Cours de Systèmes numériques

Ch. 1 : Introduction au VHDL

Nicolas.Schroeter@hefr.ch

Références

- Livres VHDL:
  - Volnei A. Pedroni, “Circuit Design with VHDL”
  - Peter J. Ashenden, “The Designer's Guide to VHDL, 2nd Edition”
  - Andrew Rushton, “VHDL for Logic Synthesis, 3rd Edition“
- Liens utiles:

  http://vhdl.renerta.com/

Qu'est ce que VHDL ?

- VHDL signifie:
  - VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
- Langage de haut niveau permettant de décrire et de simuler un système numérique, pour ensuite l'implémenter physiquement dans un circuit programmable
- Il existe deux langages de haut niveau utilisés pour la modélisation de circuits numériques:
  - VHDL (utilisé plutôt en Europe)
  - Verilog (préféré aux USA)
- Développé par le DoD (année 1980)
- Basé sur Ada: langage fortement typé.

Histoire du VHDL

- 1987: publication de IEEE Standard 1076
- 1993: révision du standard, VHDL 1076
- 2000: révision du type 1993 avec l'ajout de type protégé
- 2002: ajout notamment du type buffer port
- 2008: changements importants → pas d'outils de synthèses supportent complètement cette version

  → ce cours se concentre sur VHDL-93

- Dans le langage courant, l'année de publication des standards désigne la version VHDL.

Utilisation du VHDL

- Pour utiliser le VHDL, il faut apprendre le langage, mais il est aussi nécessaire de savoir comment les outils vont interpréter le code:

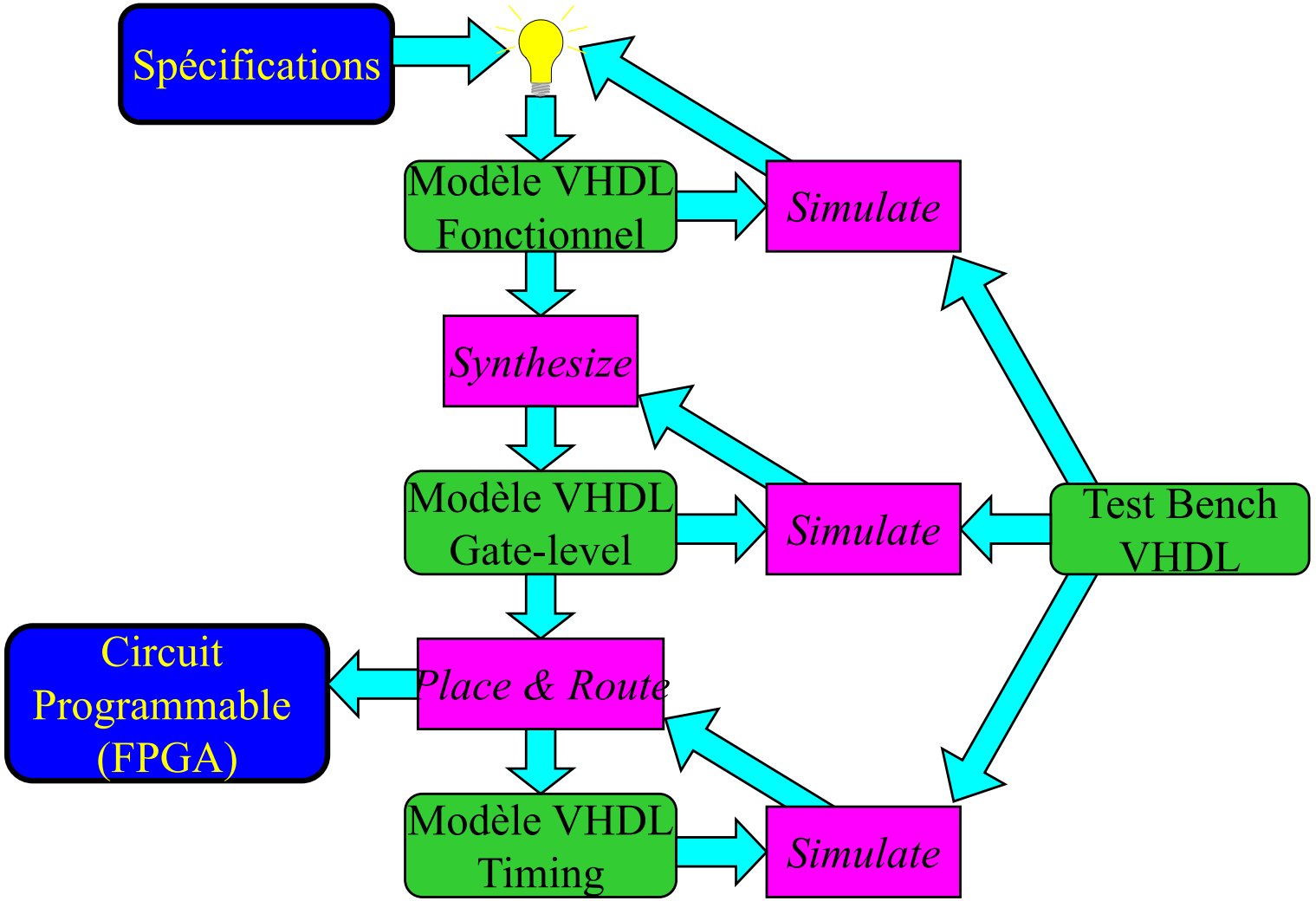
Inference == déduire

Des motifs de description sont à appliquer pour obtenir le bon circuit.

Introduction

- VHDL est destiné à la synthèse ainsi qu'à la simulation de circuits numériques
- Le VHDL est entièrement simulable, mais toutes les instructions ne sont pas synthétisables
- Ce cours mettra l'accent sur le VHDL synthétisable pour les circuits numériques intégrés
- VHDL est :
  - un langage standard
  - indépendant de la technologie
  - portable et réutilisable.
- En synthétisant un code VHDL, il est ensuite possible de l'utiliser afin programmer des dispositifs à logiques programmables (CPLD ou FPGA) ou d'implémenter directement un ASICs
- Contrairement à d'autres langages de programmation séquentiels, le VHDL a la particularité d'avoir des instructions dites concurrentes (parallèle).

Design flow


Spécifications
Modèle VHDL Fonctionnel
Simulate
Synthesize
Modèle VHDL Gate-level
Simulate
Test Bench VHDL
Circuit Programmable (FPGA)
Place & Route
Modèle VHDL Timing
Simulate

VHDL: les outils

Les outils sont:

- Le simulateur: interprète et comprend tout le langage VHDL (ex: ModelSim).
- Le synthétiseur: reconnaît/identifie automatiquement dans le code source des structures matérielles:
  - Logique combinatoire
  - Machines d'états synchrones
  - Registres ou mémoires

  et traduit le résultat en une netlist (format EDIF).
- Le placement et le routage: est fourni par le fabricant du circuit (CPLD, FPGA); il associe la netlist aux éléments disponibles du circuit.

Design flow

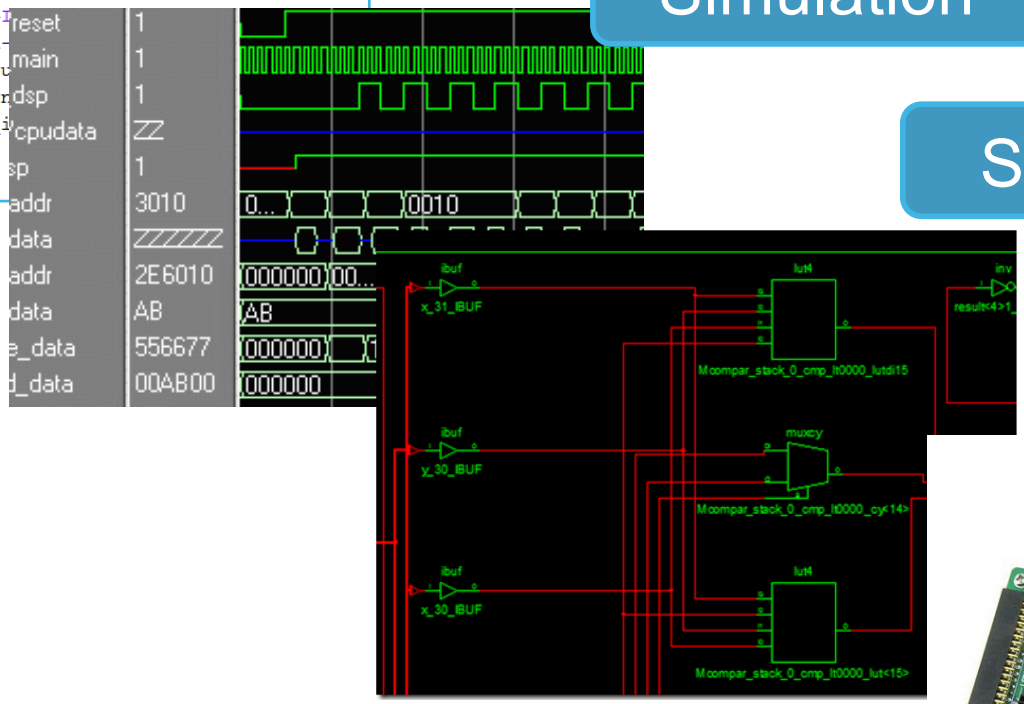
Design entry

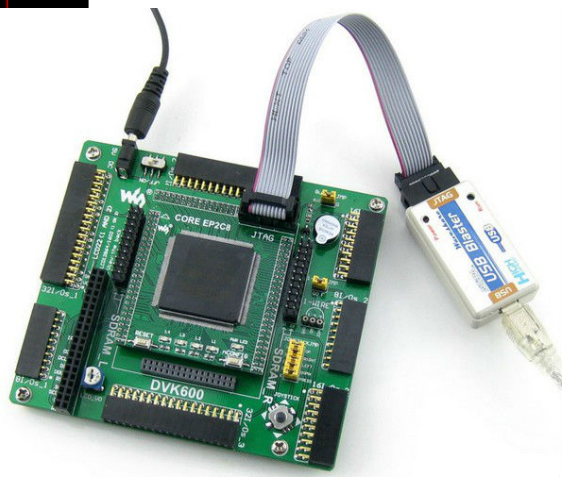
Simulation

Synthesis

Hardware

```
LIBRARY ieee ;  USE ieee.std_logic_1164.ALL;
LIBRARY work ; USE work.regulator_pack.ALL;

ENTITY smith_pred  IS
   PORT(
    clk_i :  IN STD_LOGIC;
    reset_i : IN STD_LOGIC;
    clk_en_i : IN STD_LOGI
    e_i : IN SIGNED(WL_in-
    u_o : OUT SIGNED(WL_ou
    yd_i : IN SIGNED(WL_in
    yd_o : OUT SIGNED(WL_i
    );
END ENTITY;
```

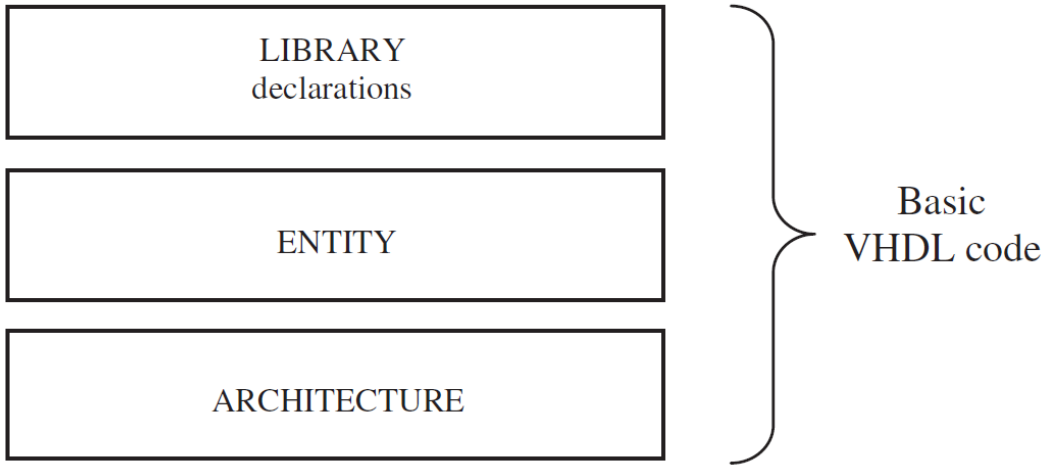




Structure d'un composant VHDL

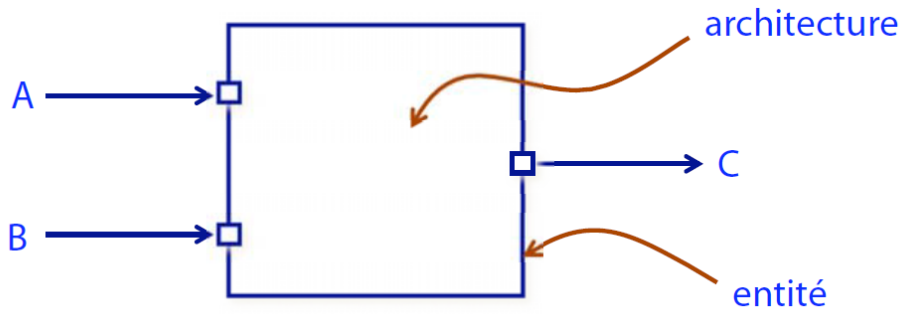
Un code VHDL se compose au minimum de trois parties fondamentales:

- Library: déclaration d'une liste de bibliothèques qui pourront être utilisé dans la description du circuit design.
  Par exemple ieee, std, work…
- Entity: spécification des I/O du circuit
- Architecture: contient le code VHDL proprement dit, qui décrit le comportement que le circuit doit avoir



Structure d'un composant VHDL

- Un composant VHDL est composé de :
  - Références à des bibliothèques
  - Entité correspond aux broches d'un circuit:
    - définit l'interface avec l'extérieur sans préciser le contenu
  - Architecture correspond à la fonction du système:
    - définit le fonctionnement interne du circuit
- Une entité doit toujours être associée avec au moins une description de son contenu → architecture



Signaux, Types et Opérateurs (1)

- Pour connecter différentes parties d'un schéma (design), VHDL utilise des signaux, équivalents au fils dans le matériel.
- A chaque signal est associé un type. Pour la synthèse, les types les plus utilisé sont std_logic, pour représenter 1 bit, et std_logic_vector pour les bus. Ces types sont définis dans le paquetage std_logic_1164 qui réside dans la bibliothèque IEEE.

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

Signaux, Types et Opérateurs (2)

- Le type std_logic est un énuméré à 9 valeurs:
  - '**U**' non-initialisé
  - '**X**' inconnu fort
  - '**0**' logique 0 fort
  - '**1**' logique 1
  - '**Z**' haute impédance
  - '**W**' inconnu faible
  - '**L**' logique 0 faible (pull-down) (collecteur ouvert)
  - '**H**' logique 1 faible (pull-up)
  - '**-**' état indifférent (don't care)
- Pour les affectations, les valeurs surtout utilisées sont '0', '1' et 'Z'.

Signaux, Types et Opérateurs (3)

Std_logic_vector:

- Type composé de std_logic:

```
type Std_Logic_Vector is array(natural range<>) of Std_Logic;
```

- Généralement, lors de la déclaration d'un signal, la taille du vecteur est spécifiée:

```
signal Vecteur : Std_Logic_Vector(7 downto 0);
signal Vecteur : Std_Logic_Vector(7 downto 0):= (others => '0');
signal Vecteur : Std_Logic_Vector(7 downto 0):= "11001100";
```

L'indice 7 est le MSB *LSB*

NB :

Il est possible de ne pas définir la taille d'un vecteur. Nous parlons alors de vecteur non-contraint. Il s'agit d'une application particulière pour les descriptions réutilisables.

Signaux, Types et Opérateurs (4)

Opérateurs dans l'ordre de précédence:

- Divers: **** abs not**
- Multiplication: *** / mod rem**
- Signe (unaire): **+ -**
- Addition: **+ - & (concaténation)**
- Décalage: **sll srl sla sra rol ror**
- Relationnel: **= /= < <= > >=**
- Logiques : **and or nand nor xor xnor**

NB :

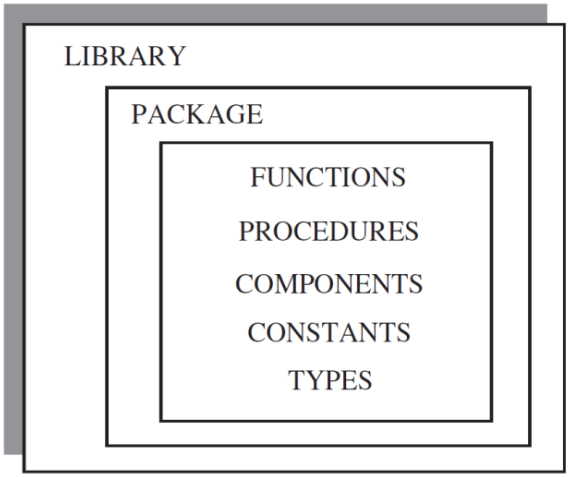
Il est possible de surcharger les opérateurs et de les définir pour les nouveaux types.

Structure d'un composant VHDL

```vhdl
-- Librairie IEEE
library IEEE;
use IEEE.Std_Logic_1164.all; --Defini type Std_Logic
entity Exemple is
    port(Entree_i : in Std_Logic;
         Vecteur_i : in Std_Logic_Vector(3 downto 0);
         Sortie_o : out Std_Logic;
         BiDir_io : inout Std_Logic
        );
end Exemple;

architecture Style_Description of Exemple is
--zone de déclaration
begin
    --Instructions concurrentes .......
    process (Liste_De_Sensibilité)
    begin
    --Instructions séquentielles .......
    end process;
end Style_Description;
```

Library

- Une library est une ensemble de packages.
- Chaque package contient des déclarations qui peuvent être utilisées par plusieurs composants (réutilisabilité/partage du code)
- Un package se compose de functions, procedures, components constantes et types:



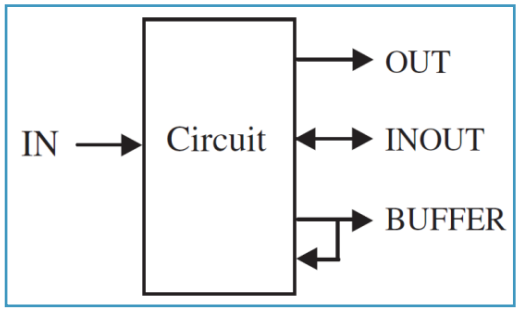
- Deux lignes sont nécessaires pour importer une library et un package:

```
LIBRARY library_name;
USE library_name.package_name.package_parts;
```

Entité

- L'entité décrit précisément les entrées et sorties (PORT) du circuit à modéliser

```
ENTITY entity_name IS
   PORT (
      port_name : signal_mode signal_type;
      port_name : signal_mode signal_type;
      ...);
END entity_name;
```

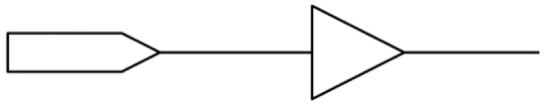


- Le type d'un port peut être soit IN, OUT, INOUT ou encore BUFFER
- IN et OUT sont purement unidirectionnelles, le port de type OUT ne peut pas être lu
- INOUT est bidirectionnelle et doit être utilisé uniquement pour des signaux tri-state dans le composant top-level.
- BUFFER est utilisé lorsque une sortie doit être lu en interne (ne pas utiliser)
- Le nom de l'entité (= composant) peut être choisi arbitrairement
- Le nom du fichier contenant le code VHDL doit contenir le même nom que l'entité avec l'extension .vhd

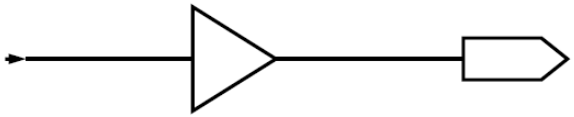
NB: Dans l'entité, uniquement les types std_logic ou std_logic_vector doivent être utilisés

Types de port

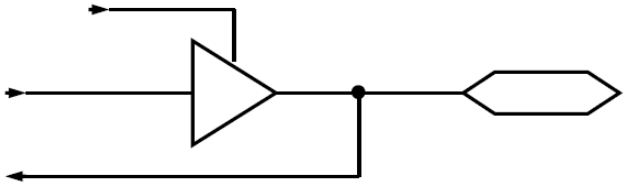
port **IN**



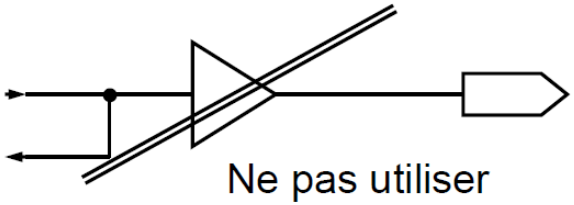
`port` **`out`**



`port` **`inout`**

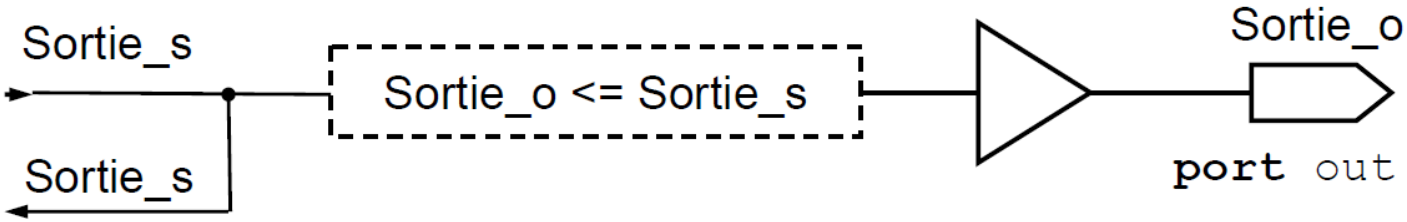


`port` **`buffer`**



Ne pas utiliser

Remplacer par un signal interne et le type de port **out**, soit:



Architecture

- L'architecture décrit le fonctionnement du composant
- L'architecture possède deux parties:
  - Déclarative: déclaration des signaux, constantes, types, procédures, fonctions où composants (component) utilisés dans le circuit
  - Code: contient le code décrivant le comportement du circuit (affectations <=, process…), ainsi que l'instanciation (mapping) de composants (port map)

```
ARCHITECTURE architecture_name OF entity_name IS
    [declarations]
BEGIN
    (code)
END architecture_name;
```

Architecture: zone de déclaration

- Déclaration de signaux internes:

```
signal Interne_s : Std_Logic;
signal Vect_s : Std_Logic_Vector(4 downto 0);
```

  - Possibilité de donner une valeur initiale

```
signal A : Std_Logic := '0';
```

Cette initialisation n'est utile que pour la simulation.
Fortement déconseillée pour la synthèse.

- Déclaration de constantes:

```
constant Val_c : Std_Logic_Vector(2 downto 0):="101";
```

- Déclaration de composants:

```
COMPONENT xor2 -- déclaration du OU-EXCLUSIF
    PORT(a,b : IN std_logic;
         q : OUT std_logic
    );
END COMPONENT;
```

- Déclaration de types, de procédures et de fonctions

Architecture: zone de code

- Dans un circuit:
  - Toutes les portes fonctionnent simultanément
  - Tous les signaux évoluent de manière concurrente

- La zone de code est constituée d'un ensemble d'instructions concurrentes.
- Chaque instruction concurrente correspond à un bloc du composant qui peut être dénommée avec un label.
- Toutes les instructions concurrentes s'exécutent en parallèle.
- Le process est décrit avec des instructions séquentielles qui sont évaluées les unes après les autres.

NB: L'ordre, dans lequel les instructions concurrentes sont décrites, n'est pas important.

Conception avec le VHDL

- Il faut penser CIRCUIT == ensemble de blocs
- Une bonne conception commence par une décomposition du système sous forme hiérarchique
- Une approche «top-down» lors de la conception:
  - Un circuit numérique complexe est composé d'un ensemble de fonctions logiques simples reliées à des registres (RTL)
  - Approche Controller (machines d'états) / Worker
- Nommer les entités et les signaux de façon à comprendre leur fonctionnalité
- Imaginer l'architecture physique du système et comment le synthétiseur va interpréter votre code
- Approche «bottom-up» dans la conception des circuits

Portabilité des descriptions

- Afin de garantir une bonne portabilité et réutilisabilité des descriptions VHDL:
  - Diviser pour régner:

    Une seule fonction par composant VHDL
  - Faite des descriptions simples et lisibles
  - Expliciter clairement chaque port de l'entité si il est registre ou combinatoire
  - Définir clairement le mode de communication des signaux, par événement ou par état
  - Utiliser uniquement les bibliothèques standardisées IEEE
  - Utiliser les mécanismes avancés de VHDL.

Instructions concurrentes

- Affectation:
  - Y <= A and C;
- Affectation avec condition:
  - Y <= … when … else …;
- Affectation de sélection:
  - With … select
- Instanciation de composants (mapping)
  - Generic map, port map
- process

Instruction d'affectation

signal1 <= *expression*; --ex: signal2 **opérateur** signal3;

- L'affectation représente un lien définitif entre le signal et le circuit **combinatoire** généré par l'expression.
- En cas d'affectations multiples d'un signal dans différentes instructions concurrentes:
  - Il n'y a pas d'erreur pour le langage VHDL, le simulateur détecte ce cas par l'état 'X'
  - Le synthétiseur détecte l'erreur et annonce que deux sorties sont connectées ensemble (court-circuit possible).
- Exemples:

```
Signal_1bit <= '0';                          -- simples guillemets
sortie <= (entreeA and Signal_1bit) or oo;   -- pas de priorité entre opérateurs → ()

Bus_4bits <= "1110"; -- Bus_4bits(3 downto 1) <= "111"; Bus_4bits(0) <= '0';
oe <= Bus_4bits(2 downto 1) or Bus_4bits(1 downto 0);
```

Affectation avec condition

```
signal1 <= expression1 when cond_booleen1 else
           expression2 when cond_booleen2 else
           … else
           expression;
```

- Toujours finir l'instruction avec un else, sinon la synthèse peut produire des circuits erronés (latch)!
- Exemple:

```
sortie <= '1' when a='0' or b='1' else '0';
z <= b or f when a='1' or (a='0' and b='0' and c='1') else d;
Out1 <= a and b when c='1' else
        d when sortie='0' else
        '1';
```

NB: L'instruction when … else implique une notion de priorité entre les différentes conditions

Affectation de sélection

```
with expression_commande select
    signal1 <= expression when choice,
               expression when choice | … | choice,
               …
               expression when others;
```

- **others** définit tous les autres choix possibles de « expression_commande » qu'il faut absolument traiter (latch)!
- Exemples:

```
with signal_sel select
    Sortie <=  Entr_A when "00" | "01", -- un choix est une valeur constante
               Entr_B when "10",
               '1' when "11",
               '0' when others; --pour couvrir toutes les autres
                                -- combinaisons "UZ",…
```

Instanciation d'un composant

- Dans la zone de déclaration de l'architecture:

```
component Nom_Composant is
    port(entre1: in std_logic;
         ...
         sortie1: out std_logic
    );
end component;

for all: Nom_Composant use entity work.Nom_entity(style_description);
```

NB: Consiste à copier l'entity du composant et remplacer par COMPONENT… END COMPONENT

- Dans la zone de description de l'architecture:

```
Label1: Nom_composant port map(
    SignalIn => Signal_architecture1_s,
    SignalOut=> Signal_architecture2_s
);
```

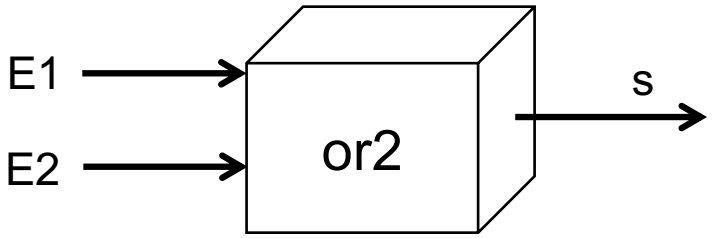
Instanciation d'un composant

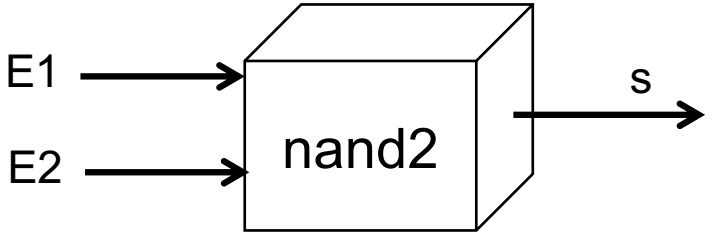
- Exemple:

```
architecture Struct of Exemple is

    component PORTE_ET is
        port(A_i, B_i : in std_logic;
             Z_o      : out std_logic
        );
    end component;

    for all: PORTE_ET use entity work.PORTE_ET(Behav);
    signal Signal_s: std_logic;
begin

    U1:PORTE_ET port map(
        A_i => Entree_i,
        B_i => Signal_s,
        Z_o => Sortie_o
    );

end Struct;
```

Instanciation d'un composant

- Autre variante: déclaration que dans la zone de code

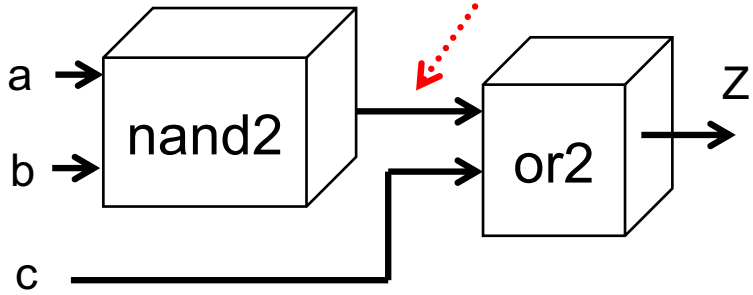




Configuration

sign_s

```
ARCHITECTURE struct OF circuit IS
  SIGNAL sign_s : std_logic;
BEGIN
   c1 : entity work.nand2 (archi_nand)
        port map( E1 => a, E2 => b, s => sign_s);
   c2 : entity work.or2 (archi_or)
        port map( E1 => sign_s, E2 => c, s => Z);
END struct;
```



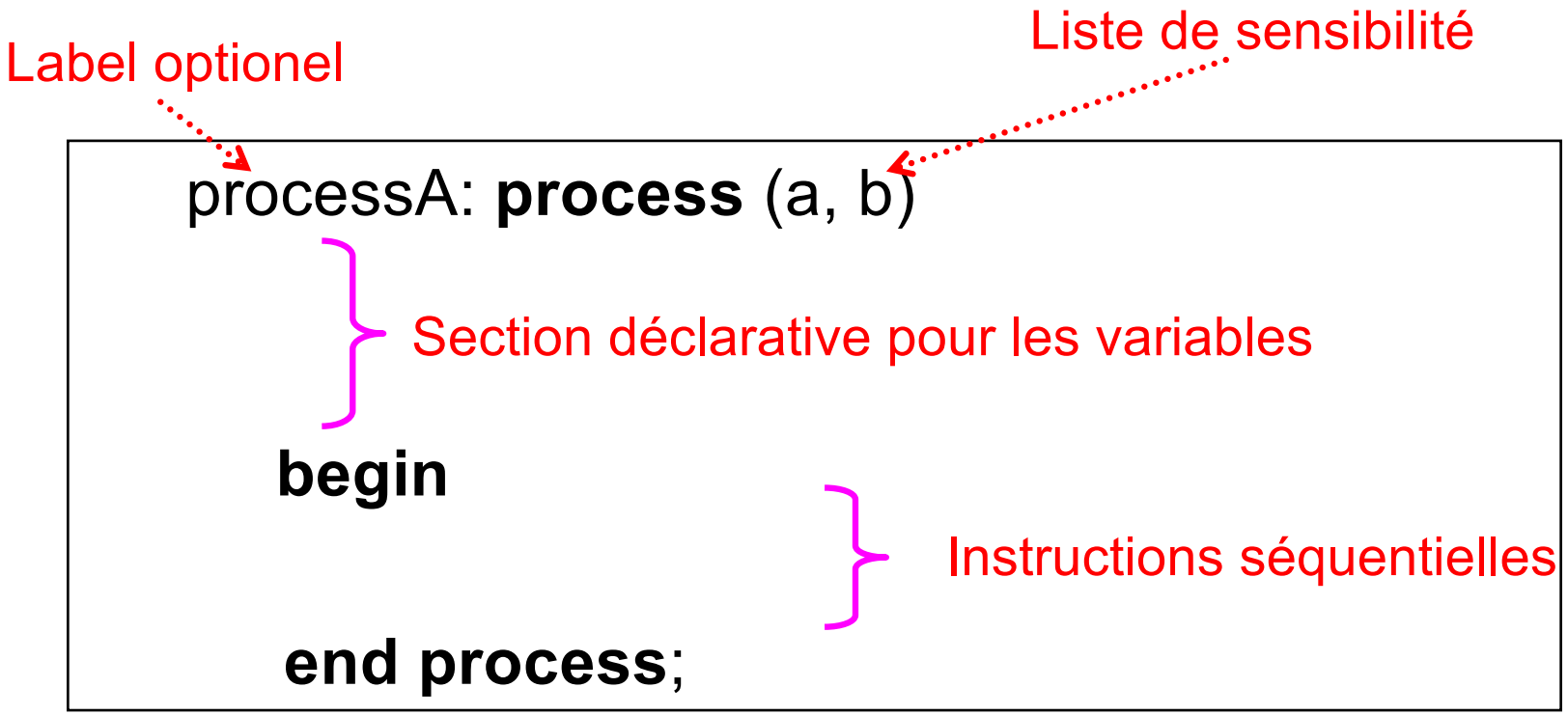
… instanciation d'un composant

- Cas d'une sortie non utilisée (unconnected):

```
architecture Struct of Exemple is

    component PORTE_ET is
        port(A_i, B_i : in std_logic;
             Z_o      : out std_logic
        );
    end component;

    for all: PORTE_ET use entity work.PORTE_ET(Behav);
    signal Signal_s: std_logic;
begin

    U1:PORTE_ET port map(
        A_i => Entree_i,
        B_i => Signal_s,
        Z_o => open
    );

end Struct;
```

Process

- Un process est une instruction concurrente permettant d'encapsuler des algorithmes.
- Il est défini par une suite d'instructions séquentielles, qui s'exécutent dans l'ordre qu'elles sont écrites.



Process (2)

- Le process
  - s'active lorsqu'un ou plusieurs signaux de sa liste de sensibilité changent de valeur.
  - se rendort lorsque toutes les instructions séquentielles (avant le **end process / wait)** ont été évaluées.

Process (3)

- Pour les besoins d'un algorithme, on peut déclarer des variables dans la zone déclarative du process.
- L'affectation se fait avec := et elle a lieu immédiatement.
- Exemple:

```
process (a)
   variable heu: integer range 0 to 15; -- déclaration de heu
begin
   heu := a + 1;  -- toujours initialisé une variable au début
   if heu > 12 then
      heu := 0;
   end if;
   sortie <= heu; -- rendre la variable visible à
                  --  l'extérieur à la fin du process
end process;
```

NB: Les variables devraient calculer des valeurs combinatoires, pas être des mémoires/latch.

Instructions séquentielles dans le process

- Affectation:

```
y <= A;
```

- Affectation avec condition:

```
if condition1 then
    -- affectation1
elsif condition2 then
    -- affectation2
else
    -- affectation3
end if;
```

- Instruction de sélection:

```
case opcode is
    when X"00" => add;
    when X"01" => substract;
    when others => illegal_opcode;
end case;
```

- Instruction d'itération: loop, while, for…

Process (4)

- Dans un process, les signaux, qui ont été affectés, changent de valeur qu'une fois le process endormi, c'est-à-dire juste avant le **end process ou wait**!
- Exemples:

```
architecture basic of toto is
   signal c: std_logic;
begin
   process (a)
   begin
      c <= a;
      if c='1' then
         b <= '1';
      else
         b <= '0';
      end if;
    end process;
end basic;
```

```
architecture baba of tata is
begin
   process (d)
   begin
      f <= '0';
      f <= '1';
      …
   end process;
end baba;
```

Process (5)

- Un signal combinatoire doit être affecté au moins une fois dans le process quelles que soient les instructions séquentielles exécutées.
- Exemple:

```
latch: process (rst, b)
begin
    if rst = '1' then
        a <= '0';
    elsif b = '1' then
        a <= '1';
    end if;
end process latch;
```

Le signal a est combinatoire.

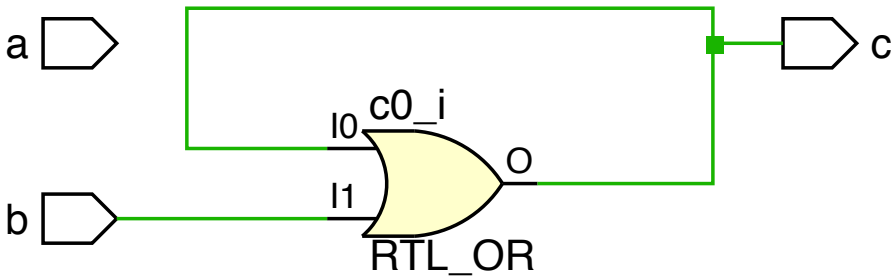
Quelle est le problème dans cette description?

Comment corriger?

Process (6)

- Un signal ne peut être en entrée et en sortie du processus (utilisé et affecté).

```
mauvais_exemple: process (a, b)
begin
        a <= a or b;
end process mauvais_exemple;

c <= a;
```

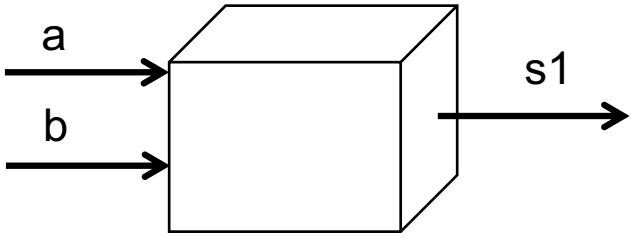


Résultat:
Circuit asynchrone

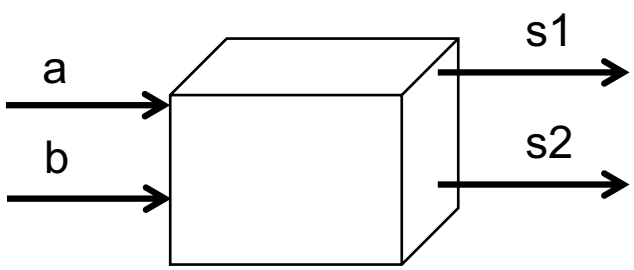
Process (7)

- Toutes les affectations d'un signal doivent être effectuées que dans 1 seule et unique instruction concurrente (process ou affectation concurrente).

```
s1 <=  a  or  b;
```




```
process (a, b)
begin
   s2 <= a;
   s1 <= b;
end process;
```



s1 affecté dans
2 instructions concurrentes
→ ERREUR

Types supplémentaires et Conversions

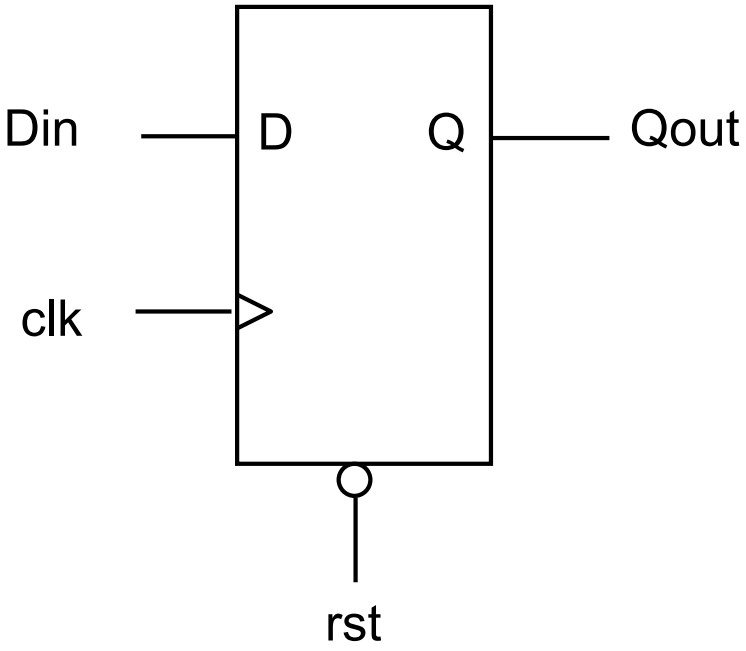
- Le package Standard définit les types usuels:
  - Boolean: énuméré des valeurs False et True
  - Integer: nombre signé en complément à 2 sur 32/64 bits.
- Le package Numeric_Std implémente deux types basés sur des array de std_logic pour la manipulation de nombres:
  - Unsigned (msb downto lsb): nombre binaire pur (positif)
  - Signed(msb downto lsb): nombre signé en complément à 2.
- VHDL étant très typé, il est nécessaire parfois de convertir des valeurs dans d'autres types:

std_logic_vector -> unsigned	unsigned(*arg*)
std_logic_vector -> signed	signed(*arg*)
unsigned, signed -> std_logic_vector	std_logic_vector(*arg*)
integer -> unsigned	to_unsigned(*arg*,*size*)
integer -> signed	to_signed(*arg*,*size*)
unsigned, signed -> integer	to_integer(*arg*)

La bascule D en VHDL

- Pour génèrer des bascules D, il faut déclarer un process dont la structure du code doit être exactement de la forme suivante:

```
PROCESS (clk, rst)
begin
   if rst = '0' then -- rst actif à l'etat bas
      Qout <= '0'; --initialisation asynchrone
   elsif rising_edge(clk) then
                              -- si flanc montant
      Qout <= Din;


   end if ;
end process ;
```



- La condition clk'EVENT and clk = '1' est synonyme de rising_edge(clk)
- Toutes affectations après le rising_edge produisent des bascules D.

Processus séquentiel

- La description d'un processus séquentiel doit suivre les règles suivantes:
  - La liste de sensibilité ne contient que clk et rst
  - Toutes les affectations sont faites après

    if rst = '1' **then**

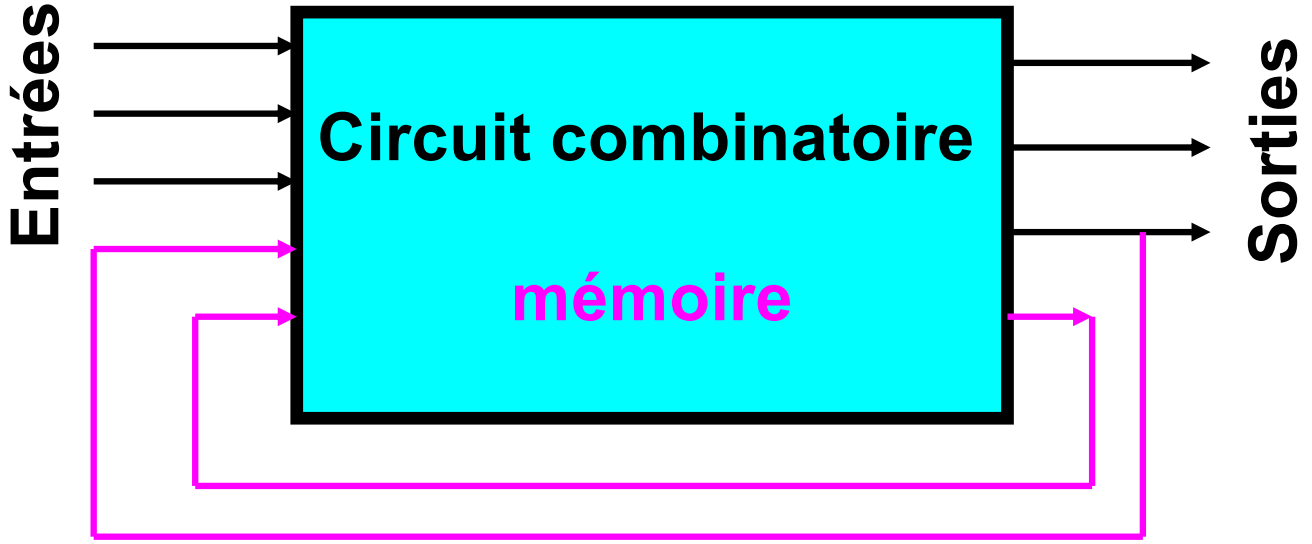
    Et après

    elsif rising_edge(clk) **then**

    Et nulle part ailleurs.
  - Les valeurs affectées au reset doivent être constantes.
  - Les conditions booléennes de rst = '1' ou rising_edge(clk) n'ont pas le droit d'être complétées avec d'autres conditions booléennes.
  - Le bloc rising_edge se termine toujours avec le **end if** et il ne peut pas y avoir d'instructions après ce bloc.

Les systèmes séquentiels

- Un système séquentiel est un circuit dont les sorties dépendent de l'état des entrées et des états précédents du système (passé).



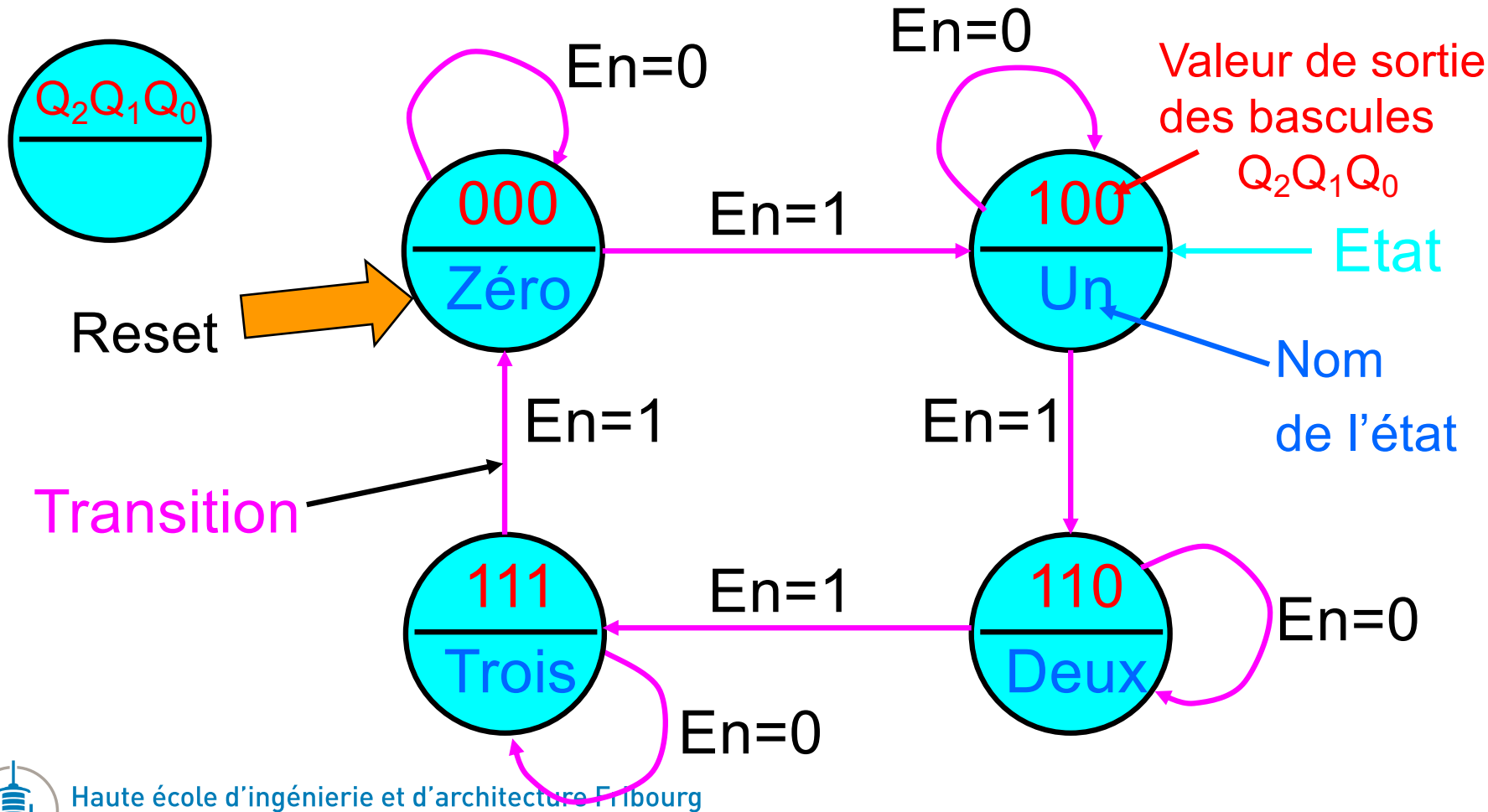
- La mémoire stocke l'état présent qui résume le passé.

Machines d'états

- Les machines d'états finies (Finite State Machine) est une abstraction très utile pour la conception de circuits numériques
- Les FSM permettent de représenter graphiquement le fonctionnement d'un système séquentiel.
- L'état présent et la combinaison des entrées déterminent les sorties ainsi que l'état futur

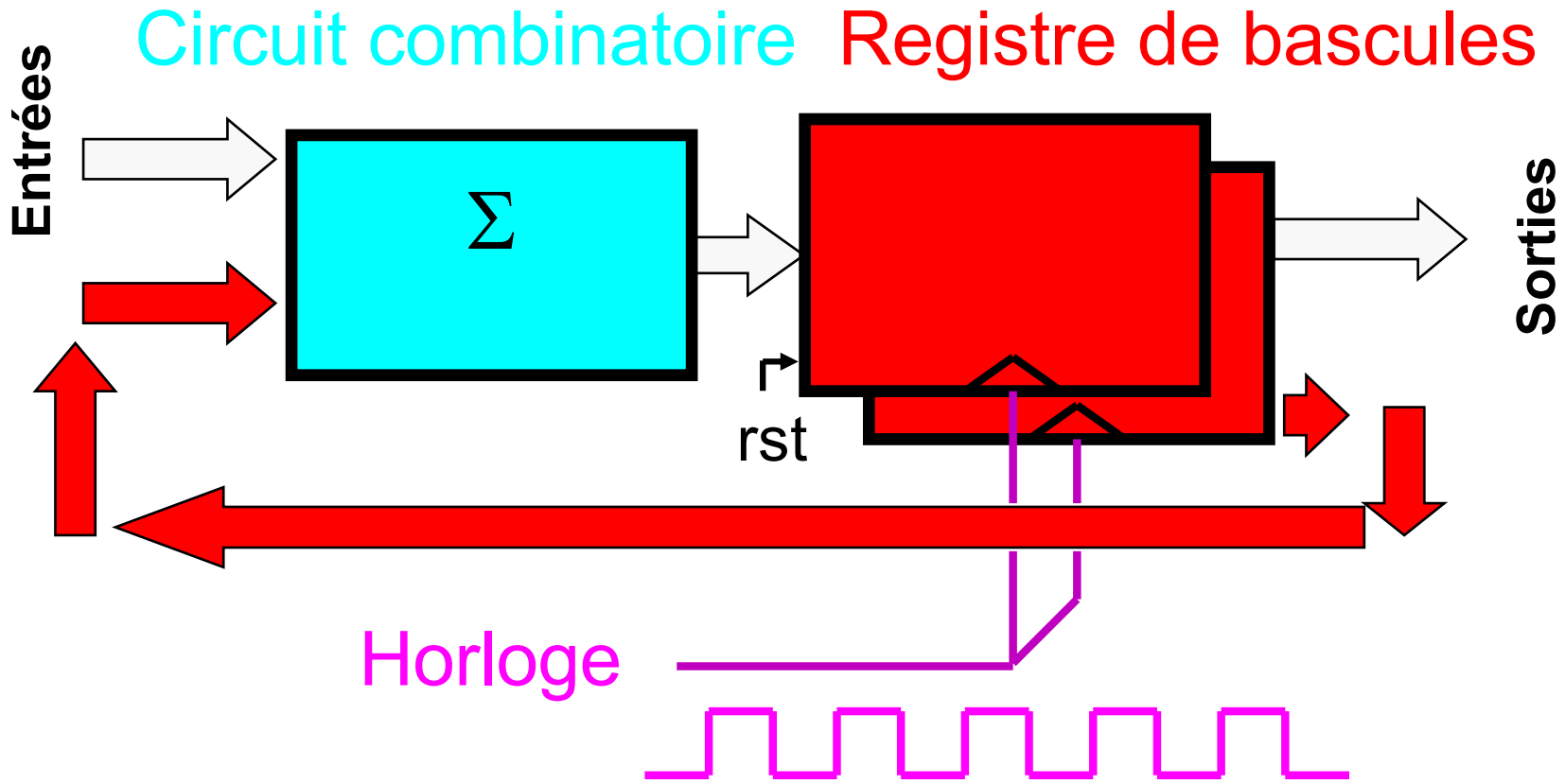
Diagramme d'états d'un compteur

- Le diagramme d'états décrit graphiquement le fonctionnement d'un circuit synchrone.



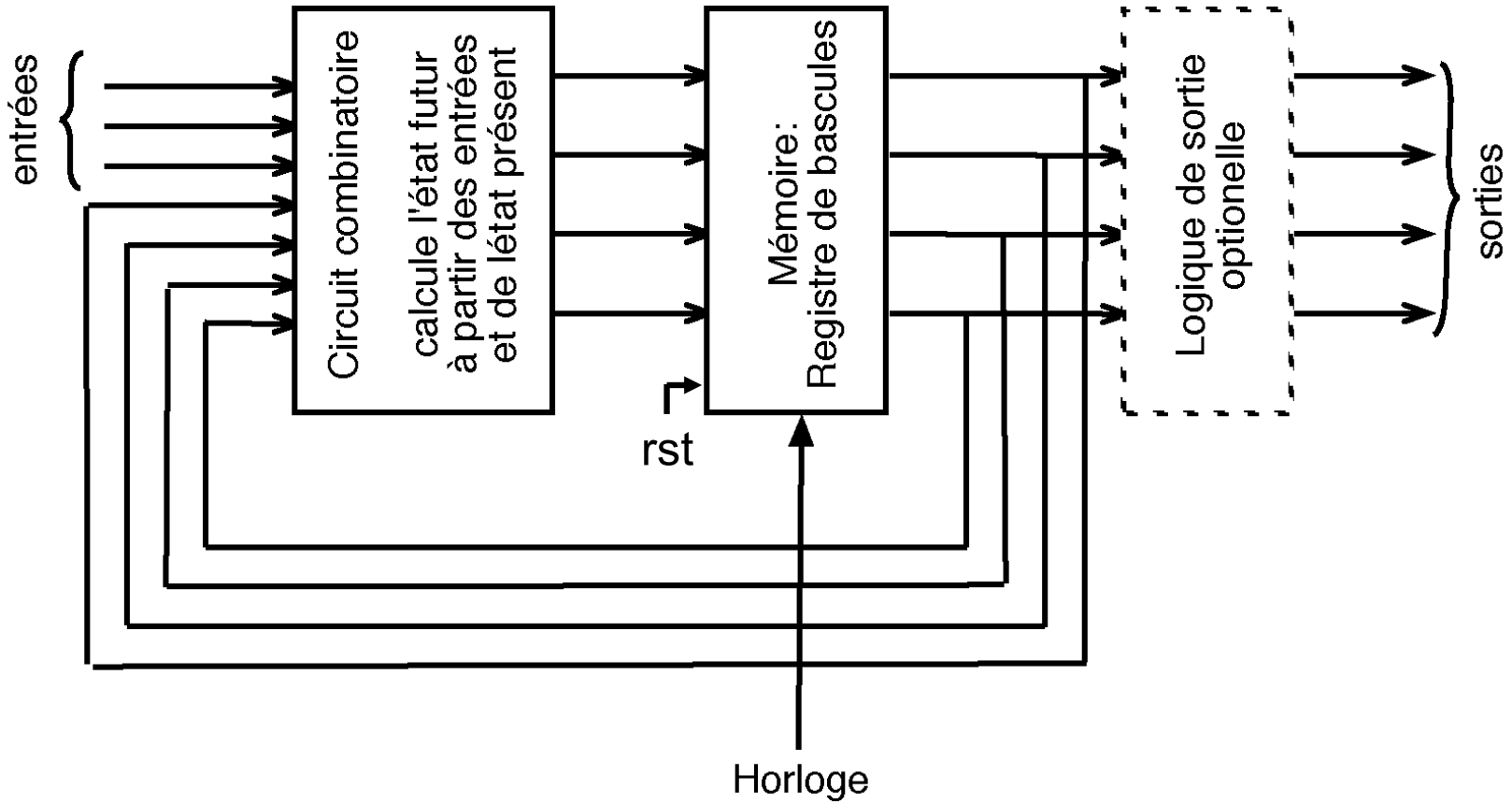
Circuit séquentiel synchrone

- Un circuit séquentiel synchrone est constitué de 2 blocs principaux:



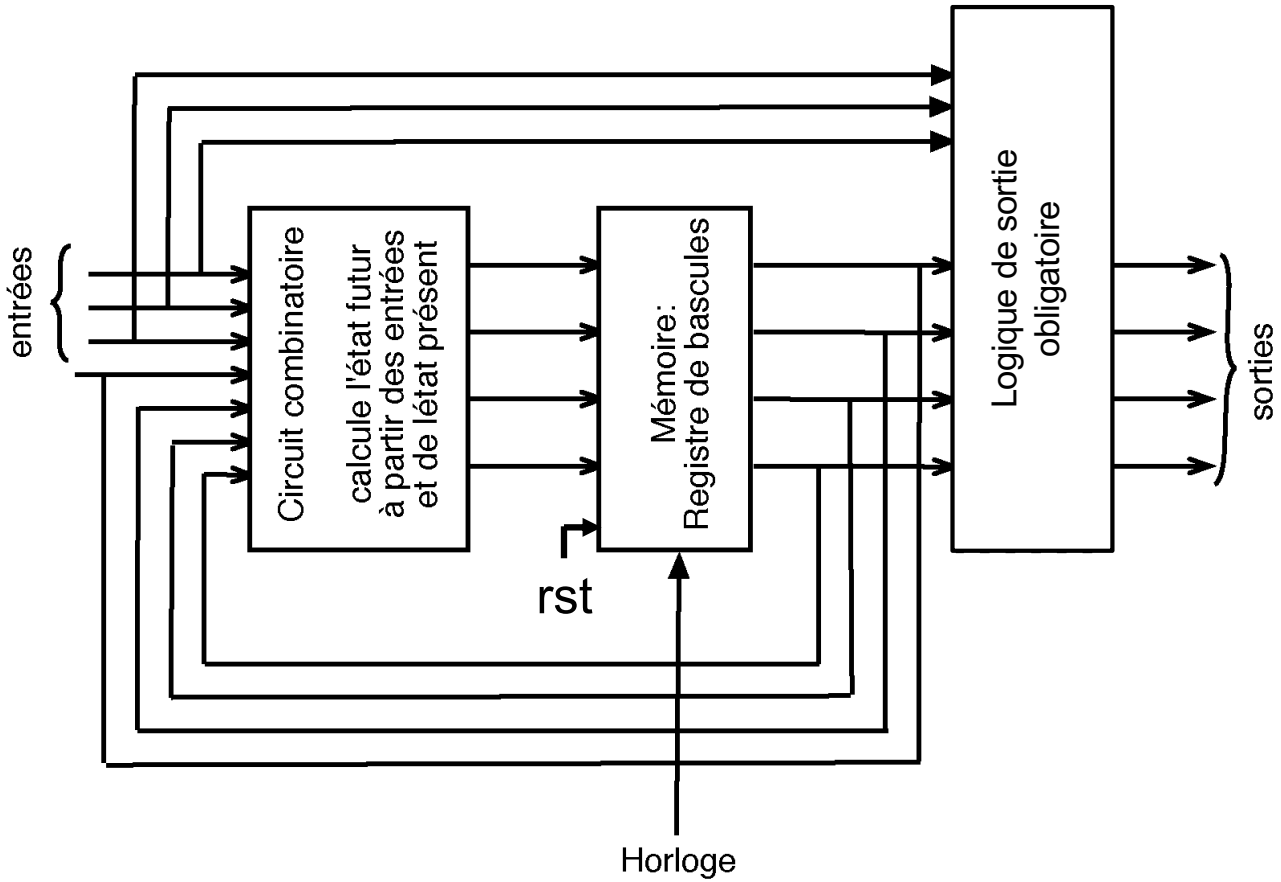
Circuit séquentiel synchrone (2)

- La machine de Moore: les sorties dépendent uniquement du contenu du registre de bascules → Sorties = F(état présent)



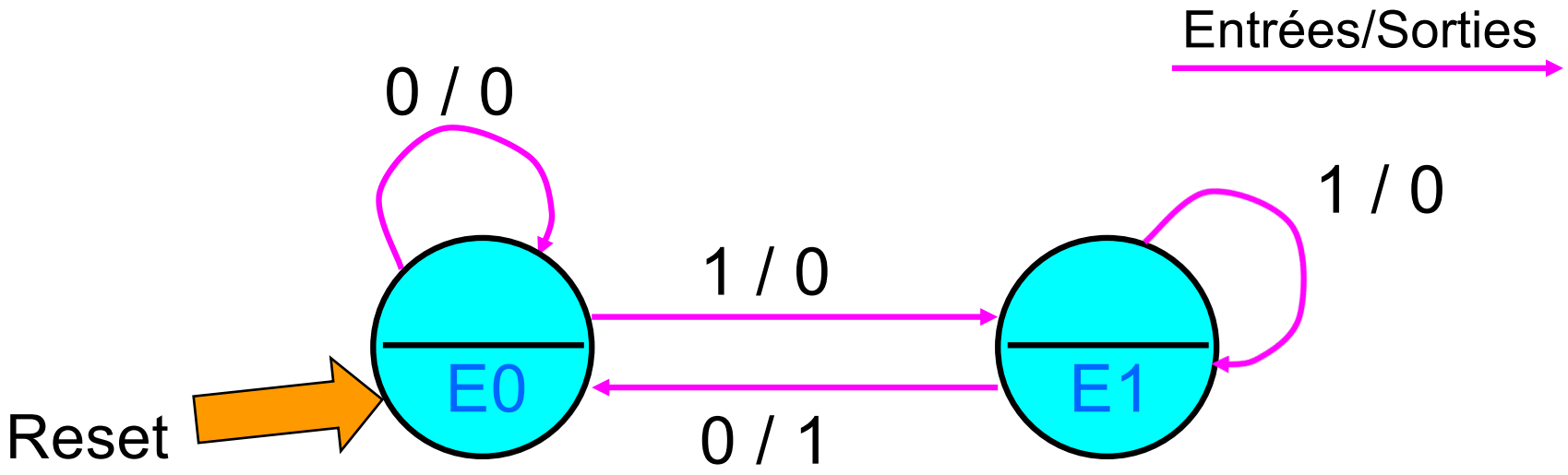
Circuit séquentiel synchrone (3)

- La machine de Mealy: les sorties dépendent du contenu du registre de bascules et des entrées. → Sorties = F(état présent, entrées)



Graphe d'états de la Machine de Mealy

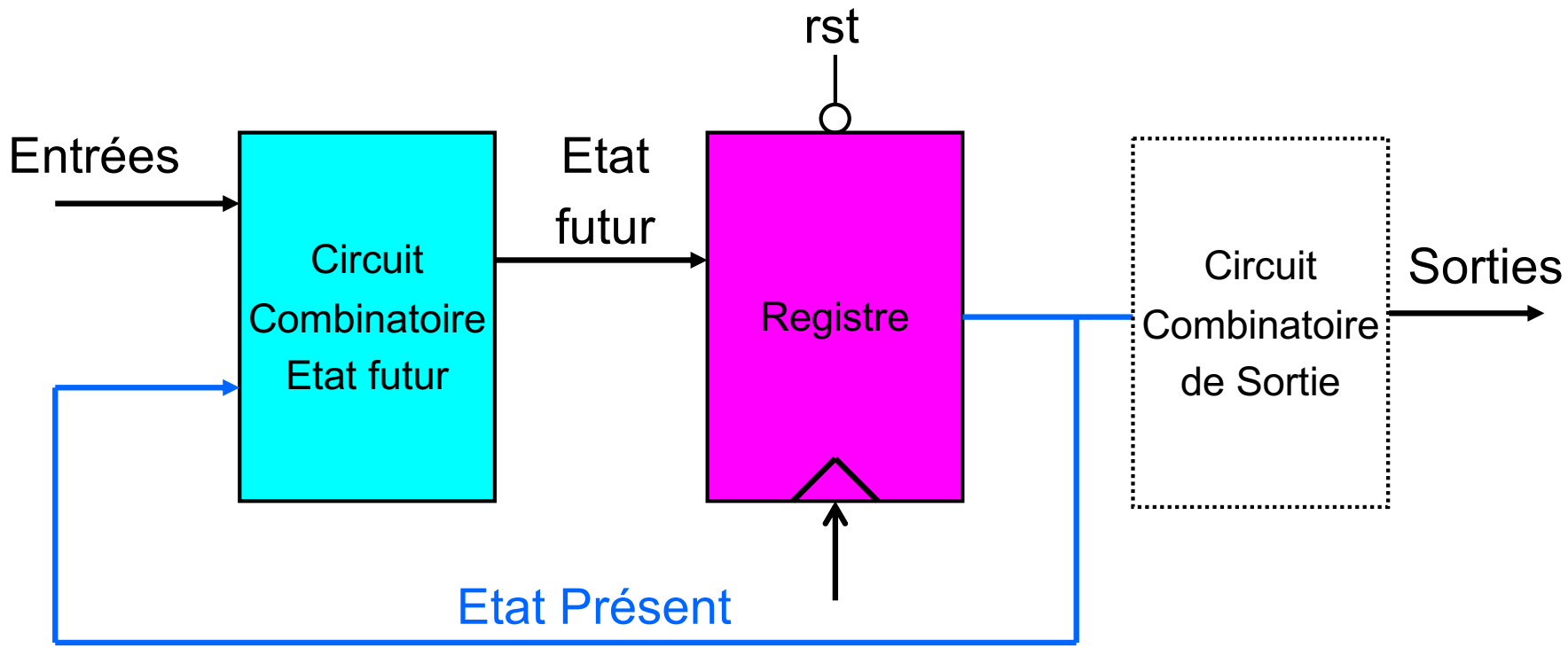
- Graphe d'états reconnaissant la séquence 10.



E0: Pas d'élément de la séquence observé

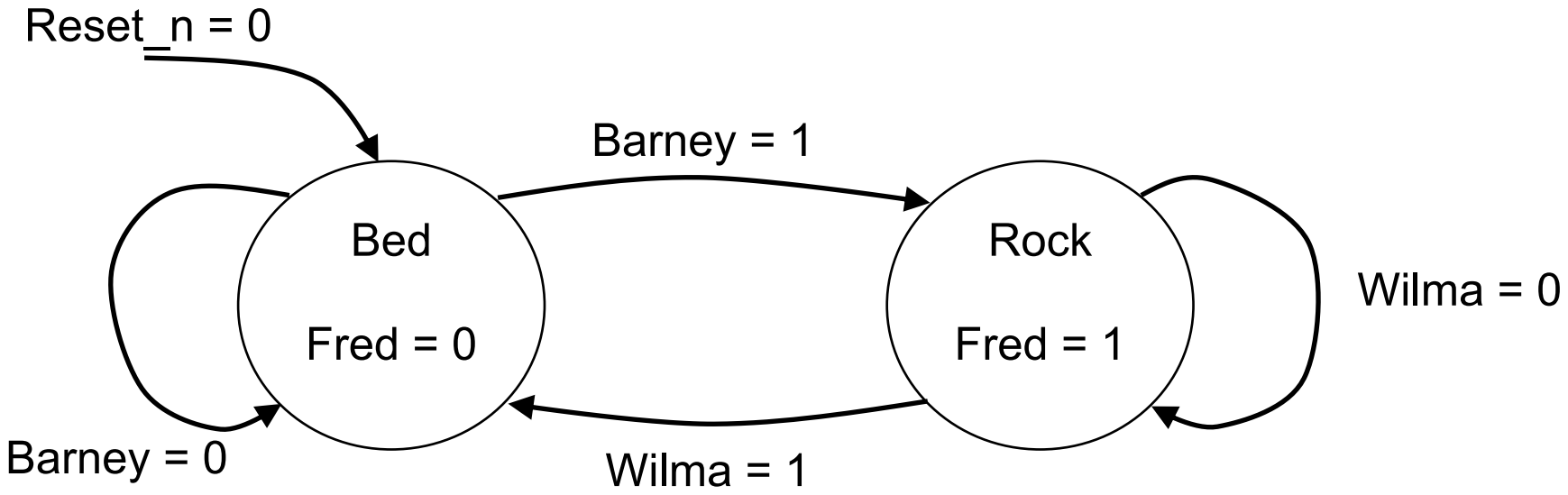
E1: « 1 » observé

Modèle VHDL des machines d'états



- Le schéma-bloc d'une machine d'états est constitué:
  - Un bloc combinatoire pour le calcul de l'état futur
  - Un registre pour mémoriser l'état actuel
  - Un bloc combinatoire de sortie qui affecte les sorties en fonction de l'état actuel.

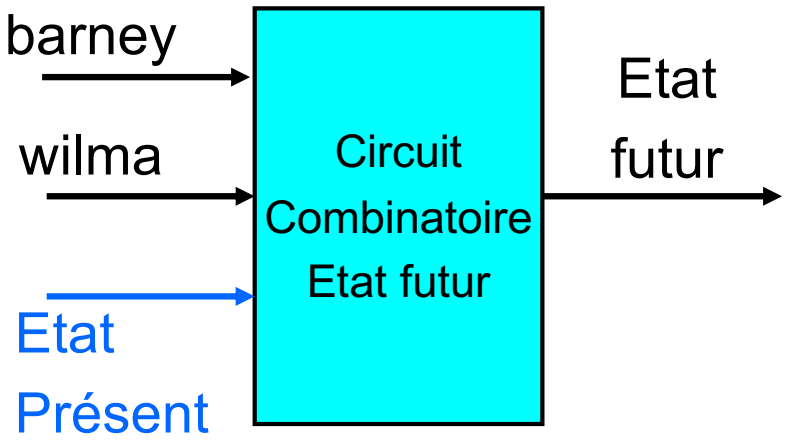
Modèle VHDL des machines d'états (2)



- En résumé:
  - 2 états: Bed, Rock
  - 2 entrées: Barney, Wilma
  - la sortie Fred qui est clairement liée à l'état
  - Le reset actif bas

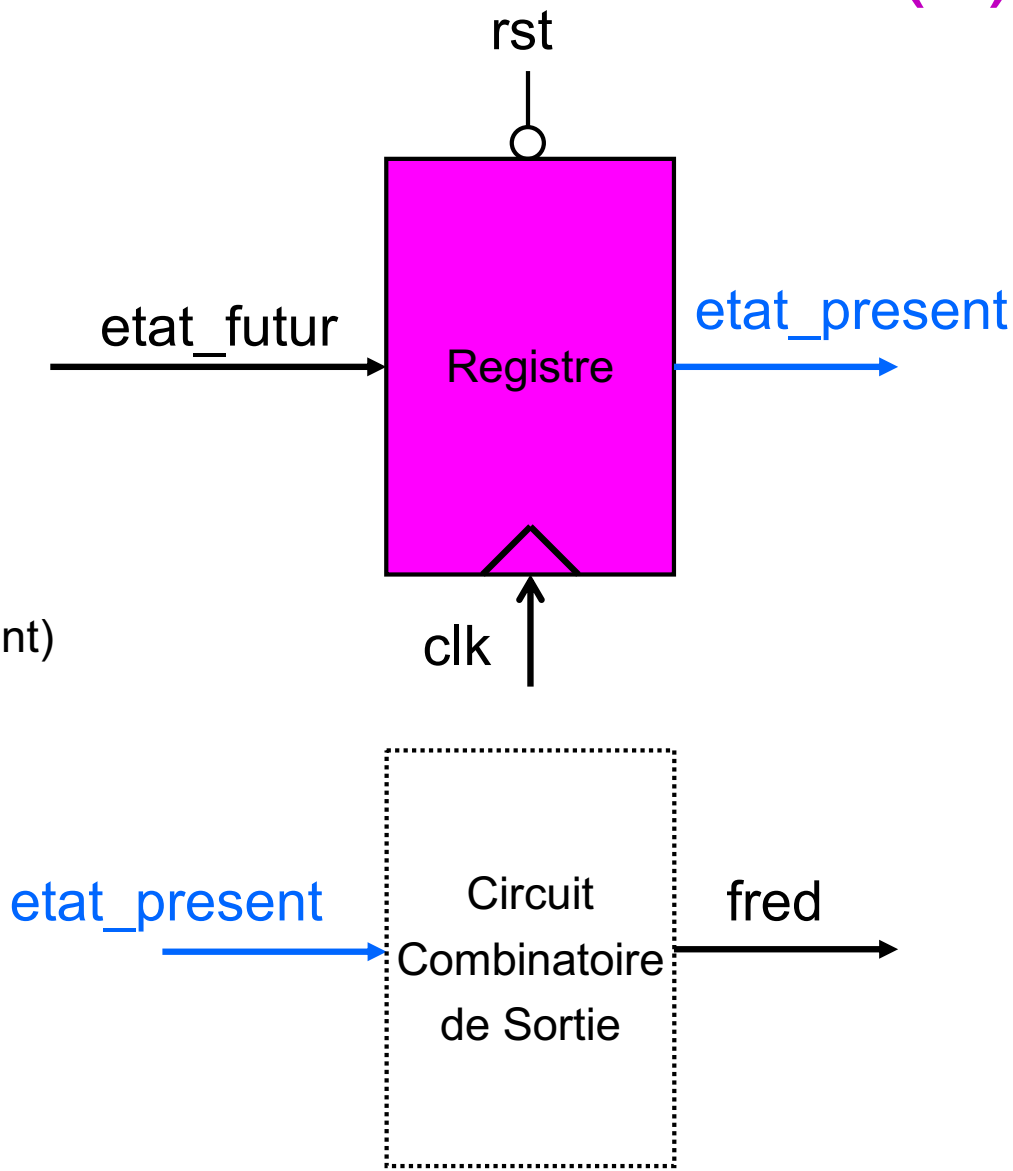
Modèle VHDL des machines d'états (3)

```
ARCHITECTURE fsm OF fsm_entity IS
   TYPE etats IS (Bed, Rock); -- déclaration des états
   SIGNAL etat_present, etat_futur : etats;
BEGIN
   combi_etat_futur : PROCESS (etat_present, barney, wilma)
   BEGIN
    CASE etat_present IS
     WHEN Bed =>
            IF Barney = '1' THEN
              etat_futur <= Rock;
            ELSE
              etat_futur <= Bed;
            END IF;
     WHEN Rock =>
            IF Wilma = '1' THEN
              etat_futur <= Bed;
            ELSE
              etat_futur <= Rock;
            END IF;
     WHEN others => etat_futur <= Bed; -- traitement états parasites
    END CASE;
   END PROCESS combi_etat_futur ;
```



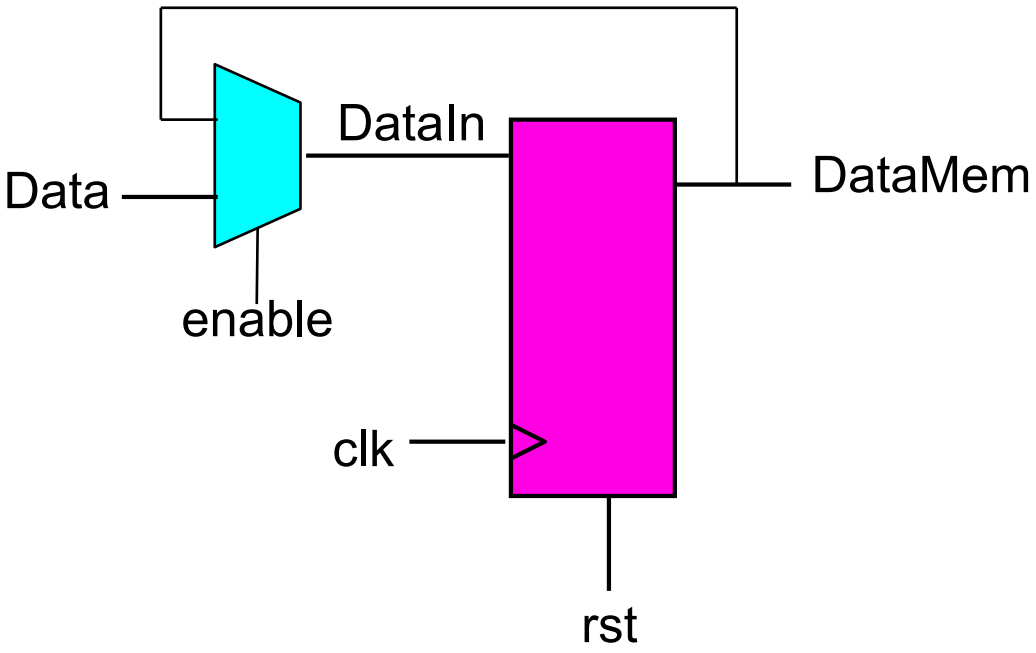
Modèle VHDL des machines d'états (4)

```
registre : PROCESS (clk, rst)
 BEGIN
  IF rst = '0' THEN
    etat_present <= Bed;
  ELSIF clk'event and clk = '1' THEN
    etat_present <= etat_futur;
  END IF;
 END PROCESS registre;

circuit_sortie : PROCESS (etat_present)
 BEGIN
  IF etat_present = Bed THEN
    fred <= '0';
  ELSE
    fred <= '1';
  END IF;
 END PROCESS circuit_sortie;
END ARCHITECTURE;
```



La mémorisation en VHDL

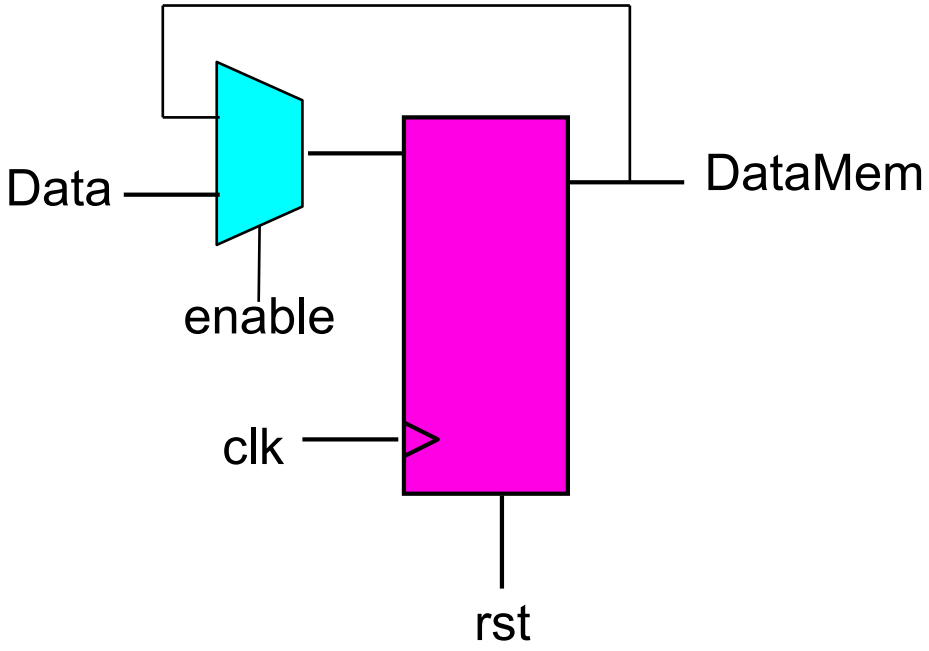
- La mémorisation d'une information est réalisée avec un multiplexeur et un registre.
- Si enable vaut '0', la valeur mémorisée actuelle est maintenue.




```
DataIn <= Data when enable = '1'
                else DataMem;

solution1:PROCESS (clk, rst)
begin
   if rst = '1' then
      DataMem <= (others => '0');
   elsif rising_edge(clk) then
      DataMem <= DataIn;
   end if;
end process;
```

La mémorisation en VHDL (2)

- Solution 2: la logique combinatoire (multiplexeur) est encapsulée dans le process du registre:




```
solution2:PROCESS (clk, rst)
begin
    if rst = '1' then
        DataMem <= (others => '0');
    elsif rising_edge(clk) then
        if enable = '1' then
            DataMem <= Data;
        end if;
    end if;
end process;
```

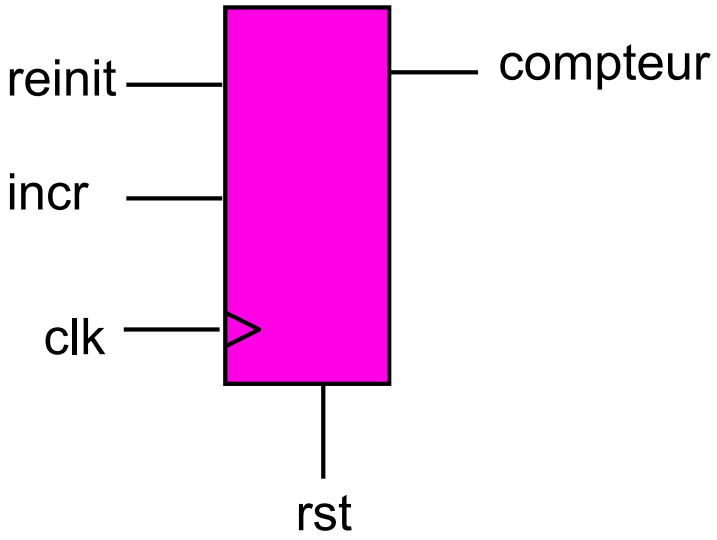
Pas besoin de else,
la valeur du signal DataMem est
implicitement conservée si le signal n'est pas affecté

Le compteur en VHDL

- Le compteur est utilisé pour implémenter des générateurs de séquence, diviseur de fréquence, produire des délais, calculer le temps, …

Compteur de 1 à 12 qui peut:

- Incrémenter
- réinitialiser la valeur à 1 de manière asynchrone et synchrone




```
ARCHITECTURE sol OF compteur1_12 IS
   SIGNAL compteur : unsigned(3 downto 0);
BEGIN
   count:PROCESS (clk, rst)
   begin
      if rst = '1' then
         compteur <= "0001";
      elsif rising_edge(clk) then
          if reinit = '1' then
            compteur <= "0001";
          elsif incr = '1' then
            if compteur = "1100" then
               compteur <= "0001";
            else
               compteur <= compteur + "0001";
            end if;
          end if;
      end if;
   end process;
```

Les éléments du langage

- Instructions VHDL terminées par le séparateurs «;»
- Commentaires débutent par deux tirets «--», se terminent avec la ligne

```
-- exemple instruction terminée par ;

Sortie_o <= (A_i or not(B_i)) and C_i;
```

- Identificateurs:
  - Doivent commencer par une lettre
  - Comportent des lettres (minuscules ou majuscule), des chiffres et des soulignés (underscore)
  - Un seul souligné (underscore) de suite

NB: Le VHDL ne distingue pas les majuscules des minuscules, il n'est pas case sensitive.

Objet

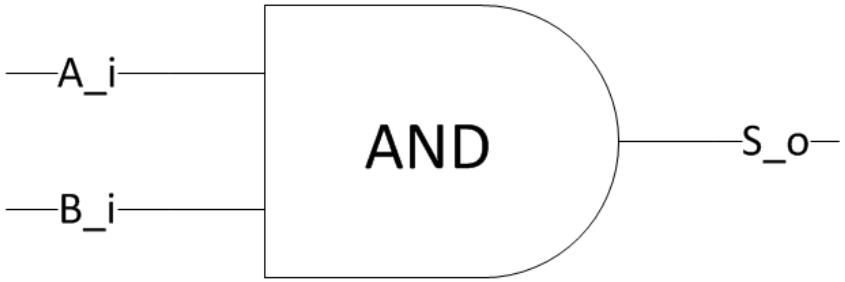
- Les objets: informations manipulées par le langage VHDL
  - Constante: objet de valeurs fixes, initialisées lors de leur déclaration
  - Variable: ce sont des objets appartement au monde du logiciel (soft)
  - Signal: ce sont des objets appartenant au monde matériel (hard)
  - Fichier: objets servant à stocker ou lire des valeurs du même type
- La portée de ces objets dépend directement de l'emplacement où ils sont déclarés:
  - Package: valable dans toutes les descriptions VHDL utilisant le package
  - Entity: valable à toutes les architectures associées à cet entité
  - Architecture: valable à toutes les instructions de cet même architecture
  - Process: Valable uniquement à l'intérieur du process

Conclusion

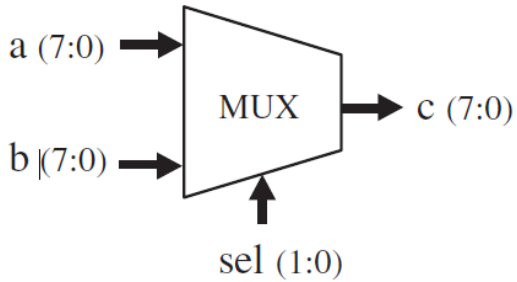
- VHDL est un langage qui permet de décrire et simuler des circuits numériques plus ou moins complexes
- Il faut réfléchir «circuit» et interpréter les instructions du code comme des blocs du circuit numérique
- Deux types d'instructions: concurrentes et séquentielles
- Dans l'entité, utilisez uniquement le type std_logic ou std_logic_vector
- Décomposition hiérarchique facilite grandement la conception de circuit numérique complexe

Exercices

1) Ecrivez le code VHDL correspondant à une simple porte AND



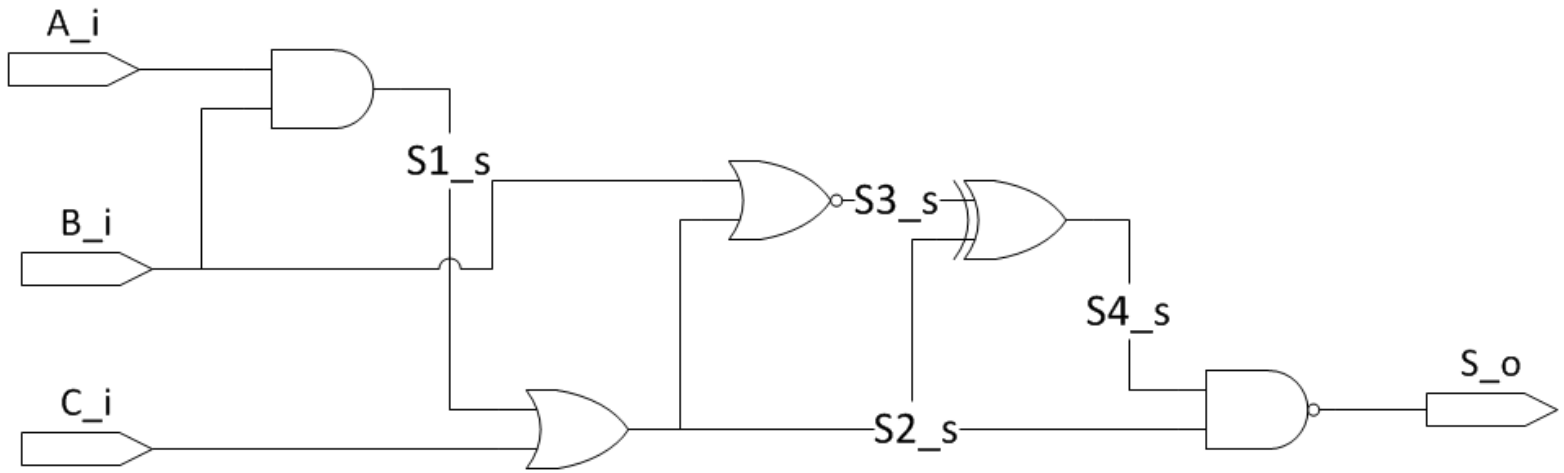
2) Modéliser un mux à 2 entrées:



sel	c
00	0
01	a
10	b
11	Z

Exercice 3

- Ecrire le composant VHDL correspondant à cette fonction logique simple
  1. Etablir la table de vérité de cette fonction logique
  2. Ecrire le code VHDL correspondant
  3. Tester et simuler le code correspondant à cette table



Exercice 4

- Développer en VHDL un compteur à 2 chiffres BCD qui a les fonctions suivantes:
  - Compte de 00 à 59
  - Si l'entrée `incr` vaut '1', le compteur prend la prochaine valeur et passe automatiquement de 59 à 00; Lors du passage de 59 à 00, la sortie `cycle` vaut 1 pendant un cycle d'horloge.
  - Si l'entrée `raz` vaut '1', le compteur est réinitialisé à 00
  - Si l'entrée `set` vaut '1', le compteur prend la valeur de l'entrée `val_i`
  - La sortie `compteur_o` prend la valeur du compteur.

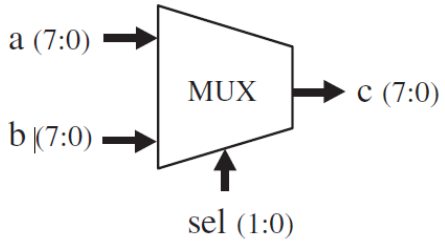
Exercice 5

- Développer en VHDL un système complet de clignotants avec feux de détresse pour une voiture:
  - Dessiner un schéma bloc du système
  - Développer chaque bloc
  - Tester le bon fonctionnement du système complet

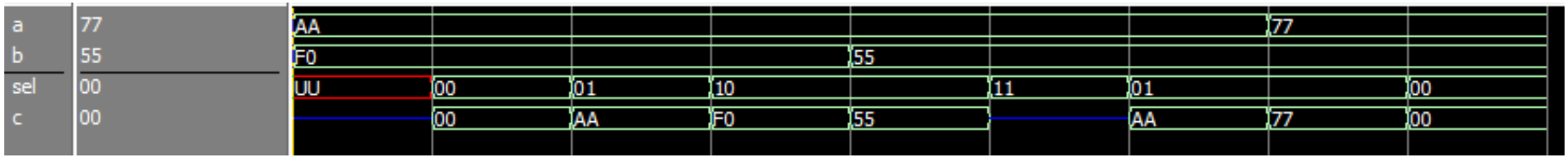
Solution:

2. Modéliser un Mux 2 entrées

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

entity mux is
  port(a,b: IN std_logic_vector(7 downto 0);
       sel: IN std_logic_vector(1 downto 0);
       c: OUT std_logic_vector(7 downto 0)
  );
end mux;

architecture example of mux is
begin
  process(a,b,sel)
  begin
    if (sel = "00") then
      c <= "00000000";
    elsif (sel = "01") then
      c <= a;
    elsif (sel = "10") then
      c <= b;
    else
      c <= (others => 'Z');
    end if;
  end process;
end example;
```



sel	c
00	0
01	a
10	b
11	Z

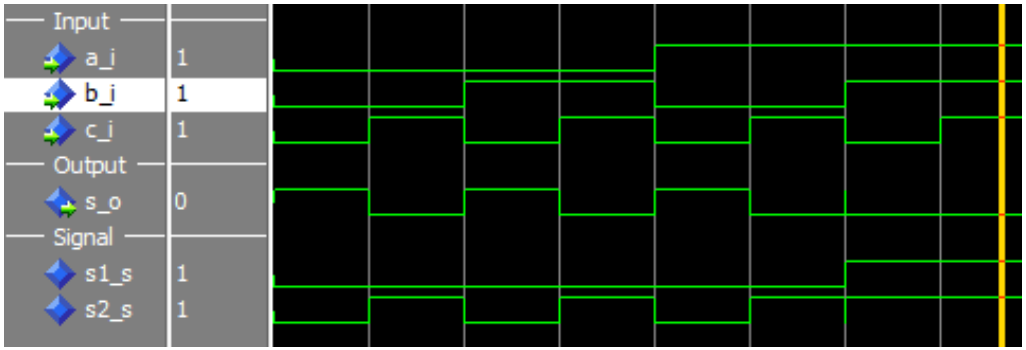


Solution 3

- Table de vérité

a_i	b_i	c_i	s1_s	s2_s	s3_s	s4_s	s_o
0	0	0	0	0	1	1	1
0	0	1	0	1	0	1	0
0	1	0	0	0	0	0	1
0	1	1	0	1	0	1	0
1	0	0	0	0	1	1	1
1	0	1	0	1	0	1	0
1	1	0	1	1	0	1	0
1	1	1	1	1	0	1	0

- Simulation



- Code VHDL

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

entity exo3_ch1 is
  port(a_i : IN std_logic;
       b_i : IN std_logic;
       c_i : IN std_logic;
       s_o : out std_logic
     );
end entity;

architecture rtl of exo3_ch1 is
  signal s1_s, s2_s, s3_s, s4_s : std_logic;
begin

  s1_s <= a_i AND b_i;
  s2_s <= s1_s OR c_i;
  s3_s <= s2_s NOR b_i;
  s4_s <= s2_s XOR s3_s;
  s_o <= s4_s NAND s2_s;

end architecture;
```

Solution 4

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Compteur2BCD is
    Port ( clk, rst, incr, raz, set : in STD_LOGIC; cycle : out STD_LOGIC;
           val_i : in STD_LOGIC_VECTOR(7 downto 0);
           compteur_o : out STD_LOGIC_VECTOR(7 downto 0));
end Compteur2BCD;
architecture Behavioral of Compteur2BCD is
    signal compteur_mem_s : unsigned(7 downto 0);
    signal unite_neuf, dizaine_cinq : std_logic;
begin
    compteur_o <= std_logic_vector(compteur_mem_s);
    unite_neuf <= '1' when compteur_mem_s(3 downto 0) = "1001" else '0';
    dizaine_cinq <= '1' when compteur_mem_s(7 downto 4) = "0101" else '0';
    cycle <= unite_neuf and dizaine_cinq and incr;
```

Solution 4 (Suite)

```
compteur: process(clk, rst) is
    begin
        if rst = '1' then
            compteur_mem_s <= (others => '0'); -- init asynchrone à 0
        elsif rising_edge(clk) then
            if raz = '1' then
                compteur_mem_s <= (others => '0'); -- init synchrone à 0
            elsif set = '1' then
                compteur_mem_s <= unsigned(val_i);
            elsif incr = '1' then
                if unite_neuf = '1' then
                    compteur_mem_s(3 downto 0) <= (others => '0');
                    if dizaine_cinq = '1' then
                         compteur_mem_s(7 downto 4) <= (others => '0');
                    else
                        compteur_mem_s(7 downto 4) <= compteur_mem_s(7 downto 4) + 1;
                    end if;
                else
                    compteur_mem_s(3 downto 0) <= compteur_mem_s(3 downto 0) + 1;
                end if;
            end if;
        end if;
   end process compteur;
end Behavioral;
```