Circuits logiques combinatoires et séquentiels

Guy Bégin

26 octobre 2023

Langages descriptifs et de modélisation

Objectifs

- Expliquer la différence entre un langage descriptif de matériel et un langage de programmation
- Expliquer les mécanismes d'assignation de signaux et la notion de concurrence
- Faire la distinction entre une description d'entité et une architecture
- Faire la distinction entre un modèle structural et un modèle comportemental
- Faire la distinction entre un signal et une variable

Objectifs ... 2

- Se familiariser avec les principaux types utilisés en descriptions de circuits
- Préparer une description (modèle) de circuit en langage VHDL
- Préparer un banc d'essai permettant de tester un modèle de circuit
- Simuler un modèle VHDL au moyen d'un banc d'essai et interpréter les résultats
- Élaborer une description complète en VHDL pour un système simple

Modélisation et simulation

- Lorsque vient le temps de concevoir, simuler, tester et élaborer des systèmes numériques complexes, les approches manuelles que nous avons employées jusqu'ici ne suffisent plus
- Il faut alors faire appel à des familles d'outils plus puissants de conception assistée par ordinateur
- Avec ces outils, il est possible de concevoir et de spécifier précisément le design voulu, en simuler le fonctionnement, le valider par une batterie de tests, afin de s'assurer que le tout correspond aux besoins de l'application avant même de solliciter une seule porte logique physique

Modélisation et simulation ... 2

- Un élément clé de cette démarche est la possibilité de décrire précisément le ou les circuits qui seront implémentés au moyen d'un langage approprié
- Un tel langage descriptif de matériel (en anglais *Hardware Description Language* (HDL)), qui s'apparente à un langage de programmation, permet de décrire de façon textuelle les différents éléments de notre circuit, leurs interconnexions et interactions
- Alors qu'un langage de programmation spécifie essentiellement des procédures et les données qui y sont associées, un HDL est un langage de modélisation qui décrit des structures matérielles et le comportement de systèmes logiques

Modélisation et simulation ... 3

- Un HDL peut spécifier des diagrammes logiques, des expressions logiques, voire des tableaux de vérité
- Il permet aussi de décrire le comportement du système à différents niveaux d'abstraction et les relations hiérarchiques entre les différents sous-systèmes qui le composent
- On peut voir un modèle HDL comme la description des relations entre les entrées et les sorties du système
- Entrées et sorties sont modélisées comme des signaux

Langage VHDL

Parmi les nombreux HDL en usage, quelques-uns ont été standardisés : les plus répandus sont *Verilog* et VHDL

- C'est ce dernier langage que nous allons utiliser
- Le V dans l'acronyme VHDL (*VHSIC Hardware Description Language*) provient d'un autre acronyme, VHSIC pour *Very High Speed Integrated Circuits* (circuits intégrés à très haute vitesse)
- On comprend que le langage a été créé dans l'optique de concevoir des circuits intégrés rapides et complexes

Langage VHDL ... 2

- Un design en VHDL est un ensemble d'entités de conception
- Une entité, celle du plus haut niveau, invoque les autres entités comme composants
- Le design dans son ensemble est structuré de façon hiérarchique

Entité

Une entité définit le nom d'un modèle et spécifie ses interfaces, c'est-à-dire les entrées et les sorties qui permettent au modèle d'interagir avec son environnement

- Le nom, la direction et le type de chaque signal d'interface sont déclarés dans le port de l'entité
- La fonction du modèle n'est aucunement précisée
- Il s'agit uniquement de décrire la «coquille» d'une boîte noire
- La déclaration d'entité spécifie le nom de l'entité et la liste des ports d'entrée et de sortie
- La forme générale est comme ci-dessous (les éléments entre crochets sont optionnels)

Déclaration d'entité

```
entity Nom_Entite is
[generic generic_declarations);]

port (noms_signaux: mode type;

noms_signaux: mode type;
--
--
--
noms_signaux: mode type);

end [Nom_Entite];
```

Déclaration d'entité ... 2

- La déclaration commence avec le mot réservé `entity`, suivi du nom et du mot réservé `is`
- Viennent ensuite les déclarations de ports avec le mot réservé `port`
- La déclaration se termine avec le mot réservé `end` et optionnellement, le nom de l'entité
- `Nom_Entite` est un nom arbitraire choisi par le concepteur ou la conceptrice
- `noms_signaux` donne une liste d'un ou de plusieurs identifiants séparés par des virgules qui définissent les signaux externes d'interface

Déclaration d'entité ... 3

mode : est un des mots réservés suivants, qui définissent la direction des signaux :

- `in` : le signal est une entrée
- `out` : le signal est une sortie de l'entité, qui peut être lue par les autres entités qui y sont raccordées
- `buffer` : le signal est une sortie qui peut être lue de l'intérieur de l'architecture de l'entité
- `inout` : le signal peut être une entrée ou une sortie

Déclaration d'entité ... 4

- `type` : est un type de signal prédéfini ou défini par le concepteur ou la conceptrice, par exemple, `bit`, `bit_vector`, `Boolean`, `character`, `std_logic`, ou `std_ulogic`
- `bit` : une valeur binaire 0 ou 1
- `bit_vector` : un vecteur de bits
- `std_logic`, `std_ulogic`, `std_logic_vector`, `std_ulogic_vector` : des valeurs binaires plus nuancées (voir librairies)
- `boolean` : deux valeurs possibles : TRUE ou FALSE
- `integer` : des valeurs entières
- `real` : des valeurs réelles
- `character` : des caractères
- `time` : des valeurs de temps

Déclarations génériques

- `generic` : les déclarations génériques sont optionnelles et spécifient des constantes locales utilisées pour préciser par exemple des valeurs de temps ou des tailles de vecteur
- Un générique peut avoir une valeur de défaut
- La syntaxe est comme ci-après

Déclarations génériques ... 2

```
generic (

nom_constante: type [:=valeur];

nom_constante: type [:=valeur];
--
--
--
nom_constante: type [:=valeur] );
```

Déclaration d'entité

Le listage qui suit montre un exemple simple de déclaration d'entité

```
entity ALU is
port (arg1, arg2: in bit_vector;
add_or_sub: in bit;
result: out bit_vector);
end ALU;
```

Architecture

- Une `architecture` est une réalisation (ou implémentation) de l'intérieur de la boîte noire
- Jumelée à une entité, elle décrit comment les sorties de l'entité sont obtenues à partir de ses entrées
- Il est possible d'associer de multiples architectures à une même entité

Une architecture peut contenir :

- des déclarations de données
- des affectations concurrentes de signaux
- des blocs processus
- des instanciations de composants

Structure typique d'une architecture

```
architecture nom_d_architecture of NOM_ENTITE is
-- Déclarations de types de données
-- Déclarations de composants
-- Déclarations de signaux
-- Déclarations de constantes
-- Déclarations de fonctions
-- Déclarations de procédures
begin
-- Énoncés concurrents ou séquentiels
end nom_d_architecture;
```

Les énoncés qui peuvent se trouver dans le corps de l'architecture (entre le begin et le end peuvent être des instanciations de composants, des assignations de signaux ou des énoncés de processus

Signaux et assignation

- Un signal représente en quelque sorte un «fil»
- Une assignation comme `A <= NOT(B);` signifie que A et B sont des signaux reliés dont l'un est l'inverse logique de l'autre
- Ainsi, `A <= B;` signifie que les deux signaux A et B auront la même valeur logique

Notes sur la syntaxe

- Des expressions aussi complexes que désiré peuvent être écrites, en utilisant des parenthèses pour spécifier les priorités d'évaluation des opérations (si elles doivent être différentes de la priorité implicite de VHDL)
- Les commentaires sont possibles : tout ce qui suit deux tirets (- -) est ignoré
- Toutes les assignations doivent se terminer avec un point-virgule (;)

Voici des exemples d'expressions

```
a <= ((b and c and f) or (t nor r)) nand p;
a(3 downto 0) <=
b(4 downto 1) when (p and q)
else (p & q & q & p);
```

Autres exemples d'assignations

```
A <= B OR C;
A <= B AND C;
A <= B NOR C;
A <= B NAND C;
A <= B XOR C;
A <= NOT B; --ceci est un commentaire
A <= (B AND C) OR (D AND E); --(aucune priorité
                             -- préétablie de AND/OR)
```

- Les parenthèses permettent de préciser l'ordre des opérations
- Nous nous contenterons de ces opérations pour le moment

Priorité d'opérations et associativité

```
A <= B AND C AND D; -- ceci fonctionne

A <= B NAND C NAND D; -- pas ceci: NAND n'est pas associatif

A <= NOT(B AND C AND D); -- ceci fonctionne
```

Assignations avec vecteurs de signaux

- Voici encore d'autres exemples d'assignations
- Le dernier de ces exemples fait appel à des vecteurs de signaux
- Nous verrons plus loin comment définir ces regroupements de signaux
- De façon générale, VHDL est insensible aux MAJUSCULES ou minuscules et ignore les espaces supplémentaires et sauts de lignes
- On doit déclarer le type de tous les objets : signaux, constantes ou variables

Assignations avec vecteurs de signaux ... 2

```
Sum <= A XOR B XOR Cin;

Cout <= (A AND B) OR (A AND Cin);

Cout <= (A AND B) OR (A AND Cin) OR (B AND Cin);

a <= b when c else a; -- sélection (multiplexage)

a(6 downto 1) <= c & d(3 downto 0) & e; -- concaténation
                                        -- bit-à-bit ("&")
```

Concurrence

- Dans un corps d'architecture, les assignations sont concurrentes
- Par exemple, dans ce qui suit, les deux énoncés sont évalués en parallèle

```
q <= r nor qb; -- énoncé 1
qb <= s nor q; -- énoncé 2
```

- Les valeurs pour q et qb sont continuellement mises à jour : dès qu'un des signaux à droite de l'assignation change (on dit qu'un évènement se produit sur le signal), l'énoncé est évalué de nouveau

Énoncés concurrents

- Ce n'est pas comme en programmation où les énoncés sont évalués l'un après l'autre, une seule fois
- Tous les énoncés concurrents sont continuellement évalués

Énoncés concurrents équivalents

- L'effet de ces énoncés sera exactement le même si on les place dans un autre ordre dans la description, comme ci-dessous

```
qb <= s nor q; -- énoncé 1
q <= r nor qb; -- énoncé 2
```

- Le langage décrit à la base un circuit et non pas une procédure : toutes les portes sont toujours alimentées et les fils sont toujours connectés
- L'ordre dans lequel on donne la description n'a aucune importance
- Nous verrons plus loin qu'il est aussi possible de définir des blocs dans lesquels l'exécution est séquentielle comme en programmation

Vecteurs de bits

- Il est possible de grouper des signaux pour en faire des vecteurs, qui sont des groupes ordonnés de bits : un mot, un bus, etc.
- De cette façon, les spécifications sont plus compactes et faciles à interpréter
- La convention la plus naturelle ordonne les indices de bits A(msb) $\longleftrightarrow$ A(lsb), mais l'ordre inverse est également possible
- Il vaut mieux établir une convention et s'y tenir pour éviter les erreurs d'interprétation

Vecteur de bits

```
A(VALEUR_HAUTE downto VALEUR_BASSE)

A(15 downto 0) -- A comporte 16 bits

A(7 downto 3) -- 5 bits du milieu de A

A(0 to 7) -- A comporte 8 bits, énumérés de façon croissante
```

Concaténation

L'opérateur de concaténation & permet de combiner des groupes de bits

```
A(15 downto 0) <= B(7 downto 0) &
                  C(7 downto 0);
```

Description compacte et lisibilité

- Considérons par exemple la partie du calcul de somme de la description d'un additionneur 8 bits ci-dessous
- Avec des déclarations adéquates, on peut écrire un calcul de somme plus compact

Calcul de somme initial

```
Sum(7 downto 0) <= A(7 downto 0)
xor B(7 downto 0)
xor C(7 downto 0);

C(7 downto 0) <= (A(7 downto 0) and
B(7 downto 0)
)
or (A(7 downto 0) and
(C(6 downto 0) & Cin)
)
or (B(7 downto 0) and
(C(6 downto 0) & Cin)
);

Cout <= C(7);
```

Calcul de somme compact

```
Sum <= A xor B xor (C(6 downto 0) & Cin);

C <= (A and B) or
(A and (C(6 downto 0) & Cin) ) or
(B and (C(6 downto 0) & Cin) );

Cout <= C(7);
```

Modèle complet

- Considérons la bascule JK maître-esclave de la figure 1 ci-dessous, construite au moyen de portes simples
- Un modèle VHDL complet pour cette bascule JK maître-esclave est donné ensuite

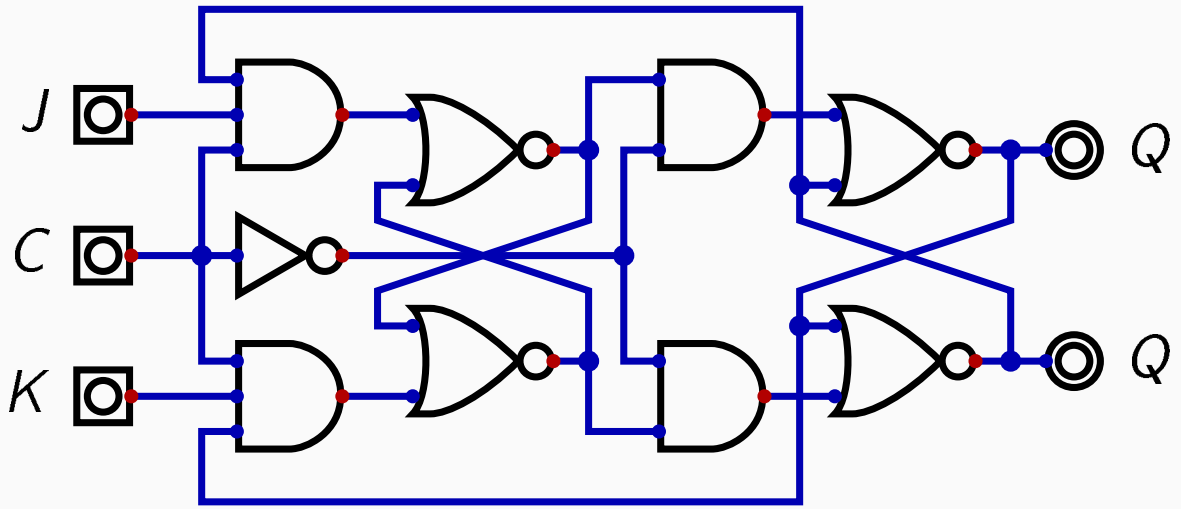


Figure 1 – Bascule JK maître-esclave à modéliser

Bascule JK maître-esclave

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
-- Bascule JK maître-esclave.
-- La bascule maître mémorise sur une horloge haute,
-- la bascule esclave mémorise sur une horloge basse.
-- Les deux rétroactions évitent les conditions
-- interdites aux entrées
entity mainJK is
  port (
    J: in std_logic; -- entrée J (set) de la bascule
    C: in std_logic; -- entrée d'horloge
    K: in std_logic; -- entrée K (reset) de la bascule
    Q: out std_logic; -- bit stocké
    notQ: out std_logic -- inverse du bit stocké
    );
end mainJK;
```

Bascule JK maître-esclave ... 2

```
18  architecture Complet of mainJK is
19    signal notQ_temp: std_logic;
20    signal Q_temp: std_logic;
21    signal s0: std_logic;
22    signal s1: std_logic;
23    signal s2: std_logic;
24  begin
25    s0 <= NOT C;
26    s2 <= NOT ((notQ_temp AND J AND C) OR s1);
27    s1 <= NOT (s2 OR (C AND K AND Q_temp));
28    Q_temp <= NOT ((s2 AND s0) OR notQ_temp);
29    notQ_temp <= NOT (Q_temp OR (s0 AND s1));
30    Q <= Q_temp;
31    notQ <= notQ_temp;
32  end Complet;
```

Modèle comportemental

- Un modèle comportemental d'une entité est un ensemble d'énoncés qui sont exécutés séquentiellement
- Ces énoncés peuvent se trouver dans des blocs `process`, `function` ou `procedure`
- Un tel bloc est concurrent avec les autres énoncés du modèle général
- Il est possible d'utiliser des variables à l'intérieur des processus

Modèle comportemental ... 2

- Un énoncé d'assignation spécifique aux variables permet d'assigner (sans délai) une valeur à une variable qui a été préalablement déclarée
- On se rapproche alors de la programmation traditionnelle
- Les variables sont locales aux processus
- Dans un bloc processus, il est possible d'avoir des boucles, des branchements conditionnels, etc.
- Dans le corps de l'entité, un bloc processus apparaît comme une grosse porte logique arbitrairement définie lors de la conception du bloc

Modèle flux de données

- Dans un modèle flux de données, c'est le mouvement des données qui est exprimé par un ensemble d'énoncés concurrents
- On peut faire appel à des opérateurs logiques AND, OR, NOT, etc., pour décrire les relations entre les signaux

Modèle structural

- Un modèle structural décrit des ensembles de composants interconnectés
- Un énoncé d'instanciation d'un composant (qui revient à dire quelque chose comme «utiliser le composant X ici» est un énoncé concurrent qui indique de créer une instance de la «chose» spécifiée
- Une telle description structurale se contente de préciser quels composants seront reliés à quels autres composants, sans référence au comportement desdits composants

Modèle structural ... 2

- Les énoncés qui suivent le begin spécifient l'instanciation de composants et les interconnexions
- Un énoncé d'instanciation de composant crée un nouveau niveau hiérarchique
- Chaque ligne commence avec un `nom d'instance`, suivi d'un deux-points (' :'), d'un nom de composant et du mot réservé `port map`
- Ce `port map` spécifie les interconnexions du composant
- Rappelons que l'ordre de ces énoncés est sans conséquences

Bloc processus

- Un énoncé `process` permet de définir un processus
- Le format est le suivant

Bloc processus

```
[etiquette_processus:] process [ (liste_sensibilité) ] [is]
[déclarations_processus]
begin
--   liste d'énoncés séquentiels tels que:
-- assignations de signaux
-- assignations de variables
-- énoncés /case/
-- énoncés /exit/
-- énoncés /if/
-- énoncés /loop/
-- énoncés /next/
-- énoncés /null/
-- appels de procedure
-- énoncés /wait/
end process [etiquette_processus];
```

Bascule D, front montant, mise à zéro asynchrone

Voici un exemple d'un modèle avec bloc processus d'une bascule D déclenchée sur un front montant avec mise à zéro asynchrone

Bascule D, front montant, mise à zéro asynchrone ... 2

```
library ieee;
use ieee.std_logic_1164.all;

entity DFF_CLEAR is
   port (CLK, CLEAR, D : in std_logic;
      Q : out std_logic);
end DFF_CLEAR;

architecture COMPORT_DFF of DFF_CLEAR is
begin
DFF_PROCESS: process (CLK, CLEAR)
   begin
      if (CLEAR = '1') then
         Q <= '0';
      elsif (CLK'event and CLK = '1') then
         Q <= D;
      end if;
   end process;
end COMPORT_DFF;
```

Bascule D, front montant, mise à zéro asynchrone ... 3

- Le processus, déclaré à l'intérieur d'une architecture, est un énoncé concurrent
- Mais tout ce qui se déroule à l'intérieur du processus est exécuté de façon séquentielle
- Comme tout énoncé concurrent, le processus lit et écrit des signaux sur ses ports d'interface
- Dans l'exemple précédent, la sortie Q reçoit des valeurs par assignation au sein du processus
- L'expression `CLK'*event* *and* CLK = '1'` teste une condition de front montant sur l'interface d'entrée CLK
- Il ne peut pas y avoir de déclarations de signaux dans un processus ; seules des variables ou des constantes peuvent être déclarées

Bascule D, front montant, mise à zéro asynchrone ... 4

- Une liste de sensibilité est un ensemble de signaux qui déclenchent l'exécution du processus
- N'importe quel changement sur un des signaux de la liste provoque l'exécution immédiate du processus
- S'il n'y a pas de liste de sensibilité, il faudra inclure un énoncé `wait` pour s'assurer que le processus se termine
- Il n'est pas permis d'avoir à la fois une liste de sensibilité et un énoncé `wait` pour un même processus

Bascule D, front montant, mise à zéro asynchrone . . . 5

- Les variables et constantes utilisées dans le processus sont définies dans la portion `déclarations_processus`
- Les énoncés entre `begin` et `end` sont exécutés séquentiellement
- Les assignations de variables, dénotées :=, sont exécutées immédiatement
- Un énoncé concurrent est comme un processus d'une seule ligne, dont la liste de sensibilité est constituée de tous les signaux qui sont à droite de l'assignation
- Il est possible de définir un processus dont le corps est une description combinatoire
- Par exemple, le processus suivant permet de modéliser une porte OU entre les entrées a et b

Processus avec porte OU combinatoire

```
proc1: process
  begin
    wait on a, b;
    s <= a or b;
  end process proc1;
```

La sensibilité d'un tel processus (ici obtenue au moyen de l'énoncé `wait on a, b;`) doit comporter tous les signaux utilisés pour que l'exécution se fasse dès qu'une des entrées change de valeur

Processus avec porte OU combinatoire ... 2

- Les assignations de signaux dans un processus ne prennent effet qu'une fois que le processus est suspendu
- Cela veut notamment dire que c'est seulement la dernière assignation à un signal donné qui sera effectivement exécutée
- Si un processus effectue une lecture d'un signal qui se verra aussi assigner une valeur par le processus, la lecture prendra en compte la valeur précédente du signal avant qu'il soit affecté par l'assignation
- Il est donc possible de créer de la rétroaction au sein d'un même processus

Modélisation du délai

Deux formes de délai peuvent être modélisés en VHDL : le délai inertiel et le délai de transport

Délai inertiel

- Le délai inertiel est la forme de délai par défaut
- Le mot réservé `after` suppose par défaut un délai inertiel
- Avec du délai inertiel, deux changements consécutifs du signal d'entrée qui sont plus rapprochés temporellement que la valeur de délai ne seront pas reflétés sur le signal de sortie
- On modélise alors une inertie du circuit, qui est trop lent pour réagir lorsque les changements d'entrée sont trop rapides pour lui

Délai inertiel . . . 2

- Par exemple, avec une assignation comme la suivante :

```
b <= a after 30 ns;
```

- Si le signal a passe de '0' à '1' à 10 ns et de '1' à '0' à 20 ns, la sortie ne changera pas du tout et restera tout le temps à '0'

Délai de transport

- Le délai de transport applique un retard dans le signal de sortie
- Par exemple, avec la spécification suivante :

```
b <= transport a after 20 ns;
```

- Si le signal a passe de '0' à '1' à 10 ns et de '1' à '0' à 20 ns, la sortie passera de '0' à '1' à 30 ns et de '1' à '0' à 40 ns, reproduisant en sortie la même forme d'onde qu'en entrée, mais retardée de 20 ns

Librairies

- Des librairies de types peuvent être définies pour préciser des types d'objets qui pourront ensuite être utilisés de façon standard
- Par exemple, la librairie IEEE `std_logic_1164` définit le type logique `std_logic` qui apporte plus de nuances que le simple type binaire, qui lui ne comporte que deux valeurs numériques '0' et '1'

Librairies : valeurs pour `std_logic`

Le tableau suivant donne la liste des valeurs possibles avec `std_logic`

Symbole	Interprétation
'1'	1 Logique
'0'	0 Logique
'Z'	Haute impédance
'W'	Signal faible, indéterminé entre 0 ou 1
'L'	0 faible, *pulldown*
'H'	1 faible, *pullup*
'-'	*Don't care*
'U'	Non initialisé
'X'	Inconnu, conflit entre sources multiples

Librairies : valeurs pour `std_logic` ... 3

- Comme on peut le voir, les signaux pourront ainsi assumer des valeurs comme `Z` (haute impédance pour les signaux trois-états), `X` (pour valeur inconnue), - pour valeur facultative, etc.
- Il y a même des nuances pour la solidité des valeurs logiques

Déclaration de librairies

- Une clause use permet de spécifier les librairies à utiliser au début du fichier de spécification
- Par exemple, le fichier VHDL pourrait commencer avec les déclarations du listage suivant :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

- Nous avons déjà vu un exemple de l'utilisation de librairies dans le modèle de la bascule JK maître-esclave

Encapsulation

- Il est possible de grouper des énoncés et de les réutiliser au besoin, dans d'autres modèles
- Ceci nous permet de «cacher» la description dans une boîte
- Dans ce cas-ci, il s'agit d'un loquet SR construit avec des portes NOR

Encapsulation

```
entity rs is
port(r, s: in bit; q, qb: out bit);
end rs;

architecture norlogic of rs is
begin
q <= r nor qb;
qb <= s nor q;
end nor_logic;
```

- L'entité définie est un composant qui peut être utilisée dans d'autres circuits, à l'intérieur d'une hiérarchie de design
- Voici comment on peut utiliser le composant loquet pour modéliser un loquet D

Utilisation d'un composant

```
1  entity Dlatch is
2  port (clk,d: in bit; q,qb: out bit);
3  end Dlatch;
4
5  architecture test of Dlatch is
6  --
7  signal db, cr, cs: bit;
8  --
9  component rs_ff
10 port (r, s: in bit;
11 q, qb: out bit);
12 end component;
13 --
14 label rs0;
15 --
16 for rs0: rs_ff
17 use entity rs(nor_logic);
```

Utilisation d'un composant ... 2

```
18  begin
19  db <= not d;
20  cr <= db and clk;
21  cs <= d and clk;
22  rs0: rs_ff port map(cr, cs, q, qb);
23  end test;
```

Bascule maître-esclave structurale

Voici un exemple de bascule maître-esclave conçue de façon structurale à partir du composant loquet

Bascule maître-esclave structurale ... 2

```
1  entity MS is
2  port (clear,reset,set,clock: in bit;
3  q, qbar: out bit);
4  end MS;
5
6  architecture ms_struct of MS is
7  signal clrbar, r0, s0, q0, qbar0,
8  r1, s1, cbar: bit;
9  label rs0, rs1;
10 component rs_ff
11 port (r,s: in bit; q,qbar: out bit);
12 end component;
13 for rs0, rs1: rs_ff
14 use entity RS(nor_logic);
```

Bascule maître-esclave structurale ... 3

```
16  begin
17  clrbar <= not clear;
18  cbar <= not clock;
19  r0 <= (reset and clock) or clear;
20  s0 <= (set and clock) and clrbar;
21  rs0: rs_ff port map(r0,s0,q0,qbar0);
22  r1 <= (qbar0 and cbar) or clear;
23  s1 <= (q0 and cbar) and clrbar;
24  rs1: rs_ff port map(r1,s1,q,qbar);
25  end ms_struct;
```

Compteur basé sur la bascule

Voici un exemple de compteur élaboré à partir de cette bascule maître-esclave

Compteur basé sur la bascule ... 2

```
1  entity CNTER is
2  port (clock, clear: in bit;
3  a: out bit_vector(3 downto 0));
4  end CNTER;
5
6  architecture count of CNTER is
7  signal b: bit_vector(3 downto 0);
8  component ms_ff
9  port (clr, r, s, c: in bit;
10 q, qbar: out bit);
11 end component;
12 label ms0, ms1, ms2, ms3;
13 for ms0, ms1, ms2, ms3: ms_ff
14 use entity MS(ms_struct);
```

Compteur basé sur la bascule ... 3

```
15  begin
16  ms0: ms_ff port map(
17  clear,a(0),b(0),clock,a(0),b(0));
18  ms1: ms_ff port map(
19  clear,a(1),b(1),a(0),a(1),b(1));
20  ms2: ms_ff port map(
21  clear,a(2),b(2),a(1),a(2),b(2));
22  ms3: ms_ff port map(
23  clear,a(3),b(3),a(2),a(3),b(3));
24  end count;
```

Description de design en VHDL

- La description complète du système à simuler est placée dans un fichier avec extension .vhdl qui est ensuite compilé et simulé
- On appelle cette description «fichier de design» ou «design» tout court
- Un design en VHDL est un ensemble d'entités de conception
- L'entité de plus haut niveau invoque les autres entités comme composants
- Le design dans son ensemble est appelé «hiérarchie de design»

Multiplicateur huit bits

- Voici l'exemple d'un multiplicateur huit bits construit à partir de plusieurs autres composants
- Les listages qui suivent donnent le détail de la modélisation

Multiplicateur 8 bits : entité `multiply`

```
entity multiply is
port (load, clock: in bit;
input1, input2: in
bit_vector(7 downto 0);
product: out
bit_vector(15 downto 0);
output_valid: out bit);
end multiply;
```

Multiplicateur 8 bits : entité adder

```
entity adder is
port(a: in bit_vector(7 downto 0);
b: in bit_vector(7 downto 0);
cin: in bit;
sum: out bit_vector(7 downto 0)
cout: out bit) ;
end adder;
```

Multiplicateur 8 bits : entité D_FF

```
entity D_FF is
port(din: in bit_vector(7 downto 0)
dout: out bit_vector(7 downto 0)
enable: in bit);
end D_FF;
```

Multiplicateur 8 bits : composant adder

```
architecture logic of adder is
signal cw, cx: bit_vector(7 downto 0);
--
begin
cw(0) <= cin;
cw(7 downto 1) <= cx(6 downto 1);
cx <= (a and b) or (a and cw) or (b and cw);
sum <= a xor b xor cw;
cout <= cx(7);
end logic;
```

Multiplicateur 8 bits : composant D_FF

```
architecture edge of D_FF is
-- une bascule D de 8 bits de large
signal x, y, z, w, qb, e :
bit_vector(7 downto 0);
begin
e <= "11111111" when enable else 0;
x <= din nand y;
y <= e nand (not w);
z <= e nand w;
w <= z nand x;
dout <= z nand qb;
qb <= y nand dout;
end edge;
```

Multiplicateur 8 bits : déclarations

```
architecture mult of multiply is
signal mux1, mux2, mux3, mux4:
bit_vector(7 downto 0);
signal control, adder_out:
bit_vector(7 downto 0);
signal accum:
bit_vector(15 downto 0);
signal carry_out: bit;
label l1, l2, l3, l4;

component add
port (arg1, arg2: in bitvec;
c_in: in bit;
result: out bitvec;
c_out: out bit);
end component;
for l4: add use entity adder(logic);
```

Multiplicateur 8 bits : déclarations ... 2

```
component latch
port (xin: in bitvec;
xout: out bitvec; enable: in bit);
end component;
for l1, l2, l3: latch use
entity D_FF(edge);
```

Multiplicateur 8 bits : descriptions

```
1  begin mux1(7 downto 0) <= 255 when load else "0" & contro1(7 downto
2  1); l1: latch port map ( mux1, control, clock);
3
4  mux2 <= 0 when load else
5  carry_out & adder_out(7 downto 1);
6  l2: latch port map (
7  mux2, accum(15 downto 8), clock);
8
9  mux3 <= input2 when load else
10 adder_out(0) & accum(7 downto 1);
11 l3: latch port map (
12 mux3, accum(7 downto 0), clock);
13
14 mux4 <= input1 when accum(0) else 0;
15 l4: add port map (
16 mux4, accum(15 downto 8), "0",
17 adder_out, carry_out);
```

Multiplicateur 8 bits : descriptions ... 2

```
18  output_valid <= not(control(0));
19  product <= accum when output_valid
20  else 0;
21  end mult;
```

Banc d'essai

- Pour simuler un circuit avec un HDL, on doit lui appliquer des signaux aux entrées afin que le simulateur puisse générer les sorties correspondantes
- Une description qui vise à générer ces signaux d'entrée est appelée un banc d'essai

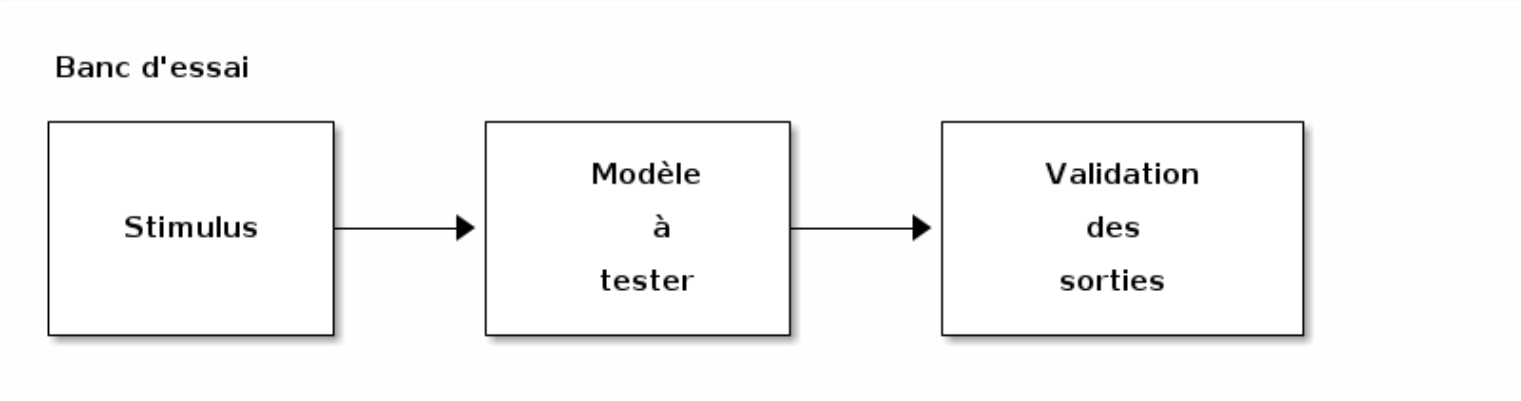


Figure 2 – Banc d'essai

Banc d'essai ... 2

- Le bloc stimulus génère les signaux d'excitation qui sont appliqués aux entrées du modèle à tester
- Un bloc se charge de valider les sorties observées, ce qui peut être fait automatiquement

Entité banc d'essai

On commence par définir une entité de niveau supérieur pour le banc d'essai

```
entity test_bench is

end entity test_bench;
```

- L'entité ne comporte pas de ports, puisque le banc d'essai ne comporte pas d'entrées ou de sorties externes

Instanciation

- Il faut ensuite instancier le modèle à tester
- L'instanciation peut se faire comme composant ou directement
- À moins de vouloir définir un `package` pour le modèle à tester, le composant doit être défini avant le code principal
- Voici un exemple de déclaration d'instanciation par composant
- Les noms de composant et de ports doivent correspondre à ceux du modèle à tester

Banc d'essai : instanciation par composant

```
component and_gate is
port (
   a : in std_logic;
   b : in std_logic;
   and_out : out std_logic
);
end component and_gate;
```

Composant lié

Le composant est ensuite relié au modèle de test

```
and_gate_instance: component and_gate
  port map (
    a       => signal_a,
    b       => signal_b,
    and_out => signal_and_out
  );
```

Composant lié ... 2

- Chaque instanciation doit avoir son propre nom
- Ici, c'est `and_gate_instance`
- Les noms de signaux à gauche sont les noms des ports du composant
- Les noms à droite sont les signaux qui sont reliés aux ports
- Ces signaux doivent être déclarés avant utilisation et avoir le bon type

Instanciation directe

Voici un exemple d'instanciation directe :

```
and_gate_instance: entity work.and_gate(ltr)
  port map (
    a       => signal_a,
    b       => signal_b,
    and_out => signal_and_out
  );
```

- Dans ce cas, il faut également préciser la librairie et l'architecture pour le modèle à tester
- Ici, la librairie est «work» et l'architecture est «ltr»

Écoulement du temps

- Pour tester le modèle, on doit typiquement produire des signaux qui varient en fonction du temps
- Pour ce faire, il est possible d'utiliser les énoncés `after` et `wait`

Type VHDL pour le temps

Il existe un type prédéfini VHDL pour le temps, qui utilise les unités suivantes. La plus petite unité de temps correspond à une femtoseconde (fs = 10^{-15} seconde)

Unité	Valeur
fs	
ps	1000 fs
ns	1000 ps
us	1000 ns
ms	1000 us
sec	1000 ms
min	60 sec
hr	60 min

Énoncés liés au temps

Voici des exemples d'énoncés liés au temps :

```
time_ex <= 100 ps; -- 100 picoseconds
time_ex <= 1.2 ns; -- 1200 picoseconds
time_ex <= 1.2 sec; -- 1200 milliseconds
```

Énoncé `after`

- L'énoncé `after` ajoute un aspect temporel à une assignation
- La partie de l'énoncé qui précède la virgule est une assignation qui fonctionne comme toute assignation normale
- La deuxième partie de l'énoncé spécifie une nouvelle valeur pour le signal, qui prendra effet au temps (futur) indiqué

```
<signal> <= <valeur_initiale>, <valeur_finale> after <temps>;
```

Signal de mise à zéro

Voici un exemple d'utilisation pour créer un signal de mise à zéro

```
reset <= '1', '0' after 1 us;
```

Signal d'horloge

- L'exemple suivant montre une méthode simple pour générer un signal d'horloge
- La période obtenue sera le double du délai, soit ici 20 ns.

```
clock <= not clock after 10 ns;
```

Énoncé `wait`

- L'énoncé `wait` suspend l'exécution dans un bloc processus pendant un certain temps
- Rappelons que le processus ne peut pas avoir de liste de sensibilité
- Trois types d'usage de `wait` sont possibles

```
wait for <time>;

wait until <condition> for <time>;

wait on <signal_name>;
```

Énoncé `wait ... 2`

- Dans le premier cas, l'exécution est stoppée pendant la durée indiquée
- Dans le deuxième cas, l'exécution est stoppée jusqu'à ce que la condition soit remplie
- Il est possible de fonder les conditions sur des macros pour les fronts montants `rising_edge` ou descendants `falling_edge`
- La portion `for` est optionnelle
- Elle permet de prévoir un temps d'attente maximal
- L'attente s'arrêtera donc si la condition est remplie ou après l'écoulement du temps spécifié
- Dans le troisième cas, on attend simplement qu'un évènement se produise sur le signal spécifié pour cesser l'attente

Liste de signaux en attente d'évènement

Il est possible de spécifier une liste de signaux en les séparant par des virgules

```
wait on sig_a, sig_b;
```

Exemples de banc d'essai

Testons un circuit séquentiel simple comportant deux entrées A et B qui passent par une porte ET et une bascule avec sortie Q

Création d'une entité vide pour le banc d'essai

L'extrait de code suivant montre le point de départ de notre banc d'essai

```
entity exemple_tb is
end entity exemple_tb;

architecture test of exemple_tb is

--

end architecture exemple_tb;
```

Instanciation du modèle à tester

- Nous faisons une instanciation directe
- L'extrait de code suivant montre comment la spécifier, en supposant que les signaux `in_a`, `in_b` et `out_q` ont été déclarés précédemment

```
-- Instanciation du modèle à tester
dut: entity work.exemple_design(rtl)
  port map (
    a => in_a,
    b => in_b,
    q => out_q
   );
```

Génération de l'horloge et du signal de mise à zéro

- Il faut ensuite générer les signaux d'horloge et de mise à zéro, en spécifiant les éléments temporels
- Les deux signaux seront définis de façon concurrente
- Nous prévoyons une inversion d'horloge à toutes les 10 ns, ce qui donne une période de 20 ns qui correspond à une fréquence de 50 MHz

Génération de signaux de test

L'extrait de code suivant montre les détails

```
-- Reset et clock
clock <= not clock after 10 ns;
reset <= '1', '0' after 50 ns;
```

Stimulus

- Le dernier élément à spécifier est le stimulus, c'est-à-dire les signaux qui seront appliqués aux entrées de notre modèle à tester
- Nous utilisons un processus pour générer les quatre combinaisons possibles de nos deux entrées

Stimulus pour entrées

```
  -- Génération du stimulus
  stimulus:
  process begin
    -- Attendre que reset soit activé
    wait until (reset = '0');

    -- Générer chaque condition, avec 2 périodes entre chaque
    -- itération pour laisser du temps pour la propagation
    and_in <= "00";
    wait for 20 ns;
    and_in <= "01";
    wait for 20 ns;
    and_in <= "10";
    wait for 20 ns;
    and_in <= "11";
    -- Fin du test
    wait;
  end process stimulus;
end architecture exemple_tb;
```

Exemples complets : 1

Dans le premier exemple, nous faisons appel ici au mot réservé `alias` qui permet de rendre le code plus facile à comprendre en nommant un sous-ensemble du type `array` qui a été utilisé pour générer les combinaisons d'entrées

Premier exemple complet de banc d'essai

```
1  entity exemple_tb is
2  end entity exemple_tb;
3
4  architecture test of exemple_tb is
5    signal clock  : std_logic := '0';
6    signal reset  : std_logic := '1';
7    signal and_in : std_logic_vector(1 down 0) := (others => '0');
8    alias in_a is and_in(0);
9    alias in_b is and_in(1);
10   signal out_q  : std_logic;
11 begin
12   -- Reset et clock
13   clock <= not clock after 10 ns;
14   reset <= '1', '0' after 50 ns;
```

Premier exemple complet de banc d'essai ... 2

```
15  -- Instanciation du modèle à tester
16  dut: entity work.exemple_design(rtl)
17    port map (
18      a => in_a,
19      b => in_b,
20      q => out_q );
```

Premier exemple complet de banc d'essai ... 3

```
21    -- Génération du stimulus
22    stimulus:
23    process begin
24      wait until (reset = '0');  -- Attendre reset relâché
25      -- Générer chaque condition, avec 2 périodes entre chaque
26      -- itération pour laisser du temps pour la propagation
27      and_in <= "00";
28      wait for 2 ns;
29      and_in <= "01";
30      wait for 2 ns;
31      and_in <= "10";
32      wait for 2 ns;
33      and_in <= "11";
34      -- Fin du test
35      wait;
36    end process stimulus;
37  end architecture exemple_tb;
```

Deuxième exemple complet de banc d'essai

- Le listage du deuxième exemple de banc d'essai, qui teste le fonctionnement d'un compteur haut/bas de quatre bits modélisé de façon comportementale, est présenté en trois portions
- Une façon différente de générer le signal d'horloge y est utilisée

Deuxième exemple complet de banc d'essai, portion 1

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity up_down_counter is
6    port( clock : in std_logic;
7          reset : in std_logic;
8          up_down : in std_logic;
9          counter : out std_logic_vector(3 downto 0));
10 end up_down_counter;
```

Deuxième exemple complet de banc d'essai, portion 2

```
11  architecture bhv of up_down_counter is
12    signal t_count: unsigned(3 downto 0);
13  begin
14    process (clock, reset)
15    begin
16      if (reset='1') then
17        t_count <= "0000";
18      elsif rising_edge(clock) then
19        if up_down = '0' then
20          t_count <= t_count + 1;
21        else
22          t_count <= t_count - 1;
23        end if;
24      end if;
25    end process;
26
27    counter <= std_logic_vector(t_count);
28  end bhv;
```

Deuxième exemple complet de banc d'essai, portion 3

```
29  library ieee;
30  use ieee.std_logic_1164.all;
31  use ieee.numeric_std.all;
32
33  entity tb_up_down is
34  end tb_up_down;
```

Deuxième exemple complet de banc d'essai, portion 4

```
35  architecture behavior of tb_up_down is
36
37  -- déclaration de composant pour le modèle à tester
38  component up_down_counter
39  port(
40  clock : in std_logic;
41  reset : in std_logic;
42  up_down : in std_logic;
43  counter : out std_logic_vector(3 downto 0)
44  );
45  end component up_down_counter;
```

Deuxième exemple complet de banc d'essai, portion 5

```
46  --Inputs
47  signal clock : std_logic := '0';
48  signal reset : std_logic := '0';
49  signal up_down : std_logic := '0';
50
51  --Outputs
52  signal counter : std_logic_vector(3 downto 0);
53
54  -- Clock period definitions
55  constant clock_period : time := 20 ns;
```

Deuxième exemple complet de banc d'essai, portion 6

```
56  begin
57
58  -- Instanciation du composant à tester
59  uut: component up_down_counter
60    port map (
61      clock => clock,
62      reset => reset,
63      up_down => up_down,
64      counter => counter
65  );
```

Deuxième exemple complet de banc d'essai, portion 7

```
66  -- Processus d'horloge
67  clock_process: process
68  begin
69  clock <= '0';
70  wait for clock_period/2;
71  clock <= '1';
72  wait for clock_period/2;
73  end process;
```

Deuxième exemple complet de banc d'essai, portion 8

```
74  -- Processus de stimulus
75  stim_proc: process
76  begin
77  -- hold reset state for 100 ns.
78  wait for 20 ns;
79  reset <= '1';
80  wait for 20 ns;
81  reset <= '0';
82  up_down <= '0';
83  wait for 200 ns;
84  up_down <= '1';
85  wait;
86  end process;
87
88  end;
```

Compilation et simulation

Voici enfin les grandes étapes permettant de passer de la spécification d'un modèle à une simulation du comportement du circuit modélisé

Créer un fichier VHDL

- Ce fichier ne contient qu'une seule paire entité.architecture de niveau supérieur, qui sera le banc d'essai
- Il y aura typiquement d'autres entités de niveau inférieur, notamment le modèle de circuit à simuler

Entité de niveau supérieur

```
entity MACHIN is
...
end MACHIN;

architecture TRUC of MACHIN is
...
end TRUC;
```

Compiler cette description

- Les commandes à utiliser dépendent du simulateur qui est utilisé, mais deux étapes sont normalement requises :
  - l'analyse du code source
  - l'élaboration du design

Simuler

- Une fois le design élaboré, on peut simuler le résultat
- Si le simulateur ne permet pas directement la visualisation des résultats, il faudra sauvegarder les résultats de simulation sous un format qui permettra ensuite de les visualiser avec un autre outil

Visualiser

- Pour visualiser le résultat, on peut utiliser un outil intégré ou encore un outil externe qui permet de visualiser les signaux obtenus (formes d'ondes, résultats interprétés)

Préparation et simulation des modèles VHDL

Éditeurs

- N'importe quel éditeur de programmation peut être utilisé pour éditer les modèles VHDL
- Certains simulateurs comportent un éditeur intégré
- Il peut être avantageux d'utiliser un éditeur avec fonction de surlignage syntaxique pour le langage

Simulateurs gratuits

Voici quelques simulateurs gratuits

Modelsim/Questa

Des versions de ce simulateur sont offertes par plusieurs fabricants de circuits intégrés programmables

- Intel
- Lattice
- Microchip

Modelsim/Questa (Intel)

FPGA Software Download Center

- Fonctionne sous Windows ou Linux (Red Hat ou Ubuntu)
- Un des simulateurs les plus populaires
- L'utilisation requiert une licence (gratuite) spécifique à un ordinateur donné, qu'il faut demander par courriel
- L'installation et l'activation comportent plusieurs étapes

Modelsim/Questa (Lattice)

iCEcube2 Design Software

- Fonctionne sous Windows ou Linux (Red Hat)
- Fait partie d'une suite logicielle en support à la gamme de FPGA du fabricant
- L'utilisation requiert une licence (gratuite) spécifique à un ordinateur donné

Modelsim/Questa (Microchip)

Libero SoC Design Suite

- Fonctionne sous Windows ou Linux (Red Hat)
- Fait partie d'un suite logicielle (Libero) en support à la gamme de FPGA du fabricant
- L'utilisation requiert une licence (gratuite) spécifique à un ordinateur donné

Active-HDL (version étudiante)

Free Active-HDL Student Edition

- Fonctionne sous Windows
- Licence gratuite pour la communauté étudiante (on doit indiquer son université)
- L'inscription donne accès à une page de téléchargement

Vivado (Xilinx)

Xilinx téléchargements

- Fonctionne sous Windows ou Linux (Red Hat ou Ubuntu)
- Cette suite pour conception de circuits intégrés programmables comporte un simulateur
- L'utilisation requiert une licence (gratuite) spécifique à un ordinateur donné

GHDL/GTKWave

`http://ghdl.free.fr/`

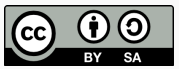
`https://github.com/ghdl/ghdl`

`http://gtkwave.sourceforge.net/`

`https://github.com/gtkwave/gtkwave`

Ces deux logiciels sont à code source ouvert, donc entièrement gratuits

- Fonctionnent sous Windows, Mac ou Linux
- GHDL est utilisé pour la simulation
- GTKWave est utilisé pour visualiser les résultats



EDA Playground

EDA Playground

- Cette option, utilisable via un fureteur Web, ne nécessite pas d'installation et peut donc s'utiliser sur toutes les plateformes
- On doit s'enregistrer
- Il est possible de choisir le simulateur
- On y trouve également des exemples de codes