Introduction à VHDL

Tanguy Risset

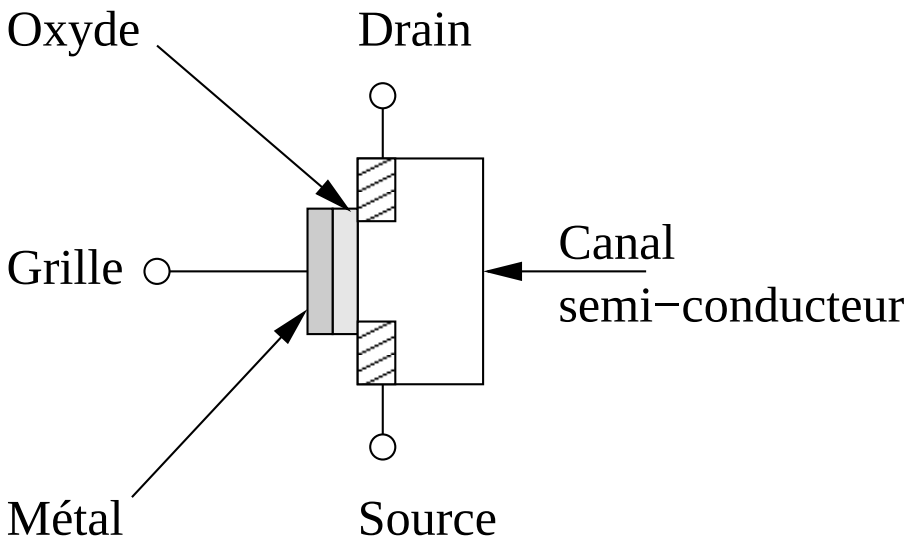
à Partir du cours de l'ENST `http://comelec.enst.fr/hdl`

Plan

- Rappel d’électronique
- Introduction à VHDL
  - Historique
  - Structure du langage
  - Types, expressions.
  - Exemples
- Codage en VHDL
  - Automate
  - Règles d’écriture
- Simulation VHDL

Le transistor

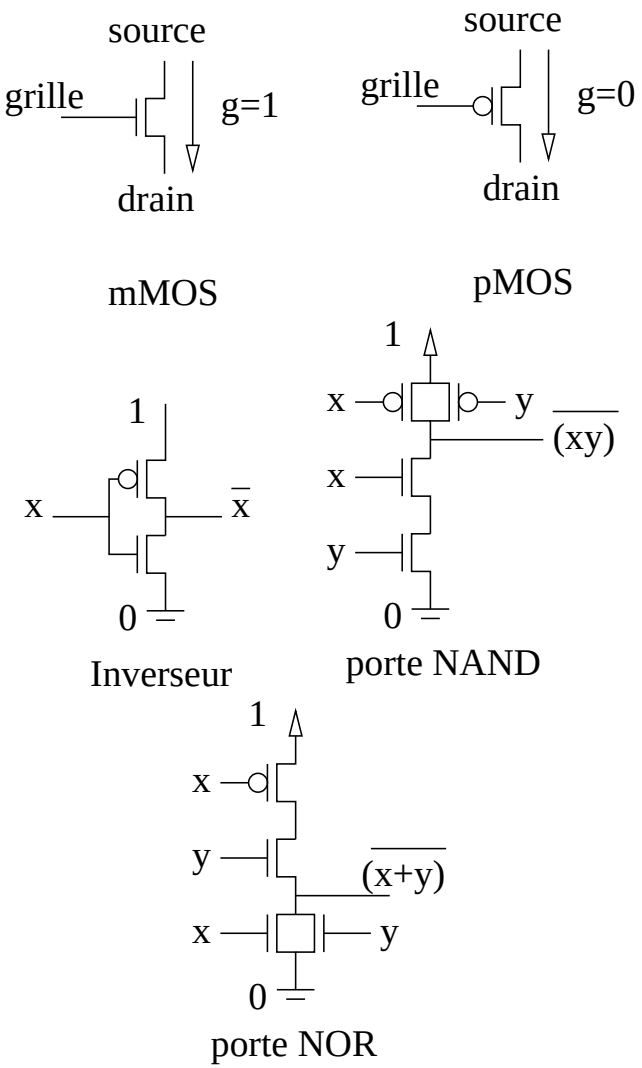
- Composant électronique de base
- Portes logiques ON/OFF



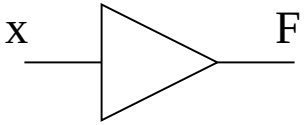
Technologie CMOS

- Complementary Metal Oxide Semiconductor
- Niveaux logiques : 0 = 0V et 1 = 3V
- Deux types de portes
  - nMOS : conducteur si la grille=1
  - pMOS : conducteur si la grille=0
- Réalisation de quelques portes de base

  Inverseur, NAND, NOR



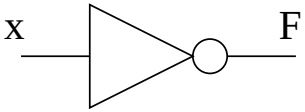
Portes élémentaires



Amplificateur:

$F = x$

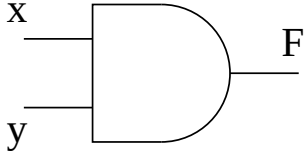
x	F
0	0
1	1



Inverseur:

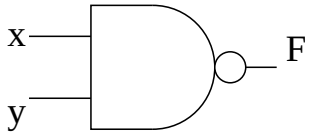
$F = \overline{x}$

x	F
0	1
1	0



ET: $F = x\ y$

x	y	F
0	0	0
0	1	0
1	0	0
1	1	1



NON ET:

$F = \overline{(x\ y)}$

x	y	F
0	0	1
0	1	1
1	0	1
1	1	0

Conception de circuits combinatoires

1. Description du problème:

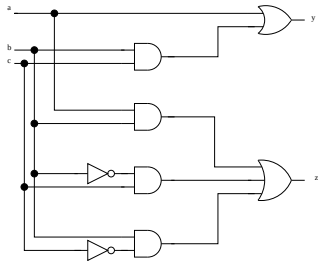
   y vaut 1 si a vaut 1 ou b et c valent 1.

   z vaut 1 si b ou c valent 1 (mais pas les deux) ou si a, b et c valent 1.
2. Table de vérité $\rightarrow$
3. Équations logiques

   $$y = \overline{a}bc + a\overline{b}\overline{c} + a\overline{b}c + ab\overline{c} + abc$$
   $$z = \overline{a}\overline{b}c + \overline{a}b\overline{c} + a\overline{b}c + ab\overline{c} + abc$$
4. Equations optimisées

   $$y = a + bc$$
   $$z = ab + \overline{b}c + b\overline{c}$$
5. Portes logiques $\rightarrow$

entrées			sorties	
a	b	c	y	z
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

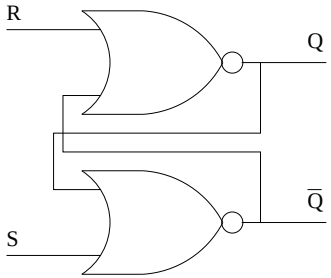


Composants combinatoires fréquents

- Multiplexeur à n entrées
- Décodeur $log(n) \rightarrow n$
- Additionneur n bits
- Comparateur n bits
- ALU n bits
- etc.

Logique séquentielle

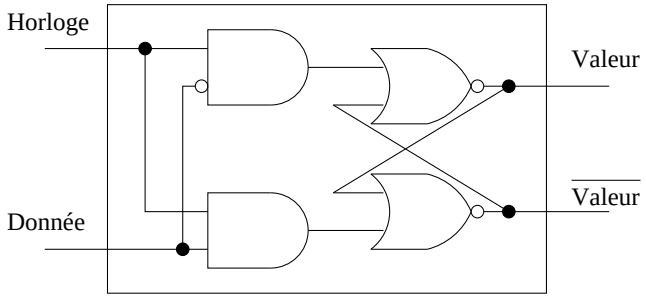
- Bascule RS: lorsque R et S passe à 0, Q et $\overline{Q}$ conservent leur valeur antérieure.



Bascule RS

S	R	Q	$\overline{Q}$
0	1	0	1
1	1	interdit	interdit
1	0	1	0
0	0	Q_{n-1}	$\overline{Q_{n-1}}$

- Bascule D: permet d'échantillonner la valeur de la donnée lorsque l'horloge passe à 1 (front montant de l'horloge) et de conserver cette valeur lorsque l'horloge passe à 0.



Composants séquentiels fréquents

- Registre n bits (avec $reset$ et eventuellement $load/ClockEnable$)
- Registre à décalage n bits
- Compteur n bits
- Machine à états

Conception de circuits séquentiels

- Extrêmement complexe en général.
- De nombreux modèles de calculs:
  - Séquentiel
    - Machine à états
    - Contrôleur + chemin de données
  - Parallélisme de tâches
    - Processus communicants
  - Parallélisme de données
  - Calcul sur des flots de données
  - Circuits multi-horloge
  - Paradigme synchrone
  - Circuits assynchrone
- Notion sous-jacente très utilisée: automate à états finit

Qu’est ce que VHDL?

- VHSIC (Very High Speed Integrated Circuit) Hardware Description Langage.
- Langage pour décrire la structure *et* le comportement de systèmes électroniques, en particulier des circuits digitaux (ASIC, FPGA, . . .).
- Standard IEEE.
- Indépendant de la technologie cible.
- Indépendant de la méthodologie de conception.
- Indépendant des outils de conception.
- Langage très général → très complexe (→ dépendent de tout!)
- VHDL **n’est pas un langage de programmation**
  c’est un langage de description (specification) de système.

Historique

- 1980: le département de défense américain lance un appel d'offre pour un langage qui permettrait de décrire tous les systèmes électroniques utilisés. Motivation affichée: réutilisabilité et réduction des coûts de conception.
- 1983 trois compagnies (Intermetics, IBM, Texas Instruments) commencent le développement.
- 1985: première version officielle de VHDL (version 7.2).
- 1986: VHDL est donné à IEEE pour en faire un standard.
- 1987: Standard IEEE 1076-1987.
- 1993: Standard IEEE 1076-1993.
- 1999: Standard IEEE 1076.6-1999

Convention lexicale

- Comme en C, un code vhdl est composé d'une suite d'éléments :
  - commentaires
  - délimiteurs
  - identificateurs,
  - expressions (terminées par un point-virgule),
  - mots-clefs
  - littéraux(constantes), par exemple:
    - 67 est un entier
    - '0' est un bit
    - "001", O"562", X"FF1" sont vecteurs de bits
    - "chaine" est une chaine de caractères
- VHDL est insensible à la casse. On écrit souvent les mots reservé du langage en majuscule, le reste en minuscule.
- Les commentaires commencent par deux tirets (- -)

Structure du langage

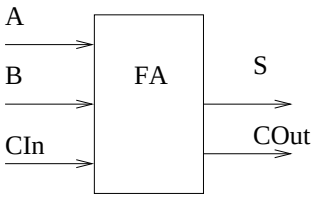
- Il existe "5 unités de compilation" permettant de décrire des composants.
  - L'**entité**: description de l'interface du composant: le nom et ses ports d'entrée/sortie
  - L'**architecture** décrit l'intérieur du composant. Il peut y avoir plusieurs architectures pour le même composant (ex: une pour la simulation efficace, une pour la synthèse). L'architecture contient les processus.
  - La **déclaration de paquetage**. Un paquetage est une collection d'objets réutilisables (constantes, types, composants, procédures)
  - Le **corps de paquetage**
  - La **configuration** indiquant quelle architecture utiliser pour chaque entité

Bibliothèques

- Les bibliothèques sont déclarée avec le mot clé `library`, elle dépendent des outils utilisés.
- Elle contiennent des *paquetages* que l'on déclare vouloir utiliser avec le mot clé `use`:
```
use BIBLIOTHEQUE.PAQUETAGE.all;
```
- La bibliothèque par défaut est `WORK`. `WORK` est aussi le nom symbolique de la bibliothèque dans laquelle sont stockés les résultats.
- La bibliothèque `STD` est une bibliothèque standard fournie avec le langage , elle contient des définitions des types et des fonctions de base (integer, bit,...).
- Par défaut, les outils considère que l'on utilise les bibliothèques `STD` et `WORK`, il y a donc implicitement:
```
library STD;
library WORK;
```
- En général, on utilise la librairie suivante IEEE qui définit le type `std_logic`:
```
library ieee;
```

L’entité

- L’entité est la description de l’interface du circuit . Elle correspond au symbole dans les représentations schématiques.




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

ENTITY FA IS
    port(A, B, Cin: in  STD_LOGIC;
         S, COut: out STD_LOGIC);
END ENTITY FA;
```

L’architecture

- L’architecture est la description interne du circuit.
- Elle est toujours associée à une entité, une entité peut avoir plusieurs architectures:

```
architecture arch1 of fa is
 signal resultat :
   STD_LOGIC_VECTOR(1 downto 0);
begin

 resultat <= (’0’ & a) +
     (’0’ & b) + (’0’ & cin);
 s <= resultat(0);
 cout <= resultat(1);

end arch1;
```

```
architecture arch2 of fa is
begin

 s <= a xor b xor cin;
 cout <= (a and b) or
       ((a xor b) and cin);

end arch2;
```

(& est l’opérateur de concaténation de vecteurs de bit)

Corps de l'architecture

- Dans le corps de l'architecture se trouvent les 3 types d'instructions concurrentes de VHDL
- Elles peuvent être écrites dans n'importe quel ordre :
  - Des processus avec le mot clé `process`
  - Des instructions concurrentes d'affectation de signaux (<=).
  - Des instanciation de composant avec le mot clé `port map`

```
begin
-- process:
p1 : process(SI, Cin)
 begin
S <= SI xor Cin;
 end process;

-- Aff. Signal
SI <= A xor B;

-- Instanciation
inst_MAJ : MAJ port map (
        X => A,
        Y => B,
        Z => Cin,
        M => Cout);
end arch3;
```

Signal et Variable

- Dans un process, on peut trouver des affectations de signaux ou de variables.
- Contrairement aux variables, l'affectation du signal n'a pas un effet immédiat.
- Ainsi, dans un process, après cinq instructions `A <= A+1;`, le signal `A` n'est pas augmenté de 5 mais seulement de 1.
- Il faut lire `A.futur <= A.présent +1;` de sorte que `A.présent` n'est jamais modifié.

Types

- Il existe des types:
  - scalaire (entier, réel, physique, énuméré),
  - composite (tableaux, enregistrement),
  - fichier et pointeur.
- Les types possède des *attibut de type*, Ils sont représentés de cette façon $< OBJET >' < ATTRIBUT >$
- Par exemple:
  - `type COULEUR is (BLEU, ROUGE, VERT);`
  - `COULEUR'left` renvoie BLEU
  - `COULEUR'right` renvoie VERT
  - `COULEUR'pos(BLEU)` renvoie 0
- Autre exemple:
  - `type MOT is STD_LOGIC_VECTOR(7 downto 0);`
  - `MOT'LEFT` renvoie 7;
  - `MOT'LENGTH` renvoie 8; `MOT'RIGHT renvoie 0;`

Type entier

- Types entier:
  - Le type entier integer prédéfini dans le paquetage standard STD permet de définir des nombres signés sur 32 bits entre -2^{31} et 2^{31}.
  - En général, on spécifie le *range* d'un entier pour qu'il soit stocké sur un nombre de bit adéquat:
    `signal my_natural is integer range 0 to 256`

Operateurs

Logiques	`and, or, nand, nor, xor, not`	
Relationnels	`=, /=, < , <=, > , <=`	
Arithmétique	`*, / , mod, rem`	(A rem B) a le signe de A (A mod B) a le signe de B
Divers	`**, abs, &`	** : exponentiation abs : valeur absolue & : concaténation

Process

- Les différents `process` d'un programme vhdl s'exécutent en parallèle les uns des autres.
- Un processus peut avoir des variables locales. Le fonctionnement du processus est régi par les règles suivantes :
  - Un processus est une boucle infinie , lorsqu'il arrive à la fin du code, il reprend automatiquement au début
  - Un processus doit être sensible des points d'arrêt, il existe 2 types de points d'arrêts :
    - Le processus est associé à une "liste de sensibilité" qui réveille le processus lors d'un changement d'un des signaux.
    - Le processus a des instructions d'arrêt wait dans sa description interne.
- Les variables sont internes au processus et sont affectées immédiatement, contrairement aux signaux qui eux ne sont pas affectés directement mais en fin de processus
- Un processus est *séquentiel* dans le sens ou les instructions sont évaluées l'une après l'autre dans l'ordre d'écriture.

Exemple: AND3


```
entity AND_3 is
  port (
     e1 : in bit;
     e2 : in bit;
     e3 : in bit;
     s  : out bit
  );
end entity

architecture ARCH of AND_3 is

begin -- ARCH

   s <= e1 and e2 and e3;

end ARCH
```

Exemple: Full Adder


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

entity full_add1 is
  port (
    a, b, cin : in std_logic;
    s, cout   : out std_logic;
  );
end entity;

architecture ARCH of full_add1 is
  signal resultat : unsigned (1 downto 0);
begin

  result <= ('0' & a) + ('0' & b) + ('0' & c);
  s      <= resultat(0);
  cout   <= resultat(1);
```

Exemple: Additionneur 4 bits


```
[..librairies ..]
entity adder4 is
  port (
    a, b : in unigned(3 downto 0);
    s    : out unigned(3 downto 0);
    cout : out std_logic
  );
end entity;

architecture ARCH of adder4 is
  signal c : unsigned (2 downto 0);
  Component full_add1 is
    port (
      a, b, cin : in std_logic;
      s, cout   : out std_logic;
    );
end entity;
```

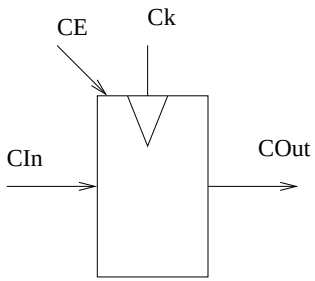
Exemple: Additionneur 4 bits


```
[... suite et fin ...]
architecture ARCH of adder4 is
  signal c : unsigned (2 downto 0);
  Component full_add1 is
    port (
      a, b, cin : in std_logic;
      s, cout   : out std_logic;
    );
end entity;

begin
inst_FA_1 : full_add1 port map(a(0),b(0),'0',s(0),c(0));
inst_FA_2 : full_add1 port map(a(1),b(1),c(0),s(1),c(1));
inst_FA_3 : full_add1 port map(a(2),b(2),c(1),s(2),c(2));
inst_FA_2 : full_add1 port map(a(3),b(3),c(2),s(3),c(3));
end ARCH;
```

Exemple: un registre

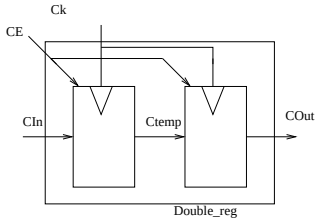
- Un registre avec clock enable.
- Le processus est réveillé lors d'une transition sur `Ck`
- Les `If` internes permettent d'exprimer la condition: front montant de l'horloge avec `CE` à 1.

```
PROCESS(ck)
BEGIN
IF (ck = '1' AND ck'EVENT)
THEN
    IF CE='1' THEN Out <= In;
    END IF;
END IF;
END PROCESS;
```



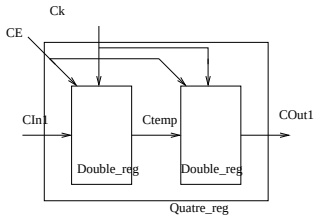
Exemple: double registre (declaration)


```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY DOUBLE_REG IS
    port(CIn, Ck, CE: IN STD_LOGIC;
         COut: out STD_LOGIC);
END ENTITY DOUBLE_REG;

ARCHITECTURE behavioural OF DOUBLE_REG IS
    -- Declaration de signaux du composant

BEGIN
    PROCESS(ck)
       Variable CTemp: STD_LOGIC;
        -- Declaration de signaux
        -- ou variables du process
    BEGIN
    IF (ck = '1' AND ck'EVENT)
    THEN
        IF CE='1' THEN
            COut <= CTemp;
            CTemp := CIn;
        END IF;
    END IF;
    END PROCESS;
END behavioural;
```

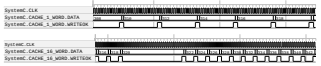


Exemple: double registre (instanciation)


```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY QUATRE_REG IS
    port(CIn1, Ck, CE: IN STD_LOGIC;
         COut1: out STD_LOGIC);
END ENTITY QUATRE_REG;

ARCHITECTURE behavioural OF QUATRE_REG IS
       Signal CTemp: STD_LOGIC;
       COMPONENT DOUBLE_REG IS
            port(CIn, Ck, CE: IN STD_LOGIC;
            COut: out STD_LOGIC);
        END COMPONENT;
    Begin
    I0: DOUBLE_REG port map (Cin => Cin1,
                             Ck  => Ck,
                             CE  => CE,
                             Cout => CTemp);
    I1: DOUBLE_REG port map (Cin => CTemp,
                             Ck  => Ck,
                             CE  => CE,
                             Cout => COut1);

END behavioural;
```



Conception en VHDL

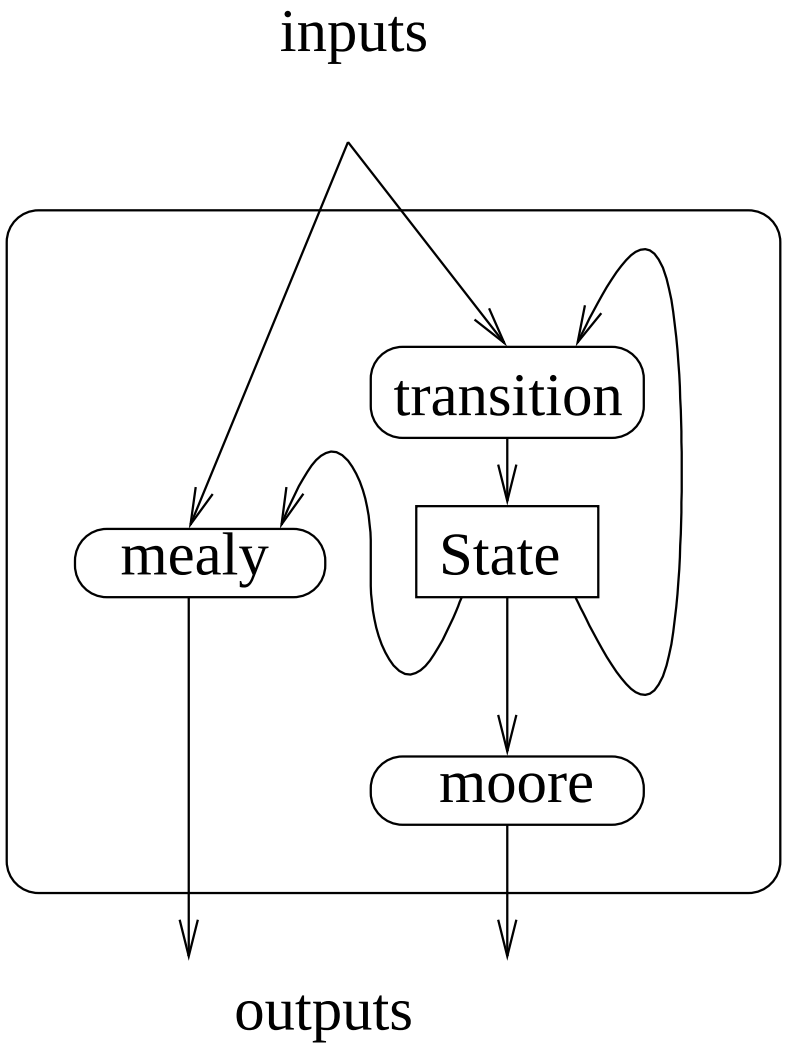
- Le mécanisme du temps symbolique rend extrêmement délicat le débugage de VHDL
- 95% du temps de développement se fait en simulation.
- On utilise les *waveforms* qui ne permettent pas de voir les transitions de granularité Δ. 
- VHDL permet énormément de constructions difficiles à comprendre (circuits asynchrones, simulation, synthèse, etc...).
- Aujourd'hui il existe des langages plus rapides pour la simulation (SystemC, etc.), VHDL est donc essentiellement utilisé pour la synthèse et la simulation bas niveau.
- $\Rightarrow$ **Se contraindre fortement lors de l'écriture de programmes VHDL**

Contraintes pour l'écriture de VHDL

- Penser *composant* (objet) plutôt que fonctionnalité (procédure).
- Distinguer les éléments de mémorisation (mémoire, bancs de registre) et les composants standards (opérations arithmétiques, filtres numériques) pour prévoir l'utilisation de bibliothèques.
- Décomposer les composants sous forme d'automate à états finis
- On peut alors soit coder directement l'automate en VHDL soit décomposer à nouveau cet automate en un contrôleur et un chemin de donnée
- Pour les traitements hautement pipelinés (*stream processing*), on utilisera plutôt une description des différents étages de pipeline interconnectés.

Notion d'automate

- Un automate est systématiquement décomposé en:
- Un état (ensemble de registre)
- Trois fonctions:
  - Transition: sensible aux entrée et à l'état, met à jour l'état.
  - Moore: sensible à l'état, met à jour des sorties
  - Mealy: sensible aux entrée et à l'état, met à jour des sorties



Codage d’un automate en VHDL (exemple)

- N’utiliser que des signaux (pas de variables). Que des types simples (`STD_LOGIC_*`), de préférence non resolu.
- Décomposer le matériel en l’état (tous les éléments mémorisés entre deux cycle d’horloge) et les autres signaux.
- Écrire un processus sensible à l’horloge qui met à jour les registres d’état
- C’est la fonction de **transition** de l’automate.

```
// Gestion du registre d’état avec reset synchrone
PROCESS(clk)
BEGIN
IF rising_edge(clk) THEN
  IF reset = ’1’ THEN
        state   <= INIT;
  ELSE  state   <= state_next;
END IF;
END PROCESS;

//Fonction de transition
PROCESS (state, input1, input2)
BEGIN
CASE state IS
    WHEN IDLE =>
        IF  i1 = ’1’
                state_next <= WRITE;
        ELSE
                state_next <= IDLE;
        END IF;

    WHEN  READ =>
        state_next <= IDLE;

    WHEN  OTHER =>
        state_next <= IDLE;
END CASE;
```

Codage d’un automate en VHDL (suite)

- Pour la fonction de **Moore** de l’automate (c’est similaire pour la fonction de Mealy): Écrire un processus sensible à l’état qui calcule les sorties.
- La manière la plus propre:
  Décrire le data-path comme des affectations entre signaux en dehors des processus (inférence systématique du matériel pour le calcul sur les données)
  Décrire l’affectation des sorties comme une simple affectation entre signaux.

```
[...]
//Equation concurrente (data-flow)
S1 <= a + b * C;
S2 <= a - b;

// Fonction de Moore
PROCESS (STATE)
BEGIN
CASE state IS
   WHEN IDLE =>
        out1 <= S1;
        out2 <= S2;

   WHEN READ =>
        out1 <= S1;
        out2 <= S1;
   WHEN OTHER =>
        out1 <= S2;
        out2 <= S2;
END CASE;
END;
```

Synthèse de VHDL

- La plaie: l'inférence accidentelle de registres.
- dès qu'un signal n'est pas affecté dans tous les chemin de contrôle.

```
PROCESS(S1,state)
    BEGIN
        IF state = 'INIT' THEN
           Out1 <= S1;
        END IF;
    END;
```

```
PROCESS(S1,state)
    BEGIN
        IF state = 'INIT' THEN
           Out1 <= S1;
        ELSE
           Out1 <= '0';
        END IF;
    END;
```

$\Rightarrow$ Compter les registres

- Boucle à contrôle constant acceptée (déroulées à la compilation).
- Clause `Wait` acceptée uniquement pour l'horloge.
- En principe, il existe un standard IEEE pour le VHDL synthétisable. En pratique, chaque outil synthetise des sous-ensembles légèrement différents.
- Principaux outils de synthèse pour les Asics: Mentor Graphics, Synopsys, Cadence.

niveaux description VHDL

- Description structurelle.
  - Proche des schématiques traditionnelles des concepteurs
  - Blocs inter-connectés.
- Description comportementale.
  - Proche de la programmation traditionnelle
  - exécution séquentielle d'instructions
- Cette classification ne correspond pas exactement à notre décomposition intuitive: description algorithmique/description architecturale

VHDL de niveau transfert de registre

- Une description de niveau transfert de registre est une description de la structure du circuit, mais qui abstrait les opérateurs.
- On peut parler de "registre" (sans dire exactement quel type), d'additionneur, etc.
- N'est pas forcément du VHDL structurel.
- Ce sous ensemble de VHDL doit pouvoir être synthétisé par les outils commerciaux (Synopsis, Cadence, etc.) selon une *sémantique définie par le standard*.

$\longrightarrow$ C'est une plate forme solide pour la synthèse haut niveau.

Principe de la simulation événementielle

- Le système est représenté comme un ensemble de **processus** (*process*) qui s'exécutent en parallèle
- On veut simuler des processus parallèles sur une machine séquentielle.
- Nouvel objet pour communiquer entre programmes séquentiels: le signal
- Moteur de simulation:
  - Le simulateur exécute tous les processus dans un ordre quelconque.
  - Lorsqu'un signal partagé par plusieurs processus est modifié, on enregistre sa nouvelle valeur, le signal conservant temporairement sa valeur.
  - Lorsque tous les processus ont été exécutés, on modifie les valeurs des signaux partagés
  - on incrément le temps symbolique ($+1\Delta$) et on recommence jusqu'à convergence

Le temps symbolique Δ

- Le temps symbolique permet d'ordonner des événements simultanés à partir de leurs dépendances.
- Une affectation à un signal: $Sig_1 <= Sig_2$ est instantanée mais la valeur de Sig_1 est modifiée après 1 Δ
- Un "événement" possède donc une date complète composée d'une date physique (ex: 1h 04m 17s) et d'une date symbolique (ex: 4Δ)

Signal et variable

- Du fait de sa sémantique spécifique, le signal se comporte très différemment d'une variable
- On peut en avoir une vision intuitive comme representant la "valeur d'un fil physique au cours du temps"
- Pour qu'un signal conserve une valeur d'un cycle à un autre il faut mettre en place explicitement un mécanisme de mémorisation qui sera interprété comme un registre.
- Pour faciliter la simulation on peut aussi introduire des variables dans les processus.
- Les variables sont locales aux processus, leur affectation est instantanée, elles conservent leur valeur au cours du temps comme dans un langage de programmation

$$Sig_1 <= Sig_2;$$
$$Sig_3 <= 3;$$

$$Var_1 := Var_2;$$
$$Var_3 := 3;$$

Le temps physique

- Le temps physique permet de simuler l'écoulement du temps réel
- Il ne peut avancer qu'avec les instruction `wait` et `after` (points de synchronisation, point d'arrêt):

```
wait 10 ns
Sig1 <= 25 after 100 s

wait until rising_edge(Clk)
```

- Entre deux points de synchronisation, le temps physique n'avance pas (l'affectation des signaux est différée).
- Le processus est (implicitement) une boucle infinie avec au moins un point d'arrêt.

```
process ...
  ...
  begin
  S <= A+B;
  wait on A;
  R:=S;
  A<= R;
  B<= A;
  wait until rising_edge(Clk)
  if (R > 1023) then
    Counter <= 127
  else
    COUNTER <=Counter-1;
  end if;
  wait until rising_edge(Clk)
  ISO <=COUNTER * 7
end process;
```

Principe du moteur de simulation

- Chaque processus possède une liste de sensibilité qui indique si il doit être "réveillé" ou pas.
  Changement d'une valeur d'un signal utilisé
  Changement du temps physique
- Le moteur effectue de manière répétitive les tâches suivantes:
  1. Choisit la date courante
  2. Positionne les signaux à leurs nouvelles valeurs
  3. Pour chaque processus
     teste la condition de reprise du point d'arrêt
     Si elle est vérifiée, le processus est réveillé et exécuté jusqu'au prochain point d'arret

Compilation de Vhdl

- Pour être simulé, un programme VHDL est
  - Compilé
  - "Élaboré" ($\simeq$édition de liens). Lors de l'élaboration les opérations élémentaires utilisées sont implémentées par des librairies fournies par l'outil de simulation, leur implémentation peut poser problème.
  - Simulé. Pour cela il nécessite un "test bench" qui fournit les entrées et l'horloge au composant simulé (stimuli).
- Si il est écrit de manière *synthétisable*, il peut être:
  - synthétisé (synthèse logique)
  - simulé après synthèse
  - placé (placement-routage)
  - simulé après placement routage.
- Une description destinée à la simulation efficace d'un circuit est très différente d'une description destinée à la réalisation. VHDL synthétisable propose un compromis entre les deux.