III Opérateurs élémentaires

Une fonction numérique complexe est construite de façon hiérarchique, comme un assemblage de « boites noires », fonctions moins complexes, définies par leurs entrées, leurs sorties et les relations entre les premières et les secondes. Tout en bas de cette hiérarchie on trouve des opérateurs élémentaires, les briques ultimes au delà desquelles intervient l'électronicien qui les réalise avec des transistors ; mais au delà de cette frontière le monde du numérique s'arrête. Nous considérerons donc que les briques élémentaires de notre construction sont ces opérateurs élémentaires, et avant d'explorer cette démarche descendante qui va du général au particulier, du complexe au simple, nous tenterons de bien comprendre le fonctionnement de ces opérateurs élémentaires.

Nous utiliserons, entre autres, un langage de haut niveau, VHDL[1], pour décrire le fonctionnement de ces opérateurs élémentaires. Il est bien évident que VHDL connaît ces opérateurs comme primitives internes, et qu'il y a donc là une redondance certaine. Mais cela nous familiarisera avec ce langage qui est en passe de devenir un standard de description des systèmes numériques, et même, à terme, des systèmes analogiques.

Sauf précision contraire, nous adopterons dans la suite une convention logique[2] positive, qui associe le 0 binaire à la valeur logique FAUX et le 1 binaire à la valeur logique VRAI.

III.1. Combinatoire et séquentiel

Certains de ces opérateurs élémentaires sont la matérialisation, sous forme de circuits, ou de parties de circuits, des opérateurs bien connus de l'algèbre de BOOLE. D'autres, et notamment (mais pas uniquement) ceux que l'on appelle *opérateurs séquentiels*, sont une spécialité de l'électronique numérique, leur description n'apparaît dans aucun traité relatif à la dite algèbre.

Si l'on cherche à classer les opérateurs par familles, et c'est en classant les choses que l'on se donne les moyens d'en comprendre éventuellement le fonctionnement, le premier critère de classement concerne la façon dont évolue, au

[1]VHDL est un acronyme de Very high speed integrated circuits Hardware description language.
[2]Voir chapitre I.

cours du temps, l'état d'un opérateur, donc ses sorties[3]. On dira qu'un opérateur est *combinatoire* si les valeurs de ses sorties sont déterminées de façon univoque par les valeurs des entrées au même instant (à un temps de propagation près, bien sûr). Un opérateur est *séquentiel* si son état à un instant donné dépend des entrées, évidemment, mais aussi de ses *états passés*, du chemin qu'il a parcouru pour en arriver là, bref, un opérateur séquentiel est doué de *mémoire*.

Prenons un exemple : une serrure mécanique à combinaisons et une serrure hypothétique à digicode.

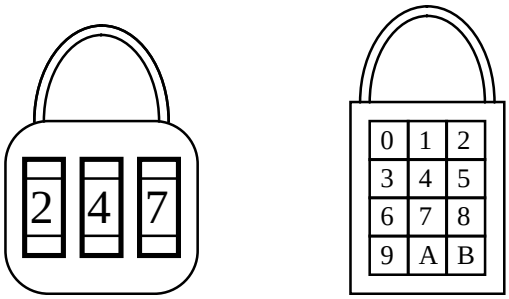


Figure III-1

L'ouverture de la première sera provoquée par une combinaison prédéfinie des trois variables d'entrée, l'ouverture de la seconde sera obtenue par la frappe d'une *séquence* prédéfinie de valeurs sur le clavier. Par exemple, si pour les deux serrures le code d'accès est 247, la première serrure s'ouvrira dès que ses variables d'entrée (les molettes) afficheront ce code, la deuxième exigera que les trois chiffres de la clé soient tapés au clavier *dans le bon ordre*. Il est clair que la deuxième serrure doit « se souvenir » du 2 quand on tape le 4, du 2 et du 4 quand on tape le 7 ; elle doit donc posséder une mémoire interne. Cette mémoire interne peut être une simple mémorisation des chiffres précédemment tapés, c'est conceptuellement la solution la plus évidente, même si ce n'est pas la plus simple à réaliser. Une autre solution consiste à doter la serrure d'une variable interne qui mémorise l'état d'avancement de la séquence, sans mémoriser les nombres eux-mêmes. On pourra alors décrire le fonctionnement du système par une sorte de diagramme, figure III-2, dans lequel chaque case représente l'état interne, auquel sont éventuellement attachées des actions (sorties), et les flèches les transitions d'un état à l'autre. A côté de chaque transition figure la condition[4] sur l'entrée qui provoque le franchissement de cette transition.

[3]Etat et sorties d'un système sont deux concepts différents, mais un système dont l'état n'est pas visible de l'extérieur n'a guère d'intérêt, l'évolution de l'état interne d'un objet a donc à priori une manifestation externe visible en sortie, même si cette manifestation n'est pas immédiate.

[4]Le diagramme présenté figure III-2 comporte en fait des erreurs par omission, l'utilisateur astucieux peut obtenir l'ouverture de la serrure, même s'il ne connaît pas le code d'accès. Question : comment ?

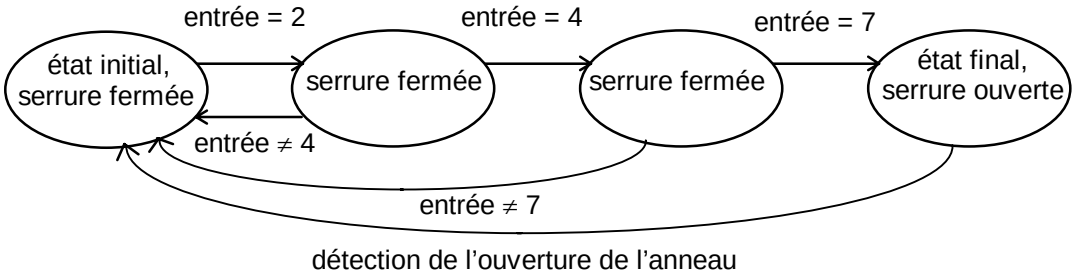


Figure III-2

La plupart des systèmes réels sont séquentiels, et beaucoup peuvent être analysés simplement par l'introduction d'une ou de plusieurs variables d'état.

L'importance des systèmes séquentiels a amené les concepteurs de circuits à imaginer des opérateurs élémentaires spécifiques, qui facilitent la tâche du concepteur. Pour illustrer le genre de questions auxquelles est amené le concepteur à répondre, reprenons l'exemple de la serrure :

- Que doit-on faire si plusieurs entrées changent simultanément ?
- Comment faire coopérer plusieurs sous-systèmes de façon prévisible ?

L'idée est progressivement venue que le travail de conception était grandement simplifié si les transitions ne pouvaient avoir lieu qu'à des temps connus, indépendamment des instants de changements des entrées, instants que, bien évidemment, le concepteur ne peut pas connaître. On rajoute alors un signal interne spécial, *l'horloge*, qui fixe la cadence de l'évolution du système. Les systèmes qui évoluent sous le contrôle d'une horloge sont appelés systèmes séquentiels *synchrones*. Les circuits synchrones ont un fonctionnement qui ne peut pas être entièrement compris dans le cadre de la logique combinatoire, ils font appel à des opérateurs élémentaires, les *bascules*, qui sont des briques de base à part entière de la logique, au même titre qu'un opérateur « ET ».

III.2. Opérateurs combinatoires

Pour chaque opérateur, nous indiquerons le ou les symboles couramment rencontrés, puis nous en donnerons une description sous forme de *table de vérité*, d'expression algébrique, de *diagramme de Venn* (héritage de la théorie élémentaire des ensembles) et sous forme d'algorithme dans le langage VHDL.

III.2.1 Des opérateurs génériques : NON, ET, OU

Les opérateurs NON (NOT), ET (AND) et OU (OR) jouent un rôle privilégié dans la mesure où ils sont « génériques », c'est à dire que toute fonction combinatoire peut être exprimée à l'aide de ces opérateurs élémentaires.

Les symboles

Bien que les symboles « rectangulaires » soient normalisés, la majorité des notices emploient les symboles curvilignes traditionnels (figure III-3). L'assemblage de symboles élémentaires dans un « schéma » porte le nom de *logigramme*, un logigramme est presque le schéma de câblage dans lequel on aurait oublié les masses et les alimentations des circuits.

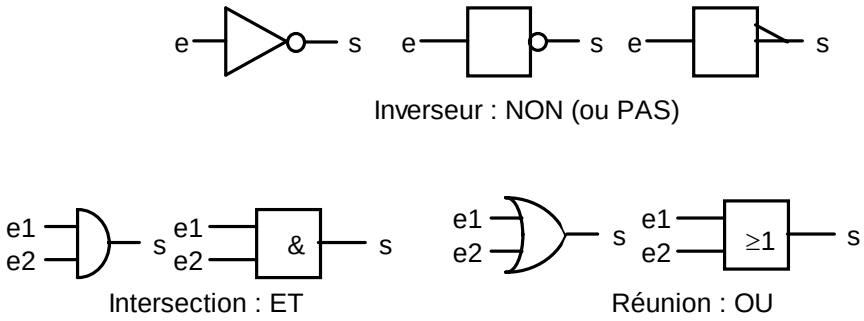


Figure III-3

Les tables de vérité

Toute fonction combinatoire peut, en dernier ressort, être décrite par une table qui énumère les valeurs prises par la (ou les) sortie(s) en fonction des valeurs des variables d'entrée. Cette méthode apparemment simple a le défaut de devenir extrêmement lourde quand le nombre de variables mises en jeu augmente. Il faut cependant y recourir quand les méthodes plus abstraites ne permettent pas de répondre à une interrogation.

Les tables peuvent être présentées sous forme linéaire ou, ce qui est souvent plus compact, sous forme de tableaux (figure III-4) :

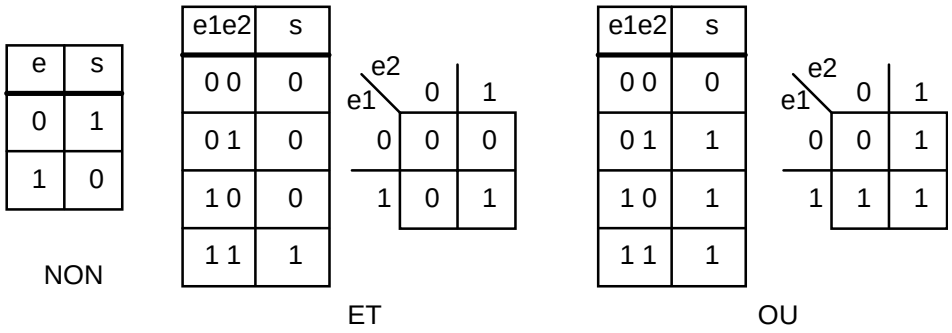


Figure III-4

Notations algébriques

Opérateur NON : $s = \bar{e}$, noté parfois, par commodité d'écriture s = /e.

Opérateur ET : $s = e1 \wedge e2$, ou $s = e1 \,\&\, e2$, ou encore $s = e1 * e2$.

Opérateur OU : $s = e1 \vee e2$, ou $s = e1 \mid e2$, ou encore $s = e1 + e2$.

Les deux notations $*$ et + pour les opérateurs ET et OU sont évidemment particulièrement ambiguës, donc inutilisables, quand on mélange dans une même description des expressions arithmétiques et des expressions logiques. Quand il n'y a pas risque de confusion, ce sont pourtant les notations les plus fréquentes.

Lors de l'écriture d'une expression qui fait intervenir les différents types d'opérateurs, les règles de priorité généralement adoptées vont, de la priorité la plus grande à la plus faible, de l'inversion (NON) au OU :

a + b * /c doit être compris a + (b * (/c))

Par contre il convient d'être extrêmement prudent quand apparaît un mélange d'opérations arithmétiques et logiques (mélange autorisé dans certains langages), la prudence élémentaire dicte, dans un tel cas, de parenthéser les expressions.

Diagrammes de Venn

Ces diagrammes illustrent bien les propriétés des expressions simples, ils soulignent l'identité de propriétés de l'algèbre de Boole et de l'algèbre des parties d'un ensemble, munie des opérations complémentation (NON), intersection (ET) et réunion (OU) (figure III-5).

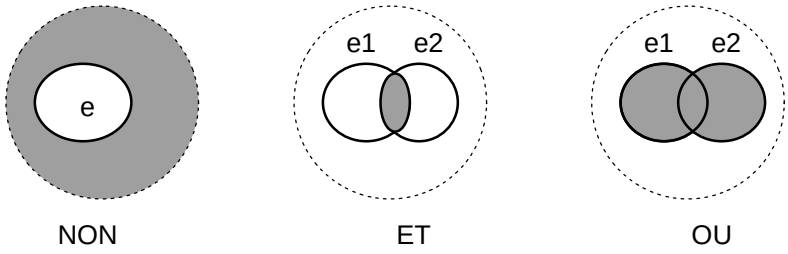


Figure III-5

Description en VHDL

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 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 pleonasme OF inverseur IS
BEGIN
s <= NOT e;
END pleonasme;
```

de même :

```
-- operateur ET
ENTITY et IS
PORT ( e1, e2 : IN  BIT ;
            s : OUT BIT );
END et;
ARCHITECTURE pleonasme OF et IS
BEGIN
s <= e1 AND e2;
END pleonasme;
```

ou encore :

```
-- operateur OU
ENTITY ou IS
PORT ( e1, e2 : IN  BIT ;
            s : OUT BIT );
END ou;
ARCHITECTURE pleonasme OF ou IS
BEGIN
s <= e1 OR e2;
END pleonasme;
```

On notera la structure générale d'un programme et le symbole « d'affectation » particulier aux objets de nature signal (s, e, e1, e2). La déclaration ENTITY correspond au prototype d'une fonction en 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. Les mots clés du langage ont été mis en majuscule, 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 logique OF inverseur IS
BEGIN
s <= '1' WHEN (e = '0') ELSE  '0';
END logique;
```

de même :

```
-- operateur ET
ENTITY et IS
PORT ( e1, e2 : IN  BIT ;
            s : OUT BIT );
END et;
ARCHITECTURE logique OF et IS
BEGIN
s <= '0' WHEN (e1 = '0' OR e2 = '0') ELSE '1';
END logique;
```

ou encore :

```
-- operateur OU
ENTITY ou IS
PORT ( e1, e2 : IN  BIT ;
            s : OUT BIT );
END ou;
ARCHITECTURE logique OF ou IS
BEGIN
s <= ‘0’ WHEN (e1 = ‘0’ AND e2 = ‘0’) ELSE ‘1’;
END logique;
```

Des exemples de modèles comportementaux

Terminons cette première découverte de VHDL par deux descriptions purement comportementales des opérateurs ET et OU :

```
ENTITY et IS -- operateur ET
PORT ( e1, e2 : IN  BIT ;
            s : OUT BIT );
END et;
ARCHITECTURE abstrait OF et IS
BEGIN
PROCESS ( e1,e2 )
BEGIN
      IF (e1 = ‘0’ OR e2 = ‘0’) THEN
            s <= ‘0’;
      ELSE
            s <= ‘1’;
      END IF;
END PROCESS;
END abstrait;
```

ou encore :

```
-- operateur OU
ENTITY ou IS
PORT ( e : IN  BIT_VECTOR(0 TO 1) ; -- ATTENTION!!!
            s : OUT BIT );
END ou;
ARCHITECTURE abstrait OF ou IS
BEGIN
PROCESS ( e )
BEGIN
      CASE e IS
            WHEN "00" =>
                  s <= '0';
            WHEN OTHERS =>
                  s <= '1';
      END CASE;
END PROCESS;
```

```
END abstrait;
```

III.2.2 Un peu d'algèbre

Nous rappelons rapidement ici quelques propriétés élémentaires des opérateurs fondamentaux de la logique combinatoire. Le lecteur désireux de parfaire sa culture sur ce sujet pourra consulter un ouvrage de mathématiques, au chapitre qui traite de l'algèbre de Boole ou de l'algèbre des parties d'un ensemble[5]. Parmi ces propriétés, les plus importantes, et de loin, dans les applications, sont les lois de De Morgan : ces deux lois permettent de passer d'une convention logique à une autre, sans calcul, ou presque.

Les démonstrations concernant l'algèbre de Boole peuvent toujours se faire, en dernier recours, par un examen des tables de vérité. Cette méthode, un peu lourde, doit être envisagée si des méthodes plus astucieuses ne sont pas trouvées ; en tout état de cause, il n'est pas pensable de rester dans le doute en ce qui concerne un résultat de logique combinatoire. L'intuition permet de gagner du temps dans l'obtention d'un résultat, son absence ne justifie pas le doute.

Propriétés des opérateurs ET et OU

Associativité, commutativité

Associativité :

$$a * (b * c) = (a * b) * c \text{ , de même : } a + (b + c) = (a + b) + c .$$

Commutativité :

$$a * b = b * a \text{ , et : } a + b = b + a .$$

Un opérateur, agissant sur deux opérandes, qui est associatif et commutatif peut être généralisé à un nombre quelconque d'opérandes, sans qu'il soit nécessaire de parenthéser les expressions, par exemple :

$a + b + c + d + e$ est défini de façon univoque quel que soit l'ordre dans lequel on effectue les « calculs ».

Pratiquement cela signifie qu'il est possible de concevoir des opérateurs ET et OU à nombre arbitraire d'entrées (figure III-6) :

[5]Par exemple : J.C. BELLOC et P. SCHILLER : *Mathématiques pour l'électronique*, Masson, 1994.

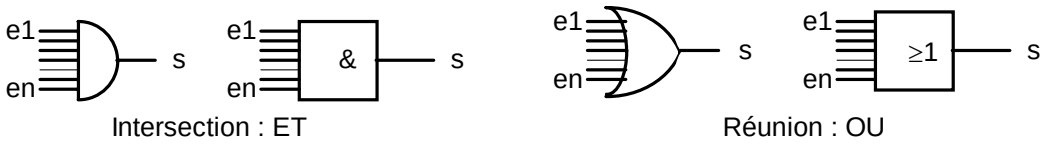


Figure III-6

On notera que la sortie d'un ET vaut 0 si l'une au moins des entrées est à 0, et que, réciproquement, la sortie d'un OU vaut 1 si l'une au moins des entrées est à 1.

Double distributivité

La ressemblance entre les propriétés des opérateurs arithmétiques, appliqués à des chiffres binaires, et celles des opérateurs logiques est grande. Une différence notable concerne la distributivité, les opérateurs ET et OU sont mutuellement distributifs l'un par rapport à l'autre :

$$a * (b + c) = (a * b) + (a * c) \ ,$$

ce qui n'étonne personne (a(b + c) = ab+ac).

$$a + (b * c) = (a + b) * (a + c) \ ,$$

qui n'a pas d'équivalent arithmétique ($a + bc \neq (a + b)\,(a + c)$).

La distributivité du OU par rapport au ET, illustrée par la deuxième des relations ci-dessus, est d'autant plus troublante que l'on adopte généralement la même convention de priorité entre les opérateurs ET et OU qu'entre leurs « analogues » arithmétiques : les opérateurs multiplicatifs sont plus prioritaires que les opérateurs additifs. Cela permet d'éviter certaines parenthèses dans l'écriture des relations un peu complexes, mais rompt l'aspect symétrique des propriétés de la réunion et de l'intersection logiques.

Pot pourri

Sans commentaire :
$a * a = a$, $a * 1 = a$ (élément neutre), $a * 0 = 0$,
$a + a = a$, $a + 1 = 1$, $a + 0 = a$ (élément neutre),
$\overline{\overline{a}} = a$, $a + \overline{a} = 1$, $a * \overline{a} = 0$, $a + (\overline{a} * b) = a + b$, $a + a * b = a$

Les lois de DE MORGAN

Les deux lois de De Morgan permettent le passage d'une fonction logique à son complément, elles sont utilisées systématiquement par les logiciels d'aide à la

synthèse de circuits logiques, pour déterminer la convention logique qui conduit à l'équation la plus simple qui rende compte d'un problème donné. Les voici :

$$\overline{(A + B)} = \overline{A} * \overline{B}$$

et :

$$\overline{(A * B)} = \overline{A} + \overline{B}$$

Bien évidemment, dans les expressions précédentes, A et B peuvent être elles-même des expressions. Sous ces formules apparemment simples se cachent parfois des calculs importants.

III.2.3 Non-ET, Non-OU

Les opérateurs NON-ET (*NAND*), et NON-OU (*NOR*), jouent un rôle particulier : ils contiennent chacun, éventuellement via les lois de De Morgan, les trois opérateurs génériques de la logique combinatoire ET, OU et NON.

- Le premier, l'opérateur NAND, est générateur de la technologie TTL (74xx00).
- Le second, l'opérateur NOR, est générateur de la technologie ECL.

Définitions et symboles

Non Et

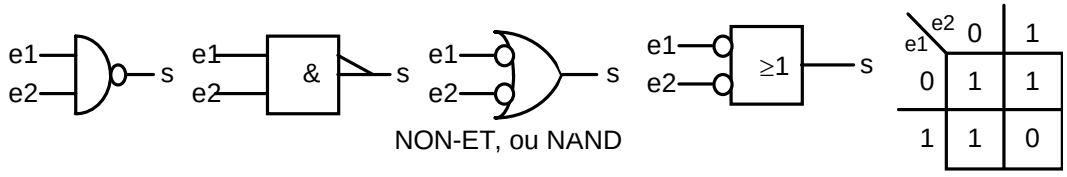


Figure III-7

Les équations de l'opérateur NAND sont, en appliquant les lois de De Morgan :

$$s = \overline{e1 * e2} = \overline{e1} + \overline{e2}$$

Non Ou

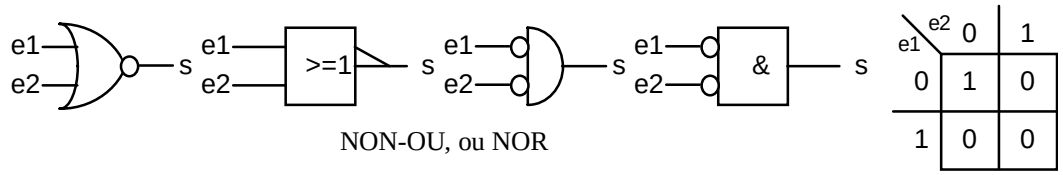


Figure III-8

Les équations de l'opérateur NOR sont, en appliquant les lois de De Morgan :

$$s = \overline{e1 + e2} = \overline{e1} * \overline{e2}$$

Les opérateurs NAND et NOR ne *sont pas* associatifs, ils ne sont donc pas généralisables, sans précaution, à un nombre quelconque d'entrées. Par contre on peut définir un opérateur qui est le complément du ET (respectivement du OU) à plusieurs entrées comme un NON-ET (respectivement NON-OU) généralisé :

$s = \overline{e1 * \ldots * en}$ (ou $s = \overline{e1 + \ldots + en}$ respectivement).

Une illustration des lois de De Morgan

A titre d'illustration des lois de De Morgan, et pour préciser ce que l'on entend par un opérateur générique, montrons qu'une expression quelconque peut être construite en n'utilisant que des opérateurs de type NAND :

$$a + b * (c + d * e) = \overline{\overline{a + b * (c + d * e)}} = \overline{\overline{a} * \overline{b * (c + d * e)}}$$
$$= \overline{\overline{a} * \overline{b * (c + d * e)}} = \overline{\overline{a} * \overline{b * \overline{\overline{c} * \overline{d * e}}}}$$

La dernière expression ne fait appel qu'à des opérateurs de type NAND, et à des inverseurs qui sont des NAND à une seule entrée.

III.2.4 Le « ou exclusif », ou somme modulo 2

Opérateur aux multiples applications, le OU EXCLUSIF (*XOR*) est sans doute l'opérateur à deux opérandes le plus riche et le moins trivial. Il trouve ses applications dans les fonctions :

- arithmétiques, additionneurs, comparateurs et compteurs ;
- de contrôle et de correction d'erreurs ;
- où l'on souhaite pouvoir programmer la convention logique ;

– de cryptage de l'information.

Après avoir donné la définition de cet opérateur, nous donnerons quelques exemples de ces applications.

Définition algébrique et symboles

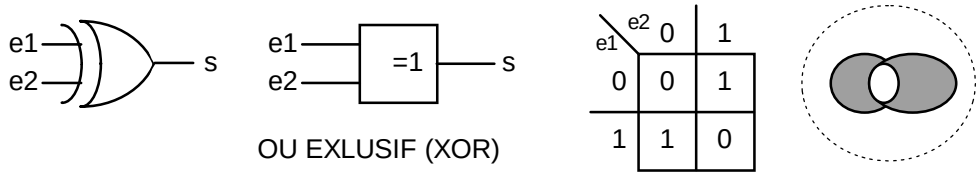


Figure III-9

On peut remarquer que cet opérateur prend la valeur 1 quand ses deux opérandes sont *différents*.

La définition algébrique du OU EXCLUSIF, à l'aide des opérateurs ET (∗) et OU (+), est :

$$s = e1 \oplus e2 = \overline{e1} * e2 + e1 * \overline{e2} = (e1 + e2) * (\overline{e1} + \overline{e2})$$

D'autres symboles sont parfois rencontrés pour désigner le OU EXCLUSIF :
:+:, ^ ou, plus rarement, ↑.

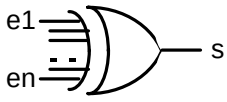
ATTENTION ! Comme on peut facilement le vérifier sur la table de vérité de la figure III-9, le OU EXCLUSIF est l'opérateur d'addition élémentaire de deux chiffres en base deux. Il est donc possible de noter cet opérateur « + », tout simplement, quand il n'y a pas de risque de confusion avec le ou inclusif. Ce type de confusion ne se pose pas dans des langages de haut niveau comme VHDL où le + est le symbole de l'addition, qui porte sur des nombres, les opérateurs logiques, dont les opérandes sont de type BIT ou BOOLEAN, sont représentés par leurs noms AND, OR et XOR.

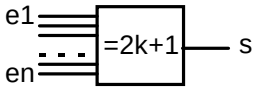
Propriétés élémentaires et applications

Algèbre

L'opérateur ou exclusif possède les propriétés de l'addition : il est associatif, commutatif et possède 0 comme élément neutre ; on peut donc le généraliser à un nombre quelconque d'opérandes d'entrée. Ainsi généralisé l'opérateur devient une *fonction qui vaut '1' quand il y a un nombre impair de '1' dans le mot d'entrée,*

d'où le symbole de la figure III-10, où la table de vérité concerne un opérateur à 4 opérandes[6].





OU EXLUSIF généralisé (XOR)

	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

Figure III-10

Complément du OU EXCLUSIF (XNOR) : L'opérateur OU EXCLUSIF a la particularité que pour obtenir son complément on peut, soit complémenter la sortie, soit complémenter l'une quelconque de ses entrées.

$$\overline{a \oplus b} = \bar{a} \oplus b = a \oplus \bar{b} = a * b + \bar{a} * \bar{b} = (a + \bar{b}) * (\bar{a} + b)$$

Comme opérateur à deux opérandes, ce nouvel opérateur indique *l'identité* entre les deux opérandes, d'où le nom parfois employé pour le désigner de « *coïncidence* ». Comme opérateur généralisé à un nombre quelconque d'opérandes, le complément du ou exclusif indique par un '1' qu'un nombre *pair* de ses opérandes vaut '1'.

Addition en binaire

Pour faire l'addition de deux nombres il faut savoir faire la somme de trois chiffres : les chiffres de rang n des deux opérandes et le report r_n issu de l'addition des chiffres de rang inférieur ; outre la somme il faut également générer le report r_{n+1} pour l'étage suivant. On appelle *additionneur complet* un tel opérateur (figure III-11).

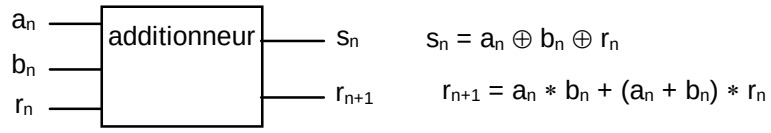


Figure III-11

[6]On notera le code particulier utilisé pour numéroter lignes et colonnes, c'est un code connu sous le nom de code de GRAY, ou code binaire réfléchi ; ce code a la particularité que l'on passe d'une combinaison à la suivante en changeant la valeur d'un seul chiffre binaire. Nous aurons l'occasion d'en reparler.

La réalisation de l'addition de deux nombres peut se faire en cascadant les opérateurs précédents, on parle alors de « propagation de retenue », ou en calculant « en parallèle » toutes les retenues, au prix d'une complexité non négligeable[7], on parle alors de « retenue anticipée ».

Erreurs : tests de parités

Lors de la transmission d'informations numériques entre deux sous-ensembles, il peut se produire des erreurs. On peut tenter de détecter, voire de corriger ces erreurs en rajoutant des redondances dans le message transmis. Ces redondances consistent à rajouter au contenu du message des bits supplémentaires élaborés conformément à une règle connue à la fois par l'émetteur et le destinataire du message. La technique la plus élémentaire, qui est très utilisée dans la transmission de caractères, codés en ASCII par exemple, consiste à rajouter un bit *de parité* calculé de telle façon que chaque caractère transmis, augmenté de cet élément de contrôle, contienne un nombre pair (parité paire, *even* parity) ou impair (parité impaire, *odd* parity) d'éléments binaires à '1'. La figure III-12 illustre le principe d'un émetteur qui utilise une convention de parité paire.

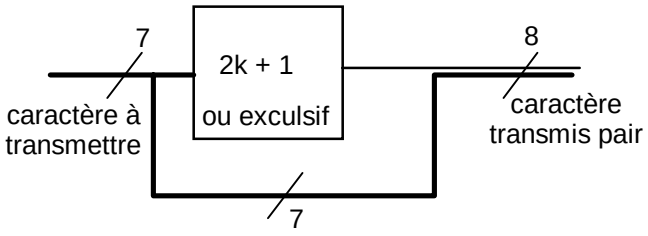


Figure III-12

Du côté du récepteur un schéma similaire permet de contrôler que la parité du caractère est bien conforme à la valeur prévue par le protocole de transmission.

Ce type de contrôle élémentaire ne permet de détecter que des erreurs simples ; si deux erreurs affectent le même caractère la parité du message reçu est correcte, le récepteur n'est alors pas pré-venu du problème.

En augmentant le nombre de clés de contrôle (les bits supplémentaires) il est possible de construire des codes autocorrecteurs, qui détectent et corrigent les erreurs de transmission, tant que leur nombre n'excède pas une valeur limite qui dépend du nombre de clés rajoutées.

Opérateur programmable

Quand on calcule une fonction combinatoire complexe, il peut être plus simple de calculer d'abord son complément et d'inverser le résultat. La plupart des circuits programmables offrent, pour ce faire, la possibilité de complémenter, ou non, la

[7]On consultera avec profit la notice d'un circuit comme le 74xx283.

sortie d'un opérateur au moyen d'un « fusible » de polarité. L'opérateur OU EXLUSIF permet de créer cette fonctionnalité, l'une de ses entrées est alors considérée comme une entrée de donnée, l'autre comme une commande de polarité, conformément au schéma de principe de la figure III-13.

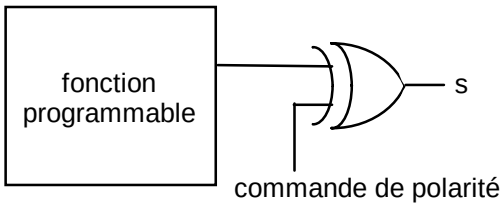


Figure III-13

Descriptions en VHDL

VHDL connaît l'opérateur XOR comme primitive ; les exemples qui suivent sont destinés à explorer, outre les propriétés de cet opérateur, des fonctionnalités du langage que nous n'avions pas abordées jusqu'ici.

Description structurelle

Ayant défini les opérateurs élémentaires ET, OU et NON comme précédemment, il est possible de les utiliser dans une construction plus complexe, comme le OU EXCLUSIF. L'exemple qui suit est, bien sûr, complètement académique, il est difficile d'imaginer plus compliqué pour réaliser un opérateur aussi simple !

```
ENTITY ouex IS  -- operateur OU exlusif
PORT ( a, b : IN  BIT ;
           s : OUT BIT );
END ouex;

use work.portelem.all ; -- rend visible le contenu de
                 -- portelem
ARCHITECTURE struct OF ouex IS
signal abar,bbar,abbar,abarb : bit;

BEGIN  -- les differents composants sont instancies ici
i1 : inverseur port map (a,abar);
i2 : inverseur port map (b,bbar);
et1 : et port map (a,bbar,abbar);
et2 : et port map (b,abar,abarb);
ou1 : ou port map (abbar,abarb,s);
```

```
END struct;
```

Pour que le programme précédent soit compris correctement, il a fallu, au préalable, créer et compiler le paquetage portelem et la description des opérateurs élémentaires qui y sont décrits comme suit :

```
package portelem is

component inverseur
PORT ( e : IN  BIT ; -- les entrees
       s : OUT BIT ); -- les sorties
END component;

component et
PORT ( e1, e2 : IN  BIT ;
            s : OUT BIT );
END component;

component ou
PORT ( e1, e2 : IN  BIT ;
            s : OUT BIT );
END component;

end portelem;
-- ce qui suit est la copie de programmes déjà vus
ENTITY inverseur IS
PORT ( e : IN  BIT ; -- les entrees
       s : OUT BIT ); -- les sorties
END inverseur;
ARCHITECTURE pleonasme OF inverseur IS
BEGIN
s <= NOT e;
END pleonasme;

-- operateur ET
ENTITY et IS
PORT ( e1, e2 : IN  BIT ;
            s : OUT BIT );
END et;
ARCHITECTURE pleonasme OF et IS
BEGIN
s <= e1 AND e2;
END pleonasme;

-- operateur OU
ENTITY ou IS
PORT ( e1, e2 : IN  BIT ;
            s : OUT BIT );
      END ou;
```

```
ARCHITECTURE pleonasme OF ou IS
BEGIN
s <= e1 OR e2;
END pleonasme;
```

L'addition élémentaire

L'opérateur OU EXCLUSIF n'est autre que l'opérateur d'addition en base deux, le programme suivant en est la conséquence directe :

```
-- operateur OU exlusif

ENTITY ouex IS
PORT ( a, b : IN  INTEGER RANGE 0 TO 1 ;
            s : OUT INTEGER RANGE 0 TO 1 );
END ouex;
ARCHITECTURE arith of ouex is
BEGIN
s <= a + b;
END arith;
```

La comparaison

Si deux opérandes binaires sont différents le résultat de l'opérateur OU EXCLUSIF est '1' :

```
-- operateur OU exlusif

ENTITY ouex IS
PORT ( a, b : IN  BIT ;
            s : OUT BIT );
END ouex;
ARCHITECTURE compare of ouex is
BEGIN
s <= '0' WHEN a = b ELSE '1';
END compare;
```

Indicateur de parité impaire

Nous terminerons cette découverte du OU EXCLUSIF par sa généralisation comme contrôleur de parité d'un mot d'entrée :

```
-- operateur OU exlusif generalise
ENTITY ouex IS
PORT ( a : IN  BIT_VECTOR(0 TO 3) ;
            s : OUT BIT );
END ouex;
ARCHITECTURE parite of ouex is
BEGIN
```

```
process(a)
variable parite : bit ;
begin
parite := '0';
FOR i in 0 to 3 LOOP
      if a(i) = '1' then
            parite := not parite;
      end if;
END LOOP;
s <= parite;
end process;
END parite;
```

Rien ne s'oppose, semble-t-il, à généraliser ce programme à un mot d'entrée de, mettons, 16 bits. Là se pose un petit problème : l'optimiseur du compilateur va tenter de « réduire » les équations logiques sous-tendues par la boucle « for » pour exprimer la fonction obtenue comme somme (logique) de produits (logiques). Mais il y a 32 768 produits logiques dans un contrôleur de parité sur 16 bits (2^{15}), d'où les dangers des descriptions abstraites....

Une solution plus raisonnable, mais, il est vrai, non optimale du point de vue vitesse de calcul est[8] :

```
ENTITY ouex4 IS  -- le même que précédemment
PORT ( a : IN  BIT_VECTOR(0 TO 3) ;
            s : OUT BIT );
END ouex4;
ARCHITECTURE parite of ouex4 is
BEGIN
process(a)
variable parite : bit ;
begin
parite := '0';
FOR i in 0 to 3 LOOP
      if a(i) = '1' then
            parite := not parite;
      end if;
END LOOP;
s <= parite;
end process;
END parite;

ENTITY ouex16 IS
PORT (e : IN BIT_VECTOR(0 TO 15);
      s : OUT BIT_VECTOR (0 TO 3);
-- force la conservation des signaux intermédiaires
```

[8]Le lecteur est instamment convié à dessiner un schéma logique en même temps qu'il lit le corps du programme.

```
      s16 : OUT BIT);  -- le résultat complet
END ouex16;

ARCHITECTURE struct OF ouex16 IS
SIGNAL inter : BIT_VECTOR(0 TO 3);
COMPONENT ouex4
PORT ( a : IN  BIT_VECTOR(0 TO 3) ;
           s : OUT BIT );
END COMPONENT;
BEGIN
par16 : for i in 0 to 3 generate
g1 : ouex4 port map (e(4*i to 4*i + 3),inter(i));
end generate;
g2 : ouex4 port map (inter,s16);
s <= inter;
END struct;
```

On notera l'intérêt des boucles « generate » pour créer des motifs répétitifs.

III.2.5 Le sélecteur, ou multiplexeur à deux entrées

Le lecteur attentif n'aura pas manqué de remarquer que beaucoup de choses, en logique combinatoire, peuvent s'exprimer par des alternatives SI... ..ALORS... ...AUTREMENT. Mais quel est donc l'opérateur élémentaire qui, en logique câblée, permet de matérialiser directement ce type de propositions ? Le *sélecteur*, ou *multiplexeur*. Nous donnons ci-dessous la description de sa version la plus simple, quand il n'y a que deux choix possibles dans l'alternative, mais il est bien sûr possible de le généraliser pour représenter des choix multiples (IF... ...THEN... ...ELSIF... ...ELSIF... ...END IF, ou, CASE... ...IS WHEN... ...WHEN... ...END CASE).

Description

Principe général

Le sélecteur est construit comme un opérateur où l'on sépare les variables d'entrée en deux groupes :

- Les entrées de *données*, qui sont en général issues d'autres fonctions ;
- L'entrée de sélection, qui est une *commande*.

Prenons un exemple. Pour faire l'addition de deux chiffres décimaux, codés en BCD, il faut commencer par faire l'addition de ces deux chiffres, sans se poser de question, comme s'il s'agissait de nombres écrits en base 2. Deux éventualités peuvent alors se produire :

1. La somme est inférieure à 10, l'opération est alors terminée.

2. La somme est supérieure ou égale à 10, ce résultat n'est alors pas correct en BCD. Il faut lui rajouter l'écart entre un nombre binaire sur 4 bits (0 à 15) et un chiffre décimal (0 à 9), soit 6.

Résumons ce qui précède sous forme d'un algorithme :

a et b sont les deux chiffres à additionner, s est le résultat.
s = a + b
si s < 10 terminé
autrement s = s + 6.

Une structure de la réalisation câblée de ce qui précède pourrait être celle de la figure III-14 :

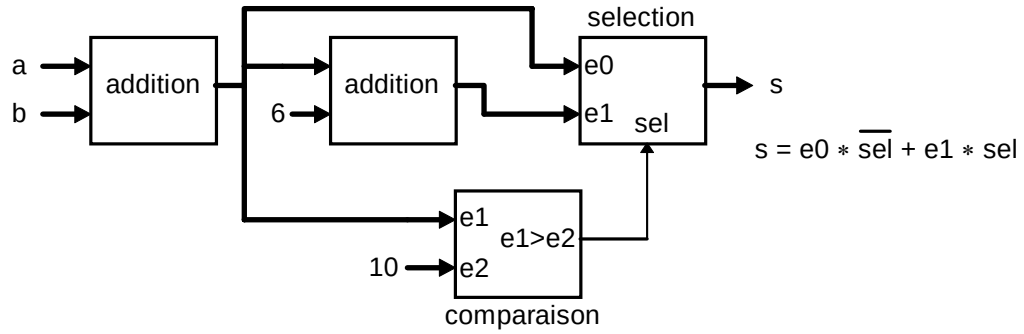


Figure III-14

Symbole et logigramme

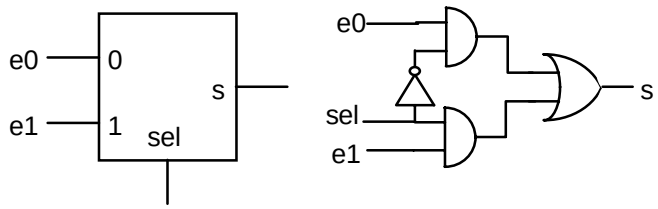


Figure III-15

Le multiplexeur élémentaire est souvent représenté par un symbole qui indique les valeurs de l'entrée de sélection à côté des entrées de données correspondantes (figure III-15).

Code source VHDL

Le multiplexeur à deux entrées est l'élément de base des descriptions dans des langages comme VHDL, nous n'en donnerons que quelques exemples :

Quelques tautologies

Nous retrouvons ici la définition même d'un multiplexeur.

```
entity sel is
port ( e0,e1,sel  : in bit;
               s  : out bit);
end sel;

architecture pleonasme of sel is
begin
with sel select
      s <= e0 when '0',
           e1 when '1';
end pleonasme;
```

ou :

```
entity selecteur is
port ( e0,e1,sel  : in bit;
               s  : out bit);
end selecteur;

architecture procif of selecteur is
begin
process (sel)
begin
if(sel = '0') then
      s <= e0 ;
else
      s <= e1;
end if;
end process;
end procif;
```

ou encore :

```
entity selecteur is
port ( e0,e1,sel  : in bit;
               s  : out bit);
end selecteur;

architecture pleonasme of selecteur is
begin
      s <= e0 when (sel = '0') else e1;
end pleonasme;
```

Une autre façon de voir : les tableaux

VHDL connait les types structurés, la recherche d'un élément d'un tableau se traduit, en logique câblée, par un multiplexeur :

```
entity selecteur is
port ( e    : in bit_vector(0 to 1);
       sel  : in integer range 0 to 1;
       s    : out bit);
end selecteur;

architecture vecteur of selecteur is
begin
       s <= e(sel);
end vecteur;
```

ou, en généralisant :

```
entity selecteur is
port ( e    : in bit_vector(0 to 7);
       sel  : in integer range 0 to 7;
       s    : out bit);
end selecteur;

architecture vecteur of selecteur is
begin
       s <= e(sel);
end vecteur;
```

III.3. Opérateurs séquentiels

Nous avons déjà évoqué l'importance de la notion de *mémoire*, ce qui différencie un opérateur séquentiel d'un opérateur combinatoire réside dans la capacité du premier à « se souvenir » des événements antérieurs : une même combinaison des entrées, à un certain instant, pourra avoir des effets différents suivant les valeurs des combinaisons précédentes de ces mêmes entrées. Pour traduire cet effet de mémoire on introduit la notion d'*état interne* de l'opérateur, l'action des entrées est alors de provoquer d'éventuels changements d'état, la situation qui suit le changement de l'une d'elles dépend des valeurs des entrées et de l'état initial de l'opérateur ; si le nouvel état est différent du précédent on dit qu'il y a eu une *transition.*

Les opérateurs séquentiels peuvent être classés en deux grandes familles qui se différencient par la façon dont peuvent arriver les transitions :

1. Les opérateurs *asynchrones*, les plus anciens et les plus simples, sont de simples circuits combinatoires sur lesquels on introduit une

réaction positive. L'une des entrées de l'opérateur, soit e_r , est connectée à une sortie, soit s_r ; de plus l'opérateur $s_r = f(e_r)$ présente une caractéristique *non inverseuse*[9]. La transition d'un état à un autre est provoquée par des changements de *niveaux* d'une ou plusieurs entrées.

2. Les opérateurs *synchrones*, plus complexes, utilisent plusieurs opérateurs asynchrones pour mémoriser leur état. L'idée qui préside à la réalisation des opérateurs synchrones est de les munir d'une entrée très particulière, *l'horloge*, dont *un front*, montant ou descendant, fixe les instants où les transitions entre états sont effectuées. En général, cette horloge est le signal issu d'un générateur d'impulsions périodiques, le même pour tous les opérateurs d'un système, qui fixe une cadence de travail commune à tous les éléments du groupe. *Entre deux transitions* actives du signal d'horloge le système est *figé*, il ne peut en aucun cas changer d'état ; c'est ce temps de latence qui est mis à profit pour permettre aux transitoires de calcul des circuits, combinatoires notamment, de se terminer sans influencer de façon aléatoire le comportement du système.

Nous commencerons la description des opérateurs séquentiels par les premiers, en raison de leur simplicité, bien que les seconds soient, et de loin, les plus utilisés.

III.3.1 Les bascules asynchrones

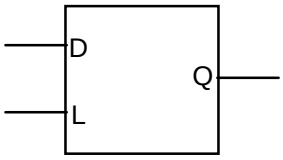
Les deux principaux types de bascules asynchrones sont la bascule D *Latch*, et la bascule R - S. La première sert simplement à mémoriser une donnée D, les entrées de la seconde doivent plutôt être comprises comme des commandes qui spécifient une action.

La bascule D « Latch », ou verrou

La bascule D-latch constitue la version la plus simple de la mémoire élémentaire.

[9]L'étude générale de la réaction nous apprend qu'une réaction positive, en continu, conduira à un système qui présente deux états stables et un état instable non oscillatoire. Une réaction négative, par contre, conduira à un état stable ou à une instabilité de type oscillatoire.

Le principe



C’est un opérateur à deux entrées et une sortie :

une entrée D de donnée,
une entrée L de commande
une sortie Q, état de l’opérateur.

Le fonctionnement est extrêmement simple : si L est au niveau haut (L = '1') la sortie prend la valeur de l’entrée D (Q = D), c’est le mode *transparent*, si L est au niveau bas (L = '0') la sortie conserve sa valeur quelle que soit celle de l’entrée D, c’est le mode *mémoire*.

Il est clair que les niveaux actifs de l’entrée de commande L peuvent être inversés.

Un exemple de réalisation

La description qui précède suggère immédiatement une réalisation qui fait appel à un multiplexeur élémentaire, ou, ce qui revient au même, à la traduction en logigramme de l’équation qui traduit cette description (figure III-16) :

$$Q = L*D + \bar{L}*Q$$

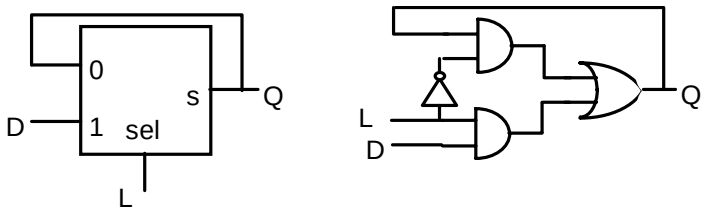


Figure III-16

L’équation précédente mérite quelque explication, quand L = '0', elle ressemble à s’y méprendre à une tautologie (Q = Q) dont on peut se demander ce qu’elle veut dire.

La figure ci-dessous (III-17) correspond au cas où L = '0', il y apparaît clairement que la bascule, en mode mémoire, se comporte comme un système bouclé. Pour analyser le type de réaction mise en jeu, une méthode simple consiste à « ouvrir la boucle », à tracer la caractéristique de transfert Q = f(E), où E est l’entrée du système en boucle ouverte, et à chercher l’intersection de cette caractéristique avec la droite d’équation Q = E.

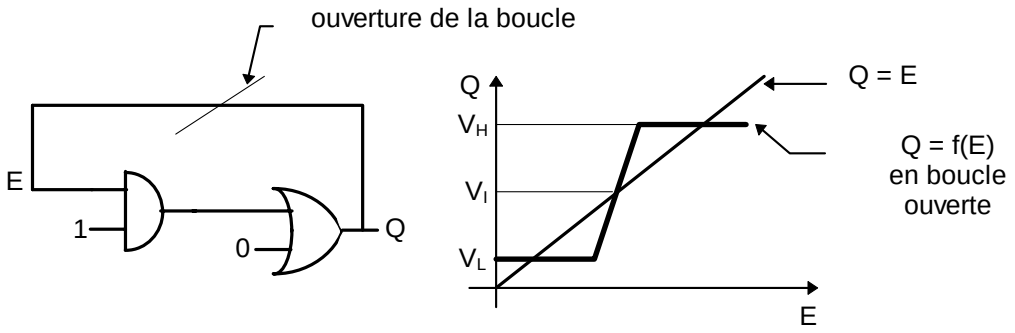


Figure III-17

Le système d'équations associé à cette construction graphique a trois solutions :

- $Q = V_L$ et $Q = V_H$, qui correspondent à deux états logiques possibles, sont des solutions stables. Si l' entrée D de la bascule place celle ci dans l'un de ces deux états, quand L = '1', le circuit conservera son état dans le mode mémoire (L = '0'), quelle que soit la valeur de D.
- $Q = V_I$ est une solution instable qui correspond à un état analogique intermédiaire. Si la bascule se trouve accidentellement dans cet état, elle évoluera vers l'un ou l'autre des états stables[10].

Quelques descriptions en VHDL

VHDL ne connait pas la fonction mémoire comme élément primitif. Une bascule asynchrone est générée soit par une description structurelle, soit par une description comportementale exhaustive, c'est à dire qui comprend la description explicite du mode mémoire, soit, *et cela constitue, pour les débutants, un piège du langage*, par une description incomplète des alternatives d'une instruction « IF ».

```
entity selecteur is  -- déjà vu précédemment
port (      a0,a1,sel : in bit;
      s     :   out bit);
end selecteur;

architecture pleonasme of selecteur is
begin
      s <= a0 when (sel = '0') else a1;
end pleonasme;

entity d_latch is
port ( D,L : in bit;
```

[10]Voir à ce sujet au paragraphe II.3.2.3 la présentation des états métastables dans les circuits synchrones, l'existence de ces états est due à cette troisième solution $Q = V_I$ dans les bascules asynchrones qui servent à réaliser une bascule synchrone.

```
      Q : out bit);
end d_latch;

architecture struct of d_latch is
 -- description structurelle
component selecteur
port ( a0,a1,sel : in bit;
      s : out bit);
end component;
signal reac : bit;
begin
Q <= reac;
s1 : selecteur port map(reac,D,L,reac);
end struct;
```

Le code qui précède n'est que la traduction naïve du premier schéma de la figure III-16. Les architectures qui suivent, qui décrivent la même entité, sont plus synthétiques :

```
architecture d_flow of d_latch is
signal reac : bit;  --  le signal de réaction
begin
Q <= reac;
reac <= D when L = '1'
          else reac; -- explicite le mode mémoire.
end d_flow;
```

Donnons enfin une forme de description qui génère le mode mémoire par omission d'une combinaison dans une alternative « IF ». La possibilité de ce type de construction présente le danger qu'elle est parfois le résultat d'un réel oubli du programmeur, et non d'une volonté de sa part :

```
architecture behav of d_latch is
signal reac : bit;
begin
Q <= reac;
process(L,D)
begin
if(L = '1') then
      reac <= D ;
end if; -- L'omission du cas où L = '0'
        -- génère le mode mémoire.
end process;
end behav;
```

Les applications

La simplicité de la constitution interne d'une bascule D-Latch en fait l'élément constitutif des mémoires statiques. Le comportement purement combinatoire de cette structure, quand elle est en mode transparent, *en interdit l'usage dans tout système qui contient des rebouclages des sorties sur les entrées*. Nous verrons dans la suite que la plupart des fonctions séquentielles font usage de tels rebouclages. Dit autrement, une bascule de ce type doit toujours être commandée en boucle ouverte, ses entrées de donnée et de commande ne doivent en aucun cas être des fonctions combinatoires de sa sortie.

A titre d'illustration, citons une application classique des bascules D-Latch dans le démultiplexage temporel d'une information : Certains microcontrôleurs utilisent les mêmes broches du circuit pour véhiculer des informations d'adresses et de données ; les mémoires, elles, attendent une adresse stable pendant toute la durée du transfert de données. Pour permettre de résoudre le conflit, le microcontrôleur fournit un indicateur, ALE (pour adress latch enable), qui indique à la périphérie que l'information disponible est une adresse. Un registre constitué de bascules D-Latch permet alors de reconstituer l'information d'adresse dont les mémoires ont besoin, conformément aux chronogrammes de la figure III-18 :

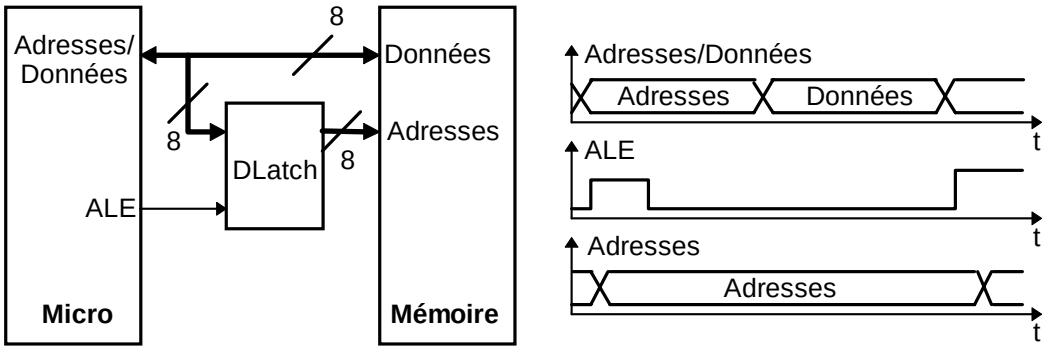
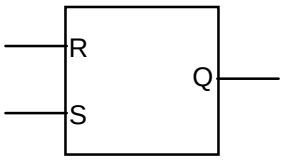


Figure III-18

La bascule R-S

Le principe

Comme la précédente, la bascule R-S est une cellule mémoire qui peut prendre deux états; le passage d'un état à l'autre est cette fois dirigé par deux entrées de commande, R et S :



l'entrée R, pour reset, provoque la mise à zéro de la sortie Q,
l'entrée S, pour set, provoque la mise à un de Q.
Quand aucune des deux commandes n'est active la bascule est en mode mémoire, quand les deux sont actives une priorité de l'une des entrées sur l'autre sera observée.

Un exemple de réalisation

Si on fixe les niveaux actifs à '1', et que l'on privilégie la mise à zéro de Q (si R et S sont simultanément actifs, R l'emporte), la traduction en équation logique de la description précédente conduit à :

$$Q = \overline{R} * S + \overline{R} * Q$$

Qui correspond au premier logigramme de la figure III-19. Le deuxième logigramme de cette figure correspond à un fonctionnement où R et S sont actifs à '0'. RS = "11" provoque le mode mémoire et la priorité est à la mise à '1' de Q si les deux commandes sont actives simultanément. On notera que, comme dans le cas de la bascule D-Latch, le mode mémoire correspond à une réaction positive sur l'ensemble du montage.

Si l'on s'impose la contrainte de *ne jamais* rendre actives les deux commandes simultanément, les sorties des deux opérateurs (nor ou nand) sont toujours complémentaires, le même schéma fournit alors Q et $\overline{Q}$[11].

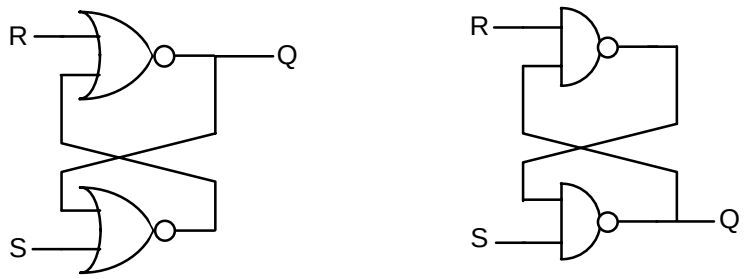


Figure III-19

[11]On rencontre parfois, et à tort, le terme de combinaison « interdite » pour la situation où les deux commandes sont actives simultanément. Les auteurs de ces lignes n'ont jamais vu de bascule exploser dans une telle situation, et ils l'ont pourtant maintes fois provoquée, bien involontairement il est vrai, en utilisant des bascules D-Edge qui sont constituées de trois bascules R-S et où cette fameuse combinaison interdite est utilisée.

La bascule RS met bien en évidence la difficulté majeure des systèmes asynchrones : si les deux commandes passent *simultanément* de l'état actif à l'état inactif (mémoire), le *résultat est imprévisible*, c'est ce qu'on appelle classiquement un *aléa*. Un phénomène analogue existe évidemment dans les bascules D-Latch, mais choque moins en raison de la dissymétrie fonctionnelle des deux entrées D et L.

Quelques descriptions en VHDL

```
entity rs is port (
R,S : in bit;
Q  : out bit);
end rs;

architecture df of rs is
signal etat : bit;
begin
q <= etat;
with R&S select
-- l'opérateur & concatène les signaux R et S
etat <=     '1' when "01",
            '0' when "10",
            etat when "00", -- mode mémoire explicite
            '0' when "11";
end df;
```

Ayant utilisé, pour décrire une bascule D-Latch, une instruction « IF » incomplète, nous donnerons ci-dessous l'exemple d'une instruction « CASE » pour générer le mode mémoire :

```
architecture bv of rs is
begin
process (R,S)
begin
   case R&S is
      when "01"        => Q <= '1';
      when "10" | "11" => Q <= '0'; -- '|' est un
                                    -- ou logique
      when others null ; -- mode mémoire : pas
                         -- d'action sur Q.
   end case;
end process;
end bv;
```

Noter que, si toutes les combinaisons de l'expression testée ne sont pas mentionnées explicitement, l'alternative « others » est obligatoire ; l'instruction « CASE » ne peut pas être incomplète. L'instruction « null » traduit l'absence

d'action, par défaut la valeur de Q est conservée, ce qui correspond bien à une cellule mémoire.

Les applications

La première application des bascules R-S réside dans les commandes de « forçage » à un ou à zéro des systèmes séquentiels, des plus simples aux plus complexes, à la mise sous tension notamment. Beaucoup de circuits offrent, à cet effet, un mode de fonctionnement de type « RS » en plus du fonctionnement normal, synchrone dans la plupart des cas. La réinitialisation matérielle (hard reset) d'un ordinateur, par exemple, correspond à une réinitialisation asynchrone (i.e. indépendante de l'horloge) de certains registres critiques du processeur, charge au programmeur d'avoir prévu la suite des événements.

Une autre application classique des bascules RS est l'élimination des rebonds des interrupteurs : Quand un interrupteur passe d'un état à un autre (ouvert ou fermé), il se produit généralement un régime transitoire oscillatoire où ces deux états se succèdent avant que l'un d'eux soit stable. Une solution classique consiste à remplacer les interrupteurs par des commutateurs, objets à deux états mécaniquement stables, F1 et F2, et un état mécaniquement instable où les deux contacts sont ouverts, deux résistances et une bascule RS, conformément au schéma de la figure III-20, dont nous laisserons l'étude détaillée à la sagacité du lecteur[12].

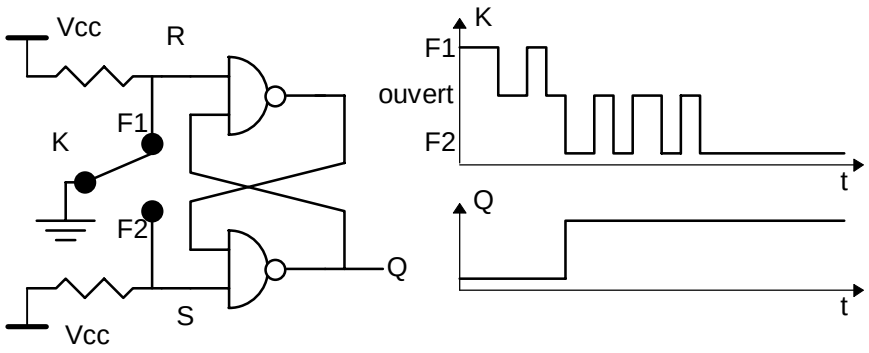


Figure III-20

III.3.2 Les bascules synchrones

Nous avons évoqué, à propos de la bascule R-S, qu'une difficulté arrive, dans les circuits séquentiels, quand plusieurs entrées susceptibles de provoquer des

[12]Guide de raisonnement : quand un commutateur passe de F1 à F2 il oscille entre F1 **ou** F2 et l'état intermédiaire où les deux contacts sont ouverts.

modifications d'états des cellules mémoire changent de niveau simultanément. Une première idée consiste à interdire de telles situations, cette politique a eu cours à une époque, mais la complexité croissante des fonctions logiques a vite mis en évidence l'inanité d'une telle solution qui compliquerait formidablement la conception du moindre circuit[13].

La solution couramment adoptée est de construire les fonctions complexes avec des bascules synchrones, qui disposent d'une entrée *réservée et unique* qui fixe les instants des éventuels changements d'états : l'horloge.

Des changements d'état bien ordonnés : l'horloge

Principe

L'idée est de dissocier les moments de définition d'une commande des instants où elle est exécutée. Pour fixer les idées, le signal d'horloge est en général issu d'un générateur d'impulsions périodiques. Une bascule peut changer d'état uniquement lors d'une *transition* (montante, par exemple) de ce signal, *l'état obtenu après* la transition active étant déterminé par la *commande présente avant* la dite transition. Le corrolaire de ce principe est évidemment que la commande doit être stable pendant un intervalle de temps non nul situé juste avant chaque transition active[14]. De très nombreux systèmes, de la vie courante, utilisent ce principe de commande à exécution différée, citons, en vrac : le rôle d'un chef d'orchestre, le pistolet qui marque le début d'une course à pieds, l'adjudant qui tente d'obtenir un quart de tour à droite d'un peloton de soldats lors d'un défilé etc.

Le fonctionnement interne des bascules synchrones relève de l'électronique analogique, et fait intervenir les temps de propagation des signaux dans les transistors, mais *ces aspects ne doivent en aucun cas être nécessaires à la compréhension du fonctionnement logique d'un système*.

La simplification apportée au concepteur par les bascules synchrones n'apparaîtra que progressivement, chaque bascule est un objet électronique plus compliqué[15], la simplification n'arrive que quand se pose la question de réaliser des fonctions complexes, qui mettent en oeuvre un grand nombre de bascules en interactions. Pour résumer on peut dire que le temps qui sépare deux fronts actifs de l'horloge (une période de celle-ci) est mis à profit par la logique classique (qui traite des niveaux) pour effectuer ses calculs, peu importe alors l'ordre dans lequel les nouvelles commandes sont élaborées, pourvu que tous les calculs soient terminés avant que n'arrive une nouvelle transition.

---

[13]L'élimination des aléas dans les fonctions séquentielles n'a, en fait, d'intérêt que pour la conception d'une bascule synchrone qui deviendra la brique élémentaire de tout l'édifice.

[14]Pour plus de précision voir le paragraphe II.3.2.

[15]Il faut deux ou trois bascules asynchrones pour réaliser une bascule synchrone, voir, par exemple, les schémas internes de la bascule D 74xx74 dans les technologies bipolaires et CMOS.

Chronogrammes et symboles

Les symboles couramment utilisés mettent en évidence le rôle particulier de l'horloge (Ck pour clock) par un triangle qui indique que cette commande agit par ses fronts, de montée ou de descente, plutôt que par des niveaux logiques, comme les entrées ordinaires (figure III-21) :

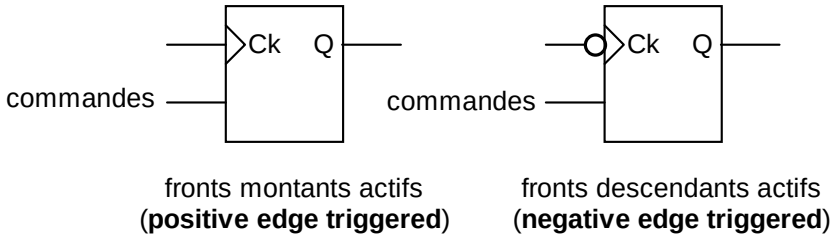


Figure III-21

Dans un ensemble synchrone le *temps devient une variable discrète,* par opposition à continue, on peut le mesurer par un nombre entier, l'unité de mesure étant la période de l'horloge. L'état d'une bascule à l'instant t est une fonction combinatoire f de la commande et de l'état à l'instant t – 1 :

$$Q(t) = f(Q(t - 1) , commande(t - 1))$$

D'où un chronogramme de principe de la figure III-22 :

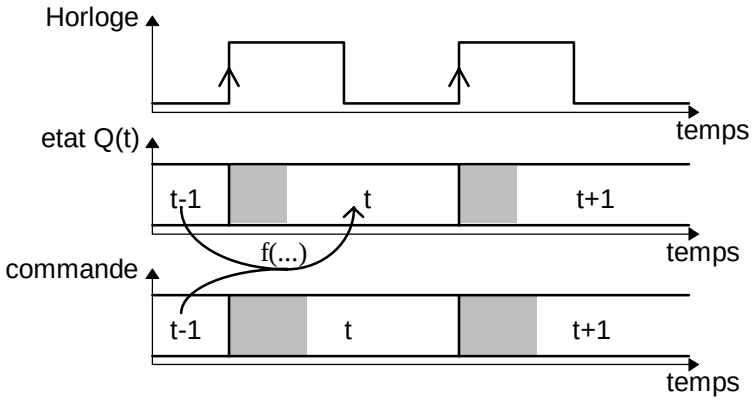


Figure III-22

Sur ces chronogrammes on a souligné par une zone grise les moments où commandes et état d'une bascule ne sont pas forcément bien déterminés, mais il s'agit là d'une simple illustration. En aucun cas le contenu de ces zones n'est nécessaire à la compréhension du principe de fonctionnement.

Diagrammes de transitions

Pour représenter de façon visuelle le fonctionnement des bascules synchrones, tout en mettant en évidence les notions centrales de la logique séquentielle que sont les états et les transitions, on utilise souvent des *diagrammes de transitions entre états* (*state transition diagram*), ou, pour abréger, diagrammes de transitions ou diagrammes d'états (figure III-23).

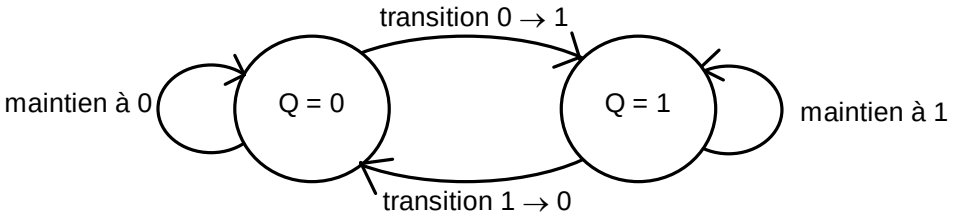


Figure III-23

Dans un tel diagramme un cercle représente un état (une bascule en a deux), une flèche une transition entre deux états (qui peut être un maintien dans l'état initial). Pour qu'une transition soit effectuée *trois* conditions doivent être vérifiées :

1. La bascule doit être dans l'état de départ,
2. il doit y avoir un front actif du signal d'horloge,
3. les entrées de commande autre que l'horloge doivent autoriser la transition.

En général le signal d'horloge est implicite, mais il ne faut bien sûr pas oublier cette condition sine qua non. La description (analyse) ou la création (synthèse) d'une bascule revient donc à préciser les équations logiques (quatre au maximum pour une bascule) qui définissent les transitions en fonction des commandes.

Une précision qui concerne VHDL

« VHDL ne contient pas le concept de signal d'horloge. Le moyen d'introduire un signal d'horloge dans vos ouvrages (designs) est d'utiliser une instruction "WAIT" dans un processus, ou d'utiliser une description structurelle »[16]!

Cela a le mérite d'être clair, il vaut mieux être prévenu.

[16]Warp2, VHDL Development system, Reference Manual, *Cypress Semiconductor*.

Pour illustrer ce qui précède on donne ci-dessous la structure d'un programme qui décrit une bascule :

```
entity basc_synchrone is port (
clock : in bit;
commande : in bit_vector( ... );
q: out bit);
end basc_synchrone;

architecture fsm of basc_synchrone is
signal etat : bit;
begin
q <= etat;
process
begin
wait until (clock = '1') -- tout est là
case etat is
      when '0' =>
  -- conditions de la transition '0'->'1'
      when '1' =>
  -- conditions de la transition '1'->'0'
end case;
end process;
end fsm;
```

D'autres constructions équivalentes existent, qui utilisent une liste de « sensibilité » dans la description du processus, nous aurons l'occasion de les examiner dans la suite. Le point important à noter est que *seuls les processus* permettent de générer à partir d'une description comportementale la synthèse d'une fonction qui utilise des bascules synchrones.

L'élément fondateur : la bascule D

Le principe

La bascule D synchrone, plus laconiquement D-edge, est la cellule mémoire fondamentale. Munie d'une entrée de donnée (en général notée « D »), et, naturellement, d'une entrée d'horloge, elle prend, à chaque transition active de l'horloge, l'état dont la valeur est celle de l'entrée de donnée. L'équation générale des bascules synchrones devient, dans ce cas, extrêmement simple :

$$Q(t) = D(t - 1) \quad \text{où t est la période d'horloge considérée.}$$

En notation abrégée, mais trompeuse car les deux membres de l'équation *ne sont pas* pris au même instant, on écrit parfois cette équation :

$$Q = D$$

Le symbole, le diagramme de transition et un exemple de chronogramme qui illustre le fonctionnement sont indiqués sur la figure III-24 :

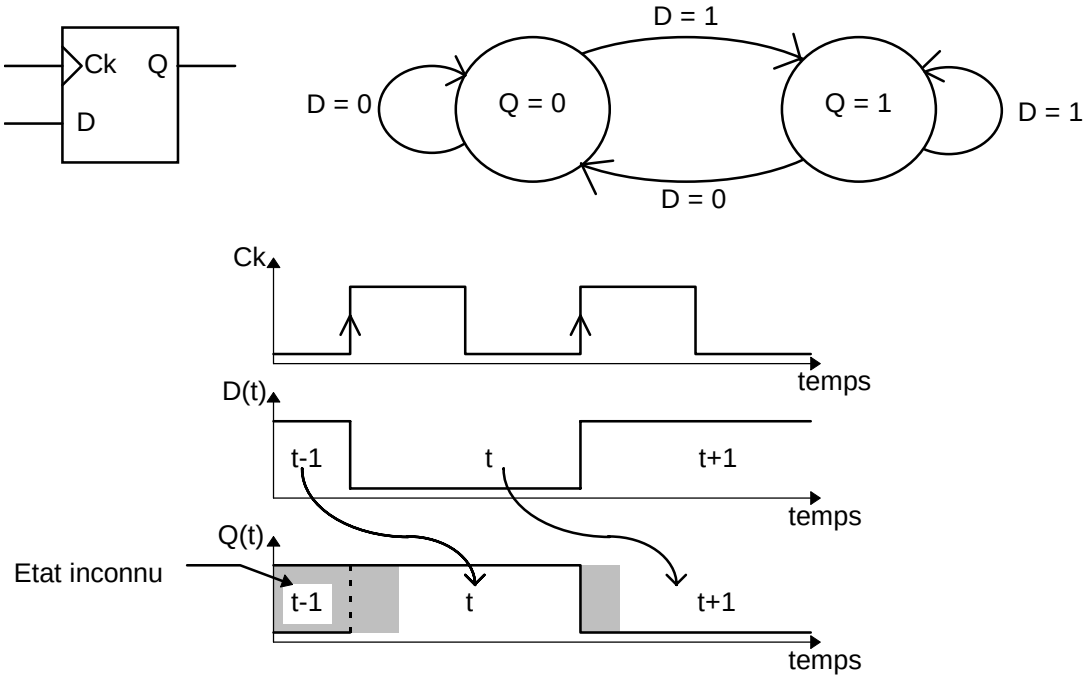


Figure III-24

On notera, dans le diagramme de transitions, le lien qui est indiqué entre les changements (ou le maintien) d'état et l'entrée de commande D. Dans la figure précédente on a pris la précaution de noter qu'à priori l'état initial de la bascule est inconnu. Ce point est important à garder en mémoire quand on se pose un problème de synthèse de système séquentiel.

Un exemple de réalisation

La réalisation interne d'une bascule D-edge *n'est en général pas* le souci du concepteur d'un ensemble logique, la bascule en question est un opérateur primitif, au même titre qu'une porte ET. Le schéma ci dessous, figure III-25, est donné à titre d'information.

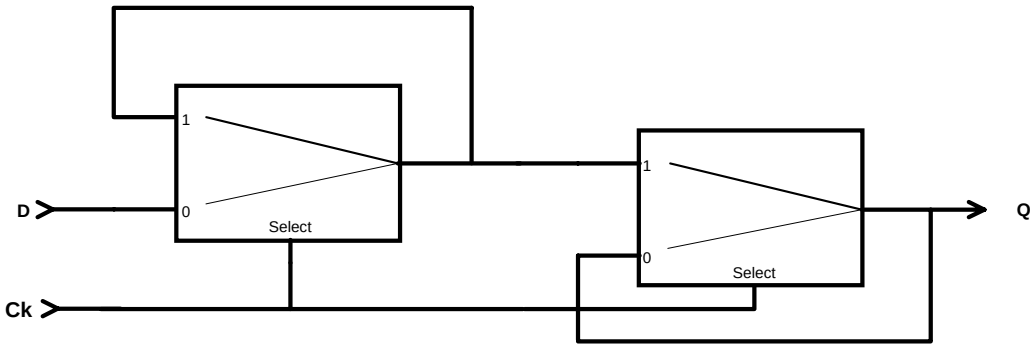


Figure III-25

Les deux multiplexeurs sont connectés en bascules D-Latch, avec des niveaux actifs inversés pour le mode mémoire. Quand l'horloge est au niveau bas la cellule de sortie est en mode mémoire, donc insensible aux variations éventuelles de son entrée, la cellule d'entrée en mode transparent. Quand l'horloge passe au niveau haut la cellule d'entrée mémorise la donnée présente, et la transfère dans la cellule de sortie. Le bon fonctionnement de l'ensemble est en fait assuré par l'existence de temps de commutation non nuls des multiplexeurs.

Cette technique de réalisation d'une bascule D, différente de celle utilisée pour la 74xx74 des familles TTL, est employée, par exemple, dans les circuits programmables (FPGAs) TPC12 (Texas Instrument).

Description en VHDL

Les deux exemples qui suivent, quoique des plus simples, sont à méditer attentivement. Ils représentent *les deux seules façons sûres* d'obtenir d'un compilateur VHDL la génération d'une bascule D synchrone générique (i.e. qui ne soit pas reconstruite au moyen de portes, ou, pire, qui ne soit pas une bascule D-Latch).

```
entity d_edge is
port ( d,hor : in bit;
       s : out bit);
end d_edge;

architecture d_primitive of d_edge is
begin
process
begin
wait until hor = '1';
s <= d ;
end process;
end d_primitive;
```

Et une variante qui remplace l'instruction « WAIT » par une liste de sensibilité du processus et un test sur *l'existence d'une transition* du signal d'horloge *et le niveau qui suit* cette transition.

```
architecture d_primitive1 of d_edge is
begin
process(hor) -- Le process ne « réagit » qu'au signal hor.
begin
if(hor'event and hor = '1') then
-- attention ! deux conditions
      s <= d ;
end if;
end process;
end d_primitive1;
```

L'omission du facteur « hor'event » dans le test conduit certains compilateurs à générer une bascule D-Latch.

La deuxième des deux formes présentées ci-dessus est un peu plus compliquée que la première, mais plus souple. Elle permet en effet d'inclure une commande d'initialisation asynchrone, reset dans l'exemple qui suit, à une bascule D synchrone. La méthode utilisée dans cet exemple présente cependant un certain danger, les compilateurs sont toujours accompagnés d'optimiseurs qui modifient éventuellement les polarités des signaux internes, en utilisant les lois de De Morgan. L'utilisateur peut alors avoir la désagréable surprise de découvrir qu'une remise à zéro asynchrone se traduit parfois par une mise à un de la sortie attachée à la bascule visée !

```
entity d_edge is
port ( d,hor,reset : in bit;
      s : out bit);
end d_edge;

architecture d_primitive of d_edge is
begin
process(hor,reset)
begin
if(reset = '1') then
      s <= '0';
elsif(hor'event and hor = '1') then
      s <= d ;
end if;
end process;
end d_primitive;
```

Terminons ce tour d'horizon des descriptions en VHDL d'une bascule D par la traduction dans ce langage du schéma construit au moyen de multiplexeurs :

```
entity d_edge is
port ( d,hor : in bit;
       s : out bit);
end d_edge;

architecture d_flow of d_edge is
signal sort,entre : bit;
begin
s <= sort;
entre <=  d when hor = '0' else entre;
sort  <=  entre when hor = '1' else sort;
end d_flow;
```

A n'utiliser qu'en dernier recours, quand on a épuisé toutes les « vraies » bascules D disponibles dans un circuit.

Les applications

Les bascules D sont la clé de voûte de toutes les applications séquentielles. Des quelques bascules (4 ou 8) couramment rencontrées dans les circuits standard de la famille TTL, on passe à plus de mille dans les « gros » circuits programmables.

Focalisée sur les transitions : la bascule T

Le principe

L'une des difficultés d'emploi des bascules D dans certaines applications réside dans le fait que la condition de maintien à '1' de son diagramme de transition ne doit pas être omise dans les équations obtenues pour la commande D, ce qui complique parfois notablement ces équations.

La bascule T (T pour Toggle, c'est à dire bascule) est un élément qui interprète son unique entrée de commande (en plus de l'horloge, évidemment), T, non comme une donnée à mémoriser, mais comme un ordre de changement d'état :

- ⇒ Si T = "actif" changer d'état à la prochaine transition de l'horloge,
- ⇒ si non conserver l'état initial.

D'où l'équation qui décrit son fonctionnement :

$$Q(\mathbf{t}) = T * \overline{Q(\mathbf{t\text{-}1})} + \overline{T} * Q(\mathbf{t\text{-}1})$$

On peut éclairer l'équation précédente par le diagramme de transitions de la figure III-26, ci-dessous :

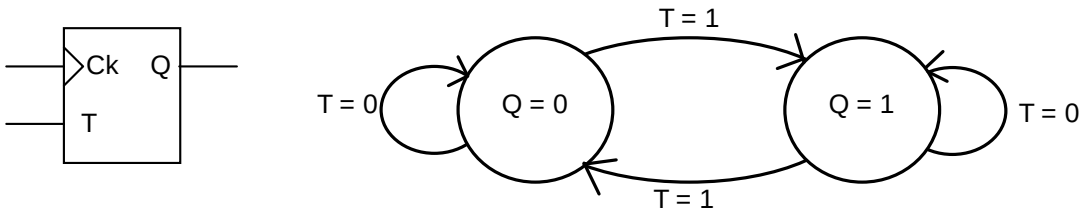


Figure III-26

Un exemple de réalisation

L'examen du diagramme de transitions de la figure III-26 nous montre que Q(t) = 1 si

Q(t-1) = '1' et T = '0',

ou

Q(t-1) = '0' et T = '1'

Ce qui nous fournit l'équation de l'entrée D d'une bascule D :

$$D = T \oplus Q$$

D'où le logigramme :

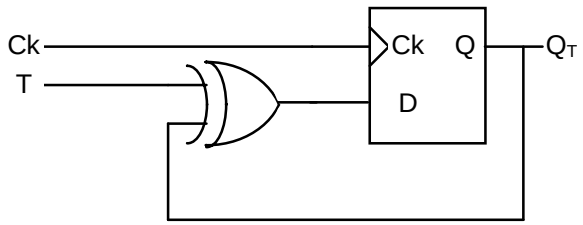


Figure III-27

Description en VHDL

La description d'une bascule T se déduit simplement du diagramme de transition :

```
entity T_edge is
```

```
port ( T,hor: in bit;
      s : out bit);
end T_edge;

architecture d_primitive of T_edge is
signal etat : bit;
begin
s <= etat ;
process
begin
wait until hor = '1' ;
if(T = '1') then
      etat <= not etat;
end if;
end process;
end d_primitive;
```

On rappelle que de tels exemples sont fournis à titre d'illustration du fonctionnement de l'opérateur considéré, et pour familiariser le lecteur avec le langage VHDL. On n'a jamais besoin, en pratique, de décrire chaque bascule utilisée dans ce langage !

Les applications

L'un des intérêts principaux des bascules de type T est qu'elles permettent de générer de façon extrêmement simple des compteurs binaires synchrones. Un compteur binaire est une fonction séquentielle synchrone dont l'état interne est un nombre entier naturel codé en binaire dont chaque chiffre binaire est matérialisé par une bascule. A chaque transition active de l'horloge ce nombre est incrémenté de 1, quand le nombre maximum est atteint, toutes les bascules sont à 1, la séquence recommence à partir de 0. Si le nombre de bits utilisés est n, on parlera d'un compteur modulo 2^n. La simple observation d'une table des entiers naturels écrits en base 2 nous fournit la clé du problème : la bascule de rang i doit changer d'état quand toutes les bascules de rang inférieur sont à 1.

D'où un exemple de réalisation d'un compteur synchrone modulo 16 dont on peut interrompre le comptage (en = '0') :

```
ENTITY cnt16 IS
PORT (ck, en : IN BIT;
      s : OUT BIT_VECTOR (0 TO 3)
      );
END cnt16;

ARCHITECTURE structurelle OF cnt16 IS
SIGNAL etat : BIT_VECTOR(0 TO 3);
SIGNAL inter: BIT_VECTOR(1 TO 3);
COMPONENT T_edge
-- la même que dans l'exemple précédent
```

```
port ( T,hor: in bit;
       s : out bit);
END COMPONENT;

BEGIN
-- Etablir le logigramme tout en lisant le texte
s <= etat ;
inter(1) <= etat(0) and en ;
inter(2) <= etat(1) and inter(1) ;
inter(3) <= etat(2) and inter(2) ;
g0 : T_edge port map (en,ck, etat(0));
g1 : for i in 1 to 3 generate
       g2 : T_edge port map (inter(i),ck,etat(i));
end generate;
END structurelle;
```

Là encore mettons en garde le lecteur, quand on a réellement besoin d'un compteur on écrit

```
etat <= etat + 1 ;
```

c'est nettement plus simple, le compilateur générera de lui même les interconnexions nécessaires entre les bascules.

L'ancêtre vénérable : la bascules J-K

Le principe

Quelque peu tombée en désuétude, la bascule J-K a régné en maître dans le monde de la logique séquentielle des décennies 60 et 70. Elle est l' héritière directe de la bascule R-S, que l'on a débarrassé progressivement de ses difficultés asynchrones. Les premières bascules J-K n'étaient, en fait, pas de réels opérateurs synchrones, elles comportaient tout un mécanisme de mémorisation interne des commandes dans des bascules R-S (maître-esclave). Les versions actuelles sont construites au moyen d'une bascule D-edge et de logique combinatoire.

Une bascule J-K dispose de deux commandes, J et K, outre l'horloge. Son fonctionnement se décrit bien au moyen d'une table qui décrit la fonction réalisée en fonction des valeurs de la commande :

J(t-1)	K(t-1)	Fonction	Equation
0	0	Mémoire	Q(t) = Q(t-1)
0	1	Mise à zéro synchrone	Q(t) = '0'
1	0	Mise à un synchrone	Q(t) = '1'
1	1	Changement d'état	$Q(t) = \overline{Q(t-1)}$

Comme pour tout opérateur synchrone, l'état de la bascule ne change pas entre deux transitions actives du signal d'horloge. La fonction « mémoire », dans la table ci-dessus, signifie que la bascule conserve son état précédent même lors d'une transition d'horloge. Dans la construction du diagramme de transitions de la figure suivante, III-28, on a tenu compte de ce que chaque transition peut être obtenue par deux combinaisons différentes des commandes J et K, la transition '0' $\rightarrow$ '1', par exemple, peut être obtenue par mise à 1 explicite (JK = "10") ou par changement d'état (JK = "11") :

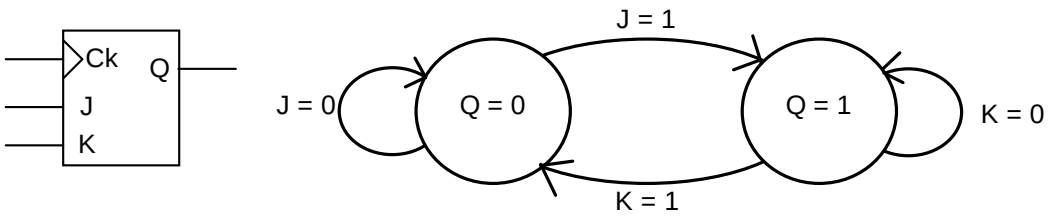


Figure III-28

On déduit aisément l'équation de la bascule J-K de son diagramme de transition :

$$Q(t) = J * \overline{Q(t-1)} + \overline{K} * Q(t-1)$$

Au lieu de raisonner sur l'équation de l'état futur, on peut décrire la bascule J-K par son équation de transition, si on introduit une variable binaire auxilliaire T_{JK} , égale à '1' si une transition doit avoir lieu, '0' autrement, on obtient :

$$T_{JK} = J * \overline{Q} + K * Q$$

Description en VHDL

La première description que nous donnerons est la simple traduction naïve de la table de vérité :

```
entity jk is port (
j,k,clock : in bit;
```

```
q: out bit);
end jk;

architecture fsm of jk is
signal etat : bit;
begin
process begin
wait until clock = '1';
      if (j = '1' and k = '1') then
            etat <= not etat;
      elsif (j = '1' and k = '0') then
            etat <= '1';
      elsif (j = '0' and k = '1') then
            etat <= '0';
      end if;
end process;
q <= etat;
end fsm;
```

Dans la version suivante, construite de la même façon, on a tenu compte des simplifications qui apparaissent dans le diagramme de transitions. Il est bien évident que le compilateur aurait, de toute façon trouvé tout seul ces simplifications.

```
architecture fsm1 of jk is
signal etat : bit;

begin
process begin
wait until clock = '1';
      IF (j = '1' and etat = '0') then
            etat <= '1';
      elsif (k = '1' and etat = '1') then
            etat <= '0';
      end if;
end process;
q <= etat;
end fsm1;
```

Les exemples précédents étaient construits à partir de la commande, ceux qui suivent le sont à partir de l'état interne de la bascule :

```
architecture fsm2 of jk is
signal etat : bit;
begin
q <= etat;
process 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 fsm2;
```

Ou, dans une variante déjà rencontrée :

```
architecture fsm3 of jk is
signal etat : bit;
begin
q <= etat;
process(clock)
begin
if(clock = '1'and clock'event) then
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 fsm3;
```

En conclusion de cette énumération précisons que les équations logiques générées par un compilateur seront les mêmes quelle que soit la forme du programme source, à savoir :

PLD Compiler Software DESIGN EQUATIONS

q.D = q.Q * /k + /q.Q * j

Ce qui est rassurant.

Les applications

Les applications des bascules J-K ne diffèrent guère de celles des autres bascules synchrones. Dans des réalisations « discrètes », qui utilisent un câblage externe et des composants standard, elles conduisent en général à des schémas plus simples que les versions réalisées avec des bascules D. Cela tient à l'existence d'un mot de commande (JK) très souple :

- Chaque transition peut être obtenue de deux façons différentes,
- le maintien dans l'état correspond à l'omission de tout terme correspondant dans les équations de J et K.

Dans les circuits programmables ou les ASICs, le problème se pose de façon un peu différente : une bascule J-K est plus compliquée qu'une bascule D, la complexité globale de la fonction réalisée peut alors être strictement équivalente dans les deux réalisations.

Où l'on apprend à passer de l'une à l'autre

Les caractéristiques communes de toutes les bascules synchrones sont :

1. Deux états possibles,
2. l'éventuel passage d'un état à l'autre a lieu lorsque survient le front actif de l'horloge,
3. les transitions sont régies par un ou des signaux de commande qui doivent être stables avant le front actif du signal d'horloge.

Elles diffèrent par les détails des signaux de commande.

Avec un type de bascule on peut, par adjonction d'une fonction combinatoire générer toutes les autres, les équations nécessaires peuvent être obtenues par identification des équations, ou par examen des diagrammes de transition.

Nous avons donné, à titre d'exemple, le logigramme d'une bascule T réalisé au moyen d'une bascule D ; le lecteur pourra, à titre d'exercice, déduire des diagrammes de transitions des différents types de bascules, les autres schémas de passage possibles :

- Réaliser une bascule J-K avec une bascule D, et réciproquement,
- réaliser une bascule J-K avec une bascule T, et réciproquement,
- réaliser une bascule D avec une bascule T.

Exercices

Opérateurs combinatoires

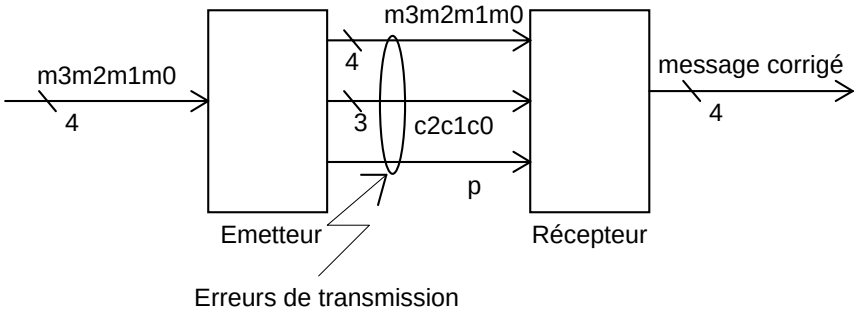
1. Le circuit 74xx86 est un « ou exclusif » en logique positive. Quelle est l'opération réalisée dans une convention logique négative ? Justifiez votre réponse par une table de vérité.

2. Donnez le logigramme de la fonction
   $f = a*b + \bar{a} * c * d + a * \bar{d}$ en n'utilisant que des opérateurs « NAND ».
   En utilisant les lois de De Morgan donner l'expression de $\bar{f}$ sous forme de somme de produits.

Notions de détection et de correction d'erreurs.

Un système de transmission comporte un émetteur et un récepteur, conformément à la figure ci-dessous. Les informations sont regroupées en messages $\mathbf{m_3m_2m_1m_0}$ de quatre bits, il peut s'agir, par exemple, de données codées en *DCB*. Au cours de la transmission des erreurs peuvent se produire. Pour tenter de détecter ces erreurs éventuelles, on rajoute au message des clés de contrôle $\mathbf{c_2c_1c_0}$ et un bit de parité générale **p**. A l'émission ces éléments de contrôle sont construits comme suit:

- L'ensemble $\mathbf{m_3m_2m_1c_2}$ contient toujours un nombre pair de bits à "1".
- L'ensemble $\mathbf{m_3m_2m_0c_1}$ contient toujours un nombre pair de bits à "1".
- L'ensemble $\mathbf{m_3m_1m_0c_0}$ contient toujours un nombre pair de bits à "1".
- L'ensemble $\mathbf{m_3m_2m_1m_0c_2c_1c_0p}$ contient toujours un nombre pair de bits à "1".



On notera qu'une erreur de transmission peut affecter n'importe quel bit, qu'il appartienne au message ou à l'un des éléments de contrôle.

a Quel opérateur permet de générer les clés $\mathbf{c_i}$ à l'émission ?

b Montrer qu'en réalité le calcul de **p** ne nécessite qu'un opérateur à trois opérandes. Préciser la nature de l'opérateur et les opérandes.

c Le récepteur recalcule la parité des ensembles définis ci-dessus. Il construit ainsi quatre variables binaires $\mathbf{e_2}$ $\mathbf{e_1}$ $\mathbf{e_0}$ et **e** qui indiquent, par un 1 logique, qu'il y a une faute de parité sur l'un de ces ensembles ; **e** indique une faute de parité générale. Proposer un schéma pour la génération des $\mathbf{e_i}$ et de **e**.

d En admettant qu'il ne peut pas y avoir plus d'une erreur de transmission, montrer que l'analyse des $\mathbf{e_i}$ permet de savoir lequel des bits $\mathbf{m_3}$ $\mathbf{m_2}$ $\mathbf{m_1}$ ou $\mathbf{m_0}$ est faux : on construira une table de vérité dont les entrées sont les $\mathbf{e_i}$ et les sorties quatre indicateurs $\mathbf{f_3}$ $\mathbf{f_2}$ $\mathbf{f_1}$ et $\mathbf{f_0}$ qui indiquent une erreur sur $\mathbf{m_3}$ $\mathbf{m_2}$ $\mathbf{m_1}$ ou $\mathbf{m_0}$ respectivement.

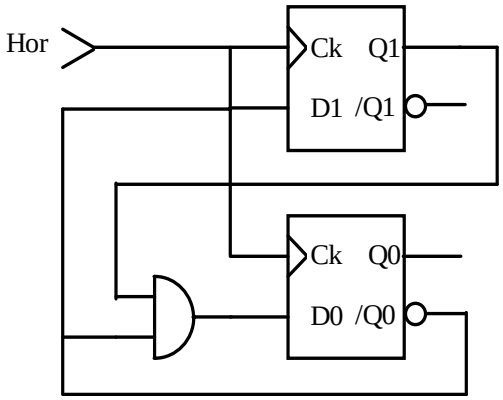
Proposer un schéma de correction de l'éventuelle erreur.

e Que se passe-t-il s'il y a deux erreurs de transmission ? A quoi sert le contrôle de parité générale ?

f La méthode précédente peut sembler extrêmement lourde ; montrer qu'en fait le nombre formé par les $\mathbf{e_i}$ (en base 2) peut être interprété comme le numéro du bit faux, clés comprises. En déduire que le nombre de clés nécessaires pour corriger une erreur dans un message est égal au logarithme en base 2 du nombre de bits de ce message, clés comprises. Application numérique: quelle est la longueur du message utile si on transmet des paquets de 256 bits ?

Du schéma au chronogramme.

On considère le schéma suivant :



– En admettant que les deux bascules sont initialement à 0, établir un chronogramme qui fait apparaître l'horloge et les deux sorties Q1 et Q0.

– L'état initial des bascules a-t-il une importance ?