Electronique numérique
Initiation à VHDL (1/3)

Le but de ce TD est une première découverte du langage VHDL, et notamment de sa syntaxe. Nous aborderons également des concepts importants, comme la notion de design hiérarchique, à travers l'exemple de l'additionneur binaire. Nous donnerons également un exemple de banc de test ("*testbench*") qui sont le moyen de simuler nos circuits, en créant un laboratoire virtuel, grâce au langage VHDL.

Remarques :

- *Nous travaillerons sous Linux.*
- *VHDL n'est pas associé à un éditeur particulier. Vous pouvez utiliser gEDIT, vim, Emacs, Atom etc.*
- *Pour accéder aux outils Linux de l'Electronique, tapez "module load electronique" dans votre terminal.*
- *La compilation se fera en ligne de commande Linux.*

1 Notion de bibliothèques (library)

La plupart des descriptions VHDL s'appuient sur des **package**s préexistants, fournis par l'organisme IEEE qui est en charge de la standardisation du langage [1]

Ces packages correspondent à des *espaces de nommage* où l'on peut regrouper différents éléments :

- création de nouveaux types de données (**type)**
- déclaration de constantes importantes (**constant)**
- déclaration de fonctions (**function)** et procédures (**procedure)**

Ces packages permettent d'être réutilisés par différents circuits. Pour utiliser les packages, il est au préalable de les compiler dans une bibliothèque (**library**). Dans notre cas, nous n'allons pas créer de nouveaux packages, mais réutiliser ceux fournis par l'IEEE. Il s'agit essentiellement de 2 packages :

- $\boxed{std\_logic\_1164}$: ce package définit entre autre les types les plus usités.

1. L'IEEE (Institute of Electrical and Electronics Engineers) est la plus grande organisation mondiale de professionnels dans le domaine scientifique et technique.

- **std_logic** : ce type correspond au booléen ('0' et '1') , mais également au 'U' (undefined), 'X'(conflit), 'Z' (haute impédance), et d'autres valeurs possibles d'un signal.
- **std_logic_vector** : ce type correspond à un ensemble de signaux, que l'on cherche à manipuler comme un vecteur.
- Différentes fonctions de conversions
- La fonction **r**ising_edge (et **f**alling_edge) qui nous seront utiles pour la description des bascules D.

— *numeric_std* définit quant à lui :

- les types **signed** et **unsigned** permettant de décrire des nombres, avec un nombre de bits arbitraires.
- des fonctions permettant de réaliser directement les opérations arithmétiques (+,-,*, etc), sur ces types **signed** et **unsigned**.
- des fonctions de conversions permettant d'utiliser facilement ces types.

On retrouve fréquemment l'inclusion de ces bibliothèques de la manière suivante :

```
1 library IEEE; --appel a la bibliotheque compilee
2 use IEEE.std_logic_1164.all ; -- on declare utiliser tous les elements du package.
3 use IEEE.numeric_std.all;
```

2 Notion d'entité

L'**entity** d'un circuit modélise son enveloppe extérieure : on y retrouve le nom du circuit, ainsi que la liste des ports d'entrées et de sortie. Ces ports possèdent également un nom, une direction (**in** ou **out** dans notre cas) et un type.

```
1  entity MonCircuit is
2    port (
3      reset_n : in std_logic;
4      horloge : in std_logic;
5      e1 : in std_logic;
6      e2 : in std_logic;
7      e3 : in std_logic_vector(3 downto 0);
8      e4,e5 : in std_logic_vector(3 downto 0);
9      o1 : out std_logic;
10     o2 : out std_logic_vector(3 downto 0);
11     o3 : out std_logic_vector(3 downto 0) --pas de ';' pour le dernier port
12   ); --noter le ; ici
13 end MonCircuit; -- le nom du circuit est rappele ici.
```

Questions :

1. Dessiner l'enveloppe externe du composant décrit par cette entité.

2. Dessiner l'entité d'un demi-additionneur, puis le code VHDL de cette entité.

3 Notion d'architecture

A l'inverse d'une entité, l'**architecture** permet de décrire l'*intérieur* d'un composant. Une **architecture** s'organise syntaxiquement de la manière suivante :

```
1  architecture nom_architecture of nom_entity is
2  --declarations diverses : signaux, types, fonctions,...
3  begin --
4   -- corps de l'architecture
5  end nom_architecture;
```

Le corps de l'architecture contient un ensemble hétérogène d'éléments de description :

— Assignations concurrentes, éventuellement conditionnelles.

— Processus exécutés de manière séquentielle par le simulateur.

— Appels à des sous-composants.

Ces éléments fonctionnant en parallèle, l'ordre dans lequel ils apparaissent au sein de l'architecture n'a pas d'importance.

A titre d'exemple , on présente ici quelques unes des possibilités du langage (restreints à l'UE 1.2), à travers un exemple purement fictif :

```
1  architecture nom_architecture of MonCircuit is
2    signal s1,s2,s3 : std_logic; --declaration de 3 "fils" d'un bit chacun.
3    signal v32 : std_logic_vector(31 downto 0); -- une "nappe" ou "bus" de 32 bits
4    signal cout : std_logic;
5  begin
6
7    -- *** assignations concurrentes ***
8    s1 <= (e1 or not(e2)) xor e3; -- formule utilisant les ports d'I/O
9    s2 <= s1 and e3;
10   v32(1) <= s2; -- on connecte le signal s2 a un des fils de la nappe.
11   o3 <= '1' & v32(2 downto 0); constitution du port de sortie o3.
12
13   -- *** assignations conditionnelles ***
14   s3 <= s1 when s2='0' else e1;
15
16   -- instanciation d'entites :
17   inst1 : entity work.half_adder(logic)
18     port map(
19       clk => horloge,--repris de l'entite
20       a => e1,
21       b => s1,
22       cout => cout, -- signal "local"
23       sum => o1 -- connecte au port de sortie de l'entite
```

```
24      );
25  end nom_architecture;
```

Questions :

4. Rappeler la constitution interne du demi-additionneur.
5. Coder l'architecture de ce demi-additionneur.

4 Décrire un design hiérarchique : cas de l'additionneur

L'exemple d'architecture précédent présentait la notion d'instanciation d'entité : il s'agit de faire appel à des circuits décrits par ailleurs (par exemple dans un autre fichier) pour décrire l'architecture du circuit courant. Il faut voir le procédé comme le fait de prendre sur une étagère un composant préexistant, puis le connecter à des fils d'entrées-sorties par des signaux. On peut instancier autant de composants qu'on le souhaite. Le connexion peut se faire de la manière présentée dans l'exemple précédent : on indique où se connecte le signal en faisant référence au port dit "formel" sur lequel il vient se connecter.

Questions :

6. Dessinez la constitution interne d'un additionneur 1 bit complet, à partir du demi-additionneur.
7. Coder en VHDL l'architecture de cet additionneur 1 bit.
8. Coder en VHDL l'entité et l'architecture d'un additionneur 8 bits.

5 Notion de banc de test

Afin de simuler le circuit précédent (additionneur 8 bits), nous devons créer un banc de test virtuel. Ce banc de test permet de simuler le comportement d'instruments comme des générateurs de signaux qui fourniront des stimuli ou à l'inverse des sondes, oscilloscopes et loggers etc, qui nous permettrons de vérifier que le circuit conçu fonctionne comme on l'espérait. L'écriture de ces bancs de tests permet d'utiliser toute la puissance du langage VHDL : il n'est pas nécessaire que ces instruments soient décrits de la même manière que le circuit que l'on cherche à tester. On dit que ces bancs de tests ne sont pas forcément "synthétisables". C'est ce banc de test qui constituera notre simulation. Concernant l'archictecture du banc de test, il inclut donc :

— le circuit VHDL à tester (souvent labelisé 'DUT' pour "design under test")
— les générateurs de stimuli virtuels : générateurs d'horloge, de reset ou flux de données plus complexes, etc.

— les instruments de mesure virtuels.

Un banc de test pour le circuit d'addition est fourni sous Moodle.

Questions

10. Le testbench présente également une entité. Localisez cette entité. En quoi est-elle particulière ? Comment peut-on l'expliquer ?
11. Un générateur d'horloge est contenu dans le banc de test. Combien de lignes sont nécessaires ? Dessinez le chronogramme de cette horloge.
12. Pour information, le simulateur est dit "à événements discrets". Il fonctionne en exécutant le code de l'instant courant, qui provoque une avalanche d'événements futurs à planifier. Si toutefois, plus aucun événement n'est à traiter, le simulateur va s'arrêter (notion de *famine*). Il est possible de provoquer cette famine afin de sortir proprement d'une simulation. Comment notre banc de test provoque t-il concrètement cet arrêt ?
13. Les autres stimuli sont ici générés dans un *processus* : ce dernier s'exécute de manière séquentiel. Il existe des instructions qui permettent d"'étaler" dans le temps ces actions séquentielles. Dessiner le chronogramme des stimuli.

6 Simulation sous GHDL

L'outil GHDL est un simulateur VHDL open source et gratuit. Il est disponible sur plateforme Linux, MacOS et Windows. Il a été développé par un jeune ingénieur français : Tristan Gingold. C'est un véritable tour de force ! Cocorico !

Pour **analyser** la syntaxe d'un fichier VHDL, il suffit de taper, en ligne de commande :

```
ghdl -a MonFichier.vhd
```

A chaque fichier VHDL va correspondre un fichier binaire, généré par cette commande. C'est exactement ce qui se passe avec le langage C (la même technologie de backend GCC est d'ailleurs utilisée par GHDL).

Il faut ensuite réaliser l'exécutable de simulation, agglomération des fichiers précédents et d'un procédé spécifique de simulation (GRT) : c'est l'*édition de liens* ou **élaboration**. Ne pas oublier d'*analyser* le testbench également !

```
ghdl -a FichierBancDeTest.vhd
ghdl -e NomEntiteBancDeTest
```

Remarque : dans la dernière commande, noter que ce n'est pas le nom du fichier qui est passé en argument, mais le nom de l'entité, écrite dans le fichier. VHDL ne force pas un nommage identique entre le fichier et l'entité qu'il contient.

Enfin, pour *simuler* l'ensemble, on lance le simulateur avec (la lettre R faire référence à "RUN")

```
ghdl -r FichierBancDeTest --wave=test.ghw
```

Un fichier de traces de simulation (waveforms) est alors enregistré sous test.ghw. Pour le visualiser, lancer :

```
gtkwave test.ghw
```

L'ensemble de ces commandes bash peut être regroupé dans un script, auquel on donne les droits en exécution (chmod +x).

Analyser le résultat. Faites preuve d'initiative pour afficher les signaux voulus !

Questions

14. Votre additionneur 8 bits fonctionne-t-il selon vous ?
15. Observez les derniers stimuli du banc de test, ainsi que le chronogramme sous Gtkwave. L'additionneur fonctionne-t-il également pour les nombres signés ?

7 Pour aller plus loin...

Instructions *generate* L'instanciation mutliple de composant peut se révéler fastidieuse : imaginez un additionneur 64 bits ! Il faudrait instancier 64 fois les additionneurs 1 bits ! Fort heureusement, VHDL fournit un moyen de décrire ce procédé répétitif à l'aide d'une construction du langage : l'instruction *generate*. Votre enseignant vous montrera cela. Ces *generate* peuvent s'appuyer également sur des paramètres liés à l'entité, qui permettent de décrire des composants à "géométrie variable". Ainsi, notre entité pourrait être décrire pour un nombre n de bits passé en paramètre **generic**...Mais on peut faire encore plus simple...

Opérateurs du package *numeric_std* L'exercice que nous avons réalisé ici nous a permis de prendre en main le langage, à travers le cas de l'additionneur 8 bits. Cet additionneur a été construit de toute pièce à partir des équations logiques, que nous connaissions. Cet exercice reste toutefois "académique", car les packages IEEE nous fournissent directement le moyen de réaliser l'addition et autres opérations arithmétiques en utilisant les signes

"+", "-" etc...fort heureusement ! Ces opérations seront également parfaitement synthétisées en portes logiques : le synthétiseur connait les équations aussi bien que vous !

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity test is
6  end test;
7
8  architecture example of test is
9    signal a,b,f : signed(7 downto 0);
10 begin
11   f <= a + b;
12 end example;
```