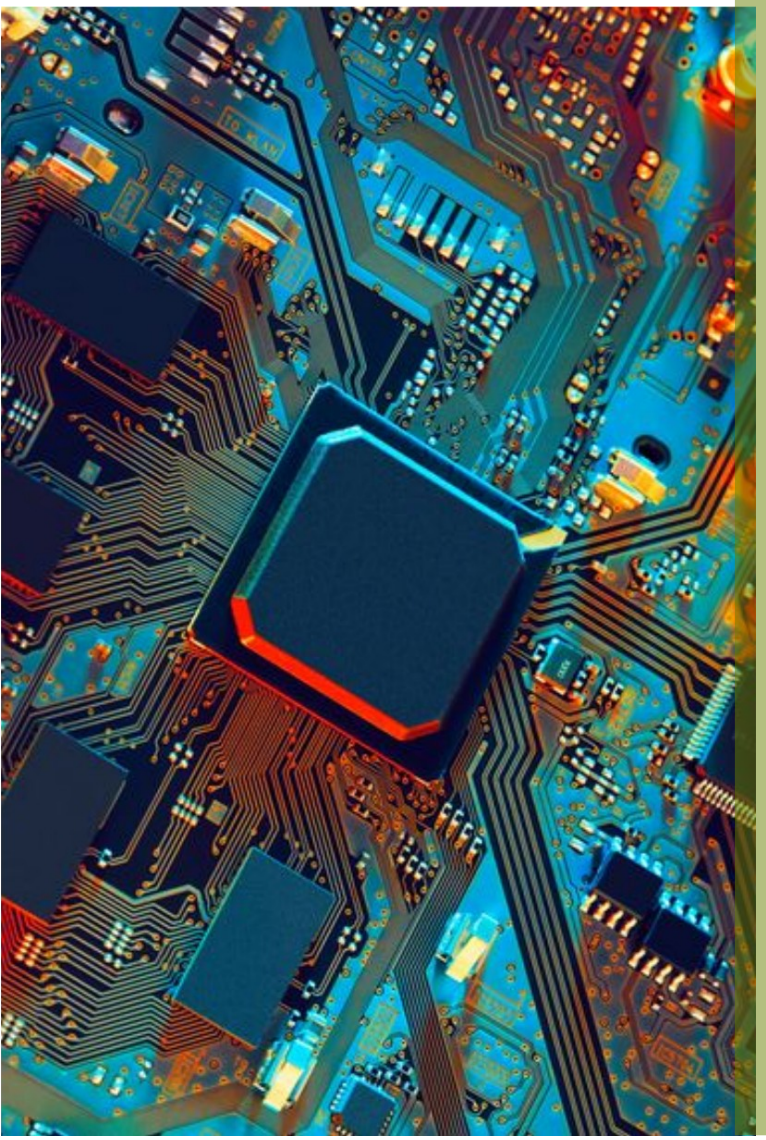
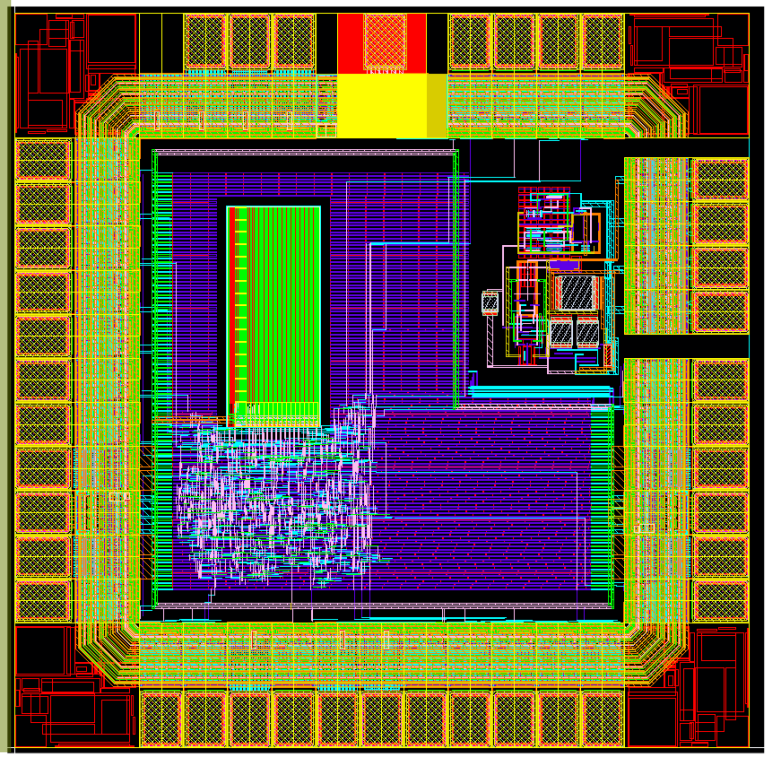


Techno des Circuits Numériques

EN208

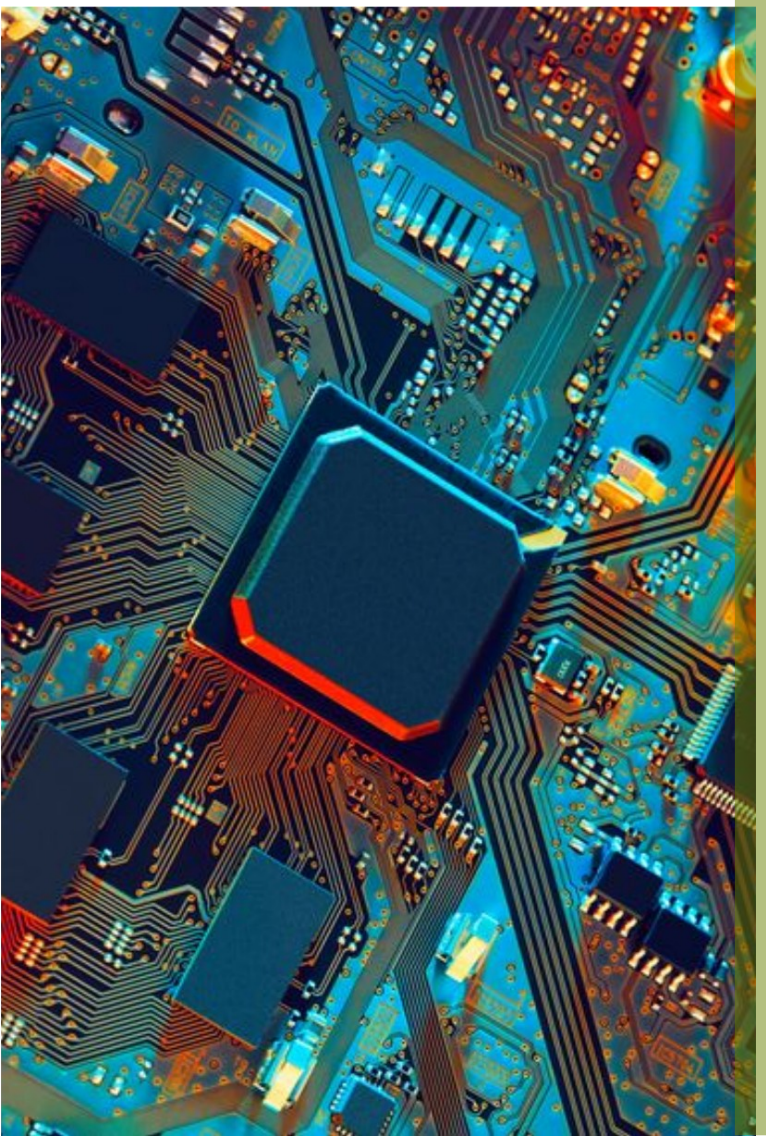
Sylvie Renaud / Yannick Bornat



Plan du cours

8 séances de CM :

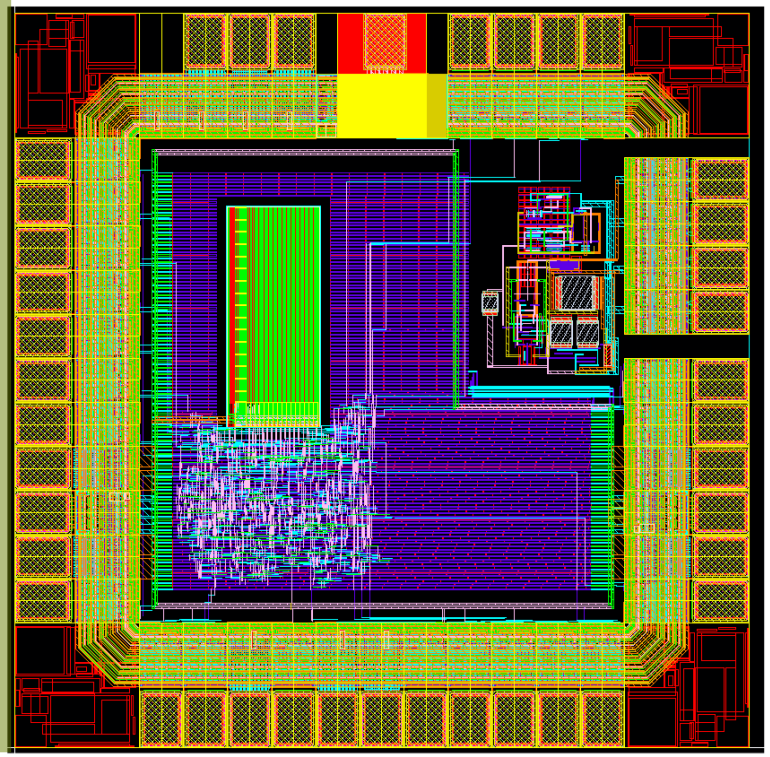
1	- Rappels sur le VHDL	
2	- Intro : culture de l'industrie	*
3	- FPGA / Architecture	
4	- Synthèse, placement/routage	
5,6,7	- familles de CI numériques :	*
	caractéristiques	*
	performances	*
	…	*
8	- Test	*



(R)appels de VHDL

EN208

Sylvie Renaud /Yannick Bornat



Rappels sur le VHDL

Dérouillage express !

- premier TD : Vendredi

- premier TP : Lundi prochain

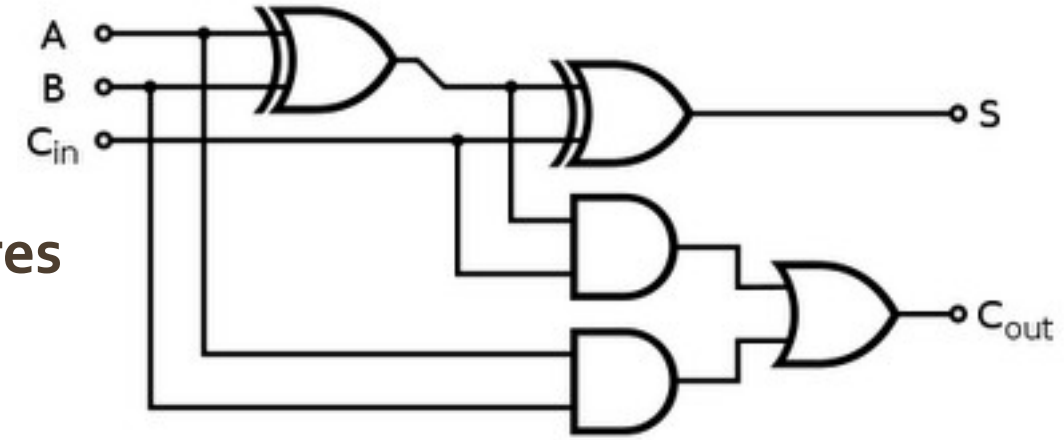
Rappels sur le VHDL

Dérouillage express !

- premier TD : Vendredi
- premier TP : Lundi prochain

Cette séance de révision NE REMPLACE PAS vos notes de 1A

Rappels sur le VHDL

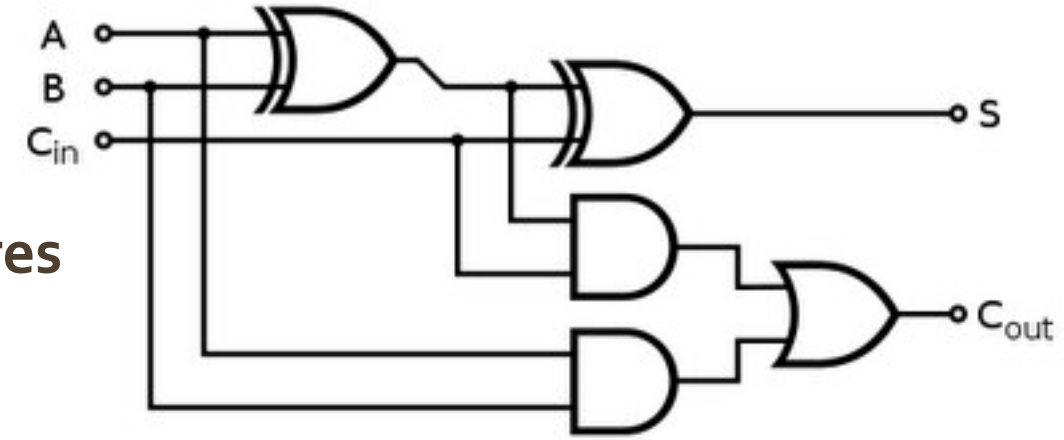


Un langage qui permet de décrire des architectures numériques

Exemple pour le schéma ci-dessus :

- **on appelle D le résultat de** **(A xor B)**
- **La sortie S vaut** **(D xor Cin)**
- **La sortie Cout vaut** **(D and Cin) or (A and B)**

Rappels sur le VHDL



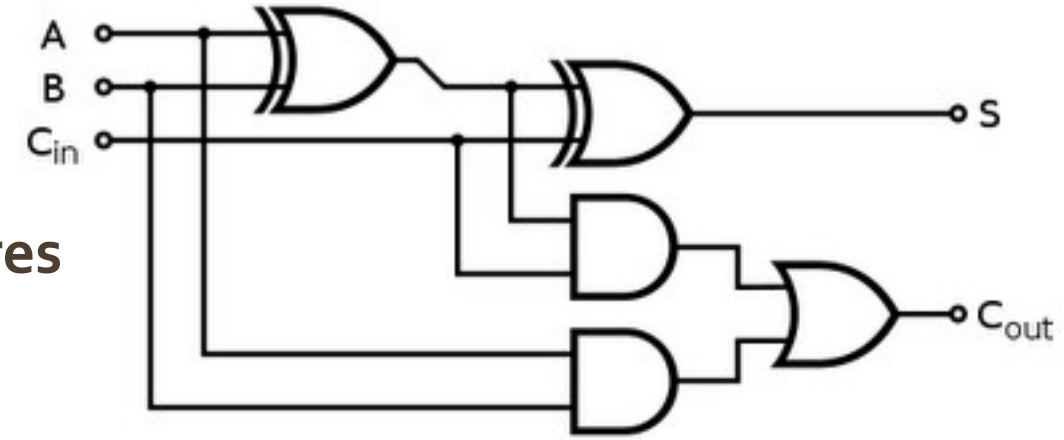
Un langage qui permet de décrire des architectures numériques

On ne programme pas une archi VHDL : on la décrit

On décrit des modules :
- pour la simulation
- pour l'implantation

→ Seul un sous-ensemble est accessible

Rappels sur le VHDL



Un langage qui permet de décrire des architectures numériques

On ne programme pas une archi VHDL : on la décrit

On décrit des modules :
- pour la simulation
- pour l'implantation

→ Seul un sous-ensemble est accessible

Internet est globalement moins doué que les exigences en 2A de l'Emmk
=> vérifiez que l'auteur fait de l'implantation !

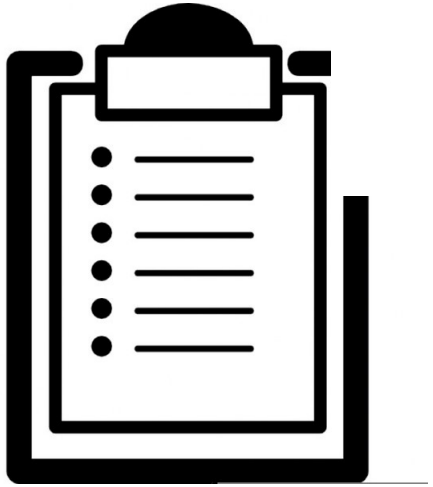
Les Bases du langage VHDL

C'est Verbeux (mais c'est pour votre bien)

- Force à structurer les descriptions pour la synthèse
- Impose de lever toutes les ambiguités

Structure d'un module :

- Une entité
- Une architecture
  - déclarations
  - process



A l'ENSEIRB et comme en entreprise :
Un fichier VHDL contient une seule entité, et une seule architecture.
Ils ont le même nom que celui du fichier.

Entité (Entity)

Elle permet de définir :

- **le nom du module**
- **Les entrées/sorties**
  - **nom**
  - **direction**
  - **type**

L'ordre n'a pas d'importance

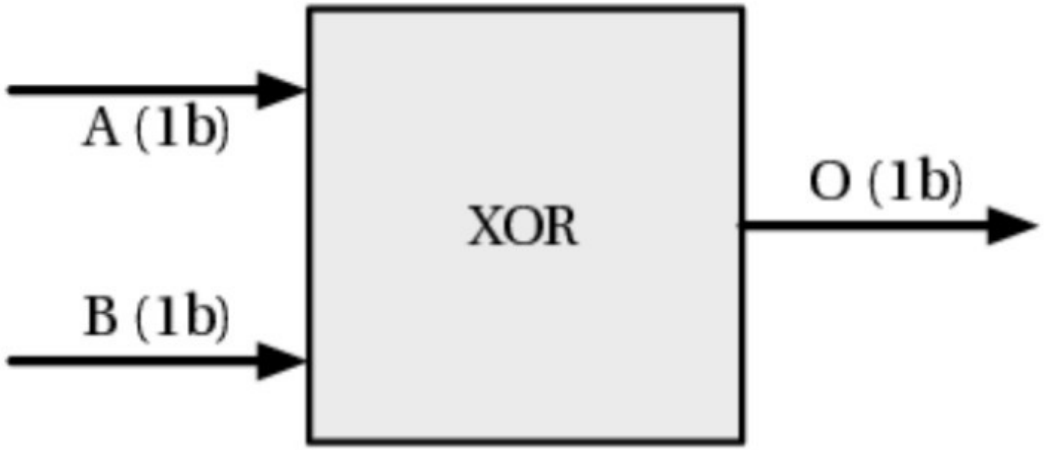
```
ENTITY nom_entite IS
     PORT (
          -- LISTE DES ENTREES
          entree_1 : IN  TYPE_ENTREE;
          -- ... ... ... ...
          entree_n : IN  TYPE_ENTREE;

          -- LISTE DES SORTIES
          sortie_1 : OUT TYPE_SORTIE;
          -- ... ... ... ...
          sortie_n : OUT TYPE_SORTIE
     );
END nom_entite;
```

Entité : exemple porte XOR

Entity :

Rien d'exceptionnel

Juste une notation différente



Les noms sont simplistes

- pour ne pas surcharger

- parce que le module est simple

- comme exemple de ce qu'il ne faut pas faire

```
ENTITY xor_v1 IS
    PORT (
        A : in  STD_LOGIC;
        B : in  STD_LOGIC;
        O : out STD_LOGIC
    );
END xor_v1;
```

L'architecture

Elle permet de décrire le comportement du module

2 parties :

- **déclarations**
  - **signaux si on a besoin de valeurs intermédiaire**
- **descriptions**
  - **les processus**

```
ARCHITECTURE nom_arch OF nom_entite IS
    --
    -- DECLARATION DES SIGNAUX INTERNES
    --
    SIGNAL signal_1 : TYPE_SIGNAL;
    -- ... ... ... ... ... ... ...
    -- ... ... ... ... ... ... ...
    SIGNAL signal_n : TYPE_SIGNAL;

    --
    -- DECLARATION DES "COMPONENTS"
    --

BEGIN

    --
    -- DU COMPORTEMENT DE L'ARCHITECTURE
    -- + PROCESSUS IMPLICITES,
    -- + PROCESSUS EXPLICITES,
    -- + INSTANCIATION DE SOUS MODULES,
    --

END nom_arch;
```

Les architectures : exemple XOR – version 1

Processus implicite

```
ARCHITECTURE Behavioral OF xor_v1 IS
BEGIN

    O <= A XOR B;

END Behavioral;
```

Processus explicite

```
ARCHITECTURE Behavioral OF xor_v2 IS
BEGIN

    PROCESS(A, B)
    BEGIN
        O <=    A XOR B;
    END PROCESS;

END Behavioral;
```

Les architectures : exemple XOR – version 2

Processus explicite

```
ARCHITECTURE Behavioral OF xor_v3 IS
BEGIN

    PROCESS(A, B)
    BEGIN
        O <= ((NOT A) AND B) OR (A AND (NOT B));
    END PROCESS;

END Behavioral;
```

Les architectures : exemple XOR – version 3

Processus explicite :

- **Liste de sensibilité (parenthèses derrière PROCESS)**
- **Elle contient TOUTES les entrées**

Exemples de process qui ne marchent pas :

```
ARCHITECTURE arch OF xor_v3 IS
BEGIN

    PROCESS(A)
    BEGIN
        O <= A XOR B;
    END PROCESS;

END Behavioral;
```

```
ARCHITECTURE Behavioral OF foo IS
  SIGNAL T : STD_LOGIC;
BEGIN

    PROCESS(A, B)
    BEGIN
        T <=   NOT A;
        O <= T AND B;
    END PROCESS;

END Behavioral;
```

Les architectures : exemple XOR – version 3

Processus explicite :

- **Liste de sensibilité (parenthèses derrière `PROCESS`)**
- **Elle doit contenir TOUTES les entrées**

Exemples de process qui ne marchent pas :

Pourquoi c'est dangereux ?!?

simu /= implantation

- **Si simu OK : aucune chance que ça marche**
- **Si simu KO : Aurez-vous l'idée d'implanter ?**

- **Le simulateur/synthétiseur vous prévient**

```
ARCHITECTURE arch OF xor_v3 IS
BEGIN

    PROCESS(A)
    BEGIN
        O <= A XOR B;
    END PROCESS;

END Behavioral;
```

```
ARCHITECTURE Behavioral OF foo IS
  SIGNAL T : STD_LOGIC;
BEGIN

    PROCESS(A, B)
    BEGIN
        T <=   NOT A;
        O <= T AND B;
    END PROCESS;

END Behavioral;
```

Écriture d'un multiplexeur

4 entrées (1 bit)
1 sortie (1 bit)

Selection : 2 bits

Process implicite →

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

entity MUX4 is
    port(   in0  : in  std_logic;
            in1  : in  std_logic;
            in2  : in  std_logic;
            in3  : in  std_logic;

            sel  : in  std_logic_vector(1 downto 0);

            dout : out std_logic);
end MUX4;


architecture Behavioral of MUX4 is

begin

    dout <= in0 when sel = "00" else
            in1 when sel = "01" else
            in2 when sel = "10" else
            in3;


end Behavioral;
```

Écriture d'un multiplexeur

4 entrées (1 bit)
1 sortie (1 bit)

Selection : 2 bits

Process implicite
Process explicite (case) →

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

entity MUX4 is
    port(   in0  : in  std_logic;
            in1  : in  std_logic;
            in2  : in  std_logic;
            in3  : in  std_logic;

            sel  : in  std_logic_vector(1 downto 0);

            dout : out std_logic);
end MUX4;


architecture Behavioral of MUX4 is

begin

    Process(in0, in1, in2, in3, sel)
    begin
        case sel is
            when "00"   => dout <= in0;
            when "01"   => dout <= in1;
            when "10"   => dout <= in2;
            when others => dout <= in3;
        end case;
    end process;

end Behavioral;
```

Écriture d'un multiplexeur

4 entrées (1 bit)
1 sortie (1 bit)

Selection : 2 bits

Process implicite
Process explicite (case)
Process explicite (if..elsif) →

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

entity MUX4 is
    port(   in0  : in  std_logic;
            in1  : in  std_logic;
            in2  : in  std_logic;
            in3  : in  std_logic;

            sel  : in  std_logic_vector(1 downto 0);

            dout : out std_logic);
end MUX4;


architecture Behavioral of MUX4 is

begin

    Process(in0, in1, in2, in3, sel)
    begin
        if    sel = "00" then dout <= in0;
        elsif sel = "01" then dout <= in1;
        elsif sel = "10" then dout <= in2;
        else                  dout <= in3;
        end if;
    end process;

end Behavioral;
```

Écriture d'un multiplexeur

4 entrées (1 bit)
1 sortie (1 bit)

Selection : 2 bits

Process implicite
Process explicite (case)
Process explicite (if..elsif)
Process incompréhensible →

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

entity MUX4 is
    port(   in0  : in  std_logic;
            in1  : in  std_logic;
            in2  : in  std_logic;
            in3  : in  std_logic;

            sel  : in  std_logic_vector(1 downto 0);

            dout : out std_logic);
end MUX4;


architecture Behavioral of MUX4 is

begin

    dout <= (in0 and not sel(1) and not sel(0)) or
            (in1 and not sel(1) and     sel(0)) or
            (in2 and     sel(1) and not sel(0)) or
            (in3 and     sel(1) and     sel(0));

end Behavioral;
```

Écriture d'un multiplexeur

4 entrées (1 bit)
1 sortie (1 bit)

Selection : 2 bits

Process implicite
Process explicite (case)
Process explicite (if..elsif)
Process incompréhensible
Process garcimore →

des fois ça marche,
des fois ça marche pas …
bon courage pour le debug !

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

entity MUX4 is
    port(   in0  : in  std_logic;
            in1  : in  std_logic;
            in2  : in  std_logic;
            in3  : in  std_logic;

            sel  : in  std_logic_vector(1 downto 0);

            dout : out std_logic);
end MUX4;


architecture Behavioral of MUX4 is

begin

    Process(in0, in1, in2, in3, sel)
    begin
        if sel = "00" then dout <= in0; end if;
        if sel = "01" then dout <= in1; end if;
        if sel = "10" then dout <= in2; end if;
        if sel = "11" then dout <= in3; end if;
    end process;

end Behavioral;
```

Écriture d'un multiplexeur

4 entrées (1 bit)
1 sortie (1 bit)

Selection : 2 bits

Process implicite
Process explicite (case)
Process explicite (if..elsif)
Process incompréhensible
Process garcimore
Process qui marche pas →

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity MUX4 is
    port(   in0  : in  std_logic;
            in1  : in  std_logic;
            in2  : in  std_logic;
            in3  : in  std_logic;

            sel  : in  std_logic_vector(1 downto 0);

            dout : out std_logic);
end MUX4;


architecture Behavioral of MUX4 is

begin

    Process(in0, in1, in2, in3)
    begin
        if sel(0) = '0' then
            if sel(1) = '0' then
                dout <= in0;
            else
                dout <= in2;
            end if;
        else
            if sel(1) = '0' then
                dout <= in1;
            else
                dout <= in3;
            end if;
        end if;
    end process;

end Behavioral;
```

Écriture d'un multiplexeur

4 entrées (1 bit)
1 sortie (1 bit)

Selection : 2 bits

- **Process implicite**
- **Process explicite (case)**
- **Process explicite (if..elsif)**
- **Process incompréhensible**
- **Process garcimore**
- **Process qui marche pas**
- **Erreur de synthèse →**

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

entity MUX4 is
    port(   in0  : in  std_logic;
            in1  : in  std_logic;
            in2  : in  std_logic;
            in3  : in  std_logic;

            sel  : in  std_logic_vector(1 downto 0);

            dout : out std_logic);
end MUX4;


architecture Behavioral of MUX4 is

begin

    dout <= in0 when sel = "00";
    dout <= in1 when sel = "01";
    dout <= in2 when sel = "10";
    dout <= in3 when sel = "11";

end Behavioral;
```

Une architecture peut DOIT contenir plusieurs process

1 signal <=> 1 process

Processus Synchrones (ou séquentiel)

Le processus réagit à l'horloge

=> et donc uniquement à l'horloge !

bascule simple →

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

entity D_flipFlop is
    port(   clk   : in  std_logic;
            reset : in  std_logic;
            in_D  : in  std_logic;
            out_Q : out std_logic);
end D_flipFlop;


architecture Behavioral of D_flipFlop is

begin

    Process(clk)
    begin
        if rising_edge(clk) then
            if reset = '1' then
                out_Q <= '0';
            else
                out_Q <= in_D
            end if;
        end if;
    end process;

end Behavioral;
```

Processus Synchrones (ou séquentiel)

Le processus réagit à l'horloge

=> et donc uniquement à l'horloge !

bascule simple

registre (16 bits) →

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Reg16 is
    port(   clk   : in  std_logic;
            reset : in  std_logic;
            in_D  : in  std_logic_vector(15 downto 0);
            out_Q : out std_logic_vector(15 downto 0));
end Reg16;


architecture Behavioral of Reg16 is

begin

    Process(clk)
    begin
        if rising_edge(clk) then
            if reset = '1' then
                out_Q <= "0000000000000000";
            else
                out_Q <= in_D
            end if;
        end if;
    end process;

end Behavioral;
```

Processus Synchrones (ou séquentiel)

Le processus réagit à l'horloge

=> et donc uniquement à l'horloge !

bascule simple

registre (16 bits) →

On n'est synchrone :

QUE sur l'horloge

sur UNE SEULE horloge

En 2A: une seule horloge dans toute l'architecture !!!

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Reg16 is
    port(   clk   : in  std_logic;
            reset : in  std_logic;
            in_D  : in  std_logic_vector(15 downto 0);
            out_Q : out std_logic_vector(15 downto 0));
end Reg16;


architecture Behavioral of Reg16 is

begin

    Process(clk)
    begin
        if rising_edge(clk) then
            if reset = '1' then
                out_Q <= "0000000000000000";
            else
                out_Q <= in_D
            end if;
        end if;
    end process;

end Behavioral;
```

Types de données

Définis dans IEEE.STD_LOGIC_1164
Ou dans IEEE.NUMERIC_STD

Std_logic
Std_logic_vector
Integer
Signed
Unsigned
Enuméré

Il y en a beaucoup plus, mais on n'en aura pas besoin ici
On verra le tableau plus tard dans l'année (en TP)

STD_LOGIC

Ce n'est pas « juste » un bit

=> parce qu'un bit, en VHDL, ça existe !

- **ça vaut '0' ou '1'**
- **personne ne s'en sert… (sauf des *gens* sur *internet* …)**

Valeurs que peut prendre un STD_LOGIC :

'U'	**non initialisé**
'0'	**état bas**
'1'	**état haut**
'-'	**don't care**
'Z'	**haute impédance**
'X'	**tu t'es planté !!**

et d'autres valeurs dont on n'aura pas besoin (en 2A) ….

STD_LOGIC_VECTOR

Permet de grouper des données binaires dans un seul signal

```
toto : std_logic_vector(15 downto 0);
```

Écriture :

```
toto <= "0101110001110110";
toto <= x"5C76";
```

Extraction de sous-parties :

```
toto(14)
toto(7 downto 4)
```

Agrégation :

```
toto(14) & toto(7) & toto(5 downto 0)
```

Affectation partielle : `toto(13 downto 11) <= "001";`

Tout ce qui est faisable avec des std_logic est faisable avec des std_logic_vector de même taille

STD_LOGIC_VECTOR

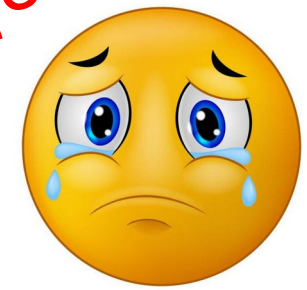
Permet de grouper des données binaires dans un seul signal

```
toto : std_logic_vector(15 downto 0);
```

Écriture :

```
toto <= "0101110001110110";
toto <= x"5C76";
```

Pas d'opérations arithmétiques



Extraction de sous-parties :

```
toto(14)
toto(7 downto 4)
```

Agrégation :

```
toto(14) & toto(7) & toto(5 downto 0)
```

Affectation partielle :

```
toto(13 downto 11) <= "001";
```

Tout ce qui est faisable avec des std_logic est faisable avec des std_logic_vector de même taille

Integer

Pour faire des opérations arithmétiques (chouette) :

```
signal year : integer range 0 to 42;
```

range :

facultatif d'après la norme

indispensable en pratique

Opérations autorisées

- en théorie : toutes

- en pratique : `+, -, *2^n, /2^n, mod 2^n, ...`

- en faisant TRÈS attention : `*, /k`

Pas associé à une représentation binaire !

Signed / unsigned

Nécessite la bilbiothèque NUMERIC_STD

```
totou : unsigned(15 downto 0);
totos : signed(15 downto 0);
```

La fusion des deux mondes…

représentation binaire

opérations arithmétiques

… mais pas de représentation possible en VHDL.

```
totou <= to_unsigned(42, 16) ; -- valeur 42 sur 16 bits
totou <= unsigned(x"00") ;
```

Signed / unsigned

Utilisable pour passer de `std_logic_vector` à `integer`

Démonstration :

```
signal sig_sv : std_logic_vector(7 downto 0);
signal sig_iu : integer range    0 to 255;
signal sig_is : integer range -128 to 127;

        sig_iu <= to_integer(unsigned(sig_sv));
        sig_is <= to_integer(signed(sig_sv));
        sig_sv <= std_logic_vector(to_unsigned(sig_iu, 8));
        sig_sv <= std_logic_vector(to_signed(sig_is, 8));
```

Pour s'en souvenir :

si on est en relation avec un `integer` il y a `to_` devant…

si on est en relation avec un `std_logic_vector`, on utilise le type directement

(`to_std_logic_vector()` n'existe pas)

Signed / unsigned

opérations arithmétiques, problèmes de taille :

Addition soustraction :

taille du résultat = max(taille de chaque terme)

taille(A+B) = max(taille(A), taille(B))

Multiplication :

taille du résultat = somme des tailles des termes

taille (A*B) = taille(A) + taille(B)

Type énuméré

Très souvent représentatif d'un état

Mais ne se limite pas aux machines d'états

```
type dalton is (joe, jack, william, averel);
signal taular : dalton;
…
```

```
if argneux = '1' then
            taular <= joe;
end if;
```

Type énuméré

Très souvent représentatif d'un état

Mais ne se limite pas aux machines d'états

```
type dalton is (joe, jack, william, averel);
signal taular : dalton;
…
```

```
if argneux = '1' then
        taular <= joe;
end if;
```

Cette ligne définit 5 choses :

- le type `dalton`

- la constante `joe` de type `dalton` de valeur inconnue

- la constante `jack` de type `dalton` de valeur inconnue mais différente de `joe`

- la constante `william` de type `dalton` de valeur inconnue mais différente de `joe` et `jack`

- la constante `averel` de type `dalton` de valeur stupide

Type énuméré

Très souvent représentatif d'un état

Mais ne se limite pas aux machines d'états

```
type dalton is (joe, jack, william, averel);
signal taular : dalton;
…
```

Cette ligne déclare un signal

- de type `dalton`

- ne pouvant prendre que 4 valeures possibles :
  - `joe`
  - `jack`
  - `william`
  - `averel`

```
if argneux = '1' then
        taular <= joe;
end if;
```

Description hiérarchique

On n'écrit pas toute une architecture dans un même fichier
On appelle une autre entité en tant que composant :

Pour utiliser une autre entité:
- déclaration comme composant

- utilisation (instantiation)
"port map"

```
architecture Behavioral of MUX4 is

    component REG8 is
        port (  clk      : in  std_logic;
                reset    : in  std_logic;
                data_in  : in  std_logic_vector(7 downto 0);
                data_out : out std_logic_vector(7 downto 0));
    end component REG8;

begin


    dut : entity work.REG8
        port map(clk          => clk,
                 reset        => reset,
                 data_in      => data_in,
                 data_out     => data_out);

end Behavioral;
```

Description hiérarchique

Utilisations multiples…

Ben comme n'importe quel composant !

```
architecture Behavioral of MUX4 is

    component REG8 is
        port (  clk      : in  std_logic;
                reset    : in  std_logic;
                data_in  : in  std_logic_vector(7 downto 0);
                data_out : out std_logic_vector(7 downto 0));
    end component REG8;

    signal data_mid : std_logic_vector(7 downto 0);

begin


    reg2 : entity work.REG8
        port map(clk          => clk,
                 reset        => reset,
                 data_in      => data_in,
                 data_out     => data_mid);

    reg1 : entity work.REG8
        port map(clk          => clk,
                 reset        => reset,
                 data_in      => data_mid,
                 data_out     => data_out);

end Behavioral;
```

Généricité

Un module (une entité) peut être configurable
=> on utilise des entrées "génériques"

- fonctionne comme un port mais uniquement en entrée
- les entrées "génériques" sont constantes

Conclusion

VHDL = pas plus compliqué que de dessiner un circuit

Mais

- si on considère que c'est un langage de programmation, c'est impossible de s'y retrouver

- Langage TRÈS riche, on ne fait que gratter la surface…

respectez les limites fixées par vos encadrants

ChatGPT est NUL² en VHDL

internet est nul en VHDL

ChatGPT est basé sur internet…