

EN201: Le VHDL pour la synthèse

Bertrand LE GAL
[bertrand.legal@ims-bordeaux.fr]

Filière Electronique - 2ème année
ENSEIRB-MATMECA - Bordeaux INP
Talence, France



Déroulement de l'enseignement de VHDL

⦿ En 1ère année à l'école (Semestre 5)

- ➔ EN102 - Logique combinatoire et logique séquentielle
- ➔ EN103 - Projet Numérique

⦿ En 2ème année à l'école (Semestre 7)

- ➔ EN201 - Synthèse VHDL
- ➔ EN202 - Projet VHDL

Rappels sur le langage VHDL

- Le langage VHDL est un langage informatique permettant de décrire une architecture numérique,
- On ne programme pas une architecture en VHDL, on la décrit !
- On **décrit des modules** en VHDL:
 - pour les **simulation**,
 - pour les **implanter** (ASIC/FPGA).
- En fonction de la finalité, toute la syntaxe du langage n'est pas accessible.



Rappels

Les bases du langage VHDL

- ◉ Le langage VHDL est un langage verbeux.
- ◉ Il a pour objectif de vous forcer a structurer vos descriptions afin de lever toutes les ambiguïtés (debugger du hardware n'est pas sympa !!!).
- ◉ Dans le langage VHDL il y a **5 concepts de base** à maitriser absolument:
 - Les entités
 - Les architectures
 - Les processus
 - Les signaux
 - Les variables



Les bases du langage VHDL - les « entités »

- **Les entités** vous permettent de définir:
 - Le nom du module,
 - La liste des E/S du module,
 - Le type des E/S du module.
- Pour chaque E/S du module, il faut définir:
 - Le sens du port (entrée ou sortie),
 - Le type des données échangées,
- L'ordre de définition des E/S n'a que peu d'impact pour la suite,
- Définir une entité revient à déclarer un composant dans votre « bibliothèque ».

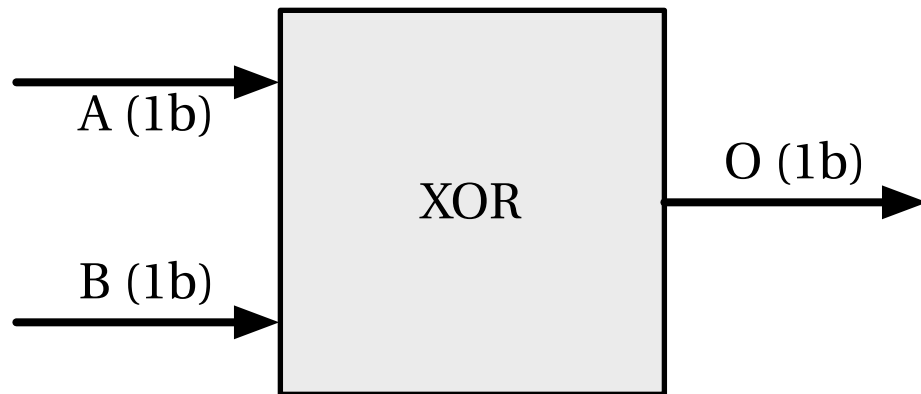
```
ENTITY nom_entite IS
    PORT (
        -- LISTE DES ENTREES
        entree_1 : IN  TYPE_ENTREE;
        -- ... ..
        entree_n : IN  TYPE_ENTREE;

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

```
END nom_entite;

-- ... ..
-- ... ..
```

Les bases du langage VHDL - les « entités »



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

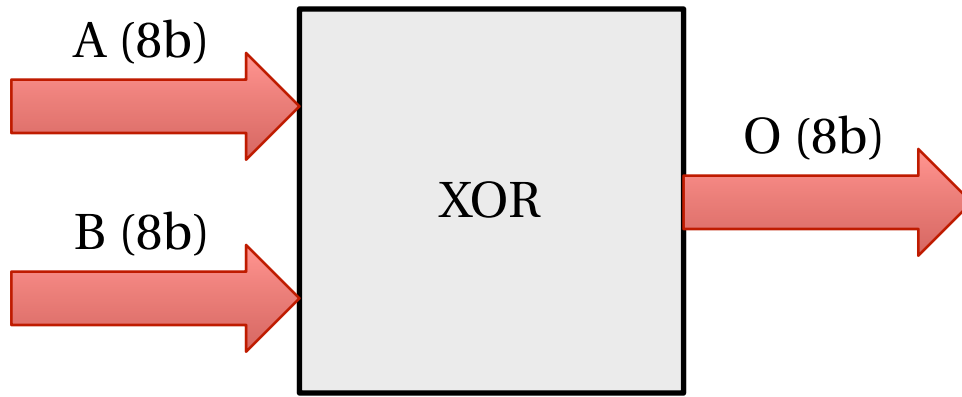
L'ordre de définition des ports n'est pas imposé mais peut avoir un impact sur le **PORT MAP** des E/S.

Le sens des ports est spécifié à l'aide des types **IN** & **OUT**

Le type **STD_LOGIC** est utilisé pour les données codées sur un bit.

Attention le dernier signal ne nécessite pas de « ; » après sa déclaration

Les bases du langage VHDL - les « entités »

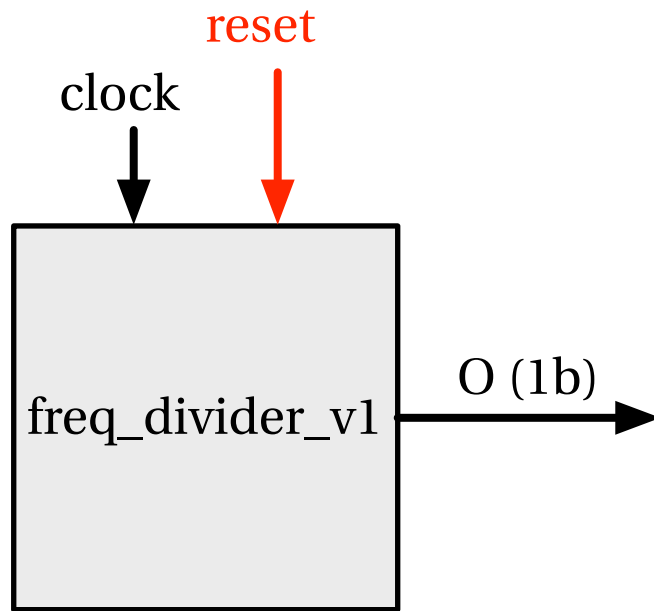


Le type **STD_LOGIC_VECTOR** est utilisé pour les données nécessitant plusieurs bits.

```
ENTITY xor_v2 IS
  PORT (
    A : in  STD_LOGIC_VECTOR(7 DOWNTO 0);
    B : in  STD_LOGIC_VECTOR(7 DOWNTO 0);
    O : out STD_LOGIC_VECTOR(7 DOWNTO 0)
  );
END xor_v2;
```

nécessitant plusieurs bits.

Les bases du langage VHDL - les « entités »

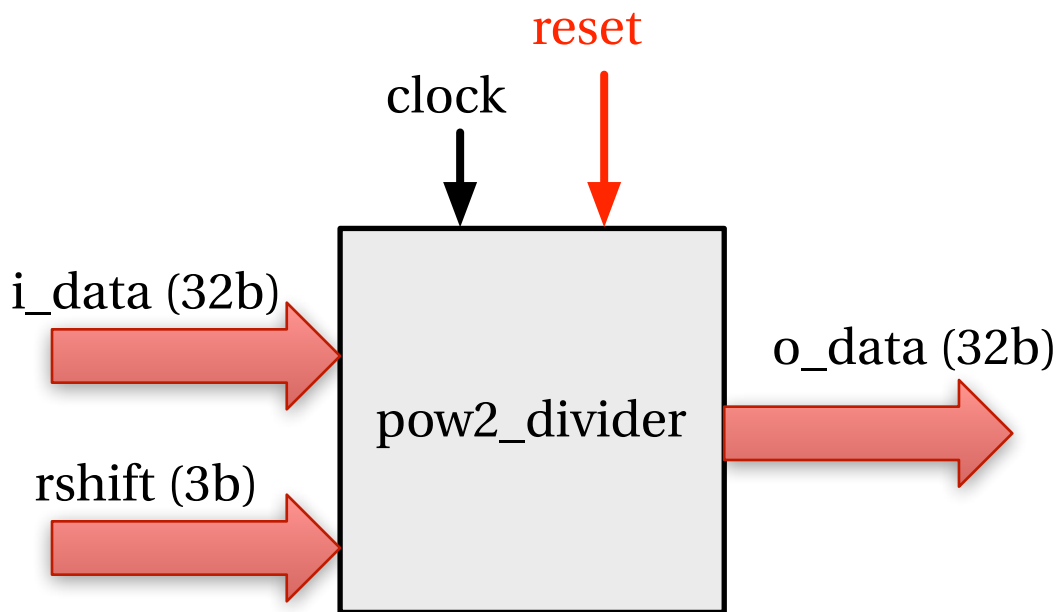


```
ENTITY freq_divider_v1 IS
  PORT (
    clk      : in  STD_LOGIC;
    reset    : in  STD_LOGIC;
    O        : out STD_LOGIC
  );
END freq_divider_v1;
```

Les signaux d'horloge et de remise à zéro sont des signaux binaires

Soyez cohérent sur la définition du signal de reset: actif soit à l'état haut soit à l'état bas.

Les bases du langage VHDL - les « entités »



L'interface de ce composant est plus complexe mais les concepts sont toujours les mêmes

sont toujours les mêmes

```
ENTITY pow2_divider_v1 IS
  PORT (
    clk      : in  STD_LOGIC;
    reset    : in  STD_LOGIC;
    i_data   : in  STD_LOGIC_VECTOR(31 DOWNTO 0);
    rShift   : in  STD_LOGIC_VECTOR( 2 DOWNTO 0);
    o_data   : out STD_LOGIC_VECTOR(31 DOWNTO 0);
  );
END pow2_divider_v1;
```

Les bases du langage VHDL - les « architectures »

- ⊙ **Les entités** définissent le nom des modules ainsi que leurs E/S,
- ⊙ **Les architectures** définissent le comportement interne des modules,
- ⊙ L'architecture se décompose en deux parties distinctes:
 - La déclaration des signaux et des sous modules utilisés dans l'architecture,
 - La description du comportement du module,
- ⊙ Le comportement du module peut être décrit de différentes manières...

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

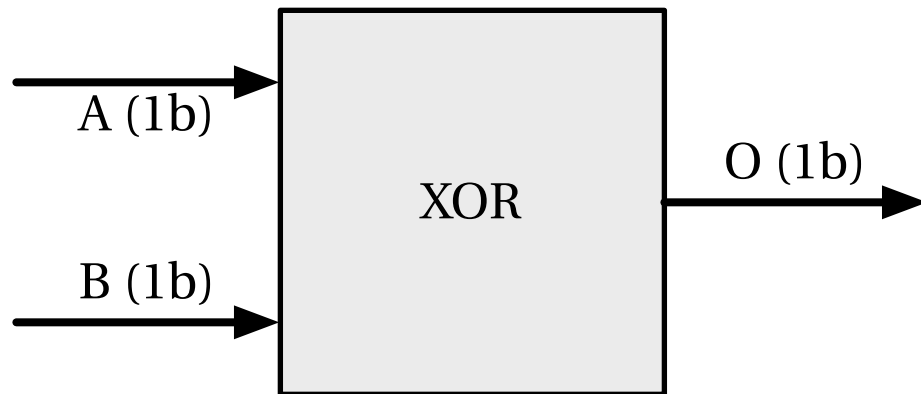
    --
    -- DECLARATION DES "COMPONENTS"
    --

BEGIN

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

END nom_arch;
```

Les bases du langage VHDL - les « architectures »



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

```
ARCHITECTURE Behavioral OF xor_v1 IS
BEGIN

    O <= A XOR B;

END Behavioral;
```

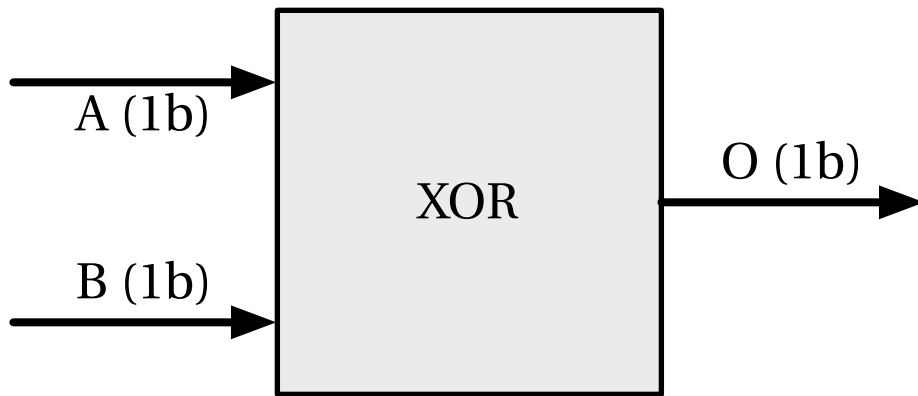
La sortie du module est calculée à l'aide d'un processus implicite. A chaque modification de **A** ou **B**, l'évaluation de (**A xor B**) est re-exécutée.

L'évaluation de (**A xor B**) est re-exécutée.

La description du comportement est concise, mais limitée aux cas simples.

concise, mais limitée aux cas simples.

Les bases du langage VHDL - les « architectures »



```
ARCHITECTURE Behavioral OF xor_v2 IS
BEGIN

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

END Behavioral;
```

```
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;
```

La sortie du module est calculée à l'aide d'un processus explicite.

Attention à la liste de sensibilité !

Les bases du langage VHDL - les « processus »

⦿ Les processus explicites:

- Permettent la modélisation de **comportements combinatoires** simples ou complexes,
- Permettent la modélisation de **comportements synchrones**.

⦿ Attention aux subtilités:

- Si un signal est manquant dans la liste de sensibilité, le comportement en simulation sera erroné,
- Dans un processus, les valeurs des signaux sont réellement mis à jour qu'une fois l'évaluation du processus terminée,

Exemples **non** fonctionnels en simulation

```
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 bases du langage VHDL - les « entités »

```
ENTITY mux41_v1 IS
    PORT (
        E0  : IN    STD_LOGIC_VECTOR(7 DOWNTO 0);
        E1  : IN    STD_LOGIC_VECTOR(7 DOWNTO 0);
        E2  : IN    STD_LOGIC_VECTOR(7 DOWNTO 0);
        E3  : IN    STD_LOGIC_VECTOR(7 DOWNTO 0);
        SEL : IN    STD_LOGIC_VECTOR(1 DOWNTO 0);
        O    : OUT   STD_LOGIC_VECTOR(7 DOWNTO 0)
    );
END mux41_v1;

ARCHITECTURE Behavioral OF mux41_v1 IS
BEGIN

    O <=      E0 WHEN SEL = "00" ELSE
              E2 WHEN SEL = "01" ELSE
              E3 WHEN SEL = "10" ELSE
              E4;

END Behavioral;
```

Multiplexeur 4 entrées vers 1 sortie

La structure **ELSE ... WHEN** est utilisée pour décrire implicitement des multiplexeurs...

Les processus implicites peuvent exprimer des structures conditionnelles « simples ».

Les bases du langage VHDL - les « entités »

La structure **CASE ... WHEN** utilisée dans les processus explicites fourni le même comportement.

même comportement

L'approche est plus verbeuse mais il est possible de rajouter des conditions dans les **WHEN**.

dans les WHEN

Le **WHEN others** est obligatoire, on l'utilise souvent pour gérer le cas que l'on ne décrit pas (ici « 11 »).

l'on ne décrit pas (ici « 11 »)

Multiplexeur 4 entrées vers 1 sortie

```
ARCHITECTURE Behavioral OF mux41_v2 IS
BEGIN

    PROCESS(E0, E1, E2, E3, SEL)
    BEGIN
        CASE SEL IS
            WHEN "00"    => O <= E0;
            WHEN "01"    => O <= E1;
            WHEN "10"    => O <= E2;
            WHEN others => O <= E3;
        END CASE;
    END PROCESS;

END Behavioral;
```

Les bases du langage VHDL - description hiérarchique

```
ARCHITECTURE Behavioral OF xor_v4 IS
    SIGNAL T1, T2, T3, T4, T5 : STD_LOGIC;

    -- DECLARATION DES COMPOSANTS
    -- ...
    -- ...
    -- ...

BEGIN

    p1 : NON_v1 PORT MAP( A, T1 );
    p2 : NON_v1 PORT MAP( B, T2 );

    p3 : AND_v1 PORT MAP( A, T2, T3 );
    p4 : AND_v1 PORT MAP( B, T1, T4 );

    p5 : OR_v1
    PORT MAP(
        A => T3,
        B => T4,
        O => O
    );

END Behavioral;
```

On peut décrire le module **XOR** comme étant l'agrégation de modules de complexités inférieures (**AND**, **OR** et **NOT**).

Les signaux (fils) qui interconnectent les modules sont déclarés entre l'**ARCHITECTURE** et le **BEGIN**.

Idem pour la déclaration des modules que l'on va utiliser.

Les différents **PORT MAP** réalisent la création des composants et le binding de leurs entrées et sorties.

Les bases du langage VHDL - les « processus » synchrones

```
ARCHITECTURE Behavioral OF register_v1 IS
    SIGNAL reg : STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN

    PROCESS(RESET, CLOCK)
    BEGIN
        IF RESET = '1' THEN
            reg <= (OTHERS => '0');
        ELSIF CLK='1' AND CLK'EVENT THEN
            reg <= i_data;
        END IF;
    END PROCESS;

    o_data <= reg;

END Behavioral;
```

Les modules synchrones utilisent
toujours des processus !

codage des processus :

Les bases du langage VHDL - les « processus » synchrones

```
ARCHITECTURE Behavioral OF register_v1 IS
    SIGNAL reg : STD_LOGIC_VECTOR(7 DOWNT0 0);
BEGIN

    PROCESS(RESET, CLOCK)
    BEGIN
        IF RESET = '1' THEN
            reg <= (OTHERS => '0');
        ELSIF rising_edge(CLK) THEN
            reg <= i_data;
        END IF;
    END PROCESS;

    o_data <= reg;

END Behavioral;
```

Il existe plusieurs méthodes pour déclarer un processus synchrone.

Attention on n'est synchrone **QUE** sur l'horloge !!!

Les types de données

Les types de données disponibles en VHDL

- ⊙ En VHDL, il existe de nombreux types de données (surtout en VHDL'14)...
- ⊙ Types prédéfinis dans le langage
 - BIT, BOOLEAN, BIT_VECTOR, INTEGER,
- ⊙ Types définis dans le package **IEEE.STD_LOGIC_1164**
 - STD_LOGIC, STD_LOGIC_VECTOR,
- ⊙ Types définis dans le package **IEEE.NUMERIC_STD**
 - SIGNED, UNSIGNED,
- ⊙ En règle générale vous ne manipulerez que les types
 - STD_LOGIC, STD_LOGIC_VECTOR, SIGNED, UNSIGNED, INTEGER.

Le type de donnée **STD_LOGIC_VECTOR**

- ◉ Le type **STD_LOGIC_VECTOR** permet de modéliser des groupes de données binaires,
 - ➔ **SIGNAL A : STD_LOGIC_VECTOR(31 DOWNT0 0) := "0101.....1101";**
 - ➔ **SIGNAL B : STD_LOGIC_VECTOR(31 DOWNT0 0) := x"89ABCDEF";**
- ◉ Il est possible d'extraire des sous ensembles de bits des variables définies sous la forme de **STD_LOGIC_VECTOR**,
 - ➔ **SIGNAL C : STD_LOGIC_VECTOR(15 DOWNT0 0) := A(23 DOWNT0 8);**
 - ➔ **SIGNAL D : STD_LOGIC := A(16);**
- ◉ Et d'agréger des informations binaires pour créer des vecteurs,
 - ➔ **SIGNAL E : STD_LOGIC_VECTOR(31 DOWNT0 0) := C & C;**
 - ➔ **SIGNAL F : STD_LOGIC_VECTOR(4 DOWNT0 0) := D & '0' & D & '1';**
- ◉ Les opérations logiques de base permettent de traiter les informations à condition qu'elles possèdent le même format,
 - ➔ **SIGNAL G : STD_LOGIC_VECTOR(15 DOWNT0 0) := A(31 DOWNT0 15) OR (NOT C);**
 - ➔ **SIGNAL H : STD_LOGIC := A(7 DOWNT0 6) /= (D & D);**

Manipulation des STD_LOGIC & STD_LOGIC_VECTOR (I)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY xor_4b_4b_4b IS
PORT (
    A : IN  std_logic_vector(3 downto 0);
    B : IN  std_logic_vector(3 downto 0);
    C : OUT std_logic_vector(3 downto 0)
);
END xor_4b_4b_4b;

ARCHITECTURE arch OF xor_4b_4b_4b IS
BEGIN

    C(0) <= A(0) XOR B(0);
    C(1) <= A(1) XOR B(1);
    C(2) <= A(2) XOR B(2);
    C(3) <= A(3) XOR B(3);

END arch;
```

Les vecteurs de bits sont décomposés sous forme scalaire pour permettre un traitement binaire de l'information

Ministère de l'Information

Ensemble des services de l'information

Manipulation des STD_LOGIC & STD_LOGIC_VECTOR (2)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY xor_4b_4b_4b IS
PORT (
    A : IN  std_logic_vector(3 downto 0);
    B : IN  std_logic_vector(3 downto 0);
    C : OUT std_logic_vector(3 downto 0)
);
END xor_4b_4b_4b;

ARCHITECTURE arch OF xor_4b_4b_4b IS
BEGIN

    C(1 DOWNT0 0) <= A(1 DOWNT0 0) XOR B(1 DOWNT0 0);
    C(3 DOWNT0 2) <= A(3 DOWNT0 2) XOR B(3 DOWNT0 2);

END arch;
```

Les vecteurs de bits sont en 2 sous ensemble pour permettre semi-groupé de l'information

semi-groupé de l'information

Manipulation des STD_LOGIC & STD_LOGIC_VECTOR (3)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY xor_4b_4b_4b IS
PORT (
    A : IN  std_logic_vector(3 downto 0);
    B : IN  std_logic_vector(3 downto 0);
    C : OUT std_logic_vector(3 downto 0)
);
END xor_4b_4b_4b;

ARCHITECTURE arch OF xor_4b_4b_4b IS
BEGIN

    C <= A XOR B;

END arch;
```

Un traitement groupé des bits du vecteur est aussi possible.

Manipulation des STD_LOGIC & STD_LOGIC_VECTOR (4)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY xor_4b_4b_4b IS
PORT (
    A : IN  std_logic_vector(3 downto 0);
    B : IN  std_logic_vector(3 downto 0);
    C : OUT std_logic_vector(3 downto 0)
);
END xor_4b_4b_4b;

ARCHITECTURE arch OF xor_4b_4b_4b IS
    SIGNAL D0, D1, D2, D3 : STD_LOGIC;
BEGIN

    D3 <= A(3) XOR B(3);
    D2 <= A(2) XOR B(2);
    D1 <= A(1) XOR B(1);
    D0 <= A(0) XOR B(0);
    C  <= (D3 & D2 & D1 & D0);

END arch;
```

Exemple d'utilisation d'une
agrégation de bits post-traitement de
l'information

Information

Les types de données **SIGNED** et **UNSIGNED**

- ⊙ Afin de réaliser « simplement » des calculs arithmétiques dans vos modules, il existe différentes solutions.
- ⊙ Pour le typage des signaux et des variables:
 - ➔ Si les données sont non signées
 - **UNSIGNED(msb DOWNT0 0)** => modélisation entre $[0, +2^{\text{MSB}-1}]$
 - **SIGNAL CPT : UNSIGNED(10 DOWNT0 0) := TO_UNSIGNED(0, 10);**
 - ➔ Si les données sont signées
 - **SIGNED(msb DOWNT0 0)** => modélisation entre $[-2^{\text{MSB}-1}, +2^{\text{MSB}-1}-1]$
 - **SIGNAL CPT : SIGNED(10 DOWNT0 0) := TO_SIGNED(0, 10);**
- ⊙ La représentation interne des données est:
 - ➔ du binaire « classique » pour le type **UNSIGNED**,
 - ➔ du complément à 2 pour le type **SIGNED**,
- ⊙ La bibliothèque **IEEE.NUMERIC_STD** doit être incluse dans vos modules VHDL.

La conversion des données depuis un STD_LOGIC_VECTOR

- ◉ Considérons une information numérique stockée ou transmise sous la forme d'un **STD_LOGIC_VECTOR**.
 - ➔ **VARIABLE** STDL : **STD_LOGIC_VECTOR**(7 DOWNT0 0) := « 1000 0001 »;
- ◉ Afin de réaliser des traitements arithmétiques il est nécessaire de la **convertir en SIGNED ou UNSIGNED**:
 - ➔ **VARIABLE** UNS : **UNSIGNED**(7 DOWNT0 0) := **UNSIGNED**(STDL);
 - La variable non signée UNS contient la valeur numérique 128.
 - ➔ **VARIABLE** SIG : **SIGNED**(7 DOWNT0 0) := **SIGNED**(STDL);
 - La variable signée (complément à 2) SIG contient la valeur numérique -63.
- ◉ La **conversion en sens inverse** est tout aussi triviale:
 - ➔ **VARIABLE** DEST1 := **STD_LOGIC_VECTOR**(7 DOWNT0 0) := **STD_LOGIC_VECTOR**(UNS);
 - La variable de DEST1 contient les valeurs binaires « 1000 0001 ».
 - ➔ **VARIABLE** DEST2 := **STD_LOGIC_VECTOR**(7 DOWNT0 0) := **STD_LOGIC_VECTOR**(SIG);
 - La variable de DEST2 contient les valeurs binaires « 1000 0001 ».

Exemple - Addition 8 bits sans retenue

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL ;

ENTITY add_8b_8b_8b IS
    PORT (
        A : IN  std_logic_vector(7 downto 0);
        B : IN  std_logic_vector(7 downto 0);
        C : OUT std_logic_vector(7 downto 0)
    );
END add_8b_8b_8b;

ARCHITECTURE arch OF add_8b_8b_8b IS
BEGIN

    PROCESS (A, B)
    BEGIN
        C <= STD_LOGIC_VECTOR( UNSIGNED(A) + UNSIGNED(B) );
    END PROCESS;

END arch;
```

Les entrées sont typées puis
additionnées avant d'être à
nouveau re-typées

Exemple - Addition 8 bits sans retenue

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL ;

ENTITY add_8b_8b_8b IS
    PORT (
        A : IN  std_logic_vector(7 downto 0);
        B : IN  std_logic_vector(7 downto 0);
        C : OUT std_logic_vector(7 downto 0)
    );
END add_8b_8b_8b;

ARCHITECTURE arch OF add_8b_8b_8b IS
BEGIN

    PROCESS (A, B)
        VARIABLE D : unsigned(7 downto 0)
        VARIABLE E : unsigned(7 downto 0)
    BEGIN
        D := UNSIGNED(A);
        E := UNSIGNED(B);
        C <= STD_LOGIC_VECTOR( D + E );
    END PROCESS;

END arch;
```

Le processus d'addition peut se décomposer afin d'être plus lisible

Déclaration des variables intermédiaires

Typage des entrées en nombre non signés

Addition des données

Exemple - Addition 8 bits avec propagation de la retenue

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL ;

ENTITY add_9b_8b_8b IS
    PORT (
        A : IN  std_logic_vector(7 downto 0);
        B : IN  std_logic_vector(7 downto 0);
        C : OUT std_logic_vector(8 downto 0)
    );
END add_9b_8b_8b;

ARCHITECTURE arch OF add_9b_8b_8b IS
BEGIN

    PROCESS (A, B)
        VARIABLE D : unsigned(8 downto 0);
        VARIABLE E : unsigned(8 downto 0);
    BEGIN
        D := '0' & UNSIGNED(A);
        E := '0' & UNSIGNED(B);
        C <= STD_LOGIC_VECTOR( D + E );
    END PROCESS;

END arch;
```

Pour récupérer le bit de retenu, il est nécessaire d'étendre manuellement les opérandes

Les données intermédiaires possèdent un bit de plus

Les données sont non signées, on ajoute un bit à 0

L'opération d'addition a maintenant lieu sur 9 bits

Exemple - Addition 8 bits avec propagation de la retenue

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL ;

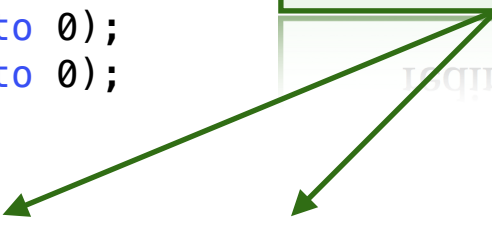
ENTITY add_9b_8b_8b IS
    PORT (
        A : IN  std_logic_vector(7 downto 0);
        B : IN  std_logic_vector(7 downto 0);
        C : OUT std_logic_vector(8 downto 0)
    );
END add_9b_8b_8b;

ARCHITECTURE arch OF add_9b_8b_8b IS
BEGIN

    PROCESS (A, B)
        VARIABLE D : unsigned(7 downto 0);
        VARIABLE E : unsigned(7 downto 0);
    BEGIN
        D := UNSIGNED(A);
        E := UNSIGNED(B);
        C <= STD_LOGIC_VECTOR( RESIZE(D, 9) + RESIZE(E, 9) );
    END PROCESS;

END arch;
```

La fonction RESIZE de la bibliothèque
NUMERIC_STD permet de
redimensionner les données



Exemple - Addition 8 bits avec propagation de la retenue

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL ;

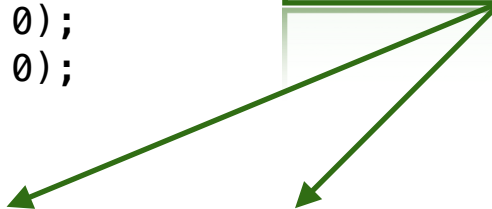
ENTITY add_9b_8b_8b IS
    PORT (
        A : IN  std_logic_vector(7 downto 0);
        B : IN  std_logic_vector(7 downto 0);
        C : OUT std_logic_vector(8 downto 0)
    );
END add_9b_8b_8b;

ARCHITECTURE arch OF add_9b_8b_8b IS
BEGIN

    PROCESS (A, B)
        VARIABLE D : signed(7 downto 0);
        VARIABLE E : signed(7 downto 0);
    BEGIN
        D := SIGNED(A);
        E := SIGNED(B);
        C <= STD_LOGIC_VECTOR( RESIZE(D, 9) + RESIZE(E, 9) );
    END PROCESS;

END arch;
```

Il en va de même sur les données de type SIGNED, mais que fait elle concrètement ?



Exemple - Multiplication 8 bits avec extension

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL ;

ENTITY mul_16b_8b_8b IS
    PORT (
        A : IN  std_logic_vector( 7 downto 0);
        B : IN  std_logic_vector( 7 downto 0);
        C : OUT std_logic_vector(15 downto 0)
    );
END mul_16b_8b_8b;

ARCHITECTURE arch OF mul_16b_8b_8b IS
BEGIN

    PROCESS (A, B)
        VARIABLE D : unsigned(7 downto 0);
        VARIABLE E : unsigned(7 downto 0);
    BEGIN
        D := UNSIGNED(A);
        E := UNSIGNED(B);
        C <= STD_LOGIC_VECTOR( D * E );
    END PROCESS;

END arch;
```

Un opération de multiplication génère un résultat sur $P \times Q$ bits si ses opérandes sont codées sur P et Q bits.

opérations sont codées sur P et Q bits

Exemple - Multiplication 8 bits sans extension

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL ;

ENTITY mul_8b_8b_8b IS
    PORT (
        A : IN  std_logic_vector(7 downto 0);
        B : IN  std_logic_vector(7 downto 0);
        C : OUT std_logic_vector(7 downto 0)
    );
END mul_8b_8b_8b;

ARCHITECTURE arch OF mul_8b_8b_8b IS
BEGIN

    PROCESS (A, B)
        VARIABLE D : unsigned(15 downto 0);
        VARIABLE E : unsigned( 7 downto 0);
    BEGIN
        D := UNSIGNED(A) * UNSIGNED(B);
        E := D(7 downto 0);
        C <= STD_LOGIC_VECTOR( E );
    END PROCESS;

END arch;
```

Pour maintenir une dynamique constante, il est nécessaire de tronquer manuellement le résultat de l'opération

10000000

Exemple - Multiplication 8 bits sans extension

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL ;

ENTITY mul_8b_8b_8b IS
    PORT (
        A : IN  std_logic_vector(7 downto 0);
        B : IN  std_logic_vector(7 downto 0);
        C : OUT std_logic_vector(7 downto 0)
    );
END mul_8b_8b_8b;

ARCHITECTURE arch OF mul_8b_8b_8b IS
BEGIN

    PROCESS (A, B)
        VARIABLE D : unsigned(15 downto 0);
        VARIABLE E : unsigned( 7 downto 0);
    BEGIN
        D := UNSIGNED(A) * UNSIGNED(B);
        E := RESIZE(D, 8);
        C <= STD_LOGIC_VECTOR( E );
    END PROCESS;

END arch;
```

La fonction RESIZE vous permet de réaliser ce traitement MAIS elle conserve les MSB du résultat

Mise en pratique des données UNSIGNED (1)

```
ENTITY counter_v1 IS
    PORT (
        CLK : in  STD_LOGIC;
        RST : in  STD_LOGIC;
        O   : out STD_LOGIC_VECTOR( 7 DOWNTO 0)
    );
END counter_v1;

ARCHITECTURE Behavioral OF counter_v1 IS
    SIGNAL COUNTER : UNSIGNED(7 DOWNTO 0);
BEGIN

    PROCESS(RST, CLK)
    BEGIN
        IF RST = '1' THEN
            COUNTER <= TO_UNSIGNED( 0, 8 );
        ELSIF CLK = '1' AND CLK'EVENT THEN
            IF COUNTER = TO_UNSIGNED( 255, 8 ) THEN
                COUNTER <= TO_UNSIGNED( 0, 8 );
            ELSE
                COUNTER <= COUNTER + TO_UNSIGNED( 1, 8 );
            END IF;
        END IF;
    END PROCESS;

    O <= STD_LOGIC_VECTOR( COUNTER );

END Behavioral;
```

On définit le signal de type
UNSIGNED

On initialise le signal à la
valeur 0 codée sur 8 bits

On compare la valeur avec
la borne maximum

On incrémente la valeur
du compteur

Mise en pratique des données UNSIGNED (2)

```
ENTITY counter_v1 IS
    PORT (
        CLK : in  STD_LOGIC;
        RST : in  STD_LOGIC;
        O   : out STD_LOGIC_VECTOR( 7 DOWNT0 0)
    );
END counter_v1;

ARCHITECTURE Behavioral OF counter_v1 IS
    SIGNAL COUNTER : UNSIGNED(7 DOWNT0 0);
BEGIN

    PROCESS(RST, CLK)
    BEGIN
        IF RST = '1' THEN
            COUNTER <= TO_UNSIGNED( 0, 8 );
        ELSIF CLK = '1' AND CLK'EVENT THEN
            COUNTER <= COUNTER + TO_UNSIGNED( 1, 8 );
        END IF;
    END PROCESS;

    O <= STD_LOGIC_VECTOR( COUNTER );

END Behavioral;
```

Codé sur 8 bits, la valeur interne
du compteur rebouche
automatiquement à zéro

automatiquement à zéro

Le type de donnée de type INTEGER (1)

- ⊙ La manipulation des données de type **SIGNED** et **UNSIGNED** n'est pas toujours aisée.
- ⊙ Afin de simplifier l'écriture des traitement arithmétiques, un autre type existe « **INTEGER** »,
 - ➔ Il est plus « sympathique » pour le concepteur,
 - ➔ Il est cependant moins « précis » quand à son implantation matérielle,
 - ➔ Une donnée de type **INTEGER** (sans range) permet de représenter un nombre dans l'intervall $[-2^{31}, +2^{31}-1]$.
 - ➔ **SIGNAL** la_data : **INTEGER** = 73;
- ⊙ On utilise rarement le type **INTEGER** sans spécification de l'intervalle de variation des données,
 - ➔ **SIGNAL** la_data : **INTEGER RANGE** 0 **TO** 124 := 24;
 - ➔ **SIGNAL** la_data : **INTEGER RANGE** -24 **TO** 13 := -4;

Le type de donnée de type INTEGER (2)

- ⊙ Considérons une information numérique stockée ou transmise sous la forme d'un **STD_LOGIC_VECTOR**.
 - ➔ **VARIABLE** STDL : **STD_LOGIC_VECTOR**(7 **DOWNTO** 0) := « 1000 0001 »;
- ⊙ La conversion sous la forme de donnée de type **INTEGER** nécessite une transformation intermédiaire:
 - ➔ **VARIABLE** INT1 : **INTEGER** := **TO_INTEGER**(**UNSIGNED**(STDL));
 - La variable INT1 contient la valeur numérique 128.
 - ➔ **VARIABLE** INT2 : **INTEGER** := **TO_INTEGER**(**SIGNED**(STDL));
 - La variable INT2 contient la valeur numérique -63.
- ⊙ La conversion type **INTEGER** nécessite une transformation intermédiaire:
 - ➔ **VARIABLE** DEST3 := **STD_LOGIC_VECTOR**(7 **DOWNTO** 0) := **STD_LOGIC_VECTOR**(**TO_UNSIGNED**(INT1, 8));
 - La variable DEST3 contient les valeurs logiques « 1000 0001 ».
 - ➔ **VARIABLE** DEST4 := **STD_LOGIC_VECTOR**(7 **DOWNTO** 0) := **STD_LOGIC_VECTOR**(**TO_SIGNED**(INT2, 8));
 - La variable DEST4 contient les valeurs logiques « 1000 0001 ».

Le type de donnée INTEGER

```
ENTITY counter_v2 IS
    PORT (
        CLK : in  STD_LOGIC;
        RST : in  STD_LOGIC;
        O   : out STD_LOGIC_VECTOR( 7 DOWNTO 0)
    );
END counter_v2;

ARCHITECTURE Behavioral OF counter_v2 IS
    SIGNAL COUNTER : INTEGER RANGE 0 TO 255;
BEGIN

    PROCESS(RST, CLK)
    BEGIN
        IF RST = '1' THEN
            COUNTER <= 0;
        ELSIF CLK = '1' AND CLK'EVENT THEN
            IF COUNTER = 255 THEN
                COUNTER <= 0;
            ELSE
                COUNTER <= COUNTER + 1;
            END IF;
        END IF;
    END PROCESS;

    O <= STD_LOGIC_VECTOR( TO_UNSIGNED(COUNTER,8) );

END Behavioral;
```

La manipulation des données numériques est plus simple

La teste avec la valeur maximum est obligatoire sinon cela génère des erreurs à la simulation.

On doit convertir la donnée entière sur 8 bits pour la transmettre en sortie.

Un peu de généricité...

Définition des constantes

- ⊙ Afin de rendre le code VHDL plus générique et donc plus réutilisable, différentes solutions existent.
- ⊙ La spécification **CONSTANT** peut être employé à la place des traditionnels **SIGNAL** ou **VARIABLE** afin de définir une variable dont la valeur ne peut être modifiée.
- ⊙ Il est possible de définir le type des autres données à partir de cette donnée constante.

```
ARCHITECTURE arch OF module IS
    CONSTANT N : INTEGER := 8;
    SIGNAL      B : STD_LOGIC_VECTOR(N-1 DOWNT0 0);
BEGIN
```

Définition des constantes

```
ENTITY counter_v1 IS
    PORT (
        CLK : in  STD_LOGIC;
        RST : in  STD_LOGIC;
        O   : out STD_LOGIC_VECTOR( 7 DOWNTO 0 )
    );
END counter_v1;

ARCHITECTURE Behavioral OF counter_v1 IS
    CONSTANT N      : INTEGER := 8;
    CONSTANT ZERO   : UNSIGNED(N-1 DOWNTO 0) := TO_UNSIGNED(0, N);
    CONSTANT ONE    : UNSIGNED(N-1 DOWNTO 0) := TO_UNSIGNED(0, N);
    SIGNAL COUNTER : UNSIGNED(N-1 DOWNTO 0);
BEGIN

    PROCESS(RST, CLK)
    BEGIN
        IF RST = '1' THEN
            COUNTER <= ZERO;
        ELSIF CLK = '1' AND CLK'EVENT THEN
            COUNTER <= COUNTER + ONE;
        END IF;
    END PROCESS;

    O <= STD_LOGIC_VECTOR( COUNTER );

END Behavioral;
```

Modifier la valeur de N modifie
l'ensemble du comportement
interne du compteur

Facile et rapide à adapter pour
un nouveau projet

Toutes les instances de ce module
posséderont les même
caractéristiques

Description d'un compteur
semi-générique

Déclaration de modules VHDL génériques

- ⊙ Il est possible de **rendre un module totalement générique** en spécifiant ses paramètres propres lors de son instantiation e.g. un module xor N bits.
- ⊙ En plus de la déclaration des E/S dans l'entité, il faut spécifier les paramètres génériques du module,
- ⊙ Attention, il ne faut pas abuser de cette possibilité qui complexifie:
 - L'écriture du VHDL dans le module,
 - Le débogage du code VHDL écrit,
 - Son utilisation dans les cas simple.
- ⊙ La spécification de valeurs par défaut est possible.

Déclaration de modules VHDL génériques

```
ENTITY counter_v3 IS
    GENERIC(
        N : INTEGER := 8
    );
    PORT (
        CLK : in  STD_LOGIC;
        RST : in  STD_LOGIC;
        O   : out STD_LOGIC_VECTOR(N-1 DOWNTO 0)
    );
END counter_v3;

ARCHITECTURE Behavioral OF counter_v1 IS
    CONSTANT ZERO  : UNSIGNED(N-1 DOWNTO 0) := TO_UNISGNED(0, N);
    CONSTANT ONE   : UNSIGNED(N-1 DOWNTO 0) := TO_UNISGNED(0, N);
    SIGNAL COUNTER : UNSIGNED(N-1 DOWNTO 0);
BEGIN

    PROCESS(RST, CLK)
    BEGIN
        IF RST = '1' THEN
            COUNTER <= ZERO;
        ELSIF CLK = '1' AND CLK'EVENT THEN
            COUNTER <= COUNTER + ONE;
        END IF;
    END PROCESS;

    O <= STD_LOGIC_VECTOR( COUNTER );

END Behavioral;
```

Déclaration de modules VHDL génériques

```
ENTITY counter_v4 IS
    GENERIC(
        N : INTEGER := 8
    );
    PORT (
        CLK : in  STD_LOGIC;
        RST : in  STD_LOGIC;
        O   : out STD_LOGIC_VECTOR(N-1 DOWNTO 0)
    );
END counter_v4;

ARCHITECTURE Behavioral OF counter_v4 IS
    SIGNAL COUNTER : INTEGER RANGE 0 TO (2**N)-1;
BEGIN

    PROCESS(RST, CLK)
    BEGIN
        IF RST = '1' THEN
            COUNTER <= 0;
        ELSIF CLK = '1' AND CLK'EVENT THEN
            IF COUNTER = (2**N)-1 THEN
                COUNTER <= 0;
            ELSE
                COUNTER <= COUNTER + 1;
            END IF;
        END IF;
    END PROCESS;

    O <= STD_LOGIC_VECTOR( TO_UNISGNED(COUNTER, N) );

END Behavioral;
```

Instanciation de modules VHDL génériques

- La configuration des modules est réalisé à la synthèse grace aux paramètre d'instanciation,
- Deux possibilités s'offrent à VOUS:
 - Utiliser les paramètres par défaut, donc pas de **GENERIC MAP**,
 - Spécifier vos propres valeurs pour les paramètres.
- Utiliser des génériques complexifie le développement des modules mais accroît leur réutilisation future.

```
-- COMPTEUR DE 0...255
c1 : ENTITY work.counter_v4
    PORT MAP(
        CLK => CLK,
        RST => RST,
        O   => VALUE
    );

-- COMPTEUR DE 0...1023
c2 : ENTITY work.counter_v4
    GENERIC MAP(
        N => 10
    )
    PORT MAP(
        CLK => CLK,
        RST => RST,
        O   => VALUE
    );
```

Configuration des architectures à l'aide de packages

- ⦿ La généricité des modules est intéressante mais peut être difficile à mettre en oeuvre dans un projet complexe.
- ⦿ Dans ce cas de figure on préfère souvent faire usage de package, cela « équivaut » au `#define` et au « `.h` » en langage C.
- ⦿ Un package est un fichier VHDL à part. On include dans l'entête des modules qui ont besoin des informations définies dedans.

Configuration des architectures à l'aide de packages

```
library IEEE;  
use IEEE.STD_LOGIC_1164.all;  
  
package conf_counter is  
  
    CONSTANT N : INTEGER := 8;  
  
end conf_counter;
```

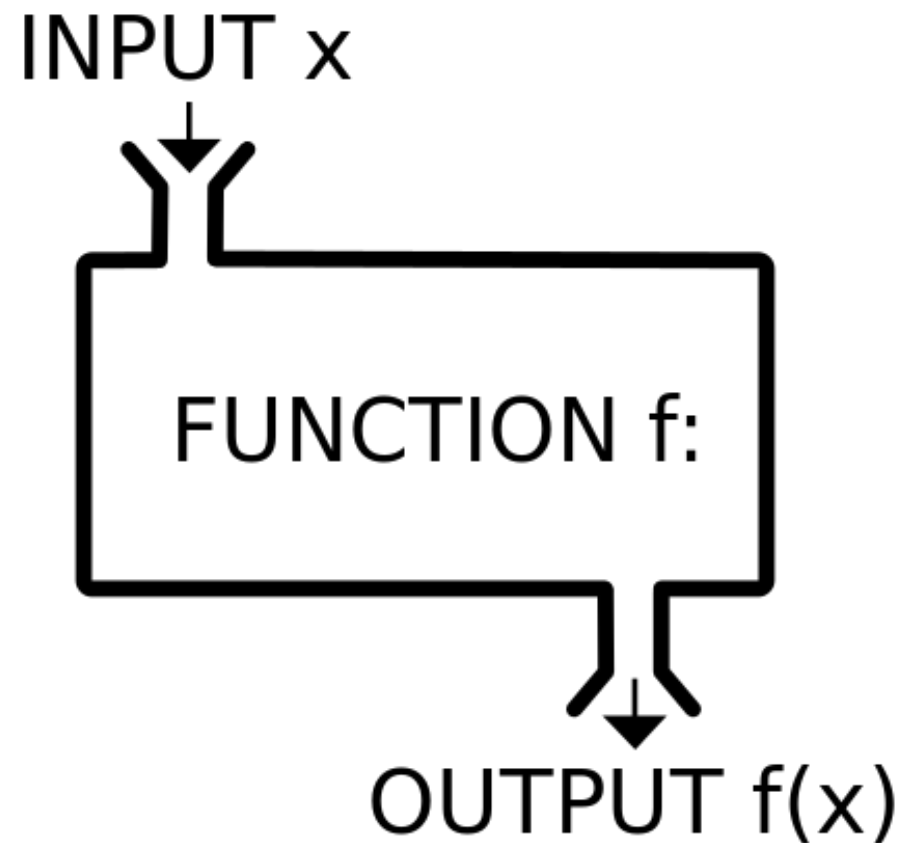
On crée un fichier VHDL contenant les informations communes
« conf_counter.vhd »

On spécifie dans notre module que l'on souhaite utiliser le package

```
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;  
use IEEE.NUMERIC_STD.ALL;  
use WORK.conf_counter.ALL;  
  
ENTITY counter_v4 IS  
    PORT (  
        CLK : in  STD_LOGIC;  
        RST : in  STD_LOGIC;  
        O   : out STD_LOGIC_VECTOR(N-1 DOWNT0 0)  
    );  
END counter_v4;
```

Définition de fonctions en VHDL

- On peut définir des fonctions en VHDL afin de simplifier les traitements récurrents
 - Cela améliore grandement la lisibilité du code source,
 - Cela simplifie les étapes de debug,
- Le code factorisé sera déroulé à la synthèse du module,
 - Une fonction appelée X fois implique X implantations distinctes dans l'architecture numérique !



Définition de fonctions en VHDL

```
ENTITY min_v1 IS
  PORT (
    A : in  STD_LOGIC_VECTOR(7 DOWNTO 0);
    B : in  STD_LOGIC_VECTOR(7 DOWNTO 0);
    C : in  STD_LOGIC_VECTOR(7 DOWNTO 0);
    D : in  STD_LOGIC_VECTOR(7 DOWNTO 0);
    S : out STD_LOGIC_VECTOR(7 DOWNTO 0)
  );
END min_v1;
```

Le composant min_v1 doit réaliser le calcul du minimum entre 4 valeurs non signées

```
ARCHITECTURE Behavioral OF min_v1 IS
BEGIN

  PROCESS(A, B, C, D)
    VARIABLE T1, T2, T3 : STD_LOGIC_VECTOR(7 DOWNTO 0);
  BEGIN
    IF UNSIGNED(A) < UNSIGNED(B) THEN
      T1 := A;
    ELSE
      T1 := B;
    END IF;
    IF UNSIGNED(C) < UNSIGNED(D) THEN
      T2 := C;
    ELSE
      T2 := D;
    END IF;
    IF UNSIGNED(T1) < UNSIGNED(T2) THEN
      S <= T1;
    ELSE
      S <= T2;
    END IF;
  END PROCESS;

END Behavioral;
```

La description du comportement interne du module est répétitive...

Définition de fonctions en VHDL

```
ENTITY min_v1 IS
  PORT (
    A : in  STD_LOGIC_VECTOR(7 DOWNTO 0);
    B : in  STD_LOGIC_VECTOR(7 DOWNTO 0);
    C : in  STD_LOGIC_VECTOR(7 DOWNTO 0);
    D : in  STD_LOGIC_VECTOR(7 DOWNTO 0);
    S : out STD_LOGIC_VECTOR(7 DOWNTO 0);
  );
END min_v1;
```

Le composant min_v1 doit réaliser le calcul du minimum entre 4 valeurs non signées

Déclaration d'une fonction générique permettant de calcul la valeur minimum entre 2 données de type **STD_LOGIC_VECTOR**

Le code source du module est plus clair et plus concis

```
ARCHITECTURE Behavioral OF min_v2 IS

  FUNCTION mini(A, B: STD_LOGIC_VECTOR )
    return STD_LOGIC_VECTOR IS
  BEGIN
    IF UNSIGNED(A) < UNSIGNED(B) THEN
      RETURN A;
    ELSE
      RETURN B;
    END IF;
  END mini;

BEGIN

  PROCESS(A, B, C, D)
    VARIABLE T1, T2 : STD_LOGIC_VECTOR(7 DOWNTO 0);
  BEGIN
    T1 := mini( A,  B);
    T2 := mini( C,  D);
    S  <= mini(T1, T2);
  END PROCESS;

END Behavioral;
```

Utiliser les propriétés des signaux pour plus de généricité

- ⊙ Les signaux et les variables ont des propriétés qui vous permettent de simplifier l'écriture du code source.
- ⊙ Vous avez déjà tous utilisé ce mécanisme l'an dernier...
 - ➔ IF CLOCK'EVENT AND CLOCK = '1' THEN
- ⊙ Beaucoup d'autres propriétés existent, cependant leur nombre et leur nature varient en fonction du type de la donnée.

Utiliser les propriétés des signaux pour plus de généricité

- ◉ S'EVENT is true if signal S has had an event this simulation cycle.
- ◉ T'LEFT is the leftmost value of type T. (Largest if downto)
- ◉ T'RIGHT is the rightmost value of type T. (Smallest if downto)
- ◉ T'IMAGE(X) is a string representation of X that is of type T.
- ◉ T'VALUE(X) is a value of type T converted from the string X.
- ◉ A'LEFT is the leftmost subscript of array A or constrained array type.
- ◉ A'LEFT(N) is the leftmost subscript of dimension N of array A.
- ◉ A'RIGHT is the rightmost subscript of array A or constrained array type.
- ◉ A'RIGHT(N) is the rightmost subscript of dimension N of array A.
- ◉ A'RANGE is the range A'LEFT to A'RIGHT or A'LEFT downto A'RIGHT .
- ◉ A'REVERSE_RANGE is the range of A with to and downto reversed.
- ◉ A'REVERSE_RANGE(N) is the REVERSE_RANGE of dimension N of array A.
- ◉ A'LENGTH is the integer value of the number of elements in array A.

Utiliser les propriétés des signaux pour plus de généricité

- Exemples au tableau...

Et le VHDL synthétisable dans tout cela...

- En VHDL on peut décrire « tout et n'importe quoi »,
- Les outils de synthèse logique ne supportent pas tout le VHDL lorsque vous générer votre circuit,
- L'outil réalise un mapping entre le comportement et les ressources matérielles disponibles,
- Par exemple,
 - Les boucles « **for** » sont déroulées,
 - Les boucles « **while** » sont interdites.

Exemple de code source qui ne sera jamais synthétisable !!!

```
-- Clock process definitions
CLK_process :process
    constant CLK_period : time := 10 ns;
begin
    CLK <= '0';
    wait for CLK_period/2;
    CLK <= '1';
    wait for CLK_period/2;
end process;
```