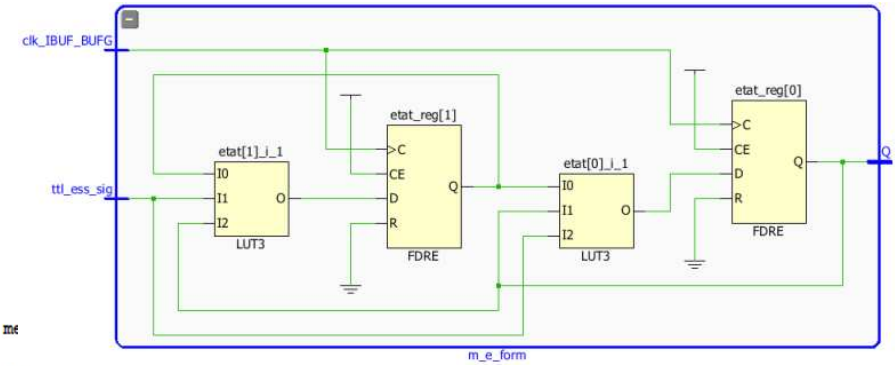
Le langage VHDL

1.1. Présentation

- Vhsic Hardware Description Langage (Vhsic = Very High Speed Integrated Circuit) est un langage de description de matériel et non un langage software comme le C.
- Standardisé en 1987 par l ’IEEE (Institute Of Electrical and Electronics Engineers) sous la référence IEEE 1076-87. Une mise à jour importante a été faite en 1993 : IEEE 1076-93 est supportée par tous les outils.
- Utilisé au début pour la modélisation et la simulation avant d’être adopté pour la synthèse logique. Toute la syntaxe n’est pas synthétisable !!!
- Permet de décrire un système avec un niveau d’abstraction élevé « algorithmique » ou un niveau proche du matériel « gate level ». Entre les deux se trouve le niveau RTL « Register Transfert Level » qui décrit le système sous forme de machine d’états.
- C’est le niveau RTL qui est utilisé pour la synthèse car il est moins lourd que le niveau « gate level » et il est indépendant de la cible. Le niveau « algorithmique » n’est pas forcément synthétisable.

1.2. La synthèse logique

```
process(clk)
begin
    if rising_edge(clk) then
        etat <= etat_suiv;
    end if;
end process;

process(etat, ttl)
begin
    case etat is
            when debut =>
                        ttl_mef <= '0';
                        if ttl='1' then
                            etat_suiv <= me
                        else
                            etat_suiv <= debut;
                        end if;
            when mef =>
                        ttl_mef <= '1';
                        etat_suiv <= fin;
            when fin =>
                        ttl_mef <= '0';
                        if ttl='0' then
                            etat_suiv <= debut;
                        else
                            etat_suiv <= fin;
                        end if;
    end case;
end process;
```



- La synthèse logique est l'opération qui consiste à traduire le code VHDL en fonctions logiques et bascules prêtes à être connectées dans le silicium.

1.3.1. Simulation / synthèse

- On rencontre deux langages de description de matériel : VHDL (populaire en Europe, proche de Ada « pgm objet ») et Verilog (populaire aux US, proche du C).
- Le langage VHDL permet de :
  1. Modéliser des circuits pour la simulation
  2. Décrire des applications pour circuits ASIC ou programmables (FPGA)

Modélisation pour la simulation

Norme IEEE 1076

La totalité de la norme peut être utilisée pour la modélisation

```
X <= '1' after 100 ns;
Wait for 200 ns;
Signal ent_s : std_logic := '0'
```

Description de système matériel

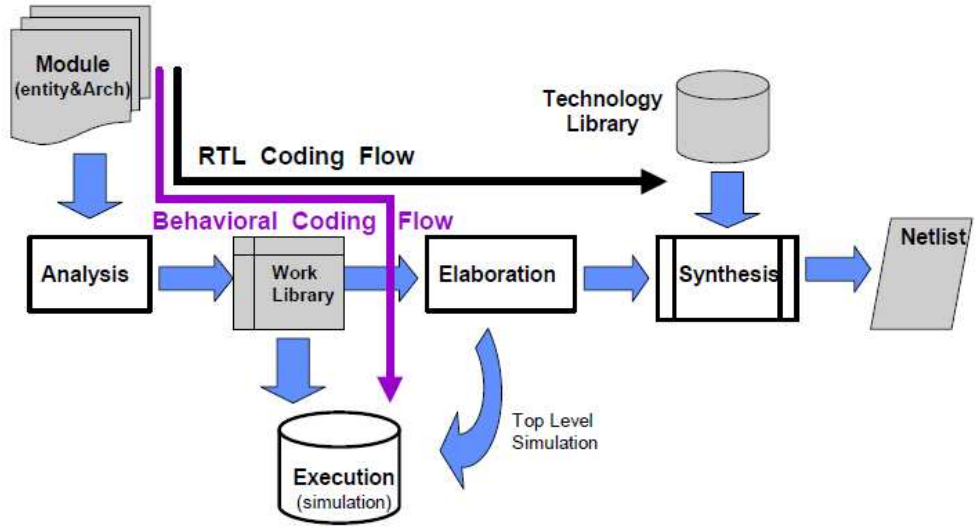
Norme IEEE 1076

Une partie seulement peut être utilisée pour la synthèse

Les instructions de gauche ne peuvent pas être utilisées pour la synthèse

1.3.2. Behavioral / RTL

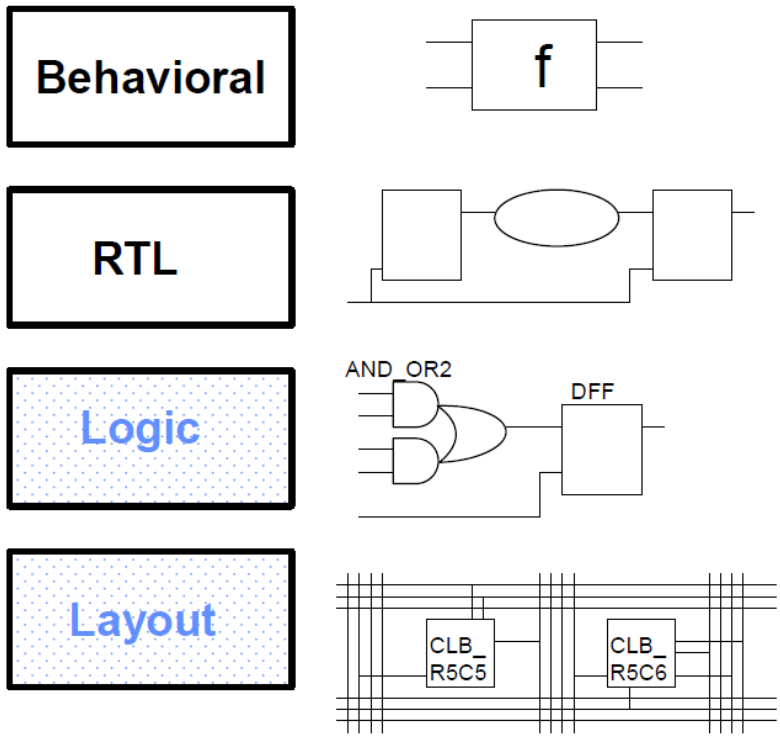
- Une description comportementale « behavioral » décrit un algorithme sans rentrer dans les détails technologiques.
- Une description « RTL » donne les détails sur la connexion des registres avec la logique combinatoire.



1.4. Niveaux d'abstraction

Peu de détails, temps de développement court

Détails technologiques, description et simulation fastidieuses



2.1. Premier exemple

```
-----------------------------------------
-- Voici un exemple simple
-----------------------------------------

--  La déclaration des bibliothèques

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-----------------------------------------
--        L'entité

entity portes is
    Port ( a : in std_logic;
           b : in std_logic;
           c : in std_logic;
           x : out std_logic;
           y : out std_logic;
           z : out std_logic);
end portes;
-----------------------------------------
--        L'architecture

architecture Behavioral of portes is
begin
     x <= a or b ;
     y <= b nand (not c) ;
     z <= a xor c ;
end Behavioral;
```

commentaire

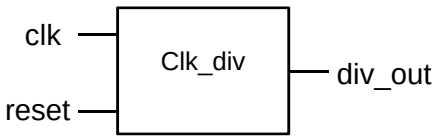
Appel librairies

entité

architecture

2.2. Deuxième exemple

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
--
--    commentaire
--
entity clk_div is
     generic (N : natural := 5);
     Port ( reset : in std_logic;
            clk : in std_logic;
            div_out : out std_logic);
end clk_div;

architecture Behavioral of clk_div is
     signal count_temp : std_logic_vector (N downto 0);

begin      -- autre commentaire

process (reset, clk)

begin
     if reset='0' then
          count_temp <= (others => '0');
     elsif (clk'event and clk='1') then
          count_temp <= count_temp + 1;
     end if;

end process;

div_out <= count_temp(N);

end Behavioral;
```



La nouveauté dans cet exemple est le bloc « process ».
Ce bloc est très pratique pour décrire des unités cadencées par une horloge.
On peut écrire plusieurs process, ils s'exécutent tous en parallèle

2.3. Trois règles de base

```
----------------------------------------
-- Programme de démonstration
----------------------------------------

library IEEE; -- librairie IEEE
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;

entity majuscule is
    Port ( a : in  STD_LOGIC;
           b : in  STD_LOGIC;
           c : in  STD_LOGIC;
           x : out  STD_LOGIC;
           y : out  STD_LOGIC;
           z : out  STD_LOGIC);
end majuscule;

architecture Behavioral of majuscule is
   signal INTER : std_logic;

begin
   inter <= a or b;
   y <= not INTER;
   z <= a xor c;
end Behavioral;
```

Les commentaires commencent par un double tiret « -- » et se terminent à la fin de la ligne.

VHDL ne distingue pas les majuscules des minuscules :

inter et INTER désignent le même signal

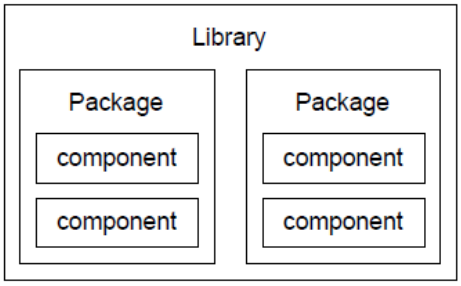
Une instruction se termine par un « ; »
les espaces ne sont pas significatifs

2.4. La librairie IEEE

- Le standard « IEEE.1164 » définit une librairie avec un certain nombre de packages dont certains sont indispensables pour tout programme VHDL.
- Les packages doivent être appelés avant la déclaration de l'entité.

```
library IEEE; -- librairie IEEE
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;
```

- L'extension « .all » du nom de la librairie signifie qu'on intègre tout le contenu.



VHDL	C
analyze	compile
elaborate	link
component	function
instantiate	call
use	#include
package	DLL
library	directory

2.4.1. Le package std_logic_1164

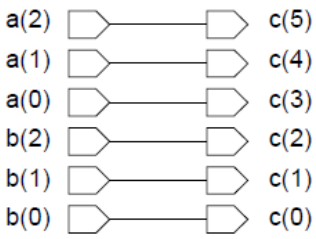
- C'est ici qu'est défini le type « std_logic » qui est une extension du type « bit »
  - 'U': uninitialized. This signal hasn't been set yet.
  - 'X': unknown. Impossible to determine this value/result.
  - '0': logic 0
  - '1': logic 1
  - 'Z': High Impedance
  - 'W': Weak signal, can't tell if it should be 0 or 1.
  - 'L': Weak signal that should probably go to 0
  - 'H': Weak signal that should probably go to 1
  - '-': Don't care.
- Les opérations applicables à ce type sont : `and,nand,or,nor,xor,xnor,not`
- Le type « std_logic_vector » est un tableau de « std_logic »

```
architecture Behavioral of portes is
   signal s1, s2, s3 : std_logic_vector(3 downto 0);
begin
   s1(0) <= '1';
   s1(1) <= '0';
   s1(2) <= '0';
   s1(3) <= '1';
   s2 <= "1100";
   s3 <= s1 and s2;
```

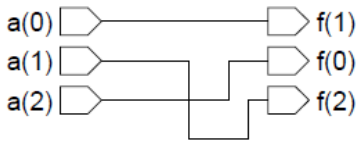

Agrégats, concaténation

```
a : in  std_logic_vector(2 downto 0);
b : in  std_logic_vector(2 downto 0);
c : out std_logic_vector(5 downto 0);
d : out std_logic_vector(5 downto 0);
e : out std_logic_vector(5 downto 0);
f : out std_logic_vector(2 downto 0);
```

```
c <= a & b;
```



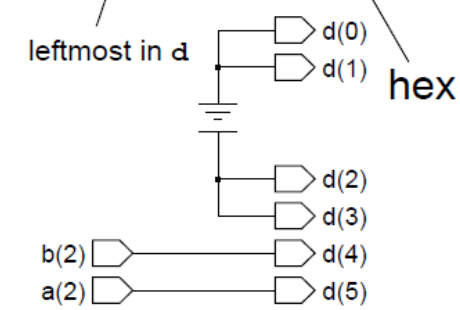

```
(f(0), f(2), f(1)) <= a;
```

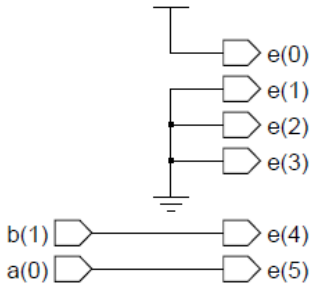



```
d <= a(2) & b(2) & X"C";
```

leftmost in d

hex





leftmost in e index

```
e <= (a(0), b(1), 0 => '1', others => '0');
```

La fonction « rising_edge »

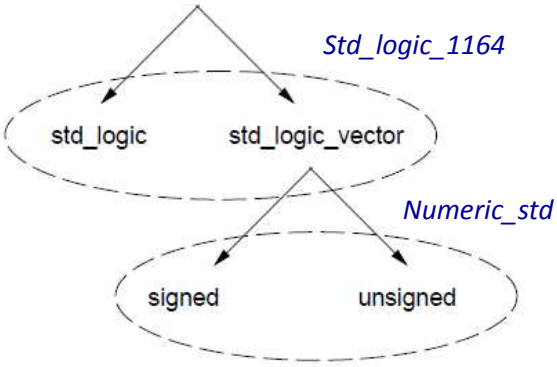
- Cette fonction, définie dans « std_logic_1164 », est très utile pour détecter les fronts montants d'une horloge.
- Elle vérifie bien que le signal part de '0' avant de passer à '1'.
- Il existe une fonction similaire qui teste les fronts descendants : « falling_edge »

```
process ( Clk, Reset )
begin
 if Reset = '1' then
   Q <= '0' ;
   elsif   rising_edge (Clk) then
   Q <= D ;
 end if ;
end process ;
```

```
package std_logic_1164 is
...
function rising_edge ( signal Clk : std_logic ) return boolean;
begin
  if ( Clk'event and Clk= '1' and Clk'last_ value = '0' ) then
    return true ;
      else
    return false ;
  end if ;
end rising edge ;
...
```


2.4.2. Le package numeric_std

- C'est ici que sont définis certains types pour représenter les entiers ainsi que les opérations arithmétiques.



- The unsigned type
- The signed type
- The arithmetic functions: +, –, *
- The comparison functions: <, <=, >, >=, =, /=
- The shift functions: shl, shr
- The conv_integer function
- The conv_unsigned function
- The conv_signed function
- The conv_std_logic_vector function

- On y trouve aussi quelques fonctions de conversion de type comme celle-ci :

```
conv_std_logic_vector(arg: unsigned, size: integer) return std_logic_vector;
```

```
architecture Behavioral of portes is
   signal b, c : std_logic_vector(3 downto 0);
   signal d : std_ulogic_vector(3 downto 0);

begin
   c <= b and d;                              -- FAUX
   c <= b and conv_std_logic_vector(d,4); -- BON
```

std_logic_unsigned / std_logic_signed

- Ces deux packages sont des extensions du package « numeric_std».
- Il ne faut appeler qu'un seul des deux à la fois !!!

std_logic_unsigned :

- Dans ce package, les fonctions sont redéfinies pour traiter les nombres de type « std_logic_vector » comme des entiers non signés.

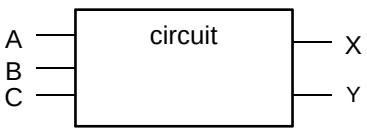
std_logic_signed :

- Dans ce package, les fonctions sont redéfinies pour traiter les nombres de type « std_logic_vector » comme des entiers signés.
- Le complément à 2 est utilisée pour le représentation des nombres négatifs.

```
library IEEE; -- librairie IEEE
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;
-- or IEEE.std_logic_signed.all;
```


2.5. Le bloc « Entity »

```
entity circuit is
    Port     ( A : in  STD_LOGIC;
               B : in  STD_LOGIC;
               C : in  STD_LOGIC;
               X : out  STD_LOGIC;
               Y : out  STD_LOGIC);
end circuit;
```



- L'entité donne une vue externe du circuit.
- Les signaux d'interface constituent dans la terminologie VHDL un « port ».
- Chaque signal doit posséder : un nom (choisi par l'utilisateur), un mode (in, out, inout) et un type (std_logic, std_logic_vector, integer, boolean, …).
- On peut déclarer un paramètre générique avec une valeur par défaut, si une architecture appelle cette entité elle pourra changer la valeur de N.

```
entity clk_div is
   generic (N : natural := 5);

   Port ( clk : in  STD_LOGIC;
          reset : in  STD_LOGIC;
          div_out : out  STD_LOGIC);
end portes;
```

2.6. Le choix des noms

- Chaque élément manipulé par VHDL (signal, constante, bus, …) doit porter un nom. Celui-ci doit respecter les règles suivantes :

  1. Caractères admis : les 26 lettres de l'alphabet, les 10 chiffres décimaux et le caractère '_'.
  2. Le 1er caractère doit être une lettre.
  3. Le caractère '_' ne doit pas terminer un nom.
  4. Un nom ne doit pas être un mot réservé.
  5. La longueur d'un nom ne doit pas dépasser une ligne.

```
architecture Behavioral of portes is
   signal inter : std_logic;
   constant indice : integer := 12;
   signal sortie : std_ulogic_vector(3 downto 0);
```

Il faut respecter une règle simple : un nom doit permettre de deviner le type d'information représentée

2.7. Le bloc « Architecture »

```
architecture Behavioral of portes is

begin
   x <= a or b;
   y <= b and (not c);
   z <= a xor c;

end Behavioral;
```

- Le bloc architecture décrit le système matériel à concevoir, cela peut être un système combinatoire ou séquentiel, simple ou complexe.
- Il existe trois façons de décrire un circuit électronique en VHDL :

  1. Description de bas niveau : on écrit des équations logiques
  2. Description modulaire : il s'agit d'associer des blocs existants
  3. Description comportementale : on décrit le comportement du circuit

2.8. Déclarations / instructions

```
architecture Behavioral of controle is
    type etat_type is (zero, un, deux, trois, quatre, cinq);
    signal etat : etat_type := zero;
    signal etat_suiv : etat_type ;
    signal compteur : std_logic_vector(26 downto 0) :=(others => '0');

begin
process(clk)
begin
    if rising_edge(clk) then
        etat <= etat_suiv;
        if (etat = un) then
          compteur <= compteur + 1;
        end if;
        if (etat = zero) then
          compteur <= (others => '0');
        end if;
    end if;
end process;

process(etat, compteur)
begin
    case etat is
            when zero =>
                        valid <= '0';
                        verr <= '0';
                        raz <= '0';
                            etat_suiv <= un;
            when un =>
```

Avant « begin » déclaration :

- Types
- Signaux
- Variables
- composants

Après« begin » code :

- affectations
- process

3.1. Syntaxe hors « process »

- Affectation inconditionnelle

```
Y <= A or B ;
```

- Affectation inconditionnelle

```
Y <= '1' when (SEL = "101") else '0' ;
```

```
Y <= A when sel="00" else
     B when sel="01" else
     C when sel="10" else
     '0';
```

- Affectation sélective

```
with SEL select
   Y <= A when "00",
        B when "01",
        C when "10",
        D when others;
```

3.2. Syntaxe dans « process »

- Affectation inconditionnelle

```
Y <= A or B ;
```

- Affectation inconditionnelle

```
if (SEL = "101") then
   Y <= '1';
else
   Y <= '0';
end if;
```

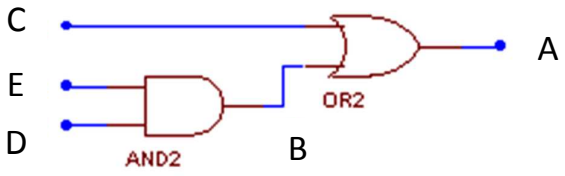
- Affectation sélective

```
case sel is
   when "00" =>   y <= '1';
   when "01" =>   y <= '0';
   when "10" =>   y <= '0';
   when others => y <= '1';
end case;
```


3.3. Fonctionnement concurrent

- En dehors du PROCESS, toutes les instructions s'exécutent en parallèle. On parle de fonctionnement concurrent, c'est le principe des systèmes combinatoires.
- L'ordre des instructions n'a aucune importance.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity concurrent is
    Port ( C : in std_logic;
           D : in std_logic;
           E : in std_logic;
           A : out std_logic);
end concurrent;

architecture Behavioral of concurrent is
   signal B : std_logic;
begin
   A <= B OR C ;  -- solution
   B <= D AND E ; --  n° 1

-- B <= D AND E ; -- solution
-- A <= B OR C ;  --  n° 2
end Behavioral;
```



Ces deux solutions donnent le même résultat après compilation, c'est-à-dire le schéma ci-dessus.

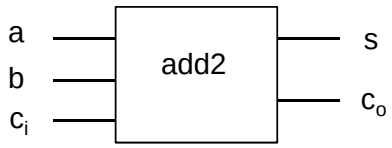
3.4. Fonctionnement séquentiel

- Dans un « process », les instructions s 'exécutent de façon séquentielle.
- Le « process » s 'exécute à chaque changement d 'état d 'un des signaux de la liste de sensibilité.
- La mise à jour des variables se fait au fur et à mesure que les instructions se déroulent.
- La mise à jour des signaux se fait à la fin du « process », après le « end ».
- Tous les « process » se déroulent en parallèle.
- L'ordre d'écriture des process n'a aucune importance.

```
process (clk)
begin

   if rising_edge (clk) then
      if (reset = '1') then
         count_temp <= (others => '0');
      else
         count_temp <= count_temp + 1;
      end if;
   end if;
end process;
```

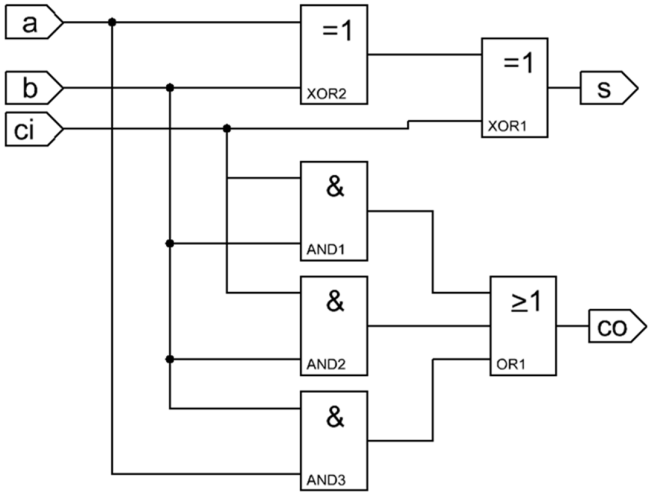
Le signal « count_temp » ne sera mis à jour qu'à la fin du « process », à la lecture du « end process »

4.1. Additionneur (1ère façon)



c_i	b	a	c_o	s
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

```
entity add2 is
    Port ( a : in  STD_LOGIC;
           b : in  STD_LOGIC;
           cin : in  STD_LOGIC;
           s : out  STD_LOGIC;
           co : out  STD_LOGIC);
end add2;

architecture Behavioral of add2 is

   signal p, g : std_logic;

begin
   p <= a xor b;
   g <= a and b;

   s <= p xor cin;
   co <= g or (p and cin);

end Behavioral;
```



4.2. Additionneur (2ème façon)

```
architecture Behavioral of add is
   signal co_s : std_logic_vector (1 downto 0);
   signal ci_b_a : std_logic_vector (2 downto 0);
begin

   ci_b_a <= ci & b & a;
   co <= co_s (1);
   s <= co_s (0);

   co_s <= "00" when ci_b_a = "000" else
           "01" when ci_b_a = "001" else
           "01" when ci_b_a = "010" else
           "10" when ci_b_a = "011" else
           "01" when ci_b_a = "100" else
           "10" when ci_b_a = "101" else
           "10" when ci_b_a = "110" else
           "11" ;

end Behavioral;
```

c_i	b	a	c_o	s
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

On écrit la table de vérité des sorties de l'additionneur, l'outil de synthèse trouvera la logique nécessaire

4.3. Additionneur (3ème façon)

```
architecture Behavioral of add2 is
begin
   with ci & b & a select
      s <= '0' when "000",
           '1' when "001",
           '1' when "010",
           '0' when "011",
           '1' when "100",
           '0' when "101",
           '0' when "110",
           '1' when others;

   with ci & b & a select
     co <= '0' when "000",
           '0' when "001",
           '0' when "010",
           '1' when "011",
           '0' when "100",
           '1' when "101",
           '1' when "110",
           '1' when others;

end Behavioral;
```

c_i	b	a	c_o	s
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

Autre façon d'écrire les tables de vérité des sorties, une table par sortie cette fois.

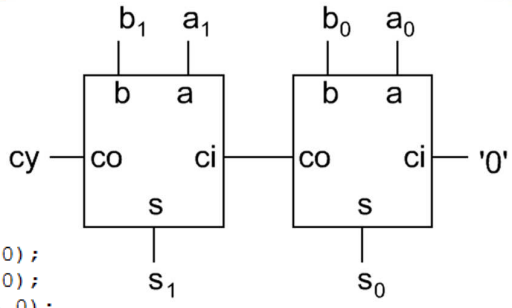
4.4. Additionneur (4ème façon)

```
architecture Behavioral of add is
   signal co_s : std_logic_vector (1 downto 0);
   signal ci_b_a : std_logic_vector (2 downto 0);
begin

process (ci_b_a) -- process (all) IEEE 2008
begin
   case ci_b_a is
      when "000" =>  co_s <= "00";
      when "001" =>  co_s <= "01";
      when "010" =>  co_s <= "01";
      when "011" =>  co_s <= "10";
      when "100" =>  co_s <= "01";
      when "101" =>  co_s <= "10";
      when "110" =>  co_s <= "10";
      when others => co_s <= "11";
   end case;
end process;

ci_b_a <= ci & b & a;
co <= co_s (1);
s <= co_s (0);

end Behavioral;
```

c_i	b	a	c_o	s
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

Encore une table de vérité écrite cette fois dans un process. Rappelez-vous ! La syntaxe n'est pas la même

4.5. Additionneur mots de 2 bits



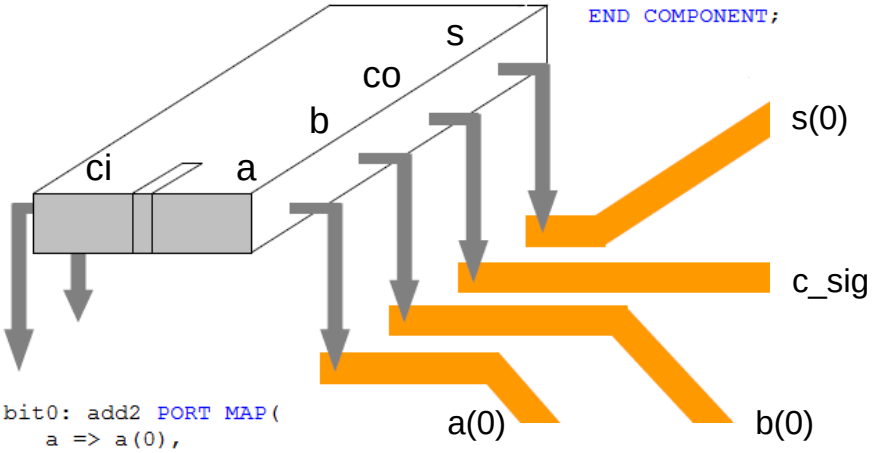

```
entity addi2bits is
    Port ( a : in  STD_LOGIC_VECTOR (1 downto 0);
           b : in  STD_LOGIC_VECTOR (1 downto 0);
           s : out  STD_LOGIC_VECTOR (1 downto 0);
           cy : out  STD_LOGIC);
end addi2bits;

architecture Behavioral of addi2bits is

   COMPONENT add2
   PORT(
      a : IN std_logic;
      b : IN std_logic;
      ci : IN std_logic;
      s : OUT std_logic;
      co : OUT std_logic
      );
   END COMPONENT;

   signal c_sig : std_logic;
```

```
begin

   bit0: add2 PORT MAP(
      a => a(0),
      b => b(0),
      ci => '0',
      s => s(0),
      co => c_sig
   );

   bit1: add2 PORT MAP(
      a => a(1),
      b => b(1),
      ci => c_sig,
      s => s(1),
      co => cy
   );


end Behavioral;
```

Description modulaire = schéma

- Un composant déclaré est comme un circuit intégré placé sur une carte

```
COMPONENT add2
PORT(
   a : IN std_logic;
   b : IN std_logic;
   ci : IN std_logic;
   s : OUT std_logic;
   co : OUT std_logic
   );
END COMPONENT;
```




```
bit0: add2 PORT MAP(
   a => a(0),
   b => b(0),
   ci => '0',
   s => s(0),
   co => c_sig
);
```

- Les signaux agissent comme des pistes imprimées sur la carte : les broches du CI sont connectées aux pistes

4.6. Additionneur mots de N bits

- Voici un additionneur générique qui utilise l'opération arithmétique « + » du package « numeric_std ».

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

entity addi_N is
   generic (N : natural := 8);
    Port ( a : in  STD_LOGIC_VECTOR (N-1 downto 0);
           b : in  STD_LOGIC_VECTOR (N-1 downto 0);
           ci : in  STD_LOGIC;
           s : out  STD_LOGIC_VECTOR (N-1 downto 0);
           co : out  STD_LOGIC);
end addi_N;

architecture Behavioral of addi_N is
   signal sum : STD_LOGIC_VECTOR (N downto 0);

begin
   sum <= ('0' & a) + ('0' & b) + ci;
   s <= sum (N-1 downto 0);
   co <= sum(N);

end Behavioral;
```

4.7. Compteur N bits

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

entity compteur1 is
    generic (N : natural := 8);
    Port ( clk : in  STD_LOGIC;
           count : out  STD_LOGIC_VECTOR (N-1 downto 0));
end compteur1;

architecture Behavioral of compteur1 is
   signal count_s : STD_LOGIC_VECTOR (N-1 downto 0);

begin

process (clk)
begin
   if rising_edge(clk) then
      count_s <= count_s + 1;
   end if;
end process;

count <= count_s;

end Behavioral;
```

Modulo variable

Signal intermédiaire

Liste de sensibilité

Front montant

Connexion de la sortie

4.8. Compteur avec reset asynchrone

```
entity compteur2 is
    generic (N : natural := 8);
    Port ( clk : in  STD_LOGIC;
           reset : in STD_LOGIC;
           count : out  STD_LOGIC_VECTOR (N-1 downto 0));
end compteur2;

architecture Behavioral of compteur2 is
   signal count_s : STD_LOGIC_VECTOR (N-1 downto 0);

begin

process (clk, reset)
begin
   if reset='1' then
      count_s <= (others => '0');
   elsif rising_edge(clk) then
      count_s <= count_s + 1;
   end if;
end process;

count <= count_s;

end Behavioral;
```

Liste de sensibilité complète

Reset prioritaire = asynchrone

Mise à ‘0’ de tous les bits

4.9. Compteur avec reset synchrone

```
entity compteur3 is
    generic (N : natural := 8);
    Port ( clk : in  STD_LOGIC;
           reset : in STD_LOGIC;
           count : out  STD_LOGIC_VECTOR (N-1 downto 0));
end compteur3;

architecture Behavioral of compteur3 is
   signal count_s : STD_LOGIC_VECTOR (N-1 downto 0);

begin

process (clk)
begin
   if rising_edge(clk) then
      if reset='1' then
         count_s <= (others => '0');
      else
         count_s <= count_s + 1;
      end if;
   end if;
end process;

count <= count_s;
```

Liste de sensibilité réduite

Front d'horloge prioritaire = Reset synchrone

4.10. Compteur avec validation

```
entity compteur4 is
    generic (N : natural := 8);
    Port ( clk : in  STD_LOGIC;
           reset : in STD_LOGIC;
           ce : in STD_LOGIC;
           count : out  STD_LOGIC_VECTOR (N-1 downto 0));
end compteur4;

architecture Behavioral of compteur4 is
   signal count_s : STD_LOGIC_VECTOR (N-1 downto 0);

begin

process (clk)
begin
   if rising_edge(clk) then
      if reset='1' then
         count_s <= (others => '0');
      elsif ce='1' then
         count_s <= count_s + 1;
      else
         count_s <= count_s;
      end if;
   end if;
end process;

count <= count_s;

end Behavioral;
```

« reset » prioritaire sur « ce »

« ce » valide le comptage

Pas de changement si ce = '0'

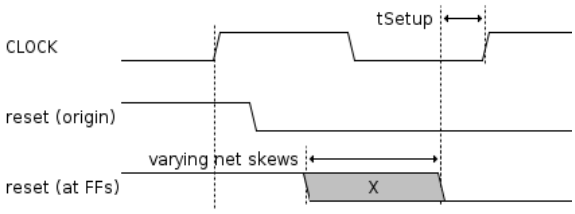
À propos du « reset »

➲ 1ère règle : n'utiliser de signal « reset » que si cela est nécessaire !

- De toute façon, toutes les bascules du FPGA sont remises à '0' à la mise sous tension.
- Les bénéfice sont les suivants :
  1. Moins de logique générée par le compilateur.
  2. Moins de problème au moment du routage.

➲ 2ème règle : le signal « reset » doit être synchrone !

- Les changements d'états des bascules ne se font que sur front montant d'horloge.
- Entre deux fronts actifs, on laisse le régime permanent s'établir pour éviter les instabilités.



4.11. Décodeur 2 vers 4

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity decode_2_4 is
    Port ( entree : in std_logic_vector(1 downto 0);
           sortie : out std_logic_vector(3 downto 0));
end decode_2_4;

architecture Behavioral of decode_2_4 is

begin

   sortie <= "0001" when entree = "00" else
             "0010" when entree = "01" else
             "0100" when entree = "10" else
             "1000" when entree = "11" else
             "ZZZZ";

end Behavioral;
```

➲ Système combinatoire décrit hors « process » en utilisant la structure « when … else »

4.12. Multiplexeur

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity multipexe is
    Port ( a : in std_logic;
           b : in std_logic;
           c : in std_logic;
           d : in std_logic;
           sel : in std_logic_vector(1 downto 0);
           sortie : out std_logic);
end multipexe;

architecture Behavioral of multipexe is

begin
   sortie <= a when sel = "00" else
             b when sel = "01" else
             c when sel = "10" else
             d when sel = "11" else
             'X';
end Behavioral;
```

- Système combinatoire décrit hors « process » en utilisant la structure « when … else »

4.13. Décodeur hexa 7 segments

```
entity hex2led is
    Port ( HEX : in std_logic_vector(3 downto 0);
           LED : out std_logic_vector(6 downto 0));
end hex2led;

architecture Behavioral of hex2led is

begin
       with HEX SELect
   LED<= "1111001" when "0001",   --1
         "0100100" when "0010",   --2
         "0110000" when "0011",   --3
         "0011001" when "0100",   --4
         "0010010" when "0101",   --5
         "0000010" when "0110",   --6
         "1111000" when "0111",   --7
         "0000000" when "1000",   --8
         "0010000" when "1001",   --9
         "0001000" when "1010",   --A
         "0000011" when "1011",   --b
         "1000110" when "1100",   --C
         "0100001" when "1101",   --d
         "0000110" when "1110",   --E
         "0001110" when "1111",   --F
         "1000000" when others;   --0
end Behavioral;
```

- Système combinatoire décrit hors « process » en utilisant la structure « with … select ». C'est une façon d'écrire la table de vérité.

4.14. Encodeur de priorité

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity encodeur is
    Port ( entree : in std_logic_vector(3 downto 0);
           sortie : out std_logic_vector(1 downto 0));
end encodeur;

architecture Behavioral of encodeur is
begin

   process (entree)
   begin
      if entree(3)='1' then sortie<="11";
      elsif entree(2)='1' then sortie<="10";
      elsif entree(1)='1' then sortie<="01";
      elsif entree(0)='1' then sortie<="00";
      else sortie<="XX";
      end if;
   end process;

end Behavioral;
```

- Système combinatoire décrit dans un « process » en utilisant la structure « if… then … else ». On peut décrire une fonction combinatoire dans un « process » !

4.15. Calcul de bit de parité

```
entity parite is
    Port ( mot : in std_logic_vector(7 downto 0);
           parite : out std_logic);
end parite;

architecture Behavioral of parite is
begin

   process (mot)
      variable x : std_logic;
   begin
      x := '0';
      for i in 0 to 7 loop
         if mot(i) = '1' then
            x := not x;
         end if;
      end loop;
         parite <= x;
   end process;

end Behavioral;
```

- Système combinatoire décrit dans un « process » en utilisant la structure « for… loop ».

4.16. Diviseur d'horloge par 2^N

```
entity clk_div is
   generic (N : natural := 5);

   Port ( clk : in  STD_LOGIC;
          reset : in  STD_LOGIC;
          div_out : out  STD_LOGIC);
end clk_div;

architecture Behavioral of clk_div is
   signal count_temp : std_logic_vector(N-1 downto 0);

begin

process (clk)
begin
   if rising_edge (clk) then
      if (reset = '1') then
         count_temp <= (others => '0');
      else
         count_temp <= count_temp + 1;
      end if;
   end if;
end process;

   div_out <= count_temp(N-1);
end Behavioral;
```

- La division d'une fréquence se fait toujours avec un compteur synchrone.

4.17.1. Conversion binaire vers BCD

- L'algorithme de conversion utilise le fait que tous les chiffres compris entre 0 et 4 produisent un résultat sur un chiffre lorsqu'ils sont multipliés par 2
- Par contre tous les chiffres compris entre 5 et 9 donnent un résultat sur 2 chiffres lorsqu'ils sont multipliés par 2

chiffre	Chiffre*2	dizaines	unités
0	0	0	0
1	2	0	2
2	4	0	4
3	6	0	6
4	8	0	8
5	10	1	0
6	12	1	2
7	14	1	4
8	16	1	6
9	18	1	8

4.17.2. Application de l'algorithme

$0 \leq nb \leq 4$

En décimal :

Nb	Nb*2
0	0
1	2
2	4
3	6
4	8

En BCD : un décalage à gauche

Nb	Nb*2
0000	0000
0001	0010
0010	0100
0011	0110
0100	1000

$5 \leq nb \leq 9$

Exemple en décimal : $2 \times 6 = 12$

En BCD : ajouter 3 et décaler à gauche

	diz	uni
6	0000	0110
+3	0000	0011
=	0000	1001
X2	0001	0010

4.17.3. Algorithme complet

- Déclarer un registre de 20 bits : 4 + 4 + 4 + 8
- Les 8 bits de poids faible représente le mot binaire à convertir
- Suivent ensuite 3 quartets dans l'ordre : les unités, les dizaines puis les centaines du résultat en BCD

Réaliser les étapes suivantes 8 fois

1. Si les unités sont ≥ 5 , ajouter 3. Faire la même chose pour les dizaines et les centaines
2. Décaler tout le registre d'un bit vers la gauche

Operation	Hundreds	Tens	Units	Binary	
HEX				F	F
Start				1 1 1 1	1 1 1 1

4.17.4. Conversion d'un mot 8 bits

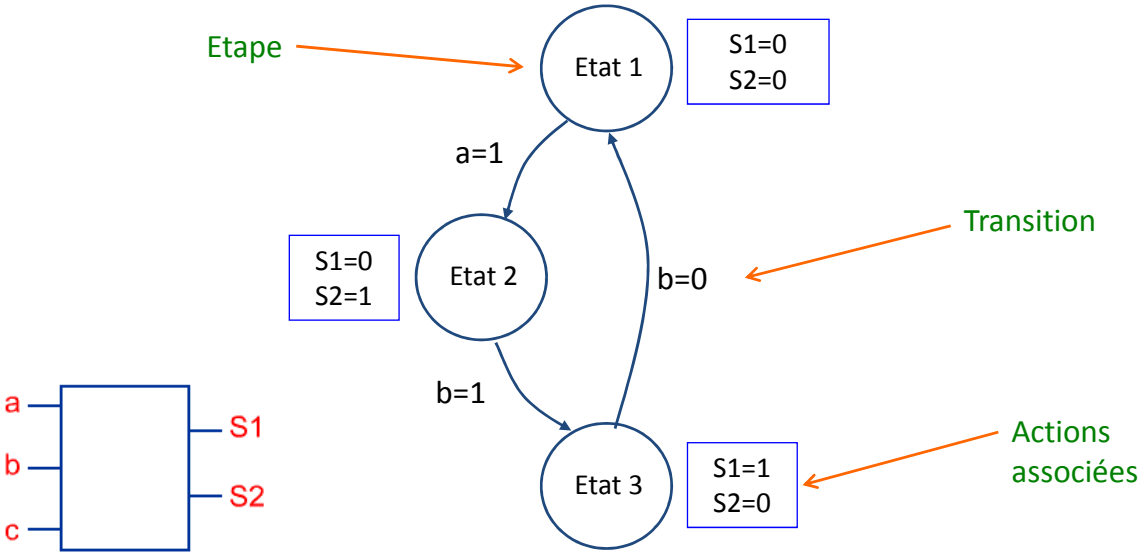
					Mot binaire	
décalage	Opération	CENT	DIZ	UNI	7654	3210
	START	0000	0000	0000	1111	1111
1	SHIFT	0000	0000	0001	1111	1110
2	SHIFT	0000	0000	0011	1111	1100
3	SHIFT	0000	0000	0111	1111	1000
	+ 3	0000	0000	**1010**	1111	1000
4	SHIFT	0000	0001	0101	1111	0000
	+ 3	0000	0001	**1000**	1111	0000
5	SHIFT	0000	0011	0001	1110	0000
6	SHIFT	0000	0110	0011	1100	0000
	+ 3	0000	**1001**	0011	1100	0000
7	SHIFT	0001	0010	0111	1000	0000
	+ 3	0001	0010	**1010**	1000	0000
8	SHIFT	0010	0101	0101	0000	0000
		2	**5**	**5**		

4.17.5. Code VHDL (binaire -> BCD)

```
process(binaire)
    variable registre : std_logic_vector(19 downto 0);
begin
    registre := x"000" & binaire;       -- registre de travail

    for i in 1 to 8 loop          -- 8 bits à convertir
        if registre(11 downto 8) > 4 then    -- centaines
            registre(11 downto 8) := registre(11 downto 8) + 3;
        end if;
        if registre(15 downto 12) > 4 then   -- dizaines
            registre(15 downto 12) := registre(15 downto 12) + 3;
        end if;
        if registre(19 downto 16) > 4 then   -- unités
            registre(19 downto 16) := registre(19 downto 16) + 3;
        end if;
        registre := registre(18 downto 0) & '0';     -- décalage G
    end loop;

    cent <= registre(19 downto 16); -- sorties décimales
    diz <= registre(15 downto 12);
    uni <= registre(11 downto 8);
end process;
```

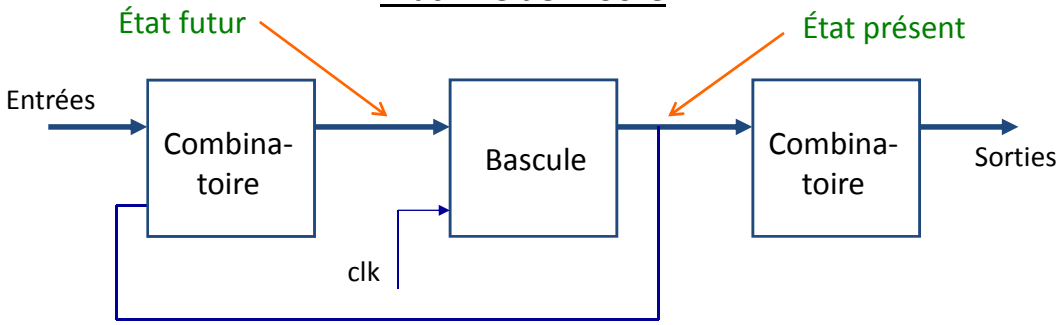
5.1. Machine d'états

- Une machine d'états est un séquenceur qui décrit le fonctionnement d'un système séquentiel aussi complexe soit-il.
- On utilise des symboles : les bulles indiquent les états du système et les arcs orientés définissent les possibilités d'évolution. Le passage d'un état à un autre est régi par une condition (transition). Les sorties dépendent de l'état présent.

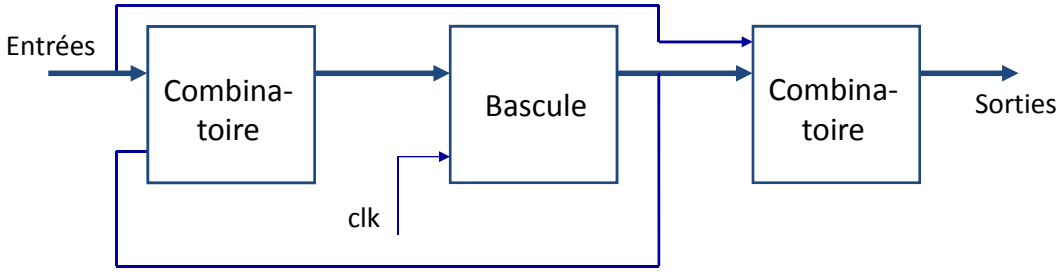


5.2. Machines de Moore et Mealy

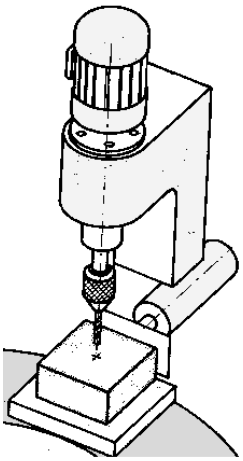
Machine de Moore

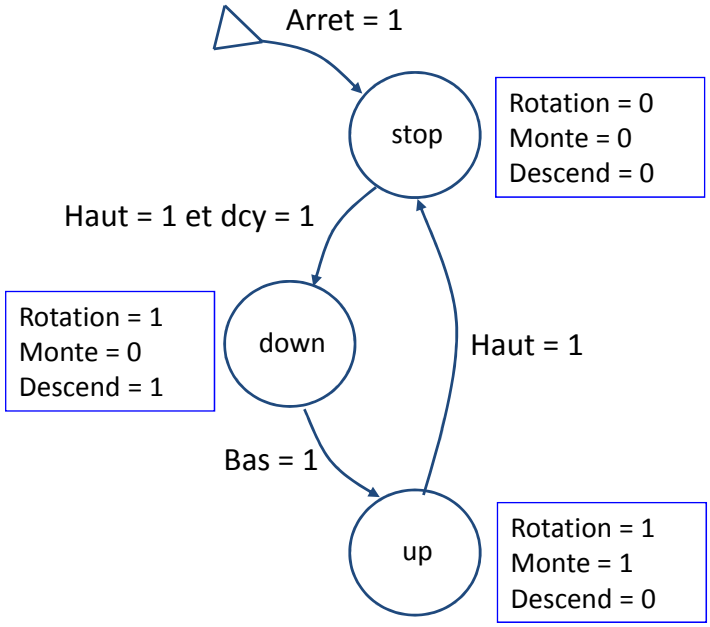


Machine de Mealy



5.3. Exemple d'une perceuse





5.4. Code VHDL de la perceuse

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

entity perce is
    port   (   clk, arret, dcy, haut, bas : in STD_LOGIC;
               rotation, monte, descend : out STD_LOGIC);
end perce;

architecture perce_arch of perce is
 type etape_type is (stop, down, up);
   signal etape, etape_suiv : etape_type;
begin

------------------------ voici un process synchrone

SYN_PROC : process (clk, arret)
begin
   if arret='1' then
      etape <= stop ;
   elsif clk'event and clk = '1' then
      etape <= etape_suiv ;
   end if ;
end process ;              -- fin du process synchrone
```

Process synchrone sensible aux signaux clk et arret. C'est ici qu'évolue le système vers l'état suivant

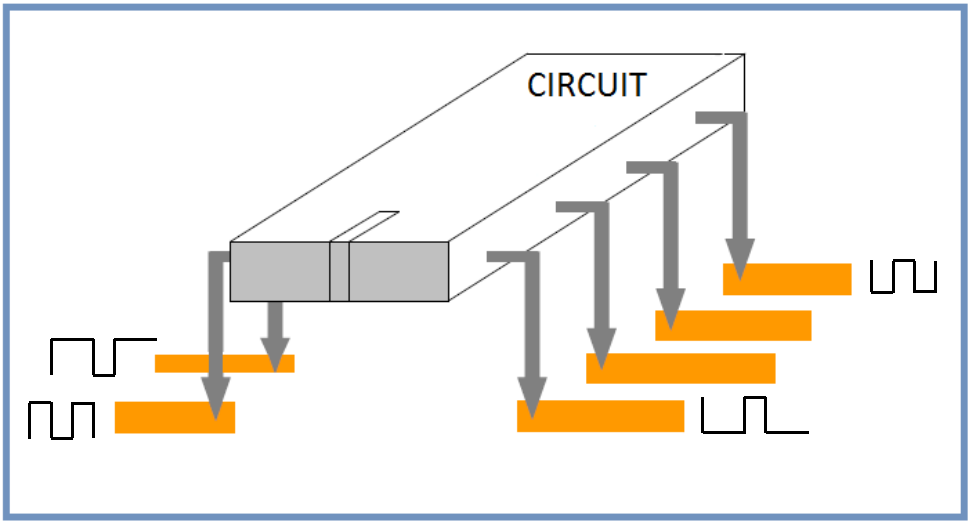
Suite code de la perceuse

```
COMB_PROC : process (etape, dcy, haut, bas)
begin
   case etape is
      when stop =>              -- etape (0)
            rotation <= '0' ;
            descend <= '0' ;monte <= '0' ;
         if dcy='1' and haut='1' then
            etape_suiv <= down;
         else
            etape_suiv <= stop ;
         end if;
      when down =>              -- etape (1)
            rotation <= '1' ;
            descend <= '1' ; monte <= '0' ;
         if bas='1' then
            etape_suiv <= up;
         else
            etape_suiv <= down ;
         end if;
      when up =>              -- etape (2)
            rotation <= '1' ;
            descend <= '0' ;monte <= '1' ;
         if haut='1' then
            etape_suiv <= stop;
         else
            etape_suiv <= up ;
         end if;
   end case;
end process;       -- fin du process combinatoire
end perce_arch;    -- fin de l'architecture
```

Process combinatoire qui calcule le prochain état en fonction de l'état actuel et active les sorties.

6.1. Principe du « Testbench »

- Tester un circuit revient à lui imposer des signaux en entrée 'stimuli' et de regarder comment évoluent les sorties . Si les réponses correspondent à ce que l'on attend, le test est bon, sinon il y a erreur.
- Le 'testbench' est un programme VHDL qui réalise toutes ces opérations .



TESTEUR

6.2. Circuit combinatoire à tester

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity additionneur1 is
    Port ( a : in std_logic;
           b : in std_logic;
           retenue_e : in std_logic;
           somme : out std_logic;
           retenue_s : out std_logic);
end additionneur1;

architecture Behavioral of additionneur1 is

begin
   somme <= a xor b xor retenue_e;
   retenue_s <= (a and b)or
     (a and retenue_e)or(b and retenue_e);

end Behavioral;
```

- Nous avons affaire à un circuit combinatoire à 3 entrées et 2 sorties, il s'agit d'un additionneur complet.

6.3. Testbench : déclarations

```
ENTITY additionneur1_testeur_vhd_tb IS
END additionneur1_testeur_vhd_tb;

ARCHITECTURE behavior OF additionneur1_testeur_vhd_tb IS

   COMPONENT additionneur1
   PORT(
      a : IN std_logic;
      b : IN std_logic;
      retenue_e : IN std_logic;
      somme : OUT std_logic;
      retenue_s : OUT std_logic
      );
   END COMPONENT;

   SIGNAL a :  std_logic;
   SIGNAL b :  std_logic;
   SIGNAL retenue_e :  std_logic;
   SIGNAL somme :  std_logic;
   SIGNAL retenue_s :  std_logic;

BEGIN
```

- En début de programme, on déclare le composant à tester ainsi que les différents signaux qui vont servir à connecter le composant.

6.4. Testbench : simulation

```
BEGIN

   uut: additionneur1 PORT MAP(
      a => a,
      b => b,
      retenue_e => retenue_e,
      somme => somme,
      retenue_s => retenue_s
   );

   tb : PROCESS
   BEGIN
      a<='0'; b<='0'; retenue_e<='0';
      wait for 100 ns;
      a<='1';
      wait for 100 ns;
      b<='1';
      wait for 200 ns;
      retenue_e<='1';
      wait for 100 ns;
      a<='0'; b<='0';
   END PROCESS;

END;
```

- La fin du programme sert à connecter le composant et lui imposer des « stimuli » en entrée pour savoir comment il réagit.

6.5. Circuit séquentiel à tester

```
entity diviseur is
    Port ( reset : in std_logic;
           clk : in std_logic;
           sortie : out std_logic);
end diviseur;

architecture Behavioral of diviseur is
   signal tempo : std_logic_vector(2 downto 0);
begin

process (reset, clk)
begin
   if (reset='0') then
      tempo <= "000";
   elsif (clk'event and clk='1') then
      tempo <= tempo + 1;
   end if;
end process;
sortie <= tempo(2);

end Behavioral;
```

- Il s'agit de simuler un diviseur de fréquence par 8 qui utilise un compteur binaire sur 3 bits.

6.6. Testbench : déclarations

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY diviseur_testeur2_vhd_tb IS
END diviseur_testeur2_vhd_tb;

ARCHITECTURE behavior OF diviseur_testeur2_vhd_tb IS

   COMPONENT diviseur
   PORT(
      reset : IN std_logic;
      clk : IN std_logic;
      sortie : OUT std_logic
      );
   END COMPONENT;

   SIGNAL reset :  std_logic;
   SIGNAL clk :  std_logic;
   SIGNAL sortie :  std_logic;

BEGIN
```

- En début de programme, on déclare le composant à tester ainsi que les différents signaux qui vont servir à connecter le composant.

6.7. Testbench : simulation

```
BEGIN

   uut: diviseur PORT MAP(
      reset => reset,
      clk => clk,
      sortie => sortie  );

   horloge : PROCESS
   BEGIN
         clk <= '0';
         wait for 10 ns;
         clk <= '1';
         wait for 10 ns;
   END PROCESS;

   raz : PROCESS
      BEGIN
         reset <= '1';
         wait for 15 ns;
         reset <= '0';
         wait for 40 ns;
         reset <= '1';
         wait;
      END PROCESS;

END;
```

- La fin du programme sert à connecter le composant et lui imposer des « stimuli » en entrée pour savoir comment il réagit.

6.8. Contrôle des résultats

```
-- Stimulus process
 stim_proc: process
begin
    -- hold reset state for 100 ns.
   reset <= '1';
      sig <= '0';
      wait for clk_period*5;
      reset <= '0';
      wait for 120ns;
      sig <= '1';
      wait for clk_period*10;

      assert sig_mef = '0'     -- teste si sig_mef = '0' sinon
      report "La valeur de sig_mef est fausse "  -- afficher ce message
      severity FAILURE;        -- et abandonner la simulation

      wait for clk_period*65535;
end process;
```