① p44→47 : Opérateurs fondamentaux : NON, ET, OU (Descriptions comportementales)

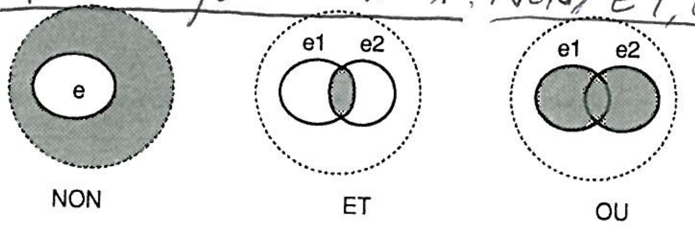

Figure III-5

Description en VHDL

①

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).

NON

```
-- 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 :

ET

```
-- 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 :

© MASSON. La photocopie non autorisée est un délit.

TD Corrigés

OU

```
-- 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

TD6. VHDL Corrigé

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.

NON

```
-- 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 :

ET

```
-- 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;
```

OU

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;
```

ET

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;
```

OU

62


```
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) :

© MASSON. La photocopie non autorisée est un délit.

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

p54 → 58 : OU exclusif : Description structurelle

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.

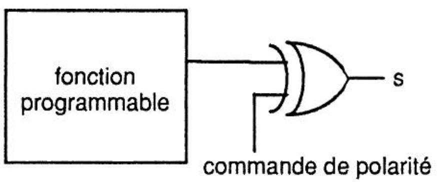

Figure III-13

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 !

OUEX

```
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);
```

© MASSON. La photocopie non autorisée est un délit.


```
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 :

OUEX

```
-- 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' :

OUEX

```
-- 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 :

OUEX

```
-- operateur OU exlusif generalise

ENTITY ouex IS
PORT ( a : IN  BIT_VECTOR(0 TO 3) ;
             s : OUT BIT );
END ouex;
```

64


```
ARCHITECTURE parite of ouex is
BEGIN
process(a)
variable parite : bit := '0';
begin
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] :

OUEX

```
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 := '0';
begin
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
```

MASSON. La photocopie non autorisée est un délit.

[8] Le lecteur est instamment convié à dessiner un schéma logi...

```
       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;
```

OUEX

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*.

65

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.

© MASSON. La photocopie non autorisée est un délit.

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 :

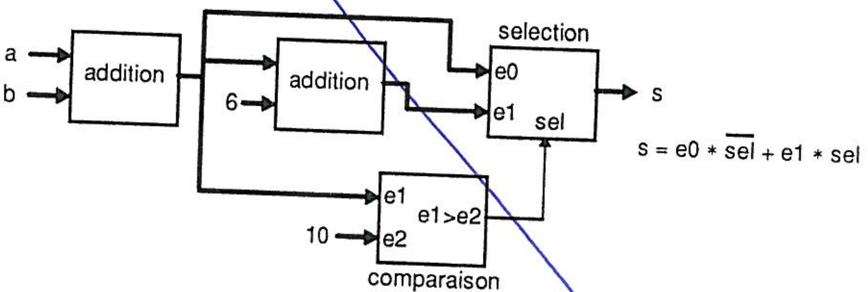

Figure III-14

Symbole et logigramme

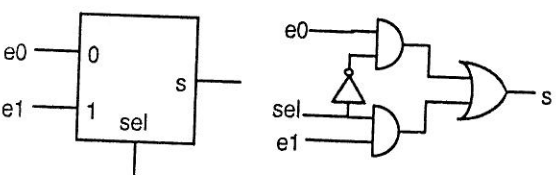

Figure III-15

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).

③ Multiplexeur 2→1 = Description a) comportementale, b) flot de données et c) structurelle

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.

③ MUX 2→1

```
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
```

66


```
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*.

© MASSON. La photocopie non autorisée est un délit.

Son contenu : une architecture

Le fonctionnement interne d'un module, son *corps*, est précisé par une architecture associée à l'entité qui décrit l'aspect extérieur de ce module. Une architecture porte un nom, ce qui autorise la création de plusieurs architectures différentes pour la même déclaration d'entité. Une unité de conception est la réunion d'une entité et d'une architecture.

Syntaxe

L'architecture est divisée en deux parties : une zone déclarative et une zone d'instructions. Ces instructions sont concurrentes, elles s'exécutent en parallèle. Cela signifie que les instructions d'une architecture peuvent être écrites dans un ordre quelconque, le fonctionnement ne dépend pas de cet ordre d'écriture. Ce point est simple à comprendre si on « pense circuits », les différentes parties d'un circuit coexistent et agissent simultanément. Il peut être un peu surprenant pour les programmeurs habitués des langages procéduraux comme C ou PASCAL, qui continuent à « penser » algorithmes séquentiels.

```
architecture_body ::=
architecture architecture_name of entity_name is
architecture_declarative_part
begin
architecture_statement_part
end [ architecture ] [ architecture_name ] ;
```

La zone déclarative permet de définir des types, des objets (signaux, constantes) locaux au bloc considéré. Elle permet également de déclarer des noms d'objets externes (composants d'une librairie, par exemple) utilisés dans le corps de l'architecture.

Exemples

Les trois exemples qui suivent correspondent aux trois exemples de déclarations d'entités donnés précédemment.

Un multiplexeur de quatre voies vers une voie :

```
architecture essai of mux4_1 is
   constant zero  : bit_vector(1 downto 0) := "00" ;
   constant un    : bit_vector(1 downto 0) := "01" ;
   constant deux  : bit_vector(1 downto 0) := "10" ;
   constant trois : bit_vector(1 downto 0) := "11" ;
begin
process (e0,e1,e2,e3,sel)
begin
      case sel is
         when zero  => sort <= e0 ;
         when un    => sort <= e1 ;
         when deux  => sort <= e2 ;
         when trois => sort <= e3 ;
      end case ;
```

67


```
end process ;
end essai ;
```

Notons en passant que <= représente l'opérateur d'affectation d'un signal.

Un multiplexeur de dimension arbitraire :

```
architecture essai of muxN_1 is
begin
sort <= entree(sel) ;
end essai ;
```

Un registre bidirectionnel de dimension arbitraire :

```
architecture essai of registre_N is
   signal temp : std_logic_vector(dimension - 1 downto 0) ;
begin
donnee <= temp when direction = '0' else (others => 'Z') ;7
process
begin
      wait until hor = '1' ;
      if direction = '1' then
         temp <= donnee ;
      end if ;
end process ;
end essai ;
```

On distingue classiquement en VHDL trois styles de descriptions, qui peuvent être utilisés simultanément.

a) *Description comportementale* du MUX 2 → 1

Une description comportementale (*behavioral*) se présente comme des blocs d'algorithmes séquentiels exécutés par des processus indépendants. Elle peut traduire un fonctionnement séquentiel ou combinatoire du circuit modélisé. Nous expliciterons ce point en détail dans la suite.

Un simple multiplexeur deux voies vers une voie peut être décrit par un algorithme qui reproduit son comportement :

```
entity mux2_1 is
   port(e0,e1 : in bit ;
        sel : in bit ;
        sort : out bit ) ;
end mux2_1 ;

architecture comporte of mux2_1 is
begin
process (e0,e1,sel)8
```

7. L'expression (others => 'Z') est un aggregat. La même valeur 'Z' est affectée à tous les éléments du tableau.

8. Les éléments mis entre parenthèse indiquent les signaux dont les changements doivent « réveiller » le processus, et donc provoquer l'évaluation du signal de sortie.

```
begin
   if sel = '0' then
      sort <= e0 ;
   else
      sort <= e1 ;
   end if ;
end process ;
end comporte ;
```

b) Description flot de données du MUX 2→1

Une description flot de données (*data flow*) correspond grosso modo à un *register transfert language*, les signaux passent à travers des couches d'opérateurs logiques qui décrivent les étapes successives qui font passer des entrées d'un module à sa sortie. Le même multiplexeur élémentaire s'écrit, dans ce style :

```
architecture flot of mux2_1 is
      signal sele0, sele1 : bit ;
begin
      sort  <= sele0 or sele1 ;
      sele0 <= e0 and not sel ;
      sele1 <= e1 and sel ;
end flot ;
```

c) Description structurelle du MUX 2→1

Une description structurelle (*structural*) utilise des composants supposés exister dans une librairie de travail, sous forme d'unités de conception. Le programme se contente alors d'*instancier* les composants nécessaires et de décrire leurs interconnexions. Le même multiplexeur utilise deux portes ET, un NON et un OU[9] :

```
architecture struct of mux2_1 is
component et
      port (a, b : in bit ; s : out bit) ;
end component ;
component ou
      port (a, b : in bit ; s : out bit) ;
end component ;
component non
      port (a : in bit ; s : out bit) ;
end component ;
signal nonsel, sele0, sele1 : bit ;
begin
result  : ou  port map(sele0, sele1, sort) ;
```

68

9. Le lecteur est vivement convié à dessiner le schéma classique d'un tel multiplexeur.


```
complem : non port map(sel, nonsel) ;
choix0  : et  port map(nonsel, e0, sele0) ;
choix1  : et  port map(sel, e1, sele1) ;
end struct ;
```

Ce dernier programme suppose que les composants portent les mêmes noms que les unités de conception auxquelles ils se réfèrent, ce n'est en rien une obligation. Des déclarations de configuration permettent de créer des liens, entre les composants instanciés et les couples entité architecture, autres que par homonymie.

Les exemples qui précèdent sont d'une naïveté qui n'utilise pas la puissance du langage, il ne resterait rien de ces programmes si on cherchait à rendre le code source plus compact, c'est une évidence.

Le plus souvent, on utilisera une description structurelle au niveau supérieur d'un projet, et des descriptions des deux autres types, suivant les fonctions décrites et les goûts du programmeur, pour les modules instanciés. Les programmes VHDL générés par les outils de placement routage, à des fins de vérification temporelle du bon fonctionnement d'une application, sont bien sûr essentiellement structurels : ils reproduisent le câblage réellement effectué dans le circuit ou sur une carte. Les fondeurs fournissent des modèles comportementaux des opérateurs élémentaires de leurs circuits qui prennent en compte leurs caractéristiques dynamiques, temps de propagation, entre autres.

II.2.3. Types et classes

VHDL est un langage fortement typé, tout objet[10] manipulé doit avoir un type défini avant la création de l'objet. Indépendamment de son type, un objet appartient à une classe[11]. Schématiquement, on peut dire que le type définit le format des données et l'ensemble des opérations légales sur ces données, alors que la classe définit un comportement dynamique, précise la façon dont évolue (ou n'évolue pas dans le cas d'une constante !) une donnée au cours du temps.

Le langage distingue quatre catégories de types :

- Les types scalaires, c'est-à-dire les types numériques et énumérés, qui n'ont pas de structure interne.
- Les types composés (tableaux et enregistrement) qui possèdent des sous-éléments.
- Les types *access*, qui sont des pointeurs.
- Le type *file*, qui permet de gérer des fichiers séquentiels.

10. Dans le langage un objet est défini comme une grandeur nommée qui contient une valeur d'un type défini. Une *entity*, par exemple, n'est donc pas un objet. Le fait qu'un objet doive être nommé souffre quelques exceptions.

11. En réalité cette indépendance entre classe et type n'est pas vérifiée pour les fichiers : la classe *file* est associée à des types qui lui sont spécifiques. La manipulation des fichiers sera abordée plus loin dans ce chapitre, à propos des outils de modélisation.

④. Bascule D latch

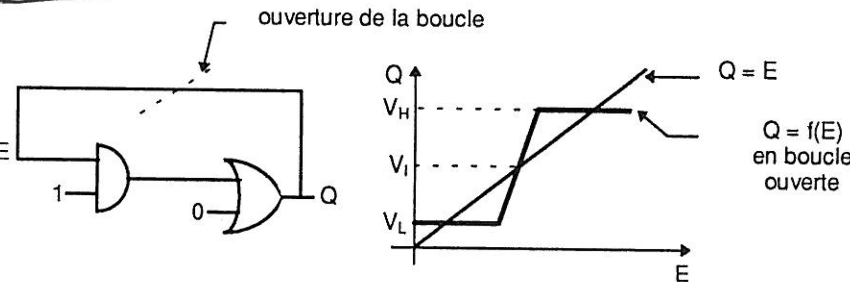

Figure III-17

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;
```

69

[10] Voir à ce sujet au paragraphe II.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.

MASSON. La photocopie non autorisée est un délit.


```
        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;
```

5. Bascule RS

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;
```

70

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].

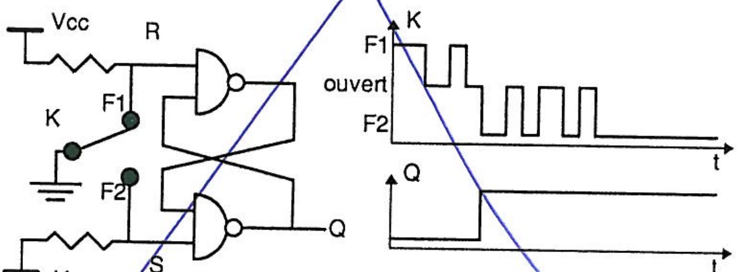

Figure III-20

© MASSON. La photocopie non autorisée est un délit.

[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.

⑥. Bascule D

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 transition

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).

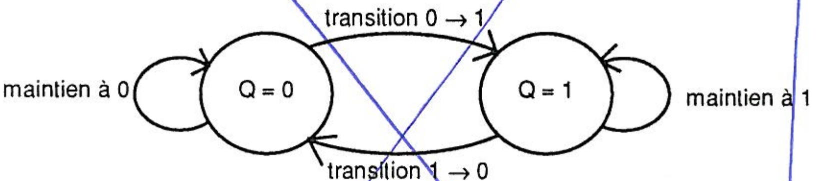

Figure III-23

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.

71

[16] Warp VHDL ...

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 :

$$Q(t) = D(t-1)$$ 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 :

$Q = D$

) MASSON. La photocopie non autorisée est un délit.

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

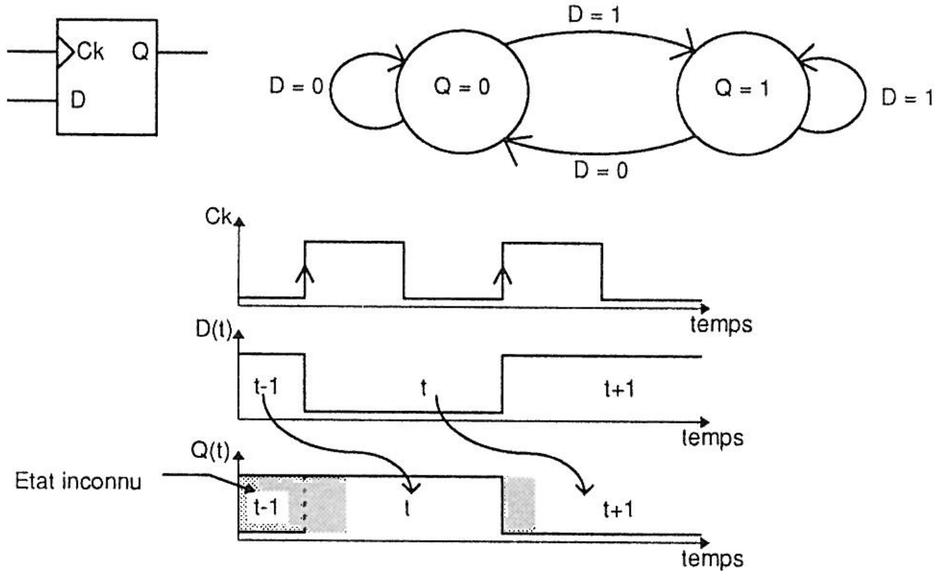

Figure III-24

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.

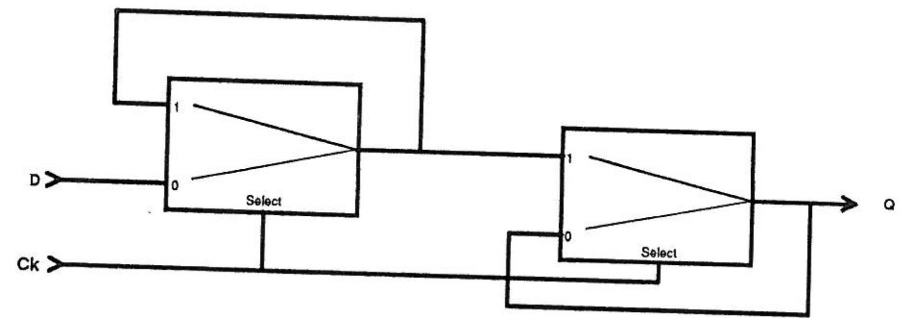

Figure III-25

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;
```

© MASSON. La photocopie non autorisée est un délit.

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_primitivel 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_primitivel;
```

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;
```

73

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
```

MASSON. La photocopie non autorisée est un délit.


```
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 :

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

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

7. Bascule T

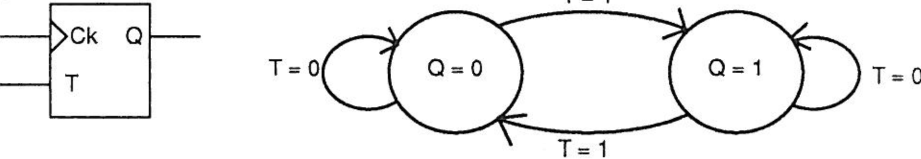

Figure III-26

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 :

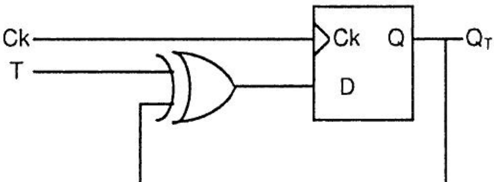

Figure III-27

Description en VHDL

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

74

© MASSON. La photocopie non autorisée est un délit.


```
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 débarassé 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 :

75

Bascule JK

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' → '1', par exemple, peut être obtenue par mise à 1 explicite (JK = "10") ou par changement d'état (JK = "11") :

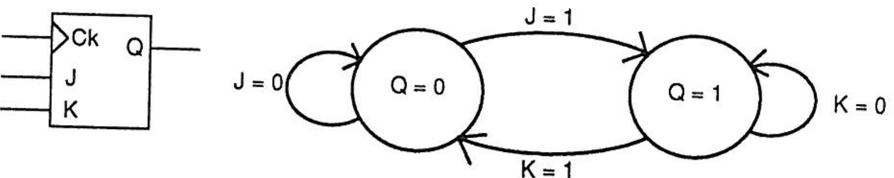

Figure III-28

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 (
```

© MASSON. La photocopie non autorisée est un délit.

```
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';
```

76


```
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.

© MASSON. La photocopie non autorisée est un délit.