opérations logiques. Indépendamment de la fonction réalisée, ce circuit dispose d'une sortie détectant l'égalité des données en entrée.

Lors des opérations arithmétiques sur des nombres de plus de quatre bits, il existe la possibilité de mise en cascade des boîtiers avec la technique de la propagation de la retenue (C_0 retenue entrante, C_4 retenue sortante).



On peut également utiliser la technique de la retenue anticipée en utilisant un circuit supplémentaire spécialisé dans le calcul des retenues (utilisation de *P* et *G*) :



2. Les réseaux logiques programmables

2.1. Structure

Les réseaux logiques programmables sont des circuits qui se programment à partir d'un logiciel qui, après saisie de l'équation logique, engendre un fichier JEDEC (.JED). Celui-ci permet la programmation du composant par une carte spécialisée.

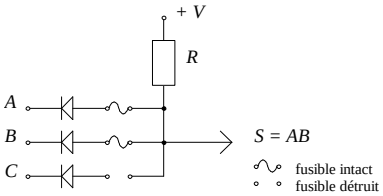
Une approche comme celle que propose le langage VHDL est similaire mais plus puissante car la programmation se fait à l'aide d'un langage puissant par sa modularité. L'entrée peut être le source VHDL ou un fichier graphique des circuits logiques ou encore une machine d'états. Un compilateur VHDL accepte l'une ou l'autre de ces entrées et permet même la génération du code source VHDL à partir d'une entrée à base de portes logiques ou de machines d'états.

Toute fonction logique de n variables peut se mettre sous la forme d'une somme logique de produits logiques (somme de mintermes) ou sous la forme d'un produit logique de sommes logiques (produit de maxtermes).

→ utilisation de 2 matrices : - matrice OU - matrice ET

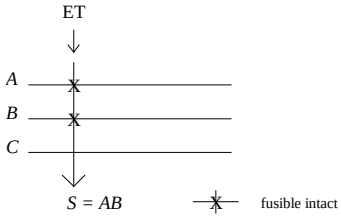
Matrice ET

Réalisation



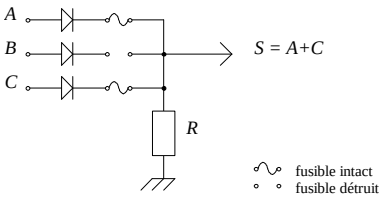
Matrice ET

Représentation symbolique



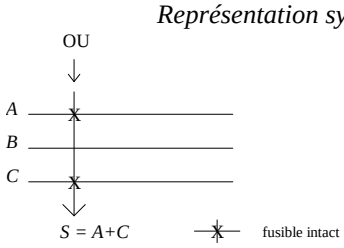
Matrice OU

Réalisation



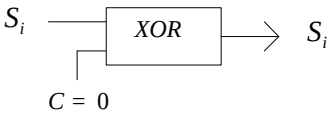
Matrice OU

Représentation symbolique

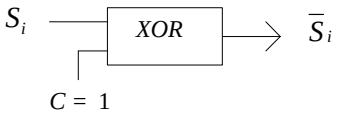


On adjoint parfois un circuit de sortie à la matrice OU :

a) Circuit d'inversion (commandé par l'entrée C)

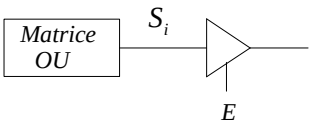


Pas d'inversion : $S_i = S_i \oplus 0$

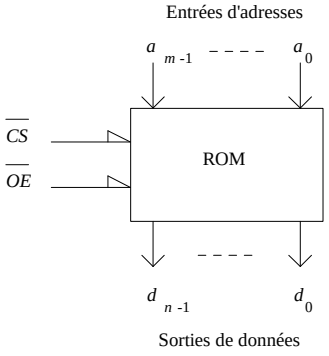


Inversion : $\overline{S}_i = S_i \oplus 1$

b) Circuit 3 états (commandé par l'entrée E de mise en haute impédance)



2.2. Mémoire Morte *(ROM) (Read only Memory) (elle contient un décodeur)*



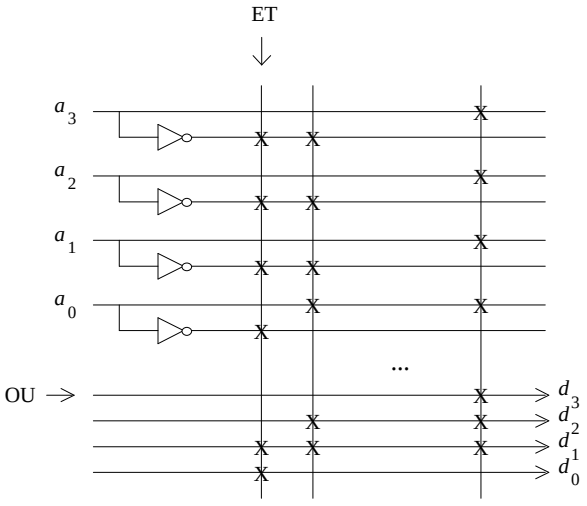
$\overline{CS}$: Chip Select (sélection de boitier)
actif à l'état bas ($CS = 0$: boitier mémoire sélectioné)

$\overline{OE}$: Output Enable (contrôle l'activité, la disponibilité des sorties)
actif à l'état bas ($OE = 0$: sorties disponibles)

Pour chaque adresse apparaît une donnée particulière définie par son adresse.

Ex. : $m = 4$, $n = 4$, avec le contenu de mémoire :

Adresse	*Donnée*
$a_3\ a_2\ a_1\ a_0$	$d_3\ d_2\ d_1\ d_0$
0 0 0 0	0 0 1 1
0 0 0 1	0 1 1 0
...	...
1 1 1 1	1 1 1 0



fusible intact

Les ROMs sont écrites en usine selon l'application voulue et sont figées (pas d'effacement ni réécriture des données). Dans une ROM la matrice ET est fixée et la matrice OU est programmable.

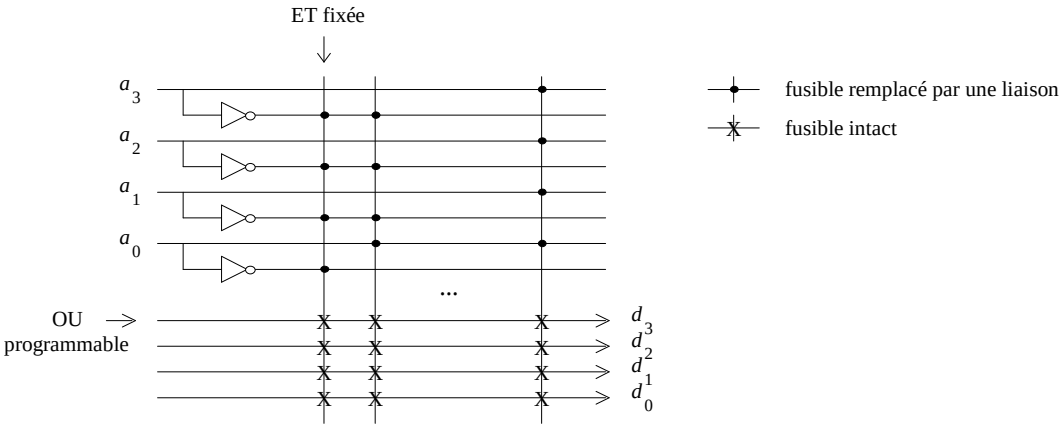
Capacité d'une ROM : couramment au minimum de l'ordre de 16 KOctets = 2^{14} Octets
$\rightarrow m = 14$ (14 entrées d'adresse)
$\rightarrow n = 8$ (8 sorties de données)

Rapidité d'une ROM : de l'ordre de 100 ns (dépendant de la technologie)

2.3. PROM - EPROM *(Programmable ROM - Erasable PROM)*

PROM : ROM à programmer par l'utilisateur
EPROM : PROM effaçable (aux UV (UltraViolets)) et reprogrammable
EEPROM : EPROM effaçable non pas aux UV mais électriquement.

Dans une PROM (ou une EPROM, ou une EEPROM) la matrice ET est fixée et la matrice OU est programmable.



2.4. PAL / GAL *(Programmable / Gate Array Logic)*

Les réseaux PALs et GALs ont une structure opposée à celle d'une PROM.
La matrice ET est programmable. La matrice OU est fixée.

Utilisation des circuits PALs

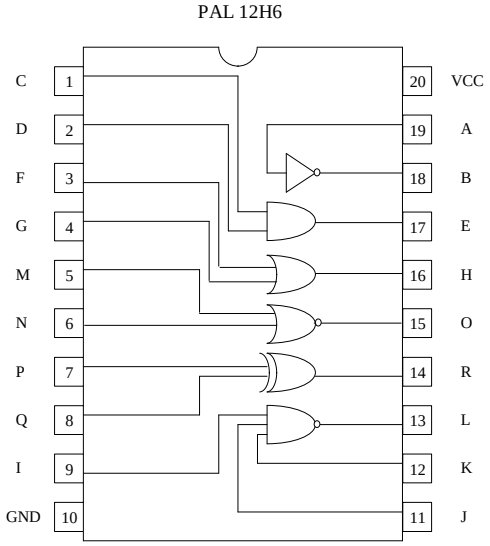
Un circuit PAL peut à lui seul assurer toutes les fonctions combinatoires conventionnelles.

$\rightarrow$ on peut à l'aide d'un réseau PAL réaliser plusieurs portes élémentaires et ainsi remplacer plusieurs circuits intégrés.

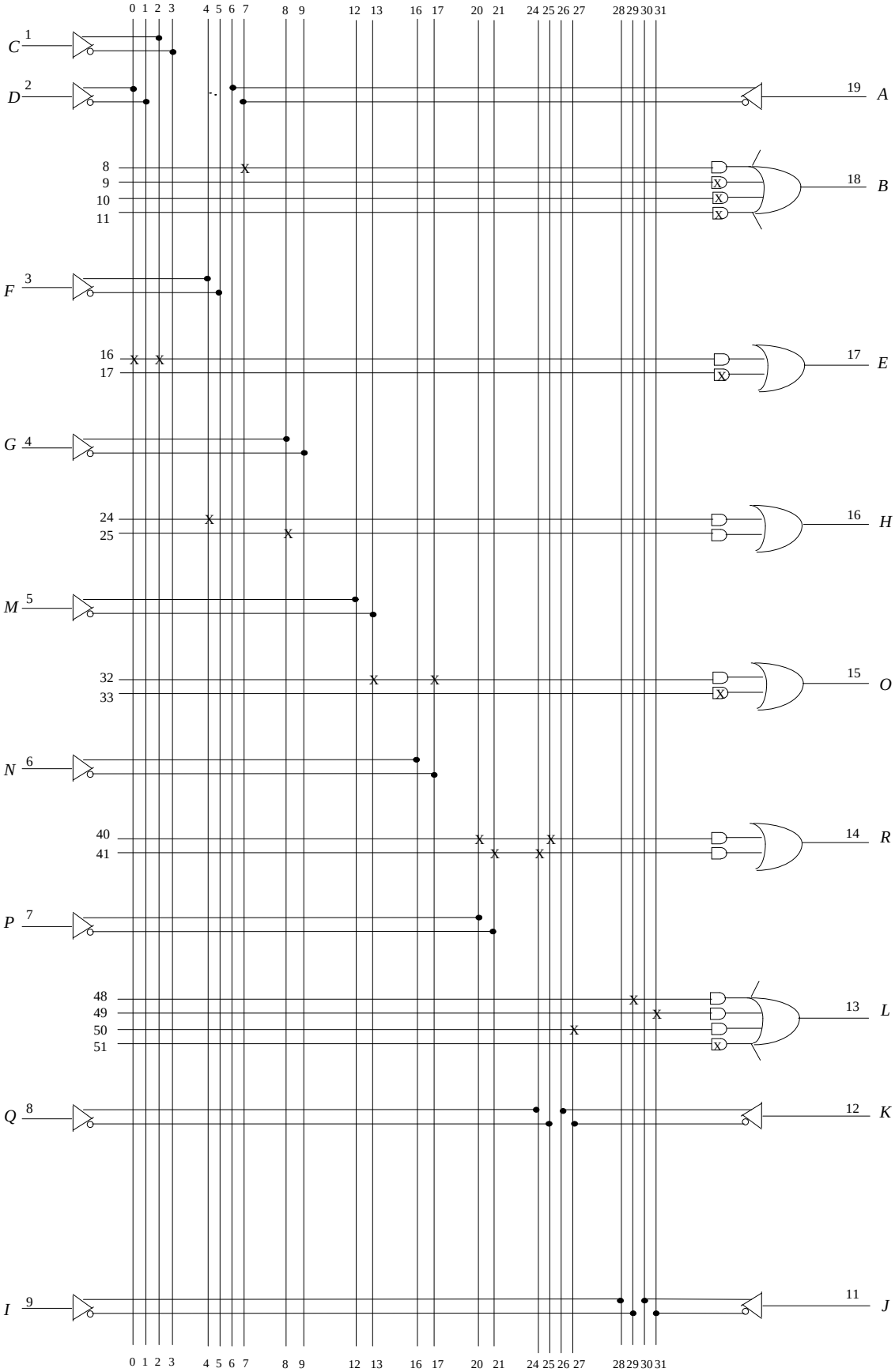
Exemples d'application :

a) Réalisation des portes élémentaires à l'aide d'un réseau PAL

Soit le réseau PAL suivant, programmé pour figurer les portes combinatoires représentées :



L'état interne de la matrice ET est alors la suivante :



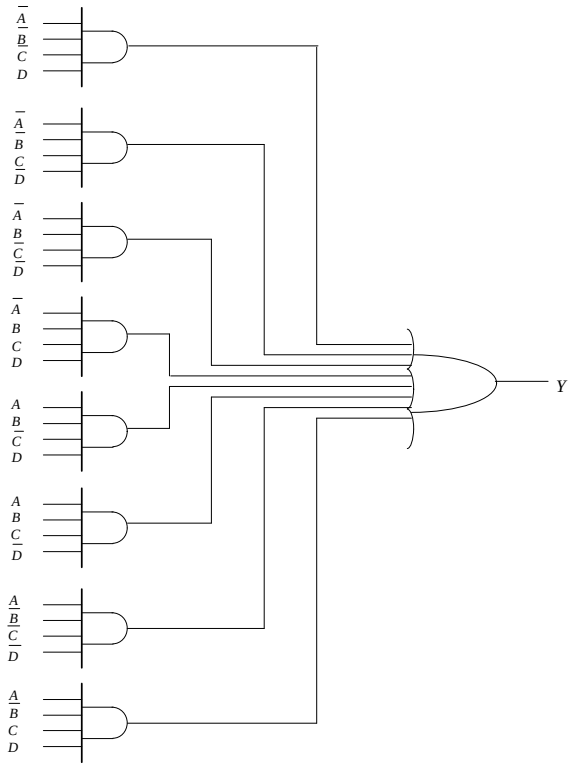
Légende

Fusible intact

Tous fusibles intacts

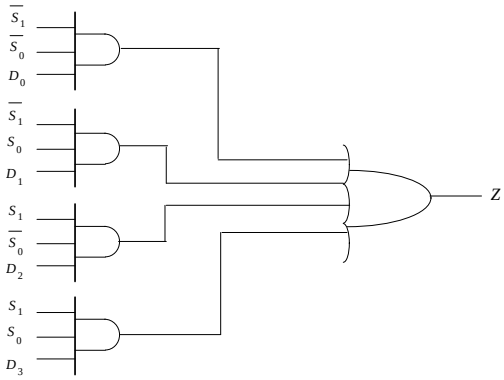
Fusible détruit

b) Réalisation d'un générateur de parité



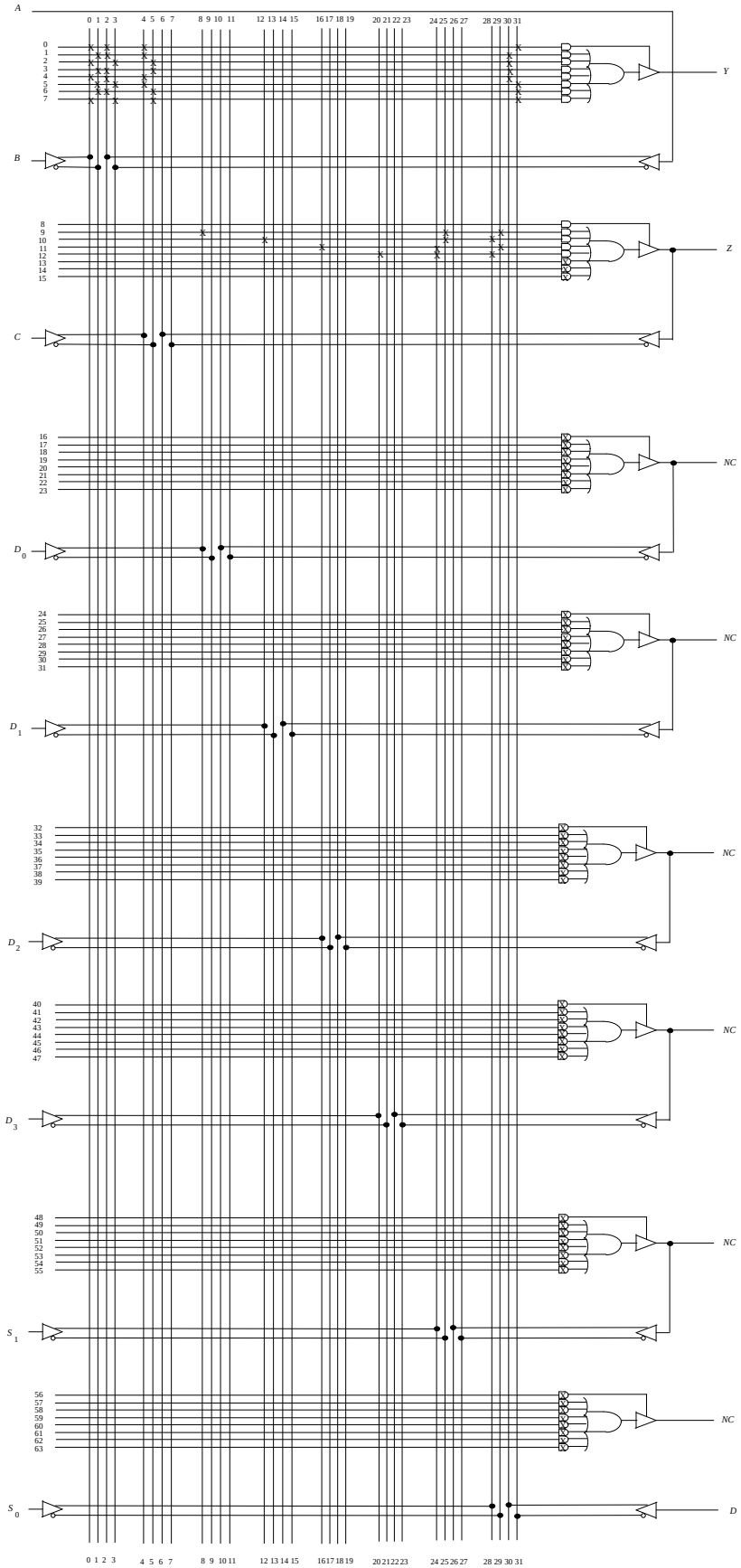
$$Y = \overline{A}\,\overline{B}\,\overline{C}\,D + \overline{A}\,\overline{B}\,C\,\overline{D} + \overline{A}\,B\overline{C}\,\overline{D} + \overline{A}\,BCD + AB\overline{C}\,D + ABC\,\overline{D} + A\overline{B}\,\overline{C}\,\overline{D} + A\overline{B}\,CD$$

c) Réalisation d'un multiplexeur 4 vers 1



$$Z = \overline{S_1}\,\overline{S_0}\,D_0 + \overline{S_1}\,S_0\,D_1 + S_1\,\overline{S_0}\,D_2 + S_1\,S_0\,D_3$$

L'état interne des fusibles est donné ci-après :



Légende

Fusible intact

Tous fusibles intacts

Fusible détruit

Non Connecté

NC

2.5. **PLA** *(Programmable Logic Array)* ou **PLD** *(Programmable Logic Device)* ou **CPLD** *(Complex PLD)* ou **EPLD** *(Erasable PLD)*

Dans un réseau PLA, les matrices ET et OU sont toutes les deux programmables.

2.6. **FPGA** *(Field Programmable Gate Array)*

Composés d'une matrice ET programmable, les réseaux FPGA n'ont pas de matrice OU. Chaque produit est relié directement à une sortie. La densité peut atteindre le million de portes logiques par circuit FPGA.

Utilisation des FPGAs

Le FPGA est un opérateur programmable spécialisé dans le décodage.

Exemple : Décodage des adresses fournies par un microprocesseur (microprocesseur MOTOROLA 6809)

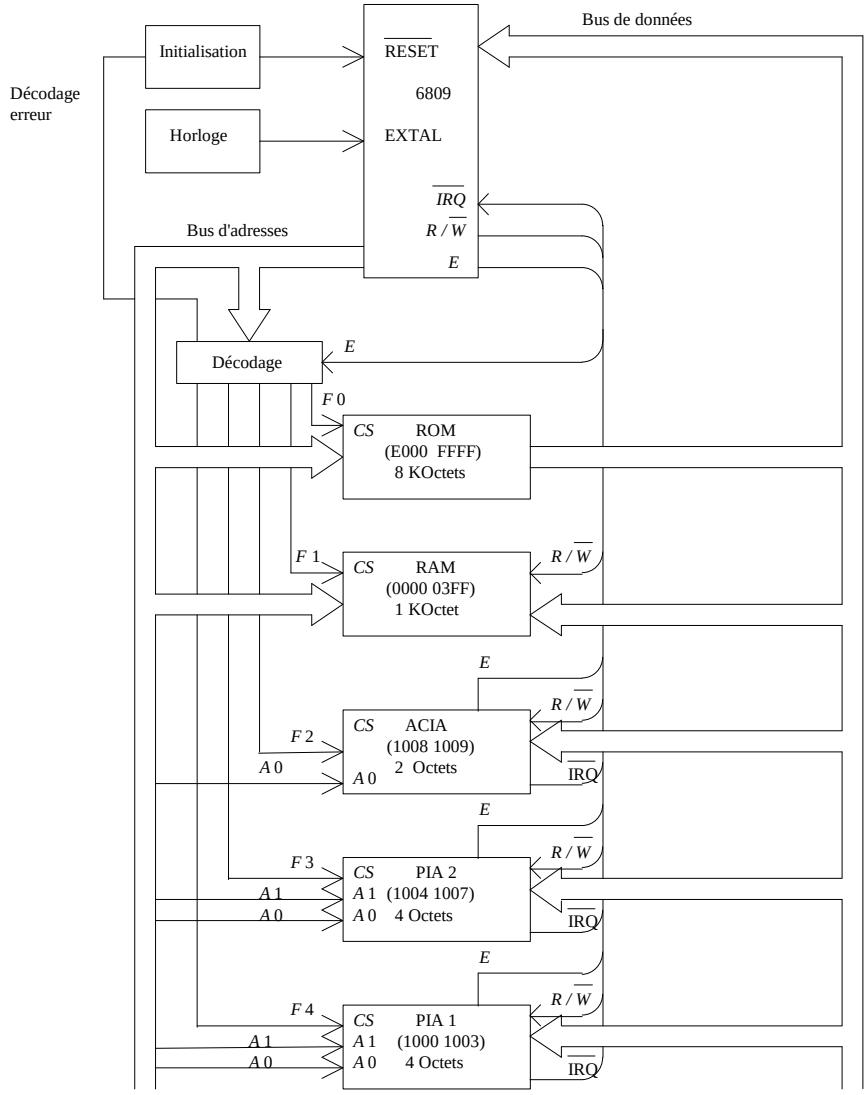


Table du FPGA

Sorties	Niveau actif	A_{15} A_{14} A_{13} A_{12} A_{11} A_{10} A_9 A_8 A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0 ↓ ↓ Variables d'entrée I_0 I_1 I_2 I_3 I_4 I_5 I_6 I_7 I_8 I_9 I_A I_B I_C I_D I_E I_F	
F_0	0	1 1 1 × × × × × × × × × × × × × ×	décodage ROM
F_1	0	0 0 0 0 0 0 × × × × × × × × × ×	décodage RAM
F_2	0	0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 ×	décodage ACIA
F_3	0	0 0 0 1 0 0 0 0 0 0 0 0 0 1 × ×	décodage PIA 1
F_4	0	0 0 0 1 0 0 0 0 0 0 0 0 0 0 × ×	décodage PIA 2
F_5	–	– – – – – – – – – – – – – – – –	
F_6	–	– – – – – – – – – – – – – – – –	Non utilisé
F_7	–	– – – – – – – – – – – – – – – –	
F_8	–	– – – – – – – – – – – – – – – –	

Légende : 0 : Niveau bas - : Etat initial des fusibles
1 : Niveau haut × : Variables n'étant pas utilisées.

2.7. RAM *(Random Access Memory)*

Par opposition aux ROMs, les mémoires vives (appeléées RAM) peuvent être lues et écrites. Contrairement aux PROMs, leur écriture n'est pas définitive dans le sens où le contenu des RAMs est perdu à l'extinction de l'alimentation électrique.
Le boîtier des RAMs possède, en plus de celui des ROMs, une entrée $R/\overline{W}$ (Read/Write) :

$R/\overline{W} = 1 \rightarrow$ lecture
$R/\overline{W} = 0 \rightarrow$ écriture

On distingue : - les RAMs statiques, constituées de Bascules élémentaires (Bascules D *(cf. logique séquentielle)*),
- les RAMs dynamiques, constituées de condensateurs (intégrables à plus grande échelle) mais qu'il est nécessaire de rafraîchir périodiquement pour en garder le contenu.
- les mémoires *flash* rapides et ne nécessitant pas d'alimentation pour sauvegarder leur contenu.

2.8. ASIC *(Application Specific Integrated Circuit)* *(Composant spécifique, dédié)*

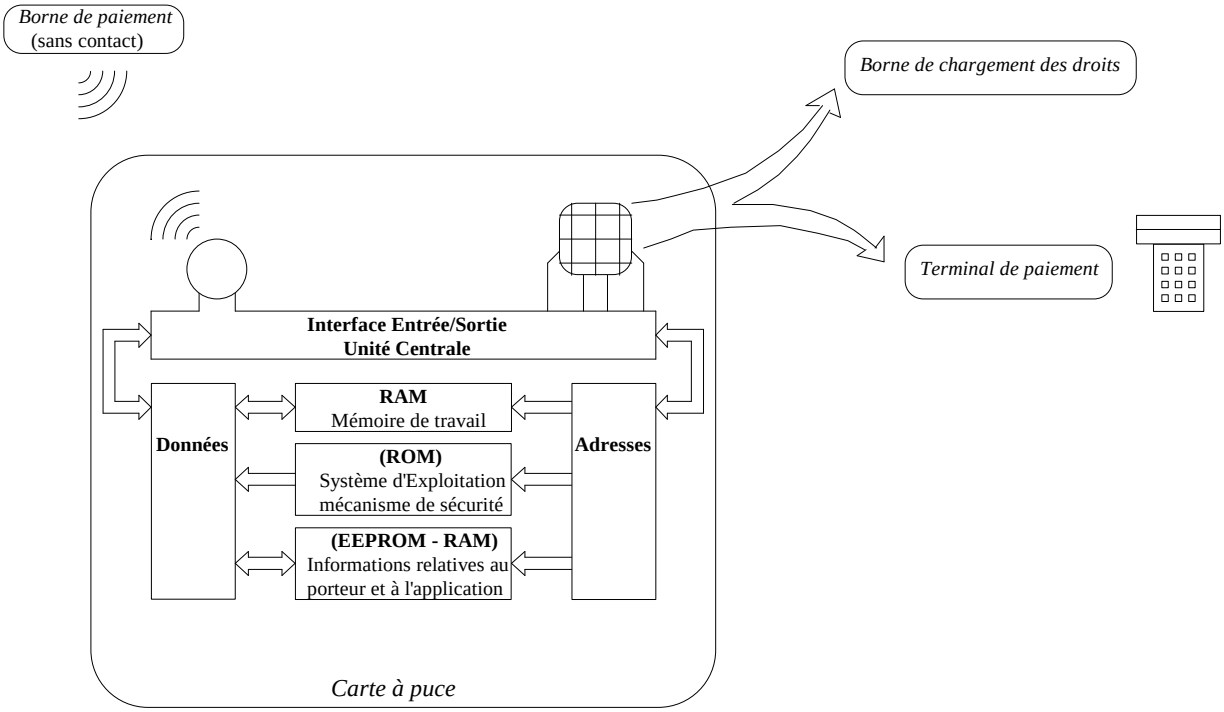
Un composant ASIC est développé spécifiquement pour une application analogique, numérique ou mixte, et destiné à une exploitation en nombre assez important. Quelques millions de portes peuvent être intégrées dans un ASIC. Des formats de bibliothèques d'ASICs sont développés (ALFAdvanced Library Format, OVI Open Verilog International). Leur rendement économique vaut à la condition d'une production à grande échelle.

2.9. Conception de PLD, de FPGA

Des langages (compilateurs) de conception (≡ spécification), de simulation et de programmation de ces composants ont été développés (langage VHDL, Verilog, C ...).

2.10. Application

Le synoptique d'une application monétique d'une carte à puce (avec ou sans contact) est présentée à titre d'exemple.

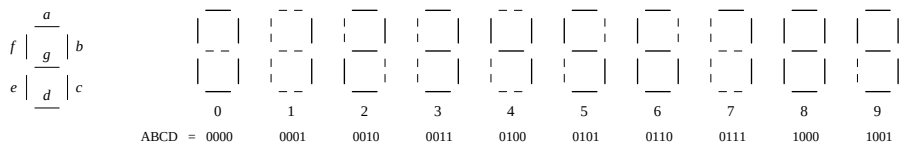

Borne de paiement
(sans contact)
Borne de chargement des droits
Terminal de paiement
Interface Entrée/Sortie
Unité Centrale
Données
RAM
Mémoire de travail
(ROM)
Système d'Exploitation
mécanisme de sécurité
(EEPROM - RAM)
Informations relatives au
porteur et à l'application
Adresses
Carte à puce

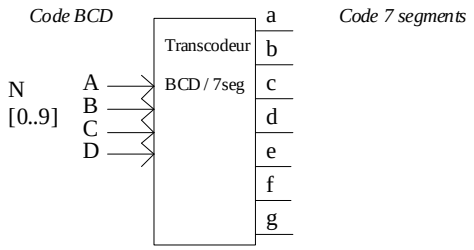
TD 2. LOGIQUE COMBINATOIRE 2

Transcodage

1. Transcodeur *BCD* - 7 segments

On désire afficher un chiffre de 0 à 9 codé en *BCD* (Décimal Codé Binaire) sur 4 bits $ABCD$ à l'aide d'un afficheur 7 segments (A est le bit de plus fort poids ≡ MSB). L'affichage se fait de la façon suivante :





a) Ecrire la table de transcodage.
b) Donner la fonction logique associée au segment a.
c) Donner la structure de réalisation du transcodeur.

Multiplexage

2. Multiplexeur pour fonction logique

Un multiplexeur peut matérialiser une fonction logique quelconque: *Ex:* $f_2(a,b) = a + b$. Schéma de réalisation ?

Transmission

3. Construction d'un code détecteur d'erreur

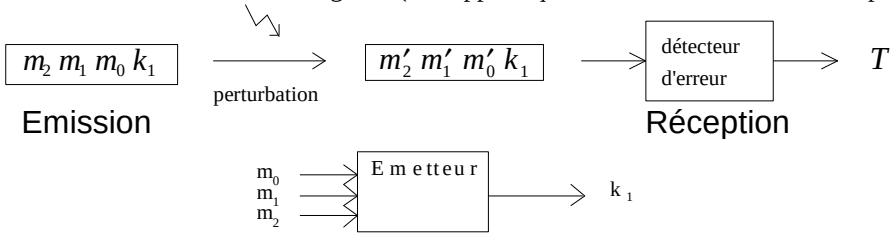
Les mots-code utilisés dans les transmissions numériques ont la structure générale suivante :

m bits d'information	*k* bits de contrôle

Le système de détection d'erreur le plus simple consiste à n'utiliser qu'un seul élément binaire de contrôle (k = 1). Cet élément est déterminé de telle sorte que le nombre de 1 parmi les (m+k) bits d'information est pair (cas du contrôle de parité, dit encore de parité paire), ou impair (cas du contrôle de parité impaire dit encore de parité impaire).

a) Emission : Déterminer un système capable de calculer cet élément de contrôle dans le cas m = 3 et k = 1 avec un contrôle de parité impaire.

b) Réception : Déterminer un système capable de détecter une erreur (1 seul bit modifié au maximum) de transmission dans ce même cas de figure : (on suppose que les bits de contrôle ne sont pas modifiés)



4. Système de transmission numérique avec correction d'une erreur

Dans un système de transmission, on veut une certaine sécurité, c'est à dire être capable de détecter et de corriger *une* erreur. Pour cela on utilise un codage particulier appelé « code de Hamming ».
Pour transmettre les quatre éléments binaires correspondant à un chiffre du système décimal, on ajoute trois éléments binaires pour assurer des contrôles de parité. Soient k_1 , k_2 , k_3 les trois bits de contrôle et m_1 , m_2 , m_3 et m_4 les quatre bits du message utile. La position relative des bits k_i et m_j est donnée par le tableau :

Numéro du bit	1	2	3	4	5	6	7
	k_1	k_2	k_3	m_1	m_2	m_3	m_4

On effectue trois tests de parité pour la détection de l'erreur :

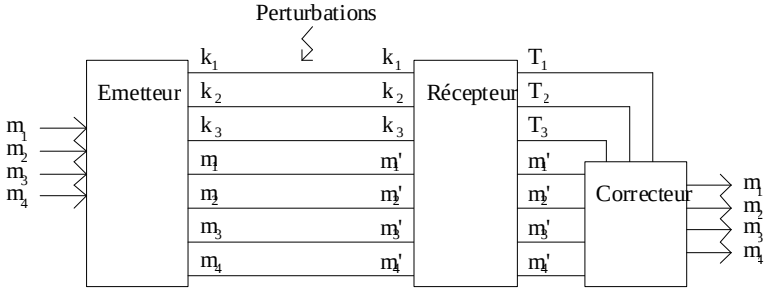
- le test de parité (test T_1 sur k_1) se fait sur les bits : 1, 4, 5, 7
- le test de parité (test T_2 sur k_2) se fait sur les bits : 2, 4, 6, 7
- le test de parité (test T_3 sur k_3) se fait sur les bits : 3, 5, 6, 7

On rappelle que le résultat d'un test de parité est égal à 0 si le nombre de 1 dans la zone considérée est pair (parité paire). La disposition est choisie de telle façon que le nombre binaire $(T_3T_2T_1)_2$ formé par les résultats des tests T_1, T_2 et T_3 donne la position du bit erronné.

1. Donner le schéma du dispositif « émetteur » permettant de générer les bits k_1 , k_2 et k_3.

2. Donner le schéma du dispositif « récepteur » permettant de générer les bits T_1, T_2 et T_3 .

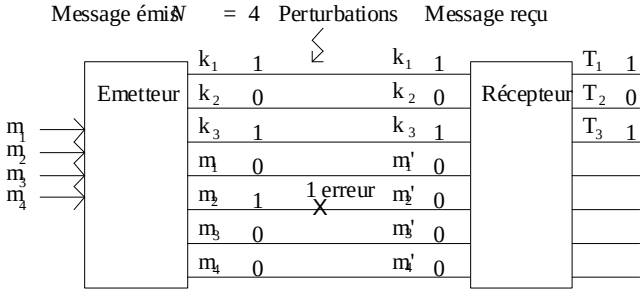
3. Proposer un dispositif simple réalisant la correction du bit erroné (1 seul bit au maximum peut être erroné ; on suppose que les bits de contrôle ne sont pas modifiés).



Code de Hamming (pour $0 \le N \le 9$)

N	k_1	k_2	k_3	m_1	m_2	m_3	m_4
0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	1
2	0	1	1	0	0	1	0
3	1	0	0	0	0	1	1
4	1	0	1	0	1	0	0
5	0	1	0	0	1	0	1
6	1	1	0	0	1	1	0
7	0	0	1	0	1	1	1
8	1	1	0	1	0	0	0
9	0	0	1	1	0	0	1

Exemple



$(T_3T_2T_1)_2 = (101)_2 = (5)_{10}$

Conclusion : le bit 5 (soit m_2) est erronné.

TD 2 ANNEXE. LOGIQUE COMBINATOIRE 2

Codage

1. Codeur BCD (déjà fait en cours)

Donner le schéma de réalisation du codeur en *BCD* (≡ Binary Coded Decimal) d'un chiffre *N* compris entre 0 et 9.

→ codeur à 10 entrées : $N \quad 0 \rightarrow 9$ et 4 sorties : $A\ B\ C\ D$ (*A* : *MSB (Most Significant Bit)*)

Décodage

2. Décodeur BCD (déjà fait en cours)

Donner le schéma de réalisation du décodeur d'un mot écrit en *BCD* sur 3 bits : $e_2\ e_1\ e_0$.

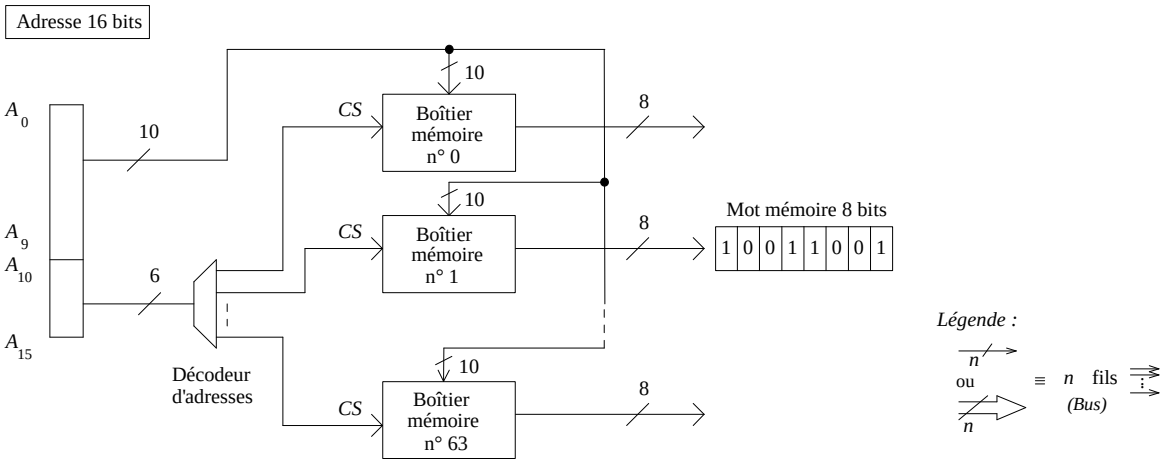
→ décodeur à 3 entrées : $e_2\ e_1\ e_0$ et 8 sorties : s_0 à s_7 (e_0 : *LSB (Less Significant Bit)*)

3. Décodeur d'adresses (déjà fait en cours)

Soit un microprocesseur délivrant une adresse sur 16 bits. Sa capacité d'adressage est donc : 2^{16} = 65536 mots de la mémoire.

Il est commode de partager cette mémoire en 64 pages de 1024 mots, chaque page pouvant correspondre à un boîtier mémoire.

La sélection du numéro de page, donc du boîtier correspondant (Chip Select *CS*) est effectuée par le décodage des 6 bits de poids fort parmi les 16 bits.
Les 10 bits restant permettant la sélection interne d'un mot mémoire : (de 8 bits par exemple)



Donner la structure du décodeur d'adresses.

Transcodage

4. Codeur de parité décimale (Transcodeur)

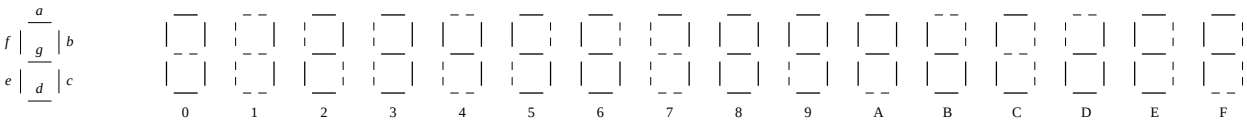
Donner le schéma de réalisation du codeur dont la sortie est à 0 si l'entrée *N* (chiffre entre 0 et 9 codé en *BCD*) est paire et 1 sinon.

5. Transcodeur *BCD* / *partie entière* (déjà fait en cours)

Soit *N* écrit en BCD sur 4 bits *ABCD*. On désire obtenir en sortie du transcodeur un mot *M* de 3 bits *XYZ* représentant en code *BCD* la partie entière de la moitié du nombre *N*. Donner le schéma de réalisation du transcodeur. (*A* et *X* : *MSBs (Most Significant Bits)*)

6. Transcodeur *hexadécimal* - 7 segments

On désire afficher un caractère hexadécimal de 0 à F (0 à 9 puis A à F) codé en *hexadécimal* sur 4 bits $ABCD$ à l'aide d'un afficheur 7 segments (A est le bit de plus fort poids). L'affichage se fait de la façon suivante :



a) Ecrire la table de transcodage.
b) Donner la fonction logique associée au segment a.
c) Donner la structure de réalisation du transcodeur.

7. Transcodeur *SVA* / *Cà2*

Soit *N* écrit en code SVA (Signe et Valeur Absolue) sur 3 bits *ABC*. On désire obtenir en sortie du transcodeur un mot *M* de 3 bits *XYZ* représentant *N* en code *Cà2* (Complément à 2). (*A* et *X* : *MSBs (Most Significant Bits)*)

N	*Code SVA* ABC	→	*Code Cà2* XYZ
+3	011		011
+2	010		010
+1	001		001
+0	000		000
-0	100		000
-1	101		111
-2	110		110
-3	111		101
-4	-		100

Multiplexage

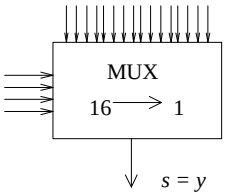
8. Multiplexeur de mot (déjà fait en cours)

Donner le schéma de réalisation du circuit de sélection d'un mot de 3 bits parmi 4 mots de 3 bits.

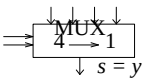
9. Multiplexeur pour fonction logique

Soit la fonction logique *y* définie par sa table de Karnaugh. Utiliser un mutliplexeur pour engendrer *y* :

y *a b* / *c d*	*00*	*01*	*11*	*10*
00	0	0	1	1
01	0	0	0	0
11	1	1	0	0
10	1	1	0	0



Remarque : il est aussi possible d'utiliser un MUX 4 -> 1



10. Multiplexeur pour conversion parallèle / série (déjà fait en cours)

Donner le chronogramme de la séquence $A_1\ A_0$ à envoyer sur les entrées d'adresses du multiplexeur pour convertir le mot $D_3\ D_2\ D_1\ D_0$ de parallèle en série.

Démultiplexage

11. Démultiplexeur pour conversion série / parallèle (déjà fait en cours)

Même question que précédemment pour convertir la séquence $D_0\ D_1\ D_2\ D_3$ de série en parallèle.
Quelle fonction est en plus nécessaire

Circuits arithmétiques

12. Addition - Comparaison - Parité (déjà fait en cours)

Addition

1. Soient 2 bits *a* et *b*. Donner le circuit élémentaire réalisant la somme arithmétique entre *a* et *b* (appelé demi-additionneur) : (*1/2 ADD*).
2. Soit en plus de a_i et b_i, le bit r_i figurant la retenue de la somme élémentaire précédente entre a_{i-1} et b_{i-1} lorsque l'on désire faire la somme de 2 mots $A = a_{n-1}\ a_{n-2} \cdots a_0$ et $B = b_{n-1}\ b_{n-2} \cdots b_0$

   a) Donner le circuit additionneur complet (*ADD*) entre a_i b_i et r_i.

   b) Donner l'additionneur à propagation de retenue entre les mots *A* et *B*.

   c) Donner l'additionneur à retenue anticipée (plus rapide) entre les mots *A* et *B*.

Comparaison

3. Soient 2 nombres $A = a_3\ a_2\ a_1\ a_0$ et $B = b_3\ b_2\ b_1\ b_0$.

   Donner le circuit dont la sortie vaut 1 si les nombres *A* et *B* sont égaux.

Parité

4. On appelle parité d'un mot binaire *N* le nombre de 1 contenus dans ce mot : le mot a une parité paire si ce nombre de 1 est pair. Afin de rendre les transmissions numériques plus robustes au bruit, on adjoint à *N* (et à chaque mot transmis) un bit dit de parité, dont la valeur est telle que le mot global formé de *N* et de ce bit de parité, ait une parité paire.

   Donner le circuit générant ce bit de parité.

_____

TP 2. LOGIQUE COMBINATOIRE 2

1. Matériel nécessaire

- Oscilloscope
- Générateur de signaux Basses Fréquences (GBF)
- Alimentation stabilisée (2x[0-30 V] ⎓ + 1x[5 V] ⎓)
- Multimètre
- Moniteur MS05 (plaquette de câblage)
- Câbles : - 1 T, 1 BNC-BNC, 1 BNC-Banane, 1sonde oscilloscope, 6 fils Banane, petits fils.

- Composants : - 7 Résistances 1 kΩ (1/4 Watt)
- 1 afficheur 7 segments à cathodes communes :
  Réf.: HDSP-5503 (10 mA) ou HDSP-7513 (2 mA).
- (1 minuterie NE 555 (circuit compatible TTL et CMOS))
- 5 LEDs rectangulaires (4 Vertes + 1 Rouge)
- 5 mini-interrupteurs

Circuits logiques de la famille CMOS 4000 :

- 1 *4030* : 4 XOR à 2 entrées
- 1 *4071* : 4 OR à 2 entrées
- 1 *4081* : 4 AND à 2 entrées
- 1 *4511* : Transcodeur BCD / 7 segments
- 1 *4520* : Compteur binaire
- 1 MUX 4→1
- 1 DEMUX 1→4

2. Notation du TP

Faire examiner par le professeur en fin de séance, les différentes parties du TP.

3. Etude Théorique

Additionneurs - Codeurs - Comparateurs

3.0. Technologie

2 grandes familles se dégagent principalement :

- la famille TTL (séries *74* et *54*) qui matérialise un 1 logique par une tension de + 5 Volts (à ≈ 1 Volt près)
- la famille CMOS (séries *4000* et *40000*) qui tolère des tensions supérieures, mais pour laquelle on code en général également un 1 logique par une tension de + 5 Volts.

Dans les 2 cas, le 0 logique est matérialisé par une tension nulle (à ≈ 1 Volt près).

On rappelle qu'une entrée d'un circuit laissée « en l'air » (≡ non connectée) se comporte comme une antenne, et prendra donc généralement le niveau logique 1 (du fait du rayonnement électromagnétique) ou bien le niveau logique 0 en cas de réception faible. Il est donc nécessaire de fixer les potentiels des entrées des portes utilisées pour les contrôler.

Ne pas oublier qu'une porte logique dont la sortie est au niveau logique haut (1) se comporte comme toujours comme un générateur de tension avec résistance interne. Le niveau 1 (+ 5 Volts) est maintenu tant que le courant de sortie ne dépasse pas l'ordre de la dizaine de mA, suffisant pour attaquer une LED par ex.

On pourra intercaler une résistance (≈ 1 kΩ) de limitation de courant entre la sortie à visualiser et la LED, ou connecter directement la LED à la sortie du circuit logique, la résistance interne de la porte logique faisant office de limiteur de courant sans perdre le niveau logique (pour une LED connectée en sortie).

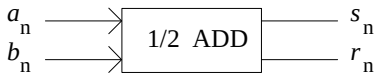
Simulation (& Câblage) :

Pour des raisons de compatibilité, n'utiliser que des circuits de la même famille (famille CMOS 4000 à ne pas mélanger avec la famille TTL 74xxx).

3.1. Les additionneurs

a) Demi-additionneur

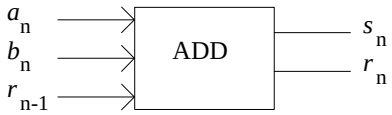
Soient a_n et b_n 2 bits à additionner.
L'expression binaire de $(a_n + b_n)$ s'écrit (r_n, s_n) où r_n représente la retenue générée de l'opération et s_n la somme.



- Donner la table de vérité du demi-additionneur (i.e. pas de prise en compte de l'éventuelle retenue r_{n-1} issue d'un étage additionneur précédent dans le cadre d'une addition de 2 mots binaires).
- Tracer le tableau de Karnaugh des deux fontions r_n et s_n.
- Donner les fonctions logiques correspondantes r_n et s_n en utilisant de préférence des portes OUX.
- Donner le schéma symbolique de réalisation.

b) Additionneur complet

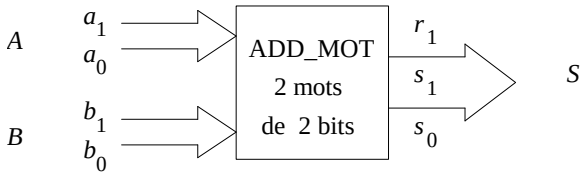
Soient a_n et b_n 2 bits à additionner en tenant compte ici de la retenue précédente r_{n-1} issue d'un étage additionneur précédent (dans le cadre d'une addition de 2 mots binaires).



- Donner la table de vérité de l'additionneur complet.
- Tracer le tableau de Karnaugh des deux fontions r_n et s_n.
- Donner les fonctions logiques correspondantes r_n et s_n en faisant de préférence apparaître des portes OUX plutôt qu'en simplifiant les fonctions au maximum.
- Donner le schéma symbolique de réalisation.

c) Additionneur de 2 mots de 2 bits

Soient 2 mots de 2 bits $A = a_1 a_0$ et $B = b_1 b_0$ à additionner (a_1, b_1 : MSB). En utilisant les résultats précédents, donner les expressions des sorties et de la retenue générée : $s_1 s_0$ et r_1 de l'additionneur de 2 mots de 2 bits (s_1 : « MSB ») après avoir établit sa table de vérité.



- Donner le schéma symbolique de réalisation.

3.2. Les codeurs

Rappels

Un code est la représentation d'un nombre tel qu'à chaque nombre corresponde une configuration et une seule, et qu'à chaque configuration ne corresponde qu'un seul nombre.

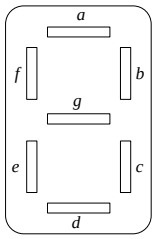
En général, le codage d'un nombre décimal s'effectue en codant chaque chiffre qui le compose. Le nombre codé est obtenu en juxtaposant leurs représentations.

Pour coder en binaire les chiffres de 0 à 9, il faut au moins 4 bits. Il existe de nombreuses possibilités pour coder ces chiffres. Cependant, seuls quelques uns revêtent une grande importance.

Le transcodage est une opération qui consiste à passer d'un code à un autre code.

Transcodeur BCD - 7 segments

Un afficheur 7 segments est constitué de 7 LEDs ayant toutes un point commun : l'anode (A) ou la cathode (C). Si l'afficheur est à anodes communes, il est nécessaire d'inverser (≡ de complémenter) le signal à afficher pour qu'une LED allumée corresponde à un 1 logique.



- Donner la table de vérité d'un transcodeur permettant de passer du code BCD (Décimal Codé Binaire) au code permettant l'affichage sur 7 segments *a* à *g* d'un mot *ABCD* de 4 bits (*A* : MSB).
- Tracer les tableaux de Karnaugh de chaque segment en simplifiant les fonctions au maximum, et en faisant apparaître si-possible des OU exclusifs.
- En déduire les fonctions correspondantes.
- Donner le schéma symbolique de réalisation du transcodeur.

3.3. Les comparateurs

On désire réaliser un montage permettant de comparer 2 mots binaires de 2 bits $A = a_1\, a_0$ et $B = b_1\, b_0$ (a_1 , b_1 : MSB).

Ce montage doit permettre d'identifier les 3 cas suivants : $A > B$, $A = B$, $A < B$.

- Donner la table de vérité du montage.
- Tracer les tableaux de Karnaugh des 3 fonctions de comparaison en les simplifiant au maximum.
- Donner les fonctions correspondantes.
- Donner le schéma symbolique de réalisation du comparateur.

Multiplexeur - Démultiplexeur

3.4. Multiplexeur

a) Rappeler la définition d'un multiplexeur à 4 entrées de données $D_3\, D_2\, D_1\, D_0$, 2 entrées d'adresses $A_1\, A_0$ (D_3 et A_1 sont les MSB) et de sortie s.

b) Donner sa table de vérité.

c) Donner l'équation de la sortie.

d) Etablir le schéma logique permettant de le réaliser en utilisant exclusivement des portes logiques ET 4 entrées, OU 4 entrées et NON plutôt que d'utiliser un multiplexeur tout fait.

e) Facultatif : Proposer un schéma, à base d'une minuterie NE 555 monté en oscillateur carré et d'une bascule D, permettant de réaliser dans le montage de la figure 3, la sélection automatique des adresses dans la séquence :

$A_1\, A_0$ =	00	→	01	→	10	→	11	→	...
notée :	①	→	②	→	③	→	④	→	...

avec une fréquence de changement d'adresse (passage de ① à ②, de ② à ③, ..) d'environ 1 Hz.
(On pourra s'aider en traçant préalablement le chronogramme des variables A_0 et A_1 supposées initialement à l'état bas).

3.5. Démultiplexeur

a) Rappeler la définition d'un multiplexeur à 4 sorties de données $D_3\, D_2\, D_1\, D_0$, 2 entrées d'adresses $A_1\, A_0$ (D_3 et A_1 sont les MSB) et d'entrée e.

b) Donner sa table de vérité.

c) Donner l'équation des sorties.

d) Etablir le schéma logique permettant de le réaliser en utilisant exclusivement des portes logiques ET 4 entrées et NON plutôt que d'utiliser un démultiplexeur tout fait.

4. Etude Expérimentale

4.0. Test des composants (en câblage uniquement)

En connectant le mini-interrupteur alternativement à la masse et à la tension + 5 Volts, on obtient ainsi des transitions franches permettant de simuler une horloge lente de test des composants synchrones (compteur ...).

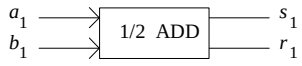
Tester chaque module d'un circuit séparément des autres modules.

Additionneurs - Codeurs - Comparateurs

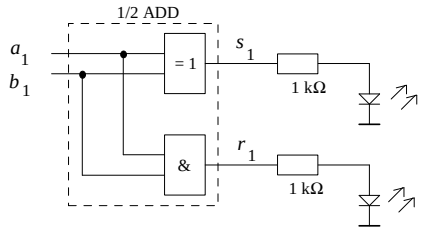
4.1. Les additionneurs (facultatif)

a) Demi-additionneur (facultatif)

Soient a_1 et b_1 2 bits à additionner.



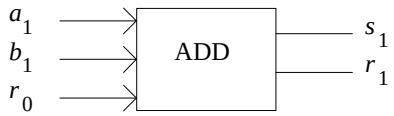
Simuler le montage suivant **[en simulation numérique, remplacer l'ensemble (résistance-LED) par un simple *logic display*)]** :



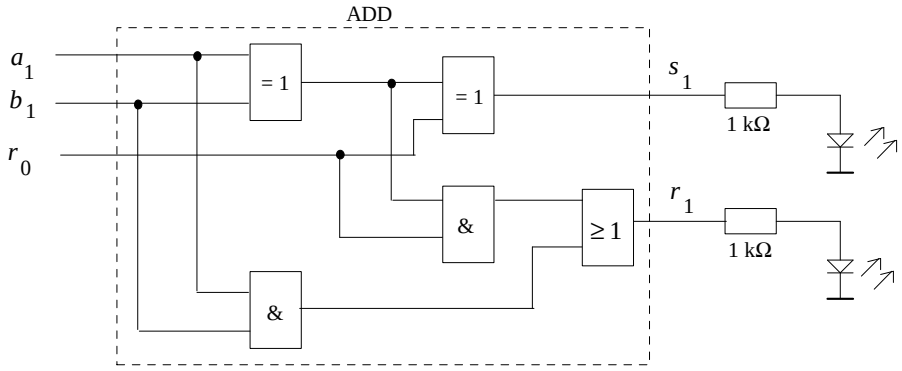
Vérifier la table de vérité du montage.

b) Additionneur complet (facultatif)

Soient a_1 et b_1 2 bits à additionner en tenant compte ici de la retenue précédente r_0.



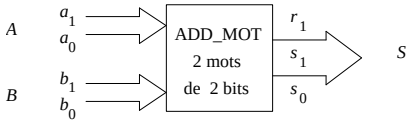
Réaliser le montage suivant (simulation) : **[en simulation numérique, remplacer l'ensemble (résistance-LED) par un simple *logic display*)]**



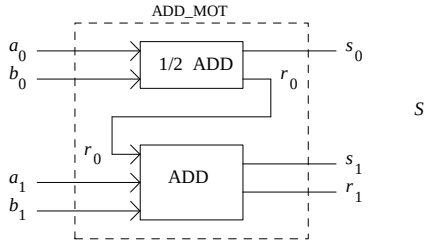
Vérifier la table de vérité du montage.

c) Additionneur de 2 mots de 2 bits (simulation) (facultatif)

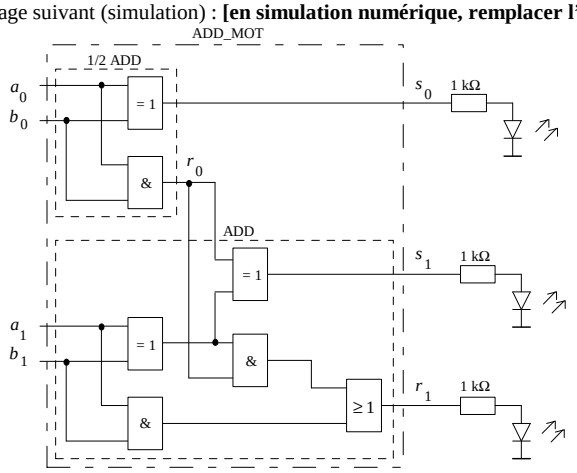
Soient 2 mots de 2 bits $A = a_1\,a_0$ et $B = b_1\,b_0$ à additionner (a_1 , b_1 : MSB).



Le schéma synoptique suivant utilise avantageusement les étages précédents demi-additionneur et additionneur :



Réaliser le montage suivant (simulation) : **[en simulation numérique, remplacer l'ensemble (résistance-LED) par un simple *logic display*)]**



Vérifier la table de vérité du montage.

4.2. Les codeurs

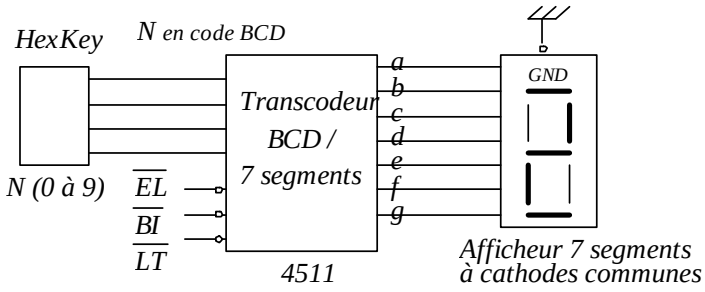
a) *(facultatif)* - Vérifier le bon fonctionnement du transcodeur BCD/7 segments déterminé dans la partie théorique pour le seul segment *a*) (simulation).

b) Tester le transcodeur BCD/7 segments tout fait (circuit 4511) pour lequel on câblera (simulation) tous les segments (*a* à *g*). (Télécharger au préalable le datasheet du circuit Transcodeur BCD/7 segments 4511).

L'afficheur 7 segments (à cathodes communes) sera connecté directement aux sorties du transcodeur ou des portes logiques sans intercaler de résistances de limitation de courant dans les LEDs de l'afficheur, la limitation se faisant déjà par la résistance interne du transcodeur ou des portes.

Note : Le Test du Transcodeur BCD/7 segments (4511) peut être fait avec l'objet Hexkey du simulateur (Switches -> Digital -> Hexkey) qui transforme un chiffre (de 0 à 9) en son code BCD sur 4 bits :

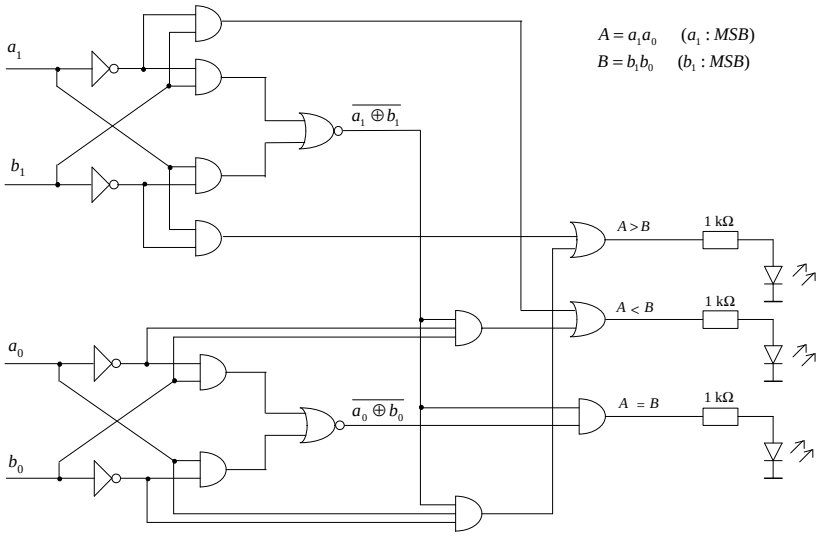
Application



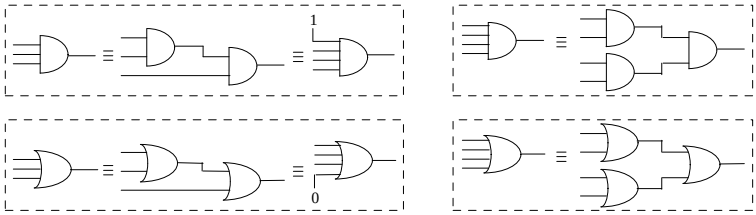
4.3. Les comparateurs (facultatif)

a) - *(facultatif)* Comparer, à l'aide du simulateur, avec le montage suivant, réalisant la même opération de comparaison, mais pour lequel les fonctions logiques décrivant les sorties n'ont pas été simplifiées au maximum (ne pas utiliser les LEDs témoins pour la simulation) :

[en simulation numérique, remplacer l'ensemble (résistance-LED) par un simple *logic display*)]



Note : Plutôt que d'utiliser des portes à 3 entrées, on pourra utiliser des portes à 2 ou 4 entrées :



b) - *(facultatif)* Utiliser le simulateur numérique pour tester le montage établi en cours.

Multiplexeur - Démultiplexeur

4.4. Multiplexeur

Schéma de principe :

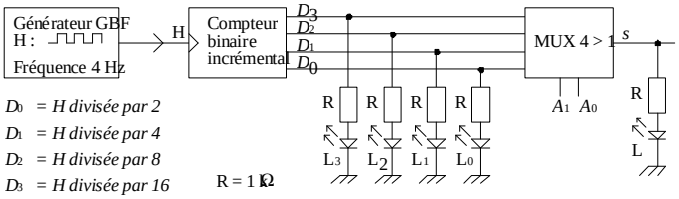


Figure 3

Simuler le multiplexeur à 4 entrées réalisé selon le schéma vu en cours avec des portes élémentaires et l'insérer dans le montage suivant :

Schéma de simulation :

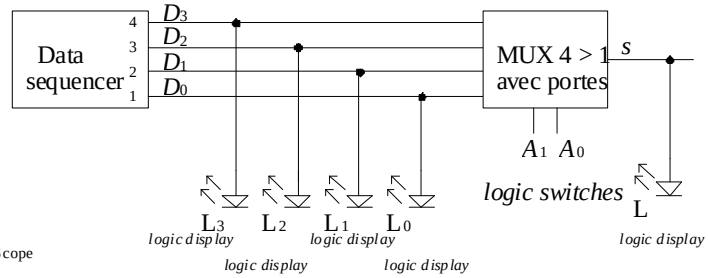


Figure 3'

Oscilloscope =Instruments / Digital / Scope

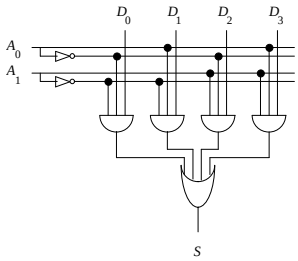
Réglages du Data sequencer :

address 0001 00001001
address 0002 00001000
address 0003 00001011
address 0004 00001110
start address : 1
stop address : 4

Résultat :

La LED L0 clignote le plus rapidement
La LED L1 clignote un peu moins vite que L3
La LED L2 clignote un peu moins vite que L2
La LED L3 est toujours allumée
La LED L clignote au même rythme que la LED Li selon l'adresse A1 A0 sélectionnée

Schéma du MUX 4 > 1 vu en cours :



Un compteur binaire incrémental est un circuit séquentiel dont le mot de sortie $D_3\ D_2\ D_1\ D_0$ (D_3 = MSB) est incrémenté (ou décrémenté s'il s'agit d'un décompteur ou compteur décrémental) à chaque fois que l'entrée d'horloge H est active (activation sur front ou sur niveau).

Le compteur utilisé, à synchronisation sur front montant de H, incrémente le mot $D_3\ D_2\ D_1\ D_0$ à chaque front montant de H. Le comptage s'effectue de façon circulaire : ainsi au mot 1111 succède le mot 0000 puis 0001, 0010, 0011 ...

L'état initial du compteur peut être, si on le désire, mais ça n'est pas nécessaire dans l'utilisation que l'on fait ici, réglé au mot voulu, 0000 par exemple.

a) En essayant les différentes combinaisons d'adresses, vérifier le bon fonctionnement du multiplexeur en comparant la sortie *S* à l'entrée de donnée sélectionnée.

b) *(facultatif)* Compléter le montage par le circuit de sélection automatique des adresses avec comme valeurs de composants de la minuterie montée en astable : R_A = 1 kΩ, R_B = 1 MΩ et C = 1 µF.

c) *(facultatif)* Remplacer le multiplexeur 4 > 1 discret (avec les portes ci-dessus) par un multiplexeur 4 > 1 intégré du simulateur (circuit 4539).

4.5. Démultiplexeur

Schéma de principe :

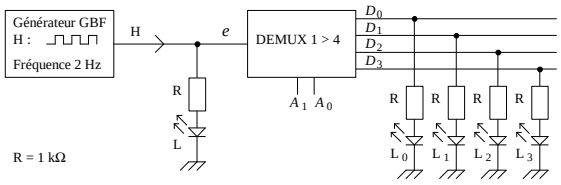


Figure 4

Simuler le démultiplexeur à 4 sorties réalisé selon le schéma vu en cours avec des portes élémentaires et l'insérer dans le montage suivant :

Schéma de simulation :

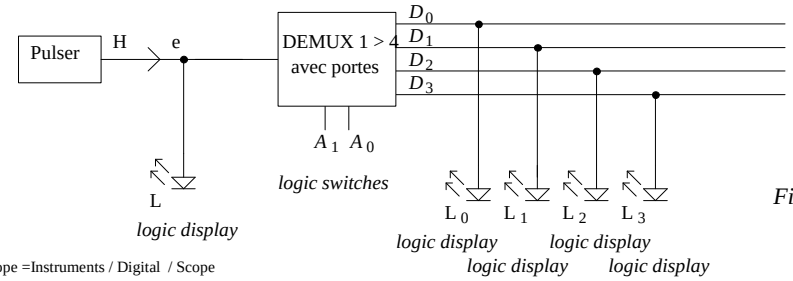
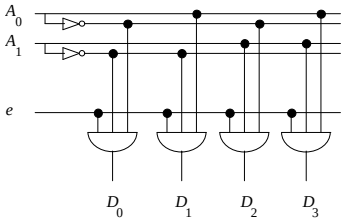


Figure 4'

Oscilloscope =Instruments / Digital / Scope
Horloge = Digital / Instruments / Pulser

Schéma du DEMUX 1 > 4 vu en cours :



a) En essayant les différentes combinaisons d'adresses, vérifier le bon fonctionnement du démultiplexeur en comparant une donnée de sortie à l'entrée e.

b) *(facultatif)* Compléter le montage par le circuit de sélection automatique des adresses avec les mêmes valeurs de composants de la minuterie montée en astable : $R_A = 1\ k\Omega$, $R_B = 1\ M\Omega$ et $C = 1\ \mu F$.

c) *(facultatif)* Remplacer le démultiplexeur 1 > 4 discret (avec les portes ci-dessus) par un démultiplexeur 1 > 4 intégré du simulateur (circuit 4555).

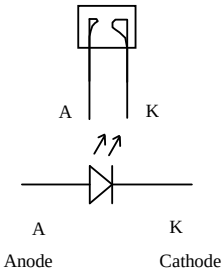
Rangement du poste de travail

Examen des différentes parties du TP et rangement (0 pour tout le TP sinon).

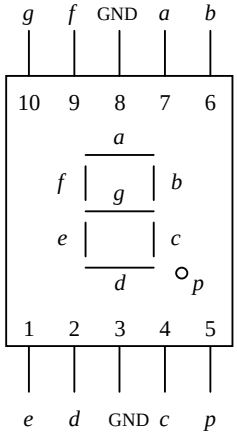
ANNEXE : DOCUMENTATION DES COMPOSANTS

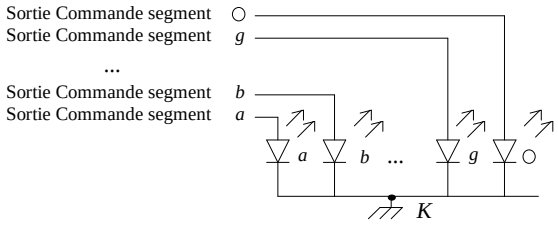
ANNEXE

- LED



- Brochage de l'afficheur 7 segments (cathodes communes *K*): **Réf.: HDSP-5503 (10 mA) ou HDSP-7513 (2 mA)**

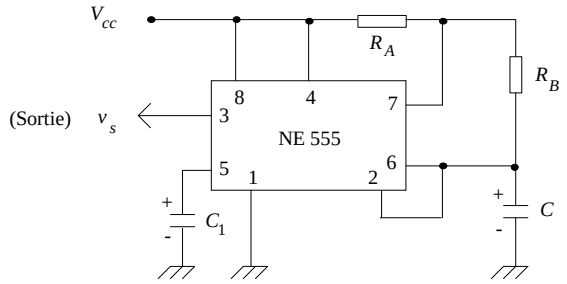




GND : masse *(Cathodes K communes)*

(Les pins 3 et 8 sont connectées de façon interne)

- Minuterie NE 555 montée en astable (≡ oscillateur) CI NE 555 ou équivalent (SN 72555 ou SFC 2555)

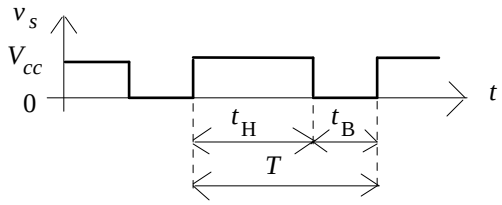


V_{cc} = + 5 Volts

C_1 = 10 nF

R_A *doit être différent de 0*

L'allure du signal de sortie est la suivante :



Sachant que : t_H correspond à la charge de *C* à travers $(R_A + R_B)$: $t_H = 0.693(R_A + R_B)C$

t_B correspond à la décharge de *C* dans R_B : $t_B = 0.693\, R_B\, C$

$T = t_H + t_B = 0.693(R_A + 2R_B)C$: période de $v_s(t)$ $f = 1/T \approx \dfrac{1.44}{(R_A + 2R_B)C}$: fréq. de $v_s(t)$

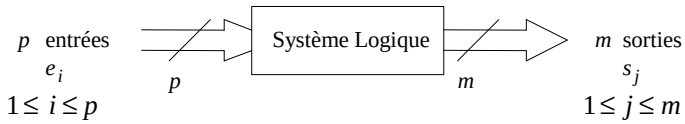
Rapport cyclique de $v_s(t)$: $R_0 = \dfrac{t_H}{T} \times 100\,\% = \dfrac{R_A + R_B}{R_A + 2R_B}$ 50 % < R_0 < 100 %

R_0 = 50 % pour $R_B >> R_A$ R_0 = 100 % pour $R_B = 0$

3. LOGIQUE SEQUENTIELLE 1 - LES BASES

1. RAPPELS

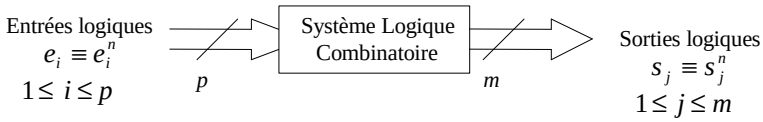
Système logique



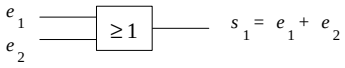
Système logique combinatoire

A l'instant discret n, une sortie s_j, notée s_j^n , d'un système logique combinatoire ne dépend que de ses entrées $e_1^n,\ldots,e_p^n$ au même instant : (la seule connaissance des entrées suffit à déterminer les sorties)

$$s_j^n = f(e_1^n,\ldots,e_p^n) \quad (1 \le j \le m)$$



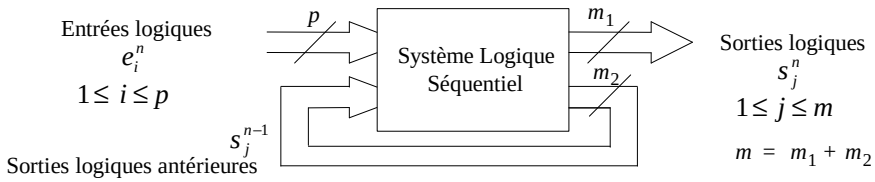
Exemple



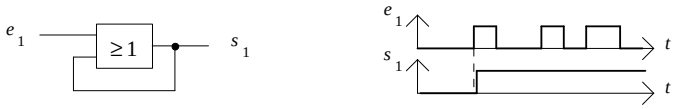
Système logique séquentiel (≡ système logique combinatoire bouclé)

A l'instant discret n, une sortie s_j^n d'un système logique séquentiel dépend de ses entrées $e_1^n,\ldots,e_p^n$ mais aussi de l'état antérieur des sorties ($s_1^{n-1},\ldots,s_m^{n-1}$) qui peuvent être considérées comme des entrées secondaires, alors que les entrées $e_1^n,\ldots,e_p^n$ sont appelées primaires. (Notion de mémoire, car les systèmes séquentiels sont bouclés, ou encore récursifs) : (la seule connaissance des entrées (primaires) ne suffit pas à déterminer l'état des sorties)

$$s_j^n = f(e_1^n,\ldots,e_p^n, s_1^{n-1},\ldots,s_m^{n-1}) \quad (1 \le j \le m)$$



Exemple



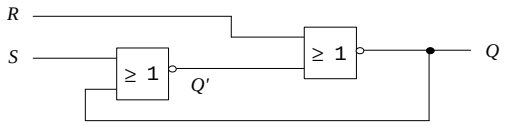
Supposons $s_1 = 0$ initialement (état initial lié à la technologie employée) → s_1 = mémorisation de ($e_1 = 1$) (dès que e_1 passe à 1, s_1 passe à 1 et y reste ensuite $\forall$ e_1).

Définition

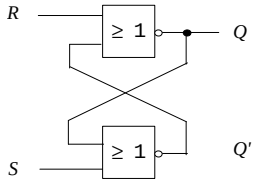
Bascule : circuit séquentiel dont les sorties possèdent 2 états stables, ces sorties étant complémentaires Q et $\overline{Q}$. (Bascule ≡ Bistable ≡ *Flip- flop*)

2. EXEMPLE FONDAMENTAL : LA BASCULE *RS*

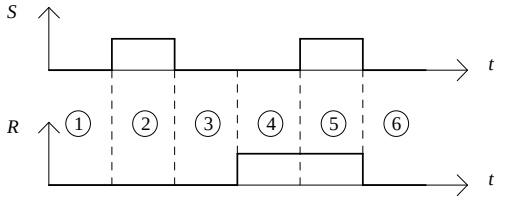
Soit le système séquentiel (bascule *RS*) : (*R* ≡ *Reset* ≡ Mise à 0 / *S* ≡ S*et* ≡ Mise à 1)



pouvant être représenté plus simplement :



Soit la séquence d'entrée suivante :



Supposons l'état initial suivant des sorties : Initialement : sortie *Q* au repos (*Q* = 0) et sortie *Q'* = 1. On a :

Phase	***S***	***R***	***Q***	***Observations*** (légende : → : passe à ≡ : reste à)	***Conclusion***
Phase ①	0	0	0	Etat initial stable : $Q \equiv 0$; $Q' \equiv 1$;	*Mémorisation* de l'état précédent $Q' = \overline{Q}$
Phase ②	1	0	1	$S \rightarrow 1$ donc $Q' \rightarrow 0$ donc $Q \rightarrow 1$; Etats stables	*Set* de Q (Mise à 1 de *Q*) $Q' = \overline{Q}$
Phase ③	0	0	1	$S \rightarrow 0$ donc $Q' \equiv 0$ donc $Q \equiv 1$; Etats stables	*Mémorisation* de l'état précédent $Q' = \overline{Q}$
Phase ④	0	1	0	$R \rightarrow 1$ donc $Q \rightarrow 0$ donc $Q' \rightarrow 1$; Etats stables	*Reset* de Q (Mise à 0 de *Q*) $Q' = \overline{Q}$
Phase ⑤	1	1	0	$S \rightarrow 1$ donc $Q' \rightarrow 0$ donc $Q \equiv 0$; Etats stables	*Combinaison interdite*, car $Q' \neq \overline{Q}$
Phase ⑥	0	0		Aléa de fonctionnement : l'état *Q* dépend de la rapidité relative entre les 2 portes, car les 2 entrées S et R changent d'état simultanément. (Si la porte NOR d'entrée *R* est plus rapide que celle d'entrée *S*, on a : $Q = 1$ et $Q' = 0$. Sinon on a : $Q = 0$ et $Q' = 1$. Dans les 2 cas, $Q' = \overline{Q}$). Etats stables	

- La combinaison d'entrées (*S* = 1, *R* = 1) de la phase ⑤ est à proscrire car elle ne conduit pas à $Q' = \overline{Q}$ (les bascules ont leurs sorties complémentées Q et $\overline{Q}$).

- Les configurations pour lesquelles les 2 entrées changent d'état simultanément (comme à la phase ⑥) sont à proscrire car elles conduisent à un aléa de fonctionnement.

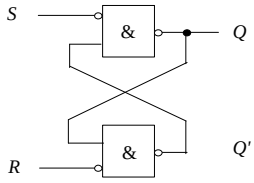
Table de vérité de la bascule RS :

(Q_n représente l'état stable de Q à l'instant discret n;
Q_{n-1} représente l'état stable de Q à l'instant précédant la configuration d'entrée courante : c'est donc l'état précédent de la sortie Q avant changement des entrées aux nouvelles valeurs que sont les valeurs courantes spécifiées.
Ce changement place Q à l'état Q_n)

Restriction de fonctionnement : 1 seule des 2 entrées doit changer d'état à la fois. Si les 2 entrées changent d'état en même temps (impossible cependant en asynchrone) → aléa pour Q.

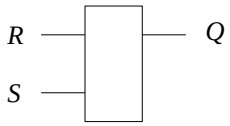
S	***R***	Q_n	***Fonction***	***Complémentarité***
0	0	Q_{n-1}	*Mémorisation*	$Q' = \overline{Q}$
0	1	0	*RESET (Mise à 0 de Q)*	$Q' = \overline{Q}$
1	0	1	*SET (Mise à 1 de Q)*	$Q' = \overline{Q}$
1	1		*Combinaison interdite car* $Q' \neq \overline{Q}$	$Q' \neq \overline{Q}$

Autre réalisation de la bascule RS avec des NAND :



La combinaison interdite engendre $Q = Q' = 1$, contrairement à la réalisation à portes NOR pour laquelle elle engendre $Q = Q' = 0$.

Symbole de la bascule RS



(Q' n'est pas systématiquement représenté)

La bascule *RS* est l'élément de base de la logique séquentielle. C'est la seule bascule asynchrone.

Fonctionnement asynchrone :

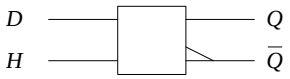
En asynchrone, la sortie de la bascule change d'état uniquement en fonction des grandeurs d'entrée.
Le système livré à lui-même, est ainsi plus rapide que les sytèmes synchrones, mais il présente des temps de propagation (≡ délais) difficiles à maîtriser → on préfère l'utilisation de systèmes synchrones.

Fonctionnement synchrone :

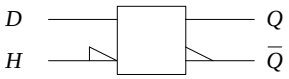
La prise en compte des entrées est conditionnée par une autorisation donnée par un signal d'horloge. Ainsi, les entrées du système sont prises en compte (provoquant alors l'état de sortie correspondant) uniquement s'il y a autorisation par l'horloge (l'horloge est alors dite active). Sinon (pas d'autorisation de la part de l'horloge), les entrées sont ignorées et leur changement d'état ne peut entraîner le basculement de la sortie : celle-ci demeure à son état antérieur (mémoire).

L'autorisation (≡ synchronisation) de l'horloge peut se faire de 3 façons : (exemple sur une bascule D)

- *Synchronisation sur niveau : (≡ latch)* il suffit d'appliquer le niveau logique convenable, dit niveau actif, sur l'entrée d'horloge, pour que la sortie de la bascule puisse réagir aux entrées de données : (*H* : signal d'horloge (noté aussi *CK*); *D* : entrée de donnée)

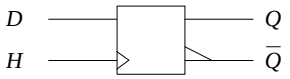


Synchro. sur niveau haut
(≡ positive latch)

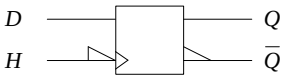


Synchro. sur niveau bas
(≡ negative latch)

- *Synchronisation sur front : ou flanc ou transition (≡ edge triggered)* la sortie de la bascule réagira aux entrées de données à l'instant où se produit un front (≡ une transition d'état) de l'horloge.
Ce front actif peut être montant ⌐↑ (≡ positif) ou descendant ↓¬ (≡ négatif) :

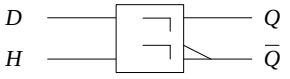


Synchro. sur front montant
(≡ positive edge triggered)

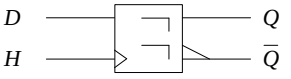


Synchro. sur front descendant
(≡ negative edge triggered)

- *Synchronisation par impulsion : (≡ pulse triggered)* une impulsion de synchronisation de l'horloge est composée de 2 fronts (l'un positif et l'autre négatif) ⌐↑↓¬ : le 1er front sert à la synchronisation des entrées, le 2nd front sert à la synchronisation des sorties.
Ce type de synchronisation est utilisé pour les systèmes maître-esclave - *sas*).



Maître-esclave sur niveau (haut)
(≡ positive pulse triggered)



Maître-esclave sur front (montant)
(≡ positive pulse edge triggered)

Un système constitué de plusieurs bascules synchrones est dit synchrone si toutes les bascules sont pilotées par une horloge et si cette horloge est identique pour toutes les bascules.

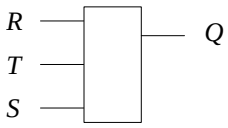
3. LES BASCULES SYNCHRONES

3.1. La bascule RST (≡ bascule RS synchronisée)

Du fait de la synchronisation, elle constitue une amélioration de la bascule *RS* asynchrone.
Bien que plus lents que les systèmes asynchrones (il faut attendre la validation d'horloge pour prendre en compte les données), les systèmes synchrones ont l'avantage d'introduire un certain déterminisme (prévision, régularité, maîtrise des séquences) dans les traitements, et sont ainsi beaucoup plus utilisés que les systèmes asynchrones.

Exemple: Bascule RST synchronisée sur niveau haut de l'horloge T (bascule RST latch > 0):

Symbole



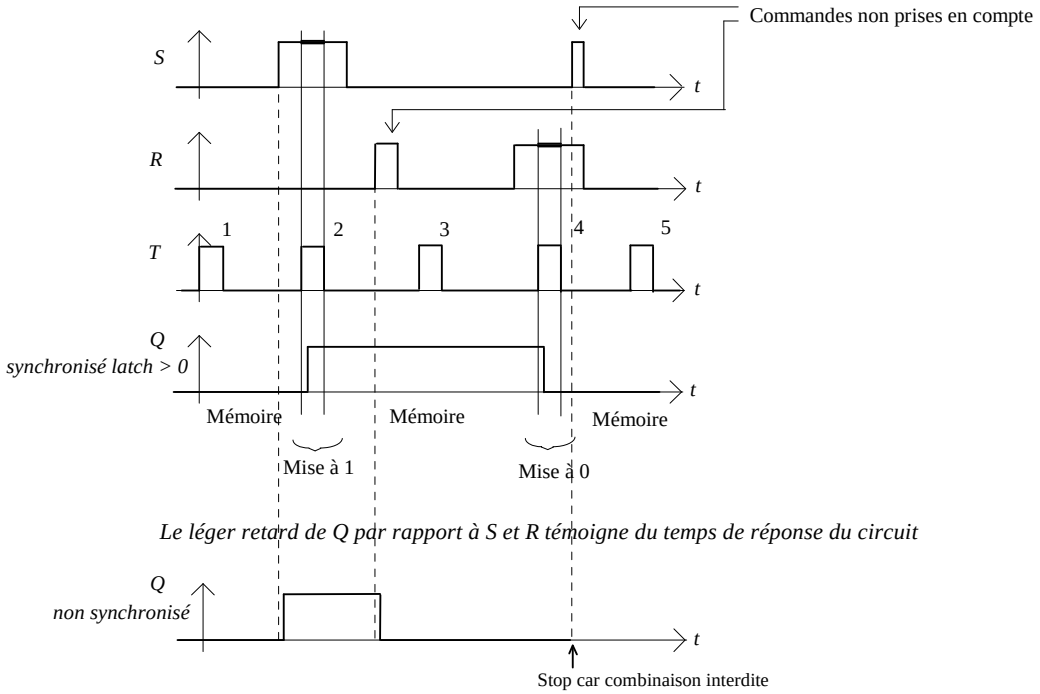
. ***T* = 0** : la sortie ne change pas quelles que soient les entrées *R* et *S*.
C'est le fonctionnement en mémoire. La bascule n'est pas synchronisée.

. ***T* = 1** : la bascule est alors synchronisée. Sa sortie respecte la table de fonctionnement de la bascule *RS* (asynchrone) avec les mêmes restrictions.

Table de fonctionnement de la bascule RST synchronisée sur niveau haut de l'horloge T :
(X signifie indifféremment 0 ou 1 (valeur quelconque binaire))

Horloge T	***T***	***S***	***R***	Q_n	***Fonction***
Horloge T inactive	0	X	X	Q_{n-1}	*Mémorisation*
Horloge T active	1	0	0	Q_{n-1}	*Mémorisation*
Horloge T active	1	0	1	0	*RESET (Remise à 0 de Q)*
Horloge T active	1	1	0	1	*SET (Mise à 1 de Q)*
Horloge T active	1	1	1	*Interdit*	

Exemple de fonctionnement (bascule RST latch > 0)

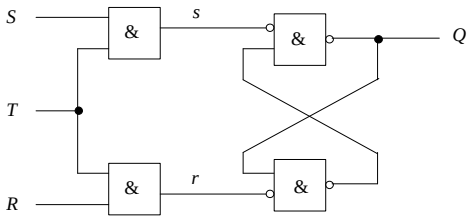


Constitution (bascule RST latch > 0) *(pour une RST latch < 0, il suffit de complémenter l'horloge)*

Etant donné que pour la combinaison $R = S = 0$ avec $T = 1$, on a un fonctionnement indentique à la combinaison $T = 0$ quels que soient R et S, il faut fabriquer deux variables r et s entrées d'une bascule RS asynchrone telles que les tables de vérité suivantes soient vérifiées :

T	*S*	*s*		*T*	*R*	*r*	
0	0	0		0	0	0	} *Fonction mémoire*
0	1	0		0	1	0	
1	0	0		1	0	0	
1	1	1	*Mise à 1*	1	1	1	*Mise à 0*

Ceci est facilement réalisé à l'aide d'une porte ET qui permet de bloquer les commandes R et S tant que $T = 0$. Le schéma de la bascule *RST* synchronisée sur niveau haut de T peut donc être le suivant :



Le circuit ET suivi de l'inverseur peut avantageusement être remplacé par un opérateur NAND.

3.2. La bascule D

Elle constitue un élément mémoire. Elle supprime la configuration interdite de la bascule *RST* . Elle possède 1 entrée de donnée D (*Data*) et est réalisée à partir d'une bascule *RST* avec les entrées R et S liées par la relation : $D = S = \overline{R}$.

Fonctionnement

Cette bascule dispose d'une seule entrée appelée D. Le signal de synchronisation peut être actif soit sur un niveau - la bascule est alors appelée *D latch* - soit sur un front (bascule *edge triggered*). C est l'horloge de synchronisation.

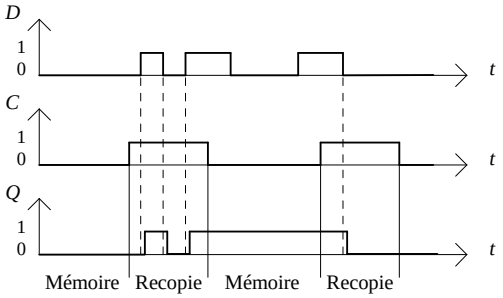
Exemple

Avec une seule entrée on ne peut trouver que deux modes de fonctionnement :

- le signal de synchronisation est actif, la sortie *Q* recopie l'entrée *D*.
- le signal de synchronisation n'est pas actif, la sortie *Q* ne change pas.

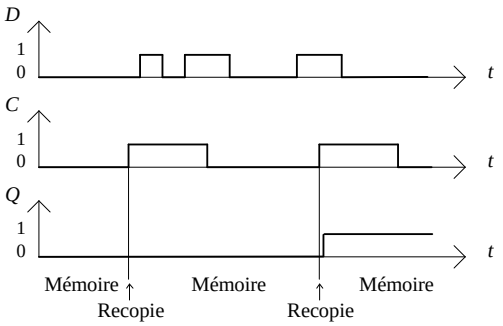
C'est le fonctionnement en mémoire. Lors du passage en position mémoire, la dernière valeur recopiée est mémorisée.

***Exemple avec une bascule D latch** > 0*



Le léger retard de Q par rapport à D témoigne du temps de propagation à travers la bascule

Exemple avec une bascule D edge triggered synchronisée sur front positif



Le léger retard de Q par rapport à la prise en compte de D témoigne du temps de propagation à travers la bascule

Constitution d'une bascule D latch > 0 *(pour une D latch < 0, il suffit de complémenter l'horloge)*

Une bascule *D* est issue d'une bascule *RST* avec les entrées *R* et *S* liées par la relation : $D = S = \overline{R}$

Pendant la phase où l'horloge de synchronisation est inactive, on a : $Q_n = Q_{n-1}$: fonction mémoire

L'équation de fonctionnement dans la phase d'activité de l'horloge est : $Q_n = S + \overline{R} \cdot Q = D_{n-1}$: fonction recopie

Le schéma de réalisation peut être le suivant :

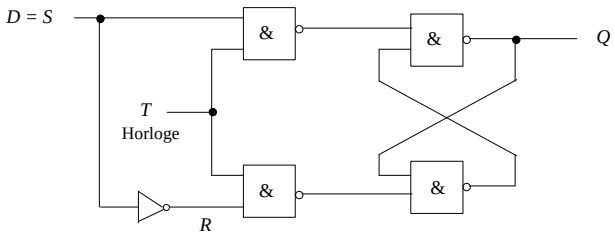


Table de vérité de la bascule D latch > 0

Horloge C	***C***	***D***	***Q_n***	***Fonction***
Horloge C inactive	0	X	Q_{n-1}	*Mémorisation*
Horloge C active	1	0	0	*Recopie*
Horloge C active	1	1	1	*Recopie*

ou encore :

Horloge C	***C***	***D***	***Q_n***	***Fonction***
Horloge C inactive	0	X	Q_{n-1}	*Mémorisation*
Horloge C active	1		*D*	*Recopie*

Constitution d'une bascule D >0 edge triggered (pour une D <0 edge triggered, il suffit de complémenter l'horloge)

La discrimination du front, c'est à dire du changement de niveau, ne s'effectue pas avec un circuit dérivateur mais par le jeu de trois mémoires internes à la bascule La réalisation simplifiée d'une bascule *D* >0 edge triggered est donné par :

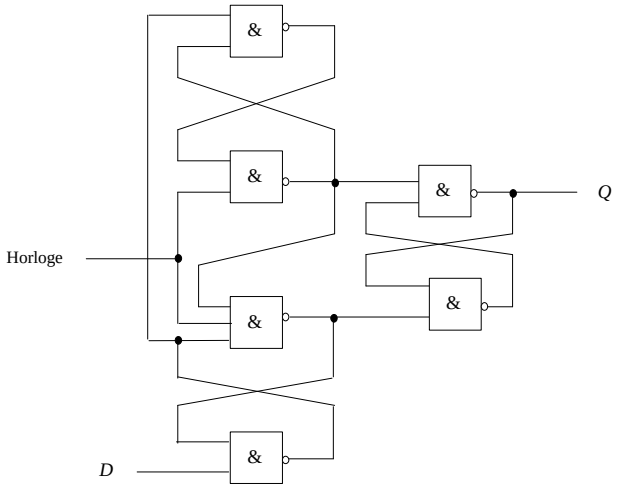


Table de vérité de la bascule D > 0 edge triggered

Horloge C	***C***	***D***	***Q_n***	***Fonction***
C inactive	X (0 ou 1 ou ↓)	X	Q_{n-1}	*Mémoire*
C active	↑	0	0	*Recopie*
C active	↑	1	1	*Recopie*

ou encore :

Horloge C	***C***	***D***	***Q_n***	***Fonction***
C inactive	X (0 ou 1 ou ↓)	X	Q_{n-1}	*Mémoire*
C active	↑		*D*	*Recopie*

L'analyse du fonctionnement de cette bascule peut être faite comme s'il s'agissait d'un système séquentiel asynchrone ayant deux variables d'entrées *D* et *C*.

La bascule *D* impose une restriction pour le bon fonctionnement :

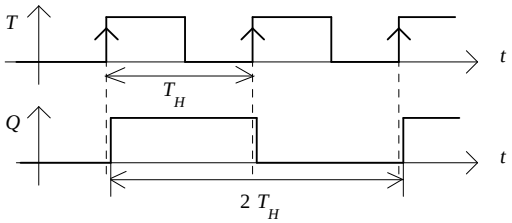
Exemple : pour une bascule *D* latch > 0, *D* ne doit pas changer d'état pendant que *C* = 1 (sinon le problème d'aléa asynchrone *RS* réapparaît, *R* et *S* changeant simultanément d'état) → la bascule *JK* va apporter une amélioration.

3.3. La bascule T

Fonctionnement

Une bascule fonctionnant suivant le type *T* dispose d'une seule commande : l'entrée d'horloge (*T*).
La sortie *Q* de la bascule change d'état à chaque impulsion de la commande.

Exemple d'un fonctionnement en type T synchronisé sur front montant :



Le léger retard de Q par rapport à la prise en compte de T témoigne du temps de propagation à travers la bascule

Remarque : Si le signal de commande est périodique de période T_H (fréquence *f*), le signal de sortie est également périodique mais de période $2T_H$ (fréquence $\frac{f}{2}$).

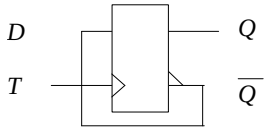
Ce mode de fonctionnement réalise une **division** par 2 **de la fréquence**.

Constitution

L'équation de fonctionnement est donnée par : $Q_n = \overline{Q}_{n-1}$

En effet, après chaque commande la sortie change d'état, ce qui signifie qu'elle prend la valeur du complément.
Il n'est pas commercialisé de bascules *T*. Il faut les fabriquer à l'aide des autres bascules.

Exemple : Transformation d'une bascule *D* (> 0 edge triggered) en bascule *T* (> 0 edge triggered).
L'équation de la bascule *D* étant : $Q_n = D$, il suffit de relier l'entrée *D* à la sortie $\overline{Q}$ pour obtenir :
$Q_n = \overline{Q} \left(\equiv \overline{Q}_{n-1}\right)$



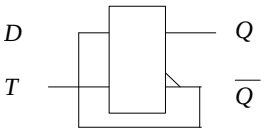
T > 0 edge triggered

Table de vérité de la bascule T latch > 0

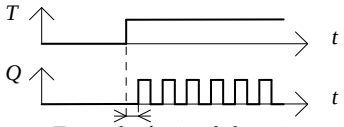
Horloge T	***T***	Q_n	***Fonction***
Horloge T inactive	0	Q_{n-1}	*Mémorisation*
Horloge T active	1	$\overline{Q}_{n-1}$	*Complémentation*

Précaution d'usage

Il ne faut pas réaliser un rebouclage qui provoque une instabilité de la sortie du montage. Par ex. , avec une bascule *D* latch > 0, la durée du niveau actif (1) de l'horloge doit être de courte durée: lorsque l'horloge passe au niveau actif (1), la sortie change après le temps nécessaire à la propagation de l'information. Comme la sortie $\overline{Q}$ est réunie à l'entrée *D*, le changement de la sortie provoque un autre changement de celle-ci après le même décalage temporel.



T latch > 0



Temps de réaction de la sortie

Rappel : Table de vérité de la bascule D latch > 0

Horloge T	***T***	***D***	Q_n	***Fonction***
C inactive	0	X	Q_{n-1}	*Mémoire*
C active	1		*D*	*Recopie*

3.4. La bascule JK

Fonctionnement

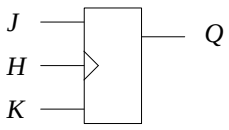
C'est une bascule disposant de deux entrées, respectivement appelées *J* et *K*. Comme pour la bascule *RS*, l'entrée *J* sert à la mise à 1 et l'entrée *K* à la remise à 0.

La différence entre la bascule *JK* et la bascule *RS* réside dans le fait qu'il n'y a plus d'état interdit pour les entrées, au profit de la combinaison *J* = *K* = 1 utilisée pour obtenir un fonctionnement type *T*.

Bascule *JK* = bascule *RS* avec : *J* = *S* , *K* = *R* et la combinaison *J* = *K* = 1 est non interdite ($Q' = \overline{Q}$) et de type bascule *T*.

Exemple: Bascule JK synchronisée sur front montant de l'horloge H (bascule JK > 0 edge triggered) :

Symbole



La table de fonctionnement est la suivante *(table de vérité)* :

Table de vérité (Analyse)

Horloge H	***H***	***J***	***K***	Q_n	***Fonction***
H inactive	X (0 ou 1 ou ↓)	X	X	Q_{n-1}	*Mémoire*
H active	↑	0	0	Q_{n-1}	*Mémoire*
H active	↑	0	1	0	*RESET* *(Remise à 0 de Q)*
H active	↑	1	0	1	*SET* *(Mise à 1 de Q)*
H active	↑	1	1	$\overline{Q}_{n-1}$	*Complémentation* *(fonctionnement type T)*

ou encore,

Table des transitions (Synthèse) - (Horloge active)

Transition $Q_{n-1} \rightarrow Q_n$	***J***	***K***
$0 \rightarrow 0$	0	X
$0 \rightarrow 1$	1	X
$1 \rightarrow 1$	X	0
$1 \rightarrow 0$	X	1

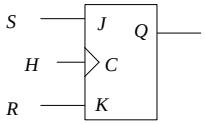
La table des transitions est utile en *synthèse* car :

- en *analyse* on regarde quel effet les entrées provoquent sur les sorties (raisonnement *déductif*),
- en *synthèse* on regarde quelles entrées il faut appliquer en fonction des sorties désirées (raisonnement *inductif*)

On remarque sur cette table de fonctionnement que la bascule *JK* peut se substituer à n'importe quelle autre bascule :

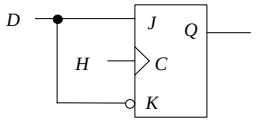
- *Bascule JK comme bascule RST :*

  Il est facile de fabriquer une bascule *RST* en faisant la correspondance $J = S$, $K = R$ et en s'interdisant $J = K = 1$:



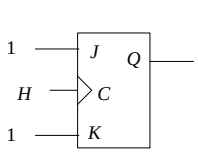
- *Bascule JK comme bascule D :*

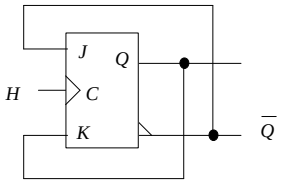
  Dans le cas où $J = \overline{K} = D$, on obtient une bascule *D* :



- *Bascule JK comme bascule T :*

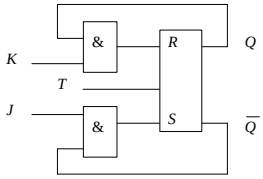
  Si $J = K = 1$ ou si $D = \overline{Q}$ (c'est à dire $J = \overline{Q}$ et $K = Q$), la bascule *JK* fonctionne suivant le type *T* :





Constitution des bascules JK

A priori une bascule *JK* est fabriquée à partir d'une bascule *RS* synchrone où il est fait un rebouclage tel que :
$S = J \cdot \overline{Q}$ et $R = K \cdot Q$:



L'équation de fonctionnement de la bascule *RS* devient : $Q_n = S + \overline{R} \cdot Q_{n-1} = J \cdot \overline{Q}_{n-1} + \overline{K} \cdot Q_{n-1}$

A cause du fonctionnement en type *T*, il y a le risque d'instabilité décrit lors de l'étude de cette bascule.

Afin d'éliminer toute instabilité il existe plusieurs solutions :

a) Limiter la durée du fonctionnement autonome du système en réduisant la période de sensibilité. Ceci conduit à rendre minimum la largeur de l'impulsion d'horloge. A la limite, cette solution consiste à synchroniser la bascule sur un front (solution utilisable également pour les bascules de type *RS* ou *D*).

b) Ouvrir la boucle du système. Cette solution impose l'utilisation d'une mémoire intermédiaire pour conserver le résultat précédent alors que la boucle est ouverte. C'est la structure maître-esclave (≡ SAS entre 2 bascules). (*Ex.* bascule *JK* maître-esclave : maître ≡ bascule *RS*, esclave ≡ bascule *D*). Cette structure peut aussi être utilisée pour les bascules de type *RS* ou *D*.

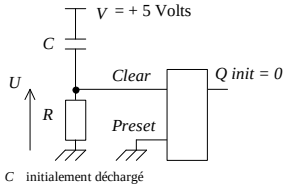
Initialisation d'une bascule (exemple d'une bascule JK)

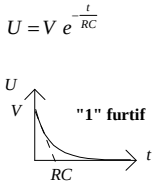
En plus des entrées de données, les bascules possèdent des entrées dites asynchrones permettant d'initialiser les sorties, ou même de fixer celles-ci à un état constant quelquesoit les entrées de données ou d'horloge.

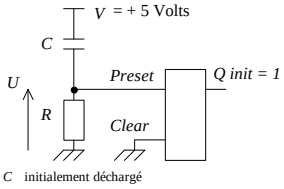
Les entrées asynchrones a_i de *mise à 0* et *mise à 1* souvent actives à l'état bas (donc notées $\overline{a_i}$) sont telles que lorsque l'ordre *mise à 0* par ex. est activé, *Q* est placé à l'état 0 quelles que soient les entrées d'horloge et de données *J*, *K*. Ce sont des commandes d'effacement et d'initialisation (appelées aussi *Clear* et *Preset* ou encore *Reset* et *Set*) qui peuvent être activées pour fixer l'état initial de la sortie et qui doivent ensuite être inactivées pour permettre le fonctionnement normal de la bascule.
Un simple circuit *RC* connecté à l'entrée *Preset* par ex. pour fixer l'état initial *Q* peut être utilisé :

Cas d'entrées asynchrones actives à l'état haut : (l'initialisation se fait en un temps de l'ordre de la Constant de temps RC)

A l'aide d'un circuit RC, on active l'entrée asynchrone un court instant au démarrage par un niveau haut, puis on ramène le signal asynchrone d'initialisation à 0. Appliqué à l'entrée *Clear*, ceci initialise *Q* à 0, mais appliqué à l'entrée *Preset*, ceci initialise *Q* à 1. A chaque fois l'entrée asynchrone non utilisée est placé à l'état inactif, soit l'état 0. (Le maintien prolongé d'une entrée asynchrone à son niveau actif fixe *Q* constant : à 0 si *Clear* est actif, à 1 si *Preset* l'est).

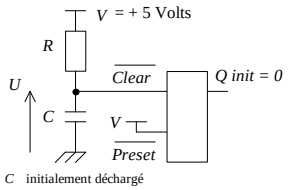


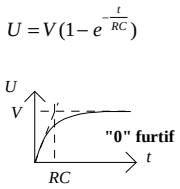


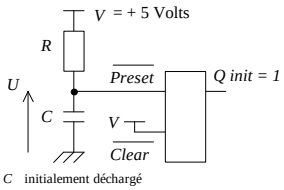


Cas d'entrées asynchrones actives à l'état bas :

A l'aide d'un circuit RC, on active l'entrée asynchrone un court instant au démarrage par un niveau bas, puis on ramène le signal asynchrone d'initialisation à 1. Appliqué à l'entrée $\overline{Clear}$, ceci initialise *Q* à 0, mais appliqué à l'entrée $\overline{Preset}$, ceci initialise *Q* à 1. A chaque fois l'entrée asynchrone non utilisée est placé à l'état inactif, soit l'état 1. (Le maintien prolongé d'une entrée asynchrone à son niveau actif fixe *Q* constant : à 0 si $\overline{Clear}$ est actif, à 1 si $\overline{Preset}$ l'est).







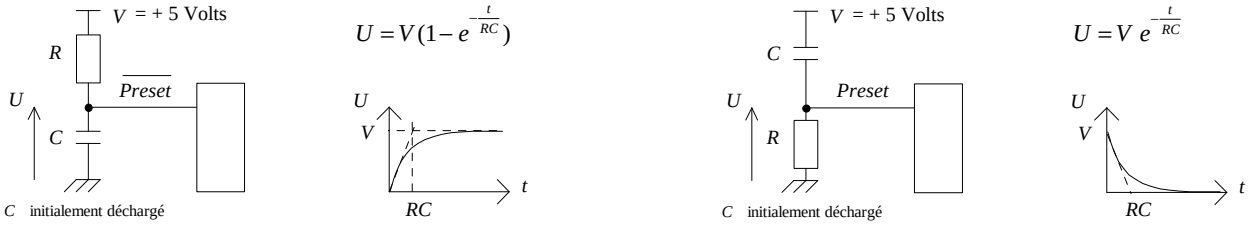
a) La bascule *JK* à déclenchement sur front

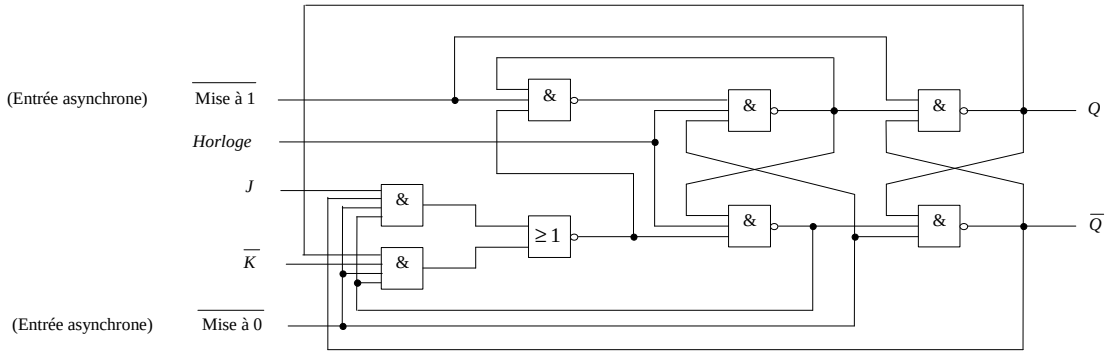
Il existe dans les circuits actuellement commercialisés deux façons de sélectionner un front :

- Le signal d'horloge est dérivé (*exemple :* 5470)
- Le signal d'horloge n'est pas dérivé. La discrimination du front s'effectue comme pour la bascule *D* edge triggered à l'aide de mémoires internes.

Par exemple le circuit type 54/74 109 qui est décrit à l'aide du schéma suivant constitue une bascule *JK* qui est synchronisée sur les fronts positifs. On peut étudier son fonctionnement en considérant cette bascule comme un système séquentiel asynchrone à trois variables d'entrée (*J, K, Horloge*).

Les entrées asynchrones a de *mise à 0* et *mise à 1* généralement actives à l'état bas (donc notées $\overline{a}$) sont telles que lorsque *mise à 0* par ex. est activée, *Q* est placé à l'état 0 quelles que soient les entrées d'horloge et de données *J, K*. Ce sont des commandes d'effacement et d'initialisation (appelées aussi *Clear* et *Preset*) qui peuvent être activées pour fixer l'état initial de la sortie et qui doivent ensuite être inactivées pour permettre le fonctionnement normal de la bascule. Un simple circuit *RC* connecté à l'entrée *Preset* par ex. pour fixer l'état initial *Q* = 1 peut être utilisé :

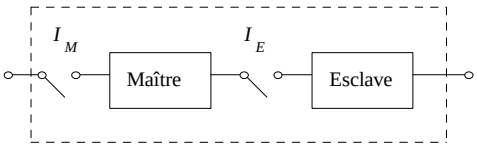




b) Structure maître-esclave (réalise une sorte de « sas »)

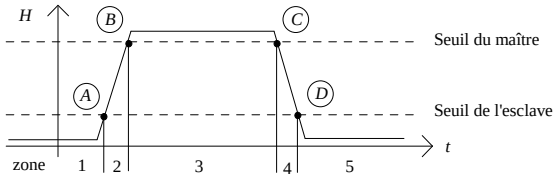
La structure maître-esclave est une structure à deux bascules synchrones. L'une, appelée le maître, est placée à l'entrée, l'autre, l'esclave, de type *D* est placée en sortie.

Pour simplifier l'étude du fonctionnement de cet ensemble, la synchronisation de chaque bascule est symbolisée par un interrupteur qui se ferme pendant la phase active de l'horloge.



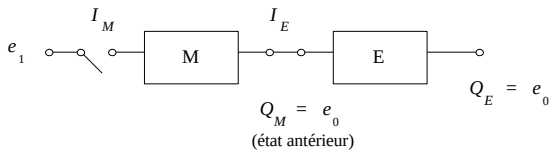
Les interrupteurs I_M et I_E sont commandés par le niveau du signal d'horloge par rapport à un seuil déterminé.

Le signal d'horloge étant une impulsion, les niveaux correspondant aux seuils définissent 4 points (5 zones distinctes) :



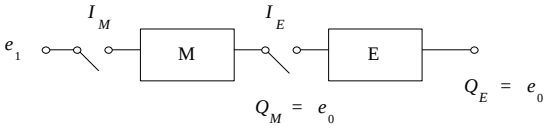
Dans la zone temporelle (1) les interrupteurs du maître et de l'esclave sont respectivement ouverts et fermés. Le maître fonctionne alors en mémoire, l'esclave recopie la sortie du maître.

$H = 0$
$I_M \equiv H$
$I_E \equiv \overline{H}$



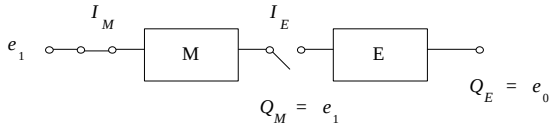
Pour accéder dans la zone (2), le seuil de l'esclave est franchi et l'interrupteur correspondant change d'état. L'esclave est alors séparé du maître et fonctionne en mémoire.

H entre états Bas et Haut
$I_M \equiv 0$
$I_E \equiv 0$



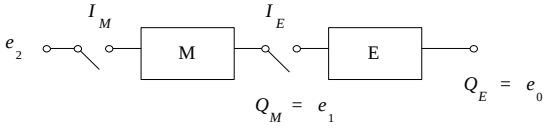
Dans la zone (3) l'esclave mémorise toujours la même information et le signal de sortie n'a pas encore changé. Par contre, le maître prend en compte l'information d'entrée.

$H = 1$
$I_M \equiv H$
$I_E \equiv \overline{H}$



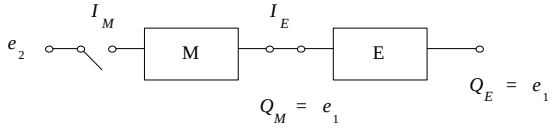
Dans la zone (4) le maître est de nouveau isolé de l'entrée. Il a mémorisé la nouvelle valeur de la sortie (e_1). L'esclave reste encore isolé du maître. par conséquent la valeur de la première sortie n'est pas encore modifiée.

H entre états Haut et Bas
$I_M \equiv 0$
$I_E \equiv 0$



C'est seulement dans la zone (5) que le maître communique la nouvelle valeur à l'esclave qui la transmet en sortie.

$H = 0$
$I_M \equiv H$
$I_E \equiv \overline{H}$

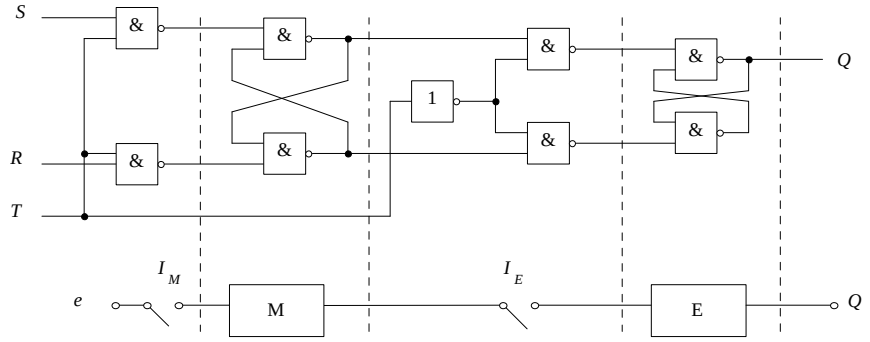


On peut remarquer qu'il n'existe pas de configuration où les deux interrupteurs sont simultanément fermés, ce qui permet d'effectuer un rebouclage sortie → entrée sans craindre une instabilité (système de *sas* réalisé).

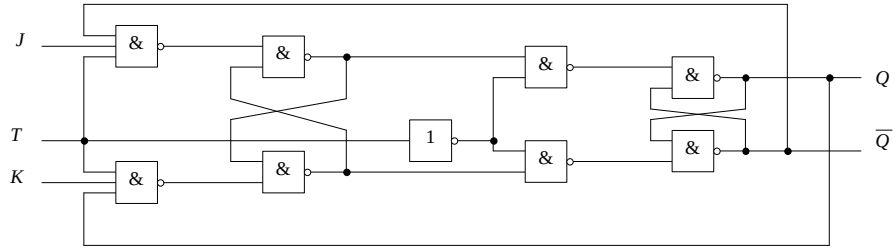
D'autre part le décalage temporel entre les commandes des interrupteurs I_M et I_E est inévitable puisqu'il est impossible d'obtenir des signaux d'horloge capables de passer *instantanément* du niveau 0 au niveau 1 (ou inversement). En d'autres termes, on peut dire que la qualité des fronts $\frac{dv}{dt}$ du signal d'horloge $v(t)$ n'influence pas le principe décrit. Néanmoins, il faut que l'impulsion de l'horloge ait une durée suffisante compte tenu de la vitesse d'évolution de la bascule.

Enfin, on remarque que si le signal d'horloge est supérieur au seuil de l'esclave, alors l'interrupteur I_E est ouvert. Et si le signal d'horloge est supérieur au seuil du maître, I_M est fermé. Les deux interrupteurs peuvent donc être commandés par des signaux complémentaires.
(H et $\overline{H}$; les phases (2) et (4) où les interrupteurs sont ouverts simultanément sont dûes à la transition *H* de $1 \rightarrow 0$ et *H* de $0 \rightarrow 1$).

Le schéma d'une bascule ***RST* maître-esclave** est donné par la figure suivante :



La bascule ***JK* maître-esclave** se déduit de la bascule précédente en réalisant le rebouclage $S = J\,\overline{Q}$ et $R = K\,Q$:



Suivant le mode de synchronisation du maître lors de la phase (3), il existe deux types de bascules maître-esclave.

En effet, l'acquisition de la nouvelle valeur peut être faite sur le front montant du signal de synchronisation ou sur son niveau.

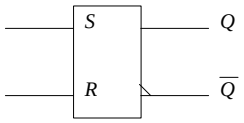
Pour distinguer ces deux types de bascules, on appelle bascule **maître-esclave à verrouillage** la structure dont le maître est synchronisé sur le front montant de l'horloge.

Enfin, il faut remarquer que la stabilité apportée à une bascule par une structure maître-esclave se fait au prix d'un coût en temps de propagation de la bascule.

Tableau récapitulatif

Chaque type de bascule est donnée avec table de fonctionnement et parfois un ex. de CI (circuit intégré) (Texas Instr.).

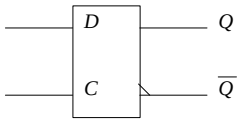
Bascule *RS*



S	R	Q_n	
0	0	Q_{n-1}	Mémoire
0	1	0	Recopie de S
1	0	1	
1	1	Interdit car $Q_n = \overline{Q_n}$	

La bascule RS est sujette à des aléas de fonctionnement (sorties imprévisibles) lorsque les 2 entrées S et R changent d'état simultanément.

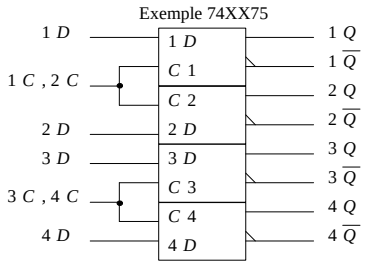
Bascule *D* Latch > 0



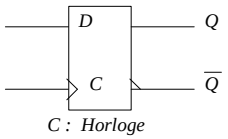
C : Horloge

C	D	Q_n	
0	X	Q_{n-1}	Mémoire
1	0	0	Recopie
1	1	1	

Exemple 74XX75



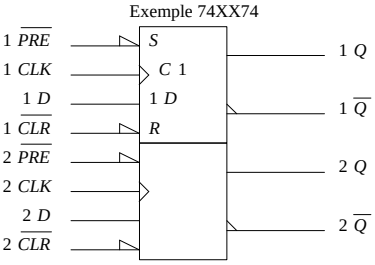
Bascule *D* > 0 edge triggered



C : Horloge

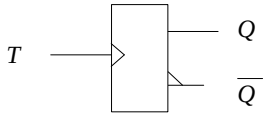
C	D	Q_n	
X	X	Q_{n-1}	Mémoire
↑	0	0	Recopie
↑	1	1	

Exemple 74XX74



Les entrées asynchrones *Preset* et *Clear*, actives à l'état bas, de mise à 1 et à 0 servent à l'initialisation ou l'effacement

Bascule *T*

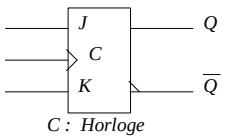


T > 0 edge triggered

Table de vérité

Horloge T	***T***	Q_n	***Fonction***
Horloge T inactive	0	Q_{n-1}	*Mémorisation*
Horloge T active	↑	$\overline{Q}_{n-1}$	*Complémentation*

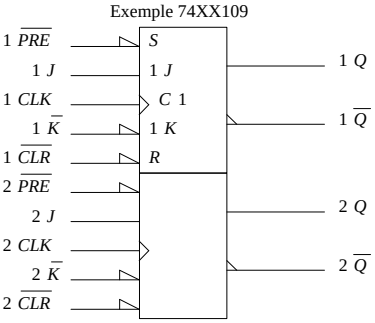
Bascule *JK* > 0 edge triggered



C : Horloge

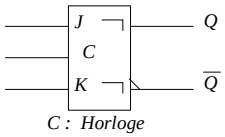
C	J	K	Q_n	
X	X	X	Q_{n-1}	Mémoire
↑	0	0	Q_{n-1}	
↑	0	1	0	Recopie de J
↑	1	0	1	
↑	1	1	$\overline{Q}_{n-1}$	Complément

Exemple 74XX109

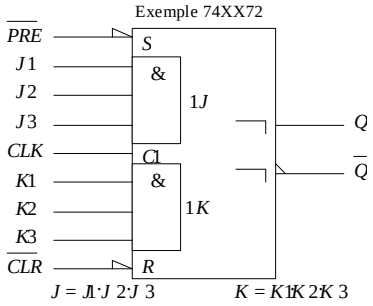


Les entrées asynchrones *Preset* et *Clear*, actives à l'état bas, de mise à 1 et à 0 servent à l'initialisation ou l'effacement

Bascule *JK* maître-esclave

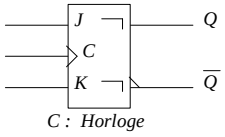


C	J	K	Q_n	
X	X	X	Q_{n-1}	Mémoire
↓	0	0	Q_{n-1}	
↓	0	1	0	Recopie de J
↓	1	0	1	
↓	1	1	$\overline{Q}_{n-1}$	Complément

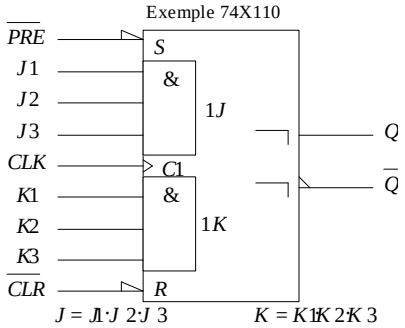


J et *K* sont calculés à l'aide d'une porte ET à 3 entrées.
Le symbole ⌐ indique que la sortie n'évolue qu'après le retour à l'état initial de l'horloge *C*.

Bascule *JK* avec verrouillage de la donnée

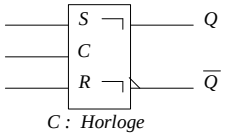


C	J	K	Q_n	
X	X	X	Q_{n-1}	Mémoire
↑↓	0	0	Q_{n-1}	
↑↓	0	1	0	Recopie de J
↑↓	1	0	1	
↑↓	1	1	$\overline{Q}_{n-1}$	Complément



Par rapport à la bascule *JK* maître-esclave, l'entrée de contrôle *C* est munie du triangle, symbole d'une activité sur un front positif, ce qui signifie que les entrées *J* et *K* sont échantillonnés sur le front montant de *C*. Le résultat n'est transmis en sortie qu'après le retour à l'état initial de l'information d'horloge *C*.

Bascule *RS* maître-esclave



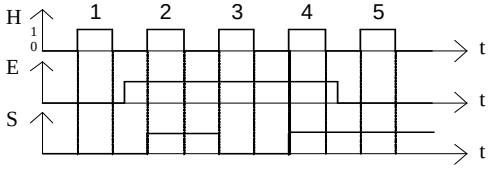
C	S	R	Q_n	
X	X	X	Q_{n-1}	Mémoire
↓	0	0	Q_{n-1}	
↓	0	1	0	Recopie de S
↓	1	0	1	
↓	1	1	Interdit	

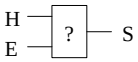
TD 3. LOGIQUE SEQUENTIELLE 1

1. Logique combinatoire et séquentielle

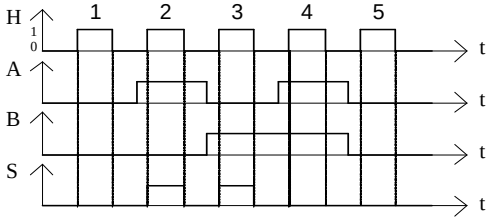
Les systèmes *a*, *b*, *c*, dont le fonctionnement est décrit par les chronogrammes suivants sont-ils combinatoires ou séquentiels ? (S : sortie; H, E, A, B : entrées)

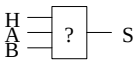
a.



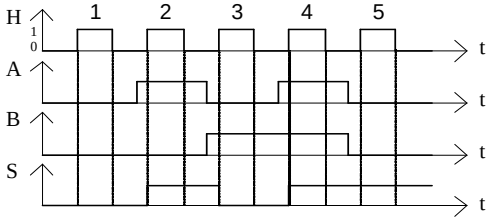


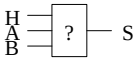
b.





c.

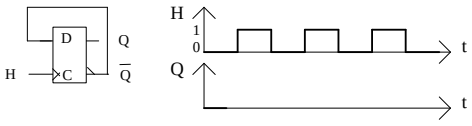




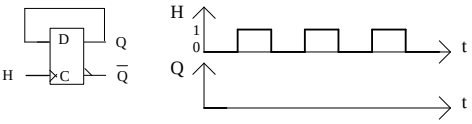
2. Bascules

Compléter les chronogrammes pour chacun des schémas suivants :

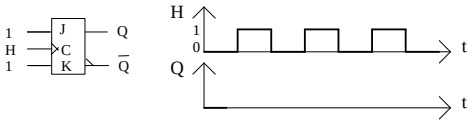
a.



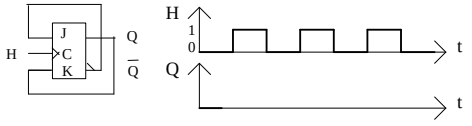
b.



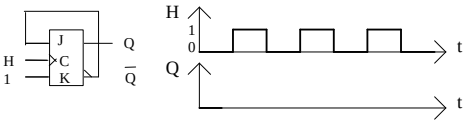
c.



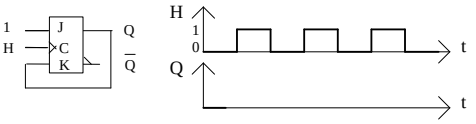
d.



e.

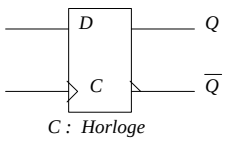


f.



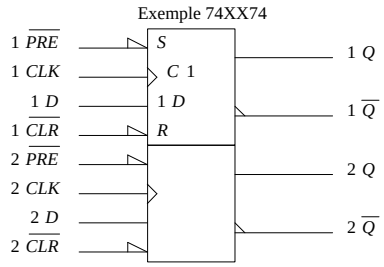
Rappel

Bascule *D* > 0 edge triggered

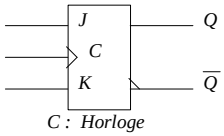


C	*D*	Q_n	
X	X	Q_{n-1}	Mémoire
↑	0	0	Recopie
↑	1	1	Recopie

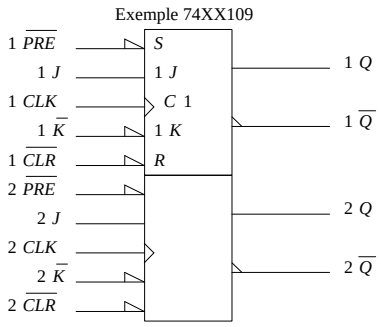
Exemple 74XX74



Bascule *JK* > 0 edge triggered

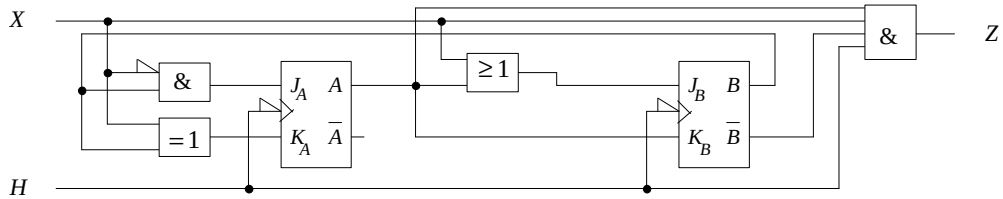


C	J	K	Q_n	
X	X	X	Q_{n-1}	} Mémoire
↑	0	0	Q_{n-1}	
↑	0	1	0	} Recopie de J
↑	1	0	1	
↑	1	1	$\overline{Q}_{n-1}$	Complément

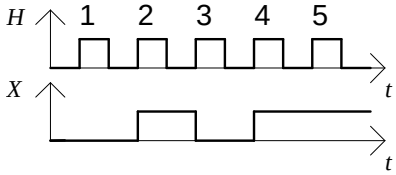


3. Système séquentiel

Soit le système séquentiel :

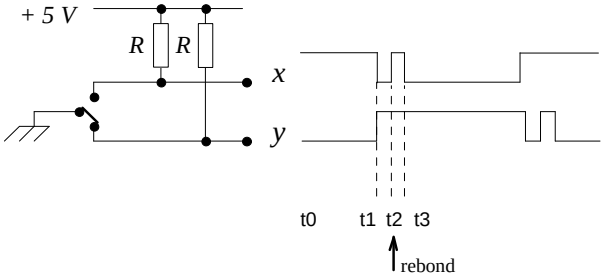


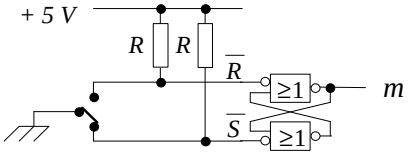
Compléter le chronogramme suivant (*A* et *B* sont initialement à l'état 0) et donner la séquence (automate) des états *AB* du compteur:



4. Bascule RS à entrées complémentées

On considère le système suivant, engendrant les signaux *x* et *y*. Déterminer l'allure du signal *m* issu de la bascule *RS*:





Rappel : Table de vérité de la bascule RS :

S	***R***	Q_n	***Fonction***	***Complémentarité***
0	0	Q_{n-1}	*Mémorisation*	$Q' = \overline{Q}$
0	1	0	*RESET (Mise à 0 de Q)*	$Q' = \overline{Q}$
1	0	1	*SET (Mise à 1 de Q)*	$Q' = \overline{Q}$
1	1		*Combinaison interdite car* $Q' \neq \overline{Q}$	$Q' \neq \overline{Q}$

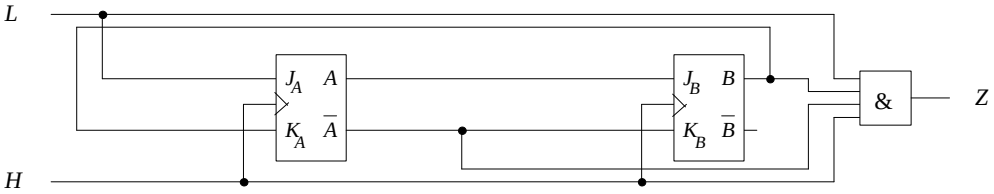
TD 3 ANNEXE. LOGIQUE SEQUENTIELLE 1

1. Détection synchrone d'une séquence (serrure électronique)

Sur une ligne électrique de transmission L arrivent des données binaires en série. Chaque bit est synchronisé, c'est-à-dire pris en compte, au front montant d'un signal d'horloge H. La séquence binaire à détecter (clé) compte 4 bits notés $a\ b\ c\ d$. La détection de la séquence a pour effet de placer quasi instantanément (c'est-à-dire au temps de retard d'une porte logique élémentaire près) après la détection de la présence du 4ème bit de la séquence au front montant d'horloge, au niveau logique haut l'état d'une ligne Z initialement à l'état bas (une impulsion de sortie $Z = 1$, de largeur sensiblement égale à une demi-période d'horloge, est produite).

1. *Solution à logique câblée (1)*

La synthèse d'un système séquentiel à base de bascules JK autorise la détection de la séquence et conduit à l'architecture suivante :

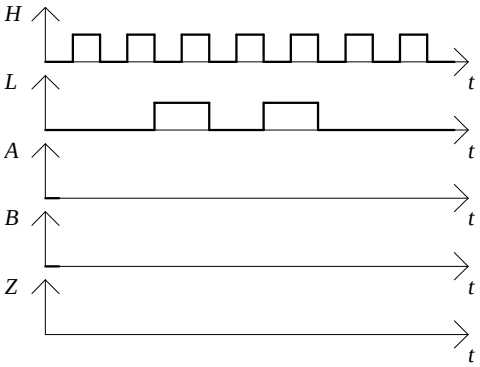


1.1. Compléter le chronogramme fourni.

1.2. En déduire la séquence $a\ b\ c\ d$ à détecter.

Solution à logique câblée (1)

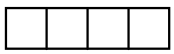
1.1. Les bascules ont un état initial bas : $A = 0$, $B = 0$



1.2. Séquence à détecter :

Séquence à détecter $a\ b\ c\ d$

? ? ? ?



2. *Solution à logique câblée (2)*

Une synthèse plus intuitive utilisant un registre à décalage est également possible. On donne la table de fonctionnement du circuit 74164 (registre à décalage 8 bits) :

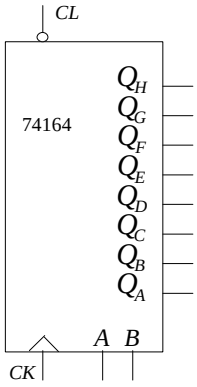
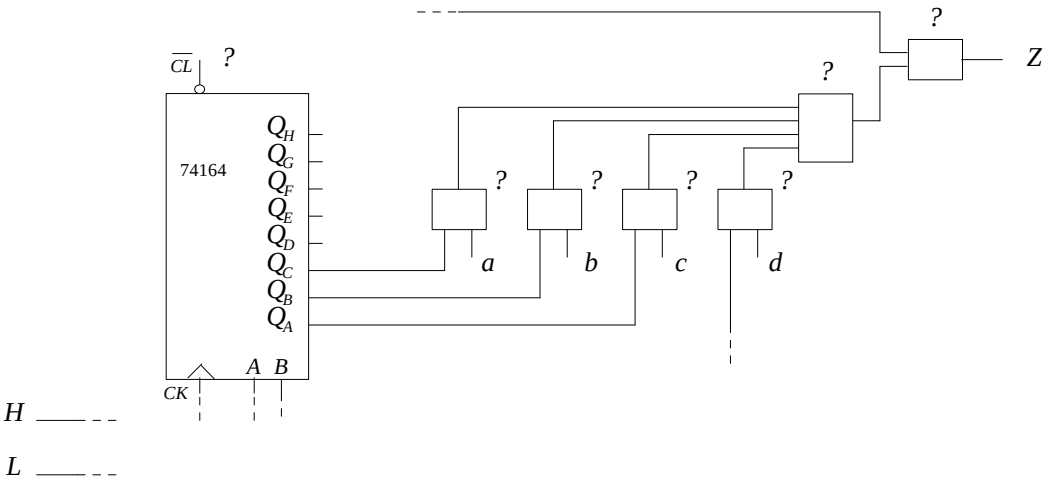


Table de fonctionnement 74164		*Entrées*		*Sorties*							
$\overline{CL}$ ($\overline{CLEAR}$)	*CK (CLOCK)*	*A*	*B*	Q_{A_n}	Q_{B_n}	Q_{C_n}	Q_{D_n}	Q_{E_n}	Q_{F_n}	Q_{G_n}	Q_{H_n}
0	X	X	X	0	0	0	0	0	0	0	0
1	X sauf ↑	X	X	$Q_{A_{n-1}}$	$Q_{B_{n-1}}$	$Q_{C_{n-1}}$	$Q_{D_{n-1}}$	$Q_{E_{n-1}}$	$Q_{F_{n-1}}$	$Q_{G_{n-1}}$	$Q_{H_{n-1}}$
1	↑ (front montant)	1	1	1	$Q_{A_{n-1}}$	$Q_{B_{n-1}}$	$Q_{C_{n-1}}$	$Q_{D_{n-1}}$	$Q_{E_{n-1}}$	$Q_{F_{n-1}}$	$Q_{G_{n-1}}$
1	↑	0	X	0	$Q_{A_{n-1}}$	$Q_{B_{n-1}}$	$Q_{C_{n-1}}$	$Q_{D_{n-1}}$	$Q_{E_{n-1}}$	$Q_{F_{n-1}}$	$Q_{G_{n-1}}$
1	↑	X	0	0	$Q_{A_{n-1}}$	$Q_{B_{n-1}}$	$Q_{C_{n-1}}$	$Q_{D_{n-1}}$	$Q_{E_{n-1}}$	$Q_{F_{n-1}}$	$Q_{G_{n-1}}$

2.1. Compléter le schéma électrique (en complétant le câblage et en remplaçant chaque ? par un bit 0 ou 1, ou par un opérateur logique) pour permettre la détection de la séquence $a\ b\ c\ d$.

Solution à logique câblée (2)



2.2. Donner deux avantages fonctionnels de cette solution par rapport à la précédente.

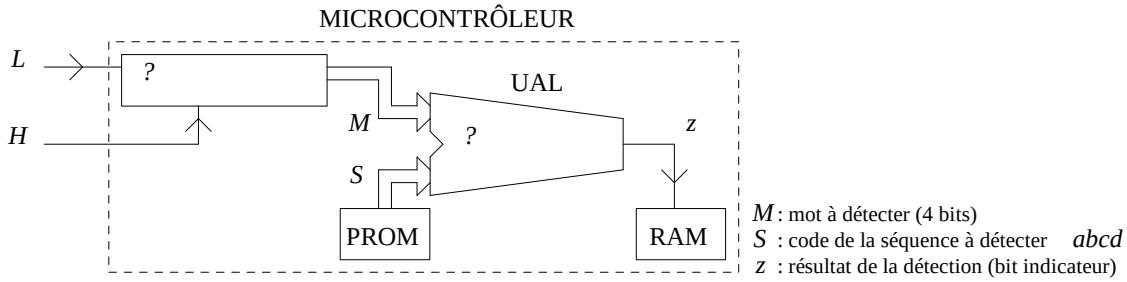
Avantage 1	*Avantage 2*
•	•

3. *Solution à logique programmée (1)*

On utilise un microcontrôleur pour la détection de la séquence $a\ b\ c\ d$.

3.1. Compléter le schéma synoptique en indiquant :

- le circuit interne au microcontrôleur réalisant l'acquisition des données de la ligne L
- le type d'instruction (symbolisée dans l'Unité Arithmétique et Logique) au coeur du programme réalisant la détection de la séquence.



3.2. Quel avantage fonctionnel peut-on attribuer à cette solution par rapport à la précédente ?
Dire aussi quel peut être l'inconvénient principal de cette solution par rapport aux précédentes.

Avantage
•

4. *Solution à logique programmée (2)*

Solution VHDL.

TP 3. LOGIQUE SEQUENTIELLE 1

1. Matériel nécessaire

- Oscilloscope
- Générateur de signaux Basses Fréquences (GBF)
- Alimentation stabilisée (2x[0-30 V] ⎓ + 1x[5 V] ⎓)
- Multimètre
- Moniteur MS05 (plaquette de câblage)
- Câbles : - 1 T, 1 BNC-BNC, 1 BNC-Banane, 1sonde oscilloscope, 6 fils Banane, petits fils.

Composants

- 1 Résistance 1MΩ
- 1 Condensateur 100 nF
- 1 mini-interrupteur (horloge manuelle)

Circuits logiques de la famille CMOS série 4000 :
- 2 *4027* : 2 Bascules JK positive edge triggered

2. Notation du TP

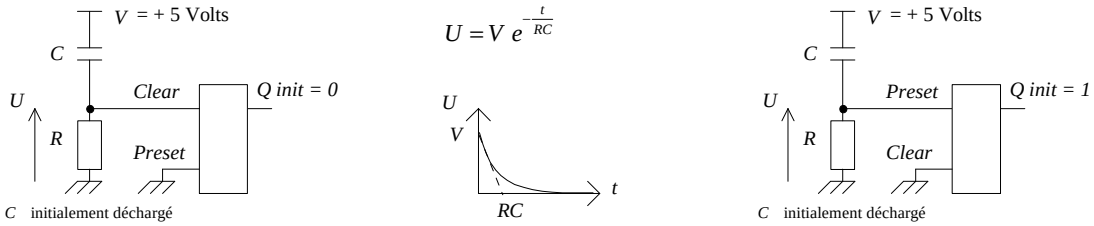
Faire examiner par le professeur en fin de séance, les différentes parties du TP.

Rappel : Initialisation d'une bascule (exemple d'une bascule JK)

Les entrées asynchrones *a* de *mise à 0* et *mise à 1* généralement actives à l'état bas (donc notées $\overline{a}$) sont telles que lorsque *mise à 0* par ex. est activée, *Q* est placé à l'état 0 quelles que soient les entrées d'horloge et de données *J, K*. Ce sont des commandes d'effacement et d'initialisation (appelées aussi *Clear* et *Preset* ou encore *Reset* et *Set*) qui peuvent être activées pour fixer l'état initial de la sortie et qui doivent ensuite être inactivées pour permettre le fonctionnement normal de la bascule.
Un simple circuit *RC* connecté à l'entrée *Preset* par ex. pour fixer l'état initial *Q* peut être utilisé :

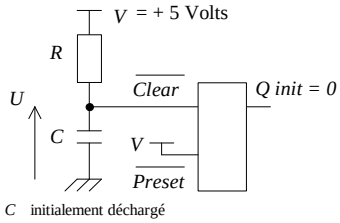
Cas d'entrées asynchrones actives à l'état haut :

A l'aide d'un circuit RC, on active l'entrée asynchrone un court instant au démarrage par un niveau haut, puis on ramène le signal asynchrone d'initialisation à 0. Appliqué à l'entrée *Clear*, ceci initialise *Q* à 0, mais appliqué à l'entrée *Preset*, ceci initialise *Q* à 1. A chaque fois l'entrée asynchrone non utilisée est placé à l'état inactif, soit l'état 0. (Le maintien prolongé d'une entrée asynchrone à son niveau actif fixe *Q* constant : à 0 si *Clear* est actif, à 1 si *Preset* l'est).

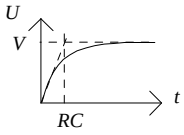


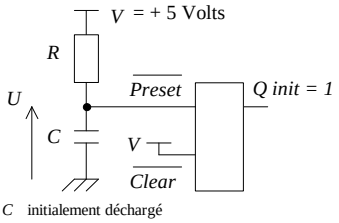
Cas d'entrées asynchrones actives à l'état bas :

A l'aide d'un circuit RC, on active l'entrée asynchrone un court instant au démarrage par un niveau bas, puis on ramène le signal asynchrone d'initialisation à 1. Appliqué à l'entrée $\overline{Clear}$, ceci initialise *Q* à 0, mais appliqué à l'entrée $\overline{Preset}$, ceci initialise *Q* à 1. A chaque fois l'entrée asynchrone non utilisée est placé à l'état inactif, soit l'état 1. (Le maintien prolongé d'une entrée asynchrone à son niveau actif fixe *Q* constant : à 0 si $\overline{Clear}$ est actif, à 1 si $\overline{Preset}$ l'est).



$$U = V(1 - e^{-\frac{t}{RC}})$$





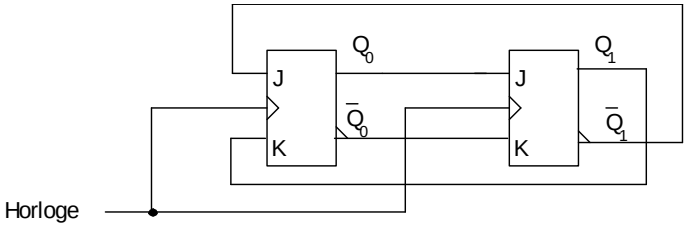
Etude expérimentale

Simulation (& Câblage) :

Pour des raisons de compatibilité, n'utiliser que des circuits de la même famille (famille CMOS 4000 à ne pas mélanger avec la famille TTL 74xxx).

3. Bascules JK (1)

Le schéma du circuit considéré est le suivant :

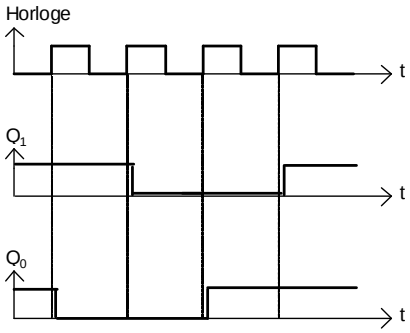


Bascules JK 4027

Etude théorique

- Par une étude théorique, prévoir à partir de l'état initial $Q_1Q_0 = 00$, la séquence (automate) des états Q_1Q_0 du compteur 4 états (Tracer le chronogramme faisant figurer les signaux H (Horloge), Q_1 et Q_0).

Etude théorique - Corrigé (Avec un état initial $Q_1Q_0 = 11$)



$Q_1\ Q_0 = 1\ 1$
$Q_1\ Q_0 = 1\ 0$
$Q_1\ Q_0 = 0\ 0$
$Q_1\ Q_0 = 0\ 1$
$Q_1\ Q_0 = 1\ 1$

etc

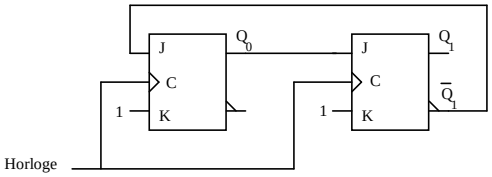
Etude expérimentale

- Vérifier expérimentalement par câblage et simulation ce résultat en visualisant avec des LEDs et avec le chronogramme les états Q_1 et Q_0 du compteur et en utilisant le Générateur Basse Fréquence (GBF) pour horloge en câblage *(Pulser de Circuit Maker pour la simulation).*

4. Bascules JK (2)

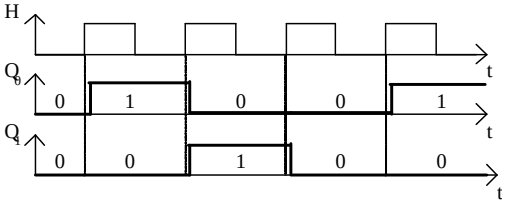
Etude théorique

En traçant le chronogramme faisant figurer les signaux H (Horloge), Q_1 et Q_0 , prévoir la séquence (automate) des états Q_1Q_0 du compteur 3 états: (*Etat initial supposé :* $Q_0 = Q_1 = 0$)



Bascules JK 4027

Etude théorique - Corrigé



On a un compteur par 3 : $Q_1Q_0 = 00 \rightarrow Q_1Q_0 = 01 \rightarrow Q_1Q_0 = 10 \rightarrow Q_1Q_0 = 00 \rightarrow$ ...

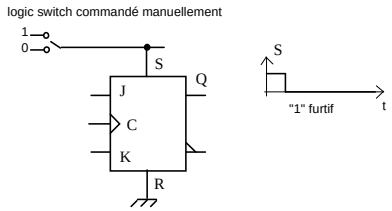
Etude expérimentale

- Réaliser le câblage et cadencer à l'aide du mini-interrupteur pour horloge.
- Vérifier par la simulation (*Circuit Maker*) avec un *Pulser* pour horloge.

(Avec le logiciel de simulation assez permissif, les entrées asynchrones d'initialisation R et S des bascules JK laissées « en l'air » sont équivalentes à R=0 et S=0 ce qui initialise la sortie de la bascule Q à 0. - En câblage, il faudrait câbler effectivement R=0, S=0)

- Compléter le schéma pour un état initial garanti $Q_1Q_0 = 00$.
- Le compteur est-il autocorrecteur ? (Initialiser les sorties à un état hors cycle normal de comptage, par exemple $Q_1Q_0 = 11$ et observer si le compteur revient à son cycle normal de comptage : si oui, le compteur est autocorrecteur). Vérifier la théorie par la simulation en traçant le chronogramme.

Note : 1. En simulation , l'initialisation de la sortie Q = 1 d'une bascule JK s'obtient en jouant sur les entrées asynchrones d'initialisation R et S de façon à envoyer un « 1 » furtif sur l'entrée S, R étant à 0 :



2. Rappel - Pour tracer un chronogramme à l'aide du simulateur, utiliser le composant :

Instruments → Digital → Scope et l'icône : 

- L'horloge est constituée du composant : *Instruments → Digital → Pulser* (sortie sur Q1)
- L'interrupteur, commutant entre +5V et 0, est constitué du composant : *Digital → Power → Logic switch*

Rangement du poste de travail

Examen des différentes parties du TP et rangement (0 pour tout le TP sinon).

4. LOGIQUE SEQUENTIELLE 2 - APPLICATIONS

1. LES REGISTRES

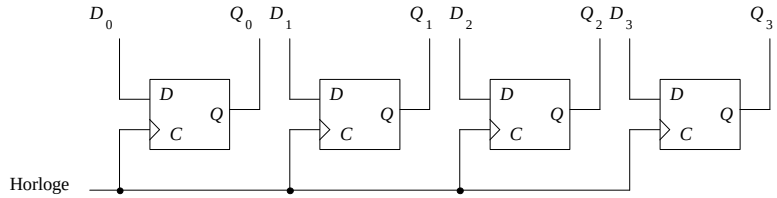
1.1. Présentation

Un registre est d'abord un ensemble de cases ou cellules mémoire capables de stocker une **information** (≡ un mot binaire). La position des cases mémoire entre elles est responsable de l'ordre des chiffres, c'est à dire de la structure de l'information. Dans le système binaire, une case mémoire est définie à l'aide d'une bascule. Un registre est donc un ensemble ordonné de bascules.
De plus, l'interconnexion entre les bascules permet certaines manipulations de l'information stockée.

1.2. Fonctionnement

Un registre sert à mémoriser un mot ou un nombre binaire. Le schéma d'un tel système comporte autant de bascule type *D* que d'éléments binaires à mémoriser. *Toutes les bascules sont commandées par le même signal d'horloge*.

Exemple : registre 4 bits *Fonctions Chargement en mémoire et Mémorisation*

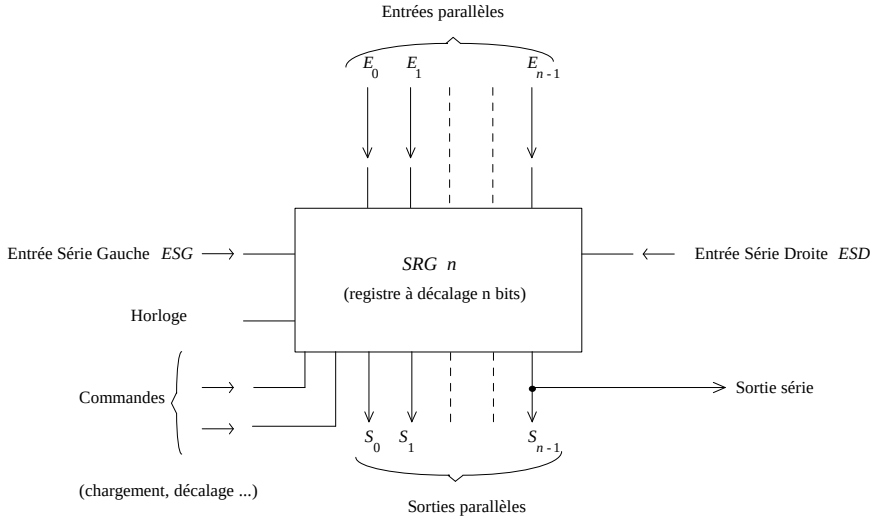


Rappel : Bascule *D* = $\begin{cases} Q_n = D & \text{si horloge active} \leftarrow \text{Chargement} \\ Q_n = Q_{n-1} & \text{sinon} \leftarrow \text{Mémoire} \end{cases}$

Moyennant une interconnexion entre les cellules, le registre précédent devient capable d'opérer une translation des chiffres (≡ bits) du nombre (≡ mot) initialement stocké. Le déplacement s'effectue soit vers la droite soit vers la gauche. Le registre est alors appelé **registre à décalage**.
De nombreuses applications résultent de cette possibilité de décalage, par exemple :

- la conversion série-parallèle d'une information numérique ;
- les opérations de multiplication et division par 2 ;
- la ligne à retard numérique.

Plus généralement un registre peut se représenter par le schéma suivant : *Fonctions Décalage en plus*

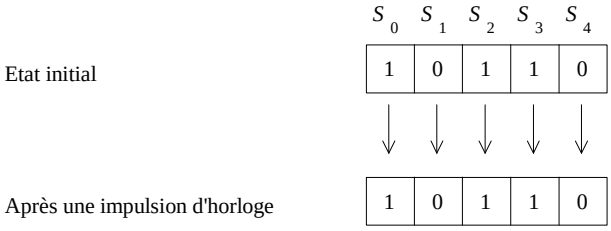


Les registres disposant de toutes ces entrées, sorties et commandes sont appelés **registres universels** à *n* cellules. Tous les registres actuellement commercialisés n'ont pas toutes les possibilités de commande du registre universel essentiellement à cause de la limitation du nombre de broches disponibles par boîtier. Les sorties $S_0 \cdots S_{n-1}$ sont les sorties des bascules constituant les cellules du registre.

Les signaux de commande du registre permettent de :

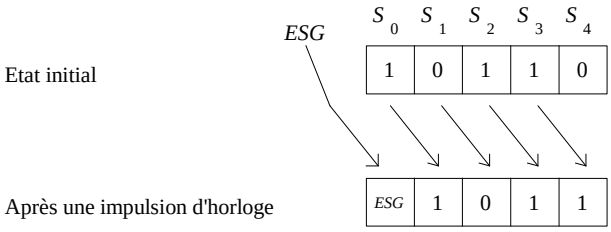
- Garder une information en mémoire. Chaque bascule conserve sa valeur malgrè les impulsions d'horloge.

Exemple :



- Décaler une information de la gauche vers la droite. Le contenu de la bascule de rang *i* est transmis à celle de rang *i* +1 à chaque impulsion d'horloge.

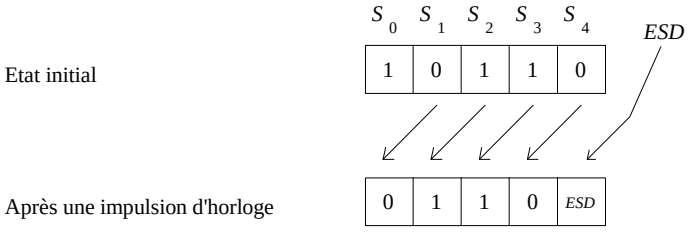
Exemple :



La sortie S_0 de la première bascule prend alors la valeur de l'entrée du registre appelé **entrée série** gauche (*ESG*).

- Décaler une information de la droite vers la gauche. Le fonctionnement est semblable à celui décrit ci-dessus en inversant le sens d'évolution des éléments binaires de chaque sortie. La bascule de rang *i* prend la valeur de la sortie *i* +1 à chaque impulsion d'horloge.

Exemple :



Cette fois c'est la valeur de l'entrée série droite (*ESD*) qui est inscrite dans la dernière bascule S_4.

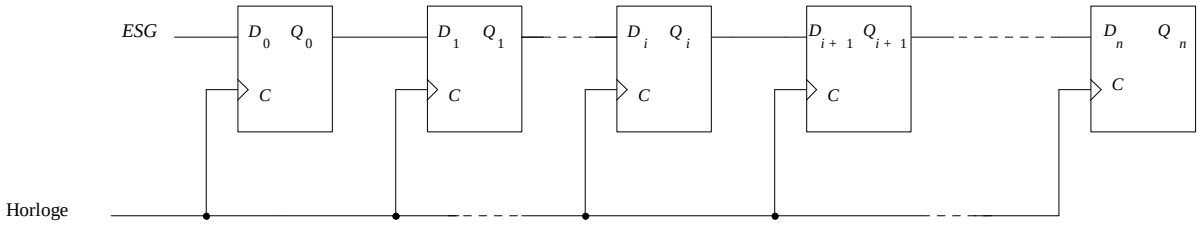
1.3. Constitution d'un registre

Pour assurer correctement la fonction de décalage, un registre doit comporter des cellules constituées de bascules de type maître-esclave ou à déclenchement par front (sans quoi le décalage n'est pas contrôlé).

Le décalage n'est pas la seule fonction que doit pouvoir accomplir un registre.

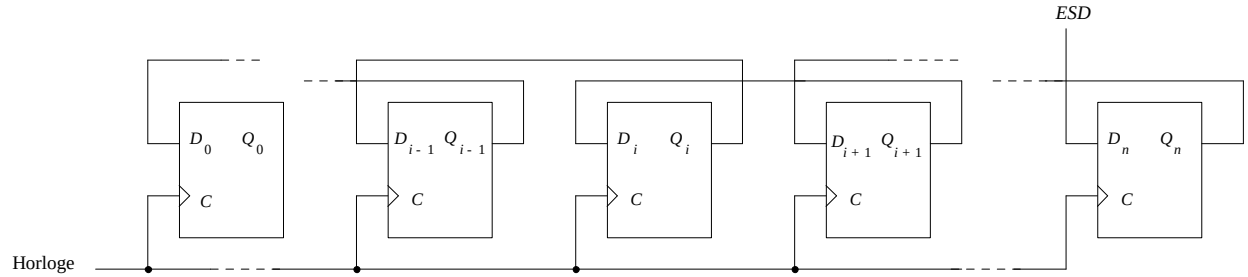
1.3.1. La fonction décalage à droite

La bascule *D* de rang *i* doit recopier la sortie de la bascule de rang *i* - 1.
Son entrée *D* doit donc être connectée à la sortie *i* - 1. Le schéma de l'interconnexion entre les bascules est donné ci-après :



1.3.2. La fonction décalage à gauche

Si l'on suppose que la position de chaque bascule est fixe, c'est la câblage qui doit réaliser le décalage vers la gauche. Par conséquent, l'entrée *D* de la bascule de rang *i* est reliée à la sortie de rang *i* + 1 :

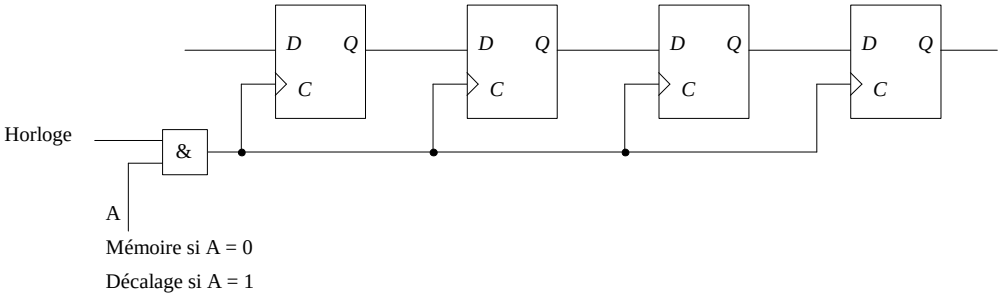


1.3.3. La fonction mémoire

Suivant la nature de la bascule utilisée dans chaque cellule, la fonction mémoire peut se réaliser de différentes façons. La table suivante donne la combinaison des entrées des bascules qui réalise la fonction mémoire :

Type	Entrées
RS	$R = S = 0$
JK	$J = K = 0$ ou $J = \overline{K} = Q$
D	$D = Q$

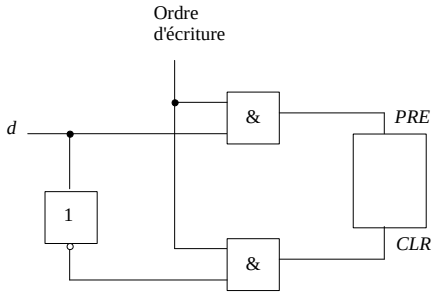
Une autre solution consiste à interdire l'action de l'horloge en intercalant une porte ET en série. Cette dernière solution est à proscrire car elle crée un décalage entre les différents signaux d'horloge d'un même système à cause du temps de propagation à travers la porte ET (phénomène de *Skew*). Le fonctionnement du registre n'est alors plus synchrone avec les autres circuits du système.



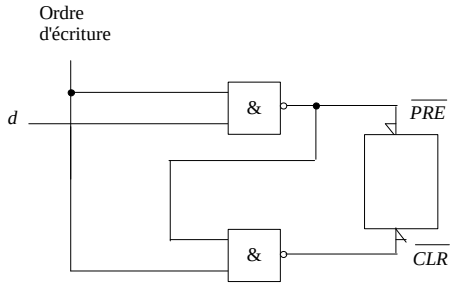
1.3.4. Ecriture asynchrone

Il faut utiliser les entrées asynchrones (entrées prioritaires PreSet PRE de mise à 1 et Clear CLR de mise à 0) de chaque bascule pour forcer l'information qui doit être écrite.
Le forçage ne doit se faire qu'au moment de l'écriture, ce qui signifie qu'il est nécessaire d'ajouter des circuits à chaque bascule de façon à synchroniser les entrées de forçage par un ordre particulier généralement appelé ordre d'écriture ou de chargement (*LOAD*).

La commande d'écriture est dans ce cas généralement asynchrone. Le principe de la synchronisation consiste à transformer une bascule asynchrone en une bascule *D* latch. Le schéma correspondant est donné par les figures ci-après :



Cas d'une bascule sensible sur les niveaux 1 des entrées asynchrones.

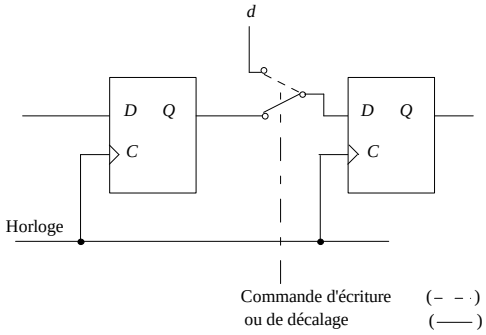


Cas d'une bascule sensible sur les niveaux 0 des entrées asynchrones.

On remarque que l'utilisation d'un opérateur NAND permet à la fois la synchronisation avec l'ordre d'écriture et la complémentation de l'information d'entrée *d*.

1.3.5. L'écriture synchrone

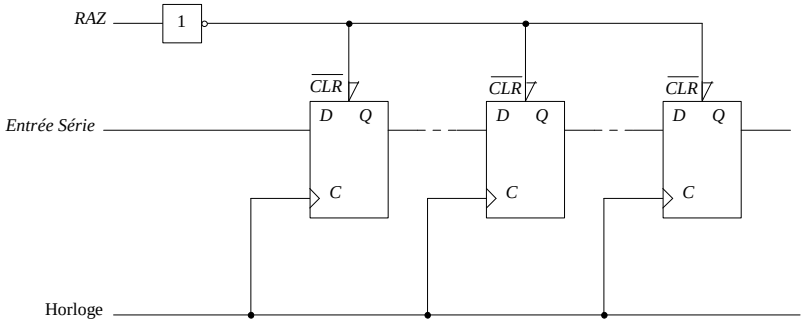
La sortie d'une bascule D recopie son entrée au moment de la phase active de l'horloge. Pour provoquer l'écriture d'une donnée en synchronisme avec l'horloge il faut placer cette donnée sur l'entrée *D* de la bascule.



1.3.6. L'initialisation

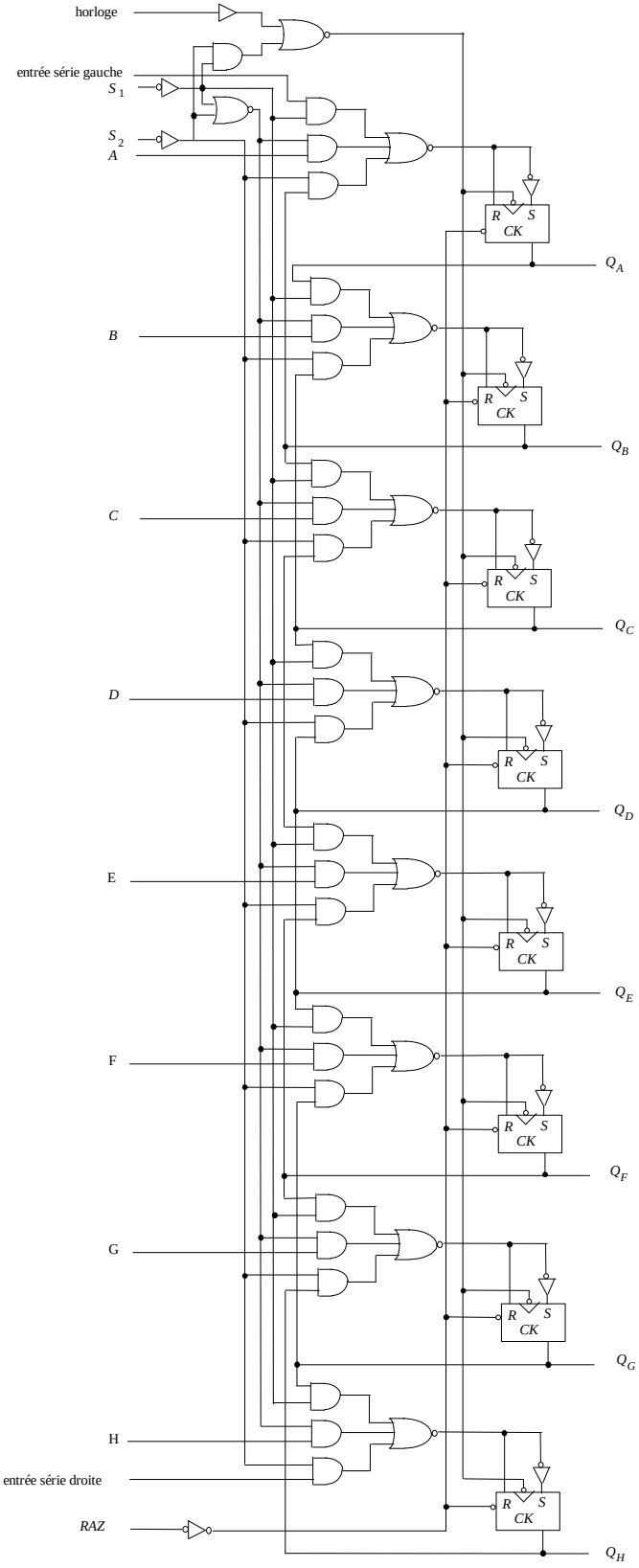
Cette initialisation consiste à imposer pour toutes les bascules du registre la même valeur, en général 0, à l'aide d'une commande asynchrone.

Le schéma d'un registre disposant d'une remise à zéro RAZ générale est donné par la figure ci-après :



1.3.7. Schéma d'un registre universel

Pour permettre la réalisation de l'une des fonctions possibles du registre, l'interconnexion entre les bascules doit être modifiée. Le choix de cette interconnexion est facilement réalisé en utilisant un circuit multiplexeur devant chaque entrée *D* des bascules. Le schéma d'un système commercialisé est donné par la figure suivante.
On distingue facilement les multiplexeurs réalisant la sélection des interconnexions et le blocage de l'horloge pour réaliser la fonction mémoire. Ce circuit dispose également d'une remise à zéro globale pour une éventuelle initialisation.



2. LES COMPTEURS *(généralisation de la notion de registre)*

Un compteur est un ensemble de bascules dont les sorties forment un mot binaire et qui compte dans une séquence donnée à chaque coup d'horloge. Il est représenté par un *automate d'états fini* encore appelé *machines d'états* : c'est l'horloge qui fait passer d'un état au suivant, contrairement au *séquenceur* dont les changements d'état sont soumis aussi à des actions et des transitions.

2.1. Présentation

Les compteurs sont des éléments essentiels de logique séquentielle ; ils permettent en effet d'établir une relation d'ordre de succession d'évènements.

Leur emploi ne se limite pas, loin de là, aux systèmes arithmétiques. Ils sont utiles partout où il est souhaitable de définir facilement une suite d'états.

L'élément de base des compteurs est, comme pour les registres, une bascule. L'état du compteur est défini par le nombre binaire formé avec l'ensemble des sorties des bascules.

La synthèse d'un compteur consiste à définir les niveaux logiques des commandes des bascules pour assurer le passage successif d'un état à l'autre suivant l'ordre du cycle prévu (échelle de comptage).

Ils sont classés en deux catégories suivant leur mode de fonctionnement.

- Les compteurs asynchrones ou compteurs série. La caractéristique principale des compteurs asynchrones est la propagation en cascade de l'ordre de changement d'état des bascules. L'horloge synchronise la 1ère bascule, dont la sortie va synchroniser la bascule suivante.

- Les compteurs synchrones ou compteurs parallèles. Le signal d'horloge synchronise toutes les bascules simultanément.

2.2. Les compteurs asynchrones

Dans un compteur asynchrone, l'horloge déclenche la bascule B_1 dont la sortie sert de signal d'horloge à la bascule B_2.

Plus généralement, le signal d'horloge d'une bascule B_i est issu d'une combinaison logique des sorties des bascules B_j (avec *j* inférieur à *i*).

Comme les sorties changent en cascade après le signal d'horloge, il va exister un désynchronisme dans l'évolution des sorties des bascules. Ceci justifie le nom de ces compteurs.

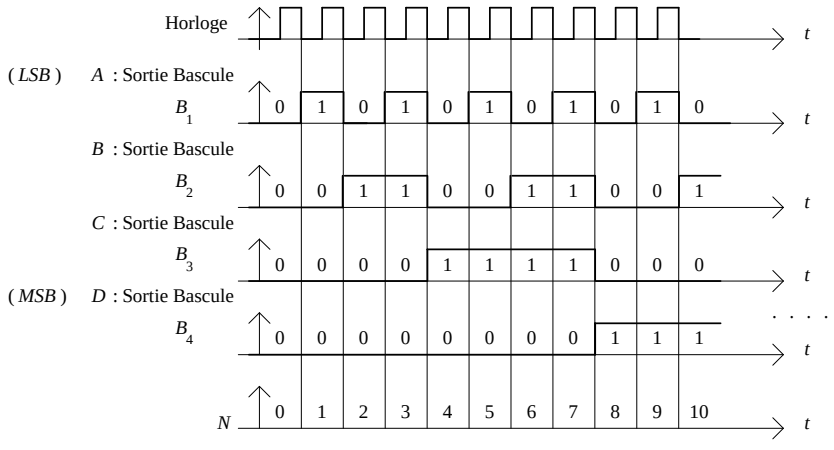
2.2.1. Compteur binaire

C'est le plus simple des compteurs asynchrones : les sorties des bascules qui le composent évoluent, au rythme de l'horloge, de manière à représenter la succession croissante des nombres exprimés en base 2 (binaire pur).

Ex.: Compteur binaire 4 bits :

Les retards ne sont pas représentés pour des raisons de lisibilité.
Tous les états possibles (16) sont ici présents dans l'automate mais on peut interrompre la séquence du compteur.

N	*D* B_4	*C* B_3	*B* B_2	*A* B_1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
	.	.	.	.
15	1	1	1	1



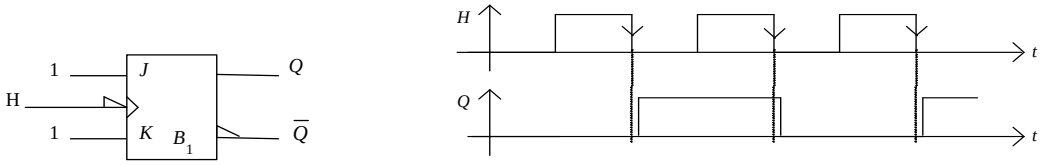
Temps de propagation non représentés

Si l'on affecte les poids 1, 2, 4, 8 respectivement aux sorties *A, B, C, D* des bascules B_1, B_2, B_3, B_4, le nombre binaire ainsi représenté suit l'ordre croissant de 0 à 15.

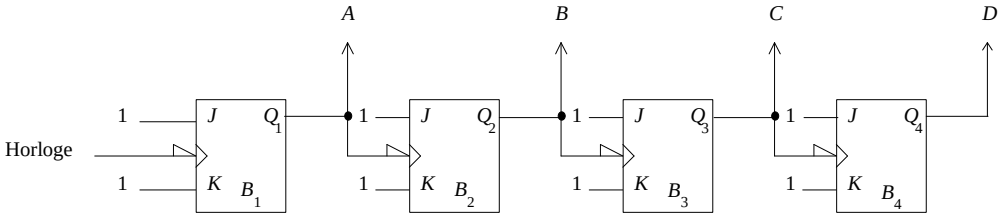
Un compteur binaire m bits compte donc de 0 à 2^{m-1}.

On remarque facilement que la première bascule B_1 (*A*) change d'état à chaque impulsion d'horloge, que la sortie *B* change d'état à chaque fois que la sortie *A* présente une transition descendante ($1 \rightarrow 0$) et ainsi de suite. En d'autres termes, la période de la sortie *A* est égale au double de la période d'horloge, la période de *B* est égale à 2 fois celle de la sortie *A*, etc ...
Ceci est facilement réalisé en utilisant des bascules câblées en type *T*, par exemple des bascules *JK* avec $J = K = 1$.

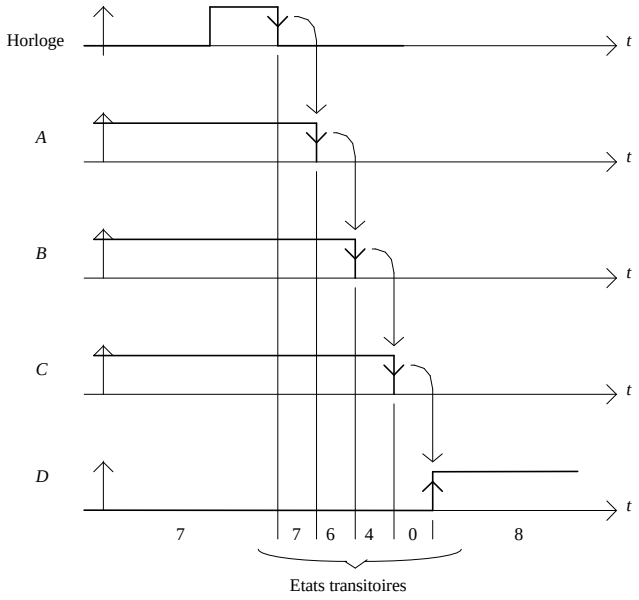
Rappel : Bascule JK câblée en Diviseur par 2 (fonctionnement type *T*)



Le schéma d'un compteur binaire est dans ce cas :



Afin d'illustrer le désynchronisme qui apparaît entre les sorties, examinons la transition entre 7 et 8. Ce cas est particulièrement perturbé puisque toutes les sorties changent.



La structure en cascade implique l'accumulation des retards entre la transition descendante de l'horloge et la stabilisation des sorties des bascules.

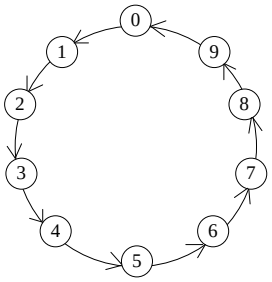
Comme les sorties changent les unes après les autres, il apparaît des états transitoires indésirables.
Sur le chronogramme ci-dessus, ces différents états sont repérés par leur équivalent décimal.

Pour un compteur asynchrone, si *n* bascules changent d'état après une impulsion d'horloge, il existe (*n* -1) états transitoires.

2.2.2. Compteur modulo N (Interruption de la séquence)

On appelle compteur modulo *N*, un compteur décrivant la succession des nombres binaires compris entre 0 et *N* -1, c'est à dire la suite des chiffres d'une base *N* traduite en binaire.

Par exemple, pour *N* = 10, la succession des états du compteur est donnée par le cycle suivant (*N* = 10 états), alors que le cycle complet non interrompu compte jusqu'à 16 états :



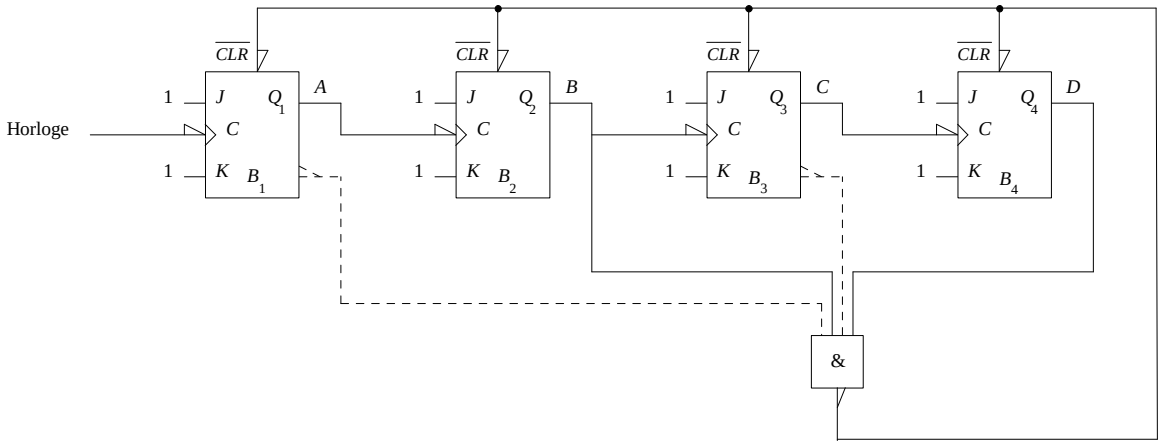
Pour réaliser un tel compteur il existe deux types de solutions :

- Un rebouclage asynchrone,
- Un conditionnement des entrées des bascules.

Rebouclage asynchrone

Le compteur modulo *N* est dans ce cas considéré comme un compteur binaire m bits comptant de 0 à 2^{m-1} dont le cycle est interrompu à $N = 2^{m-1}$. En effet, entre 0 et 9 par exemple, la succession des états est identique pour les deux types de compteurs : compteur binaire 4 bits et compteur modulo 10.

Si, dans l'état 9, une impulsion d'horloge est appliquée au compteur, celui-ci inscrira la combinaison 10 en binaire. Cet état est indésirable. Par conséquent, dès qu'il est détecté par décodage, une remise à zéro (Clear CLR) est automatiquement imposée au compteur pour satisfaire le cyle voulu. Ce qui donne le schéma suivant :



Remarques :

- Le décodage de 10 se limite à vérifier que *B* et *D* sont à 1, les combinaisons 11 à 15 ne devant pas apparaître en fonctionnement normal.

- Il ne faut pas limiter la remise à zéro aux seules bascules qui sont à 1. En effet, si seules les bascules *B* et *D* sont remises à 0, le changement d'état de *B* provoque le changement d'état de *C*.

- Enfin cette solution ne donne satisfaction que si toutes les bascules ont des temps de réaction semblables.

Conditionnement des entrées

Dans un compteur asynchrone il existe deux possibilités de commander une bascule :

- par action sur l'horloge,
- par action sur les entrées des bascules.

La succession des états dans l'exemple précédent ($N = 10$) est donné par la table suivante dans laquelle le passage d'une ligne à la suivante est conditionné par une impulsion d'horloge.

N	*D*	*C*	*B*	*A*
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
0	0	0	0	0

La bascule *A* change d'état à chaque impulsion d'horloge. Elle est donc câblée en type T.

La bascule *B* change d'état sur chaque transition $1 \rightarrow 0$ de la sortie A sauf pour la transition $9 \rightarrow 0$.

Dans ce cas particulier il faut agir sur les entrées de la bascule pour qu'elle ne change pas d'état malgrè la transistion active sur son horloge.

Pour tous les états de 0 à 7 inclus il faut impérativement que les entrées soient telles que le fonctionnement correspondant soit du type *T*.

Pour la combinaison 9 il faut conditionner la bascule pour qu'elle reste en mémoire.

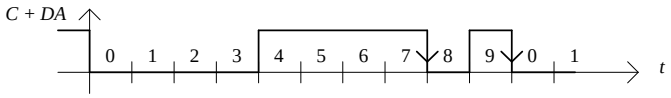
Pour la combinaison 8 il existe un degré de liberté.

Pour une bascule JK par exemple, la solution consiste à imposer les entrées :
$J = K = 1$ pour les états 0 à 7, et $J = K = 0$ pour l'état 9 et éventuellement 8

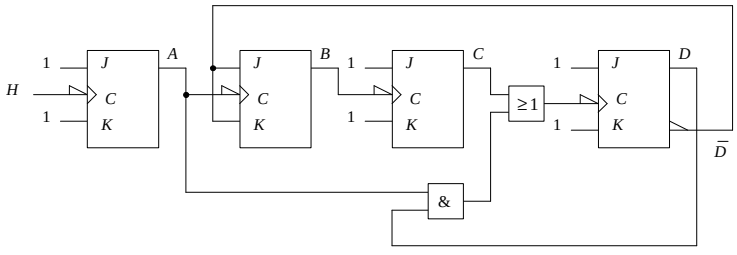
La bascule *C* change d'état à chaque transition négative de la sortie *B*.

Enfin la bascule *D* ne peut pas être commandée uniquement par la sortie *C*. En effet, cette sortie présente une transition négative qui va provoquer le passage de 7 à 8 mais il n'existe pas d'autre transition négative pour le changement de 9 à 0.

Il faut donc trouver un autre signal qui présente une transition négative après l'état 9 et après l'état 7, pour commander la bascule de sortie *D*. Le signal fabriqué avec l'équation logique $C + DA$ satisfait cette condition.



Le schéma du compteur modulo 10 réalisé par la méthode du conditionnement des entrées est le suivant :



2.3. Les compteurs synchrones

Les compteurs synchrones permettent d'une part d'éliminer les états transitoires des sorties et d'autre part de rendre possible l'exécution d'un cycle quelconque. Pour satisfaire ces exigences et en particulier la première, il est indispensable que toutes les bascules soient synchronisées par le même signal : le signal d'horloge. En conséquence, il ne reste plus qu'un seul degré de liberté pour conditionner l'évolution de chaque bascule. Les entrées synchrones de chaque bascule doivent être calculées pour que le compteur suive la succession d'état prévue.

La synthèse d'un compteur synchrone, qui consiste à concevoir un système séquentiel à partir du cycle de fonctionnement souhaité, peut être effectuée par exemple par la méthode de Marcus.

Exemple : Réaliser la séquence suivante avec des bascules *JK* : N = 0, 8, 12, 14, 7, 11, 13, 6, 3, 9, 4, 10, 5, 2, 1, 0, ...

Compteur 15 états ($\rightarrow$ 4 bits $A,B,C,D \rightarrow$ 4 bascules de sorties respectives A,B,C,D)

Ces nombres décimaux s'écrivent en binaire avec 4 bits. En conséquence, cette séquence impose l'utilisation de quatre bascules, d'où la table suivante (utilisant la table des transitions d'une bascule JK et où *ABCD* sont codées en code BCD) :

	Sorties				Entrées							
N	A	B	C	D	J_A	K_A	J_B	K_B	J_C	K_C	J_D	K_D
0	0	0	0	0	1	X	0	X	0	X	0	X
8	1	0	0	0	X	0	1	X	0	X	0	X
12	1	1	0	0	X	0	X	0	1	X	0	X
14	1	1	1	0	X	1	X	0	X	0	1	X
7	0	1	1	1	1	X	X	1	X	0	X	0
11	1	0	1	1	X	0	1	X	X	1	X	0
13	1	1	0	1	X	1	X	0	1	X	X	1
6	0	1	1	0	0	X	X	1	X	0	1	X
3	0	0	1	1	1	X	0	X	X	1	X	0
9	1	0	0	1	X	1	1	X	0	X	X	1
4	0	1	0	0	1	X	X	1	1	X	0	X
10	1	0	1	0	X	1	1	X	X	1	1	X
5	0	1	0	1	0	X	X	1	1	X	X	1
2	0	0	1	0	0	X	0	X	X	1	1	X
1	0	0	0	1	0	X	0	X	0	X	X	1

X : état indifférent (0 ou 1) ≡ également noté $\not{0}$

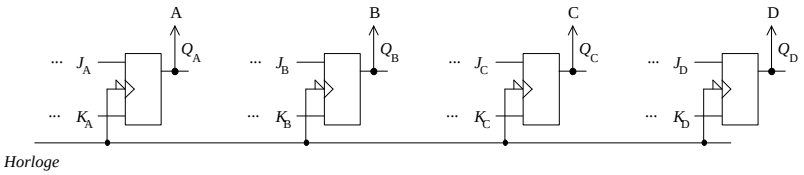
Rappel

Table des transitions Bascule JK (Synthèse) - (Horloge active)

Transition $Q_{n-1} \rightarrow Q_n$	***J***	***K***
$0 \rightarrow 0$	0	X
$0 \rightarrow 1$	1	X
$1 \rightarrow 1$	X	0
$1 \rightarrow 0$	X	1

La combinaison 1111 (15 en décimal) n'apparaît jamais dans la séquence désirée. C'est donc une combinaison disponible pour la simplification et dans les tableaux de Karnaugh cette case sera remplie avec un X.

Squelette du circuit (incomplet) : *Synthèse avec des bascules JK negative edge triggered par exemple*
(l'utilisation de bascules JK positive edge triggered est reviendrait au même)



Après simplification des états équivalents et des fonctions logiques, les entrées *J* et *K* des bascules ont pour équation (8 tables de Karnaugh d'entrées A, B, C, D et de sortie J_A, K_A, J_B, K_B, ..., J_D, K_D) :

Exemple : Calcul de J_A / K_A :
(double table de Karnaugh)

J_A / K_A CD \ AB	00	01	11	10
00	1X	10	X0	X0
01	0X	0X	X1	X1
11	1X	1X	XX	X0
10	0X	0X	X1	X1

Calcul de J_A :
(simple table de Karnaugh)

J_A CD \ AB	00	01	11	10
00	1	1	X	X
01	0	0	X	X
11	1	1	X	X
10	0	0	X	X

$J_A = \overline{C}\,\overline{D} + CD = \overline{C \oplus D}$

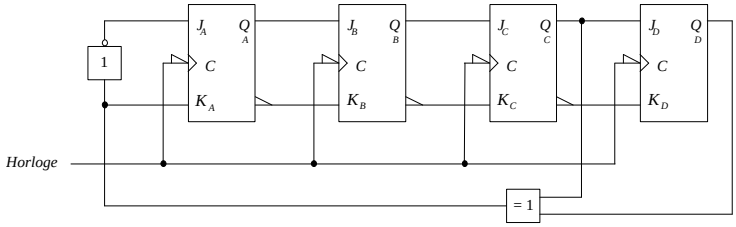
$J_A = \overline{C \oplus D}$ $\quad K_A = C \oplus D$

$J_B = A$ $\quad K_B = \overline{A}$

$J_C = B$ $\quad K_C = \overline{B}$

$J_D = C$ $\quad K_D = \overline{C}$

Circuit (final)



Automate des états du compteur

On retrouve bien le cycle voulu :

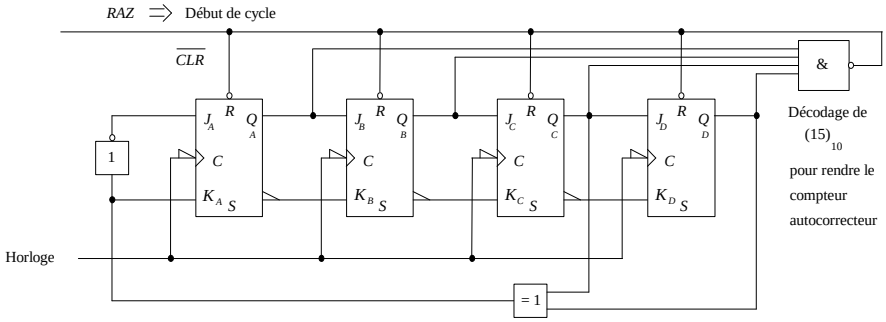
N = 0, 8, 12, 14, 7, 11, 13, 6, 3, 9, 4, 10, 5, 2, 1, 0, ...

avec la séquence des états du compteur :

ABCD = 000→1000→1100→1110→0111→1011→0110→0011→1001→0100→1010→0101→0010→0001→0000→ …

Si, à la mise sous tension, le compteur se positionne dans la combinaison interdite (15 ≡ 1111) compte tenu du schéma, il y restera malgrè les impulsions d'horloge.

Il faut donc prévoir une action manuelle ou automatique pour retourner dans le cycle (utilisation des entrées asynchrones d'initialisation *S* et *R* d'une bascule *JK*).



Un circuit de décodage de la combinaison interdite permet de remettre le compteur dans son cycle (**compteur autocorrecteur**) si par malheur cette combinaison apparaissait. Dans le schéma ci-dessus la remise en cycle est faite par une remise à zéro de toutes les bascules puisque la combinaison $(0)_{10}$ appartient au cycle.

Initialisation d'un compteur

L'état initial d'un compteur est défini à l'aide des entrées asynchrones d'initialisation (*Preset* et *Clear*).

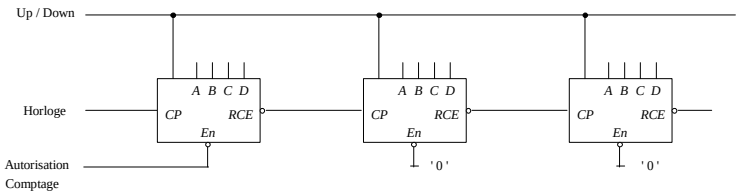
2.4. Mise en cascade de compteur

(ex : chronomètre à plusieurs digits, chaque digit est un compteur → les compteurs doivent être connectés entre eux → mise en cascade)

Les compteurs commercialisés délivrent bien évidemment la valeur contenue dans le compteur, et éventuellement une information supplémentaire permettant la mise en cascade de plusieurs boîtiers, soit *RCE* cette sortie (également appelée *TC* : Terminal Count).

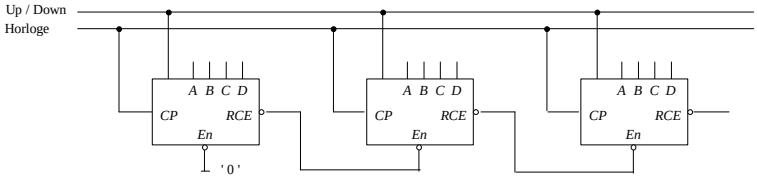
La commande des différents boîtiers de comptage peut se faire suivant deux grands principes :

- La mise en cascade asynchrone. La sortie *RCE* (ripple count enable) d'un boîtier sert d'horloge du boîtier suivant.



La conséquence de la mise en cascade asynchrone est que les sorties du second boîtier sont décalées dans le temps par rapport aux sorties du premier.

- La mise en cascade synchrone. Comme l'horloge est commune à tous les boîtiers, il faut conditionner l'évolution du second boîtier par la commande (*En* ou *CE*) d'autorisation de comptage.



3. LES SEQUENCEURS *(généralisation de la notion de compteur)*

3.1. Présentation

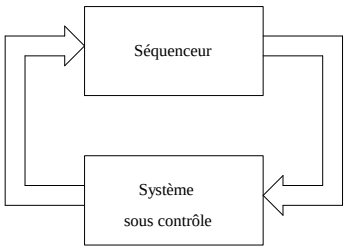
Les séquenceurs sont des systèmes séquentiels dont l'*état* évolue en fonction d'évènements (qui peuvent être divers : horloge, contact, interruption ... et non plus seulement l'horloge comme pour les compteurs) appelés *actions*. La séquence des états et des transitions marquant les changements d'états (les actions) s'appelle un *automate* ou *séquenceur* ou encore *machine d'états*. (*Ex.:* distributeur de boissons)

On trouve principalement des séquenceurs :

- Dans les automatismes ou processus automatiques industriels où ils jouent le rôle d'un automate capable de diriger les opérations qui doivent se dérouler dans un ordre prévu à l'avance;

- Dans les calculateurs où ils sont chargés d'un rôle d'organisateur de la succession des opérations à réaliser selon un programme préétabli.

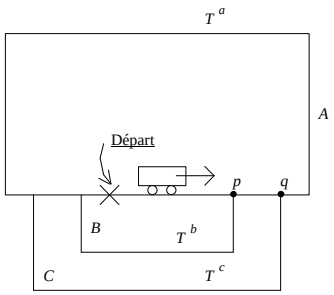
Un séquenceur n'est qu'exceptionnellement seul. En général il commande un système. Dans la plupart des cas, ce dernier l'informe de son état, par exemple la fin d'exécution d'un tâche, la présence d'un débordement ou d'un résultat nul, etc . Le séquenceur doit pouvoir, si nécessaire, prendre des décisions suivant l'état du système.



3.2. Synthèse d'un séquenceur (câblé → bascules JK)

Exemple : Train électrique

Soit un train électrique devant effectuer 3 boucles *A*, *B*, *C* selectionnables par aiguillages *p* et *q*. Le passage dans une boucle est détecté par un contact (*T*) remontant après le passage du train → (le train roule en marche avant uniquement (pas de marche arrière) ou peut aussi se trouver à l'arrêt).



Aiguillages :

$$\left|\begin{matrix} p \\ q \end{matrix}\right. = \begin{cases} 0 : \text{non devie} \\ 1 : \text{devie} \end{cases}$$

Contacts : (T)

$$\left|\begin{matrix} a \\ b \\ c \end{matrix}\right. = \begin{cases} 0 : \text{repos} \\ 1 : \text{passage du train(retombee a 0 apres passage)} \end{cases}$$

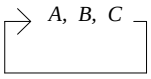
Etat initial :

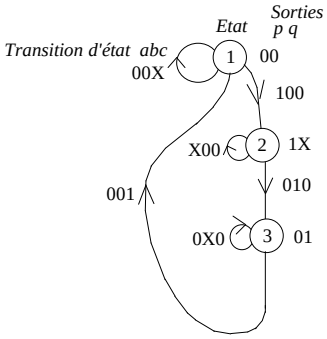
Train au départ sorties *p* et *q* à 0

Si on change d'itinéraire, il faut refaire toute la synthèse (→ nouveau circuit à base de bascules JK) ≠ solution programmable où seul le programme change.

Les contacts *a*, *b*, *c* sont exclusifs (le train ne peut être à la fois en *a*, en *b* et en *c*).

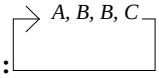
3.2.0. Automate (des états) ≡ Graphe de fluence pour l'itinéraire désiré :





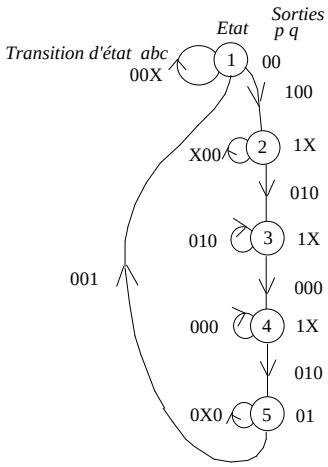
Note : Il y a changement d'état chaque fois que les sorties changent. *Les états sont liés aux sorties. Les transitions sont liées aux entrées* : une modification des entrées provoque (éventuellement) une transition d'état (modification des sorties).

Etude pour un second itinéraire désiré :



3.2.1. Automate (des états) ≡ Graphe de fluence :

Entrées du système : *a*, *b*, *c*
Sorties du système : *p*, *q*



(X : 0 ou 1 : sans importance)

3.2.2. Table des états :

Etat résultant / transition (c b a) / état	000	100	110	010	011	111	101	001	Sorties p	Sorties q
1	1	2						1	0	0
2	2	2		3					1	X
3	4			3					1	X
4	4			5					1	X
5	5			5				1	0	1
	a b c									

Les cases vides sont des X (cases indifférentes)

3.2.3. Simplification de la table : (Recherche des états équivalents)

2 états sont *équivalents* s'ils ont *mêmes sorties et mêmes transitions.*

Etats pouvant être équivalents : 2 -3 si 2 - 4 le sont
2 - 4 si 3 - 5 le sont
3 - 4 si 3 - 5 le sont

or 3 et 5 ne sont pas équivalents → 2 - 4 ne le sont pas
→ 2 - 3 ne le sont pas

→ pas d'états équivalents → pas de simplification de la table (la simplification aurait conduit à remplacer dans la table les états équivalents à un état donné par cet état donné et réunir les transitions correspondantes)

3.2.4. Attribution des variables de sortie des bascules (Q_i) : table des adresses

On a d'autant plus besoin de variables Q_i (sorties de bascules) qu'il y a d'états à adresser dans la proportion :

n états à adresser
m variables Q_i $\longrightarrow$ $n = 2^m$

On doit donc avoir suffisamment de variables Q_i pour adresser tous les états : $n \leq 2^m$.

Ici : $n = 5$ → au moins $m = 3$ variables sont nécessaires : $Q_2\ Q_1\ Q_0$ pour le codage des états.

	état (Q2 Q1 Q0 / transition c b a)	000	001						000	p	q
000 ≡	1	000	001						000	0	0
001 ≡	2	001	001		011					1	X
011 ≡	3	010			011					1	X
010 ≡	4	010			110					1	X
110 ≡	5	110			110				000	0	1

Exemple Synthèse avec des bascules J K synchrones (possibilité aussi avec des bascules asynchrones)

3.2.5. Table de Karnaugh des bascules $J_i\ K_i$ (1 bascule *JK* par variable *Q*)

3 variables $Q \rightarrow$ 3 bascules *J K* : 3 doubles tables de Karnaugh : $J_0 K_0,\ J_1 K_1,\ J_2\ K_2$

Ex : Table de Karnaugh de $J_2\ K_2$ (bit *MSB*) pour la sortie Q_2

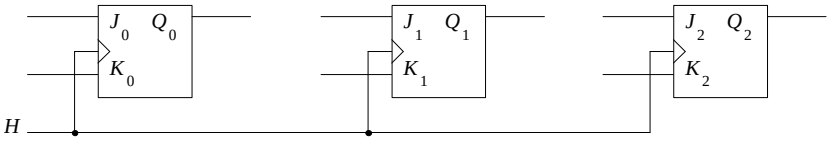
rappel :	transition	*J K*
	$0 \rightarrow 0$	0 X
	$0 \rightarrow 1$	1 X
	$1 \rightarrow 1$	X 0
	$1 \rightarrow 0$	X 1

$J_2\ K_2$

Q_2 \ $Q_1 Q_0$ \ $c b a$	000	001	011	010	110	111	101	100
0	0 X	0 X						0 X
1	0 X	0 X		0 X				
1	0 X			0 X				
1	0 X			1 X				
0	X 0			X 0				X 1
0								
0								

$$\rightarrow \begin{cases} J_2 = \overline{Q_2} Q_1 \overline{Q_0}\ \overline{a} b \overline{c} \\ K_2 = Q_2 Q_1 \overline{Q_0}\ \overline{a}\ \overline{b} c \end{cases}$$

sans simplifier au maximum car ici la table est à 6 variables (>4) → méthode de Karnaugh inutilisable.



3.2.6. Equation des sorties :

$p\ q$

Q_2 \ $Q_1 Q_0$	00	01	11	10
0	00	1 X	1 X	1 X
1	X X	X X	X X	0 1

D'après la table *3.2.4.* :

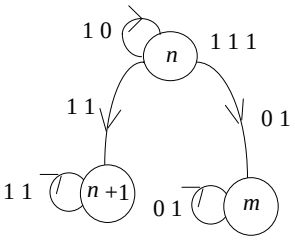
$$\rightarrow \begin{cases} p = Q_0 + Q_1 \overline{Q_2} \\ q = Q_1 \end{cases}$$

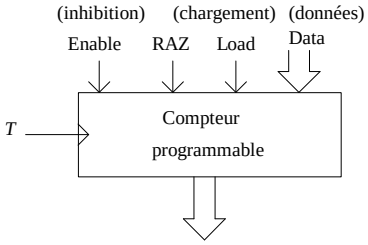
4 ANNEXE. LOGIQUE SEQUENTIELLE 2

LES SEQUENCEURS PROGRAMMABLES

1. Utilisation d'un compteur programmable

Toute transition d'état est du type général :





→ Commandes appliquées :
- on reste dans l'état n : → enable
- on passe à $n + 1$: → rien (fonctionnement normal du compteur) → horloge
- on passe à m : load data
- initialisation du compteur : RAZ

Plus de table de Karnaugh à calculer, ni entrées *J K*, mais il faut programmer les bonnes instructions sur les entrées :

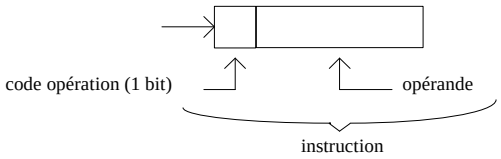
Enable
RAZ
Load
Data

1.1. Codage et décodage des actions

- actions sur les sorties :

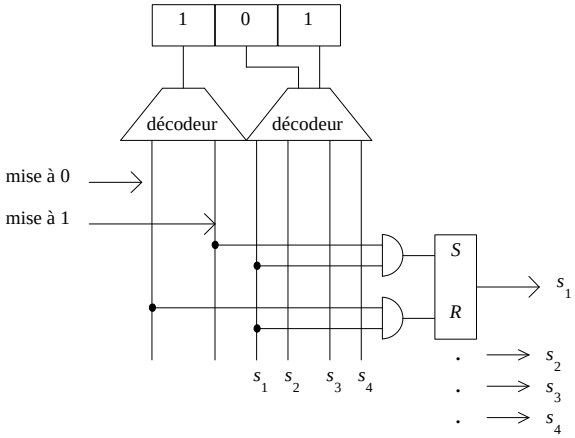
mise à 0
mise à 1 ⟶ il suffit d'un bit

mais il faut préciser sur quelle sortie porte l'action :



code opération :	1 :	mise à 1
	0 :	mise à 0
opérande :	0 0 :	sortie s_1
	0 1 :	sortie s_2
	1 0 :	sortie s_3
	1 1 :	sortie s_4
		(exemple à 4 sorties)

Ex : mise à 1 de s_2
→ décodage de :



Architecture interne du séquenceur programmable

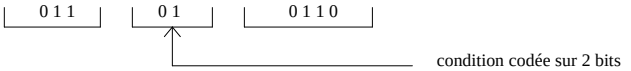
- *opérations*	- inhibition	conditionnelle
	- rupture de séquence (load)	conditionnelle ou inconditionnelle
	- mise à 0	conditionnelle ou inconditionnelle

→ 5 opérations : → 3 bits pour le code opération

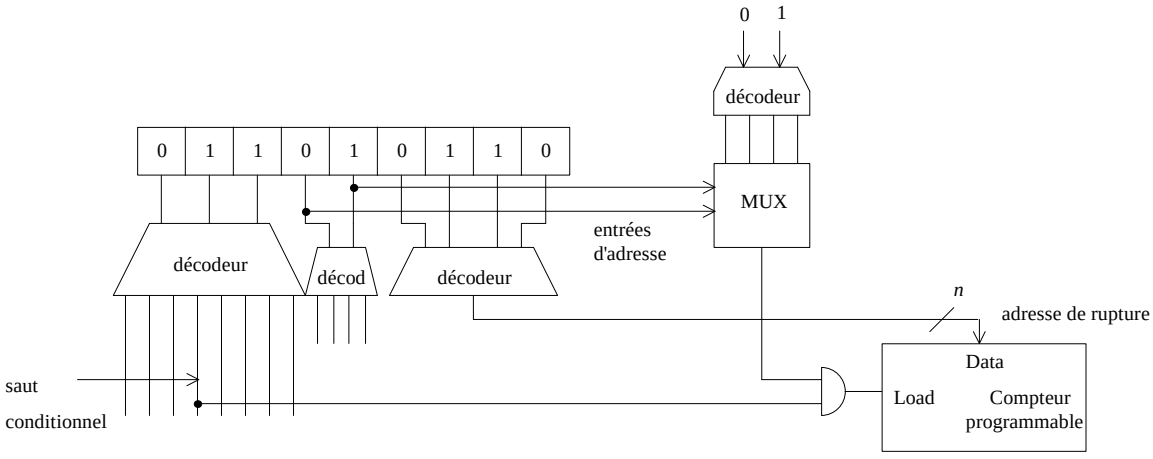
opération :	0 0 0	mise à 0 d'une sortie
	0 0 1	mise à 1 d'une sortie
	0 1 0	inhibition conditionnelle
	0 1 1	rupture conditionnelle
	1 0 0	rupture inconditionnelle
	1 0 1	RAZ conditionnelle
	1 1 0	RAZ inconditionnelle

opérande :	sortie
	condition
	adresse de rupture

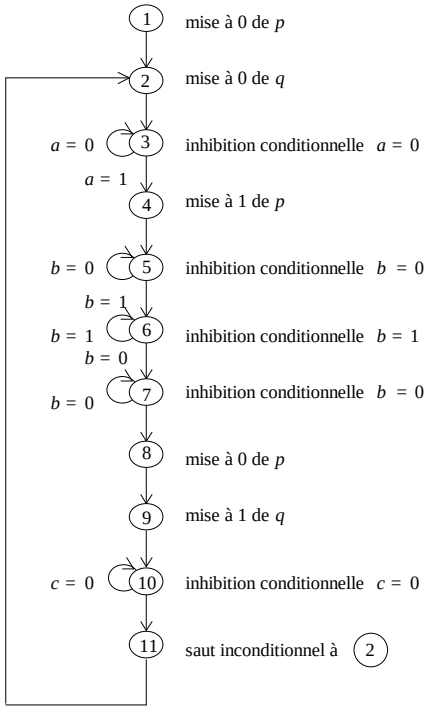
Ex : Rupture de séquence à l'adresse 0 1 1 0 (conditionnel)



→ décodage de :



2. Application sur l'exemple du train électrique



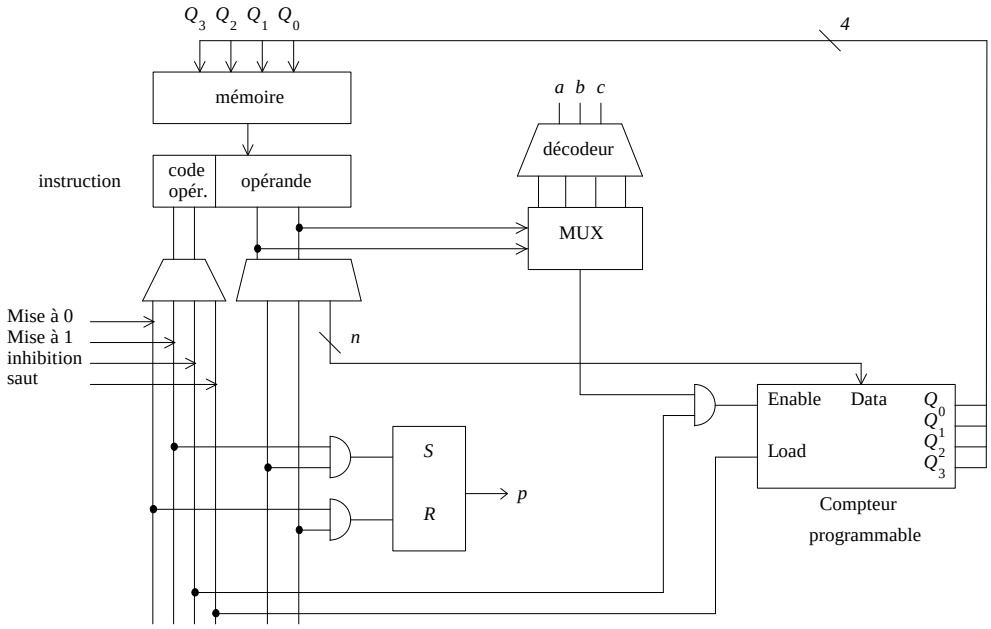
codage :

actions :	mnémorique :	code binaire :
Mise à 0 de sorties	MAZ	0 0
Mise à 1 de sorties	MAU	0 1
Inhibition conditionnelle	INH	1 0
Saut Inconditionnel	JMP	1 1
opérandes :		
Sortie *p*	*P*	0 0
Sortie *q*	*Q*	0 1
conditions :		
a = 0	*A* 0	0 0
b = 0	*B* 0	0 1
c = 0	*C* 0	1 0
b = 1	*B* 1	1 1
adresse de saut 2	2	0 0

Programme :

état	*mnémorique*	*code binaire*
0 0 0 0	MAZ *P*	0 0 0 0
0 0 0 1	MAZ *Q*	0 0 0 1
0 0 1 0	INH *A* 0	1 0 0 0
0 0 1 1	MAU *P*	0 1 0 0
0 1 0 0	INH *B* 0	1 0 0 1
0 1 0 1	INH *B* 1	1 0 1 1
0 1 1 0	INH *B* 0	1 0 0 1
0 1 1 1	MAZ *P*	0 0 0 0
1 0 0 0	MAU *Q*	0 1 0 1
1 0 0 1	INH *C* 0	1 0 1 0
1 0 1 0	JMP 2	1 1 0 0

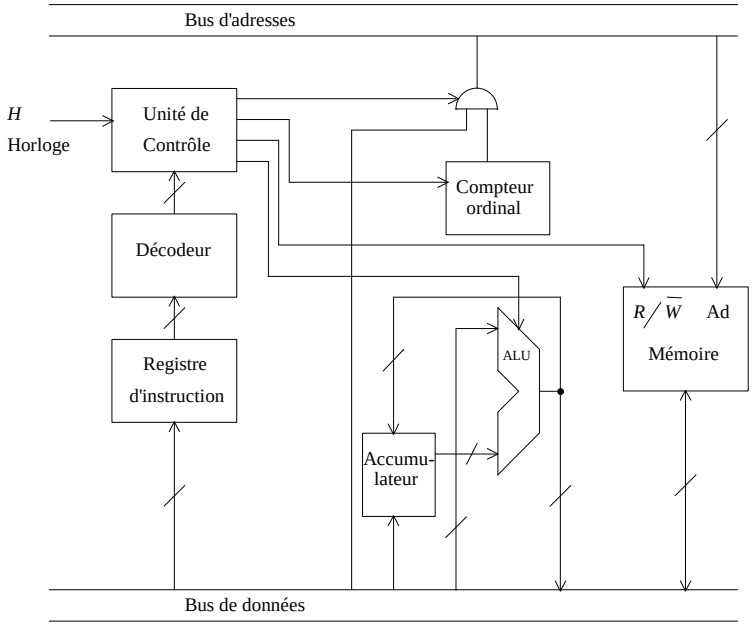
Matérialisation :



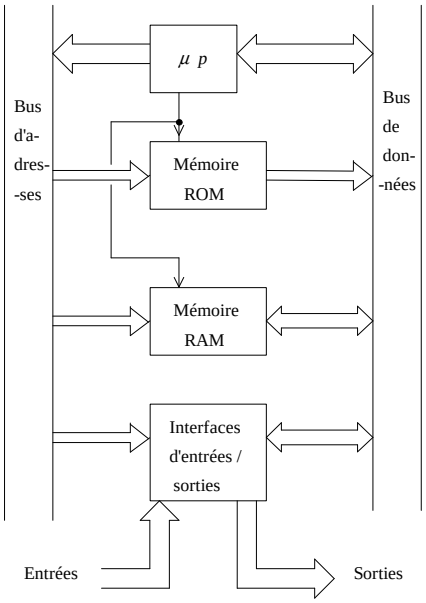
LES MICROPROCESSEURS

3. Utilisation d'un microprocesseur

Structure



Circuits associés



(μ p ≡ micro - processeur)

Le programme à exécuter (code machine) peut être développé en langage assembleur (codes mnémoniques symbolisant les codes binaires) ou en langage évolué soumis à un compilateur.

Les microproceseurs les plus répandus sont notamment à base d'une architecture 32 bits :
. Pentium de la famille INTEL,
. série 68000 de la famille MOTOROLA.

Des processeurs spécialisés ont des performances (puissance, rapidité ...) optimales pour des applications dédiées :
.Traitement du signal : processeur TMS de TEXAS INSTRUMENTS
.Graphisme ...

LES MICROCONTROLEURS

4. Utilisation d'un microcontrôleur

Un microcontrôleur est un mini-système, constitué de l'ensemble microprocesseur avec RAM, ROM, EEPROM, convertisseurs CAN/CNA et interfaces (≡ gestionnaires) d'Entrées/Sorties séries/parallèles, intégré dans une même structure (une même puce, un même circuit de silicium). Des programmes peuvent être développés sur émulateur (systèmes de développement) en langage assembleur du microcontrôleur (ou même en langage évolué à l'aide d'un compilateur), et exécutés autour d'un système d'exploitation logé également dans la mémoire du microcontrôleur.

Du fait de la richesse de sa constitution, un microcontrôleur est bien un mini système et peut être envisagé comme une solution générale à une majorité d'applications analogiques, numériques, mixtes, informatiques , embarquées ... ne nécessitant pas une vitesse extrême, pour laquelle il est préférable d'utiliser des circuits câblés, ou à la rigueur, des circuits programmables FPGA.

Les microcontrôleurs les plus répandus sont notamment à base d'une architecture 8 bits :
. 80C51 de la famille INTEL,
. 68HC11 de la famille MOTOROLA,
. les microcontrôleurs PICS intéressants pour leurs faibles encombrement et coût.

LES COMPOSANTS PROGRAMMABLES

5. Les composants programmables

L'utilisation de composants programmables (FPGA), programmables via un compilateur VHDL, présente l'avantage par rapport aux circuits câblés (bascules, portes logiques) de la souplesse tout en n'ayant pas l'inconvénient de lenteur d'exécution des autres solutions programmables puisque le composant programmable n'exécute pas un programme mais est constitué de matrices ET et OU programmées à volonté en bascules et portes logiques (synthèse effectuée par le compilateur VHDL).

6. Conclusion

Logique câblée ≡ utilisation de portes combinatoires et séquentielles.

- Avantage : rapidité du sytème.
- Inconvénient : structure figée → manque de souplesse pour une modification ou adaptation du système.

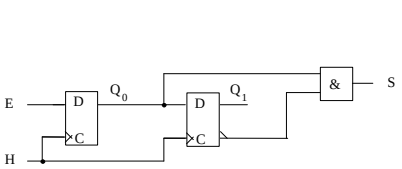
Logique programmée ≡ utilisation de séquenceur (compteur), d'un microprocesseur, d'un microcontrôleur ou d'un composant programmable (FPGA ...).

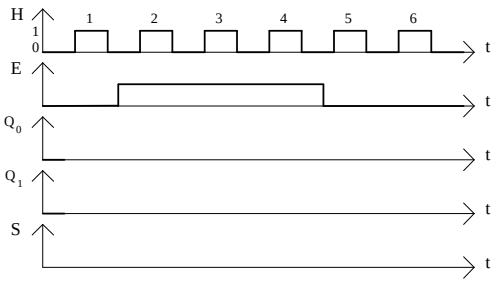
- Avantage : plus souple à modifier et concevoir.
- Inconvénient : plus lent que la logique câblée à cause du temps de décodage des instructions requis.

TD 4. LOGIQUE SEQUENTIELLE 2

1. Registres (Analyse)

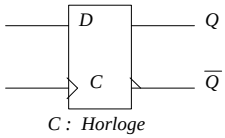
Analyser le fonctionnement du montage suivant et compléter le chronogramme : (Q_0 et Q_1 sont à l'état initial 0)





Rappel

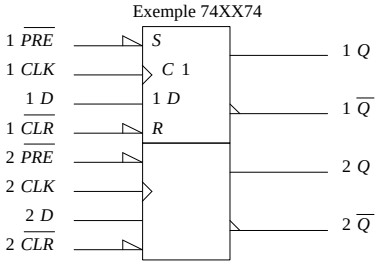
Bascule *D* > 0 edge triggered



C : Horloge

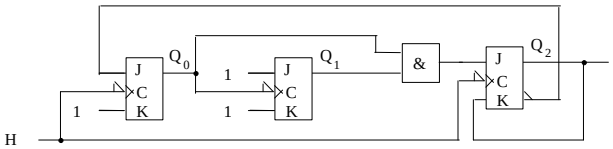
C	*D*	Q_n	
X	X	Q_{n-1}	Mémoire
↑	0	0	Recopie
↑	1	1	Recopie

Exemple 74XX74



2. Compteur asynchrone (Analyse)

- Donner la succession des états du compteur suivant, celui-ci étant supposé à l'état $Q_2Q_1Q_0 = 000$ initialement :

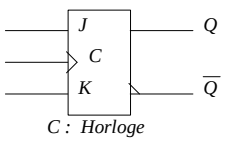


(les états furtifs ne sont pas à prendre en compte - leur durée est très petite devant la période d'Horloge car due au temps de propagation, au temps de réponse des circuits qui lui, n'est jamais à négliger)

- S'agit-il d'un compteur modulo 8 ? (un compteur 3 bits compte au maximum $2^3 = 8$ états possibles)
- Le compteur est-il autocorrecteur ?

Rappel

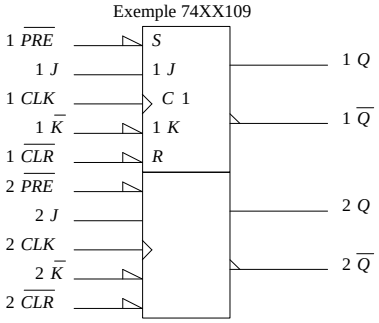
Bascule *JK* > 0 edge triggered



C : Horloge

C	*J*	*K*	Q_n	
X	X	X	Q_{n-1}	Mémoire
↑	0	0	Q_{n-1}	Mémoire
↑	0	1	0	Recopie de *J*
↑	1	0	1	Recopie de *J*
↑	1	1	$\overline{Q}_{n-1}$	Complément

Exemple 74XX109



3. Compteur synchrone (Synthèse)

a) Synthétiser un compteur 2 bits synchrone avec des bascules JK *positive edge triggered*, qui compte selon le cycle suivant (Initialisation à $Q_1Q_0 = 00$) : $Q_1Q_0 = 00 \rightarrow 01 \rightarrow 11 \rightarrow 00 \rightarrow \ldots$

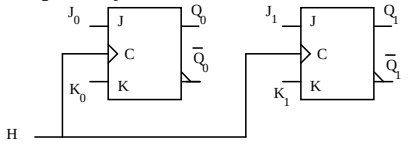
b) Vérifier en faisant l'analyse (chronogramme)

c) Que se passe-t-il si on démarre à l'état $Q_1Q_0 = 10$? Y-a-t-il autocorrection ?

d) S'il n'y a pas autocorrection, rendre le compteur autocorrecteur en reprenant la synthèse *a)* et en éliminant les choix *xx* effectués dans les tables de Karnaugh pour les forcer selon le cycle autocorrigé.

Rappel : Démarche de Synthèse

1. Se demander combien d'états *n* dans la séquence : $n = 3$
2. Combien de variables *m* de sortie utiliser : $2^{m-1} < n \leq 2^m \quad \rightarrow \quad m = 2$
3. Combien de bascules *JK* utiliser : m
4. Schéma squelette d'un compteur synchrone à bascules *JK* :



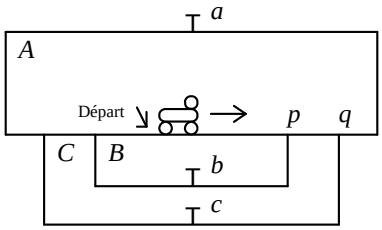
5. Tables de Karnaugh pour déterminer les entrées $J_i K_i$ des bascules en se servant de la table des transitions d'une bascule *JK* :

Transition $Q_{n-1} \rightarrow Q_n$	J	K
$0 \rightarrow 0$	0	X
$0 \rightarrow 1$	1	X
$1 \rightarrow 1$	X	0
$1 \rightarrow 0$	X	1

TD 4 ANNEXE. LOGIQUE SEQUENTIELLE 2

1. Séquenceur (commande de train électrique)

On considère un train électrique devant effectuer 3 boucles *A*, *B*, *C* sélectionnables par les aiguillages *p* et *q*.
Le passage dans une boucle est détecté par un contact (⊤) remontant après le passage du train :



(Train en marche avant ou à l'arrêt; pas de marche arrière)

Les contacts *a*, *b*, *c* représentent les entrées du système logique, avec la convention :
a, *b*, *c* = 0 à l'état de repos; *a*, *b*, *c* = 1 **pendant** le passage du train.

Les aiguillages *p* et *q* en constituent les sorties :
p, *q* = 0 si non déviation; *p*, *q* = 1 si déviation.

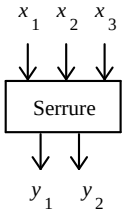
L'itinéraire désiré est le suivant : *A, B, B, C*

avec l'état initial : Train en position de Départ et les sorties *p* et *q* sont à 0.

Etablir le graphe de fluence (≡ graphe des états et ransitions) du système et effectuer la synthèse de ce système logique :
- en utilisant des bascules *JK* (logique câblée)
- à partir d'un compteur programmable (logique programmée).

2. Séquenceur (serrure électronique)

On considère une serrure électronique à 3 entrées x_1 , x_2 , x_3 et 2 sorties y_1 , y_2 :



En séquence, une entrée x_i ne peut être activée (≡ mise à 1) qu'une seule fois.
A un instant donné, 1 seule entrée x_i est à 1 à la fois.

Séquence à reconnaître : $x_1 = 1$ puis $x_2 = 1$ puis $x_3 = 1$, avec passages intermédiaires par 0, soit :
$x_1\ x_2\ x_3 =$ 100 → 010 → 001.
$y_1 \equiv 1$ à la fin de la séquence; $y_2 \equiv 1$ pour toute autre séquence.
(≡ 1 signifie mise à 1 et stabilisation à cet état, et non passage momentané à la valeur 1).

Etat initial : $x_1\ x_2\ x_3 = 000$ et $y_1\ y_2 = 00$

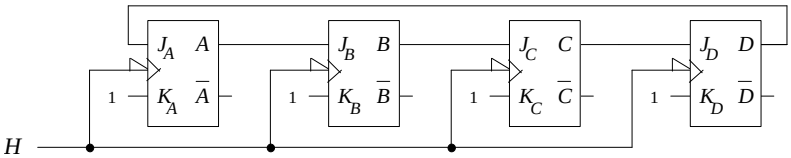
Etablir le graphe de fluence (≡ graphe des états et transitions) du système et effectuer la synthèse de ce système logique :
- en utilisant des bascules *JK* (logique câblée)
- à partir d'un compteur programmable (logique programmée).

3. Compteurs synchrones en anneau et non bouclé (Analyse)

Un compteur est dit autocorrecteur si, se trouvant dans un état hors de son cycle normal de comptage, il revient dans le cycle, éventuellement en plusieurs coups d'hologe.

1. Compteur en anneau à bascules JK

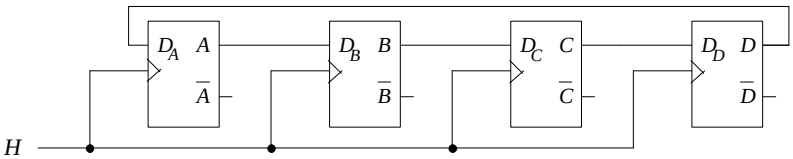
Soit le compteur :



1. Initialisé à l'état $DCBA = 0001$, tracer le chronogramme des signaux D, C, B, A.
2. En déduire son cycle de comptage (cycle normal de comptage) représenté comme une machine d'états (automate d'états finis).
3. L'état initial du compteur n'est plus précisé. Donner pour les 16 états initiaux possibles $DCBA$ du compteur, l'état $DCBA$ immédiatement futur (au coup d'horloge actif suivant l'état initial).
4. Dire si le compteur est autocorrecteur en justifiant votre réponse.

2. Compteur en anneau à bascules D

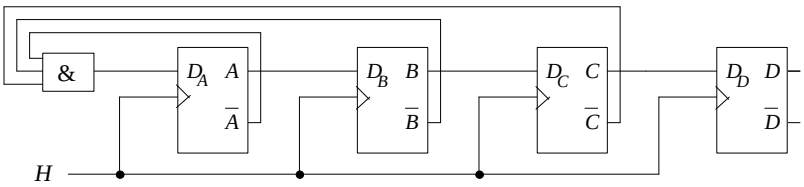
Soit le compteur :



1. Initialisé à l'état $DCBA = 0001$, tracer le chronogramme des signaux D, C, B, A.
2. En déduire son cycle de comptage (cycle normal de comptage) représenté comme une machine d'états (automate d'états finis).
3. L'état initial du compteur n'est plus précisé. Donner pour les 16 états initiaux possibles $DCBA$ du compteur, l'état $DCBA$ immédiatement futur (au coup d'horloge actif suivant l'état initial).
4. Dire si le compteur est autocorrecteur en justifiant votre réponse.

3. Compteur ouvert à bascules D

Soit le compteur :



1. Initialisé à l'état $DCBA = 0001$, tracer le chronogramme des signaux D, C, B, A.
2. En déduire son cycle de comptage (cycle normal de comptage) représenté comme une machine d'états (automate d'états finis).
3. L'état initial du compteur n'est plus précisé. Donner pour les 16 états initiaux possibles $DCBA$ du compteur, l'état $DCBA$ immédiatement futur (au coup d'horloge actif suivant l'état initial).
4. Dire si le compteur est autocorrecteur en justifiant votre réponse.

TP 4. LOGIQUE SEQUENTIELLE 2

1. Matériel nécessaire

- Oscilloscope
- Générateur de signaux Basses Fréquences (GBF)
- Alimentation stabilisée (2x[0-30 V] ⎓ + 1x[5 V] ⎓)
- Multimètre
- Moniteur MS05 (plaquette de câblage)
- Câbles : - 1 T, 1 BNC-BNC, 1 BNC-Banane, 1sonde oscilloscope, 6 fils Banane, petits fils.
- **Composants :** - 1 Résistances 1 kΩ (1/4 Watt)
  - 1 Résistance 1MΩ
  - 1 Condensateur 100 nF
  - 1 LED
  - 1 mini-interrupteur (horloge manuelle)

  Circuits logiques de la famille CMOS 4000

  - 1 *4049* : 6 NOT
  - 2 *4027* : 2 Bascules JK positive edge triggered

2. Notation du TP

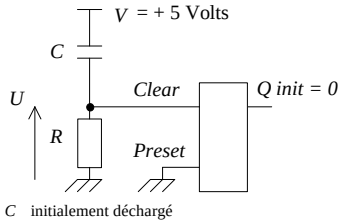
Faire examiner par le professeur en fin de séance, les différentes parties du TP.

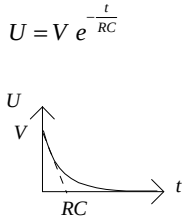
Rappel : Initialisation d'une bascule (exemple d'une bascule JK)

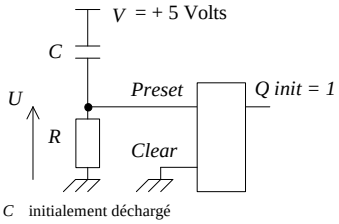
Les entrées asynchrones a de *mise à 0* et *mise à 1* généralement actives à l'état bas (donc notées $\overline{a}$) sont telles que lorsque *mise à 0* par ex. est activée, *Q* est placé à l'état 0 quelles que soient les entrées d'horloge et de données *J, K*. Ce sont des commandes d'effacement et d'initialisation (appelées aussi *Clear* et *Preset* ou encore *Reset* et *Set*) qui peuvent être activées pour fixer l'état initial de la sortie et qui doivent ensuite être inactivées pour permettre le fonctionnement normal de la bascule.
Un simple circuit *RC* connecté à l'entrée *Preset* par ex. pour fixer l'état initial *Q* peut être utilisé :

Cas d'entrées asynchrones actives à l'état haut :

A l'aide d'un circuit RC, on active l'entrée asynchrone un court instant au démarrage par un niveau haut, puis on ramène le signal asynchrone d'initialisation à 0. Appliqué à l'entrée *Clear*, ceci initialise *Q* à 0, mais appliqué à l'entrée *Preset*, ceci initialise *Q* à 1. A chaque fois l'entrée asynchrone non utilisée est placé à l'état inactif, soit l'état 0. (Le maintien prolongé d'une entrée asynchrone à son niveau actif fixe *Q* constant : à 0 si *Clear* est actif, à 1 si *Preset* l'est).

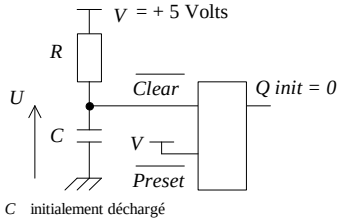


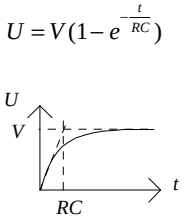


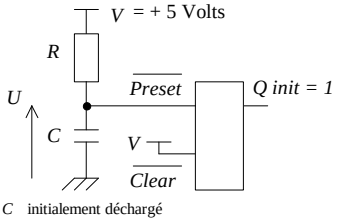


Cas d'entrées asynchrones actives à l'état bas :

A l'aide d'un circuit RC, on active l'entrée asynchrone un court instant au démarrage par un niveau bas, puis on ramène le signal asynchrone d'initialisation à 1. Appliqué à l'entrée $\overline{Clear}$, ceci initialise *Q* à 0, mais appliqué à l'entrée $\overline{Preset}$, ceci initialise *Q* à 1. A chaque fois l'entrée asynchrone non utilisée est placé à l'état inactif, soit l'état 1. (Le maintien prolongé d'une entrée asynchrone à son niveau actif fixe *Q* constant : à 0 si $\overline{Clear}$ est actif, à 1 si $\overline{Preset}$ l'est).







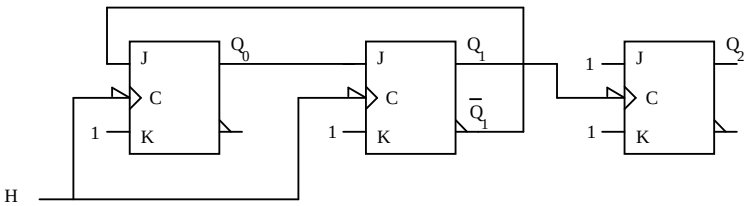
Simulation (& Câblage) :

Pour des raisons de compatibilité, n'utiliser que des circuits de la même famille (famille CMOS 4000 à ne pas mélanger avec la famille TTL 74xxx).

3. Compteurs

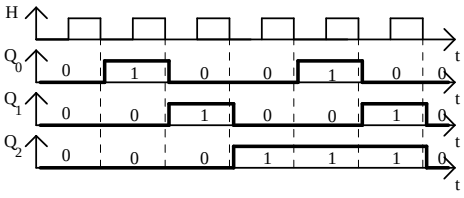
Etude théorique

Donner la succession des états du compteur suivant (automate des états), avec la notation $Q_2Q_1Q_0$, celui-ci étant supposé à l'état initial: $Q_2Q_1Q_0 = 000$:



(0)

Etude théorique - Corrigé

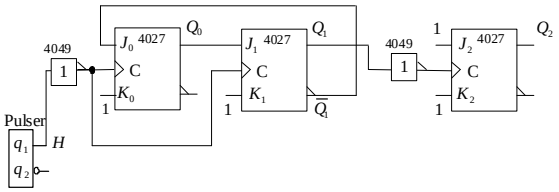


Compteur modulo 6

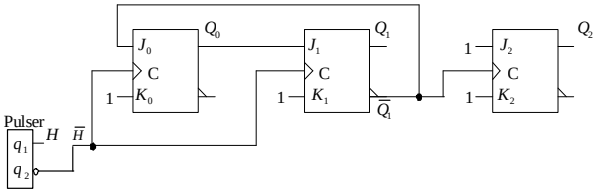
Etat initial : $Q_2Q_1Q_0 =$ **000**
Après la 1ère impulsion de H : $Q_2Q_1Q_0 =$ **001**
Après la 2nde impulsion de H : $Q_2Q_1Q_0 =$ **010**
Après la 3ème impulsion de H : $Q_2Q_1Q_0 =$ **100**
Après la 4ème impulsion de H : $Q_2Q_1Q_0 =$ **101**
Après la 5ème impulsion de H : $Q_2Q_1Q_0 =$ **110**
Après la 6ème impulsion de H : $Q_2Q_1Q_0 =$ **000** **... et le cycle recommence**

Etude expérimentale

- *Câblage :* Réaliser le câblage et cadencer à l'aide du mini-interrupteur pour horloge.
- *Simulation :* Effectuer la simulation (*Circuit Maker*) (prendre un *pulser* pour horloge) en effectuant le choix de montage (1) ou (2), équivalents au montage de base (0) :



(1) ≡



(2)

Le schéma initial (1) peut être remplacé par le schéma (2) identique, mettant en jeu des bascules JK > 0 edge triggered (4027).
Si le choix du schéma (1) est fait, l'utilisation de bascules JK > 0 edge triggered (4027) implique d'intercaler avant chaque entrée d'horloge des bascules un circuit inverseur 4049 (même famille CMOS que les bascules JK).

4. Synthèse de Compteur 2 bits

Etude théorique

Synthétiser un compteur 2 bits synchrone avec des bascules JK *positive edge triggered*, qui compte selon le cycle suivant : $Q_1Q_0 =$ 00 → 01 → 11 → 10 → 00 → …

Etude théorique - Corrigé

Rappel : Table de transitions d'une bascule JK :

Transition $Q_{n-1} \rightarrow Q_n$	***J***	***K***
0 → 0	0	X
0 → 1	1	X
1 → 1	X	0
1 → 0	X	1

. Table de Karnaugh établissant les entrées J_0K_0 de la bascule de sortie Q_0 :

J_0K_0 / Q_1 \ Q_0	0	1
0	1X	X0
1	0X	X1

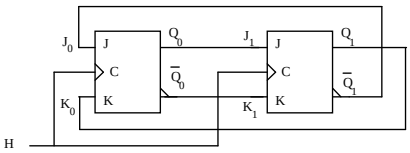
$$\begin{cases} J_0 = \overline{Q}_1 \\ K_0 = Q_1 \end{cases}$$

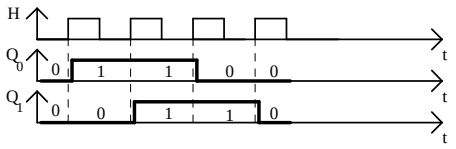
. Table de Karnaugh établissant les entrées J_1K_1 de la bascule de sortie Q_1 :

J_1K_1 / Q_1 \ Q_0	0	1
0	0X	1X
1	X1	X0

$$\begin{cases} J_1 = Q_0 \\ K_1 = \overline{Q}_0 \end{cases}$$

. Schéma de câblage :

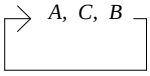




Etude expérimentale

- *Câblage :* Réaliser le câblage et cadencer à l'aide du mini-interrupteur pour horloge.
- *Simulation :*Vérifier par la simulation (*Circuit Maker*) (prendre un *pulser* pour horloge).

5. Séquenceur Train électrique (facultatif)

a) Donner l'automate des états correspondant à la trajectoire 

b) Effectuer la synthèse avec des bascules *JK* synchrones positive *edge triggered*, permettant de commander les aiguilleurs *p* et *q*.
c) Vérifier par la simulation.

Rangement du poste de travail
Examen des différentes parties du TP et rangement (0 pour tout le TP sinon).

5. VHDL

Introduction

VHDL est l'abréviation de « Very high speed integrated circuits Hardware Description Langage ». L'ambition des concepteurs du langage est de fournir un outil de description homogène des circuits, qui permette de créer des modèles de simulation et de « compiler » le silicium (c'est-à-dire synthétiser une fonction électronique dans un circuit programmable) à partir d'un programme unique.
Initialement réservé au monde des circuits numériques, VHDL est en passe d'être d'une part étendu aux circuits analogiques, et d'autre part rattaché à un langage de plus haut niveau : SystemC.

Deux des intérêts majeurs du langage sont :

- Des *niveaux de description* très divers : VHDL permet de représenter le fonctionnement d'une application tant du point de vue système que du point de vue circuit, en descendant jusqu'aux opérateurs les plus élémentaires. A chaque niveau, la description *(architecture)* peut être **structurelle** (portrait des interconnexions entre des sous fonctions), **flot de données** (portrait des signaux d'entrées/sorties) ou **comportementale** (langage évolué).

- Son aspect « *non propriétaire* » : le développement des circuits logiques a conduit chaque fabriquant de composants programmables (FPGAs : Field Programmable Gate Array ...) à développer son propre langage de description. VHDL est en passe de devenir le langage commun à de nombreux systèmes de CAO, indépendants ou liés à des producteurs de circuits, des (relativement) simples outils d'aide à la programmation des PALs aux ASICs, en passant par les FPGAs.

Héritier d'ADA, VHDL est un gros langage.

Historique

Au cours des 15 dernières années, les méthodes de conception des fonctions numériques ont subi une évolution importante. Dans les années 70, la majorité des applications de la logique câblée étaient construites autour de circuits intégrés standard, souvent pris dans la famille TTL. Au début des années 80 apparurent, parallèlement, les premiers circuits programmables par l'utilisateur (PALs : Programmable Array Logic) du côté des circuits simples, et les circuits intégrés spécifiques (ASICs : Application Specific Integrated Circuit) pour les fonctions complexes fabriquées en grande série. La complexité de ces derniers a nécessité la création d'outils logiciels de haut niveau qui sont à la description structurelle (schémas au niveau des portes élémentaires) ce que les langages évolués sont au langage machine dans le domaine de la programmation. Les premières générations de circuits programmables étaient conçues au moyen de simples programmes de traduction d'équations logiques en table de fusibles. A l'heure actuelle, l'écart de complexité entre circuits programmables et ASICs s'est restreint : on trouve une gamme continue de circuits qui vont des héritiers des premiers PALs, équivalents de quelques centaines de portes, à des FPGAs (Field Programmable Gate Array) ou des LCAs (Logic Cell Array) de quelques dizaines de milliers de portes équivalentes. Les outils d'aide à la conception se sont unifiés : un même langage, VHDL par exemple, peut être employé quels que soient les circuits utilisés, des PALs aux ASICs.

La démarche de conception d'un circuit est alors simple : on spécifie en langage VHDL les fonctionnalités du circuit à réaliser en termes d'entrée/sortie. Le compilateur VHDL engendre un fichier JEDEC (.JED) autorisant une simulation (visualisation des signaux d'entrée/sortie) en fonction du composant programmable choisi pour matérialiser le circuit désiré. Enfin, Il ne reste plus qu'à programmer ledit composant à l'aide d'une carte spécialisée fournie par le constructeur de composants programmables (FPGAs ...). Les principaux constructeurs : XILINX, ACTEL, CYPRESS, MENTOR GRAPHICS, CADENCE, VIEW LOGIC, ALTERA ... fournissent leur propre compilateur VHDL en grande partie portable (Warp de CYPRESS, Max+Plus d'ALTERA ...).

Méthodes de synthèse

Le remplacement, dans la plupart des applications, des fonctions standard complexes par des circuits programmables, s'accompagne d'un changement dans les méthodes de conception :

- on constate un « retour aux sources » : le concepteur d'une application élabore sa solution en descendant au niveau des bascules élémentaires, au même titre que l'architecte d'un circuit intégré.

- l'utilisation systématique d'outils de conception assistée par ordinateur (CAO), sans lesquels la tâche serait irréalisable, rend caducs les fastidieux calculs de minimisation d'équations logiques. Le concepteur peut se consacrer entièrement aux choix d'architecture qui sont eux, essentiels.

- la complexité des fonctions réalisables dans un seul circuit pose le problème du test. Les outils traditionnels de test de cartes imprimées, du simple oscilloscope à la « planche à clous » en passant par l'analyseur d'états logiques ne sont plus d'un grand secours, dès lors que la grande majorité des équipotentielles sont inaccessibles de l'extérieur. Là encore, la CAO joue un rôle essentiel. Encore faut-il que les solutions choisies soient analysables de façon sûre. Cela interdit formellement certaines astuces, parfois rencontrées dans des schémas traditionnels de logique câblée, comme des commandes asynchrones utilisées autrement que pour une initialisation lors de la mise sous tension, par exemple.

- les langages de haut niveau comme VHDL privilégient une approche globale des solutions. Dès lors que l'architecture générale d'une application est arrêtée, que les algorithmes qui décrivent le fonctionnement de chaque partie sont élaborés, le reste du travail de synthèse est extrêmement simple et rapide.

1. Principes généraux

1.0. Description descendante : le « top down design »

Une application un tant soit peu complexe est découpée en sous-ensembles qui échangent des informations suivant un protocole bien défini. Chaque sous-ensemble est, à son tour, subdivisé, et ainsi de suite jusqu'aux opérateurs élémentaires.

Un système est construit comme une hiérarchie d'objets, les détails de réalisation se précisant au fur et à mesure que l'on descend dans cette hiérarchie. A un niveau donné de la hiérarchie, les détails de fonctionnement interne des niveaux inférieurs sont *invisibles*. C'est le principe même de la programmation structurée.
Plusieurs réalisations d'une même fonction pourront être envisagées, sans qu'il soit nécessaire de remettre en cause la conception des niveaux supérieurs; plusieurs personnes pourront collaborer à un même projet, sans que chacun ait à connaître tous les détails de l'ensemble.

La conception descendante consiste à définir le système en partant du sommet de la hiérarchie, en allant du général au particulier. VHDL permet, par exemple, de tester la validité de la conception d'ensemble, avant que les détails des sous fonctions ne soient complètement définis. A titre d'exemple, l'architecture générale d'un processeur peut être évaluée sans que le mode de réalisation de ses registres internes ne soit connu, le fonctionnement des registres en question sera alors décrit au niveau comportemental.

1.1. Simulation et/ou synthèse

VHDL a été, initialement, connu comme un langage de simulation, il est fortement marqué par cet héritage très informatique, ce qui est parfois un peu déroutant pour l'électronicien, proche du matériel, qui n'est pas toujours un spécialiste des langages de programmation. Citons quelques exemples :

- Contrairement à C ou PASCAL, VHDL est un langage qui comprend le « parallélisme », c'est à dire que des blocs d'instructions peuvent être exécutés simultanément, par opposition à séquentiellement comme dans un langage procédural traditionnel. Autant ce parallélisme est fondamental pour comprendre le fonctionnement d'un simulateur logique, et peut être déroutant pour un programmeur habitué au déroulement séquentiel des instructions qu'il écrit, autant il est évident que le fonctionnement d'un circuit ne dépend pas de 1'ordre dans lequel ont été établies les connexions. L'utilisateur de VHDL gagnera beaucoup en ne se laissant pas enfermer dans l'aspect langage de programmation, en se souvenant qu'il est en train de créer un vrai circuit. Les parties séquentielles du langage, car il y en a, doivent, dans ce contexte, être comprises soit comme une facilité offerte dans l'écriture de certaines fonctions, soit comme *le* moyen de décrire des opérateurs fondamentalement séquentiels : les opérateurs synchrones.

- La modélisation correcte d'un système suppose de prendre en compte, au niveau du simulateur, les imperfections du monde réel. VHDL offre donc la possibilité de spécifier des retards, de préciser ce qui se passe lors d'un conflit de bus, etc. Pour simuler toutes ces vicissitudes, le langage offre toute une gamme d'outils : signaux qui prennent une valeur inconnue, messages d'erreurs quand un « circuit » détecte une violation de *set up time*, changements d'états retardés pour simuler les temps de propagation. Toutes les constructions associées de ce type ne sont évidemment *pas* synthétisables! La difficulté principale est que, suivant les compilateurs, la frontière entre ce qui est synthétisable et ce qui ne l'est pas n'est pas toujours la même, même pour des compilateurs qui respectent la norme IEEE-1076. Avant d'utiliser un outil de synthèse, le concepteur de circuit a tout à gagner à lire très attentivement la présentation du sous-ensemble de VHDL accepté par cet outil.

- Trois classes de données existent en VHDL : les constantes, les variables (affectation par le symbole :=) et les signaux (affectation par le symbole <=). La nature des signaux ne présente aucune ambiguïté, ce sont des objets qui véhiculent une information logique tant du point de vue simulation que dans la réalité. Les signaux qui ont échappé aux simplifications logiques, apportées par l'optimiseur toujours présent, sont des vraies équipotentielles du schéma final. Les variables sont destinées, comme dans tout langage, à stocker temporairement des valeurs, dans l'optique d'une utilisation future, sans chercher à représenter la réalité. Certains compilateurs considèrent que les variables n'ont aucune existence réelle, au niveau du circuit, qu'elles ne sont que des outils de description fonctionnelle. D'autres transforment, éventuellement (cela dépend de l'optimiseur), les variables en cellules mémoires ...

Il est clair que VHDL, les outils de synthèse et d'optimisation ne peuvent pas transformer un mauvais concepteur en un bon. Ce sont de simples outils supplémentaires qui peuvent aider un ingénieur à réaliser plus rapidement et plus efficacement un matériel quand ils sont utilisés correctement.

Il est toujours nécessaire de comprendre les détails physiques de la façon dont est implémentée une réalisation. L'ingénieur doit « regarder par dessus 1'épaule » de l'outil pour s'assurer que le résultat est conforme à ses exigences et à sa philosophie, et que le résultat est obtenu en un temps raisonnable.

1.2. Exemples

Des tautologies

Les exemples de code source VHDL ci-dessous ne nous apprennent rien sur les propriétés des opérateurs concernés, ils nous montrent l'aspect d'un programme VHDL et nous rappellent que les opérations *NON*, *ET* et *OU* sont définies sur les objets de type bit (appartenant à la classe signal) comme sur ceux de type boolean, avec une convention logique positive (1≡TRUE, 0≡FALSE).

```
-- inverseur  (ceci est un commentaire)
entity inverseur is
port    (e : in  bit ;      -- les entrees
         s : out bit) ;     -- les sorties
end inverseur;
architecture dataflowlowlevel of inverseur is
begin
        s <= not e ;
end dataflowlowlevel;
```

De même :

```
-- operateur et
entity et is
port    (e1, e2 : in  bit ;
         s : out bit) ;
end et ;
architecture dataflowlowlevel of et is
begin
        s <= e1 and e2 ;
end dataflowlowlevel;
```

Ou encore :

```
-- operateur ou
entity ou is
port    (e1, e2 : in  bit ;
         s : out bit) ;
end ou ;
architecture dataflowlowlevel of ou is
begin
        s <= e1 or e2 ;
end dataflowlowlevel;
```

On notera la structure générale d'un programme et le symbole « d'affectation » particulier aux objets de nature signal (*s*, *e*, *el*, *e2*). La déclaration entity correspond au prototype d'une fonction en langage C, elle décrit l'interaction entre l'opérateur et le monde environnant. La partie architecture du programme correspond à la description interne de l'opérateur, elle décrit donc son fonctionnement. L'architecture peut être de type structurel, flot de données ou comportemental. Plusieurs architectures peuvent décrire une même entité. Les mots clés du langage sont notés en non *italique* ou en MAJUSCULES, c'est une habitude de certains, pas une obligation.

Des affectations conditionnelles

Dans les programmes qui suivent on voit apparaître la notion de « haut niveau » du langage. Des expressions purement booléennes sont utilisées pour décrire le fonctionnement d'un circuit. Ici elles traduisent strictement les tables de vérité, mais permettent évidemment des constructions beaucoup plus élaborées.

```
-- inverseur
entity inverseur is
port     (e : in  bit ;
          s : out bit) ;
end inverseur ;
architecture dataflowhighlevel of inverseur is
begin
         s <= '1'  when  (e = '0')  else  '0' ;
end dataflowhighlevel;
```

De même :

```
-- operateur et
entity et is
port     (e1, e2 : in  bit ;
          s : out bit) ;
end et ;
architecture dataflowhighlevel of et is
begin
         s <= '0'  when  (e1 = '0'  or  e2 = '0')  else  '1' ;
end dataflowhighlevel;
```

ou encore :

```
-- operateur ou
entity ou is
port     (e1, e2 : in  bit ;
          s : out bit) ;
end ou ;
architecture dataflowhighlevel of ou is
begin
         s <= '0'  when  (e1 = '0'  and  e2 = '0')  else  '1' ;
end dataflowhighlevel;
```

Des exemples de modèles comportementaux

Terminons cette première découverte de VHDL par 2 descriptions d'architecture purement **comportementales** des opérateurs *ET* et *OU* :

```
entity et is       -- operateur et
port     (e1, e2 : in  bit ;
          s : out bit) ;
end et ;
architecture behaviour of et is
begin
        process (el, e2)
        begin
                if (e1 = '0'  or  e2 = '0') then
                        s <= '0' ;
                else
                        s <= '1' ;
                end if ;
        end process ;
end behaviour;
```

ou encore :

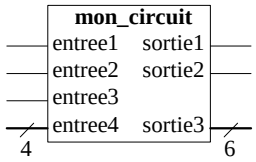
```
entity ou is  -- operateur ou
port  (e :  in  bit_vector (0 to 1);
         s : out bit );
end ou ;
architecture behaviour of ou is
begin
        process ( e )
        begin
                case e is
                        when ("00")  =>
                                s <= '0' ;
                        when others  =>
                                s <= '1' ;
                end case ;
        end process ;
end behaviour;
```

Le rôle de l'instruction process est de décrire explicitement un processus défini par un algorithme *séquentiel* (≡ *non parallèle*, contrairement aux descriptions *structurelle* et *flot de données*).

1.3. L'extérieur de la boîte noire : une « ENTITE »

Dans une construction hiérarchique, les niveaux supérieurs n'ont pas à connaître les détails des niveaux inférieurs. Une fonction logique sera vue, dans cette optique, comme un assemblage de « boîtes noires » dont, syntaxiquement parlant, seules les modes d'accès sont nécessaires à l'utilisateur.

La construction qui décrit l'extérieur d'une fonction est l'entité (entity). La déclaration correspondante lui donne un nom et précise la liste des signaux d'entrée et de sortie (équivalent en langage C du prototypage) :




```
entity mon_circuit is port (
        entree1 : in bit ;
        entree2, entree3 : in bit ;
        entree4 : in bit_vector ( 0 to 3 ) ;
        sortie1, sortie2 : out bit ;
        sortie3 : out bit_vector ( 0 to 5 ) )
end mon_circuit ;
```

Dans l'exemple qui précède, les noms des objets, qui dépendent du choix de l'utilisateur, sont écrits en *italique*, les autres mots sont des mots-clés du langage (ces derniers peuvent indifféremment être écrits en minuscules ou en MAJUSCULES).
Les choix possibles pour le sens de transfert sont : in, out, inout et buffer (une sortie qui peut être « lue » par l'intérieur du circuit).
Les choix possibles pour les types de données échangées sont les mêmes que pour les signaux.

1.4. Le fonctionnement interne : une « ARCHITECTURE »

L'architecture est la matérialisation de la fonction définie dans l'entité. Elle décrit le fonctionnement interne d'un circuit auquel est attaché une entité. Ce fonctionnement peut être décrit de différentes façons :

Description structurelle

Le circuit est vu comme un assemblage de composants de niveau inférieur, c'est une description « schématique ». Souvent ce mode de description est utilisé au niveau le plus élevé de la hiérarchie, chaque composant étant lui-même défini par un programme VHDL (entité et architecture).

Description comportementale

Le comportement matériel du circuit est décrit par un algorithme, indépendamment de la façon dont il est réalisé au niveau structurel.

Description par un flot de données

Le fonctionnement du circuit est décrit par un flot de données qui vont des entrées vers les sorties, en subissant, étape par étape, des transformations élémentaires successives. Ce mode de description permet de reproduire l'architecture logique, en couches successives, des opérateurs combinatoires.

Flot de données et représentation comportementale sont très voisines, dans les deux cas le concepteur peut faire appel à des instructions de haut niveau. La première méthode utilise un grand nombre de signaux internes qui conduisent au résultat par des transformations de proche en proche. La seconde utilise des blocs de programme (les processus explicites), qui manipulent de nombreux signaux avec des algorithmes séquentiels.

La syntaxe générale d'une architecture comporte une partie de déclaration et un corps de programme :

architecture *exemple* of *mon_circuit* is
partie déclarative optionnelle : types, constantes, signaux locaux, composants.
begin
corps de l'architecture.
suite d'instructions parallèles :
affectations de signaux ;
processus explicites ; -- mais l'intérieur d'un processus est séquentiel
blocs ;
instanciation (i.e. importation dans un schéma) de composants (appel de sous-programmes).
end *exemple ;*

1.5. Des algorithmes séquentiels décrivent un câblage parallèle : les « PROCESSUS »

« Un processus est une instruction *concurrente* (deux instructions concurrentes sont simultanées) qui définit un comportement qui doit avoir lieu quand ce processus devient actif. Le comportement est spécifié par une suite d'instructions *séquentielles* exécutées dans le processus. »

Qu'est-ce que cela signifie ? Trois choses :

1. Les différentes parties d'une réalisation interagissent simultanément, peu importe l'ordre dans lequel un câbleur soude ses composants, le résultat sera le même. Le langage doit donc comporter une contrainte de « parallélisme » entre ses instructions. Cela implique des différences notables avec un langage procédural comme le langage C.

En langage VHDL :

a <= *b* ;
c <= *a* + *d* ;
et
c <= *a* + *d* ;
a <= *b* ;

représentent la même chose, ce qui est notablement différent de ce qui se passerait en langage C pour :

a = *b* ;
c = *a* + *d*;
et
c = *a* + *d* ;
a = *b* ;

Les affectations de signaux, à *l'extérieur* d'un processus explicite, sont traitées comme des processus tellement élémentaires qu'il est inutile de les déclarer comme tels. Ces affectations sont traitées en parallèle, de la même façon que plusieurs processus indépendants.

2. L'algorithmique fait grand usage d'instructions séquentielles pour décrire le monde. VHDL offre cette facilité à *l'intérieur* d'un processus explicitement déclaré. Dans le corps d'un processus il sera possible d'utiliser des variables, des boucles, des conditions, dont le sens est le même que dans les langages séquentiels. Même les affectations entre signaux sont des instructions séquentielles quand elles apparaissent à l'intérieur d'un processus. Seul sera visible de l'extérieur le résultat final obtenu à la fin du processus.

3. Les opérateurs séquentiels, surtout synchrones, mais pas exclusivement eux, comportent « naturellement » la notion de mémoire, qui est le fondement de l'algorithmique traditionnelle. Les processus sont *la* représentation privilégiée de ces opérateurs (ce n'est pas la seule, les descriptions structurelle et flot de données, plus proches du câblage du circuit, permettent de décrire tous les opérateurs séquentiels avec des opérateurs combinatoires élémentaires. Pour les circuits qui comportent des bascules comme éléments primitifs, connus de l'outil de synthèse, les deux seules façons d'utiliser ces bascules sont les processus et leur instanciation comme composants dans une description structurelle).

Mais attention, *la réciproque n'est pas vraie*, il est parfaitement possible de décrire un opérateur purement combinatoire par un processus, le programmeur utilise alors de cet objet la seule facilité d'écriture de l'algorithme.
Outre les simples affectations de signaux, qui sont en elles mêmes des processus implicites à part entière, la description d'un processus obéit à la syntaxe suivante *(en particulier, la structure séquentielle « if … then » est à inclure obligatoirement dant un processus) :*

Processus : syntaxe générale avec liste de sensibilité (liste de paramètres d'activation du processus)

[*étiquette* :] process [*(liste de sensibilité)*]
partie déclarative optionnelle : variables notamment
begin
corps du processus
instructions séquentielles
end process [*étiquette*] ;

Les éléments mis entre crochets sont optionnels, ils peuvent être omis sans qu'il y ait d'erreur de syntaxe. La liste de sensibilité est la liste des signaux qui déclenchent, par le changement de valeur de l'un quelconque d'entre eux, l'activité du processus (*paramètres* ou *arguments* du processus). Cette liste peut être remplacée par une instruction « wait » dans le corps du processus : la syntaxe du processus devient alors la suivante :

Processus : syntaxe générale sans liste de sensibilité mais avec l'instruction « wait »

[*étiquette* :] process
partie déclarative optionnelle : variables notamment
begin
wait [on *liste_de_signaux*] [until *condition*] ;
instructions séquentielles
end process [*étiquette*] ;

L'instruction wait

Cette instruction indique au processus que son déroulement doit être suspendu dans l'attente d'un événement sur un signal (un signal change de valeur), et tant qu'une condition n'est pas réalisée.

Sa syntaxe générale est :

wait [on *liste_de_signaux*] [until *condition*] ;

On peut spécifier un temps d'attente maximum (wait … for *temps*), mais cette clause n'est pas synthétisable.

La liste des signaux dont l'instruction attend le changement de valeur joue exactement le même rôle que la liste de sensibilité du processus, mais *l'instruction* wait *ne peut pas être utilisée en même temps qu'une liste de sensibilité.* La tendance, pour les évolutions futures du langage, semble être à la suppression des listes de sensibilités, pour n'utiliser que les instructions d'attente.

Description d'un opérateur séquentiel

La représentation des horloges : pour représenter les opérateurs synchrones de façon comportementale, il *faut* introduire l'horloge dans la liste de sensibilité, ou insérer dans le code du processus une instruction « wait » explicite. Rappelons qu'il est interdit d'utiliser à la fois une liste de sensibilité et une instruction wait. Quand on modélise un opérateur qui comporte à la fois des commandes synchrones et des commandes asynchrones, il *faut,* avec certains compilateurs, mettre ces commandes dans la liste de sensibilité.

Exemple :

```
architecture fsm of jk_raz is
signal etat : bit ;
begin                   -- la déclaration de q a eu lieu précédemment dans l'entité
q <= etat ;             -- l'état va être défini ci-dessous (parallélisme)
                        -- l'instruction « q <= etat » et le processus sont 2 instructions concurrentes
                        -- (parallèles) alors que l'intérieur d'un processus est séquentiel
process (clock, raz)    -- deux signaux d'activation externe
begin
if (raz = '1') then     -- raz asynchrone
        etat <= '0';
elseif (clock = 'l' and clock 'event) then      -- front montant d'horloge
case etat is
        when '0' =>
                if (j = '1' ) then
                        etat <= '1';
                end if ;
        when '1' =>
                if (k = '1' ) then
                        etat <= '0';
                end if ;
end case ;
end if ;
end process ;
end fsm ;
```

Dans l'exemple précédent, la priorité de la mise à zéro asynchrone sur le fonctionnement synchrone normal de la bascule JK, apparaît par l'ordre des instructions de la structure if … elseif. Le processus est utilisé là à la fois pour modéliser un opérateur essentiellement séquentiel, la bascule, et pour faciliter la description de l'effet de ses commandes par un algorithme séquentiel. Pour modéliser un comportement purement synchrone on peut indifféremment utiliser la liste de sensibilité ou une instruction wait :

```
architecture fsm_liste of jk_simple is
signal etat : bit ;
begin
q <= etat ;             -- la déclaration de q a eu lieu précédemment dans l'entité
process (clock)         -- un seul signal d'activation
begin
if (clock = 'l' and clock 'event) then
case etat is
        when '0' =>
                if (j = 'l' ) then
                        etat <= 'l' ;
                end if;
        when 'l' =>
                if (k = 'l' ) then
                        etat <= '0' ;
                end if;
end case ;
end if ;
end process ;
end fsm_liste ;
```

Ou, de façon strictement équivalente, en utilisant une instruction « wait » :

```
architecture fsm_wait of jk_simple is
signal etat : bit ;
begin
q <= etat ;
process         -- pas de liste de sensibilité
begin
wait until (clock = '1') ;
case etat is
        when '0' =>
                if (j = '1' ) then
                        etat <= '1' ;
                end if;
        when '1' =>
                if (k = '1' ) then
                        etat <= '0' ;
                end if ;
end case ;
end process ;
end fsm_wait ;
```

Description par un processus d'un opérateur combinatoire ou asynchrone

Un processus permet de décrire un opérateur purement combinatoire ou un opérateur séquentiel asynchrone, en utilisant une démarche algorithmique.
Dans ces deux cas la liste de sensibilité, ou l'instruction wait équivalente, est obligatoire; le caractère combinatoire ou séquentiel de l'opérateur réalisé va dépendre du code interne au processus. On considère un signal qui fait l'objet d'une affectation dans le corps d'un processus :
- si au bout de l'exécution du processus, pour *toutes* les combinaisons possibles des valeurs de la liste de sensibilité, la valeur de ce signal, objet d'une affectation, est connue, l'opérateur correspondant est combinatoire.
- si certaines des combinaisons précédentes de la liste de sensibilité conduisent à une indétermination concernant la valeur du signal examiné, objet d'une affectation, ce signal est associé à une cellule mémoire (opérateur séquentiel).

Précisons ce point par un exemple :

```
entity comb_seq is
port (
        el, e2 : in bit ;
        s_et, s_latch, s_edge : out bit
        ) ;
end comb_seq ;

architecture exproc of comb_seq is
begin

-- porte logique et
et : process (e1, e2)                   -- ou encore    process (e1)
begin
if ( e1 = 'l' ) then
        s_et <= e2 ;
else
        s_et <= '0' ;                   -- équivalent à s_et <= e1 and e2 ;
end if ;
end process ;                           -- opérateur combinatoire

-- bascule D latch, e1 est la commande
latch : process (e1,e2)                 -- ou encore    process (e1)
begin
if ( e1 = '1' ) then
        s_latch <= e2 ;
end if;                         -- si e1 = '0' la valeur de s_latch est « inconnue » (→ mémorisation).
end process ;                   -- opérateur séquentiel

-- bascule D edge, e1 est l'horloge
edge : process (e1)                     -- ou encore    process (e1,e2)
begin
if ( e1 'event and e1 = '1') then       -- e1 agit sur un front montant.
        s_edge <= e2 ;
end if ;
end process ;                   -- en dehors d'un front montant : mémorisation de l'état précédent
end exproc ;                            -- opérateur séquentiel
```

Dans l'exemple qui précède, le premier processus est combinatoire, le signal *s_et* a une valeur connue à la fin du processus, quelles que soient les valeurs des entrées *e1* et *e2*. Dans le deuxième processus, l'instruction « if » ne nous renseigne pas sur la valeur du signal *s_latch* quand *e1* = '0'. Cette méconnaissance est interprétée, par le compilateur VHDL, comme un maintien de la valeur précédente, d'où la génération d'une cellule mémoire dont la commande de mémorisation, *e1*, est active sur un niveau. Le troisième processus conduit également, et pour le même type de raison, à la synthèse d'une cellule mémoire pour le signal *s_edge*. Mais la commande de mémorisation est, cette fois, active sur un front, explicitement mentionné dans la condition de l'instruction « if » : *e1* 'event. La façon dont est traitée la commande de mémorisation *e1* dépend donc de l'écriture du test : niveau ou front

2. Eléments du langage

2.1. Les données appartiennent à une classe et ont un type

VHDL, héritier d'ADA, est un *langage fortement typé.* Toutes les données ont un type qui doit être déclaré avant l'utilisation (sauf, et c'est bien pratique, les variables entières des boucles « for ») et aucune conversion de type automatique (une souplesse et un piège immense du langage C, par exemple) n'est effectuée. Pour passer du type entier au type bit_vector, par exemple, il faut faire appel à une fonction de conversion.
Une donnée appartient à une classe qui définit, avec son type, son comportement. Des données de deux classes différentes, mais de même type, peuvent échanger des informations directement : on peut affecter la valeur d'une variable à un signal, par exemple (nous verrons ci-dessous que variables et signaux sont deux classes différentes).
La portée des noms *est,* en général, locale. Un nom déclaré à l'intérieur d'une architecture, par exemple, n'est connu que dans celle-ci. Des objets globaux sont possibles, on peut notamment définir des constantes, comme zero ou one, extérieures aux unités de programmes que constituent les couples entité-architecture. A l'intérieur d'une architecture, les objets déclarés dans un bloc (délimité par les mots-clés begin et end) sont visibles des blocs plus internes uniquement.
Les objets déclarés dans une entité sont connus de toutes les architectures qui s'y rapportent.

Les classes : signaux, variables et constantes

Signaux

Les signaux représentent les données physiques échangées entre des blocs logiques d'un circuit. Chacun d'entre eux sera matérialisé dans le schéma final par une *équipotentielle* et, éventuellement, une *cellule mémoire* qui conserve la valeur de l'équipotentielle entre deux commandes de changement. Les « ports » d'entrée et de sortie, attachés à une entité, par exemple, sont une variété de signaux qui permettent l'échange d'informations entre différentes fonctions. Leur utilisation est similaire à celle des arguments d'une procédure en langage PASCAL, le sens de transfert de l'information doit être précisé.

Syntaxe de déclaration (se place dans la partie déclarative d'une architecture - ou d'un paquetage, voir plus loin) :

signal *noml* , *nom2* : *type ;*

Affectation d'une valeur (se place dans le corps d'une architecture ou d'un processus) :

nom <= valeur_compatible_avec_le_type ;

La valeur affectée peut être le résultat d'une expression, simple ou conditionnelle (when), ou la valeur renvoyée par l'appel d'une fonction.

A l'extérieur d'un processus toutes les affectations de signaux sont concurrentes, *c'est donc une erreur* (sémantique, pas syntaxique) *d'affecter plus d'une fois une valeur à un signal.* L'affectation d'une valeur à un signal traduit, en fait, la connexion de la sortie d'un opérateur à l'équipotentielle correspondante. Il s'agit là d'une opération permanente, une soudure sur une carte, par exemple, qu'il est hors de question de modifier ailleurs dans le programme. Si un signal est l'objet d'affectations multiples, ce qui revient à mettre en parallèle plusieurs sorties d'opérateurs (trois-états ou collecteurs ouverts, par exemple), il faut adjoindre à ce signal, pour les besoins de la simulation, une fonction de résolution qui permet de résoudre le conflit (dans les applications de synthèse les portes à sorties non standard sont généralement introduites dans une description structurelle).

Variables

Les variables sont des objets qui servent à stocker un résultat intermédiaire pour faciliter la construction d'un *algorithme séquentiel*. Elles ne peuvent être utilisées *que dans les processus*, *les procédures* ou *les fonctions* et dans les boucles « generate » qui servent à créer des schémas répétitifs.

Syntaxe de déclaration (se place dans la partie déclarative d'un processus, d'une procédure ou d'une fonction) :

variable *noml*, *nom2* : *type* [:= expression] ;

L'expression facultative qui apparaît dans la déclaration précédente permet de donner à une variable une valeur initiale choisie par l'utilisateur. A défaut de cette expression *le compilateur, qui initialise toujours les variables* (le programmeur ne doit, notamment, pas s'attendre à retrouver les variables d'un processus dans l'état où il les avait laissées lors d'une activation précédente de ce processus), utilise une valeur par défaut qui dépend du type déclaré.

Affectation d'une valeur :

nom := *valeur_compatible_avec_le_type* ;

La valeur affectée peut être le résultat d'une expression ou la valeur renvoyée par l'appel d'une fonction.
Les variables de VHDL jouent le rôle des variables automatiques des langages procéduraux, comme C ou PASCAL, elles ont une portée limitée au module de programme dans lequel elles ont été déclarées, et sont détruites à la sortie de ce module.
La différence entre variables et signaux est que les premières n'ont pas d'équivalent physique dans le schéma, contrairement aux seconds. Certains outils de synthèse ne respectent malheureusement pas cette distinction. On notera qu'il est possible d'affecter la valeur d'une variable à un signal, et inversement, pourvu que les types soient compatibles.

Constantes

Les constantes sont des objets dont la valeur est fixée une fois pour toute.

Exemples de valeurs constantes simples : '0', '1', "01101001", 2006, "azerty"
2006 est du type entier. "01101001" est du type string (chaîne de caractères). '0' est du type caractère.

On peut créer des constantes nommées : constant nom1 : type [:= valeur_constante] ;

On notera que les vecteurs de bits (bit_vector) sont traités comme des chaînes, on peut préciser une base différente de la base 2 pour ces constantes : X"3A007", O"237015" pour hexadécimal et octal.

De même, les valeurs entières peuvent être écrites dans une autre base que 1a base 10 :
16#ABCDEFO123#, 2#001011101# ou 2#0_0101_1101#, pour plus de lisibilité.

En général les nombres flottants ne sont pas acceptés par les outils de synthèse.

Des types adaptés à l'électronique numérique

VHDL connaît un nombre limité de types de base, qui reflètent le fonctionnement des systèmes numériques (pour l'instant, VHDL est en passe de devenir un langage de description des circuits analogiques), et offre à l'utilisateur de construire à partir de ces types génériques :

- des sous-types (sous-ensembles du type de base), obtenus en précisant un domaine de variation limité de l'objet considéré,
- des types composés, obtenus par la réunion de plusieurs types de base identiques (tableaux) ou de types différents (enregistrements).

En plus des types prédéfinis et de leurs dérivés, l'utilisateur a la possibilité de créer ses propres types sous forme de types énumérés.

Les entiers

VHDL manipule des valeurs entières qui correspondent à des mots de 32 bits, soit comprises entre -2147483648 et +2147483647.

Attention, sur les PCs qui sont des machines dont les entiers continuent à hésiter entre 16 et 32 bits, l'utilisateur peut rencontrer de désagréables surprises. Les nombres négatifs ne sont pas toujours acceptés dans la description des signaux physiques.

Déclaration :

signal *nom :* integer ;

ou

variable *nom :* integer ;

ou encore :

constant *nom :* integer ;

que l'on résume classiquement par :

signal | variable | constant *nom :* integer ; le symbole | signifiant « ou ».

On peut spécifier une plage de valeurs inférieure à celle obtenue par défaut, par exemple :

signal *etat* : integer range 0 to 1023 ; permet de créer un compteur 10 bits.

La même construction permet de créer un sous-type :

subtype *etat_l0* is integer range 0 to 1023 ;
signal *etatl* , *etat2* : *etat_l0* ;

Attention ! La restriction d'étendue de variation est utilisée pour générer le nombre de chiffres binaires nécessaires à la représentation de l'objet, l'arithmétique sous-jacente n'est (pour l'instant) pas traitée par les compilateurs. Cela veut dire que :

signal *chiffre* : integer range 0 to 9 ; permet de créer un objet codé sur quatre bits, mais :

chiffre <= *chiffre* + 1 ; ne crée *pas* un compteur décimal.

Pour ce faire il faut écrire explicitement (compteur décimal) :

```
if (chiffre < 9 ) then
        chiffre <= chiffre + 1 ;
else
        chiffre <= 0 ;
end if ;
```

La déclaration, au niveau le plus élevé d'une hiérarchie, de ports d'entrée ou de sortie comme nombres entiers pose un problème de contrôle par l'utilisateur, de l'assignation des broches physiques du circuit final aux chiffres binaires générés. Cette assignation sera faite automatiquement par l'outil de développement. Si ce non contrôle est gênant, il est possible de transformer un nombre entier en tableau de bits, via les fonctions de conversion de la librairie associée à un compilateur.

Les types énumérés

L'utilisateur peut créer ses propres types par simple énumération de constantes symboliques qui fixent toutes les valeurs possibles du type. Par exemple :

```
type drinkState is
(zero, five, ten, fifteen, twenty, twentyfive, owedime);
signal drinkStatus : drinkState;
```

Les bits

Il s'agit là, évidemment, du type de base le plus utilisé en électronique numérique. Un objet de type bit peut prendre deux valeurs : '0' et '1'. Il s'agit, en fait, d'un type énuméré prédéfini.

Déclaration :

signal | variable *nom :* bit ;

Ce qui précède s'applique aux descriptions synthétisables, pour les besoins de la simulation de nombreux compilateurs proposent un type bit plus étoffé, pouvant prendre, par exemple, les valeurs '0', '1', 'X' (X pour inconnu) et 'Z' (pour haute impédance). Ces types sont traduits, en synthèse, par des types bit ordinaires.
De même la gestion des portes trois-états, qui se fait par une description structurelle, nécessite une extension du type bit. Par exemple :

```
entity basc_tri_state is
port ( clk, oe : in bit;
        sort : inout x0lz );            -- type x0lz défini dans la librairie
end basc_tri_state ;
```

Toutes ces extensions ne sont, à priori, pas portables telles quelles. Mais les outils de développement VHDL sont fournis avec les sources (en VHDL) de toutes les extensions au langage de base, ce qui permet de porter d'un système à l'autre les librairies nécessaires.

Les booléens

Autre type énuméré, le type booléen peut prendre deux valeurs: "true" et "false". Il intervient essentiellement comme résultat d'expressions de comparaisons, dans des if, par exemple, ou dans les valeurs renvoyées par des fonctions.

Les tableaux

A partir de chaque type de base on peut créer des tableaux, collection d'objets du même type. L'un des plus utilisés est le type bit_vector, défini dans la librairie standard par :

```
subtype natural is integer range 0 to integer 'high ;
type bit_vector is array (natural range <>) of bit ;
```

Dans l'exemple qui précède, le nombre d'éléments n'est pas précisé dans le type, ce sera fait à l'utilisation. Par exemple :

```
signal etat : bit_vector (0 to 4) ;
```

définit un tableau de cinq éléments binaires nommé *état*. On aurait également pu définir directement un sous-type :

```
type cinq_bit is array (0 to 4) of bit ;
signal etat : cinq_bit ;
```

Le nombre de dimensions d'un tableau n'est pas limité, les indices peuvent être définis dans le sens croissant (2 to 6) ou décroissant (6 downto 2) avec des bornes quelconques (mais cohérentes avec le sens choisi).
On notera *qu'il faut* passer par une définition de type, ce qui n'est pas le cas en C ou en PASCAL. Une fois défini, un objet composé peut être manipulé collectivement par son nom :

```
signal etatl : bit_vector (0 to 4) ;
variable etat2 : bit_vector (0 to 4) ;
etat1 <= etat2 ;          -- parfaitement correct
```

Le compilateur contrôle que les dimensions des deux objets sont les mêmes. On remarquera, à partir de l'exemple précédent, que les classes des deux objets peuvent être différentes.
On peut, bien sûr, ne manipuler qu'une partie des éléments d'un tableau :

```
signal etat : bit_vector (0 to 4) ;
signal sous_etat : bit_vector (0 to 1) ;
signal flag : bit;
…
sous_etat <= etat ( 1 to 2 ) ;
flag <= etat ( 3 ) ;
```

Il est possible de fusionner deux tableaux (concaténation) pour affecter les valeurs correspondantes à un tableau plus grand :

```
signal etat : bit_vector (0 to 4) ;
signal sous_etat2 : bit_vector (0 to 1) ;
signal sous_etat3 : bit_vector (0 to 2) ;
…
etat <= sous_etat2 & sous_etat3 ;          -- concaténation.
```

Les enregistrements

Les enregistrements (*record*) définissent des collections d'objets de types, ou de sous types, différents. Ils correspondent aux structures du C ou aux enregistrements de PASCAL.

Définition d'un type :

```
type clock_time is record
hour : integer range 0 to 12 ;
minute, seconde : integer range 0 to 59 ;
end record;
```

Déclaration d'un objet de ce type :

variable *time_of_day* : *clock_time* ;

Utilisation de l'objet précédent :

time_of_day.hour := 3 ;
time_of_day.minute := 45 ;
chrono := *time_of_day.seconde* ;

L'ensemble d'un enregistrement peut être manipulé par son nom.

2.2. Les attributs précisent les propriétés des objets

Déterminer, de façon dynamique, la taille d'un tableau, le domaine de définition d'un objet scalaire, l'élément suivant d'un type énuméré, détecter la transition montante d'un signal, piloter l'optimiseur d'un outil de synthèse, attribuer des numéros de broches à des signaux d'entrées-sorties ... etc. Les attributs permettent tout cela.
Un attribut est une propriété, qui porte un nom, associée à une entité, une architecture, un type ou un signal. Cette propriété, une fois définie, peut être utilisée dans des expressions.
L'utilisation d'un attribut se fait au moyen d'un nom composé : le préfixe est le nom de l'objet auquel est rattaché l'attribut, le suffixe est le nom de l'attribut. Préfixe et suffixe sont séparés par une apostrophe « ' ».

Nom_objet 'nom_de_l'attribut

Par exemple :

hor 'event and hor = '1' renvoie la valeur booléenne true si le signal hor, de type bit, vaut 1 après un changement de valeur, ce qui revient à tester la présence d'une transition montante de ce signal.

Certains attributs sont prédéfinis par le langage, d'autres sont attachés à un outil de développement; l'utilisateur, enfin, peut définir, et utiliser ses propres attributs.

Attributs prédéfinis dans le langage

Les attributs prédéfinis permettent de déterminer les contraintes qui pèsent sur des objets ou des types : domaine de variation d'un type scalaire, bornes des indices d'un tableau, éléments voisins d'un objet de type énuméré, etc…
Ils permettent également de préciser les caractéristiques dynamiques de signaux, comme la présence d'un front, évoquée précédemment.

Le tableau ci-dessous précise le nom de quelques uns des attributs prédéfinis les plus utilisés, les catégories d'objets qu'ils permettent de qualifier et la valeur renvoyée par l'attribut.

attribut	**agit sur**	**valeur retournée**
'left 'left(n)	type scalaire type tableau	élément de gauche borne de gauche de l'indice de la dimension n, n=1 par défaut
'right 'right(n)	type scalaire type tableau	élément de droite borne de droite de l'indice de la dimension n, n=1 par défaut
'high 'high(n)	type scalaire type tableau	élément le plus grand borne maximum de l'indice de la dimension n, n=1 par défaut
'low 'low(n)	type scalaire type tableau	élément le plus petit minimum de l'indice de la dimension n, n=1 par défaut
'length(n)	type tableau	nombre d'éléments de la dimension n, n=1 par défaut
'pos(v)	type scalaire	position de l'élément v dans le type
'val(p)	type scalaire	valeur de l'élément de position p dans le type
'succ(v)	type scalaire	valeur qui suit (position + 1) l'élément de valeur v dans le type
'pred(v)	type scalaire	valeur qui précède (position - 1) l'élément de valeur v dans le type
'leftof(v)	type scalaire	valeur de l'élément juste à gauche de l'élément de valeur v
'rightof(v)	type scalaire	valeur de l'élément juste à droite de l'élément de valeur v
'event	signal	valeur booléenne "TRUE" si la valeur du signal vient de changer
'base	tous types	renvoie le type de base d'un type dérivé
'range(n)	type tableau	renvoie la plage de variation de l'indice de la dimension n, défaut n=1, dans une boucle : "for i in bus 'range loop ..."
'reverse_ range(n)	type tableau	renvoie la plage de variation, retournée (to ↔ downto), de l'indice de la dimension n, défaut n=1

Attributs spécifiques à un système

Chaque système de développement fournit des attributs qui aident à piloter l'outil de synthèse, ou le simulateur, associé au compilateur VHDL.
Ces attributs, qui ne sont évidemment pas standard, portent souvent sur le pilotage de l'optimiseur, permettent de passer au routeur des informations concernant le brochage souhaité, ... etc.

Par exemple :

attribute synthesis_off of *som4* : signal is true ;
permet, avec le compilateur « WARP », d'empêcher l'élimination du signal *som4* par l'optimiseur.

attribute pin_numbers of *T_edge* : entity is "s:20 " ;
permet, avec le même outil, de préciser que le port *s*, de l'entité *T_edge*, doit être placé sur la broche n° 20 du circuit.

Attributs définis par l'utilisateur

Syntaxe :

déclaration
attribute *att_nom* : type *;*
spécification
attribute *nom_att* of *nom_objet : nom_classe* is *expression ;*

utilisation
nom_objet 'att_nom

2.3. Les opérateurs élémentaires

Les opérateurs connus du langage sont répartis en 6 classes, en fonction de leurs priorités. Dans chaque classe les priorités sont identiques; les parenthèses permettent de modifier l'ordre d'évaluation des expressions, modifiant ainsi les priorités, et sont obligatoires lors de l'utilisation d'opérateurs non associatifs comme l'opérateur « nand ». Le tableau ci-dessous fournit la liste des opérateurs classés par priorités croissantes, de haut en bas :

classe	**opérateurs**	**types d'opérandes**	**résultat**
opérateurs logiques	and or nand nor xor	bits ou booléens	bit ou booléen
opérateurs relationnels	= /= < <= > >=	tous types	booléen
opérateurs additifs	+ - &	numériques tableaux (concaténation)	numérique tableau
signe	+ -	numériques	numérique
opérateurs multiplicatifs	* / mod rem	numériques (restrictions) entiers (restrictions)	numérique entier
opérateurs divers	not abs **	bit ou booléen numérique numériques (restrictions)	bit ou booléen numérique numérique

Ce tableau appelle quelques remarques :

- Les opérateurs multiplicatifs et l'opérateur d'exponentiation (**) sont soumis à des restrictions, notamment en synthèse où seules les opérations qui se résument à des décalages sont généralement acceptées.

- Certaines librairies standard (int_math et bv_math) surdéfinissent (au sens des langages objets) les opérateurs d'addition et de soustraction pour les étendre au type bit_vector.

- On notera que tous les opérateurs logiques ont la même priorité, il est donc plus que conseillé de parenthéser toutes les expressions qui contiennent des opérateurs différents de cette classe.

- La priorité intermédiaire des opérateurs unaires de signe interdit l'écriture d'expressions comme « a * -b », qu'il faut écrire « a * (-b) ».

2.4. Instructions concurrentes

Les instructions concurrentes interviennent à l'intérieur d'une architecture, dans la description du fonctionnement d'un circuit. En raison du parallélisme du langage, ces instructions peuvent être écrites dans un ordre quelconque. Les principales instructions concurrentes sont :

- les affectations concurrentes de signaux,
- les « processus » (décrits précédemment),
- les instanciations de composants
- les instructions « generate »
- les définitions de blocs.

Affectations concurrentes de signaux

Affectation simple

L'affectation simple traduit une simple interconnexion entre deux équipotentielles. L'opérateur d'affectation de signaux (<=) a été vu précédemment :

nom_de_signal <= *expression_du_bon_type* ;

Affectation conditionnelle

L'affectation conditionnelle permet de déterminer la valeur de la cible en fonction des résultats de tests logiques :

cible <= *source_1* when *condition_booléenne_1* else
source_2 when *condition_booléenne_*2 else
…
source_n ;

On notera un danger de confusion entre l'opérateur d'affectation et l'un des opérateurs de comparaison, l'instruction suivante est syntaxiquement juste, mais fournit vraisemblablement un résultat fort différent de celui escompté par son auteur :

-- Résultat bizarre :
cible <= *source_l* when *condition* else
cible <= *source_2*; -- <= est ici une comparaison entre *cible* et *source_2* ! dont le résultat est affecté à cible si la condition est fausse.

Affectation sélective

En fonction des valeurs possibles d'*une* expression, il est possible de choisir la valeur à affecter à un signal :

with *expression* select
cible <= *source_l* when *valeur_11* | *valeur_12* … ,
source_2 when *valeur_2*1 | *valeur_22* … ,
…
source_n when others ;

Un exemple typique d'affectation sélective est la description d'un multiplexeur.

Instanciation de composant

Le mécanisme qui consiste à utiliser un sous-ensemble (une paire entité-architecture), décrit en VHDL, comme composant dans un ensemble plus vaste est connu sous le nom *d'instanciation.* Trois opérations sont nécessaires :

.1. Le couple entité-architecture du sous-ensemble doit être créé et annexé à une librairie de l'utilisateur, par défaut la librairie « work ».
.2. Le sous-ensemble précédent doit être déclaré comme composant dans l'ensemble qui l'utilise, cette déclaration reprend les éléments principaux de l'entité du sous-ensemble.
.3. Chaque exemplaire du composant que l'on souhaite inclure dans le schéma en cours d'élaboration doit être connecté aux équipotentielles de ce schéma, c'est le mécanisme de l'instanciation.

Syntaxe de la déclaration **(.2.)** :
(simplifiée, nous omettons volontairement ici la possibilité de créer des composants « génériques », c'est à dire dont certains paramètres peuvent être fixés au moment de l'instanciation, une largeur de bus par exemple)

component *nom_composant* -- même nom que l'entité
port (*liste_ports*) ; -- même liste que dans l'entité
end component ;

Cette déclaration est à mettre dans la partie déclarative de l'architecture du circuit utilisateur, ou dans un paquetage qui sera rendu visible par une clause « use ».

lnstanciation d'un composant **(.3.)** :

Etiquette : *nom* port map (liste_d'association) ;

La liste d'association établit la correspondance entre les équipotentielles du schéma et les ports d'entrée et de sortie du composant. Cette association peut se faire par position, les noms des signaux à connecter doivent apparaître dans l'ordre des ports auxquels ils doivent correspondre, ou explicitement au moyen de l'opérateur d'association « => » :

(.1.)

```
architecture exemple of xyz is
component et
port ( a , b : in bit ;
        a_et_b : out bit) ;
end component ;
signal s_a, s_b, s_a_et_b, s1, s2, s_1_et_2 : bit ;
begin
…
-- utilisation :
et1 : et port map (s_a, s_b, s_a_et_b ) ;
-- ou :
et2 : et port map (a_et_b => s_1_et_2 , a => s1, b => s2 ) ;
…
end exemple ;
```

En raison de sa simplicité, l'association par position est la plus fréquemment employée.

Generate

Les instructions « generate » permettent de créer de façon compacte des structures régulières, comme les registres ou les multiplexeurs. Elles sont particulièrement efficaces dans des descriptions structurelles.
Une instruction generate permet de dupliquer un bloc d'instructions concurrentes un certain nombre de fois, ou de créer un tel bloc si une condition est vérifiée.

Syntaxe :

```
-- structure répétitive :
etiquette : for variable in debut to fin generate
        instructions concurrentes
end generate [etiquette] ;
```

ou :

```
-- structure conditionnelle :
etiquette : if condition generate
        instructions concurrentes
end generate [etiquette] ;
```

Donnons à titre d'exemple le code d'un compteur modulo 16, construit au moyen de bascules T, disposant d'une remise à zéro (raz) et d'une autorisation de comptage (en) actives à 'l' :

```
entity cnt16 is
        port ( ck, raz, en : in bit ;
        s : out bit_vector (0 to 3) ) ;
end cnt16 ;
```

```
architecture struct of cnt16 is
        signal etat : bit_vector (0 to 3) ;
        signal inter : bit_vector (0 to 3);
component T_edge                -- supposé présent dans la librairie work
        port ( T, hor, zero : in bit;
        s : out bit ) ;
begin
        s <= etat ;
gen_for : for i in 0 to 3 generate
        gen_ifl : if i = 0 generate
                inter(0) <= en ;
        end generate gen_ifl ;
        gen_if2 : if i > 0 generate
                inter(i) <=  etat(i - 1) and inter(i - 1) ;
        end generate gen_if2 ;
        compl_3 : T_edge port map (inter(i), ck, raz, etat(i));          -- instanciation (appel) du
composant T_edge
end generate gen_for ;
end struct ;
```

Block

Une architecture peut être subdivisée en blocs, de façon à constituer une hiérarchie interne dans la description d'un composant complexe.

Syntaxe :

```
etiquette : block [ ( expression_de_garde ) ]
-- zone de déclarations de signaux, composants, etc...
begin
-- instructions concurrentes
end block [ etiquette ] ;
```

Dans des applications de synthèse, l'intérêt principal des blocs est de permettre de contrôler la portée et la visibilité des noms des objets utilisés (signaux notamment) : un nom déclaré dans un bloc est local à celui-ci.

2.5. Instructions séquentielles

Les instructions séquentielles sont *internes* aux processus, aux procédures et aux fonctions (pour les deux dernières constructions voir paragraphes suivants). Elles permettent d'appliquer à la description d'une partie d'un circuit une démarche algorithmique, même s'il s'agit d'une fonction purement combinatoire. Les principales instructions séquentielles sont :

- L'affectation séquentielle d'un signal, qui utilise l'opérateur « <= », a une syntaxe qui est identique à celle de l'affectation concurrente simple. Seule la place, dans ou hors d'un module de programme séquentiel, distingue les deux types d'affectation; cette différence, qui peut sembler mineure, cache des comportements différents : alors que les affectations concurrentes peuvent être écrites dans un ordre quelconque, pour leurs correspondantes séquentielles, rarement utilisées hors d'une structure de contrôle, l'ordre d'écriture n'est pas indifférent.

- L'affectation d'une variable, qui utilise l'opérateur « : = », est *toujours* une instruction séquentielle.
- Les tests « if » et « case ».
- Les instructions de contrôle des boucles « loop », « for » et « while ».

Les instructions de test

Les instructions de tests permettent de sélectionner une ou des instructions à exécuter, en fonction des valeurs prises par une ou des expressions. On notera que, dans un processus, si toutes les branches possibles des tests ne sont pas explicitées, une cellule mémoire est générée pour chaque affectation de signal.

L'instruction « if … then … else … endif *»*

L'instruction if permet de sélectionner une ou des instructions à exécuter, en fonction des valeurs prises par *une* ou *des* conditions.

Syntaxe :

if *expression_logique* then
instructions séquentielles
[elsif *expression_logique* then]
instructions séquentielles
[else]
instructions séquentielles
end if ;

Son interprétation est la même que dans les langages de programmation classiques comme C ou PASCAL.

L'instruction « case… when … end case *»*

L'instruction case permet de sélectionner une ou des instructions à exécuter, en fonction des valeurs prises par *une* expression.

Syntaxe :

case *expression* is
when *choix* | *choix* | … | *choix* => *instruction sequentielle* ;
when *choix* | *choix* | … | *choix* => *instruction sequentielle* ;
…
when others => instruction sequentielle ;
end case ;

« | choix », pour « ou … », et « when others » sont syntaxiquement facultatifs. Les choix représentent différentes valeurs possibles de l'expression testée; on notera que *toutes* les valeurs possibles doivent être traitées, soit explicitement, soit par l'alternative « others ». Chacune de ces valeurs ne peut apparaître que dans une seule alternative. Cette instruction est à rapprocher du « switch » de C, ou de « case of » de PASCAL.

Les boucles

Les boucles permettent de répéter une séquence d'instructions.

Syntaxe générale

[*etiquette* :] [*schéma itératif*] loop
séquence d'instructions ;
end loop [*etiquette*] ;

Trois catégories de boucles existent en VHDL, suivant le schéma d'itération choisi :

- Les boucles simples, sans schéma d'itération, dont on ne peut sortir que par une instruction « exit ».
- Les boucles « for », dont le schéma d'itération précise le nombre d'exécutions.
- Les boucles « while », dont le schéma d'itération précise la condition de maintien dans la boucle.

Les boucles « for *»*

[*etiquette* :] for *parametre* in *minimum* to *maximum* loop
séquence d' instructions
end loop [*etiquette*] ;

ou :

[*etiquette* :] for *parametre* in *maximum* downto *minimum* loop
séquence d' instructions
end loop [*etiquette*] ;

Les boucles « while »

[*etiquette* :] while *condition* loop
séquence d'instructions
end loop [*etiquette*] ;

Les boucles « next » *et «* exit *»*

next [*etiquette*] [when *condition*] ; -- permet de passer à l'itération suivante d'une boucle.
exit [*etiquette*] [when *condition*] ; -- provoque une sortie de boucle.

3. Programmation modulaire

Small is beautiful : un gros programme ne peut être écrit, compris, testable et testé que s'il est subdivisé en petits modules que l'on met au point indépendamment les uns des autres et rassemblés ensuite. VHDL offre, bien évidemment, cette possibilité.
Chaque module peut être utilisé dans plusieurs applications différentes, moyennant un ajustage de certains paramètres, sans avoir à en réécrire le code. Les outils de base de cette construction modulaire sont les sous-programmes, procédures ou fonctions, les *paquetages* et librairies, et les paramètres génériques.

3.1. Procédures et fonctions

Les sous programmes sont le moyen par lequel le programmeur peut se constituer une bibliothèque *d'algorithmes séquentiels* qu'il pourra inclure dans une description. Les deux catégories de sous programmes, procédures et fonctions, diffèrent par les mécanismes d'échanges d'informations entre le programme appelant et le sous-programme.

Les fonctions

Une fonction retourne au programme appelant une valeur unique, elle a donc un type. Elle peut recevoir des arguments, exclusivement des signaux ou des constantes, dont les valeurs lui sont transmises lors de l'appel. Une fonction ne peut en aucun cas modifier les valeurs de ses arguments d'appel.

Déclaration :

function *nom* [(*liste de paramètres formels*)]
return *nom_de_type* ;

Corps de la fonction :

function *nom* [(*liste de paramètres formels*)]
return *nom_de_type* is
[déclarations]
begin
instructions séquentielles
end [*nom*] *;*

Le corps d'une fonction ne peut pas contenir d'instruction wait, les variables locales, déclarées dans la fonction, cessent d'exister dès que la fonction se termine.

Utilisation :

nom (*liste de paramètres réels*)

Lors de son utilisation, le nom d'une fonction peut apparaître partout, dans une expression, où une valeur du type correspondant peut être utilisée.

Exemple

Les librairies d'un compilateur VHDL contiennent un grand nombre de fonctions, dont le programme source est fourni. L'exemple qui suit, issu de la librairie bv_math du compilateur WARP, incrémente de 1 un vecteur de bits. On peut l'utiliser, par exemple, pour créer un compteur binaire.

Déclaration :

function *inc_bv* (*a*: bit_vector) return bit_vector ;

Corps de la fonction :

```
function inc_bv (a: bit_vector) return bit_vector is
variable s : bit_vector (a 'range) ;
variable carry : bit;
begin
carry := '1' ;
for i in a'low to a'high loop    -- les attributs low et high déterminent les dimensions du vecteur.
        s(i) := a(i) xor carry ;
        carry := a(i) and carry ;
end loop ;
return (s) ;
end inc_bv ;
```

Utilisation dans un compteur :

```
architecture behavior of counter is
begin
process
begin
wait until ( clk = '1') ;
if reset = '1' then
        count <= "0000" ;
elseif load = '1' then
        count <= datain ;
else
        count <= inc_bv(count) ;        -- increment du bit vector
end if ;
end process ;
end behavior ;
```

Les procédures

Une procédure, comme une fonction, peut recevoir du programme appelant des arguments : constantes, variables ou signaux. Mais ces arguments peuvent être déclarés de modes « in », « inout » ou « out » (sauf les constantes qui sont toujours de mode « in »), ce qui autorise une procédure à renvoyer un nombre quelconque de valeurs au programme appelant.

Déclaration :

procedure *nom* [(*liste de paramètres formels*)] ;

Corps de la procédure :

procedure *nom* [(*liste de paramètres formels*)] is
[déclarations]
begin
instructions séquentielles
end [*nom*] ;

Dans la liste des paramètres formels, la nature des arguments doit être précisée :

```
procedure exemple ( signal a, b : in bit;
        signal s : out bit) ;
```

Le corps d'une procédure peut contenir une instruction wait, les variables locales, déclarées dans la procédure, cessent d'exister dès que la procédure se termine.

Utilisation :

nom (*liste de paramètres réels*) ;

Une procédure peut être appelée par une instruction concurrente ou par une instruction séquentielle, mais si l'un de ses arguments est une variable, elle ne peut être appelée que par une instruction séquentielle. La correspondance entre paramètres réels (dans l'appel) et paramètres formels (dans la description de la procédure) peut se faire par position, ou par associations de noms :

exemple (*entree1*, *entree2*, *sortie*) ;

ou :

exemple (*s* => *sortie*, *a* => *entreel*, *b* => *entree2*) ;

3.2. Les paquetages (packages) et les librairies

Un paquetage (*package*) permet de rassembler des déclarations et des sous-programmes, utilisés fréquemment dans une application, dans un module qui peut être compilé à part, et rendu visible par l'application au moyen de la clause use. Un paquetage est constitué de deux parties : la déclaration, et le corps *(body).*

- La déclaration contient les informations publiques dont une application a besoin pour utiliser correctement les objets décrits par le paquetage : essentiellement des déclarations, des définitions de types, des définitions de constantes ... etc.
(on peut rapprocher la partie visible d'un package des fichiers « *.h » du langage C; ces fichiers contiennent, entre autres, les prototypes des objets, variables ou fonctions, utilisés dans un programme. La clause « use » de VHDL est un peu l'équivalent, dans cette comparaison, de la directive #include <xxx.h> du langage C)

- Le corps, qui n'existe pas obligatoirement, contient le code des fonctions ou procédures définies par le paquetage, s'il en existe.

L'utilisation d'un paquetage se fait au moyen de la clause use :

```
use work.int_math.all ;        -- rend le paquetage int_math, de la librairie work, visible dans sa totalité.
```

Le mot clé work indique l'ensemble des librairies accessibles, par défaut, au programmeur. Ce mot cache, notamment, des chemins d'accès à des répertoires de travail. Ces chemins sont gérés par le système de développement, et l'utilisateur n'a pas besoin d'en connaître les détails. Le nom composé qui suit la clause use doit être compris comme une suite de filtres : « utiliser tous les éléments du module int_math de la librairie work ».

Les paquetages prédéfinis

Un compilateur VHDL est toujours assorti d'une librairie, décrite par des paquetages, qui offre à l'utilisateur des outils variés :

- Définitions de types, et fonctions de conversions entre types : VHDL est un langage objet, fortement typé. Aucune conversion de type implicite n'est autorisée dans les expressions, mais une librairie peut offrir des fonctions de conversion explicites, et redéfinir les opérateurs élémentaires pour qu'ils acceptent des opérandes de types variés. Un bus par exemple, peut être vu, dans le langage, comme un vecteur (tableau à une dimension) de bits, et il est possible d'étendre les opérateurs arithmétiques et logiques élémentaires pour qu'ils agissent sur un bus, vu comme la représentation binaire d'un nombre entier.

- Les blocs structurels des circuits programmables, notamment les cellules d'entrées-sorties, peuvent être déclarés comme des composants que l'on peut inclure dans une description. Une porte trois-états, par exemple, sera vue, dans une architecture, comme un composant dont l'un des ports véhicule des signaux de type particulier : aux deux états logiques vient se rajouter un état haute impédance. L'emploi d'un tel opérateur dans un schéma nécessite, outre la description du composant, une fonction de conversion entre signaux logiques et signaux « trois-états ».

- Un simulateur doit pouvoir résoudre, ou indiquer, les conflits éventuels. Les signaux utilisés en simulation ne sont pas, pour cette raison, de type binaire : on leur attache un type énuméré plus riche qui rajoute aux simples valeurs '0' et 'l' la valeur 'inconnue', des nuances de force entre les sorties standard et les sorties collecteur ouvert, etc.

- La bibliothèque standard offre également des procédures d'usage général comme les moyens d'accès aux fichiers, les possibilités de dialogue avec l'utilisateur, messages d'erreurs, par exemple.

L'exemple qui suit illustre l'utilisation, et l'intérêt des paquetages prédéfinis. Le programme décrit le fonctionnement d'un circuit qui effectue une division par 50 du signal d'horloge, en fournissant en sortie un signal de rapport cyclique égal à un demi. Pour obtenir ce résultat, on a utilisé la possibilité, offerte par le paquetage « int_math » du compilateur WARP , de mélanger, dans des opérations arithmétiques et logiques, des éléments de types différents : les vecteurs de bits et les entiers (VHDL est un langage objet qui permet de surcharger les opérateurs. A chaque opérateur il est possible d'associer une fonction équivalente, ce qui permet de traiter des opérandes et un résultat de types différents de ceux qui sont définis par défaut).

```
-- div50.vhd
entity div50 is
port ( hor : in bit ;
        s : out bit ) ;          -- sortie
end div50 ;
use work.int_math.all ;          -- rend le paquetage visible
architecture arith_bv of div50 is
signal etat : bit_vector (0 to 5) ;
begin
s <= etat(5) ;
process
begin
        wait until hor = '1' ;
                if etat = 24 then                -- opérateur "=" surchargé.
                        etat <= i2bv(39,6) ;     -- fonction de conversion d'un entier en bit_vector de 6 bits.
                else
                        etat <= etat + 1 ;       -- opérateur "+" surchargé.
                end if ;
end process ;
end arith_bv ;
```

Pour assurer la portabilité des programmes, d'un compilateur à un autre, les paquetages prédéfinis sont fournis sous forme de *fichiers sources* VHDL. Cette règle permet à un utilisateur de passer d'un système de développement à un autre sans difficulté, il suffit de recompiler les paquetages qui ne seraient pas communs aux deux systèmes.
Un autre exemple de paquetage prédéfini concerne la description des opérateurs élémentaires connus d'un système. Pour synthétiser une application, avec pour cible un circuit de type 22V10 par exemple, le système de développement utilise un paquetage qui décrit les composants disponibles dans ce circuit. Ce paquetage est accessible à l'utilisateur ; mentionnons, en particulier, que l'utilisation d'une description structurelle, en terme de composants instanciés, est indispensable pour utiliser les portes trois-états de sortie du circuit.

Les paquetages créés par l'utilisateur

L'utilisateur peut créer ses propres paquetages. Cette possibilité permet d'assurer la cohérence des déclarations dans une application complexe, évite d'avoir à répéter un grand nombre de fois ces mêmes déclarations et donne la possibilité de créer une librairie de fonctions et procédures adaptée aux besoins des utilisateurs.

La syntaxe de la déclaration d'un paquetage est la suivante :

```
package identificateur is
déclarations de types, de fonctions, de composants, d'attributs,
clause use, ... etc.
end [ identificateur ] ;
```

S'il existe, le corps du paquetage doit porter le même nom que celui qui figure dans la déclaration :

```
package body identificateur is
        corps des sous-programmes déclarés.
end [ identificateur ] ;
```

Dans l'exemple qui suit on réalise un compteur au moyen de 2 bascules, dans une description structurelle. La déclaration du composant bascule est mise dans un paquetage :

```
package T_edge_pkg is
component T_edge                 -- une bascule T avec mise à 0.
port ( T, hor, raz : in bit ;
        s : out bit ) ;
end component ;
end T_edge_pkg ;
```

Le compteur proprement dit :

```
entity cnt4 is
port (ck, razero, en : in bit ;
        s : out bit_vector (0 to 1)
        ) ;
end cnt4 ;
use work.T_edge_pkg.all ;      -- rend le contenu du package précédent visible.
architecture struct of cnt4 is
signal etat : bit_vector (0 to 1) ;
signal inter:bit;
begin
s <= etat ;
inter <= etat(0) and en ;
g0 : T_edge port map ( en, ck, razero, etat(0) ) ;
g1 : T_edge port map ( inter, ck, razero, etat(l) ) ;
end struct ;
```

Les librairies

Une librairie est une collection de modules VHDL qui ont déjà été compilés. Ces modules peuvent être des paquetages, des entités ou des architectures.
Une librairie par défaut, work, est systématiquement associée à l'environnement de travail de l'utilisateur. Ce dernier peut ouvrir ses propres librairies par la clause library :

library *nom_de_la_librairie* ;

La façon dont on associe un nom de librairie à un, ou des chemins, dans le système de fichiers de l'ordinateur, dépend de l'outil de développement utilisé.

3.3. Les paramètres génériques

Lorsque l'on crée le couple entité-architecture d'un opérateur, que l'on souhaite utiliser comme composant dans une construction plus large, il est parfois pratique de pouvoir laisser certains paramètres modifiables par le programme qui utilise le composant. De tels paramètres, dont la valeur réelle peut n'être fixée que lors de l'instanciation du composant, sont appelés paramètres génériques.
Un paramètre générique se déclare au début de l'entité, et peut avoir une valeur par défaut :

generic (*nom* : *type* [:= *valeur_par_defaut*]) ;

La même déclaration doit apparaître dans la déclaration de composant, mais au moment de l'instanciation la taille peut être modifiée par une instruction « generic map », de construction identique à l'instruction « port map », précédemment rencontrée :

etiquette : nom generic map (*valeurs*)
        port map (*liste_d'association*) ;

Dans l'exemple ci-dessous, on réalise un compteur, sur 4 bits par défaut, qui est ensuite instancié comme un compteur 8 bits. Bien évidemment, le code du compteur ne doit faire aucune référence explicite à la valeur par défaut.

```
entity compteur is
generic ( taille : integer := 4 ) ;
port ( hor : in bit;
        sortie : out bit_vector ( 0 to taille - 1) ) ;
end compteur ;
use work.int_math.a11 ;
architecture simple of compteur is
signal etat : bit_vector (0 to taille - 1) ;
begin
sortie <= etat ;
process
begin
        wait until hor = '1' ;
        etat <= etat + 1 ;
end process ;
end simple;

entity compt8 is
port ( ck : in bit;
        val : out bit_vector ( 0 to 7) ) ;
end compt8 ;

architecture large of compt8 is
component compteur
generic ( taille : integer ) ;
port ( hor : in bit ;
        sortie : out bit_vector ( 0 to taille – 1 ) ) ;
end component ;
begin
u1 : compteur
        generic map (8)
        port map ( ck , val ) ;
end large ;
```

Les paramètres génériques prennent toute leur efficacité quand leur emploi est associé à la création de librairies de composants, décrits par des paquetages. Il est alors possible de créer des fonctions complexes au moyen de programmes construits de façon hiérarchisée, chaque niveau de la hiérarchie pouvant être mis au point et testé indépendamment de l'ensemble.

4. Modélisation et synthèse

Langage de modélisation des circuits intégrés complexes, VHDL est devenu un outil de synthèse. Sans parler des logiciels de conception d'ASICs sur stations de travail, la plupart des fabricants de circuits programmables offrent des solutions VHDL, plus ou moins complètes, qui ne nécessitent qu'un équipement de type PC.

4.1. Tout ce qui est synthétisable doit être simulable

Tout module synthétisable est simulable, et les résultats de simulations sont identiques à ceux que l'on observera dans le vrai circuit. Telle est la règle, généralement bien respectée. Mes des compilateurs VHDL peuvent cependant être pris en défaut, par des constructions « à la limite », et transformer par exemple des bascules *edge* (en simulation) en bascules *latch* (dans le circuit), et réciproquement.
Un programme bien construit est cependant toujours compilé correctement.

4.2. La réciproque n'est pas vraie mais ...

Toutes les constructions du langage ne sont pas synthétisables, et il n'y a rien là que de tout à fait normal. Certaines ne le sont pas par nature, elles ne prétendent pas avoir de sens dans un circuit, d'autres ne le sont pas pour un compilateur VHDL donné, à une époque donnée, mais il ne serait pas absurde, dans le principe, qu'elles le deviennent un jour. Tenant compte de ces remarques, nous classerons ci-dessous les catégories non synthétisables en 4 familles, illustrées par des exemples non exhaustifs :

- *Constructions non synthétisables par nature :* les manipulations de pointeurs, les manipulations de types non contraints (tableaux de dimensions non spécifiées - cela n'empêche cependant pas d'utiliser des fonctions dont les arguments sont non contraints, il suffit qu'à l'élaboration finale de l'unité de conception, donc à l'appel de la fonction, les dimensions soient connues -, nombres sans limitation de leur domaine de définition), la modélisation des retards, les tests de violations de *timing*, les envois de messages sur la console, les lectures de directives au clavier, les références au temps du simulateur (le type time est refusé en tant que tel par de nombreux outils de synthèse), les accès dynamiques aux fichiers.

- *Constructions non synthétisables, mais acceptées par plusieurs logiciels de synthèse comme outils généraux de conception :* certains accès aux fichiers. La lecture de fichiers de données permet, par exemple, d'initialiser des tables ou des mémoires. Vues du code synthétisable, les données issues d'un fichier n'ont de sens que si ce sont des constantes.

- *Constructions non synthétisables en raison de la complexité sous-jacente :* essentiellement les opérations arithmétiques. Tous les systèmes imposent des limites aux opérateurs et aux types d'opérandes acceptés dans ce domaine. Ces limites varient grandement d'un compilateur à l'autre et progressent d'une version à l'autre d'un même compilateur. Seule une lecture attentive de la documentation spécifique de l'outil permet de les connaître. En tout état de cause rappelons que les opérations arithmétiques « brutales » génèrent rapidement des schémas extrêmement complexes, qui se heurteront éventuellement à la réalité du circuit cible.

- *Constructions non synthétisables qui font appel à un sens caché*. L'affaire est plus délicate : à la définition de certains objets est attaché une signification qui pilote le synthétiseur, à condition qu'on ait réussi à lui faire comprendre ce que l'on souhaite obtenir. Prenons un exemple : pour réaliser une porte à sortie trois-états il peut venir à l'esprit de créer un type et une fonction de conversion associée :

```
package troisetpkg is
        type z_0_l is ( '0', 'l', 'Z' ) ;
        function to_z_0_1 ( e, oe : bit ) return z_0_l ;
end package troisetpkg ;

package body troisetpkg is
        function to_z_0_l ( e, oe : bit ) return z_0_l is
        begin
                if oe = '0' then
                        if e = '0' then
                                return '0' ;
                        else
                                return 'l' ;
                        end if ;
                else
                                return 'Z' ;
                end if ;
        end to_z_0_l ;
end package body troisetpkg ;

use work.troisetpkg.all ;

entity tri_buffer is
        port ( e, oe : in bit ;
                s : out z_0_1 ) ;
end tri_buffer ;
```

```
architecture test of tri_buffer is
begin
        s <= to_z_0_l ( e, oe ) ;
end test ;
```

Ce programme est parfaitement synthétisable, mais ne produit pas vraiment le résultat escompté. Le type énuméré est codé sur deux éléments binaires qui prennent 3 des 4 valeurs possibles ("00", "01", "10" et "11") en fonction des entrées, sans l'ombre d'état haute impédance ...

La définition d'un sous-type, héritier du type logique multivalué std_ulogic, change tout, dans le bon sens :

```
library ieee ;
use ieee.std_logic_1164.all ;

package troisetpkgieee is
        subtype z_0_l is std_ulogic range '0' to 'Z' ;
        function to_z_0_l ( e, oe : bit ) return z_0_l ;
end package troisetpkgieee ;

package body troisetpkgieee is
        function to_z_0_l ( e, ce : bit ) return z_0_l is
        begin
                -- même programme source que précédemment
        end to_z_0_l ;
end package body troisetpkgieee ;

use work.troisetpkgieee.all ;

entity tri_bufi is
        port ( e, oe : in bit ;
                s : out z_0_1 ) ;
end tri_bufi ;

architecture test of tri_bufi is
begin
        s <= to_z_0_l ( e, oe ) ;
end test ;
```

Le circuit synthétisé réalise bien un tampon trois-états, dont la sortie pilote une équipotentielle simple. L'état haute impédance est commandé par l'entrée oe. En synthèse certains types ont un sens caché mais précis, le rôle de la norme IEEE 1164 est de pousser tous les concepteurs de logiciels à tenir le même langage. Les deux programmes précédents ont évidemment strictement le même comportement en simulation, le contraire n'aurait aucun charme.

Un compilateur de synthèse peut réagir diversement face à une instruction non synthétisable : la traiter comme une erreur, l'ignorer si le programme a malgré tout un sens, voire ne pas détecter le piège, ce qui est assurément dangereux. Les restrictions apportées par un logiciel à la norme du langage sont bien évidemment documentées.

Penser « circuit »

Il y a quantité de programmes VHDL dont les algorithmes seraient justes, s'ils étaient traduits en C et exécutés de point d'arrêt en point d'arrêt, pas à pas. Diviser un projet en (petits) blocs autonomes, nommer les signaux qui permettent à ces blocs de dialoguer, imaginer l'architecture physique du circuit en cours d'élaboration représente une bonne partie du travail initial de conception VHDL.

De l'ordre dans les horloges

Tous les outils de synthèse attendent une identification claire des signaux d'horloges. Les instructions de test des fronts doivent être séparées de celles qui surveillent les commandes. Par exemple :

```
entity T_edge is
        port ( T, hor : in bit ;
                s : out bit) ;
end T_edge ;

architecture mauvaise of T_edge is
        signal etat : bit ;
begin
s <= etat ;
process ( hor )
begin
        if hor 'event and hor = '1' and T = '1' then
                etat <= not etat ;
        end if ;
end process ;
end mauvaise ;
```

est une mauvaise bascule : la même instruction recherche les fronts montants d'horloge et teste la valeur de l'entrée *T*.

La version correcte du programme sépare ces deux actions, ce qui correspond d'ailleurs à la structure physique du circuit :

```
architecture bonne of T_edge is
        signal etat : bit ;
begin
s <= etat ;
process ( hor )
begin
        if hor 'event and hor = '1' then                -- recherche du front
                if T = '1' then
                        etat <= not etat ;              -- action synchrone
                end if ;
        end if ;
end process ;
end bonne ;
```

La même remarque peut être faite avec l'instruction wait, si on l'utilise, elle ne doit exprimer que l'attente du front actif de l'horloge, à l'exclusion de tout autre test. Rappelons ici que certains compilateurs de synthèse ne sont pas très regardants en ce qui concerne les listes de sensibilités des processus. C'est évidemment une mauvaise habitude que d'en profiter pour faire de même.

5. Conclusion

VHDL est un langage qui peut déconcerter. au premier abord, le concepteur de systèmes numériques, plus habitué aux raisonnements traditionnels sur des schémas que familier des langages de description abstraite. Il est vrai que le langage est complexe, et peut présenter certains pièges, la description des horloges en est un exemple.
Ayant fait l'effort de « rentrer dedans », l'utilisateur découvre que ce type d'approche est d'une très grande souplesse, et d'une efficacité redoutable. Des problèmes de synthèse qui pouvaient prendre des heures de calcul, dans une démarche traditionnelle, sont traités en quelques lignes de programme.
N'oubliez jamais que vous êtes en train de créer un circuit, et que le meilleur des compilateurs ne peut que traduire la complexité sous-jacente de vos équations, il n'augmentera pas la capacité de calcul des circuits que vous utilisez. Le simple programme de description d'un additionneur 4 bits, comme le 74_283 :

```
entity addit is
port ( a, b : in integer range 0 to 15 ;
        cin : in integer range 0 to 1 ;
        som : out integer range 0 to 31 ) ;
end addit ;
architecture behavior of addit is
begin
som <= a + b + cin ;
end behavior ;
```

génère plus d'une centaine de termes, quand ses équations sont ramenées brutalement à une somme de produits logiques. Charge reste à l'utilisateur de piloter l'optimiseur de façon un peu moins sommaire que de demander la réduction de la somme à une expression canonique en deux couches logiques.

Tutorial 5. VHDL
Licensing + Compilation + Simulation

1. Licensing - Obtention de la licence (fichier license.dat)

1. Aller sur le site **http://www.altera.com**
2. Cliquez tout en haut de la page sur → **licensing**
3. Cliquez sur → Get licenses dans la rubrique **Get My License File**
4. Cliquez tout en bas de la page sur → **MAX+PLUS II Software for Students and Universities**
5. Cochez **MAX+PLUS II Student Edition software** → Version 10.2, 10.1, or 9.23 puis → Continue
6. → *Enter your hard disk volume serial number*, obtenu en tapant **dir /p** dans une fenêtre Terminal
(*exemple* **62E4-5A74**) puis → Continue
7. Remplir les questionnaires (donner une adresse **e-mail** valide, le reste est sans importance)
8. Recuperer par mail le fichier **license.dat** et l'inclure sous le compilateur ALTERA Max++ Baseline par la commande :
Options → License Setup → Browse
9. Le fichier **license.dat** a l'allure suivante :

```
FEATURE maxplus2web alterad 2010.03 permanent uncounted 3DB5C8857B8C \
        HOSTID=DISK_SERIAL_NUM=182fc3d1
FEATURE maxplus2vhdl alterad 2010.03 permanent uncounted 41E79E188D5C \
        HOSTID=DISK_SERIAL_NUM=182fc3d1
FEATURE maxplus2verilog alterad 2010.03 permanent uncounted \
        A52A49FB166D HOSTID=DISK_SERIAL_NUM=182fc3d1
```

2. Compilation - Fichier source d'exemple : **inverseur.vhd** pour le test du compilateur ALTERA Max++

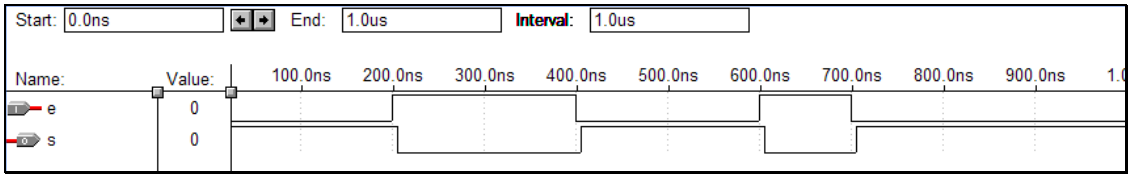
-- le fichier source .vhd doit porter le même nom que l'entité

```
entity inverseur is
port (e : in bit;
      s : out bit);
end inverseur;

architecture test of inverseur is
begin
      s <= not e;
end test;
```

3. Simulation - Exemple : **inverseur.vhd**

1. Ouvrir ou saisir le fichier **inverseur.vhd**
2. File → Project → **Set project to current file**
3. Compilation : Max+plusII → Compiler → **Compile**
4. Simulation :
4.1. File → New → Waveform Editor File (SCF File)
4.2. Node → Enter Nodes from SNF → List (Import)
4.3. File → End time
4.4. Simulation → Run
4.5. File → Project → Save, Compile and Simulation : Start + Open SCF



TD 5. VHDL

Compilateur ALTERA MaxPlus+

1. Programme VHDL des Opérateurs fondamentaux : NON (1 entrée), ET (2 entrées), OU (2 entrées)

1. Descriptions comportementales

2. Programme VHDL de l'Opérateur OU exclusif (2 entrées)

1. Description structurelle

3. Programme VHDL d'un Multiplexeur 2 → 1

1. Description comportementale
2. Description flot de données
3. Description structurelle

4. Programme VHDL d'une Bascule D latch (> 0)

TP 5. VHDL

Compilateur ALTERA MaxPlus+

1. Programme VHDL d'une Bascule RS (asynchrone)

2. Programme VHDL d'une Bascule D (>0 edge triggered)

3. Programme VHDL d'une Bascule T (>0 edge triggered)

4. Programme VHDL d'une Bascule JK (>0 edge triggered)

ELECTRONIQUE

NUMERIQUE

ANNEXE

TD 6R. REVISION LOGIQUE COMBINATOIRE & SEQUENTIELLE

1. Transcodeur Grey sur 3 bits *abc* → BCD sur 3 bits *xyz*

On donne la table de Transcodage permettant de passer du code BCD au code Grey sur 3 bits. Compléter les tables suivantes, afin de déterminer les équations décrivant les sorties *x*, *y* et *z* du Transcodeur :

Table de Transcodage

code Grey *abc*	**code BCD** *xyz*
000	000
001	001
011	010
010	011
110	100
111	101
101	110
100	111

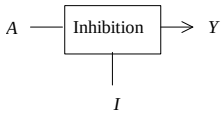
y : *a* \ *bc*	00	01	11	10
0				
1				

z : *a* \ *bc*	00	01	11	10
0				
1				

$x =$ $y =$ $z =$

2. Inhibition

On rappelle la définition de la fonction logique *Inhibition* :



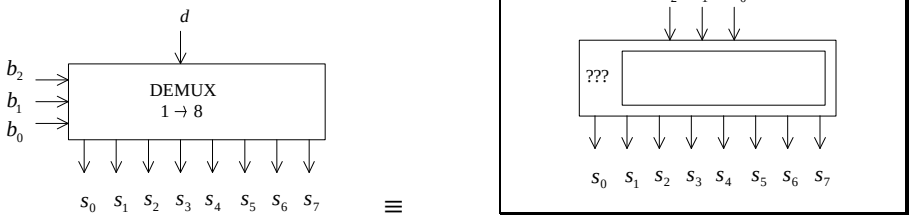
Si le signal de contrôle I vaut 0 alors le signal de sortie Y est égal au signal d'entrée A, sinon le signal de sortie Y vaut 0, quelle que soit la valeur du signal d'entrée A.

- Etablir l'expression la plus simple de la sortie *Y* en fonction de l'entrée *A* et du signal de contrôle *I*

3. Démultiplexeur 1→8

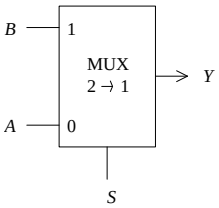
Les bits $b_2 b_1 b_0$ représentent le code BCD sur 3 bits (b_2 : MSB : bit de plus fort poids).

En choisissant le bit d à la valeur 1 et en considérant le démultiplexeur 1→8 ci-dessous comme système d'entrée $b_2 b_1 b_0$ et de sortie $s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7$, décrire la fonction ainsi réalisée :



4. Multiplexeur 2 → 1

On rappelle la définition d'un *Multiplexeur 2 → 1* :



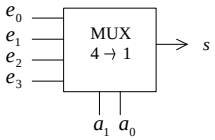
Si le signal de contrôle S vaut 0 alors le multiplexeur transmet le signal d'entrée A vers la sortie Y, sinon le multiplexeur transmet le signal d'entrée B vers la sortie Y.

a- Exprimer la sortie *Y* sous la forme d'une somme de produits logiques

b- Exprimer la sortie *Y* sous la forme d'un produit de sommes logiques

5. Multiplexeurs 4→1

On souhaite réaliser le transcodeur CBA→FED suivant, avec 3 multiplexeurs 4→1 (a_0, e_0 :LSB:bits de plus faible poids)



On donne la table du transcodeur . Compléter les 3 tables suivantes décrivant les 3 multiplexeurs 4→1 :

C	B	A	F	E	D
0	0	0	0	1	1
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	1	1
1	0	1	0	0	1
1	1	0	0	0	0
1	1	1	1	0	0

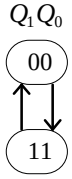
Multiplexeur 1
S = F
a_0 = B
a_1 = A
e_0 =
e_1 =
e_2 =
e_3 =

Multiplexeur 2
S = E
a_0 = B
a_1 = A
e_0 =
e_1 =
e_2 =
e_3 =

Multiplexeur 3
S = D
a_0 = B
a_1 = A
e_0 =
e_1 =
e_2 =
e_3 =

6. Synthèse de Compteur synchrone 2 bits à bascules JK

a- Synthétiser le compteur synchrone 2 bits avec 2 bascules JK *positive edge triggered* qui compte dans la séquence :



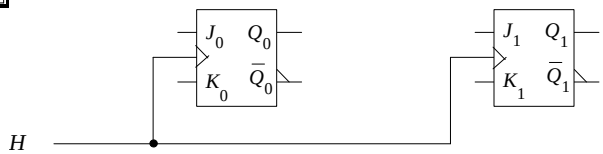
en complétant les tables suivantes ainsi que le schéma ci-dessous :

J_0 K_0 / Q_1 \ Q_0	0	1
0		
1		

J_1 K_1 / Q_1 \ Q_0	0	1
0		
1		

$$\begin{cases} J_0 = \\ K_0 = \end{cases}$$

$$\begin{cases} J_1 = \\ K_1 = \end{cases}$$

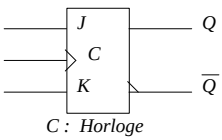


Rappel : Table de synthèse pour une bascule JK

Transition $Q_{n-1} \rightarrow Q_n$	J	K
$0 \rightarrow 0$	0	X
$0 \rightarrow 1$	1	X
$1 \rightarrow 1$	X	0
$1 \rightarrow 0$	X	1

b- Vérifier que la séquence voulue est bien obtenue en faisant l'analyse du circuit synthétisé.

Rappel : Table d'analyse pour une bascule JK



C	J	K	Q_n	
X	X	X	Q_{n-1}	} Mémoire
↑	0	0	Q_{n-1}	
↑	0	1	0	} Recopie de J
↑	1	0	1	
↑	1	1	$\overline{Q}_{n-1}$	Complément

ARCHITECTURE

DES

ORDINATEURS

0. Préambule

ADO Introduction

Objectif : Comprendre l'architecture matérielle des ordinateurs, les différents composants, le cheminement des données, l'adaptation aux différentes applications.

Plan du cours

1. Architecture Von Neuman *(1hC + 1h30TD + 1h30TP)*

ALU, accumulateur, registres

TD Microprogrammation (1 ou 2 bus)
TP Chemin des données - Circuit Maker

2. Gestion de la mémoire *(1hC + 1h30TD + 1h30TP)*

E/S, décodage adresses

TD Espace adressable, décodage d'adresses
TP Contrôleur mémoire, MMU - Circuit Maker

3. Processeurs *(1hC + 1h30TD + 1h30TP)*

Architecture, assembleur, pile

TD Langage, Instructions
TP ASM Z80 - Interpréteur

4. Representation des nombres en machine *(1hC + 1h30TD + 1h30TP)*

Codage, représentation des nombres relatifs, flottants : virgule fixe, flottante

TD Représentation - Opérations
TP Additionneur, Nombres relatifs signés/non signés - Circuit Maker

5. Processeurs actuels *(1hC + 1h30TD + 1h30TP)*

Pipelining, Prédiction de branchement, Caches, Processeurs superscalaires

TD Pipelining
TP Pipelining

6. Préparation examen *(1h30TD)*

Electronique Numérique + ADO Intro

Examen : Examen écrit sur papier avec documents et calculatrice autorisés sauf ordinateur et incluant l'ensemble (Electronique Numérique + ADO Intro)

Note :

Le chapitre manquant : *Système d'exploitation (E/S,driver, processus,signaux,threads,sémaphores,mémoire)* est abordé en cours de Système d'Exploitation (ING1 Semestre 1).

Bibliographie

[1]	**O. Temam**	« Architecture des Ordinateurs »	*Poly EP*
[2]	**A. Tanenbaum**	« Architecture de l'Ordinateur »	*IIA InterEditions*

EISTI

ARCHITECTURE DES ORDINATEURS

Introduction

Guy Almouzni

Bureau CY307b

ga@eisti.eu

Objectif : Comprendre l'architecture matérielle des ordinateurs, les différents composants, le cheminement des données, l'adaptation aux différentes applications.

Plan du cours

1. Codage - Représentation des nombres en machine

Nombres entiers, flottants.

2. Architecture de Von Neuman

Architecture, Bus, Processeurs.

3. Assembleur

Architecture processeur, Registres, Jeu d'instructions.

4. Processeurs actuels

Pipelining, Prédiction de branchement, Caches, Processeurs superscalaires.

Annexe : Gestion de la mémoire

Bibliographie

[1] O. Temam « Architecture des Ordinateurs »

[2] A. Tanenbaum « Architecture de l'Ordinateur »

CODAGE - Représentation des nombres en machine

Code Signe & Valeur Absolue (SVA) et Code Complément à 2 (C2)

N	*Code SVA (en BCD)*	*Code C2*
+3	011	011
+2	010	010
+1	001	001
+0	000	000
-0	100	-
-1	101	111
-2	110	110
-3	111	101
-4	-	100

Exemple

Calcul en **Complément à 2** sur **8 bits** de (données en *décimal*) :

122 + (-7) :

1 1111 000 (retenues)
0111 1010 (122)
+ **1111 1001** (-7)
---
1 **0111 0011** (115) => **0111 0011** représente bien **$115_{(10)}$**

Nombres à virgule fixe

Conversion décimal binaire : Partie entière

Exemple : convertir **125**

On divise le nombre 125 par 2 autant qu'il est possible,
les restes successifs étant les poids binaires (dans l'ordre des puissances croissantes)

125	: 2 =	62	reste	1
62	: 2 =	31	reste	0
31	: 2 =	15	reste	1
15	: 2 =	7	reste	1
7	: 2 =	3	reste	1
3	: 2 =	1	reste	1
1	: 2 =	0	reste	1

$125_{(10)}$ = 0111 $1101_{(2)}$ sur **8 bits**

Conversion : Partie fractionnaire

- Binaire vers décimal. *Exemple : Conversion de* **$0{,}101_{(2)}$**

La partie fractionnaire représente les coefficients des puissances négatives de 2 :

$2^{-1} = ½ = 0{,}5$; $2^{-2} = 1/4 = 0{,}25$; $2^{-3} = 1/8 = 0{,}125$ … $2^{-p} = 1/(2p)$

$$\mathbf{0{,}101_{(2)}} = \mathbf{1}x2^{-1} + \mathbf{0}x2^{-2} + \mathbf{1}x2^{-3} = 0{,}5 + 0{,}125 = \mathbf{0{,}625_{(10)}}$$

-Décimal vers binaire. *Exemple : Conversion de* **$125{,}625_{(10)}$**

On multiplie la partie fractionnaire par 2, la partie entière obtenue est le poids binaire,
la nouvelle partie fractionnaire étant à nouveau multipliée par 2 :

0,625 x 2 = **1**,25
0,25 x 2 = **0**,5
0,5 x 2 = **1**,0

→ **$0{,}625_{(10)}$** = **1**x 2^{-1} + **0**x2^{-2} + **1**x2^{-3} = **$0{,}101_{(2)}$**

Comme **$125_{(10)}$ = 0111 $1101_{(2)}$** on a :

→ **$125{,}625_{(10)}$** = **0111 1101,1010 0000$_{(2)}$** sur 8 bits

Nombres à virgule flottante

Norme IEEE754

Décomposition	Signe	Exposant (entier)	Mantisse
Simple précision (32 bits)	**1**	**8**	**23**
Double précision (64 bits)	1	11	52

Nombre normalisé :

nombre = $(-1)^{signe}$ * 1,mantisse * $2^{(exposant - biais)}$

Biais : 127 (simple précision) et 1023 (double précision)

Exemple (32 bits)

1/10 = $2^k \cdot (1, \cdots) = 0{,}0625 \times 1{,}6 = 2^{-4} \times (1 + 0{,}6) = 2^{123-127} \times (1 + 0{,}6)$ $(k \in \mathbf{Z})$

s (1) = **0**

e (8) = 123 = **0111 1011**

m (23) = 0,6 = **100 1100 1100 1100 1100 1100** (partie fractionnaire infinie)

sem = **0011 1101 1100 1100 1100 1100 1100 1100 = 3D CC CC $CC_{(H)}$**

Addition en IEEE 754

Ramener les deux nombres au même exposant
Restaurer le bit de poids fort
Effectuer l'addition ou la soustraction des valeurs absolues comme pour les entiers
Renormaliser le résultat (arrondi, bit de poids fort, exposant)

Exemple

1/10 + 1/10

1/10 = $2^k \cdot (1, \cdots) = 0{,}0625 \times 1{,}6 = 2^{-4} \times (1 + 0{,}6) = 2^{123-127} \times (1 + 0{,}6)$ $(k \in \mathbf{Z})$

s (1) = **0**

e (8) = 123 = **0111 1011**

m (23) = 0,6 = **100 1100 1100 1100 1100 1100** (partie fractionnaire infinie)

sem = **0011 1101 1100 1100 1100 1100 1100 1100 = 3D CC CC $CC_{(H)}$**

1/10 + 1/10 =
1,100 1100 1100 1100 1100 1100 * $2^{123-127}$
+ 1,100 1100 1100 1100 1100 1100 * $2^{123-127}$

11,001 1001 1001 1001 1001 1000 * $2^{123-127}$ → **à normaliser**
1,100 1100 1100 1100 1100 1100 * $2^{124-127}$ **dernier bit (0) perdu**

→ résultat : **1/10 + 1/10 =**
s (1) = 0
e (8) = 124 = 0111 1100
m (23) = 100 1100 1100 1100 1100 1100 (partie fractionnaire infinie)
sem = 0011 1110 0100 1100 1100 1100 1100 1100 = 3E 4C CC $CC_{(H)}$

Multiplication en IEEE754

Calculer le signe puis la somme des exposants
Restaurer le bit de poids fort
Effectuer la multiplication des valeurs absolues comme pour les entiers
Eventuellement, arrondir, ajuster l'exposant et renormaliser.

Exemple **-18 x 10**

X = -18 = $2^k \cdot (1, \cdots) = -16 \times 1{,}125 = -2^4 \times (1 + 0{,}125) = -2^{131-127} \times (1 + 0{,}125)$

s (1) = **1**

e (8) = 4 % 127 = 131 = **1000 0011**

m (23) = 0,125 = 2^{-3} = **001 0000 0000 0000 0000 0000** = m_X

sem = **1100 0001 1001 0000 0000 0000 0000 0000 = C1 90 00 $00_{(H)}$**

Y = 10 = $2^k \cdot (1, \cdots) = 8 \times 1{,}25 = 2^3 \times (1 + 0{,}25) = 2^{130-127} \times (1 + 0{,}25)$

s (1) = **0**

e (8) = 3 % 127 = 130 = **1000 0010**

m (23) = 0,25 = 2^{-2} = **010 0000 0000 0000 0000 0000** = m_Y

sem = **0100 0001 0010 0000 0000 0000 0000 0000 = 41 20 00 $00_{(H)}$**

Z = X x Y = (-18) x 10 =

signe : 1 (xor entre les 2 signes des 2 opérandes)

exposant : 1000 0011 + 1000 0010 - $(127)_2$ = 1000 0011 + 1000 0010 – 0111 1111
= 1000 0011 + 1000 0010 + $(-127)_{C2}$ = 1000 0011 + 1000 0010 + 1000 0001 = 0000 0101 + 1000 0001 = 1000 0110
(= 134_{10}) = 7 + 127 (**OK : 7 = 4 + 3**)

mantisse : Multiplication des mantisses m_X et m_Y :
remarque : $(1 + m_X)(1 + m_Y) = 1 + m_X.m_Y + m_X + m_Y \rightarrow m_Z = m_X.m_Y + m_X + m_Y$

```
    1,mX =          1,001 0…0
x   1,mY =        x 1,01  0…0
                  ----------
                    1001
                   0000
                  1001
                  ----------
    1,mZ =        1,01101 0…0
```

m_Z = 01101 0…0 → m_Z (23) = 011 0100 0000 0000 0000 0000

Interprétation du résultat : Z
s (1) : 1
e (8) : 1000 0110 → e = 134 → exposant = 134-127 = 7 % 127
m (23) : 011 0100 0000 0000 0000 0000 → mantisse = $2^{-2} + 2^{-3} + 2^{-5}$ = 0,40 625

sem = 1100 0011 0011 0100 0000 0000 0000 0000 = C3 34 00 $00_{(H)}$
Z = - 1,40 625 x 2^7 = - 1,40 625 x 128 = - 180

TD 1 ADO. Codage. Représentation des nombres en machine

On pourra s'aider de la calculatrice Windows – mode scientifique – pour vérification des conversions Décimal/Binaire/Hexadécimal

Exercice 1 : nombre entiers relatifs sur machine

0. Quel est l'intervalle des entiers naturels représentables en code BCD sur 16 bits ?

1. Quel est l'intervalle des entiers relatifs représentables en complément à 2 sur 16 bits ?

2. a) Donner la conversion Hexa et Binaire de 2010_{10}. b) Donner la conversion en Décimal de $7DA_H$.

3. Exprimez en base 10 les nombres dont le codage en complément à 2 sur 16 bits est le suivant :

a) 0110 1100 0001 1011

b) 1011 0110 1011 0011

4. Calculer en complément à 2 sur 8 bits les additions suivantes (données en décimal) :

a) 122 + (-7) ; b) (-111) + (-17) ; c) 111 + 17.

Vous ferez bien apparaître toutes les retenues intermédiaires. Précisez si le résultat est correct ou s'il y a dépassement de capacité. Vérifiez les propriétés suivantes de l'addition en Complément à 2 :

- il n'y a pas dépassement de capacité si les signes des opérandes sont différents. Il y a dépassement si les signes des opérandes sont égaux avec un changement de signe dans le résultat ;
- il y a dépassement de capacité lors d'une addition de deux entiers relatifs si et seulement si les deux dernières retenues (de poids les plus élevés) sont différentes.

Exercice 2 : nombres à virgule fixe

1. Donner l'équivalent binaire (parties entière et décimale sur 8 bits) des nombres décimaux suivants :

a) 118,625 ; b) 1/10 ; c) 4/3.

2. Exprimez en base 10 le nombre binaire suivant : 101,101.

Exercice 3 : nombres à virgule flottante (IEEE754)

1. Donner l'équivalent binaire en simple précision des nombres décimaux suivants :

a) 1,5 ; b) 0,5 ; c) -142,625 ; d) 10 ; e) 1/10

2. Faire l'addition en IEEE754 simple précision de 1/10 et 1/10. Reconnaître le résultat.

3. Faire la multiplication en IEEE754 simple précision de -18 par 10.

Rappel de la norme IEEE754

Décomposition	**Signe**	**Exposant (entier)**	**Mantisse**
Simple précision (32 bits)	**1**	**8**	**23**
Double précision (64 bits)	1	11	52

Nombre normalisé : nombre = $(-1)^{\text{signe}} * 1,\text{mantisse} * 2^{(\text{exposant} - \text{biais})}$

Nombre dénormalisé : nombre = $(-1)^{\text{signe}} * 0,\text{mantisse} * 2^{(\text{exposant} - \text{biais})}$

Biais : 127 (simple précision) et 1023 (double précision)

Cas particuliers :

Signe	Exposant	Mantisse	Valeur
-	0…0	0…0	+/- 0
-	1…1	0…0	+/- ∞
-	1…1	-	NaN
-	0…0	-	Nombre dénormalisé

Rappel : Code Signe & Valeur Absolue (SVA) et Code Complément à 2 (C2) - Table des puissances de 2

N	*Code SVA (en BCD)*	*Code C2*
+3	011	011
+2	010	010
+1	001	001
+0	000	000
-0	100	-
-1	101	111
-2	110	110
-3	111	101
-4	-	100

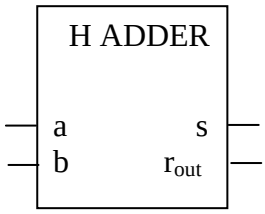
Table	2^n
2^0	1
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256

Table	2^n		*Table*	2^n
2^9	512		2^{-1}	0.5
2^{10}	1024		2^{-2}	0.25
2^{11}	2048		2^{-3}	0.125
2^{12}	4096		2^{-4}	0.0625
2^{13}	8192		2^{-5}	0.03125
2^{14}	16384		2^{-6}	0.015625
2^{15}	32768		2^{-7}	0.0078125
2^{16}	65536		2^{-8}	0.00390625

TP 1. ADO - Codage

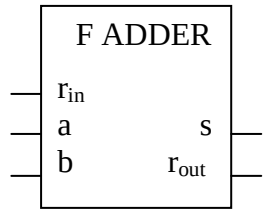
Exercice 1

a) Réaliser sous CircuitMaker une macro (composant réutilisable) un composant demi-additionneur 1 bit : **HalfAdder**.
b) Faire un test de la macro avec un fichier .ckt utilisant la macro **HalfAdder**.
c) Simuler votre test (scénarios de test avec des *logic displays*).



Exercice 2

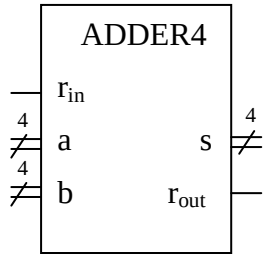
a) Réaliser sous CircuitMaker un additionneur complet 1 bit : **FullAdder** qui réutilise votre demi-additionneur.
b) Faire un test.
c) Simuler votre test (scénarios de test avec des *logic displays*).



Exercice 3

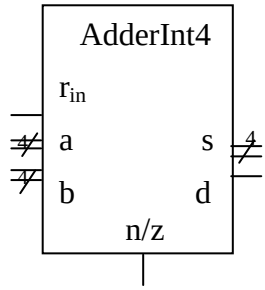
a) Réaliser sous CircuitMaker un additionneur 4 bits : **Adder4** qui réutilise 4 additionneurs complets. Ce composant a une retenue entrante et une retenue sortante.
b) Faire un test.
c) Simuler votre test (scénarios de test avec des *hex displays*).

Devices → Digital animated → Displays → Hex display



Exercice 4 (Facultatif)

a) Réaliser sous CircuitMaker un additionneur sur des nombres sur 4 bits naturels ou relatifs : **AdderInt4** qui réutilise l'additionneur 4 bits précédent. On garde la retenue entrante et on rajoute un signal n/z qui permet de sélectionner le type des opérandes (1 pour les naturels, 0 pour les relatifs). En sortie, on remplace la retenue sortante par un signal de dépassement de capacité.
b) Faire un test.
c) Simuler votre test (scénarios de test avec des *hex displays*).



Compte-rendu

Faire un compte-rendu du TP1 incluant les schémas de vos composants, les sources CircuitMaker et les scénarios de simulation commentés.

Facultatif : Optimisation du calcul de retenue (retenue par anticipation *versus* propagation de retenue)

Le temps de traversée de l'additionneur 4 bits est dû en grande partie à la propagation de la retenue entre les différents additionneurs complets (exemple 0001 + 1111). Nous allons étudier une solution qui permet d'optimiser ce calcul. On définit les fonctions suivantes pour i = 0 à 3 (0 poids faible, 3 poids fort).

- génération de retenue : $g_i = a_i.b_i$
- propagation de retenue : $p_i = a_i \text{ xor } b_i$
- calcul de la retenue : $c_i = g_i + p_i.c_{i-1}$
- calcul de la somme : $s_i = p_i \text{ xor } c_{i-1}$

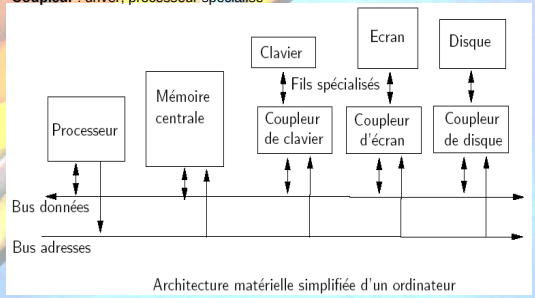
a) expliquer ces formules (on pourra remarquer que (g_i,p_i) constituent la sortie d'un ½ additionneur) ;
b) donner les formules de s_0, c_0, s_1, c_1, s_2, c_2, s_3, c_3 en fonction de g_0, p_0, g_1, p_1, g_2, p_2, g_3, p_3 et c_e (la retenue entrante) ;
c) développer ces formules pour obtenir un temps de propagation minimal ;
d) en déduire le schéma du nouvel additionneur 4 bits à partir de 4 ½ additionneurs
e) réaliser le circuit sous CircuitMaker
f) vérifier le gain dans le temps de traversée de l'additionneur 4 bits ; on pourra utiliser des portes OR et AND à un nombre quelconque d'entrées.

Facultatif : VHDL *versus* CircuitMaker

Reprendre les différentes questions du TP avec une solution logicielle (VHDL) plutôt que matérielle (CircuitMaker).

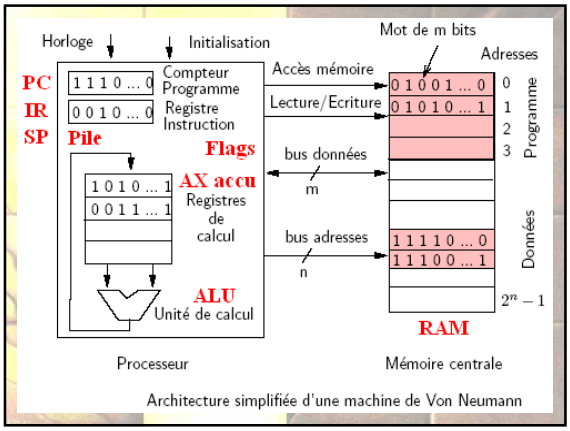
ARCHITECTURE DE VON NEUMANN

Coupleur : driver, processeur spécialisé



Architecture matérielle simplifiée d'un ordinateur

Une **unité de contrôle** (avec séquenceur câblé ou programmé et avec bus de contrôle) gère les échanges (ente les autres unités) et les interruptions (contrôleur d'interruptions).



Architecture simplifiée d'une machine de Von Neumann

Chemin de données

Le *chemin de données* d'un processeur est la partie matérielle qui effectue les opérations nécessaires à exécuter un programme.

Unité de contrôle

L'*unité de contrôle* est la partie du processeur qui gère le déroulement des opérations dans le chemin de données.

Exécution d'une instruction

- Recherche de l'instruction
- Décodage de l'instruction
- Recherche des opérandes
- Exécution de l'opération
- Stockage du résultat

Etapes d'exécution d'une instruction

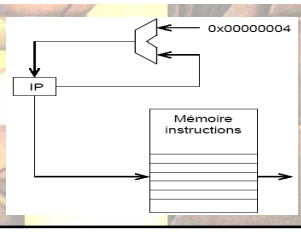
L'exécution d'une instruction dans le chemin de données est composée de cinq étapes :

1. Chargement de l'instruction
2. Décodage de l'instruction + lecture des registres
3. UAL
4. Accès mémoire
5. Ecriture aux registres

Chemin de données - Chargement de l'instruction

Etape 1 :

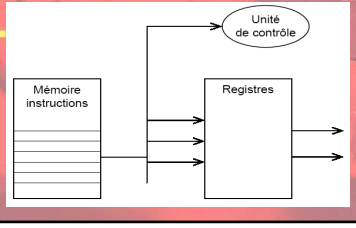
- Le processeur consulte le registre IP (*Instruction Pointer*) encore appelé *Program Counter (PC)* ou encore ***Compteur Ordinal (CO)*** qui contient l'adresse de la prochaine instruction à exécuter
- l'instruction est chargée dans IR (Instruction Register) depuis la mémoire.
- L'adresse contenue dans IP est incrémentée.



Chemin de données - Décodage + lecture registres

Etape 2 :

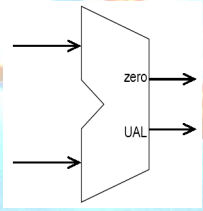
- Les différents champs de l'instruction sont décodés.
- Le code de l'opération est envoyé à l'unité de contrôle afin de déterminer le type d'instruction et la longueur des champs.
- Les données des registres des opérandes source sont lues.



Chemin de données - UAL

Etape 3 :

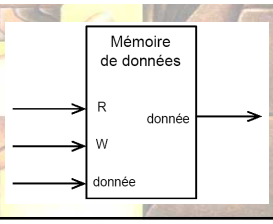
- L'UAL effectue l'opération arithmétique ou logique de l'instruction.
- Pour les instructions nécessitant un accès à la mémoire, le calcul des adresses s'effectue à cette étape.



Chemin de données - Accès mémoire

Etape 4 :

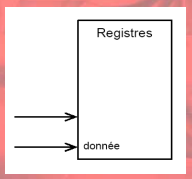
- Une donnée est lue ou écrite en mémoire.
- Cette étape n'est utilisée que par les instructions de type `load` ou `store`.
- Le système de cache permet de rendre cette étape aussi rapide en moyenne que les autres.



Chemin de données - Ecriture registre

Etape 5 :

- Le résultat de l'instruction est écrit dans un registre.
- Le numéro du registre destination a été décodé à l'étape 2.

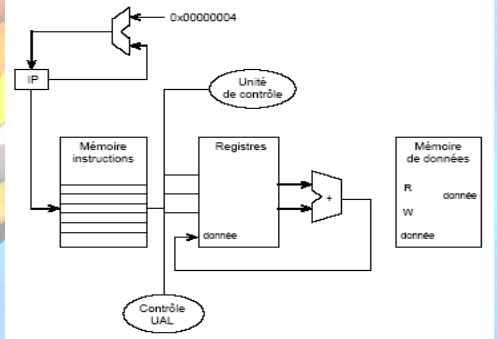


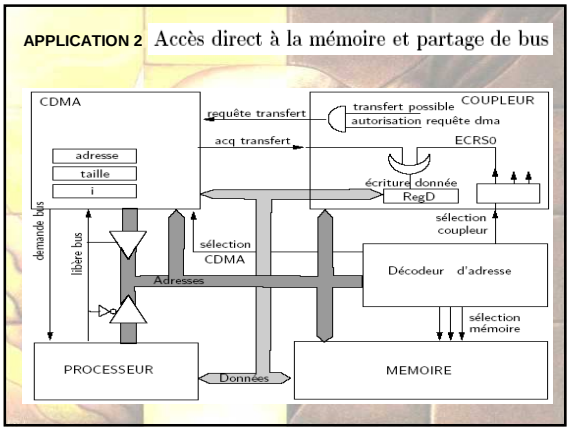
Améliorations de l'architecture de Von Neumann

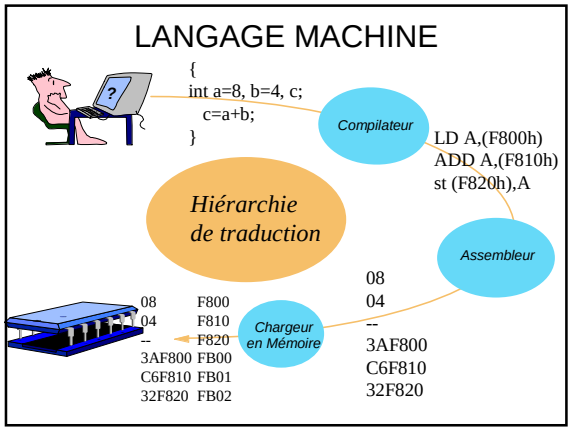
- **DMA** (Direct Memory Access)
- **Mémoire cache** (Tampon rapide)
- **Pipelining (Archi. Superscalaire)**
  (Découpage des instructions en étages simultanés)
- **Machine vectorielle**
  (Parallélisation des données traitées par 1 processeur unique)
- **Parallélisme** (Parallélisation avec plusieurs processeurs)

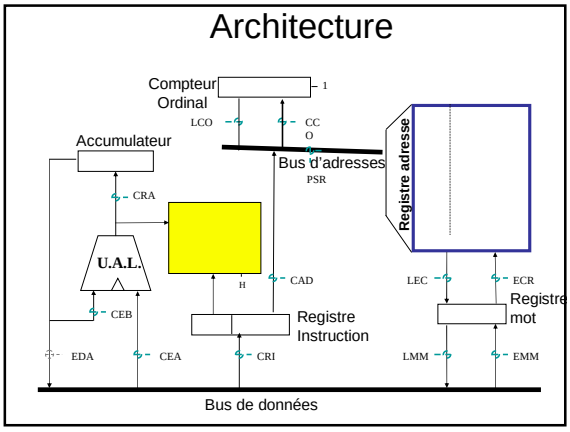
ANNEXE

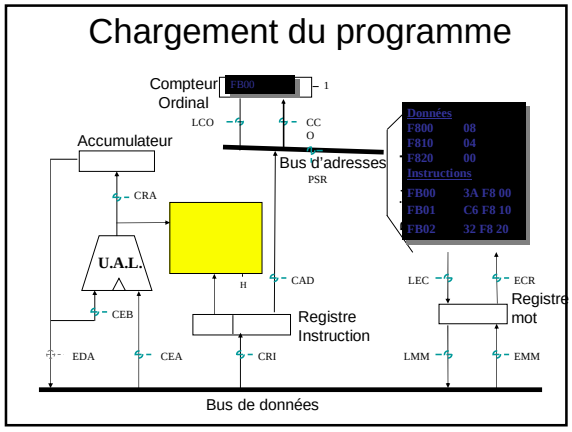
APPLICATION 1 Instruction add - Chemin de données

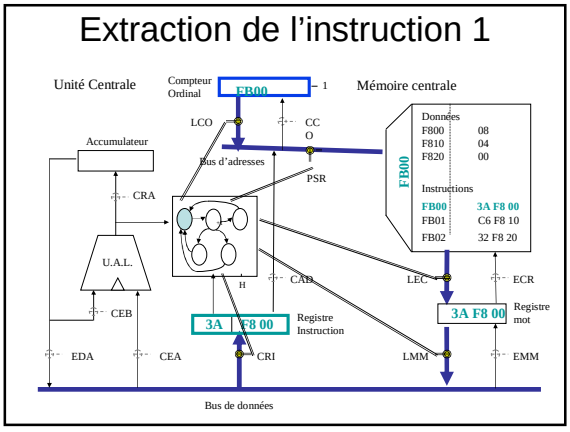


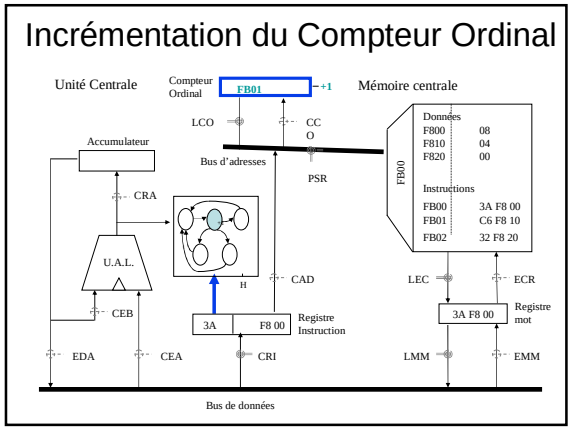

APPLICATION 2 Accès direct à la mémoire et partage de bus
CDMA
requête transfert
acq transfert
adresse
taille
i
transfert possible
autorisation requête dma
COUPLEUR
ECRS0
écriture donnée
RegD
sélection coupleur
demande bus
libère bus
sélection CDMA
Adresses
Décodeur d'adresse
sélection mémoire
PROCESSEUR
Données
MEMOIRE


LANGAGE MACHINE
{
int a=8, b=4, c;
c=a+b;
}
Compilateur
LD A,(F800h)
ADD A,(F810h)
st (F820h),A
Hiérarchie de traduction
Assembleur
08
04
--
3AF800
C6F810
32F820
Chargeur en Mémoire
08 F800
04 F810
-- F820
3AF800 FB00
C6F810 FB01
32F820 FB02

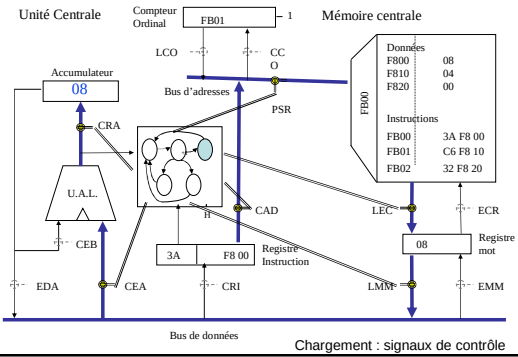

Architecture
Compteur Ordinal
LCO
CC
O
Accumulateur
Bus d'adresses
PSR
Registre adresse
CRA
U.A.L.
H
CAD
LEC
ECR
CEB
Registre Instruction
Registre mot
EDA
CEA
CRI
LMM
EMM
Bus de données


Chargement du programme
Compteur Ordinal
FB00
LCO
CC
O
Accumulateur
Bus d'adresses
PSR
Données
F800 08
F810 04
F820 00
Instructions
FB00 3A F8 00
FB01 C6 F8 10
FB02 32 F8 20
CRA
U.A.L.
H
CAD
LEC
ECR
CEB
Registre Instruction
Registre mot
EDA
CEA
CRI
LMM
EMM
Bus de données


Extraction de l'instruction 1
Unité Centrale
Compteur Ordinal
FB00
Mémoire centrale
LCO
CC
O
Accumulateur
Bus d'adresses
PSR
Données
F800 08
F810 04
F820 00
Instructions
FB00 3A F8 00
FB01 C6 F8 10
FB02 32 F8 20
CRA
U.A.L.
H
CAD
LEC
ECR
CEB
3A
F8 00
Registre Instruction
3A F8 00
Registre mot
EDA
CEA
CRI
LMM
EMM
Bus de données

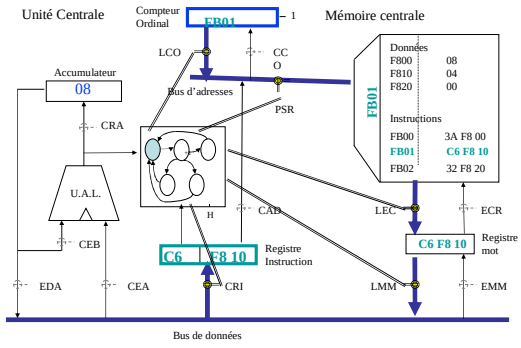

Incrémentation du Compteur Ordinal
Unité Centrale
Compteur Ordinal
FB01
+1
Mémoire centrale
LCO
CC
O
Accumulateur
Bus d'adresses
PSR
Données
F800 08
F810 04
F820 00
Instructions
FB00 3A F8 00
FB01 C6 F8 10
FB02 32 F8 20
CRA
U.A.L.
H
CAD
LEC
ECR
CEB
3A
F8 00
Registre Instruction
3A F8 00
Registre mot
EDA
CEA
CRI
LMM
EMM
Bus de données

Exécution de l'instruction 1

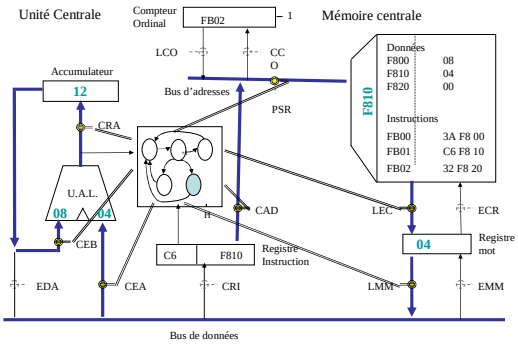


Chargement : signaux de contrôle

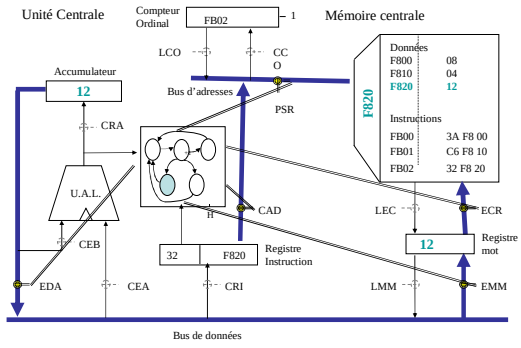
Extraction de l'instruction 2



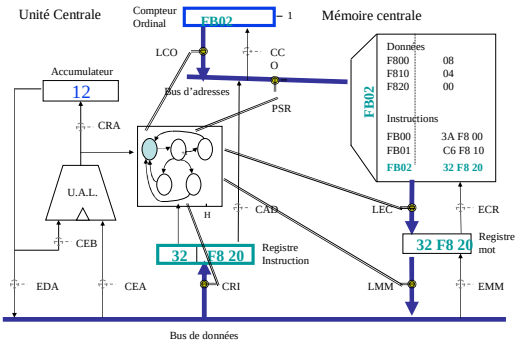
Incrémentation du Compteur Ordinal



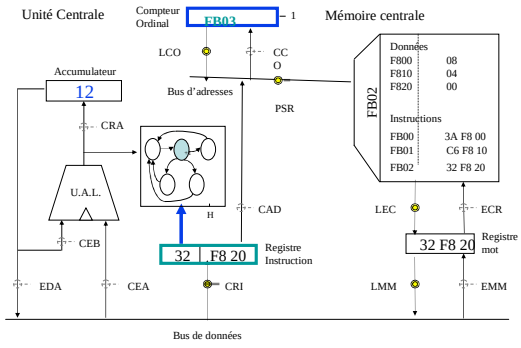
Exécution de l'instruction 2



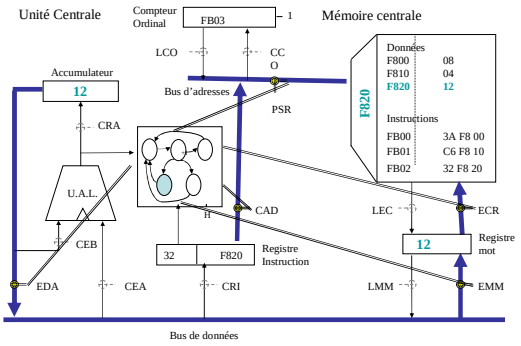
Extraction de l 'instruction 3



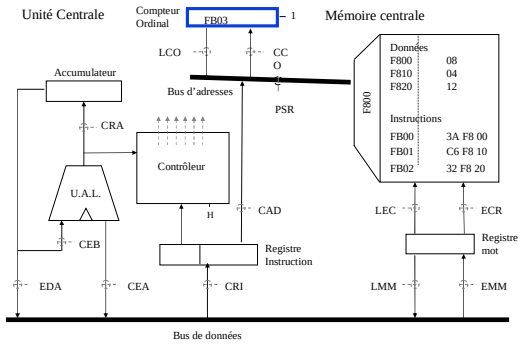
Incrémentation du Compteur Ordinal



Exécution de l'instruction 3

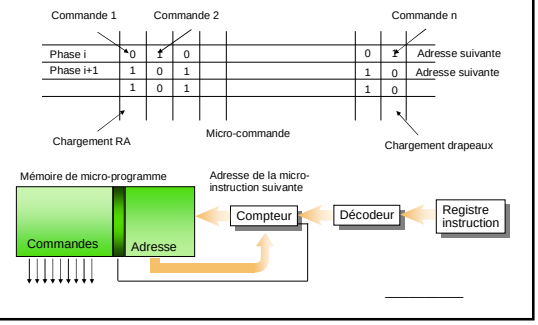


Programme terminé



La micro-programmation

Séquenceur (des signaux de contrôle) microprogrammé versus séquenceur câblé



TD 2 ADO. Architecture de Von Neumann

Exercice 1 : Modes d'adressage (Complément de cours)

Les modes d'adressage

Un mode d'adressage est une méthode permettant d'interpréter, d'accéder à un opérande (aux données) lors de l'exécution d'une instruction. Par exemple l »assembleur MC68000 de Motorola présente 6 modes d'adressage :

1. Adressage Direct : l'opérande est un registre de données ou d'adresse.

2. Adressage Indirect : l'opérande est désigné :
- soit par le contenu d'un registre d'adresse,
- soit par l'addition du contenu d'un registre d'adresse et d'une constante (*offset*) et/ou du contenu d'un registre de donnée ou d'adresse (*index*).

3. Adressage Immédiat : la donnée est fournie dans le code instruction.

4. Adressage Absolu : l'adresse de la donnée est fournie dans le code instruction.

5. Adressage Relatif : l'adresse de la donnée est calculée par addition du contenu du Compteur Ordinal et d'un *offset* et/ou d'un *index*.

6. Adressage Implicite : les registres impliqués sont les registres de contrôle (Registre d'Etat, Compteur Ordinal, Pile).

Motorola donne le tableau 4.1 récapitulant les modes d'adressage avec les codes associés. An et Dn désignent respectivement les registres d'adresse et de donnée.

Mode	Code	Champ registre	Syntaxe
Direct	000	Num. reg.	`Dn`
Direct	000	Num. reg.	`An`
Indirect	010	Num. reg.	`(An)`
Indirect	011	Num. reg.	`(An)+`
Indirect	100	Num. reg.	`-(An)`
Indirect	101	Num. reg.	`d(An)`
Indirect	110	Num. reg.	`d(An,Rm)`
Absolu	111	000	`xxxx`
Absolu	111	001	`xxxxxxxx`
Relatif	111	010	Rel. CO
Relatif	111	011	Rel. CO+Rm
Immédiat	111	100	`#xxxx`

Table 4.1: Modes d'adressage du M68000.

Exemple 1 (Instructions de transfert)

`move,w d3,-(a4)` : les 16 bits de poids faible (extension `,w`) du registre `d3` sont transférés à l'adresse donnée par le contenu du registre `a4` décrémenté de 1 avant transfert. Le rangement en mémoire se fait octet de poids faible d'abord

Exemple 2 (Instructions de comparaison)

`cmp,l d4,d2`: compare d4 à d2 en effectuant la soustraction d2−d4. Les drapeaux (sauf X) sont modifiés en conséquence.

Exemple 3 (Instructions de branchement inconditionnel)

`bra plus_loin`: l'argument du branchement est un déplacement sur 8 ou 16 bits qui est ajouté au contenu du compteur ordinal (celui-ci contient alors l'adresse de l'instruction qui suit).

`jmp (a3)` : branchement à l'adresse donnée par le contenu de a3.

L'instruction ASSEMBLEUR INTEL 8086 :

AND AX, 06 *(AX est un registre accumulateur 16 bits)*

a pour code Machine :

25 06 00$_H$ *(convention INTEL Little Endian)*

Elle est implantée à l'adresse 01 00$_H$.

a) Indiquer le contenu des registres IR et IP juste avant exécution de l'instruction.

b) Indiquer le contenu du registre IP juste après exécution de l'instruction.

Exercice 2 : Programme ASSEMBLEUR (ASM)

Soit l'extrait de programme ASSEMBLEUR INTEL 8086 suivant, stocké à l'adresse 01 00$_H$ (via le code ASM **ORG 100h**) avec les valeurs initiales : **AX = 00 00$_H$** , **BX = 00 00$_H$** et l'état de pile (STACK) suivant : **STACK** : **FF FE$_H$: 00 00$_H$**
FF FC$_H$: 00 00$_H$
FF FA$_H$: 00 00$_H$
…

Code ASM	*Code Machine*	*Commentaire* *(AX) signifie : contenu de AX*
MOV AX, 0100h	B8 00 01	Ecrit 01 00$_H$ dans le registre AX ; **(AX) = 01 00** (convention INTEL Little Endian)
MOV BX, 0304h	BB 04 03	Ecrit 03 04$_H$ dans le registre AX ; **(AX) = 03 04** (convention INTEL Little Endian)
Boucle: ADD AL, 1	04 01	Ajoute 1 à l'octet de poids faible de AX noté AL : **(AL) = (AL) + 1**
CMP AL, 2	3C 02	Compare (AL) à 2 ; place le bit de Flag **Z** à **1** en cas d'**égalité** de la comparaison
JNE Boucle	75 FA	Saut à l'étiquette ***Boucle*** si le bit **Z = 0 (≡** s'il n'y a pas égalité) (*Jump Not Equal*)
PUSH AX	50	Empile le contenu de AX dans la pile (STACK) : **(AX) → STACK**
PUSH BX	53	Empile le contenu de AX dans la pile (STACK) : **(BX) → STACK**

Attention : ne pas introduire de caractère « espace » dans l'étiquette « Boucle: »

a) Compléter ce tableau lors de l'exécution complète et pas à pas du programme, en indiquant le contenu des registres spécifiés :

Instruction ASM	**IP**	**AX**	**BX**	**Flag Z**	**SP**	**STACK (**FFFF,**FFFE**,FFFD,**FFFC**,FFFB,**FFFA)**
Etat initial				**0**		
MOV AX, 0100h						

TP 2 ADO. Architecture de Von Neumann

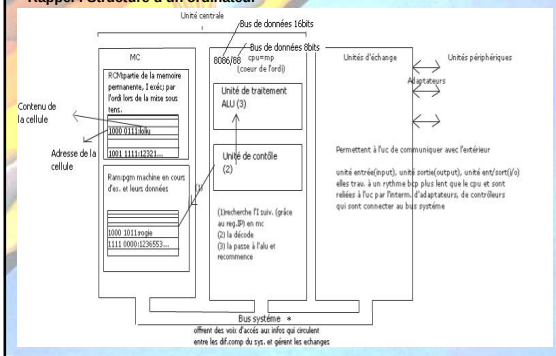
b) Vérifier les résultats de la question précédente(TD 2) en assemblant (menu *emulate*) le programme précédent avec le simulateur Assembler with Microprocessor Simulator 8086 (fichier **emu8086v408r.exe**) téléchargeable à l'adresse :

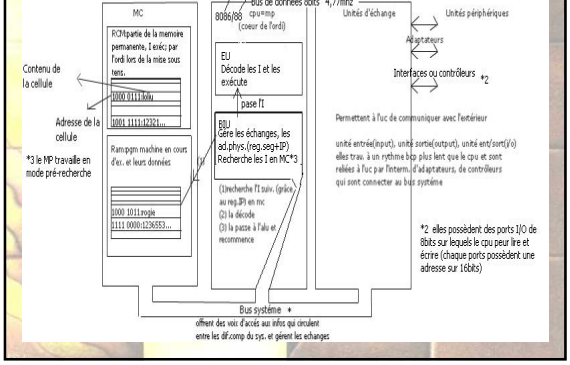
http://pcwin.com/Software_Development/Debugging/Assembler_with_Microprocessor_Simulator_8086/index.htm (rubrique *download*)

_____

ASSEMBLEUR 80x86

Rappel : Structure d'un ordinateur





Les registres

Les registres généraux (16 bits)

AX (accumulateur) :	instructions d'I/O et certaines opérations arithmétiques
BX (registre de base) :	registre d'adressage
CX (compteur) :	compteur de boucles
DX :	adresses des ports pour les instructions I/O

Le registre des indicateurs (16 bits) **FR**

Flags indicateurs d'état

CF : retenue (opération arithmétique)
SF : résultat négatif (entier signé)
ZF : résultat nul ou opérandes égaux
OF : débordement

Jeu d'instructions 80x86

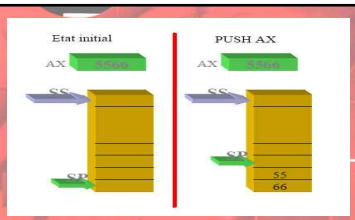
Instructions de transfert de données.
Instructions arithmétiques.
Instructions de bits (logiques).
Instructions de sauts de programme.
Instructions de chaîne de caractères.
Instructions de contrôle de processus.
Instructions d'interruptions.

Les instructions de transfert de données

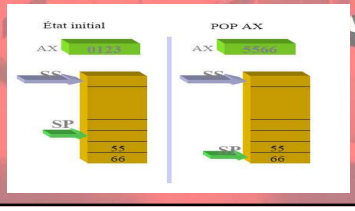
Usage	Nom	Fonction
Général	MOV	Transfert d'octets ou de mots
	PUSH	Chargement de la pile
	POP	Déchargement de la pile
	PUSHA	Chargement de tous les registres dans la
	POPA	pile Déchargement de tous les registres
	XCHG	dans la pile Echange d'octet ou de mot
	XLAT	Translation d'octet
Entrées-sorties	IN	Entrée de mot ou d'octet
	OUT	Sortie de mot ou d'octet
Adresses	LEA	Chargement de l'adresse effective
	LDS	Chargement du pointeur avec DS
	LES	Chargement du pointeur avec ES
Indicateurs	LAHF	Transfert des indicateurs dans AH
	SAHF	Rangement de AH dans les indicateurs
	PUSHP	Chargement des indicateurs dans la pile
	POPF	Déchargement des indicateurs de la pile.

Exemple MOV destination, source
MOV AX, BX ;Copie du contenu d'un registre 16 bits vers un registre 16 Bits: (BX) → AX

PUSH



POP



Instructions arithmétiques

Usage	Nom	Fonction
Addition	ADD	Addition sur un octet ou un mot
	ADC	Addition sur un octet ou un mot avec retenue
	INC	Incrémentation de 1
	AAA	Ajustement ASCII
	DAA	Ajustement décimal
Soustraction	SUB	Soustraction sur un octet ou un mot
	SBB	Soustraction sur un octet (mot) avec retenue
	DEC	Décrémentation de 1
	NEG	Mêtre un octet ou un mot en négatif
	CMP	Comparaison d'octet ou mot
	AAS	Ajustement ASCII
	DAS	Ajustement décimal
Multiplication	MUL	Multiplication d'octet ou de mot **non signée**
	IMUL	Multiplication d'octet ou de mot **signée**
	AAM	Ajustement ASCII
Division	DIV	Division d'octet ou de mot **non signée**
	IDIV	Division d'octet ou de mot **signée**
	AAD	Ajustement ASCII
	CBW	Conversion d'un octet en un mot
	CWD	Conversion d'un mot en double mots

Les instructions logiques

Usage	Nom	Fonction
Logique	NOT	Inversion logique sur un octet ou un mot
	AND	Et logique
	OR	Ou logique
	XOR	Ou exclusif
	TEST	Et logique sans résultat, affecte uniquement les indicateurs du registre des flags.
Décalages	SHL	Décalage logique à gauche
	SAL	Décalage arithmétique à gauche
	SHR	Décalage logique à droite
	SAR	Décalage arithmétique à droite
Rotation	ROL	Rotation à gauche
	ROR	Rotation à droite
	RCL	Rotation à gauche à travers le bit de retenue
	RCR	Rotation à droite à travers le bit de retenue

Les instructions de Saut conditionnel

JC : (Saut si retenue)	Si CF=1 alors IP = IP + déplacement
JE/JZ : (Saut si égal/Si zéro)	Si ZF=1 alors IP = IP + déplacement
JNC : (Saut si pas de retenue)	Si CF=0 alors IP = IP + déplacement
JNE/JNZ : (Saut si non égal/Non zéro)	Si ZF=0 alors IP = IP + déplacement
JNO : (Saut si pas de débordement)	Si OF=0 alors IP = IP + déplacement
JNP : (Saut si pas de parité)	Si PF=0 alors IP = IP + déplacement
JNS : (Saut si pas de signe)	Si SF=0 alors IP = IP + déplacement
JO : (Saut si débordement)	Si OF=0 alors IP = IP + déplacement
JP/JPE: (Saut si parité (paire))	Si PF=1 alors IP = IP + déplacement
JS : (Saut si signe (négatif))	Si SF=1 alors IP = IP + déplacement

Les instructions de boucle Elles décrémentent le contenu de **CX**

LOOP

Exemple :

```
        MOV  AX, 05
        MOV  CX, 05
Début:  INC AX          SI CX<>0
        LOOP Début
SI CX=0 MOV BX, AX
```

LOOPE / LOOPZ

Exemple :

```
        MOV AX, 03
        MOV CX, 6
DEBUT:  ADD DH, DL      SI CX<>0
        INC AX          Et  AL=06
        CMP AL, 06
SI CX=0 LOOPZ DEBUT
Ou  AL<>06  MOV AH, 03
```

LOOPNE / LOOPNZ

```
        MOV AX, 03
        MOV CX, 6
DEBUT:  ADD DH, DL      SI CX<>0
        INC AX          Et  AL<>06
        CMP AL, 06
SI CX=0 LOOPZ DEBUT
Ou  AL=06   MOV AH, 03
```

Les instructions de commande du processeur

Type	Nom	Fonction
Indicateur (FLAGS)	STC	Met à 1 la retenue CF
	CLC	MET à 0 la retenue CF
	CMC	Complémente la retenue
	STD	Met à 1 la direction DF
	CLD	Met à 0 la direction DF
	STI	Met à 1 l'autorisation d'interruption
	CLI	Met à 0 l'autorisation d'interruption
Synchronisation	HLT	Halte jusqu'à interruption ou RESET
	WAIT	Attente jusqu'à broche TEST passe à 0
	ESC	Pour un coprocesseur
	LOCK	Verrouillage des bus pendant la prochaine instructions
Sans opération	NOP	Pas d'opération

Interruptions logicielles système

Exemples

Interruptions DOS

Lecture d'un caractère au clavier avec écho

Paramètre d'entrée :	AH = 1	
Appel d'interruption :	INT 21H	
Paramètre de sortie :	AL	; (AL) = code ASCII du caractère

Interruptions BIOS

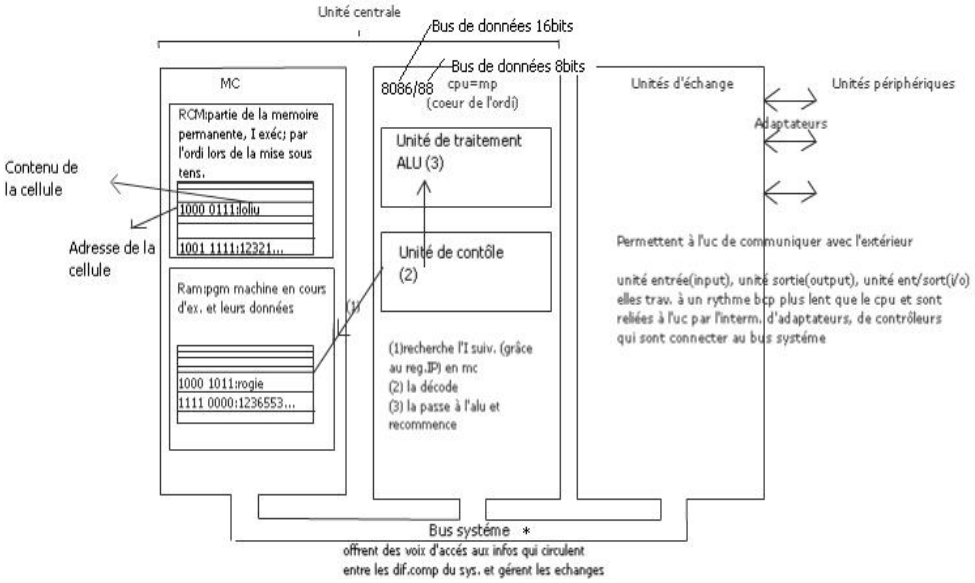
Affichage d'un caractère à l'écran

Paramètres d'entrée :	AH = 0EH AL = Code ASCII du caractère a afficher à l'écran
Appel d'interruption :	INT 10H
Paramètres de sortie :	aucun

TD-TP 3 ADO. Assembleur 80x86

Complément de cours 1 : Architecture des processeurs 80x86 (INTEL)

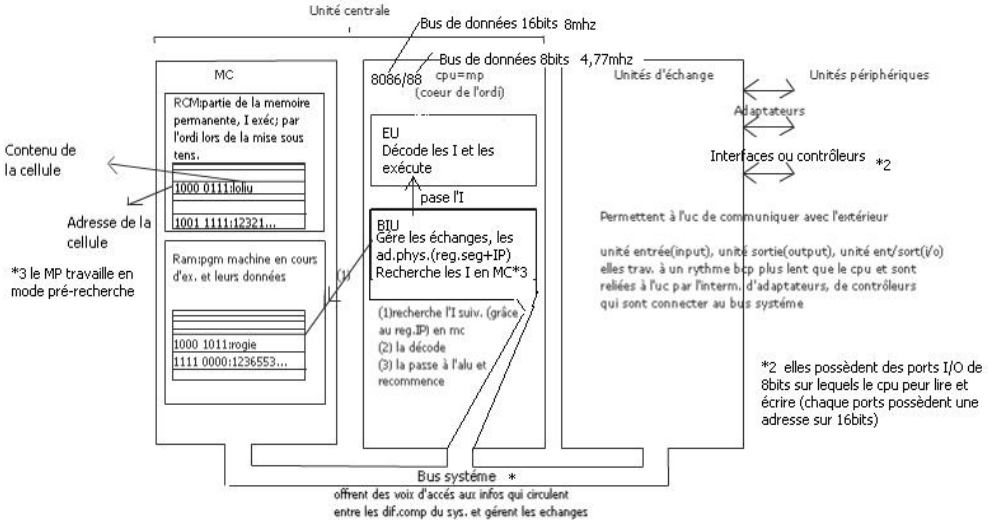
Structure d'un ordinateur



Bus d'adresses (20bits) : Utilisé par le cpu (unité centrale – Central Processor Unit) pour adresser une case mémoire ou un port I/O, le cpu calcule d'abord l'adresse physique (20bits) puis la positionne sur le bus d'adresse

Bus de données (16bits) : transfert des informations entre la mémoire et le cpu

Bus de commande : circulation des signaux de commande



Les registres

Zones mémoires processeur pour stocker des informations afin d'y accéder plus rapidement que par des accès à la mémoire centrale (RAM).

Les registres généraux

AX, BX, CX et DX sont à usage général et servent à stocker temporairement une information de 16 bits.

Chacun de ces registres est divisible en 2 registres de 8 bits. Exemple pour AX : AH = octet de poids fort de AX. AL = octet de poids faible de AX.

AX (accumulateur) : instructions d'I/O et certaines opérations arithmétiques.

BX (registre de base) : peut être aussi utilisé comme registre d'adressage lors de l'adressage direct à la mémoire.

CX (compteur) : peut être aussi utilisé comme compteur de boucles

DX : peut être aussi utilisé pour contenir les adresses des ports pour les instructions I/O.

Les registres d'index et pointeurs

SI et DI peuvent être utilisés comme registres généraux de données mais le plus souvent sont utilisés avec les instructions spécialisées de manipulation des chaînes de caractères et aussi comme registres d'adressage (indexé).

SP et BP accèdent aux données de la pile (SS : SP = sommet de la pile et BP accède à des données dans la pile).

CS (Code Segment) et IP pointeur d'instruction : CS : IP = adresse de l'instruction suivante à exécuter. (CS contient la partie haute de l'adresse)

Les registres de segments

CS, DS, ES et SS permettent de calculer l'adresse Physique d'une donnée (en spécifiant la partie haute de l'adresse) à partir de son adresse logique

Le registre des indicateurs

FR est un masque de 16 bits dont 9 seulement sont significatifs; ils représentent à tout moment l'état logique du cpu (processeur – Central Processor Unit) et décrivent la manière dont se sont déroulées certaines opérations.

Flags de contrôle qui modifient le fonctionnement du cpu

IF quand il est à 1 le cpu peut être modifié par des événements extérieurs

TF quand il est à 1 le cpu fonctionne en mode pas à pas

DF permet de modifier la mise à jour des registres d'index lors des opérations de manipulation de chaînes de caractères

Flags indicateurs d'état

CF à 1 il indique la retenue lors de la dernière opération arithmétique

PF à 1 nombre de bits pair

SF à 1 résultat négatif (entier signé)

ZF à 1 résultat nul ou opérandes égaux

OF à 1 il indique qu'il y a eu débordement

AF à 1 indique une retenue

Plus d'informations

http://www.google.fr/url?sa=t&source=web&cd=1&ved=0CB8QFjAA&url=http%3A%2F%2Fentraide-epfc.sirenacorp.be%2Fscript.redirDownload.php%3FID_download%3D103%26url%3Ddownload%2F2007_01_12_Rsumerdassembleur.doc&rct=j&q=architecture%208086&ei=4VQOTdySMJGo8QPBvM2DBw&usg=AFQjCNFof1l86I105-h5-N_DvadhXay-xg&cad=rja

Complément de cours 2 : Jeu d'instructions 80x86

I) introduction

On peut diviser les instructions du 8086/88 en 6 groupes comme suit :

- Instructions de transfert de données.
- Instructions arithmétiques.
- Instructions de bits (logiques).
- Instructions de sauts de programme.
- Instructions de chaîne de caractères.
- Instructions de contrôle de processus.
- Instructions d'interruptions.

II) Les instructions de transfert de données

Elles sont divisées en 4 sous- groupes comme le montre le tableau suivant :

Usage	Nom	Fonction
Général	MOV	Transfert d'octets ou de mots
	PUSH	Chargement de la pile
	POP	Déchargement de la pile
	PUSHA	Chargement de tous les registres dans la
	POPA	pile Déchargement de tous les registres
	XCHG	dans la pile Echange d'octet ou de mot
	XLAT	Translation d'octet
Entrées-sorties	IN	Entrée de mot ou d'octet
	OUT	Sortie de mot ou d'octet
Adresses	LEA	Chargement de l'adresse effective
	LDS	Chargement du pointeur avec DS
	LES	Chargement du pointeur avec ES
Indicateurs	LAHF	Transfert des indicateurs dans AH
	SAHF	Rangement de AH dans les indicateurs
	PUSHP	Chargement des indicateurs dans la pile
	POPF	Déchargement des indicateurs de la pile.

II-1) Les instructions d'usage général

II-1-1) MOV

Copie de données (un octet ou un mot) d'un registre vers un autre registre ou d'un registre vers une case mémoire, sa syntaxe est comme suit :

Exemples

```
Syntaxe :  MOV destination, source
```

```
MOV   AX, BX     ; Copie du contenu d'un registre 16 bits vers un registre 16 Bits :  (BX) → AX
MOV   AH, CL     ; Copie du contenu d'un registre 8 bits vers un registre 8 bits :    (CL) → AH
MOV   AX, Val1   ; Copie du contenu d'une case mémoire 16 bits vers AX :              (Val1) → AX
MOV   Val2, AL   ; Copie du contenu de AL vers une case mémoire d'adresse Val2 :      (AL) → Val2
```

Remarques

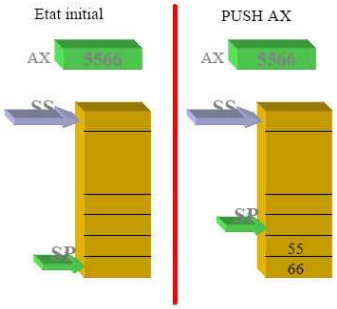
- Il est strictement interdit de transférer le contenu d'une case mémoire vers une autre case mémoire
- On n'a pas le droit aussi de transférer un registre segment vers un autre registre segment sans passer par un autre registre.

II-1-2) PUSH

Elle permet d'empiler les registres du CPU sur le haut de la pile

```
Syntaxe : PUSH  SOURCE
```

Exemple :

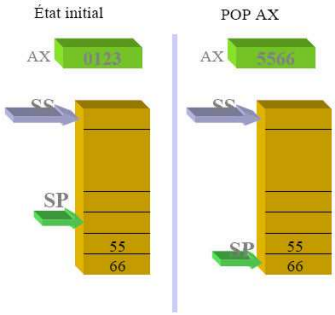


II-1-3) POP

Elle permet de dépiler les registres du CPU sur le haut de la pile

```
Syntaxe : POP  destination
```

Exemple



III) Instructions arithmétiques

Les instructions arithmétiques peuvent manipuler quatre types de nombres :

- Les nombres binaires non signés
- Les nombres binaires signés.
- Les nombres décimaux codés binaires (DCB), non signés.
- Les nombres DCB non condensés, non signés.

Les instructions arithmétiques sont divisées en quatre sous-groupes comme le montre le tableau suivant :

Usage	Nom	Fonction
Addition	ADD	Addition sur un octet ou un mot
	ADC	Addition sur un octet ou un mot avec retenue
	INC	Incrémentation de 1
	AAA	Ajustement ASCII
	DAA	Ajustement décimal
Soustraction	SUB	Soustraction sur un octet ou un mot
	SBB	Soustraction sur un octet (mot) avec retenue
	DEC	Décrémentation de 1
	NEG	Mètre un octet ou un mot en négatif
	CMP	Comparaison d'octet ou mot
	AAS	Ajustement ASCII
	DAS	Ajustement décimal
Multiplication	MUL	Multiplication d'octet ou de mot **non signée**
	IMUL	Multiplication d'octet ou de mot **signée**
	AAM	Ajustement ASCII
Division	DIV	Division d'octet ou de mot **non signée**
	IDIV	Division d'octet ou de mot **signée**
	AAD	Ajustement ASCII
	CBW	Conversion d'un octet en un mot
	CWD	Conversion d'un mot en double mots

III-1) Addition

III-1-1) ADD: (Addition)

```
Syntaxe :    ADD   Destination, source
```

Elle permet d'additionner le contenu de la source (octet ou un mot) avec celui de la destination le résultat est mis dans la destination

```
  Destination <---------- Destination  +  source
```

Exemples

```
ADD AX, BX  ; AX = AX + BX (addition sur 16 bits) ADD  AL,BH
            ; AL = AL + BH (addition sur 8 bits )
ADD AL, [SI]  ; AL = AL + le contenu de la case mémoire pointée par SI
ADD [DI], AL ; le contenu de la case mémoire pointé par DI est additionnée avec AL, le résultat est mis
             ; dans la case mémoire pointée par DI
```

III-1-3) INC : (Incrémentation)

```
Syntaxe :    INC    Destination
```

Elle permet d'incrémenter le contenu de la destination : Destination <---------- Destination + 1

Exemples

```
INC AX     ; AX = AX +  1 (incrémentation sur 16  bits).
INC AL     ; AL = AL +1 (incrémentation sur 8 bits).
INC [SI]   ; [SI] = [SI] + 1 le contenu de la case mémoire pointée par SI sera incrémenté
```

III-2) Soustraction

III-2-1) SUB : (Soustraction)

```
Syntaxe :    SUB   Destination, source
```

Elle permet de soustraire la destination de la source (octet ou un mot) le résultat est mis dans la destination

```
  Destination <---------  Destination  --  source
```

Exemples

```
SUB AX,BX    ; AX = AX - BX (Soustraction sur 16 bits )
SUB  AL,BH   ; AL = AL - BH ( Soustraction sur 8 bits )
SUB AL,[SI]  ; AL = AL - le contenu de la case mémoire pointé par SI
SUB [DI],AL  ; le contenu de la case mémoire pointée par DI est soustraite de AL , le résultat est mis
             ; dans la case mémoire pointée par DI
```

III-2-3) DEC : (Décrémentation)

```
Syntaxe :    DEC   Destination
```

Elle permet de décrémenter le contenu de la destination Destination <---------- Destination - 1

Exemples

```
DEC AX    ; AX = AX -  1 (décrémentation sur 16  bits).
DEC  AL   ; AL = AL -1 (décrémentation sur 8 bits).
DEC [SI]  ; [SI] = [SI] - 1 le contenu de la case mémoire pointée par SI sera décrémenté
```

III-2-5) CMP : (Comparaison)

```
Syntaxe :    CMP  Destination   , Source
```

Elle soustrait la source de la destination , qui peut être un octet ou un mot , le résultat n'est pas mis dans la destination , en effet cette instruction touche uniquement les indicateurs pour être tester avec une autre instruction ultérieure de saut conditionnel
Les indicateurs susceptibles d'être touché sont : AF, CF, OF, PF, SF, ZF
Donc cette instruction va nous permettre de comparer deux nombres comme le montre le tableau suivant :

	Opérande non signé				Opérande signé			
	OF	SF	ZF	CF	OF	SF	ZF	CF
Source < destination	-	-	0	0	0/1	0	0	-
Source = destination	-	-	1	0	0	0	1	-
Source > destination	-	-	0	1	0/1	1	0	-

III-3) La multiplication :

III-3-1) MUL : (Multiplication pour les nombres non signés)

MUL effectue une multiplication non signée de l 'opérande source avec l'accumulateur :

```
Syntaxe : MUL   Source
```

- Si la source est un octet alors elle sera multipliée par l'accumulateur AL le résultat sur 16 bits sera stocké dans le registre AX.
- Si la source est un mot alors elle sera multipliée avec l'accumulateur AX le résultat de 32 bits sera stocké dans la paire des registres AX et DX

En conclusion :

Multiplication	Opérande 1	Opérande 2	Résultat
Octet x Octet	AL	Registre ou memoire	AX
Mots x Mots	AX	Registre ou memoire	DX AX
Mots x Octet	AL= Octet, AH=0	Registre ou memoire	DX AX

III-4) La division

III-4-1) DIV : (Division des nombres non signés)

```
Syntaxe :   DIV  Source
```

Elle effectue une division non signée de l'accumulateur par l'opérande source :

Exemples

- Si l'opérande est un octet : alors on récupère le quotient dans le registre AL et le reste dans le registre AH.
- Si l'opérande est un mot : alors on récupère le quotient dans le registre AX et le reste dans le registre DX

a)

```
MOV AH,00h
MOV AL,33H
MOV DL,25H
DIV DL
```

b)

```
MOV AX,500H
MOV CX,200H
DIV CX
```

IV) Les instructions logiques (de bits)

Ils sont divisés en trois sous-groupes comme le montre le tableau suivant :

Usage	Nom	Fonction
Logique	NOT	Inversion logique sur un octet ou un mot
	AND	Et logique
	OR	Ou logique
	XOR	Ou exclusif
	TEST	Et logique sans résultat, affecte uniquement les indicateurs du registre des flags.
Décalages	SHL	Décalage logique à gauche
	SAL	Décalage arithmétique à gauche
	SHR	Décalage logique à droite
	SAR	Décalage arithmétique à droite
Rotation	ROL	Rotation à gauche
	ROR	Rotation à droite
	RCL	Rotation à gauche à travers le bit de retenue
	RCR	Rotation à droite à travers le bit de retenue

Syntaxe des instructions de rotation et de décalage :

Exemple : Décalage logique à droite

```
SHR  destination, compteur
```

Exemple

```
SHR AX,1          ; décale le contenu de AX, noté (AX) d'un cran vers la droite :
                  ; réalise une division par 2 de (AX)
                  ; l'ancien LSB est perdu à cause du décalage; le nouveau MSB est un 0.
```

V-1) Branchement inconditionnel

V-1-1) CALL : notion de procédure :

La notion de procédure en assembleur correspond à celle de fonction en langage C, ou de sous-programme dans d'autres langages.

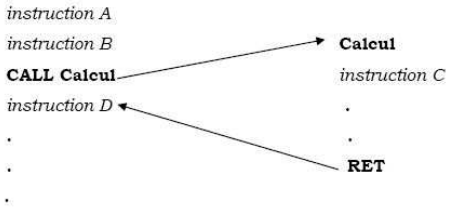


FIG. – Appel d'une procédure.

La procédure est nommée calcul. Après l'instruction B, le processeur passe à l'instruction C de la procédure, puis continue jusqu'à rencontrer RET et revient à l'instruction D.

Une procédure est une suite d'instructions effectuant une action précise, qui sont regroupées par commodité et pour éviter d'avoir à les écrire à plusieurs reprises dans le programme.

Les procédures sont repérées par l'adresse de leur première instruction, à laquelle on associe une étiquette en assembleur.

L'exécution d'une procédure est déclenchée par un programme *appelant*. Une procédure peut elle-même appeler une autre procédure, et ainsi de suite.

Instructions CALL et RET

L'appel d'une procédure est effectué par l'instruction CALL.

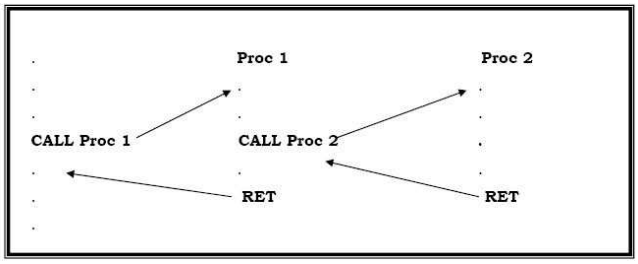
CALL *adresse_debut_procedure*

L'adresse est sur 16 bits, la procédure est donc dans le même segment d'instructions. CALL est une nouvelle instruction de branchement inconditionnel. La fin d'une procédure est marquée par l'instruction RET :

V-1-2) RET :

RET ne prend pas d'argument ; le processeur passe à l'instruction placée immédiatement après le CALL.

RET est aussi une instruction de branchement : le registre IP est modifié pour revenir à la valeur qu'il avait avant l'appel par CALL. Comment le processeur retrouve-t-il cette valeur ? Le problème est compliqué par le fait que l'on peut avoir un nombre quelconque d'appels imbriqués, comme sur la figure suivante :



L'*adresse de retour*, utilisée par RET, est en fait sauvegardée sur la pile par l'instruction CALL. Lorsque le processeur exécute l'instruction RET, il dépile l'adresse sur la pile (comme POP), et la range dans IP.

L'instruction CALL effectue donc les opérations :

- Empiler la valeur de IP. A ce moment, IP pointe sur l'instruction qui suit le CALL.
- Placer dans IP l'adresse de la première instruction de la procédure (donnée en argument).

Et l'instruction RET :

- Dépiler une valeur et la ranger dans IP.

Remarque 1 :

Si la procédure appartient au même segment que le programme principal elle est dite de type NEAR sinon elle est dite de type FAR, la différence entre eux c'est que dans le premier cas le processeur doit empiler une seule valeur dans la pile c'est le registre IP mais dans le deuxième cas il faut empiler le registre IP ainsi que le registre segment CS et bien sur il les dépiler pendant le retour de la procédure.

Remarque 2 : Passage de paramètres

En général, une procédure effectue un traitement sur des données
(*paramètres*) qui sont fournies par le programme appelant, et produit un résultat qui est transmis à ce programme. Plusieurs stratégies peuvent être employées :

1. *Passage par registre* : les valeurs des paramètres sont contenues dans des registres du processeur. C'est une méthode simple, mais qui ne convient que si le nombre de paramètres est petit (il y a peu de registres).

2. *Passage par la pile* : les valeurs des paramètres sont empilées. La procédure lit la pile.

V-1-3) JMP : (Saut inconditionnel)

Syntaxe : JMP cible

Si le JMP est de type NEAR alors IP = IP + Déplacement

Si le JMP est de type FAR alors CS et IP sont remplacé par les nouvelles valeurs obtenues à partir de l'instruction.

JMP transfert, sans condition, la commande à l'emplacement de destination. L'opérande Cible peut être obtenu à partir de l'instruction elle- même (JMP direct) ou à partir de la mémoire ou à partir d'un registre indiqué par l'instruction.

V-2 saut conditionnel

JC : (Saut si retenue)	Si CF=1 alors IP = IP + déplacement
JE/JZ : (Saut si égal/Si zéro)	Si ZF=1 alors IP = IP + déplacement
JNC : (Saut si pas de retenue)	Si CF=0 alors IP = IP + déplacement
JNE/JNZ : (Saut si non égal) Non zéro)	Si ZF=0 alors IP = IP + déplacement
JNO : (Saut si pas de débordement)	Si OF=0 alors IP = IP + déplacement
JNP/JPO : (Saut si pas de parité/ Si parité impaire)	Si PF=0 alors IP = IP + déplacement
JNS : (Saut si pas de signe)	Si SF=0 alors IP = IP + déplacement
JO : (Saut si débordement)	Si OF=0 alors IP = IP + déplacement
JP/JPE: (Saut si parité (paire))	Si PF=1 alors IP = IP + déplacement
JS : (Saut si signe (négatif))	Si SF=1 alors IP = IP + déplacement

V-3) Les instructions de boucle

V-3-1) LOOP : (boucle) :

Elle décrémente le contenu de CX de 1. Si (CX) ≠ 0 alors IP = IP + déplacement. Si (CX) = 0 l'instruction suivante est exécutée.

Exemple :

```
        MOV  AX, 05
        MOV  CX, 05
Début : INC AX                 <-- SI CX<>0
SI CX=0 LOOP Début
        MOV BX, AX
```

L'exécution de l'instruction MOV BX, AX sera faite après l'exécution de la boucle 5 fois.

V-3-2) LOOPE / LOOPZ : (boucle si égale ou si égale à zéro) : Le registre CX est décrémenter de 1 automatiquement

Si CX est différent de zéro et ZF=1 alors IP = IP + déplacement

Exemple :

```
        MOV AX, 03
        MOV CX, 6
DEBUT:  ADD DH, DL
        INC AX                 <-- SI CX<>0 Et AL=06
        CMP AL, 06
SI CX=0 LOOPZ DEBUT
Ou AL<>06
        MOV AH, 03
```

V-3-3) LOOPNE / LOOPNZ : (boucle si égale ou si égale à zéro) : Le registre CX est décrémenter de 1 automatiquement

Si CX est différent de zéro et ZF=0 alors IP = IP + déplacement

Exemple :

```
        MOV AX, 03
        MOV CX, 6
DEBUT:  ADD DH, DL
        INC AX                 <-- SI CX<>0 Et AL<>06
        CMP AL, 06
SI CX=0 LOOPZ DEBUT
Ou AL=06
        MOV AH, 03
```

VII) Les instructions de commande du processeur

Ces instructions agissent sur le processeur et ses indicateurs (Flags) ils sont en nombre de 12 comme le montre le tableau suivant

Type	Nom	Fonction
Indicateur (FLAGS)	STC	Met à 1 la retenue CF
	CLC	MET à 0 la retenue CF
	CMC	Complémente la retenue
	STD	Met à 1 la direction DF
	CLD	Met à 0 la direction DF
	STI	Met à 1 l'autorisation d'interruption
	CLI	Met à 0 l'autorisation d'interruption
Synchronisation	HLT	Halte jusqu'à interruption ou RESET
	WAIT	Attente jusqu'à broche TEST passe à 0
	ESC	Pour un coprocesseur
	LOCK	Verrouillage des bus pendant la prochaine instructions
Sans opération	NOP	Pas d'opération

VII-1) Indicateurs :

VII-1-1/ STD :
Met CF à 1 ; les registres d'indexation SI et/ou DI sont alors automatiquement décrémenter par les instructions de chaîne de caractère.

VII-1-2) STI :
Met IF à 1, permettant ainsi au CPU de reconnaître des demandes d'interruption masquables apparaissant sur la ligne d'entrée INTR.

VII-2) Synchronisation :

VII-2-1) HALT :
Maintient le processeur dans un état d'attente d'un RESET ou d'une interruption externe non masquable ou masquable (avec IF=1).

VII-2-2) WAIT :
Met le CPU en état d'attente tant que sa ligne de TEST n'est pas active. En effet toutes les cinq périodes d'horloge le CPU vérifie est ce que cette entrée est active ou non, si elle est active le processus exécute l'instruction suivante à WAIT.

VII-2-3) ESC :
L'instruction Escape fournit un mécanisme par lequel des coprocesseurs peuvent recevoir leurs instructions à partir de la suite d'instructions du 8086.

VII-2-4) LOCK :
Elle utilise dans les systèmes Multiprocesseur en effet elle permet le verrouillage du bus vis-à-vis des autres processeurs.

VII-3 Sans opération :

VII-3-1) NOP (No operation) :
Le CPU ne fait rien on peut s'en servir pour créer des temporisations.

Plus d'informations :

http://www.technologuepro.com/microprocesseur/chap4_microprocesseur.htm

Complément de cours 3 : Interruptions système

Interruptions DOS

.Lecture d'un caractère (unique) au clavier avec écho *(avec écho signifie que le caractère tapé au clavier apparaît sur l'écran après la frappe)*

Pas de nécessité de validation avec la touche "Entrée" : le caractère est acquis dès sa frappe au clavier.
Lecture d'un caractère frappé au clavier; après appel de l'interruption, le code ASCII du caractère est stocké dans le registre AL.

Paramètre d'entrée : AH = 1
Appel d'interruption : INT 21H
Paramètre de sortie : AL ; (AL) = code ASCII du caractère

Interruptions BIOS

.Affichage d'un caractère à l'écran

Paramètres d'entrée : AH = 0EH
AL = Code ASCII du caractère a afficher à l'écran
Appel d'interruption : INT 10H
Paramètres de sortie : aucun

.Affichage d'une chaîne de caractères à l'écran
…

Complément de cours 4 : Codage ASCII

Codage ASCII des caractères

Le code ASCII du caractère '0' est 30h
Le code ASCII du caractère '1' est 31h
Le code ASCII du caractère '2' est 32h
...
Le code ASCII du caractère '9' est 39h

d'où l'algorithme de conversion *chiffre* $\rightarrow$ *code ascii* : **code ascii = chiffre + 30h**.

Exercice 1 : Programmation ASSEMBLEUR (ASM)

a) Installer le simulateur 8086 : Assembler with Microprocessor Simulator 8086 (fichier **emu8086v408r.exe**) téléchargeable à l'adresse :

http://pcwin.com/Software_Development/Debugging/Assembler_with_Microprocessor_Simulator_8086/index.htm **(rubrique *download*)**

b) A l'aide de l'émulateur 8086 (menu *emulate*) et des compléments de cours, réaliser et vérifier le fonctionnement des programmes suivants : (il est conseillé d'utiliser les registres généraux AX, BX, CX, DX ainsi que la pile)

0. Calcul de N ! (N<4) avec Entrées/Sorties **(E/S)**

Calcul de la somme S des N premiers entiers

1- Sans Entrées/Sorties : 0 < N < 100 (N entier)

1a- Par itération : S = somme de i (de i = 1 à N) : chargement de N en dur et visualisation du résultat via l'émulateur

1b- Par formule de Gauss : S = N(N+1)/2 : chargement de N en dur et visualisation du résultat via l'émulateur

Test : N = 99; S = 4950 = 1356h

2- Avec Entrées/Sorties : 0 < N < 4 (N entier)

2a- Par itération : S = somme de i (de i = 1 à N) : lecture de N au clavier et affichage du résultat à l'écran (utilisations des interruptions système)
Conseil : se placer en clavier Anglais pour l'acquisition de N au clavier

2b- Par formule de calcul : S = N(N+1)/2 : lecture de N au clavier et affichage du résultat à l'écran (utilisations des interruptions système)
Conseil : se placer en clavier Anglais pour l'acquisition de N au clavier

Test : N = 3; S = 6 = 0006h

3- *Facultatif :* Avec Entrées/Sorties : 0 < N < 10 (N entier)

3a- Par itération : S = somme de i (de i = 1 à N) : lecture de N au clavier et affichage du résultat à l'écran (utilisations des interruptions système)
Conseil : se placer en clavier Anglais pour l'acquisition de N au clavier

Test : N = 9; S = 45 = 002Dh

Calcul de la somme S2 des carrés des N premiers entiers

4- Calcul de la somme S2 des N premiers carrés des entiers

Sans Entrées/Sorties : N (entier) : 0 < N < 9 :

4a- Par itération : S2 = somme de (i*i) (de i = 1 à N) : chargement de N en dur et visualisation du résultat via l'émulateur

4b- Par formule : S2 = N(N+1)(2N+1)/6 : chargement de N en dur et visualisation du résultat via l'émulateur

Test1 : N = 3; S2 = 14 = 000Eh
Test 2: N = 8; S2 = 204 = 00CCh

4. Les Processeurs actuels.
Pipelining. Prédiction de branchement. Caches. Processeurs superscalaires

2 principales améliorations de l'architecture de Von Neumann sont présentées :
- le pipelining
- la mémoire cache

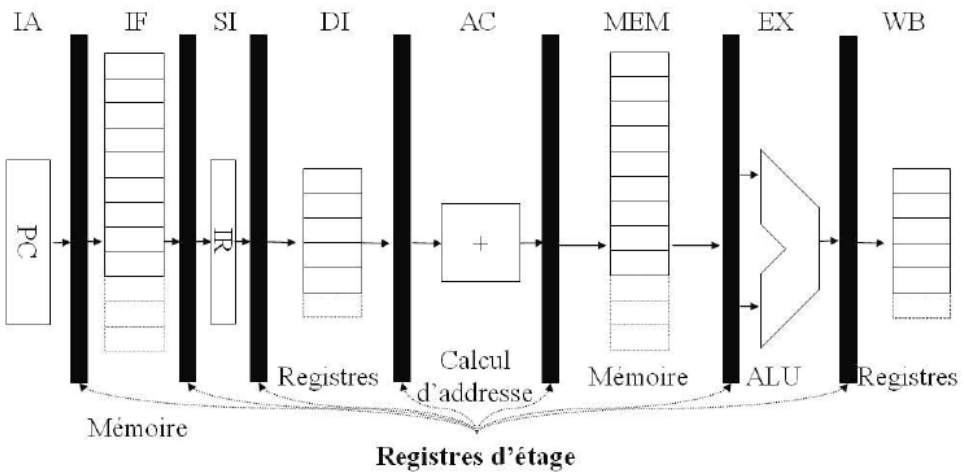
1. Pipelining

L'exécution d'une instruction est décomposée en une succession d'étapes, et chaque étape correspond à l'utilisation d'un des composants du processeur. Lorsqu'une instruction se trouve dans l'une des étapes, les composants associés aux autres étapes ne sont pas utilisés. Le fonctionnement d'un processeur simple est donc inefficace. L'exécution pipelinée des instructions permet d'améliorer l'efficacité d'un processeur : on charge une première instruction dans la première étape, et au cycle suivant, cette instruction passe dans la seconde étape, on charge alors une seconde instruction dans la première étape, et ainsi de suite... En régime permanent, il peut y avoir une instruction en cours d'exécution dans chacune des étapes, et donc chacun des composants du processeur peut être utilisé à chaque cycle, l'efficacité est maximale. Le temps d'exécution d'une instruction n'est pas réduit, en revanche, le débit de sortie des instructions est considérablement augmenté : jusqu'à une instruction exécutée par cycle ; sur un processeur comportant cinq étapes d'exécution, pipeliner l'exécution permet donc de multiplier la performance par cinq.

Implémentation du pipeline. Pour obtenir une version pipelinée d'un processeur comme le LC-2, on peut physiquement découper le processeur en étape/étage en séparant les composants les uns des autres à l'aide de registres, appelés **registres d'étage**. Chaque composant traitant une instruction différente des autres composants, il doit disposer de l'ensemble des informations nécessaires à l'exécution de l'étape pour cette instruction (opcode, bits de contrôle, opérandes...). Le contrôle devient alors décentralisé au niveau de chaque étage. Pipeliner un processeur ayant pour but d'accroître sa performance, on utilise un contrôle câblé plutôt qu'un contrôle microprogrammé.

Exemple. Pour le LC-2, on peut par exemple découper l'exécution d'une instruction en 8 étapes, comme indiqué ci-dessous :

1. Envoi adresse instruction (IA)
2. Chargement instruction (IF)
3. Stockage instruction (SI)
4. Décodage; lecture des opérandes (DI)
5. Calcul d'adresse (AC)
6. Accès mémoire (ME)
7. Exécution (EX)
8. Ecriture du résultat (WB)



On pourra cependant noter que l'étape IA est inutile puisque le PC est déjà un registre, et que l'étape SI est également inutile puisque chaque registre d'étage joue le rôle de registre instruction en stockant toutes les informations nécessaires à l'exécution de l'instruction.

Dans les processeurs récents, l'exécution pipelinée a un autre avantage : puisqu'il est possible de découper l'exécution d'une instruction en étapes sans affecter la performance, bien au contraire, on peut réduire la durée d'une étape en augmentant le nombre d'étapes. Cela permet de réduire le temps de cycle du processeur, et donc d'augmenter le débit de sortie des instructions. On va voir cependant que plus le pipeline est long, et plus le nombre de cas où il n'est pas possible d'atteindre la performance maximale est élevé.

Il existe trois principaux cas où la performance d'un processeur pipeliné peut être dégradée ; ces cas de dégradation de performance sont appelés des **aléas** (***pipeline hazard***) :

- aléas structurels
- aléas de données
- aléas de contrôle

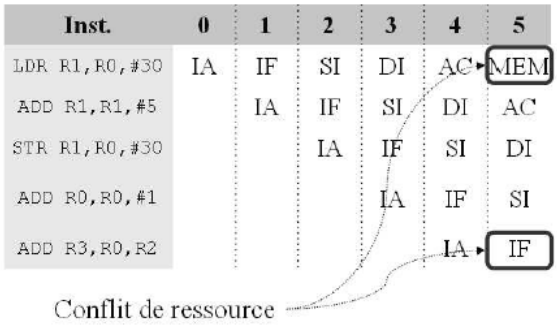
Lorsqu'un aléa se produit, cela signifie qu'une instruction ne peut continuer à progresser dans le pipeline. Pendant un ou plusieurs cycles, l'instruction va rester bloquée dans un étage du pipeline, mais les instructions situées plus en avant pourront continuer à s'exécuter jusqu'à ce que l'aléa ait disparu. Les étages du pipeline vacants sont appelés des « bulles » de pipeline, une bulle correspondant en fait à une instruction NOP (*No OPeration*). En pratique, le registre d'un étage bloqué émet effectivement une instruction NOP à la place de l'instruction bloquée.

Exemple. On suppose que Instr2 *est bloquée pendant deux cycles à cause d'un aléa.*

Inst 1	IA	IF	SI	DI	AC	MEM	EX	WB			
Inst 2		IA	IF	SI	DI	●	●	AC	MEM	EX	WB

Aléas structurels. Un aléa structurel correspond au cas où deux instructions ont besoin d'utiliser la même ressource du processeur.

Exemple. Dans le pipeline du LC-2, il peut y avoir un aléa structurel entre une instruction load/store et une instruction quelconque en cours de chargement : en effet, l'instruction load/store utilise la mémoire pour lire ou écrire une donnée, tandis que l'autre instruction doit également être chargée depuis la mémoire.

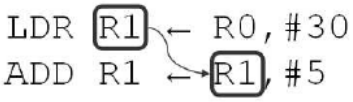


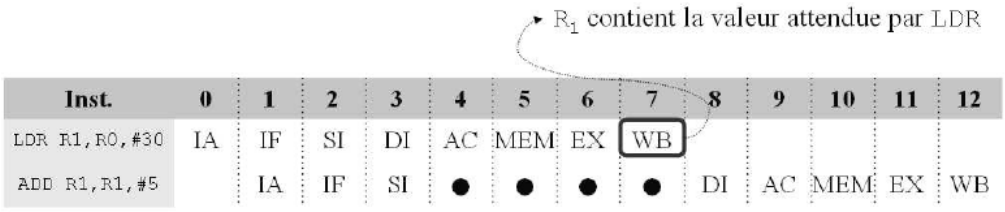
On peut résoudre cet aléa en utilisant une mémoire séparée pour les données et les instructions ; dans les processeurs actuels, c'est ce que l'on fait à l'aide des caches : il y a un cache pour les données et un cache pour les instructions.

Aléas de données. Un aléa de données intervient lorsqu'une instruction produit un résultat, et qu'une instruction suivante utilise ce résultat avant qu'il ait pu être écrit dans le banc de registres.

Un mécanisme de *bypass* permet de résoudre ce problème.

Exemple.

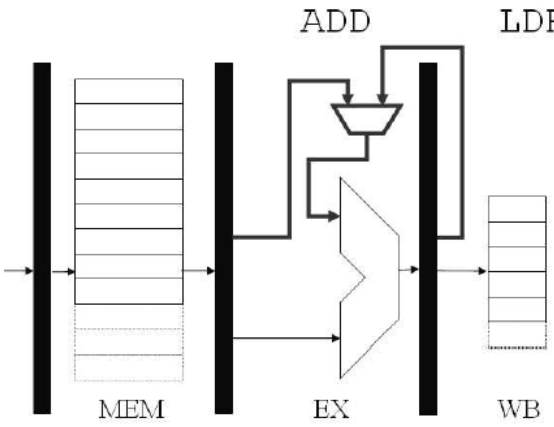




L'instruction ADD ne peut effectuer son étage DI (pendant lequel on récupère les registres sources dans le banc de registres), le registre R1 produit par l'instruction LDR n'étant disponible qu'à la fin de l'étage WB de cette instruction.

Cependant, il est souvent inefficace d'attendre qu'une donnée ait été écrite dans le banc de registres avant de l'utiliser. En effet, la donnée est souvent disponible plus tôt ; ainsi, la donnée est disponible à la fin de l'étage MEM dans l'exemple ci-dessus, et l'instruction ADD a réellement besoin de la donnée au début de l'étage EX seulement. On peut donc réduire l'impact des aléas de données en créant des chemins supplémentaires entre les différents étages du pipeline afin de passer les données plus rapidement d'une instruction à l'autre. Ce mécanisme s'appelle le ***forwarding***. Dans l'exemple ci-dessus, il permet d'éliminer les quatre cycles de gel du pipeline.

Exemple. Chemin de données à rajouter dans l'exemple ci-dessus pour implémenter le ***forwarding****.*



Aléas de contrôle. Un aléa de contrôle se produit à chaque fois qu'une instruction de branchement est exécutée. Lorsqu'une instruction de branchement est chargée, il faut normalement attendre de connaître l'adresse de destination du branchement pour pouvoir charger l'instruction suivante. Or, cette information n'étant en général connue que plusieurs cycles après le chargement de l'instruction, il est nécessaire de bloquer le pipeline.
Dans le cas d'un branchement conditionnel, il faut également connaître la valeur de la condition (branchement pris ou non pris).

Exemple. Aléa de contrôle dans le LC-2.

```
LOOP    LDR R1 ← R0,#30
        ...
        BRn LOOP
```

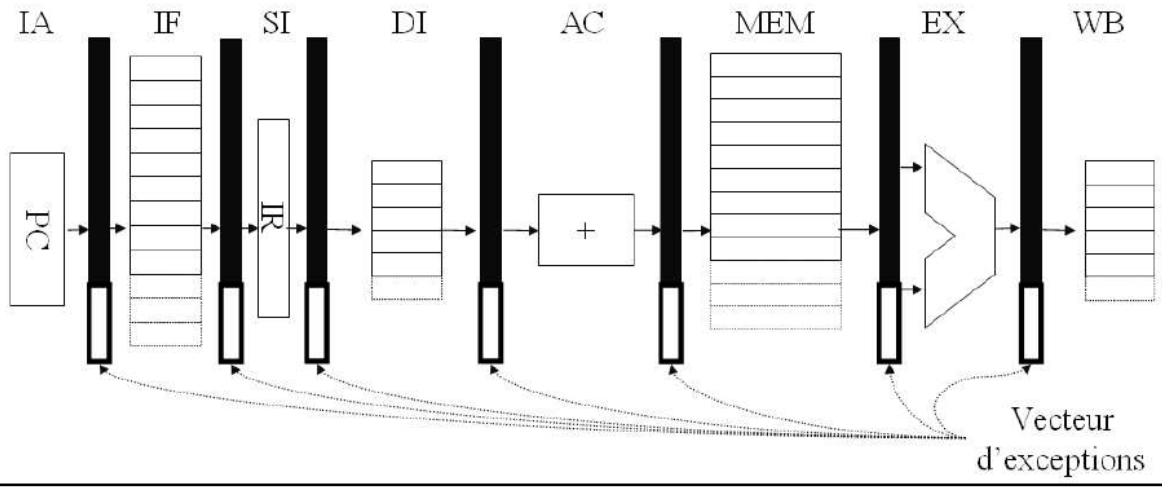
La condition (bits *n,p,z*) est lue à la fin de cette étape

L'adresse de destination du branchement est disponible à la fin de cette étape

Inst.	0	1	2	3	4	5	6	7	8	9	10	11	12
BRn LOOP	IA	IF	SI	DI	AC	MEM	EX	WB					
LDR R1,R0,#30		●	●	●	●	IA	IF	DI	AC	MEM	EX	WB	

Pipeline et exceptions. Lorsque le programme doit s'interrompre brutalement (faute dans l'exécution du programme ou interruption provoquée par le système d'exploitation), il faut pouvoir éventuellement redémarrer le programme précisément à l'endroit où il s'est arrêté, c'est-à-dire exactement à l'instruction assembleur où l'exception s'est produite ; cette propriété du couple architecture/système d'exploitation est appelée une « exception précise ». Or, dans un processeur pipeliné, lorsqu'une exception intervient pour une instruction dans l'une de ses étapes d'exécution, il y a souvent plusieurs instructions précédentes qui n'ont pas terminé leur exécution. Si le programme est redémarré à l'instruction qui a provoqué l'exception, son exécution sera incorrecte puisque les instructions précédentes n'auront pas été exécutées entièrement.
Il est donc nécessaire de retarder la gestion des exceptions jusqu'au moment où l'on sait que toutes les instructions précédentes ont terminé de s'exécuter. Pour ce faire, on ajoute aux registres d'étage un « vecteur d'exception » qui contient un bit par exception possible. Lorsqu'une instruction produit une exception, elle met à 1 le bit correspondant, et ne peut plus modifier l'état du processeur (ni écrire en mémoire, ni écrire dans le banc de registres), i.e., elle est transformée en NOP. L'exception ne sera traitée que lorsque l'instruction arrive dans l'étage final, e.g., WB dans le LC-2. À ce moment-là, les instructions précédentes auront terminé de s'exécuter.
En pratique, la présence d'instructions multi-cycles, comme les instructions de calcul flottant, rend l'implémentation des exceptions précises encore plus complexe.

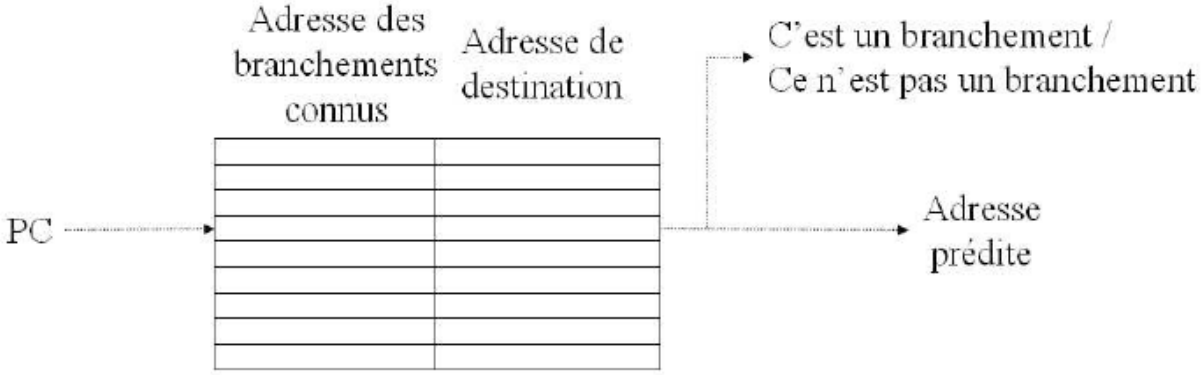
Exemple. Ajout d'un vecteur d'exceptions dans le pipeline du LC-2.



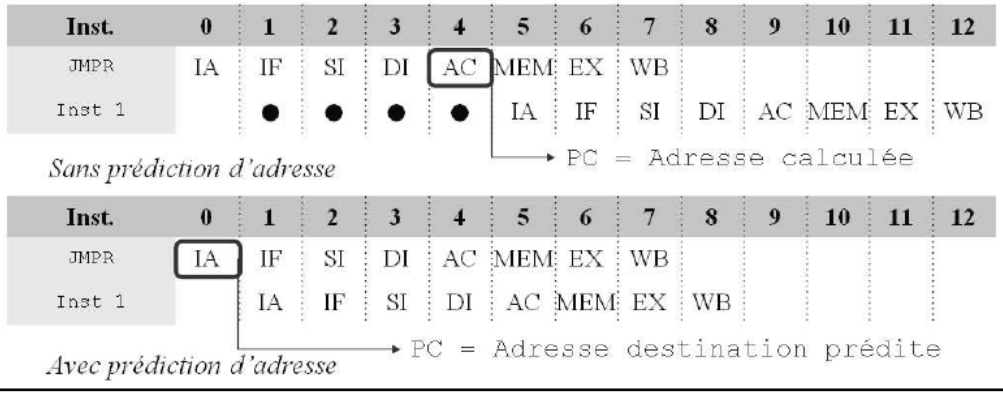
2. Prédiction de branchement

En raison de l'allongement de la taille du pipeline (liée à l'augmentation de la fréquence), le délai entre le moment où l'on charge une instruction de branchement et le moment où l'on connaît l'adresse de destination du branchement et la valeur de la condition tend à s'allonger considérablement (20 cycles sur le Pentium 4). Comme, en moyenne, il y a une instruction de branchement toutes les cinq instructions, la dégradation de performances induites par le délai de branchement serait intolérable. Aussi, depuis plusieurs années, les processeurs comportent des mécanismes de **prédiction de branchement**, destinés à prédire l'adresse de destination de branchement et la valeur de la condition pour les branchements conditionnels.

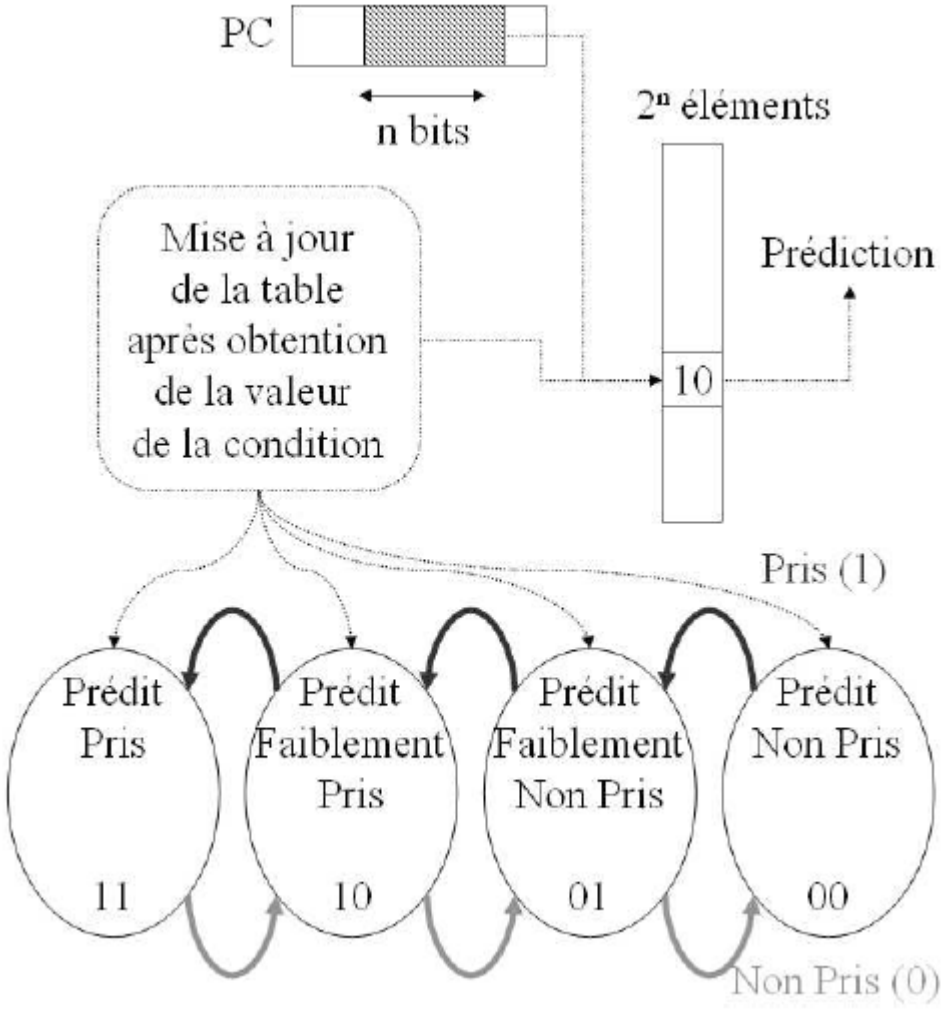
Prédiction d'adresse. La prédiction d'adresse est souvent la plus fiable parce que la plupart des instructions de branchement ont des adresses de destination fixe. Pour effectuer cette prédiction d'adresse, on ajoute au processeur une table (*Branch Target Buffer*) indexée par le PC des instructions et qui contient les adresses de destination des branchements. Lorsqu'on exécute une instruction de branchement, on stocke dans la table son adresse de destination à l'emplacement déterminé par son PC (approximativement, PC modulo la taille de la table). Lorsque l'on charge une instruction, i.e., que l'on envoie l'adresse contenue dans le PC à la mémoire, on envoie également le PC dans cette table. La table peut être lue très rapidement, contrairement à la mémoire, et en un cycle on sait si l'instruction en cours de chargement est un branchement, et si c'est le cas, on dispose également de son adresse de destination. Dès le cycle suivant, on peut donc envoyer à la mémoire l'adresse de l'instruction de destination et éviter tout gel du pipeline.
Ce mécanisme de prédiction fonctionne moins bien pour les instructions RETURN (RET dans le LC-2) car une procédure peut être appelée de plusieurs endroits différents, et l'adresse de destination de cette instruction de branchement est donc variable.



Exemple. Dans le cas du LC-2 pipeliné, la prédiction d'adresse permet d'éviter jusqu'à quatre cycles de pénalité.



Prédiction de condition. La prédiction de la condition d'un branchement conditionnel est nettement plus complexe et plus aléatoire parce que la valeur de la condition peut changer fréquemment d'une exécution du branchement à l'autre. Les mécanismes de prédiction existants ont pour but d'analyser le comportement du branchement pour détecter un éventuel comportement régulier. Les mécanismes de prédiction les plus simples effectuent une analyse uniquement locale. Comme pour la prédiction d'adresse, on dispose d'une table indexée par le PC de l'instruction de branchement. Chaque entrée de la table contient un automate à quatre états (2 bits) qui indique la prédiction courante : *pris, faiblement pris, faiblement non pris, non pris*. À chaque fois que ce branchement conditionnel est exécuté, une fois que sa condition est connue, l'automate est mis à jour.
Cette table est utilisée de la façon suivante. En plus de la table de prédiction d'adresse mentionnée ci-dessus, le PC de chaque instruction à charger est également envoyé à la table de prédiction de condition (*Branch Prediction Table*). On lit alors la valeur de la condition, et si la table de prédiction d'adresse indique que l'instruction en cours de chargement est un branchement conditionnel, on utilisera cette valeur pour déterminer si le branchement est prédit pris ou non pris.



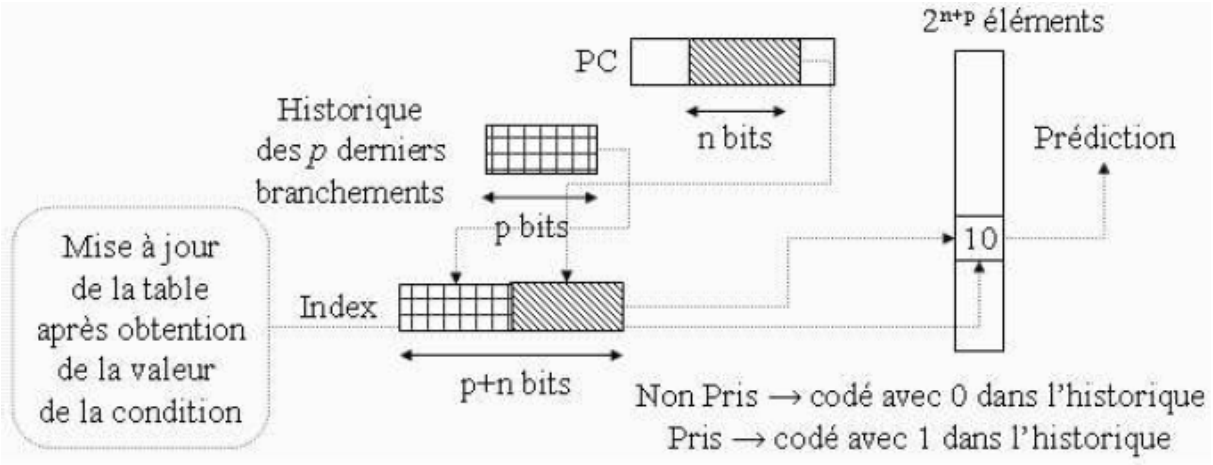
Plus le pipeline du processeur est long, plus le délai de branchement est grand et plus la qualité de la prédiction de branchement doit être élevée. Pour améliorer la précision de la prédiction de branchement, les nouveaux prédicteurs ne se contentent pas d'analyser le comportement local du branchement mais également le comportement des branchements conditionnels précédemment exécutés. En effet, le comportement des branchements conditionnels est parfois corrélé au chemin du graphe de flot de contrôle suivi par le programme.

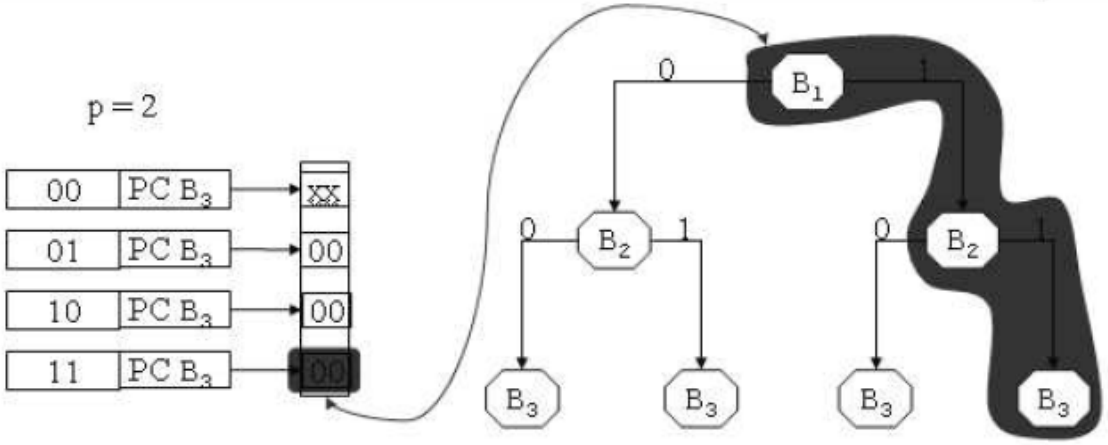
Exemple.

```
if (a == 1) a = 0;       /* Branchement B1 */
...
if (b == 0) b = 1;       /* Branchement B2 */
...
if (a == b) ...;         /* Branchement B3 */
```

La valeur de la condition du dernier branchement conditionnel est en partie déterminée par le comportement des deux branchements précédents.

Pour ce faire, les processeurs intègrent un registre à décalage donnant la valeur de la condition des N derniers branchements conditionnels (*Branch History Register*). La table de prédiction de condition est alors indexée non pas seulement par le PC, mais par une concaténation du PC et de ce registre. Intuitivement, cela signifie que l'on distingue le comportement local du branchement selon le comportement des branchements conditionnels précédents.

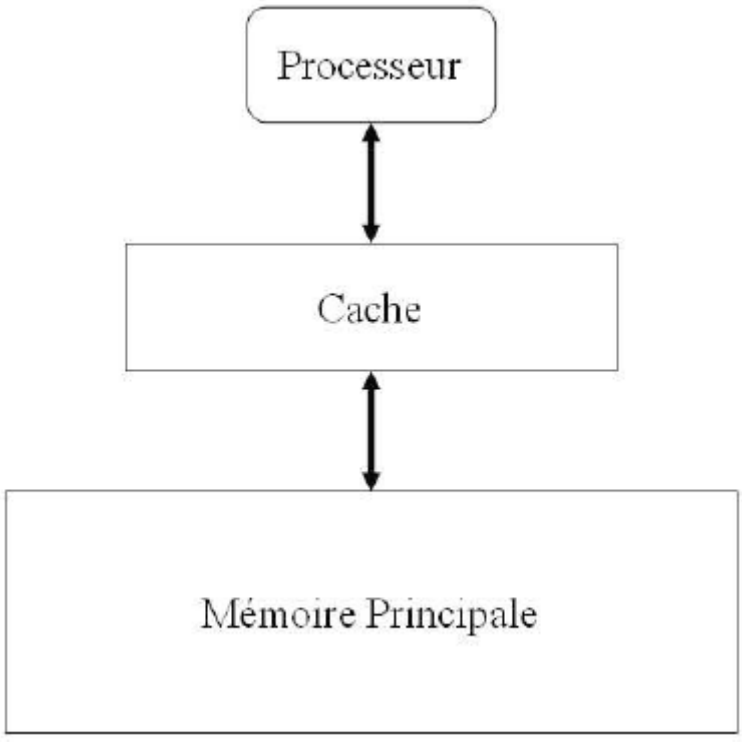




3. Caches

Depuis de nombreuses années l'écart entre la performance du processeur et celle de la mémoire tend à s'accroître. Cet accroissement s'est accéléré avec l'avènement des processeurs RISC, plus à même d'exploiter rapidement l'évolution de la technologie. Les composants mémoire bénéficient des mêmes progrès de la technologie, mais le décodage de l'adresse et le chargement des lignes communes permettant de lire ou d'écrire les différentes cellules d'une mémoire sont des étapes difficiles à accélérer. En conséquence, le temps d'accès à la mémoire décroît moins vite que le temps de cycle du processeur. Aujourd'hui, charger une information depuis la mémoire requiert de plusieurs dizaines à plusieurs centaines de cycles selon les processeurs. Sachant qu'il y a en moyenne une instruction d'accès à la mémoire toutes les trois instructions, en l'absence de mécanismes permettant de masquer cette latence d'accès à la mémoire, la performance des processeurs serait beaucoup plus faible qu'elle ne l'est aujourd'hui.

Depuis le début des années 80, le principal mécanisme architectural permettant de masquer la latence de la mémoire est le **cache**. Le principe est de placer une mémoire rapide entre le processeur et la mémoire principale. Cette rapidité, liée à une technologie différente de la mémoire principale (SRAM au lieu de DRAM) et une taille réduite, s'accompagne d'un coût plus élevé (6 transistors au lieu de 1 par cellule). Le principe du cache est très simple : le processeur n'a pas conscience de sa présence et lui envoie toutes ses requêtes comme s'il s'agissait de la mémoire principale. Soit la donnée ou l'instruction requise est présente dans le cache et elle est alors renvoyée immédiatement au processeur, soit elle n'est pas dans le cache, et le contrôleur du cache envoie alors une requête à la mémoire principale, puis renvoie la donnée au processeur et la stocke en même temps dans le cache. Le premier cas s'appelle un **succès** de cache (*cache hit*), tandis que le second cas s'appelle un **défaut** de cache (*cache miss*). En cas de défaut, le cache n'apporte bien sûr aucun gain, la performance du cache est donc entièrement liée au taux de succès.



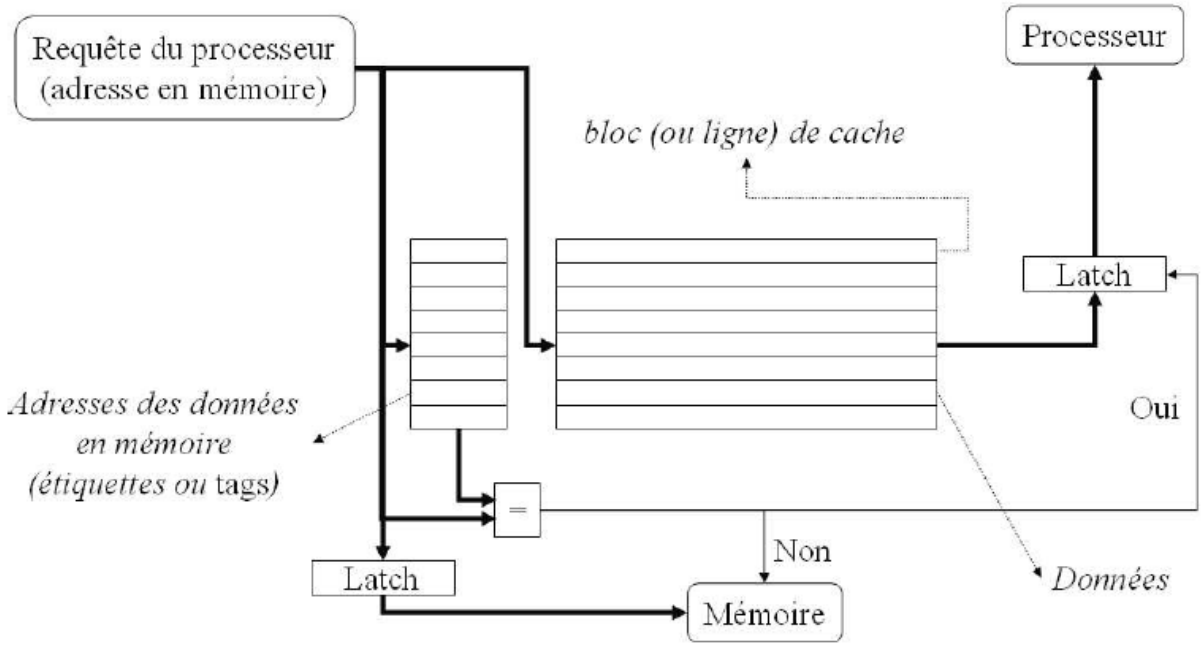
En pratique, on observe fréquemment des taux de succès moyens de l'ordre de 80 à 90 %. Cette performance s'explique par les fortes propriétés de **localité** des programmes. Il existe deux types de localités : la **localité temporelle**, et la **localité spatiale**. Une définition intuitive de la localité temporelle est la suivante : si une donnée située à une adresse *A* en mémoire est référencée, elle a une forte probabilité d'être référencée à nouveau dans un court intervalle de temps. Et pour la localité spatiale : si une donnée située à une adresse *A* en mémoire est référencée, il y a une forte probabilité de référencer une donnée située à une adresse voisine dans un court intervalle de temps.

Exemple. On peut considérer l'exemple du produit matrice-vecteur.

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    y[i] = y[i] + a[i][j] * x[j]
  }
}
```

- `y[i]` : propriétés de localités temporelle et spatiale.
- `a[i][j]` : propriétés de localité spatiale.
- `x[j]` : propriétés de localité temporelle et spatiale.

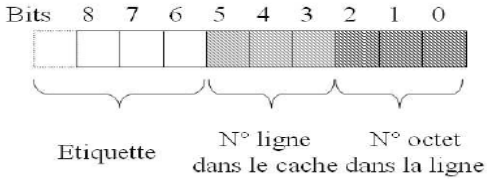
Implémentation du cache. La taille du cache étant nettement inférieure à celle de la mémoire, le cache ne pourra contenir qu'une partie des données de la mémoire. En conséquence, la position d'une donnée dans le cache ne correspond donc plus à son adresse, contrairement à ce qui se passe en mémoire principale. Avec chaque donnée stockée dans le cache, il est donc nécessaire de conserver également son adresse. Aussi, un cache comporte deux sous-composants principaux : la table des étiquettes et la table des données. La table des étiquettes stocke les adresses des données, et la table des données stocke bien sûr les données elles-mêmes.



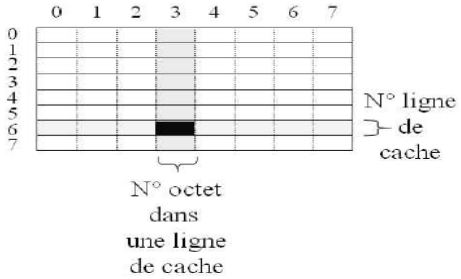
Lorsque le processeur envoie au cache l'adresse de la donnée requise, le contrôleur du cache compare cette adresse à celles stockées dans le banc d'étiquettes. Si la donnée est présente, elle est renvoyée au processeur et sinon il y a défaut de cache. Afin de réduire le temps de cette comparaison, le contrôleur de cache utilise une politique de placement simple : grossièrement, une donnée située à une adresse *A* est placée dans la case du cache *A modulo Taille du Cache*. En d'autres termes, il existe un seul emplacement possible pour une donnée dans le cache, et il y a donc une seule comparaison à effectuer lors d'une requête du processeur (on verra que cette contrainte peut avoir un impact sur la performance du cache et on peut la relâcher au prix d'une complexité supérieure de l'implémentation du cache). La simple présence d'une donner dans le cache permet d'exploiter la localité temporelle. Pour exploiter la localité spatiale, en cas de défaut de cache, on ne charge pas seulement la donnée requise mais cette donnée et plusieurs des données voisines. Les données chargées simultanément forment une **ligne** ou un **bloc** du cache (les deux termes sont utilisés indifféremment).

Plus précisément, lorsque le processeur fait une requête il envoie deux informations : l'adresse du premier octet de la donnée requise et le nombre d'octets à charger. L'adresse est alors décomposée en trois parties par le contrôleur du cache : les bits de poids faible correspondent à l'emplacement de l'octet à l'intérieur d'une ligne du cache ; les bits suivants donnent le numéro de la ligne du cache (l'index des tables) ; les bits de poids fort restants correspondent donc à l'adresse de la ligne (tout comme on avait défini la notion «d'adresse de page» pour le système exploitation), et ce sont ces bits qui sont stockés dans la table des étiquettes.

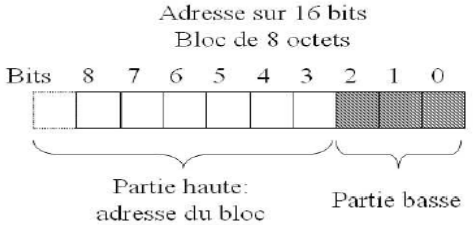
Exemple. On considère un cache de 64 octets agencé en huit lignes de huit octets chacune. L'adresse d'une requête du processeur se décompose ainsi :



Le contrôleur du cache commence à retrouver la ligne dans le cache :

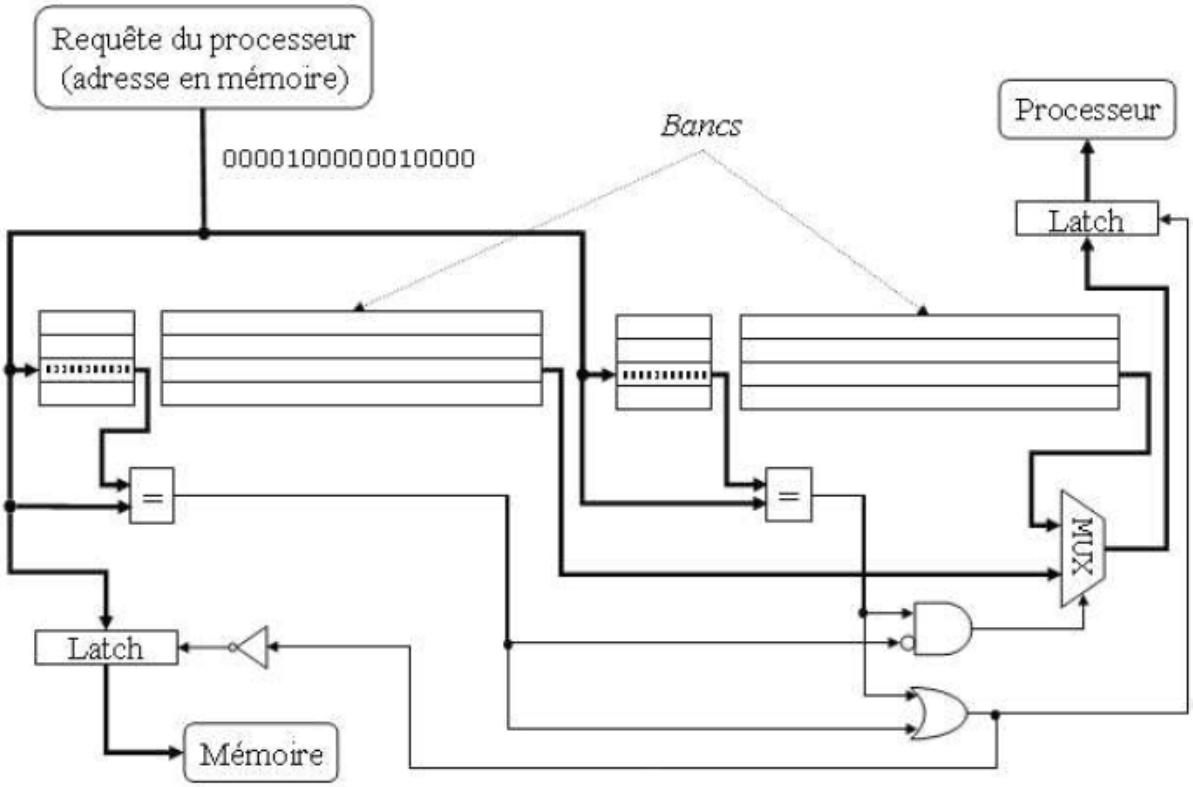


puis l'octet dans la ligne :



Exemple:
0...010100000
0...010100111
0...010101000
Même ligne de cache
Lignes distinctes, adresses consécutives

Associativité. En raison de la politique de placement dans les caches, il peut arriver que deux données soient en compétition pour la même ligne de cache bien que les autres lignes ne soient pas utilisées. Il suffit pour cela que les bits de l'adresse permettant de déterminer la ligne du cache soient les mêmes pour les deux données. On parle alors de **conflit de cache**. Pour limiter le nombre de conflits de cache, plusieurs processeurs utilisent des caches **associatifs**. Dans un cache associatif, on divise la table des données et celle des étiquettes en *n* bancs. La politique de placement dans un banc est la même que dans un cache, mais une donnée peut se trouver dans n'importe où lequel des *n* bancs. On dit alors que le degré d'associativité du cache est *n* (*n-way associative cache*).

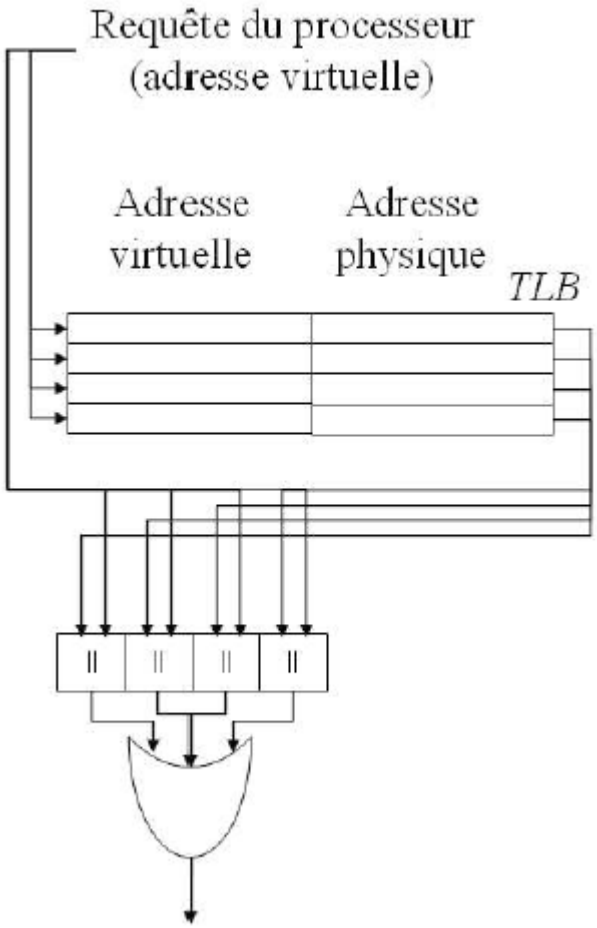


La politique de placement dans un cache est donc légèrement plus complexe : lorsque l'on charge une nouvelle ligne, il faut déterminer dans quel banc la placer. Les caches associatifs comportent donc une politique de remplacement destinée à choisir la donnée à éjecter du cache. Les politiques les plus classiques sont : *Random* (choix aléatoire), *LRU (Least Recently Used* : on choisit la ligne la plus anciennement utilisée), *Pseudo-LRU* (on n'éjecte pas la ligne la plus récemment utilisée, puis on fait un choix aléatoire entre les autres lignes).

Translation Lookaside Buffer (Cache de traduction d'adresses)

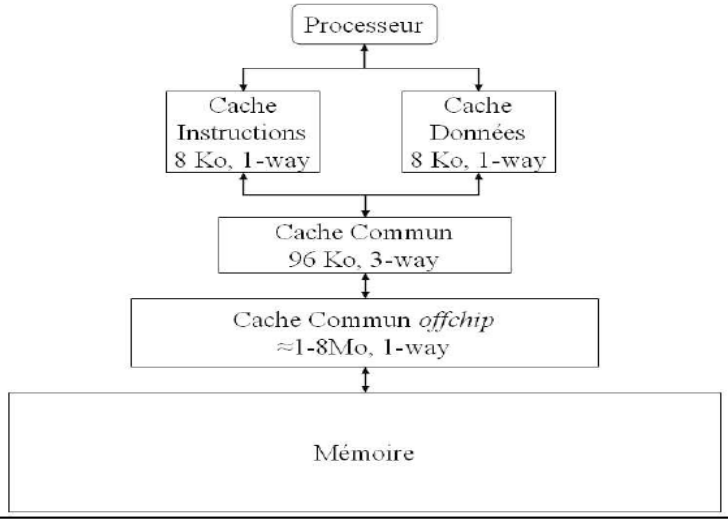
TLB. Le processeur utilisant des adresses virtuelles, et les données étant placées à des adresses physiques en mémoire principale, le processeur accède en général au cache à l'aide d'adresses virtuelles, tandis que les données peuvent être rechargées dans le cache à l'aide d'adresses physiques, i.e., les étiquettes correspondent à des adresses physiques.

Une façon astucieuse de procéder, utilisée dans plusieurs caches, est de dimensionner les bancs du cache de façon à ce qu'ils soient inférieurs à la taille d'une page. Les bits utilisés pour chercher le numéro d'octets dans la ligne et le numéro de ligne dans le cache sont alors les mêmes pour l'adresse physique et pour l'adresse virtuelle, ils correspondent aux bits utilisés pour chercher le numéro d'octets dans la page. Le contrôleur du cache peut alors commencer l'accès au cache et récupérer les étiquettes tandis que le processeur obtient une traduction en adresse physique de son étiquette virtuelle. Pour accélérer cette traduction et éviter d'aller la chercher en mémoire pour chaque instruction load/store, la plupart des processeurs intègrent un TLB (*Translation Lookaside Buffer*) qui n'est rien d'autre qu'un cache de traductions d'adresses ; il contient donc les adresses virtuelles et physiques des pages les plus récemment et fréquemment utilisées.



Hiérarchie de caches. La différence de performance entre processeur et mémoire continuant à s'accroître, la plupart des processeurs récents intègrent non pas un cache, mais une hiérarchie de caches dans la taille augmente et la vitesse décroît au fur et à mesure que l'on se rapproche de la mémoire principale.

Exemple. Hiérarchie de caches dans le processeur Compaq Alpha 21164.

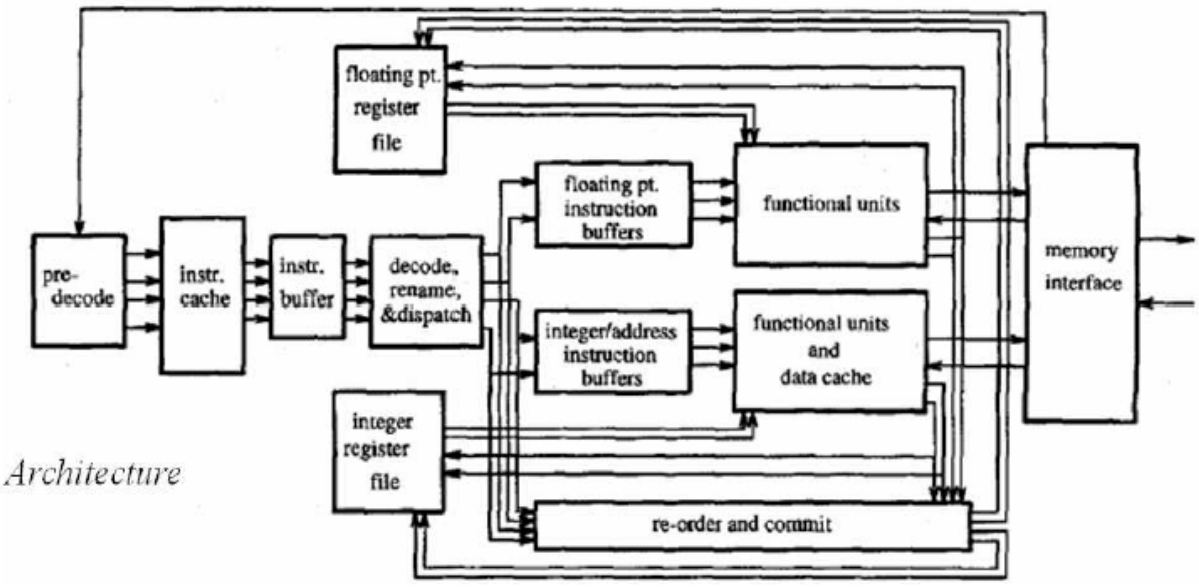


4. Processeurs superscalaires

Une façon majeure de gagner en performance, au-delà de l'exécution pipelinée des instructions, est d'exécuter en parallèle plusieurs instructions. La plupart des processeurs haute-performance ont aujourd'hui cette capacité, on parle de processeurs **superscalaires**. Pour réaliser un processeur superscalaire, il faut:

- Assurer un flux d'instructions suffisant.
- Déterminer quelles instructions peuvent s'exécuter en parallèle.
- Passer les données entre les instructions (le résultat d'une instruction *i* est l'opérande d'une instruction *j*).
- Disposer de plusieurs unités de calcul en parallèle.

La plupart des processeurs superscalaires ont une structure similaire à celle indiquée dans la figure ci-dessous :



Parallélisme entre instructions. Bien que notre façon de raisonner soit essentiellement séquentielle, une fois le programme traduit en assembleur, on constate qu'il y a un degré important de parallélisme entre instructions (*ILP : Instruction-Level Parallelism*).

Exemple de parallélisme entre instructions.

```
for (i=0; i<last; i++) {
  if (a[i] > a[i+1]) {
    temp = a[i];
    a[i] = a[i+1];
    a[i+1] = temp;
    change++;
  }
}
```

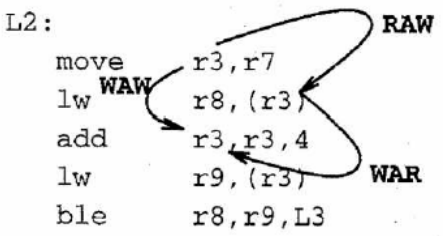
```
L2:
    move    r3,r7       #r3->a[i]
    lw      r8,(r3)     #load a[i]
    add     r3,r3,4     #r3->a[i+1]
    lw      r9,(r3)     #load a[i+1]
    ble     r8,r9,L3    #branch a[i]>a[i+1]

    move    r3,r7       #r3->a[i]
    sw      r9,(r3)     #store a[i]
    add     r3,r3,4     #r3->a[i+1]
    sw      r8,(r3)     #store a[i+1]
    add     r5,r5,1     #change++
L3:
    add     r6,r6,1     #i++
    add     r7,r7,4     #r4->a[i]
    blt     r6,r4,L2    #branch i<last
```

Chargement des instructions. Une des principales difficultés des processeurs superscalaires actuels est de charger des instructions à un rythme suffisant pour alimenter le pipeline. Le principal écueil est la présence de nombreux branchements. Pour limiter leur impact sur le flux des instructions, on utilise plusieurs techniques : avant tout, les techniques de prédiction de conditions et d'adresses mentionnées ci-dessus, mais aussi le chargement anticipé des instructions depuis le cache des instructions dans un tampon de préchargement, et également l'insertion de bits de prédécodage dans le cache d'instructions afin de déterminer très rapidement la nature de l'instruction (branchement conditionnel, branchement inconditionnel, instruction n'influant pas sur le flot de contrôle).

Décodage des instructions. Une fois les instructions chargées, la première étape consiste à déterminer les dépendances entre les instructions, plus exactement entre les registres lus et écrits par les instructions. Cette analyse permettra de déterminer quelles instructions peuvent s'exécuter en parallèle. Elle permet également d'éliminer les fausses dépendances entre les instructions liées à la réutilisation des registres ; cette dernière action s'accompagne du renommage des registres.

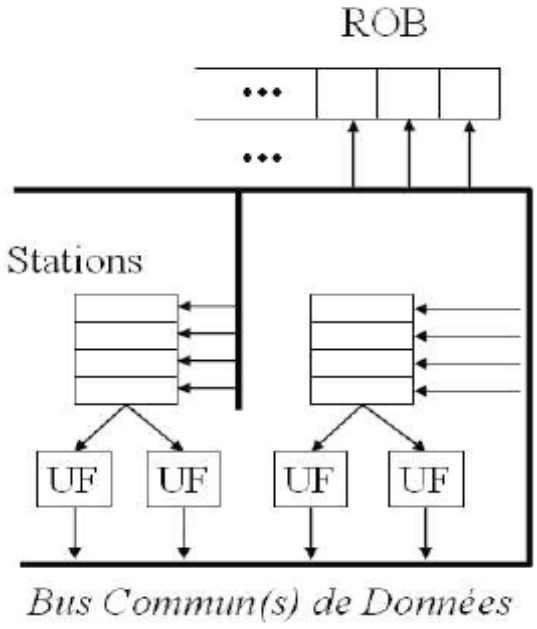
Exemple. Les instructions move r3 *et* lw r8 *ci-dessous ne peuvent s'exécuter en parallèle en raison d'une dépendance sur le registre* r3. *Par ailleurs, le fait que l'instruction* move r3 *utilise le même registre de destination que l'instruction* add r3 *va ralentir la fin de l'exécution de cette seconde instruction en raison de la réutilisation du registre* r3.



Renommage des registres et tampon de réordonnancement. Le nombre de registres que peuvent manipuler les instructions assembleur est limité par le jeu d'instructions. Or la taille des processeurs permet aujourd'hui de disposer de beaucoup plus de registres. Comme ces registres additionnels ne peuvent pas être exhibés à l'utilisateur par le biais du jeu d'instructions, les architectures de processeurs superscalaires effectuent un renommage à la volée des registres. On parle alors de registres logiques, ceux du jeu d'instructions et du programme, et de registres physiques, ceux contenus dans le processeur.

En pratique, on ne dispose pas toujours de registres physiques supplémentaires mais d'un **tampon de réordonnancement** (*ROB : ReOrder Buffer*, le *ROB* est une FIFO) qui permet à la fois de mémoriser l'ordre des instructions dans le programme (on verra plus tard pourquoi cette fonctionnalité est nécessaire) et de fournir l'équivalent de registres physiques additionnels. Outre le ROB, le processeur contient également un banc de registres correspondant aux registres logiques. A chaque instruction chargée est associée une entrée dans le ROB. Le registre logique de destination de l'instruction est remplacé par le numéro d'une entrée du ROB: on renomme le registre. En outre, la valeur produite par l'instruction est également stockée dans le ROB, et une fois qu'une instruction a terminé son exécution et sort du ROB, sa valeur est écrite dans son registre logique de destination, dans le banc de registres. Une table indique où se trouve la valeur courante d'un registre logique (dans le banc de registres ou dans le ROB), et lors du chargement d'une instruction, ses opérandes sont donc soit dans le ROB, soit dans le banc de registres.

Exécution. Une fois ces étapes préliminaires effectuées, on envoie l'instruction pour exécution. Si l'un de ses opérandes n'est pas encore disponible, ou si aucune unité fonctionnelle n'est libre, on envoie l'instruction dans une des **stations de réservation** associée à l'unité fonctionnelle. Ces stations de réservation forment une file d'attente devant l'unité fonctionnelle et espionnent en permanence les bus de sortie des unités fonctionnelles ; dès qu'une station de réservation repère que l'instruction dont elle attend le résultat vient de terminer son exécution, elle se saisit du résultat, et si l'unité fonctionnelle est libre, l'instruction en attente commence son exécution. Implicitement, les instructions vont s'exécuter dans l'ordre de disponibilité de leurs opérandes : à l'intérieur du processeur, il s'agit donc d'un modèle d'exécution de type *Dataflow* et non de type *Von Neumann*.



Fin de l'exécution. Après exécution de l'instruction, le résultat est donc propagé aux stations de réservation par le biais de bus communs de données et est également stocké dans le ROB. Le ROB étant une FIFO, les instructions en sortent dans l'ordre du programme ; vu de l'extérieur, un processeur superscalaire se comporte donc bien comme un processeur de type *Von Neumann*. De même qu'il est possible de charger plusieurs instructions à la fois, en un seul cycle, il est possible de sortir plusieurs instructions à la fois du ROB. Lorsqu'une instruction sort du ROB, son résultat est écrit dans le banc de registres.

Une des principales raisons pour lesquelles on force les instructions à terminer dans l'ordre est la nécessité d'implémenter des exceptions précises : lorsque l'on traite une exception, en fin de pipeline, on doit s'assurer que toutes les instructions précédentes ont bien terminé de s'exécuter. Sachant qu'une instruction *store* n'est effectivement envoyée à la mémoire que lorsqu'elle sort du ROB, on constate donc que l'état du processeur (registres et mémoires) est bien modifié dans l'ordre spécifié par les instructions du programme assembleur.

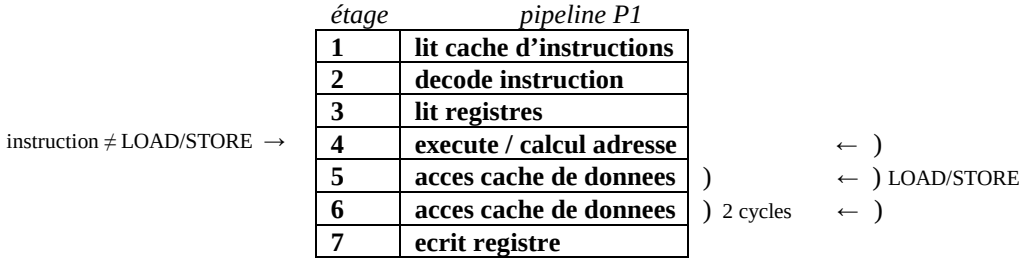
5. Conclusion

La complexité des processeurs haute-performance est telle, aujourd'hui, qu'il est difficile d'écrire un programme qui en exploite réellement la performance maximale. Il existe une différence très importante, et surtout croissante, entre la performance maximale d'un processeur et sa performance soutenue. En outre, il est de plus en plus difficile de dimensionner un processeur (d'en accroître la capacité et la performance) au fur et à mesure de l'évolution de la technologie. Il est donc possible que, dans les années à venir, on assiste à une modification relativement profonde du paradigme d'exécution des processeurs haute-performance.

TD 4. Les Processeurs actuels. Pipelining

1. Pipelining

On considère le pipeline d'instructions (P1) représenté ci-dessous :



Le débit théorique maximum de ce pipeline est de 1 instruction/cycle.
(1 cycle représente le temps nécessaire pour exécuter 1 étage du pipeline).

On suppose que toutes les instructions, sauf les LOAD/STORE, sont exécutées à l'étage 4. Pour les instructions LOAD/STORE, le calcul d'adresse du/des opérandes du LOAD/STORE se fait à l'étage 4 et l'accès au cache de données est pipeliné sur 2 cycles (étages 5 et 6). Pour les instructions qui ne sont pas des LOAD/STORE, les étages 5 et 6 se comportent comme des étages vides.

On suppose qu'il y a un mécanisme de *bypass* permettant de transmettre le résultat d'une instruction aux instructions suivantes sans attendre l'écriture registre. Si un opérande source d'une instruction n'est pas disponible au moment où celle-ci va rentrer dans l'étage 4, les étages 1 à 3 sont bloqués jusqu'à ce que l'opérande indisponible soit accessible via le mécanisme de *bypass*, ce qui conduit à l'insertion de *bulles* dans le pipeline.

On supposera un prédicteur de branchement parfait. On supposera également que toutes les lectures d'instructions font des *hits* dans le cache d'instructions et que tous les LOAD/STORE font des *hits* dans le cache de données (cela signifie qu'il n'y a que des *cache-hits* et pas de *cache-miss*, c'est-à-dire que les accès cache sont toujours tels que le cache respectivement d'instructions / données contient les cibles, instructions / données).

1.

Lorsqu'un LOAD est immédiatement suivi d'une instruction utilisant le résultat du LOAD, combien de bulles sont insérées dans le pipeline ?

2.

On considère le programme ci-dessous écrit dans un assembleur donné (*asm*), qui calcule la somme des N éléments d'un tableau. Le registre R4 est initialisé avec N, le registre R1 est initialisé avec l'adresse A du 1er élément du tableau et le registre R3 contenant en fin d'exécution le résultat de la somme est initialisé à 0. Les éléments du tableau sont stockés sur 4 octets.

```
1:      boucle:  R2 = LOAD R1+0        // lit element du tableau
2:               R3 = R3 ADD R2        // ajoute a la somme
3:               R1 = R1 ADD 4         // R1 = R1 + 4
4:               R4 = R4 SUB 1         // R4 = R4 - 1
5:               BNZ R4, boucle        // BNZ (Branch Not Zero) : boucle si R4 ≠ 0
```

En supposant N grand, quel est le débit d'exécution de cette boucle en instructions par cycle ?

3.

Changez l'ordre des instructions dans la boucle afin d'obtenir un débit de 1 instruction/cycle.

4.

On considère le programme ci-dessous qui calcule la somme des éléments du tableau mais en parcourant le tableau dans l'autre sens. R4 est initialisé à 4N, R1 est initialisé à A−4N et R3 toujours initialisé à 0.

```
1:      boucle:  R2 = LOAD R1+R4        // lit element du tableau
2:               R4 = R4 SUB 4          // R4 = R4 - 4
3:               R3 = R3 ADD R2         // ajoute a la somme
4:               BNZ R4, boucle         // BNZ (Branch Not Zero) : boucle si R4 ≠ 0
```

a) Toujours en supposant N grand, quel est le débit d'exécution du programme en instructions par cycle ?

b) Ce programme est-il plus performant que le programme de la question **2** ? Est-il plus performant que le programme modifié à la question **3** ?

5.

On modifie le pipeline d'instructions comme représenté ci-dessous (P2), c'est-à-dire en déplaçant l'étage d'exécution de l'étage 4 à l'étage 6. Le calcul d'adresse continue à se faire à l'étage 4.

étage	*pipeline P2*
1	**lit cache d'instructions**
2	**decode instruction**
3	**lit registres**
4	**calcul adresse**
5	**Acces cache de donnees**
6	**execute / acces cache de donnees**
7	**ecrit registre**

La performance du programme de la question **4** sur le nouveau pipeline (P2) est-elle supérieure ou inférieure à celle sur l'ancien pipeline (P1) ?

TP 4. Les Processeurs actuels. Pipelining

1. Pipelining (1)

On considère le pipeline ci-dessous dédié aux instructions en virgule flottante (pour données FP Floating Point) :

étage	*pipeline P*
1	**lit cache d'instructions**
2	**decode instruction**
3	**lit registres FP**
4	**execute FP / acces cache de donnees**
5	**execute FP / acces cache de donnees**
6	**execute FP / acces cache de donnees**
7	**ecrit registre FP**

Le débit théorique maximum de ce pipeline est de 1 instruction/cycle.

On suppose un prédicteur de branchements parfait et les instructions déjà dans les caches.

Les instructions en virgule flottante (FP) sont pipelinées sur 3 cycles.

Une instruction FP dépendant de l'instruction immédiatement précédente doit attendre 2 cycles avant de rentrer dans le 1er étage d'exécution $\rightarrow$ 2 bulles sont insérées si une instruction utilise en opérande le résultat de l'instruction immédiatement avant.

On veut calculer l'expression : $ay^3 + by^2 + cy + d$. On propose 2 méthodes pour implémenter ce calcul :

- méthode 1 (de Horner) : $d + y(c + y(b + ya))$ opérations : 3 × ; 3 +

- méthode 2 : $(d + cy) + ((y \times y) \times (ay + b))$ opérations : 4 × ; 3 +

On remarquera que la méthode 2 demande 1 multiplication de plus que la méthode 1 (de Horner).

On supposera que y,a,b,c,d se trouvent déjà dans les registres flottants respectifs F0, F1, F2, F3, F4 et que le autres registres sont libres. On veut obtenir le résultat dans le registre F5.

a) Ecrire le pseudo-code assembleur pour les 2 méthodes.

b) Quelle méthode est la plus performante sur le pipeline considéré ?

2. Pipelining (2)

1. Pourquoi le pipelining améliore t-il la performance ?

2. Quelles sont les limites de l'amélioration de performance apportée par le pipelining ?

3. Calcul du temps d'exécution d'une séquence d'instructions :

Exemple : **Rappel :** latence = durée d'exécution, temps de traitement, d'une instruction

Supposons qu'un processeur non pipeliné possedant un temps de cycle (temps de traitement d'1 instruction) de 25 ns soit divisé en 5 étages de pipeline de latences respectives de 5, 7, 3, 6 et 4 ns. La latence de latch du pipeline (registre de sortie de chaque étage du pipeline) est de 1 ns. Le pipeline est supposé sans délais de branchement (prédiction de branchement parfaite, sans aléas).

3.1. Quelle est le temps de cycle (temps de traitement d'1 étage du pipeline) du processeur pipeliné ?

3.2. Quelle est la latence totale du pipeline (temps de traitement de tous les étages du pipeline) ?