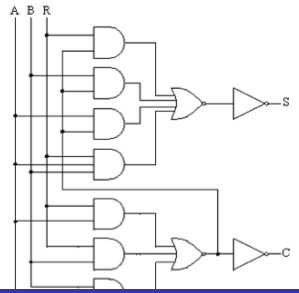
Design Reuse and integration

Cours ME2

S. Pillement

ETN 5 options SETR et SMTR –
IETR/Polytech – Université de Nantes

4 octobre 2021



Présentation du cours

- Objectifs du cours :
  - Revoir les connaissances de base pour la conception de système complexe (système à $\mu-$processeurs, systèmes embarqués)
  - Utilisation et développement de code réutilisable (*IP-based design*)
  - Découverte de Verilog
- Organisation du cours
  - 3 h de CM : Rappel de VHDL et généricité, Introduction à Verilog
  - 12h TP mini-projet : Réalisation d'un cœur de processeur mixte VHDL-Verilog par réutilisation de codes
  - Evaluation : rapport de mini-projet

Déroulement du cours

- **Chapitre 1 : Introduction - Définitions**
- **Chapitre 2 : Le langage VHDL**
  - Rappel
  - La généricité
  - La simulation
- **Chapitre 3 : Le langage Verilog**
  - Introduction
  - Notions avancées

Partie I

Introduction - Définitions

I Introduction - Définitions

I Introduction - Définitions

Pourquoi la réutilisation ?

Les systèmes embarqués :

- Il s'agit de systèmes complexes
- incluant des parties matérielles et des parties logicielles
- soumis à de nombreuses contraintes (coût, efficacité, consommation, fiabilité, . . .)

Pourquoi la réutilisation ?

Autres considérations à prendre en compte :

- Considérer le "Time To Market", un retard d'un mois à la mise sur le marché peut induire une perte de l'ordre de 30% dans la rentabilité !
- Les aspects liés à la vérification peuvent représenter de 60 à 70% du cycle de conception
- Faciliter la maintenance et les mises à jour est crucial

Pourquoi la réutilisation ?

Approche par réutilisation

concevoir à partir de composants existants (IP – Intellectual Property components), réponds à ces challenges

Nécessite

- des méthodes de conception qui doivent pouvoir prendre en compte tous ces aspects
- des caractéristiques des IP bien documentées
- des IP génériques (ou du moins paramétrables)

I Introduction - Définitions

Différents types d'IP

IP Hard :

- proposées au format GDSII (Graphic Data System II - standard de facto pour l'industrie des semi-conducteurs)
- optimisées pour une implémentation ASIC
- peformances parfaitement connues
- processeurs, standard cells, mémoires, I/Os, blocs analogiques
- pas portables

Différents types d'IP

IP Soft :

- Ecrite au niveau RTL. Utilisation de HDL portable
- non optimisées pour la fabrication en fonderies
- les performances (énergie, temps, surface) ne sont pas complètement connues
- Valeur dans la fonctionnalité, la réusabilité et la conformité aux standards
- beaucoup utilisé pour une implémentation FPGA mais pas que

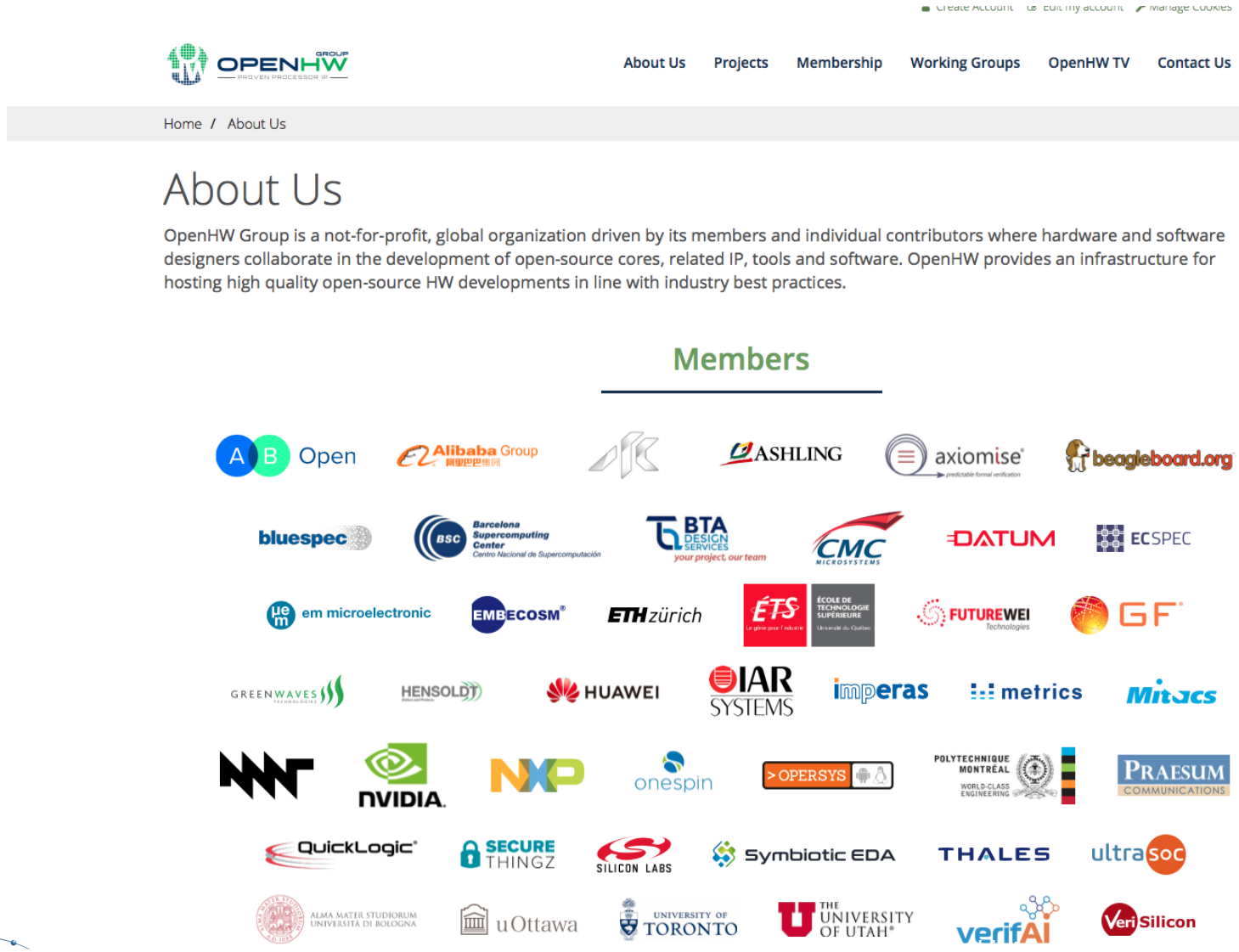
Flot d'intégration

La définition d'une IP inclut :

- Expertise de conception et des données
- La connaissance des process de fabrication
- L'encapsulation et des procédures de tests et intégration

Nécessite un flot d'intégration, validation robuste basé sur des langages de description génériques

Une approche industrielle



I Introduction - Définitions

Les langages de description matériel

HDL : Hardware Description Language

Deux langages normalisés quasi-équivalents :

- VHDL : Very High Speed Integrated Circuit Hardware Description Language (Europe)
  - Syntaxe dérivée du langage ADA
  - Normes : 1987, 1993, 2002, 2008
- Verilog (USA)
  - Conçu en 1985 par Gateway Design System
  - Syntaxe dérivée du langage C

mais aussi plein d'autres

SystemC, SystemVerilog, Chisel, ABEL, etc . . .

Les HDLs

Généralités

- Deux fonctions :
  - Synthèse logique sur ASIC ou FPGA pour l'implémentation
    - Mots clés adaptés à la conception matérielle
  - Simulation avec des mots clés dédiés non synthétisable
- Syntaxe d'un langage impératif

Attention danger

Les langages de description HDL ne sont **PAS** des langages de programmation.

Les HDLs

Généralités

- Deux fonctions :
  - Synthèse logique sur ASIC ou FPGA pour l'implémentation
    - Mots clés adaptés à la conception matérielle
  - Simulation avec des mots clés dédiés non synthétisable
- Syntaxe d'un langage impératif

Attention danger

Les langages de description HDL ne sont **PAS** des langages de programmation.

Partie II

Le langage VHDL

II Le langage VHDL

II Le langage VHDL

Structure d'une description VHDL

Une description VHDL est composée de 2 parties indissociables à savoir :

- L'entité (entity), elle définit les entrées et sorties du composant. Elle décrit une interface externe unique.
- L'architecture (architecture), elle contient les instructions VHDL permettant de réaliser le fonctionnement attendu. Il peut exister différentes architectures pour une même entité.

Structure d'une description VHDL

```
-- Ceci est un commentaire
-- Déclaration des bibliothèques
library ieee;
Use ieee.std_logic_1164.all;
Use ieee.numeric_std.all;

    -- déclaration des entrées-sorties
entity MonEt is
  port(IN0, IN1: in std_logic;
    S : out std_logic
  );
end MonEt;
```

Structure d'une description VHDL

```
architecture DESCRIPTION of MonEt is
-- zone de déclaration, signaux et composants
begin
-- zone d'instructions, ici tout est concurrent
S <= (IN1 and IN0);
end DESCRIPTION;
```

Déclaration des bibliothèques

Les bibliothèques sont divisées en paquetage. Il faut appeler tous les paquetages à utiliser.

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

Elles contiennent les définitions :

- des types de données,
- des fonctions et sous programmes permettant de réaliser des opérations arithmétiques et logiques,...

Type de signaux

Les TYPES principaux utilisés pour les données sont :

- le std_logic pour une variable binaire (’0’, ’1’, ’Z’, ’X’).
- le std_logic_vector pour un bus (signé ou pas) composé de plusieurs std_logic. (deprecated)

On favorisera l’utilisation des types :

- signed pour pour un bus signé
- unsigned pour un bus non signé

Type de signaux

Il existe aussi :

- integer : entier négatif ou positif
- natural : entier positif ou nul
- positive : entier positif
- bit et bit_vector : équivalent au std_logic mais avec deux valeurs seulement (’0’, ’1’)
- boolean : les deux valeurs possibles sont false et true
- real : flottant compris entre -1.0E38 et 1.0E38

L'entité VHDL

```
entity NOM_DE_L_ENTITE is
port ( Description des signaux d'entrées /sorties );
end NOM_DE_L_ENTITE;
```

Attention : Après la dernière définition de signal de l'instruction port il ne faut jamais mettre de point virgule.
On doit définir pour chaque signal :

- le NOM_DU_SIGNAL, le premier caractère doit être une lettre, sa longueur est quelconque, mais ne doit pas dépasser une ligne. VHDL n'est pas sensible à la "casse".
- le sens, in, out, inout, buffer (à éviter)
- le type.

L'architecture VHDL

L'architecture décrit le fonctionnement souhaité pour un circuit ou une partie du circuit.

```
architecture nom_architecture of nom_entité is
{ déclaration_de_modules
| déclaration_de_constantes
| déclaration_de_signaux_internes

begin
{ instructions_concurrentes
end [architecture] [nom_architecture];
```

Un couple Entité/Architecture définit un module (ou component) qui peut être réutilisé dans un autre module.

Exemple d'un composant

```
-- Opérateurs logiques de base
entity PORTES is
port (A,B :in std_logic;
Y : out unsigned (3 downto 0));
end PORTES;
architecture DESCRIPTION of PORTES is
begin
   Y(0) <= A and B;
   Y(1) <= A or B;
   Y(2) <= A xor B;
   Y(3) <= not A;
end DESCRIPTION;
```

Exercice

Complétez le code VHDL suivant :

```
library ieee;
Use ieee.std_logic_1164.all;
...
entity ADD_4bits is
   port(IN0, IN1 : in ...;
      S : ... ...
   );
end ADD_4bits;
architecture Behavior of ADD_4bits is
begin
...
end Behavior;
```

Exercice

```
library ieee;
Use ieee.std_logic_1164.all;
Use ieee.numeric_std.all;
entity ADD_4bits is
   port(IN0, IN1 : in unsigned (3 downto 0);
      S : out unsigned (3 downto 0)
   );
end ADD_4bits;
architecture Behavior of ADD_4bits is
begin
s<=IN0+IN1;
end Behavior;
```

Exercice

Complétez le code VHDL suivant :

```
library ieee;
Use ieee.std_logic_1164.all;
...
entity ADD_4bits_SIGNE is
   port(IN0, IN1 : in ...;
      S : ... ...
   );
end ADD_4bits_SIGNE;
architecture Behavior of ADD_4bits_SIGNE is
begin
...
end Behavior;
```

Exercice

```
library ieee;
Use ieee.std_logic_1164.all;
Use ieee.numeric_std.all;
entity ADD_4bits_SIGNE is
   port(IN0, IN1 : in signed (3 downto 0);-- changement ici
      S : out signed (3 downto 0)-- et ici
   );
end ADD_4bits_SIGNE;
architecture Behavior of ADD_4bits_SIGNE is
begin
s<=IN0+IN1;
end Behavior;
```

Quelques caractéristiques du langage

- Les instructions modélisent un câblage matériel lorsqu'elles sont destinées à la synthèse logique.
- L'ordre des instructions n'a pas d'importance car, elles modélisent naturellement le parallélisme d'un circuit.
- Plusieurs styles de description
  - tous les styles ne conviennent pas à toutes les applications
  - toutes les instructions ne sont pas synthétisables

Mux 4 vers 1

```
entity mux_4to1_top is
Port (SEL : in unsigned (1 downto 0);
   A : in unsigned (3 downto 0);
X : out STD_LOGIC);
end mux_4to1_top;
architecture Behavioral of mux_4to1_top is
begin
with SEL select
X <= A(0) when "00",
A(1) when "01",
A(2) when "10",
A(3) when "11",
'0' when others;
end Behavioral;
```

Mux 4 vers 1 V2

```
entity mux_4to1_top is
Port (SEL : in unsigned (1 downto 0);
   A : in unsigned (3 downto 0);
X : out STD_LOGIC);
end mux_4to1_top;
architecture Behavioral2 of mux_4to1_top is
begin
X <= A(0) when SEL= "00"else
A(1) when SEL= "01" else
A(2) when SEL = "10" else
A(3) when SEL= "11" else
'0' ;
end Behavioral2;
```

Processus

Dans une description VHDL tout est concurrent. L'ordre des instructions n'importe pas.

Processus

Certains comportements d'un sous-ensemble matériel peuvent être décrits de façon algorithmique ; il faut alors les définir comme des processus.

Dans l'architecture utilisatrice, un processus est considéré comme une instruction concurrente. Dans un processus les instructions sont séquentielles.

Processus

- À l'intérieur d'un processus, on peut utiliser des instructions conditionnelles et d'itération
  - if . . . then . . . else . . . end if ;
  - case . . . when . . . end case ;
  - for . . . loop . . . end loop ;
  - while . . . loop . . . end loop ;
- L'ordre d'écriture des instructions à l'intérieur du processus est déterminant. Un processus est un ensemble de phases séquentielles
- Sémantiquement, c'est une boucle infinie, à moins qu'il n'y ait une liste de sensibilité
- Un modèle VHDL peut être vu comme un ensemble de processus exécutés en parallèle

Process

- le nom (Etiquette) du process est optionnel

```
Etiquette : process(liste sensibilité)
  déclaration variables et signaux
    begin
      Instructions séquentielles
    end process Etiquette;
```

Mux 4 vers 1 V3

```
architecture Behavioral3 of mux_4to1_top is
begin
process(A,SEL)
      begin
         case SEL is
         when "00" => X <= A(0);
         when "01" => X <= A(1);
         when "10" => X <= A(2);
         when "11" => X <= A(3);
         when others => X <= '0';
   end case;

      end process;
end Behavioral3;
```

Description structurelle en VHDL

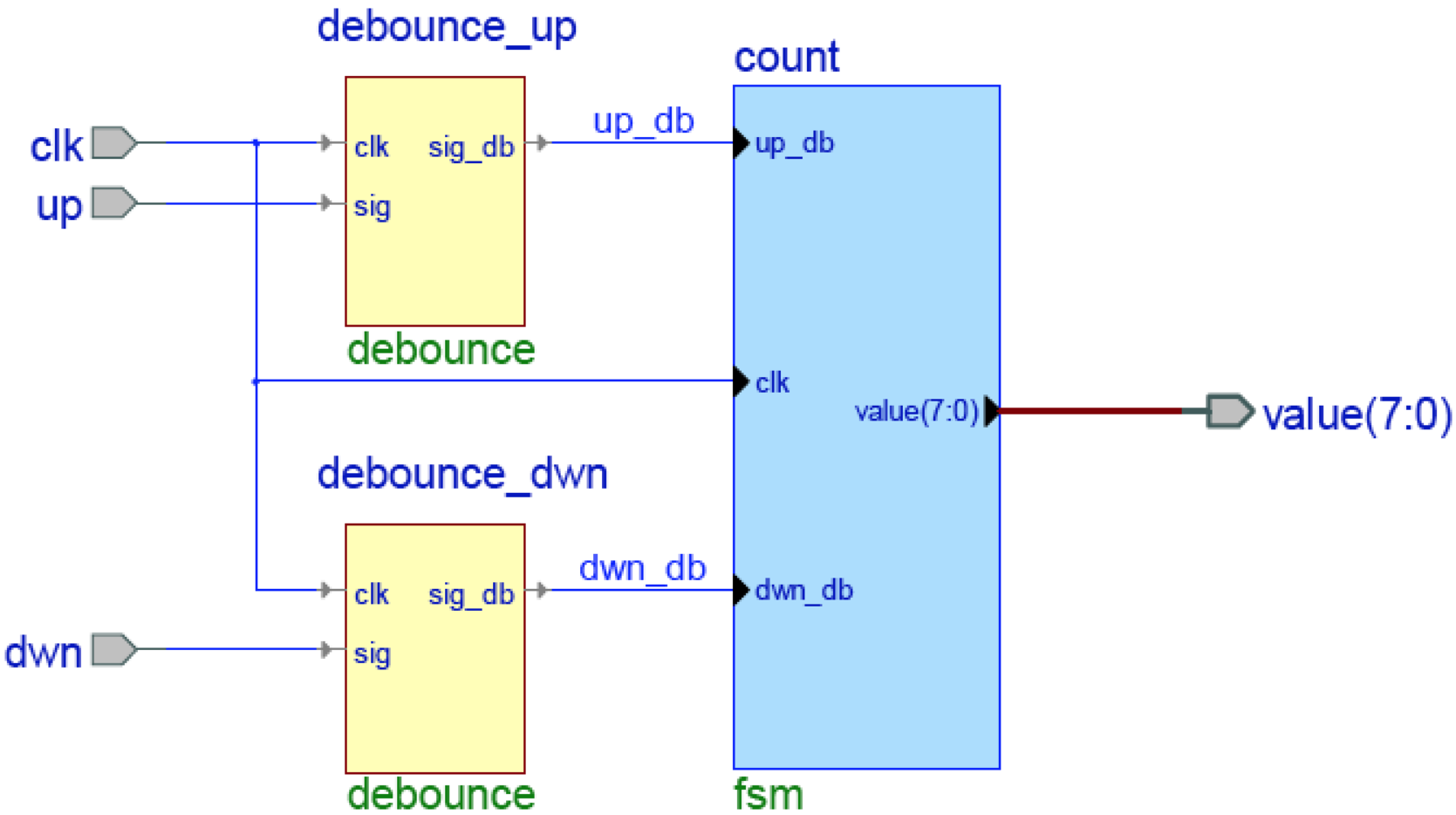
- Lors de la conception d'un système complexe, il est conseillé de faire une décomposition en blocs dont les comportements sont suffisamment simples à décrire.
- Le concepteur a aussi intérêt à procéder à une décomposition lorsqu'une même fonction peut être employée plusieurs fois.
- Dans le style structurel, les blocs constituants sont appelés des composants.

Description structurelle en VHDL

Un composant permet :

- de décrire une seule fois en HDL une fonction qu'on a l'intention d'utiliser plusieurs fois,
- de ranger la description HDL dans une bibliothèque,
- d'utiliser la fonction associée, à volonté, par un simple procédé d'instanciation (graphique ou textuel)

Exemple


debounce_up
count
clk
up
clk
sig_db
sig
up_db
up_db
debounce
clk
value(7:0)
value(7:0)
debounce_dwn
dwn_db
dwn_db
clk
sig_db
dwn
sig
debounce
fsm

Description structurelle en VHDL 1/3

Le système est composé de :

- Un premier fichier, debounce.vhd, qui comporte le couple entité-architecture descriptif du bloc debounce.
- Un deuxième fichier, fsm.vhd, comporte le couple entité-architecture descriptif du bloc fsm.
- Un troisième fichier, main.vhd, qui est qualifié « TOP LEVEL UNIT » qui comporte :
  - une entité décrivant les entrées-sorties du système global
  - une architecture comportant des déclarations et la description HDL du comportement du système global

Description du système

```
-- FICHIER main.vhd "TOP LEVEL UNIT"

library IEEE;
use IEEE.std_logic_1164.all;
entity main is
port(
    clk : in STD_LOGIC;
    dwn : in STD_LOGIC;
    up : in STD_LOGIC;
    value : out STD_LOGIC_VECTOR(7 downto 0));
end main;
```

Description structurelle en VHDL 2/3

- les composants *debounce* et *fsm* sont déclarés afin que le compilateur puisse accéder à leur description par une simple référence à leur nom

Description du système

```
architecture structural of main is
----- Component declarations ------
component debounce
 port ( clk : in STD_LOGIC;
    sig : in STD_LOGIC;
    sig_db : out STD_LOGIC);
 end component;
 component fsm
  port ( clk : in std_logic;
    dwn_db, up_db : in std_logic;
    value : out STD_LOGIC_VECTOR(7 downto 0));
 end component;
 ----- Signal declarations -----
signal dwn_db, up_db : std_logic;
```

Description structurelle en VHDL 3/3

- L'instanciation a pour effet d'associer des signaux effectifs (ceux de l'architecture appelante) aux signaux formels (ceux du composant)
- Une instance de composant est une copie distincte d'une description dans une architecture utilisatrice ; en synthèse, chaque instance emploie des ressources physiques propres
- Un composant instancié est toujours marqué d'une étiquette, à la manière d'un repère sur un schéma
  - le composant *debounce* est instancié deux fois
  - Le composant *fsm* est instancié une fois

Description du système

```
begin
----- Component instantiations -----
count : fsm
    port map( clk => clk,
              dwn_db => dwn_db,
              up_db => up_db,
              value => value);
debounce_dwn : debounce
    port map( clk, dwn, dwn_db);
debounce_up : debounce
    port map( clk => clk, sig => up, sig_db => up_db);
end structural;
```

II Le langage VHDL

La généricité

C'est un moyen de transmettre une information à un bloc :

- Vu de l'extérieur du bloc, la généricité == paramètre(s)
- Vu de l'intérieur du bloc, paramètres == constantes

Intérêts de la généricité :

- description de composants généraux
- permettre la réutilisation des composants :
  - enrichissement progressif de la bibliothèque de travail
  - description de la bibliothèque par des modèles génériques
- assure une plus grande rapidité de développement

Généricité

Généralement, c'est l'entité qui est générique :

```
entity ... is
      generic (...;
           ... ;
         ... );
   port (...;
          ... );
end ... ;
```

Registre générique

```
entity registre is
   generic (NbBits : INTEGER := 8);
   port (D : in unsigned (NbBits−1 downto 0);
         clk : in std_logic ;
         Q : out unsigned (NbBits−1 downto 0));
end registre ;
architecture comport of registre is
begin
   stock:process(clk)
      begin
         if(clk'event and clk='1') then Q<=D;
      end if;
   end process stock;
end comport;
```

Utilisation d'une entité générique

```
architecture structure of Systeme is
-- declaration des composants
signal BusData : unsigned(15 downto 0);
signal BusAddress : unsigned(31 downto 0);
begin
   ...
   Regi : registre
      generic map (16)
      port map (BusData, SClock, BusData);
   Regj : registre
      generic map (32)
      port map (BusAddress, SClock, BusAddress);
   ...
end structure;
```

Registre générique

Mais l'architecture aussi doit être générique

```
architecture comport of registre is
begin
   process
      begin
         if (clk'event and clk = '1') then
            if (Enable = '1') then
                  Q <= D;
            else
                  Q <= "ZZZZZZZZ"; -- pas générique
            end if;
         end if;
   end process;
end comport;
```

Registre générique

```
architecture comport of registre is
begin
   process
      begin
         if (clk'event and clk = '1') then
            if (Enable = '1') then
                  Q <= D;
            else
                  Q <= (others => 'Z'); -- générique sur le nb de
                        bits
            end if;
         end if;
   end process;
end comport;
```

Attributs VHDL

Les attributs sont des propriétés spécifiques que l'on peut associer aux signaux et aux types.

- Attribut valeur sur des types scalaires ou des éléments (signaux, constantes, variables) de type scalaire
  - 'left, 'right, 'high, 'low, 'length
- Attribut fonction sur des types discrets ordonnés
  - 'pos, 'val, 'succ, 'pred, 'leftof, 'rightof
- Attribut fonction sur des signaux
  - 'event, 'last_event, 'last_value
- Attribut intervalle sur un signal dimensionné
  - 'range, 'reverse_range

Attributs VHDL

```
type address is integer range 7 downto 0;
address'left = 7
address'right = 0
address'low = 0
address'high = 7
```

La valeur d'un attribut peut être exploitée dans une expression.

```
if (clk'event and clk='1') then
...
end if;
```

Différents attributs

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

entity attributs is port(
  vector_dwn      : in   std_logic_vector(15 downto 0);
  vector_up : in   std_logic_vector(0 to 7);
  x           : out  std_logic_vector(7 downto 0);
  y           : out  std_logic_vector(0 to 11);
  z           : out  std_logic_vector(7 downto 0));
end attributs;

architecture arch_attributs of attributs is
begin
x(0) <= vector_dwn(vector_dwn'left);
x(1) <= vector_dwn(vector_dwn'right);
x(2) <= vector_up(vector_up'left);
x(3) <= vector_up(vector_up'right);
x(4) <= vector_dwn(vector_dwn'high);
x(5) <= vector_dwn(vector_dwn'low);
x(6) <= vector_up(vector_up'high);
x(7) <= vector_up(vector_up'low);
y(vector_up'range)              <= "00001111";
z(vector_up'reverse_range)    <= "00110011";

end arch_attributs;
```

```
x(0) = vector_dwn(15)
x(1) = vector_dwn(0)
x(2) = vector_up(0)
x(3) = vector_up(7)
x(4) = vector_dwn(15)
x(5) = vector_dwn(0)
x(6) = vector_up(7)
x(7) = vector_up(0)

y(0) = GND
y(1) = GND
y(2) = GND
y(3) = GND
y(4) = VCC
y(5) = VCC
y(6) = VCC
y(7) = VCC

z(0) = VCC
z(1) = VCC
z(2) = GND
z(3) = GND
z(4) = VCC
z(5) = VCC
z(6) = GND
z(7) = GND
```

Un ET générique

```
entity Et_N is
   generic ( N : Natural )
   port ( Entrees : in unsigned ( 1 to N ) ;
   sortie : out std_logic );
end Et_N ;
architecture comportement of Et_N is
begin
   process
      variable V : std_logic := '1' ;
      begin
         for i in 1 to N loop
            V := V and Entrees (i) ;
         end loop ;
         Sortie <= V ;
end process ;
```

Boucle de génération

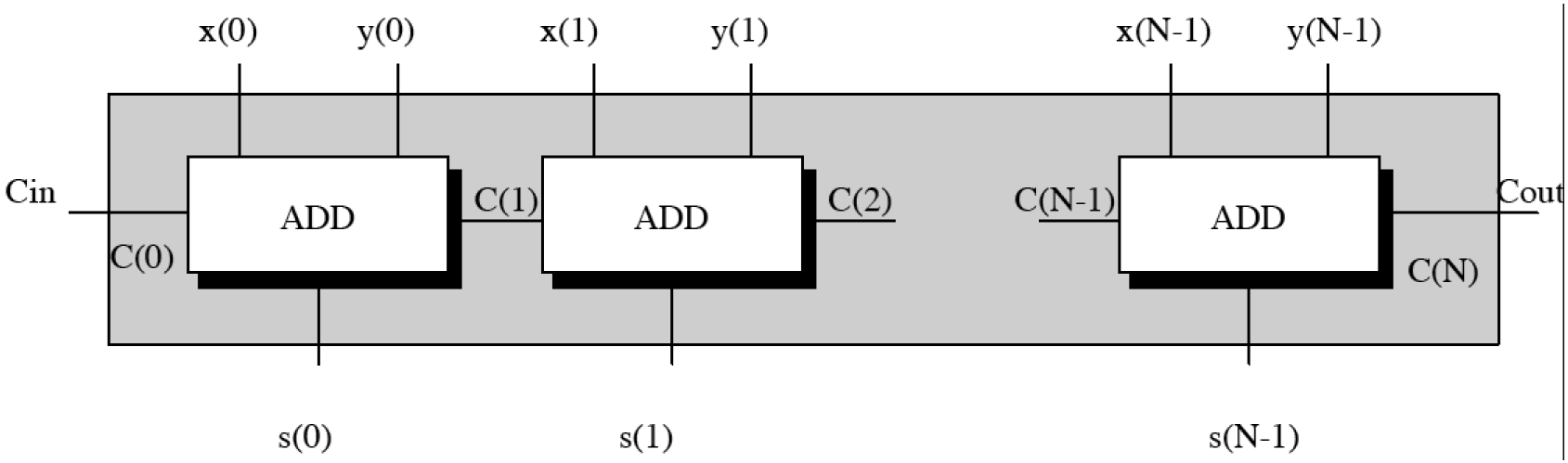
```
entity Demux is
    generic (NbBits : Natural := 8;
             NbCmd : Natural := 4;
             NbSorties : Natural := 16);
    port ( Entree : in std_logic_vector(NbBits -1 downto 0) ;
           Selection : in std_logic_vector(NbCmd -1 downto 0);
           Sorties : out array (0 to NbSorties -1) of
                std_logic_vector(NbBits -1 downto 0));
end Demux ;
```

Boucle de génération

```
architecture comportement of Demux is
begin
    boucle : for i in Sorties'range generate
        Sorties(i) <= Entree when Selection = i
                                  else (others => '0');
    end generate;
end comportement;
```

Un autre exemple

additionneur structurel générique :



- construit à partir d'un additionneur 1 bits
- assemblage des N additionneurs 1 bits afin de réaliser l'additionneur complet
- la valeur de N est inconnue avant l'instanciation du composant

Additionneur générique

On dispose de l'additionneur de base :

```
entity Add is
   port ( A, B, Cin : in std_logic;
          S, Cout : out std_logic);
end Add ;
```

L'entité de l'additionneur générique :

```
entity AdditionneurN is
   generic (N : Natural);
   port ( X, Y : in unsigned ( N-1 downto 0) ;
          Cin : in std_logic;
          S : out unsigned (N-1 downto 0);
          Cout : out std_logic);
end AdditionneurN ;
```

Additionneur générique

```
architecture structurelle of AdditionneurN is
component Add
   port ( A, B, Cin : in std_logic;
          S, Cout : out std_logic);
end component;
signal C : unsigned(0 to N);
begin
   boucle:for I in 0 to N−1 generate
      Instance : Add
         port map (X(I), Y(I), C(I), S(I), C(i+1));
   end generate;
   C(0) <= Cin;
   Cout <= C(N);
end structurelle ;
```

II Le langage VHDL

Validation

- Une bonne maîtrise de la conception passe par la validation de chaque étape
- Une première simulation comportementale pour servir de référence. complet
- Utilisation du même benchmark pour toutes les étapes
- Différents outils et approches :
  - modèle mathématique, comportemental, structural, physique

Validation

Comment faire une simulation ?

- instanciation du composant à tester
- initialisation des signaux d'entrées
- application d'une séquence de stimuli :
  - à partir d'un process et d'affectations des signaux d'entrées
  - à partir d'un fichier contenant des vecteurs de test
- analyse des résultats, analyse des transitions des sorties :
  - affichage des erreurs éventuelles

Instructions de simulation

```
constant TempsCycle : time := 10 ns ;
type TableauEntrees is array (0 to 2) of std_logic;
type TableauVecteur is array (0 to 7) of TableauEntrees;
constant Vecteur : TableauVecteur :=
(
('0', '0', '0', '0', '0'),
('0', '0', '1', '0', '1'),
('0', '1', '0', '0', '1'),
('0', '1', '1', '1', '0'),
('1', '0', '0', '0', '1'),
('1', '0', '1', '1', '0'),
('1', '1', '0', '1', '0'),
('1', '1', '1', '1', '1')
);
signal SA, SB, SOut, SCin, SCout : bit ;
```

Instructions de simulation

```
Simulation : process
begin
   wait for 10 ns;
   for i in Vecteur'range(1) loop
      SA <= Vecteur(i)(0);
      SB <= Vecteur(i)(1);
      SCIn <= Vecteur(i)(2);
      wait TempsCycle;
         assert (SCout = Vecteur(i)(3))
            report "Probleme sur la sortie Cout"
            severity warning;
         assert (SOut = Vecteur(i)(4))
            report "Probleme sur la sortie S"
            severity warning;
   end loop;
```

Lecture d'un fichier

```
...
file VecteursIN : integer is in "VecteursIN";
...
begin
   wait for 10 ns;
   readline(VecteursIN, ligne);
   read(ligne, Tps);
   TempsCycle = Tps ns;
   while not endfile(VecteursIn) loop
      read(ligne, VA);
      read(ligne, VB);
      ...
```

Quelques caractéristiques du langage

- Syntaxe complexe : les outils de développement proposent des modèles (templates) et des convertisseurs de schémas en code VHDL
- Langage strict par rapport aux types et aux dimensions des données : élimination d'un grand nombre d'erreurs de conception dès la compilation
- Bonne portabilité à condition d'écrire un code indépendant de la technologie

Partie III

Le langage Verilog

III Le langage Verilog

III Le langage Verilog

Structure d'une description Verilog

Une description Verilog est monolithique, l'unité de base s'appelle un (module), elle définit :

- les entrées et sorties du composant. Elle décrit une interface externe unique et contient les éléments
- la fonctionnalité attendu.

Structure d'une description Verilog

```
   //Ceci est un commentaire


module module_name ( port_list );
   port declarations;
   ...
   variable declaration;
   ...
   description of behavior
endmodule
```

Règles de nommage Verilog

- Utilisation des lettres (minuscules ou majuscules), chiffres et _ pour les identifiants
- Le premier caractère doit être une lettre, longueur max de 1024 caractères.
- Verilog est sensible à la "casse".

Premier exemple

```
module HalfAdder (A, B, Sum, Carry);
   input A, B;
   output Sum, Carry;
   assign Sum = A ^ B;
   //^ denotes XOR
   assign Carry = A & B;
   // & denotes AND
endmodule
```

Les supports de données

Le langage Verilog utilise deux classes de supports de données distinctes pour les modes concurrents et procedural.

- classe "net" dans le mode concurrent : un net est équivalent à un signal en VHDL, son état est contrôlé en permanence par les éléments aux sorties desquels il est connecté.
- l'état d'un net n'est modifié que par connexion structurelle ("branchement" d'un instance) ou par assignation continue ("branchement" d'un net à une expression).

Les supports de données

- classe "reg" dans le mode procédural : un reg est équivalent à une variable : il subit des affectations instantanées par instructions et conserve son état jusqu'à la prochaine affectation.
- l'état d'un reg n'est modifié que par assignation procédurale (une instruction exécutée de manière séquentielle).

Le bon usage des nets et regs est la seule réelle difficulté de Verilog.

Type de données

- Classe Net : wire (c'est le plus "utile" de cette classe), tri, wor, trior, wand, triand, supply0, supply1 (par défaut les ports du module sont de type net).
- Classe Register : reg (c'est le plus "utile" de cette classe), integer, time, real, realtime
- Valeurs par défaut :
  - Types Net : z
  - Types Reg : x
- Les regs et nets doivent être déclarés :
  - à l'intérieur des modules, mais
  - à l'extérieur des blocs procéduraux
- La déclaration des regs est toujours obligatoire.

Type de données

- Type Wire

```
wire [ msb : lsb ] wire1, wire2, ...
```

  - Exemples

```
wire Reset; // A 1-bit wire
wire [6:0] Clear; // A 7-bit wire
```

- Type Reg

```
reg [ msb : lsb ] reg1, reg2, ...
```

  - Exemples

```
reg [ 3: 0 ] cla; // A 4-bit register
reg cla; // A 1-bit register
```

Types de description Verilog

- Description structurelle, le circuit est définit comme une interconnexion de modules ou de primitives Verilog
- Exemple :

```
not n1(sel_n, sel);
and a1(sel_b, b, sel_b);
and a2(sel_a, a, sel);
or o1(out, sel_b, sel_a);
```

- l'ensemble des signaux ci-dessus sont de la classe net
- sur les primitives la liste des signaux ordonnées commence par la sortie

Types de description Verilog

- Description flot de données, spécifie la sortie comme une fonction des entrées
- Exemple :

```
assign out = (sel & a) | (~sel & b);
```

- le signal out est obligatoirement de la classe net
- le mot clé assign réalise une assignation continue
- l'ordre de description n'a pas d'influence. Assignation concurrente.

Types de description Verilog

Un Exemple avec définition de temps et de retard :

```
‘timescale 1ns/100ps
module HalfAdder (A, B, Sum, Carry);
   input A, B;
   output Sum, Carry;
   assign #3 Sum = A ^ B;
   assign #6 Carry = A & B;
endmodule
```

‘timescale 1ns/100ps indique

- 1 Time unit = 1 ns avec une précision de 100ps (0.1 ns)
- 10.512ns sera interprété comme 10.5ns

Types de description Verilog

- Description comportementale, spécifie le comportement comme un algorithme.
- Approche procédurale.
- Exemple :

```
case (X)
    2'b00: Y = A + B;
    2'b01: Y = A − B;
    2'b10: Y = A / B;
endcase
```

- le signal Y doit être de la classe reg

Types de description Verilog

Un Multiplexeur 2 vers 1 :

```
module mux_2x1(a, b, sel, out);
   input a, b, sel;
   output out;
   reg out;
   always @(a or b or sel)
      begin
         if (sel == 1)
               out = a;
            else out = b;
      end
endmodule
```

Bloc procédural

Il existe deux mots clés permettant de déclarer un bloc procédural (les instructions sont exécutées séquentiellement dans ces blocs) :

- initial ce bloc n'est exécuté qu'une seule fois.
- le bloc marqué par always est exécuté dans une boucle.
  - always @(a or b or sel) déclenche le bloc à chaque changement de valeur d'un des signaux de la liste de sensibilité (changement de valeur de a, b ou sel)
  - always @(posedge clk) (ou always @(negedge clk)) déclenche le bloc sur un front montant (ou descendant) du signal clk

Bloc procédural

- Délais dans une assignation procédurale :

```
Sum = A ^ B;
#2 Carry = A & B;
```

```
Sum = A ^ B;
Carry = #2 A & B;
```

Bloc procédural Exemple 1/3

Simulation d'un registre à décalage à chargement parallèle :

```
'define p=5000; // periode horloge
module test;
   reg [3:0] data_in; // reg vectoriel de 4 bits
   reg ck, load;
   wire ser_data;

   piso4 piso ( .clk(ck),   // instance avec connections par
                .load(load),       // nommage des ports
                .par_in(data_in),
                .ser_out(ser_data) );
   ...
```

Bloc procédural Exemple 2/3

```
...
  initial //============== sequence principale
    begin
      load = 1; data_in = 1;
      #p load = 0;
      #(p*6) load = 1; data_in = 10;
      #p load = 0;
      #(p*6) load = 1; data_in = 'b1101;
      #p load = 0;
      #(p*6) $finish;
    end
...
```

Bloc procédural Exemple 3/3

```
...

  initial // ============== generation horloge
    begin
      ck = 0;
      #(p/4) forever #(p/2) ck = ~ck;
    end
  always @ (posedge ck) // ===== affichage echantillonne
    $display("ld=", load, "␣data_in=%b", data_in,"␣
        ser_data=", ser_data );
endmodule
```

III Le langage Verilog

La généricité en Verilog

Comme en VHDL l'utilisation de paramètres permet de faire des descriptions "génériques".

Deux approches de généricité :

- les directives parameter et defparameter. Permettent de propager des valeurs de paramètres dans la hiérarchie de composants.
- La directive define (comme en C) qui permet de définir des constantes dans un module.

Généricité

Exemple de registre à décalage générique :

```
// registre piso de N bits
module pisoN( clk, load, par_in, ser_out );
   parameter N=4;
   input clk, load;
   input [(N-1):0] par_in;
   output ser_out;
   reg [(N-1):0] contenu;
   always @ (posedge clk)
      contenu = (load)?(par_in):(contenu >> 1); //syntaxe C
      assign #3 ser_out = contenu[0];
endmodule
```

Généricité

Instanciation du registre précédent et propagation de paramètre :

```
defparam piso_inst.N = 16;
pisoN piso_inst ( ck, load, data_in, ser_data );
```

La directive defparameter a permis de fixer une valeur de N différente de la valeur par défaut.

Généricité

Utilisation de la directive define :

```
‘define BUS_WIDTH 16
reg [ ‘BUS_WIDTH − 1 : 0 ] System_Bus;
```

Il est possible de supprimer une définition via la directive undef

```
‘define BUS_WIDTH 16
    ...
    reg [ ‘BUS_WIDTH − 1 : 0 ] System_Bus;
  ...
‘undef BUS_WIDTH
```

Instructions de simulation

- Affichage
  - $display : Affiche la liste spécifiée lors de l'appel

```
$display("Id=", load, "␣data_in=%b", data_in,
"␣ser_data=", ser_data );
```

  - $monitor : Affiche la liste à la fin du "time step" dès qu'il y a un changement sur une des variables ".
- Contrôle de la simulation
  - $finish : Sort du simulateur
  - $stop : suspend la simulation
- Lecture d'un fichier ($readmemh) et bien d'autres ...