



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

Systeme numérique

Résumé

Auteurs :
Marc ROTEN

Professeur :
Nicolas SCHROETER



26 octobre 2018

Table des matières

1 Chapter 1 : Introduction au VHDL	3
1.1 Procédure	3
1.2 Signaux types et opérateurs	3
1.3 Std logic vector	4
1.3.1 Les non-contraint	4
1.4 Ordre des opérateurs	4
1.5 Rappel structure du code	5
1.6 Description d'un composant, Entity	6
1.7 Rappel le la syntaxe pour l'architecture	6
1.7.1 Zone de Déclaration	7
1.7.2 Zone de code	7
1.8 Conception avec le VHDL	8
1.9 Portabilité	8
1.10 Outils d'instructions concurrentes.	9
1.10.1 Affectation	9
1.10.2 Affectation avec condition	10
1.10.3 Affectation de sélection	10
1.11 Instanciation d'un composant	11
1.11.1 Methode Schroeter	11
1.11.2 Méthode Etudiand	12
1.12 Process	12
1.12.1 Déclaration à l'intérieur	12
1.13 Instructions Séquentielles	13
1.14 Types supplémentaires et conversion	13
1.15 Bascule D	14
1.16 Final State Machine VHDL Model	15
1.16.1 Architecture	15

1.16.2	Registre	15
1.16.3	Circuit de sortie	16
1.16.4	Mémoire	16
1.17	IMPORTANT	17
1.17.1	Signal	17
1.17.2	Un combinatoire	17
1.17.3	Mémoire	17
1.17.4	Longue mémoire	17
1.17.5	RTL	17
1.18	Exercices	18
2	Chapter 2 Conception hiérarchique	20
2.1	Concept de découpage hiérarchique	20
2.2	Communication	21
2.3	A COMPLETER	21
3	Types, Opérateurs et Conversions	22
3.1	types scalaires entiers	22

1 Chapter 1 : Introduction au VHDL

1.1 Procédure

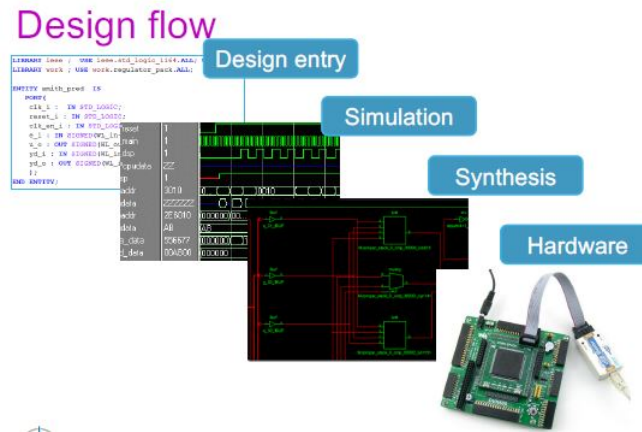


FIGURE 1 – procédure

1.2 Signaux types et opérateurs

- ❑ Le type `std_logic` est un énuméré à 9 valeurs:
 - **'U'** non-initialisé *n'existe pas dans le silicium (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 opérateurs

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 précedence:

- ❑ 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 précedence

Exemples ci-dessous :

```
A <= "10111"
A ssl 1- > "01110"
A sra 2- > "11101"
A rol 3- > "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;

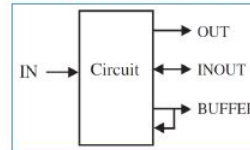
architecture Style_Description of Exemple is
--zone de déclaration
begin
    --Instructions concurrentes .....
    process (Liste_De_Sensibilité)
    begin
        --Instructions séquentielles .....
    end process;
end Style_Description;
```

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 composant

1.7 Rappel de 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 ou 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
 - Faire 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.

Instructions concurrentes

- ❑ **Affectation:**
 - $Y \leq A \text{ and } C;$
- ❑ **Affectation avec condition:**
 - $Y \leq \dots \text{ when } \dots \text{ else } \dots;$
- ❑ **Affectation de sélection:**
 - With ... select
- ❑ **Instanciation de composants (mapping)**
 - Generic map, port map
- ❑ **process**

FIGURE 11 – Zone de code

1.10.1 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);
```

FIGURE 12 – Zone de code

En simulation, on peut affecter le même signal à plusieurs circuits. 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.

```

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

FIGURE 13 – Zone de code

1.10.3 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",...
  
```

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(entree1: 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

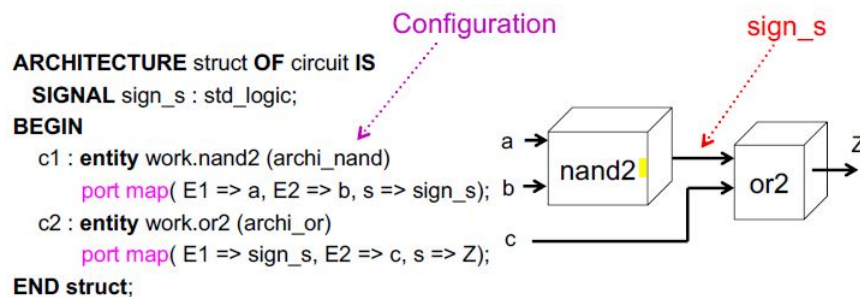


FIGURE 17 – Lien avec une bibliothèque

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.

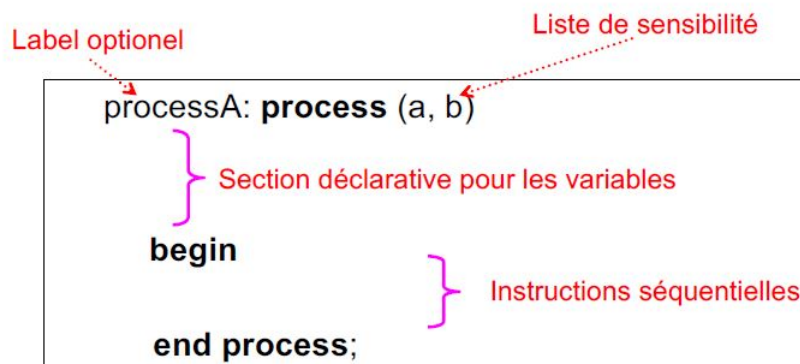


FIGURE 18 – Lien avec une bibliothèque

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

interdit d'avoir une variable sans valeur, leur donner une valeur initiale

- ❑ 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" => subtract;
  when others => illegal_opcode;
end case;
```
- ❑ **Instruction d'itération:** loop, while, for...

FIGURE 20

1.14 Types supplémentaires et conversion

le langage 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 générer 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'état bas

Qout <= '0'; --initialisation asynchrone

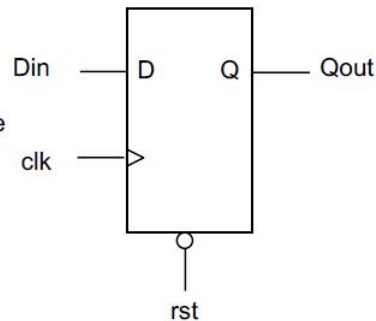
elsif rising_edge(clk) then

-- 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;

```

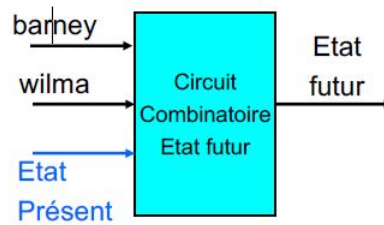


FIGURE 23

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;

```

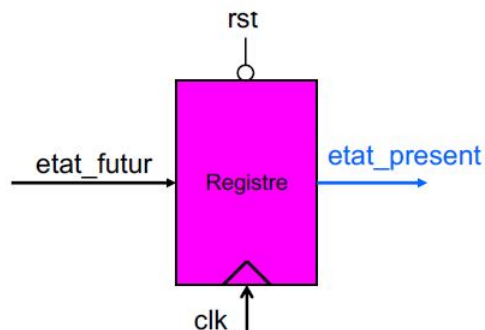


FIGURE 24

1.16.3 Circuit de sortie



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.

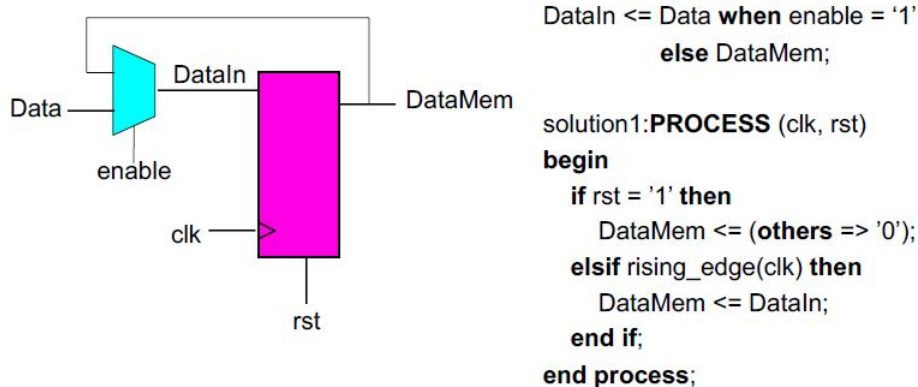


FIGURE 26

Data est toujours une constantge de type signal. On fait des opérations à l'intérieur des process sur des unsigned.

Pas besoin de else

la valeur du signal DataMem est implicitement conservée si le signal n'est pas affecté

1.17 IMPORTANT

1.17.1 Signal

Un signal c'est un fil, AUCUNE MEMOIRE

1.17.2 Un combinatoire

SI LA SORTIE CHANGE IMMEDIATEMENT

affecter un signal combinatoire à un circuit combinatoire par l'opérateur " \leq " lien définitif comme si on soudait le signal à un composant

Si un élément de la liste de sensibilité change, INSTANTANEMENT la sortie change

1.17.3 Mémoire

LA SORTIE CHANGE AU COUP DE CLOCK

Une mémoire change au **flanc actif d'horloge**. Dans une mémoire, on stocke une valeur et lorsque flanc actif d'horloge, mise à jour de la sortie.

1.17.4 Longue mémoire

CHANGE AU COUP DE CLOCK AVEC CONDITION FIXEE

Dans la partie rising edge, on va mettre un If et si la condition choisie est respectée on affecte une nouvelle valeur.

1.17.5 RTL

$[Combinatoire] ==> [Mémoire] ==> [Combinatoire] ==> [Mémoire]$

????????????Page 56 à rajouter ??????????????

1.18 Exercices

- ❑ 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.

FIGURE 27 – Question

```
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;
```

FIGURE 28 – réponse

```
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;
```

FIGURE 29 – réponse

2 Chapter 2 Conception hiérarchique

Système de plus en plus complexe

1) Pour répondre à la complexité qui augmente, on applique une technique de travail universelle:

Diviser pour régner = réduire la complexité en plusieurs sous-problèmes

2) Un système numérique sera découpé en plusieurs composants qui interagissent/communiquent.

Forme de découpage la plus fréquente: HIERARCHIQUE

2.1 Concept de découpage hiérarchique

Découpage hiérarchique

- Une structure hiérarchique est composée d'un controller et de un ou plusieurs workers.
- Un **controller**, qui est une machine d'états, commande les workers avec les signaux **Startx**
- Des **workers** démarrent le traitement selon la valeur de **Startx** et indiquent sa fin avec le signal **Endx**

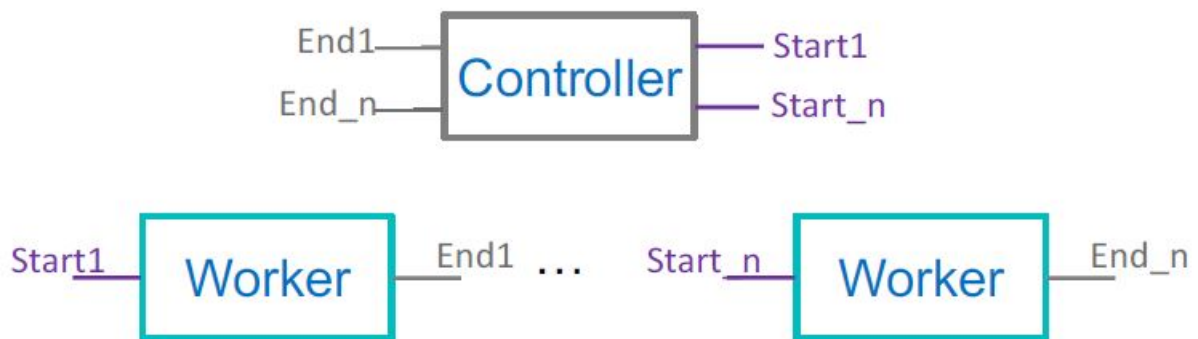


FIGURE 30 – découpage hiérarchique

2.2 Communication

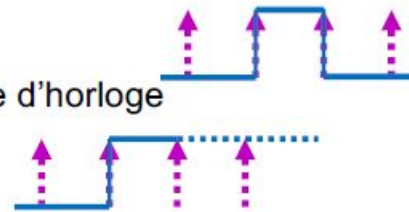
- La communication entre le controller et un worker peut se faire selon 2 modes:

- par **événement**:

1 impulsion d'une durée d'un cycle d'horloge

- par **état**:

Valeur maintenue



- Les signaux **Startx** sont généralement codés en mode état. Ce codage offre la possibilité d'interrompre le traitement en cours dans le Worker en changeant simplement la valeur du signal.
- Les signaux **Endx** sont en principe en mode événement.

FIGURE 31 – Type de communication

2.3 A COMPLETER

3 Types, Opérateurs et Conversions

VHDL est un langage **typé** où il est obligatoire de spécifier le type des **objets** utilisés.

□ Un sous-type (subtype)

- est contraint, il définit un sous-ensemble des valeurs du type père.
- hérite des opérateurs du type père.

NB :

L'utilisation des sous-types permet de diminuer la quantité de ressources utilisées dans le circuit programmable.

FIGURE 32 – Les sous-types

3.1 types scalaires entiers

□ Types prédéfinis du packaging standard:

```
type INTEGER is range -2'147'483'648 to 2'147'483'647;  
subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;  
subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;
```

□ Exemples de définition:

```
type short is range -128 to 127;  
subtype nat4 is natural range 0 to 15;  
subtype offset is nat4 range 14 to 15;
```

NB:

Un sous-type d'un sous-type est un sous-type du type père.

Tous les types/sous-types INTEGER incluent, à cause du codage, la valeur 0, même si l'intervalle ne le contient pas.

FIGURE 33 – Types scalaires entiers