Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

# Système numérique

Résumé

*Auteurs :*
*Marc* Roten

*Professeur :*
*Daniel* Oberson

27 septembre 2018

# Table des matières

# 1 Chapter 1 : Introduction au VHDL

## 1.1 Procédure

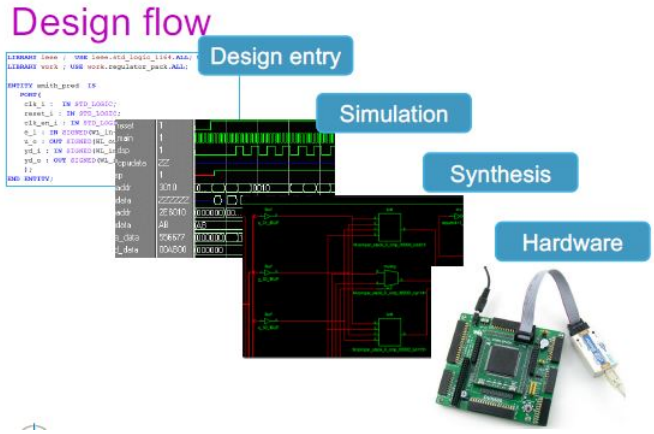



FIGURE 1 – procédure

## 1.2 Signaux types et opérateurs

- Le type std_logic est un énuméré à 9 valeurs:
  - 'U' non-initialisé n'existe pas dans le sillicium (uniquement dans le simulateur
  - 'X' inconnu fort
  - '0' logique 0 fort
  - '1' logique 1
  - 'Z' haute impédance utile dans le cas du tri-state (0 ou 1 ou Z)
  - 'W' inconnu faible resistance à l'alimentation pour limiter le courant (pas utile pour l'instant)
  - 'L' logique 0 faible (pull-down) (collecteur ouvert)
  - 'H' logique 1 faible (pull-up)
  - '-' état indifférent (don't care) ne jamais utiliser ces don't care, JAMAIS
- Pour les affectations, les valeurs surtout utilisées sont '0', '1' et 'Z'. et des fois le X en simulation

FIGURE 2 – Rappel des différents états, types et opérteurs

## 1.3 Std logic vector

- Type composé de std_logic: Déclaration d'un vecteur non-contraint

```
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*

FIGURE 3 – STD LOGIC VECTOR

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.

### 1.3.1 Les non-contraint

On peut réaliser des vecteur non-contraint, pour la réutilisabilité du code, mais lors de l'instanciation, il faut rajouter les dites-contraintes. Dans un FPGA, en VHDL, TOUT EST DETERMINISTE.

## 1.4 Ordre des opérateurs

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**

FIGURE 4 – ordre de précédence

Exemples ci-dessous :

$$A <= \text{"10111"}$$
$$A\ ssl\ 1 -> \text{"01110"}$$
$$A\ sra\ 2 -> \text{"11101"}$$
$$A\ rol\ 3 -> \text{"11101"}$$

## 1.5 Rappel structure du code

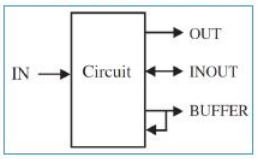
```
-- Librairie IEEE
library IEEE;
use IEEE.Std_Logic_1164.all; --Defini type Std_Logic
entity Exemple is
    port(Entree_i : in Std_Logic;
         Vecteur_i : in Std_Logic_Vector(3 downto 0);
         Sortie_o : out Std_Logic;
         BiDir_io : inout Std_Logic
        );
end Exemple;

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

FIGURE 5 – Structure du code

## 1.6 Description d'un composant, Entity

- 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.
- 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

FIGURE 6 – Description d'un conposant

## 1.7 Rappel le la syntaxe pour l'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;
```

FIGURE 7 – architecture

### 1.7.1 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

FIGURE 8 – zone de déclaration

### 1.7.2 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.

FIGURE 9 – Zone de code

## 1.8 Conception avec le VHDL

*IL FAUT PENSER CIRCUIT*

*Faire des descriptions SIMPLES et LISIBLE*

==========

==========

Rajouter du texte de la page 23

==========

==========

done jusqu'à la page 23

## 1.9 Portabilité

- 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.

FIGURE 10 – Zone de code

## 1.10 Outils d'instructions concurrentes.

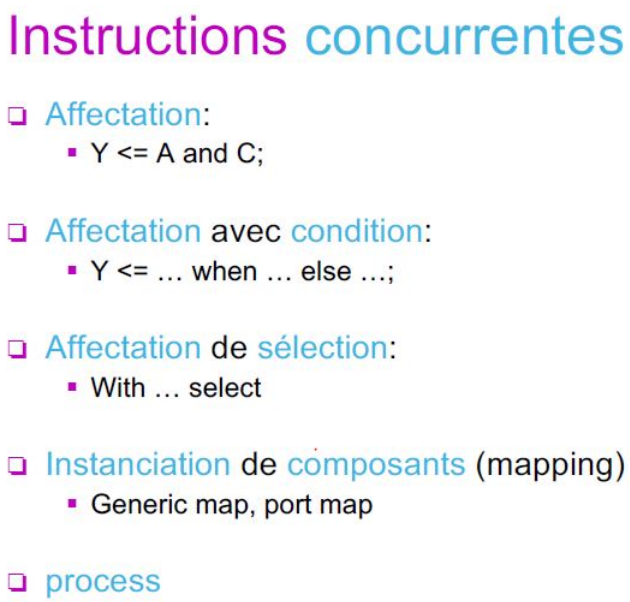



FIGURE 11 – Zone de code

### 1.10.1 Affectation

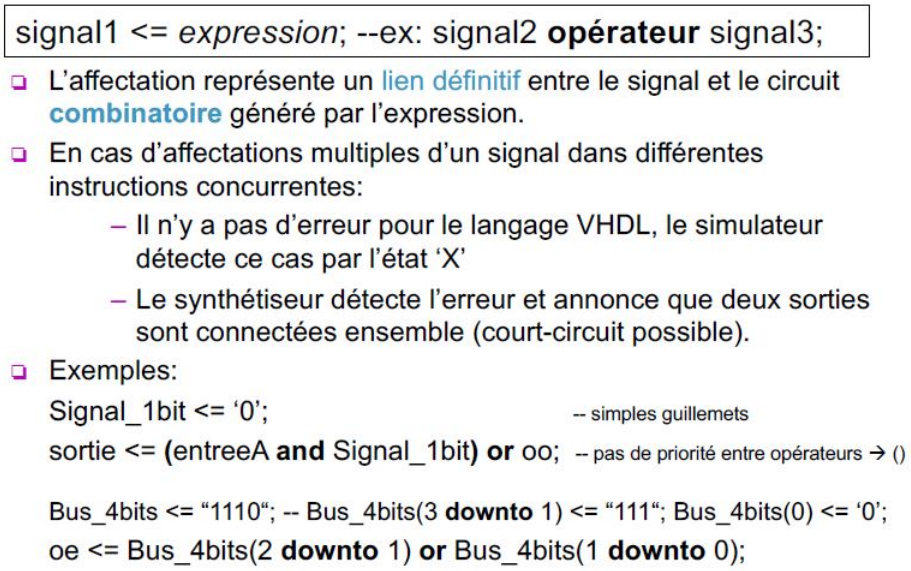



FIGURE 12 – Zone de code

En simulation, on peut affecter le meme signal à plusieurs circuit. La synthèse nous laisse uniquement faire 1 Signal sur 1 circuit

### 1.10.2 Affectation avec condition

L'avantage de cette méthode est qu'elle ne crée aucun latch, et en VHDL, il ne faut aucun latch.

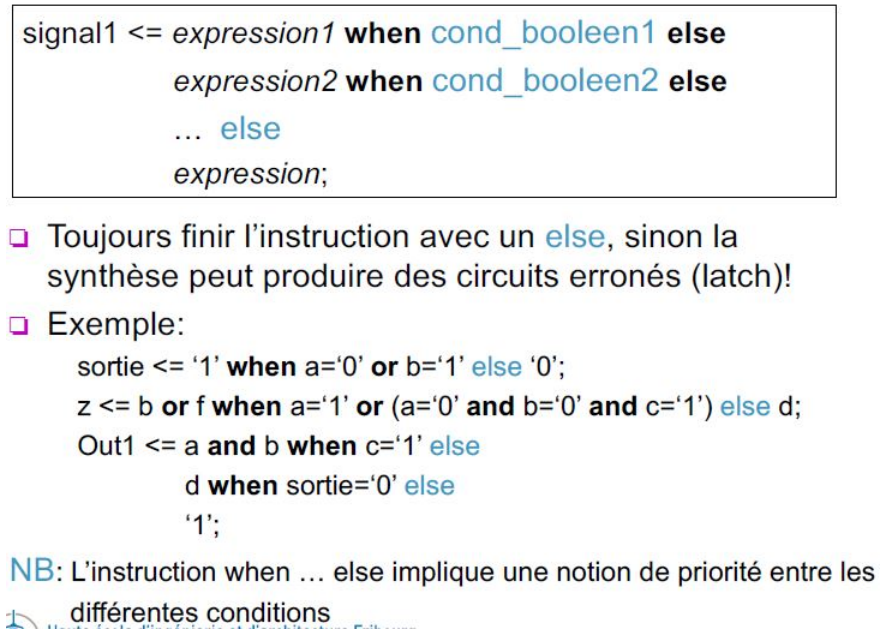



FIGURE 13 – Zone de code

### 1.10.3 Affectation de séléction

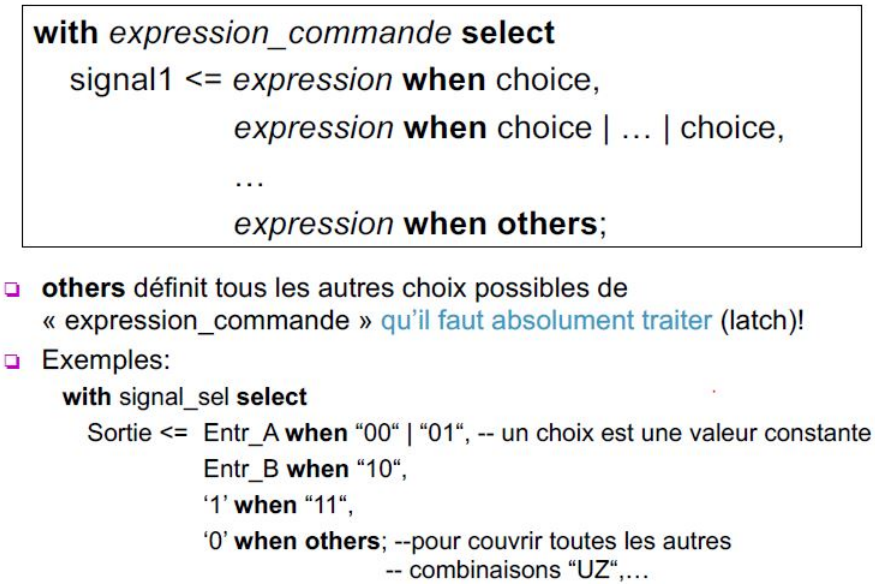



FIGURE 14 – Zone de code

## 1.11 Instanciation d'un composant

### 1.11.1 Methode Schroeter

- 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
);
```

FIGURE 15 – Zone de code

- 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;
```

FIGURE 16 – Lien avec une bibliothèque

### 1.11.2 Méthode Etudiand

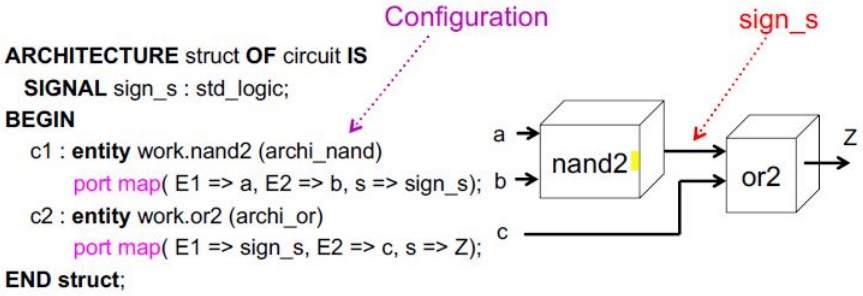



FIGURE 17 – Lien avec une bibliothèque

## 1.12 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.

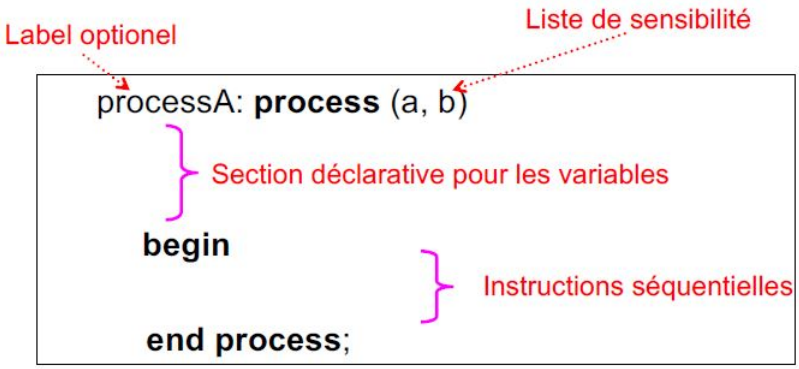



FIGURE 18 – Lien avec une bibliothèque

**Un seul bit d un std logic vector bouge, et on réveille toute la liste de sensibilité**

$serendortlorsquetouteslesinstructionsséquentielles(avantleendprocess/wait)ontétéévaluées.$

### 1.12.1 Déclaration à l'intérieur

$interditd'avoirunevariablesansvaleur, leurdonnerunevaleurinitiale$

- 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.

FIGURE 19 – Lien avec une bibliothèque

## 1.13 Instructions Séquentielles

- 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…

FIGURE 20

## 1.14 Types supplémentaires et conversion

**le languqge est fortement typé**

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:

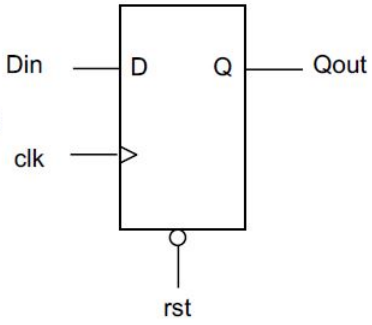
| | |
|---|---|
| std_logic_vector -> unsigned | unsigned(*arg*) |
| std_logic_vector -> signed | signed(*arg*) |
| unsigned, signed -> std_logic_vector | std_logic_vector(*arg*) |
| integer -> unsigned | to_unsigned(*arg*,*size*) |
| integer -> signed | to_signed(*arg*,*size*) |
| unsigned, signed ->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

FIGURE 21

## 1.15 Bascule D

- 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.

FIGURE 22

## 1.16 Final State Machine VHDL Model

### 1.16.1 Architecture

```
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 ;
```

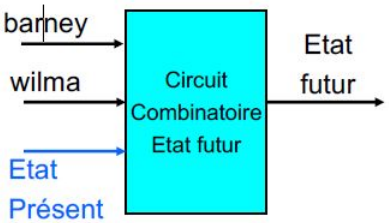



FIGURE 23

### 1.16.2 Registre

```
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;
```

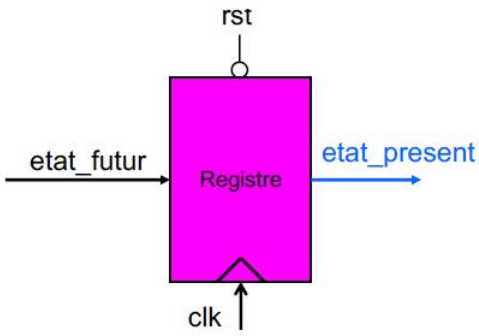



FIGURE 24

### 1.16.3 Circuit de sortie

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

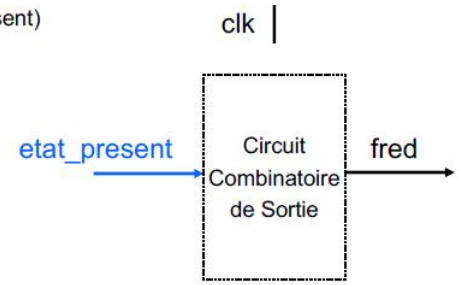
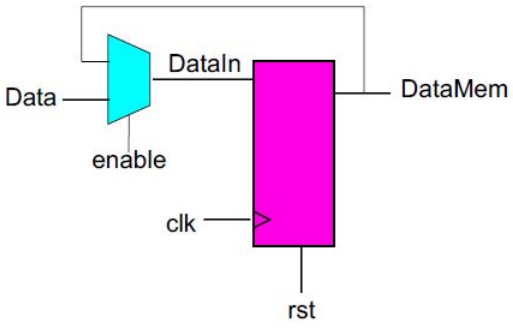



FIGURE 25

### 1.16.4 Mémoire

- 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;
```

FIGURE 26