Travaux pratiques : programmation en VHDL, particularités, pièges et fonctionnalités avancées.

L'objectif de cette séance est de se familiariser avec le langage VHDL et ses particularités. Lors de séances précédentes, nous avons passé en revue quelques exemples simples. Nous continuons toujours avec des descriptions simples, mais mettant en évidence à chaque fois une particularité du langage, pour ce qui concerne la première partie. La seconde partie présente des notions avancées : définition et utilisation de composants, paquetages et bibliothèques.
Nous utiliserons le logiciel Max+plus II pour compiler et tester les programmes.

1 Particularités et pièges

1.1 Le process

1.1.1 Influence sur le résultat

Objectif : étudier l'influence du choix de la description d'une fonction simple sur les connexions générées dans le circuit ?

Le process est généralement affecté à la description d'un système séquentielle, mais rien n'empêche de l'utiliser pour celle d'un système combinatoire.

Nous limiterons notre étude à des cas élémentaires. Comparons en particuliers la description la plus simple que nous puissions faire d'une fonction logique ET avec la même fonction décrite dans un process.

Saisir et compiler le programme suivant

```
--------------------------------------------------------------------------------------------------------------
entity ET is
        port (E1, E2    : in bit;
                S       : out bit);
end ET;

architecture ARCH_ET of ET is
begin
        S <= E1 and E2;
end ARCH_ET;
--------------------------------------------------------------------------------------------------------------
```

On laissera au logiciel le choix d'un circuit cible parmi la série MAX7000.

Observons maintenant comment le compilateur a synthétisé la fonction au sein du circuit.
Ouvrir l'éditeur de placement : **Max+plus II / Floorplan Editor** puis **Layout / LAB View**.
Cet éditeur nous donne une vue du placement des portes utilisées au sein des blocs logiques du circuit cible.

On peut retrouver la même information (en moins claire car textuelle) au sein du fichier rapport de compilation du même nom que la description mais d'extension **.rpt** (report file).

Ce fichier texte nous donnera par contre les équations liant les grandeurs.
Ouvrir ce fichier : **Open / File**.

L'aide sur ce fichier est disponible en tapant **F1** puis en sélectionnant **Report file Format**.
Observer les différentes indications fournies dans la partie **Device Summary** et vérifier la cohérence avec les informations de l'éditeur de placement.

Rechercher la partie ****EQUATION****. On trouvera de l'aide sur cette partie de la même manière que précédemment en sélectionnant cette fois **Device Specific Information** puis **Equation Section**.
Noter la relation entre les grandeurs E1, E2 et S (les symboles &, #, ! et $ représentent respectivement les fonctions ET, OU, NON et OU exclusif).

Une fonction combinatoire peut très bien être décrite avec des instructions séquentielles (donc inclues au sein d'un process). Refaire la même étude avec la description suivante :

```
---------------------------------------------------------------------------------------------------------------
entity ET4 is
        port    (E1, E2 : in bit;
                 S      : out bit);
end ET4;

architecture ARCH_ET of ET4 is
begin
        process (E1, E2)
        begin
                case E1='1' and E2='1' is
                        when true       =>      S<='1' ;
                        when others     =>      S<='0' ;
                end case ;
        end process ;
end ARCH_ET;
---------------------------------------------------------------------------------------------------------------
```

Conclure sur l'influence de la description (dans un cas simple) sur l'implantation des fonctions au sein d'un circuit cible.

1.1.2 Liste de sensibilité et instruction wait

Objectif : savoir mettre en place la liste de sensibilité ; utiliser l'instruction « wait » ; synthétiser les déclenchements sur front ; différencier les entrées synchrones et asynchrones.

1.1.2.1 Etude d'un verrou D (D latch)

On souhaite maintenant synthétiser un verrou de type D (D latch). La sortie du verrou doit recopier l'entrée D lorsque l'entrée de chargement LD est au NL1.
Compiler et simuler la description suivante :

```
---------------------------------------------------------------------------------------------------------------
ENTITY D_latch IS
        PORT (  D, LD           : IN BIT;
                S               : OUT BIT:='0');
END D_latch;

--attention ce programme comporte une erreur
```

```
ARCHITECTURE archi OF D_latch IS

BEGIN
        PROCESS(LD)
        BEGIN
                if LD='1' then S<=D;
                end if;
        END PROCESS;
END archi;
-------------------------------------------------------------------------------------------------------------
```

Est-ce le résultat attendu ? Justifier le dysfonctionnement.
Modifier la liste de sensibilité et simuler de nouveau.

1.1.2.2 Etude d'une bascule D (D flip flop)

En vous inspirant des exemples de l'aide mémoire, synthétiser et simuler une bascule D à l'aide d'un process et d'une liste de sensibilité, puis à l'aide de l'instruction wait. Comparer les structures obtenues à celle de la description du verrou.

Synthétiser et simuler maintenant une bascule avec remise à 0 synchrone et asynchrone. Vérifier le fonctionnement.

2 Tests séquentiels

Les instructions de test séquentielles, en particulier la suite d'instruction « if elsif else », semble simple à utiliser ; elle présente cependant quelques pièges.

2.1 Influence de l'ordre des conditions

Objectif : comprendre la dualité « instructions concurrentes », « instructions séquentielles » ; mettre en évidence l'importance de la position mutuelle des conditions dans un test « if elsif else »

On désire réaliser un décodeur 1 vers 4 avec entrée de validation, la sortie sélectionnée se plaçant au NL0 et restant à 1 dans tous les autres cas.
Compiler la description ci-après.

```
-------------------------------------------------------------------------------------------------------------
ENTITY dec IS
        PORT (VALID   : IN BIT;
                SEL     : IN BIT_VECTOR (1 downto 0);
                  S       : OUT BIT_VECTOR (3 downto 0));
END dec;

--attention ce programme comporte une erreur

ARCHITECTURE archi OF dec IS
BEGIN
        PROCESS(VALID, SEL)
        BEGIN
                if SEL=''00'' then S<="1110";
                elsif SEL=''01'' then S<="1101";
                elsif SEL=''10'' then S<="1011";
                elsif SEL=''11'' then S<="0111";
                elsif VALID='0' then S<="1111";
                end if;
        END PROCESS;
END archi;
-------------------------------------------------------------------------------------------------------------
```

Effectuer une simulation fonctionnelle en gardant l'entrée VALID au NL0. Est-ce le résultat attendu ? Justifier le dysfonctionnement et modifier l'ordre des conditions pour rectifier.

Modifier ce programme en utilisant cette fois, la suite d'instruction « case when when others », puis des instructions concurrentes. Le même problème se pose t-il ?

2.1.1 Mémoires implicites

Objectif : comprendre le phénomène de mémoire implicite créé dans un test séquentiel ; : montrer la nécessité, lorsqu'une instruction conditionnelle est utilisée dans un process décrivant un système combinatoire, de définir tous les cas de figure.

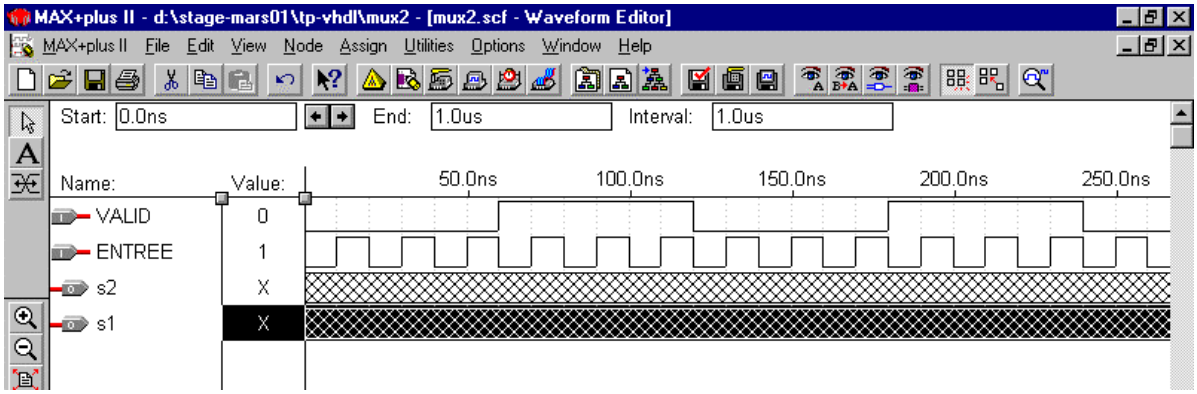
Nous avons vu dans les exemples précédents du verrou et de la bascule D que l'instruction « process » associée aux instructions de test « if elsif else » permettait la création d'une fonction mémoire (bascule ou verrou D). Cette mémoire n'est pas toujours souhaitée et pose parfois problème.

Le programme suivant décrit deux fois la même fonction autorisant ou non le passage d'un signal à travers une porte. Si le passage n'est pas autorisé, la sortie reste au NL0 (il s'agit en fait d'une fonction ET).

```
-------------------------------------------------------------------------------------------------------------
ENTITY porte IS
        PORT (  VALID, ENTREE                   : IN BIT;
                S1, S2                          : OUT BIT);
END porte;

ARCHITECTURE archi OF porte IS
        BEGIN
        PROCESS(VALID, ENTREE)
                BEGIN
                        IF VALID='1'    THEN S1 <=  ENTREE;                     END IF;
                        IF VALID='1'    THEN S2 <=  ENTREE; ELSE S2 <='0';  END IF;
        END PROCESS;
END archi;
-------------------------------------------------------------------------------------------------------------
```

Saisir ce programme le compiler pour effectuer une simulation fonctionnelle avec les chronogrammes suivants :



Justifier la différence entre les deux signaux.
Effectuer maintenant une simulation temporelle (on laissera au logiciel le soin de choisir le circuit cible).
Noter la différence entre S1 et S2.

Justification : l'état de S1 n'est pas imposé dans le cas où VALID est au niveau logique 0.
Pour une simulation fonctionnelle, cela se traduit simplement par une valeur de S1 non définie dans le cas où VALID est au NL0 et une affectation par défaut, c'est à dire à 0, par la suite.
Dans la pratique (voir la simulation temporelle), le logiciel interprète la commande comme la volonté de mettre en mémoire les valeurs de S1 dans le cas où VALID est au NL0. Il utilise donc une bascule supplémentaire pour mémoriser le signal.

Ouvrir le rapport de compilation afin de vérifier la justification précédente ; observer les équations de S1 et S2.

Avec le test séquentiel « case when when others », l'instruction « when others » règle le problème du cas non spécifié. Suivant les compilateurs utilisés, cette instruction est obligatoire ou pas.

Le problème ne se posera pas de la même manière avec une description utilisant des instructions concurrentes. Compiler le programme suivant :

```
---------------------------------------------------------------------------------------------------------------
ENTITY porte_bis IS
        PORT (  VALID, ENTREE         : IN BIT;
                S1, S2                : OUT BIT);
END porte_bis;

ARCHITECTURE archi OF porte_bis IS
        BEGIN
                S1 <=   ENTREE when VALID='1';
                S2 <=   ENTREE when VALID='1'        ELSE
                '0';

END archi;
---------------------------------------------------------------------------------------------------------------
```

On pourra vérifier dans le rapport de compilation que les équations utilisées pour synthétiser S1 et S2 sont identiques. Pour S1, le cas où VALID est au NL0 n'est cependant pas maîtrisé et le compilateur impose un NL0 par défaut en sortie.

3 Boucles spatiales

Objectif : utiliser les descriptions par boucle, utiliser les attributs

L'utilisation des composants programmés tels quel les microcontrolleurs nous a habitués à la description de boucles temporelle d'attente d'événements. En VHDL, ce type de boucle peut être fait de manière très simple au moyen de l'instruction « process ». Ce langage permet aussi la description de redondances de la structure d'un composant, ce qui peut être vu comme une boucle spatiale. Cette notion n'a évidemment aucun sens avec un microcontrolleur, composant pour lequel la structure est figée une fois pour toutes.

Le programme suivant décrit un détecteur de parité qui place la sortie au NL0 lorsque le nombre de « 1 » est pair sur le bus d'entrée E.

```
---------------------------------------------------------------------------------------------------------------

ENTITY parite IS
        PORT (  E       : IN bit_vector (7 downto 0);
                S       : OUT bit);
END parite;
```

```
ARCHITECTURE archi OF parite IS
        SIGNAL X : bit_vector (8 downto 0) ;
        BEGIN
                X(0)<='0';
                boucle:for i in 0 to 7 generate
                        X(i+1)<= X(i) xor E(i);
                end generate;
                S<=X(8);
END archi;
-------------------------------------------------------------------------------------------------------------
```

Ce programme n'est absolument pas optimisé d'un point de vue de la vitesse de fonctionnement et de l'utilisation des ressources du circuit, notre but étant ici d'avoir une description simple.
On peut noter l'absence de déclaration pour la variable i de la boucle.
Compiler et faire une simulation fonctionnelle.
Modifier l'expression des valeurs extrêmes de i en utilisant un attribut.

4 Types et paquetages

Objectifs : maîtriser le type std_logic, comparer avec le type entier ; observer ; utiliser l'instruction generic ; comparer signaux et variables

Le VHDL est un langage très typé, et il n'est pas question à priori d'effectuer des opérations avec de opérateurs de types différents, ce qui peut être pourtant très utile lors de la description de compteurs par exemple, où nous aurons besoins de faire des additions (pour incrémenter le compteur), donc à priori avec un type « entier » et de placer les sortie en haute impédance, donc avec un type « std_logic ».
Les surcharges d'opérateurs permettent de résoudre simplement ce problème.

4.1 Etude d'un compteur

Compiler et synthétiser la description suivante :

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

ENTITY compteur1 IS
        PORT (H                 : IN std_logic;
                S               : OUT std_logic_vector(3 downto 0));
END compteur1;

ARCHITECTURE archi OF compteur1 IS
        signal X :  std_logic_vector (3 downto 0);
        BEGIN
        PROCESS(H)
                BEGIN
                if H'event and H='1' then X<=X+1;
                end if;
        END PROCESS;
        S<=X;
END archi;
-------------------------------------------------------------------------------------------------------------
```

Observer le rapport de compilation et noter en particulier le taux d'occupation du circuit et le nombre de cellules logiques utilisées.

Modifier la déclaration de la sortie en std_logic_vector(0 to 3) et observer les effets sur la simulation (on regardera à la fois le bus S et chaque sortie S_X). On changera le nom du fichier afin de garder une copie intacte pour la suite.

Supprimer une par une la déclaration de chaque paquetage et observer les erreurs obtenues lors de la compilation.

Déclarer les bus S et X comme des entiers sans autres précisions, modifier le programme pour un fonctionnement correct. Compiler et observer le nombre de cellules logiques utilisées et le taux d'occupation du circuit. (ce n'est pas le même circuit que précédemment) dans le rapport de compilation. Comment remédier à cette dégradation ? Essayer votre solution.

Reprendre la description d'origine du compteur et à l'aide de l'instruction « generic » introduire une largeur de bus paramétrable sur N bits. On initialisera cette valeur à 4. Compiler et simuler.

De la même manière, définir un modulo M initialisé à 5. On aura pour cela besoin d'utiliser une conversion de type pour associer la valeur d'un entier (M) à un std_logic_vector (X). On n'oubliera pas de déclarer le paquetage correspondant. Simuler et vérifier le fonctionnement pour M=9.

Modifier le programme afin de remplacer le signal X par une variable X (qui sera donc déclarée à l'intérieur du process, avec « := » comme symbole d'assignation). Y a t-il une différence éventuelle au niveau du comportement du montage et des ressources utilisées.

Que se passe t-il si on remplace le paquetage ieee.std_logic_unsigned par ieee.std_logic_signed ? Justifier.

4.2 Logique trois états

Objectif :utilisation des vecteurs, intérêt du type std_logic.

Proposer une description VHDL d'un « buffer » 3 états pour signaux de 8 bits. On pourra pour cela s'inspirer des divers exemples proposés dans l'annexe de l'aide mémoire. La logique 3 états impose le type std_logic, qui rend indispensable la déclaration de ressources externes.
La manipulation d'octets est facilitée par la déclaration de vecteurs.

La synthèse d'un bus bidirectionnel trois états peut être délicate ; pour vous en convaincre proposez et testez votre propre programme.

Voici la solution que propose Altera.

```
--------------------------------------------------------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY bidir IS
PORT(
                bidir     : INOUT STD_LOGIC_VECTOR (7 DOWNTO 0);
                oe, clk   : IN STD_LOGIC;
                inp       : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                outp      : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END bidir;
```

```
ARCHITECTURE maxpld OF bidir IS
        SIGNAL  a  : STD_LOGIC_VECTOR (7 DOWNTO 0);
        SIGNAL  b  : STD_LOGIC_VECTOR (7 DOWNTO 0);
    BEGIN
    PROCESS(clk)
        BEGIN
        IF clk = '1' AND clk'EVENT THEN
                                        a <= inp;
                                        outp <= b;
        END IF;
    END PROCESS;
    PROCESS (oe, bidir)
        BEGIN
        IF( oe = '0') THEN
            bidir <= "ZZZZZZZZ";
            b <= bidir;
        ELSE
            bidir <= a;
            b <= bidir;
        END IF;
    END PROCESS;
END maxpld;
---------------------------------------------------------------------------------------------------------------------------
```

5 Notions avancées

Objectif : apprendre à utiliser les composants, les paquetages et les bibliothèques.

5.1 Utilisation de composants

Nous allons reprendre le projet décrit dans le premier TP. Celui-ci réalisait l'affichage des secondes sur un afficheur de la carte. Il comprenait un diviseur de fréquence pour passer de la fréquence de 25,175 MHz de la carte à une fréquence de 1 Hz (le rapport cyclique de 0,5 en sortie n'est pas utile), un compteur BCD, un décodeur BCD/7 segments.
Mettre au point un fichier VHDL pour chacune de ces fonctions ; pour le diviseur, on pourra se contenter de reprendre celui donné lors de la première séance.
Pour des raisons de compatibilité entre composants, les entrées et sorties seront de type std_logic ou std_logic_vector.

Tous les fichiers VHDL seront placés dans le même répertoire et on utilisera le concept de composant pour faire la description du projet global.
Le fait de placer tous les éléments dans le même répertoire ne nécessite que la déclaration et l'instantiation des composants utilisés.

En vous inspirant de l'aide mémoire, proposer une description VHDL du projet. Compiler et programmer la carte.

5.2 Utilisation d'un paquetage et bibliothèques

La déclaration des composants, surtout lorsqu'ils sont nombreux, alourdit la description principale.
Il est possible pour éviter ce problème de compiler un paquetage contenant les déclarations.
Généralement, afin d'éviter de mélanger les ressources communes à plusieurs projets et les projets eux-mêmes, on sauvegarde et compile le paquetage dans un répertoire à part.
Ce répertoire constitue alors une bibliothèque de ressources.

Compiler le paquetage dans le répertoire **c:\bibli_perso**.

Sauvegarder et compiler la description principale dans un répertoire différent de tous ceux utilisés jusqu'à présent.
Cette description ne contiendra plus que l'instantiation des composants, ainsi que la déclaration de l'utilisation de la bibliothèque et du paquetage au sein de cette dernière.

Dans l'éditeur, déclarer l'utilisation de ressources externes dans **Users Librairies** du menu **Option.**

Dans le compilateur, déclarer cette utilisation dans **VHDL Netlist Reader Setting** du menu **Interface**.

Compiler et tester sur la carte de développement le projet principal.