Exo 1 : Modélisation simple d'une bascule D_Latch

Ce premier exercice vous familiarisera avec les outils de compilation et de simulation de Leapfrog.

• Taper le modèle de la bascule ci-dessous en appelant le fichier de description **bascule.vhd**.

```
-- declaration de l'entité
Entity bascule is
        port (d,clk : in bit;
                q : out bit);
end bascule;

-- description de l'architecture
Architecture simple of bascule is
begin
-- on définit le process copie dans lequel on affecte la valeur des signaux
-- c'est une description comportementale
copie:  process
        begin
                -- on attend une transition sur clk
                -- -> le process est bloqué sur cette instruction tant que clk n'évolue pas
                wait on clk;
                -- si il y a transition on teste si elle est montante
                if clk='1' then
                        -- dans ce cas d est copié sur q après 5 ns
                        q <= d after 5 ns;
                end if;
        end process copie;
end simple;
```

• Compiler le modèle. Vous devez alors avoir dans la librairie WORK l'entité et l'architecture du modèle.

• Essayer d'élaborer le fichier de simulation (Elaborate)

✗ Cela ne marche pas! En effet l'entité de votre modèle possède des signaux (d,clk,q) qui ne sont pas définis au niveau du simulateur. C'est comme si vous essayer d'exécuter un programme en Pascal qui ne possède qu'une procédure ! Pour pouvoir effectuer une simulation il faut définir une description de test

dont l'entité ne possède pas de signaux et dont l'architecture comporte des signaux internes connectés à votre modèle de bascule.

• Taper le modèle de test ci-dessous en appelant le fichier de description **test_bascule.vhd**.

```
entity test_bascule is
-- cette entité n'a pas de signaux car c'est le test
end test_bascule;

architecture test_bench of test_bascule is

component latch                          -- composant que l'on utilise dans le modèle de test
        port( d,clk : in bit;
        q  : out bit);
end component;

-- on doit spécifier quelle entité et quelle architecture on utilise
-- pour le composant. C'est la configuration
for all : latch use entity work.bascule(simple);

-- définition de signaux internes
signal din,dout: bit;
signal horloge : bit;
begin
        -- on relie un composant du type latch à nos signaux c'est une description structurelle
        IC1 :  latch port map (din,horloge,dout);
        -- on donne des valeurs aux signaux, ce sont des descriptions de type flux de données
        horloge <= not horloge after 20 ns;
        din     <= '1','0' after 35 ns,'1' after 70 ns;
end test_bench;
```

• Compiler le modèle. Vous devez alors avoir dans la librairie WORK l'entité et l'architecture du modèle.

• Elaborer le fichier de simulation et lancer cette dernière.

Exo_2 : Modélisation avancée d'une bascule D_Latch

Nous allons reprendre la description de la bascule de l'exercice 1 en lui ajoutant une sortie complémentée et en utilisant des signaux de type **std_logic**.
Pour gagner du temps, vous devez utiliser les descriptions déjà faites en changeant le nom des fichiers. Pour cela, dans une fenêtre Unix et sous le répertoire VHDL exécuter les 2 commandes suivantes :

cp bascule.vhd dlatch.vhd
cp test_bascule.vhd test_dlatch.vhd

• Editer ensuite le fichier dlatch.vhd depuis l'HDL Desktop.
• Rajouter au début du fichier les deux lignes suivantes :

```
library IEEE;
use IEEE.std_logic_1164.all;
```

On fait ainsi référence à une librairie IEEE dans laquelle est défini le type std_logic. Ce dernier est une amélioration du type bit car il est résolu (si deux sources modifient le même signal il n'y a pas de conflit) et c'est un standard dans l'industrie. Il permet en outre d'effectuer plus facilement la synthèse. Plusieurs états sont définis pour ce type de signal dont :

'0'	état bas	(idem type bit)
'1'	état haut	(idem type bit)
'U'	état non initialisé	
'X'	état indéterminé	(c'est dans cet état que se trouve le signal si une source le force à 1 et l'autre à 0)
'Z'	état haute impédance	

• Renommer l'entité bascule->**basculeD** et l'architecture simple->**mediocre**
• Changer le type des signaux en std_logic et ajouter dans l'entité le signal **qb**.
• Ajouter dans le process *copie* un test qui permet d'affecter à **qb** le complément de **q**. On pourra aussi simplifier le process en utilisant un **wait until** au lieu du **wait on** puis **if**…:

```
wait on clk;
if clk='1' then                    wait until clk='1';
      instructions          <->          instructions
end if;                                  -- suite
-- suite
```

• Sauver et compiler le fichier

✗ Cela ne marche pas ! En effet vous avez du dans le process effectuer une affectation du style 'qb <= not q'. Or q est définit en **out**, et donc il ne peut pas être utilisé pour affecter dans le modèle un autre signal. Deux solutions sont possibles pour pallier à ce problème :

- utiliser une variable locale
- mettre le signal q en entrée/sortie dans sa définition : **q inout std_logic**

• Modifier et compiler.

Il faut modifier aussi le fichier de test :

• Changer le nom de l'entité de test en **test_basculeD**.

• Modifier les éléments nécessaires en tenant compte du nouveau modèle de basculeD

☞ Pour que le simulateur donne des résultats exploitables il faut nécessairement initialiser les signaux de test à zéro dans leur déclaration (ex : signal horloge : std_logic :='0' ;)

• Compiler et simuler test_basculeD.

Problème sur le fonctionnement de qb :

La simulation doit vous donner un comportement étrange de qb. En effet qb donne l'air de ne pas suivre la valeur de q de façon immédiate. Ceci s'explique par le fait que lors du réveil d'un process (une clause **wait** dans un process est validée) les signaux gardent leurs valeurs durant tout le déroulement du process jusqu'au prochain wait. Dans notre cas q est modifié par le process, mais qb prend la valeur de q au front montant de l'horloge.

• Faire une nouvelle architecture (architecture **bonne** of basculeD) en ajoutant un second process qui régit qb.

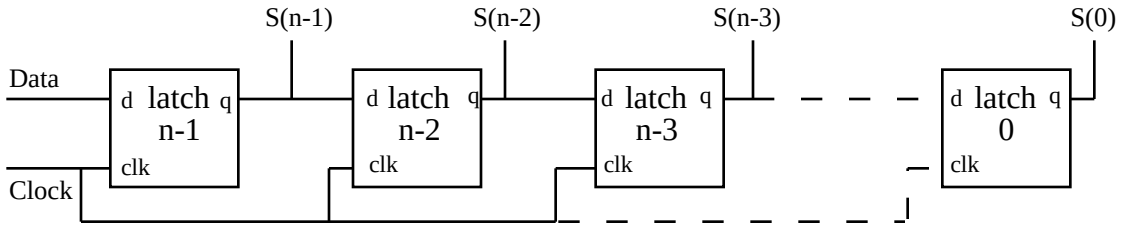
• Compiler et Simuler.

• Faire une autre architecture (architecture **bonne2** of basculeD) en gardant le premier process copie et en ajoutant une instruction concurrente de type flux de données.

• Compiler et Simuler.

Exo 3 : Modélisation d'un registre à décalage

On veut établir le modèle VHDL d'un registre à décalage générique représenté sur la figure suivante :



L'entrée Data permet de charger en série le registre, à chaque coup d'horloge, Clock, la donnée est décalée vers le registre suivant. La sortie est un vecteur de type **std_logic_vector**.
Les bascules Dlatch utilisées font référence au modèle de bascule précédemment établit, dans lequel qb ne sera pas utilisé.

a) cas d'une description structurelle pour un registre sur 4 bits

• Dans un fichier **registre4.vhd** déclarer une entité **registre4** dont les entrés sont **Data** et **Clock**, la sortie **S** est un `std_logic_vector(3 downto 0)`. Le nombre de bascule est donc fixé à 4.
• Dans le même fichier écrire l'architecture structurelle de ce registre, on pourra d'inspirer des indications suivantes :

```
architecture struct of registre4 is
      component pointmem port (d,clk: in std_logic ; q: inout std_logic;
                                          qb:out std_logic);
      end component;
       for all: pointmem use entity work -- à compléter
begin
..... - 4 instanciations de pointmem  en connectant les signaux suivant le schéma de principe
..... - ci-dessus
end struct;
```

✗ Quand on n'utilise pas un signal déclaré en sortie dans un modèle, on peut utiliser le mot clef **open** dans l'appel de ce modèle, ce qui revient à laisser une patte non connectée.

Test du modèle registre4 :

Pour tester le modèle précédent, nous allons décrire un modèle de test.

• Ecrire un fichier de test **test_registre4.vhd** qui possède une entité vide **test_registre4**. L'architecture de ce modèle doit posséder :

- une déclaration de type **component** qui fait référence au modèle registre4
- une déclaration de signaux internes data_in, horloge, sortie....
- une instanciation du registre
- des affectations sur les signaux de test

• Compiler et Simuler

b) cas d'une description structurelle générique

On va dans cette partie écrire un nouveau modèle de la structure registre dont la taille ne sera pas contrainte. Exécuter les commandes suivantes :

cp registre4.vhd registre.vhd

cp test_registre4.vhd.vhd test_registre.vhd

Dans registre.vhd modifier le nom du modèle qui doit s'appeler maintenant **registre**. De même, le signal **S** est un `std_logic_vector(n-1 downto 0)`.

On doit ajouter une définition dans l'entité qui correspond à la généricité de la structure :

```
generic (n:natural:=2);
```

Ceci permet de faire référence dans le modèle au paramètre **n** qui correspond au nombre de bascules utilisées. Ce chiffre n devra être, par la suite, précisé à l'instanciation du modèle, sinon sa valeur sera 2 par défaut.

L'instanciation des n bascules se fait maintenant dans une boucle generate :

```
boucle: for i ..... generate
        -- instanciation d'un pointmem pour la bascule i
    end generate boucle;
```

Test du modèle générique :

En reprenant le fichier test_registre.vhd vous devez maintenant tester la structure générique en fixant le nombre de bascules à 8 (registre sur 8 bits). Pour cela vous devrez inclure :

- une déclaration de type **component** qui fait référence au modèle registre mais dans laquelle on fixera la longueur du vecteur de sortie -> std_logic_vector (7 downto 0)
- une configuration de qui doit pointer sur le modèle **registre**, il faudra en même temps spécifier la valeur de **n** :

```
type for all : decale8 use entity work.....
                    generic map (n=>8);
```

• Compiler et Simuler

c) cas d'une description comportementale

Nous allons écrire une deuxième architecture de **registre** où la description du modèle sera uniquement comportementale. Cette description doit utiliser un seul process :

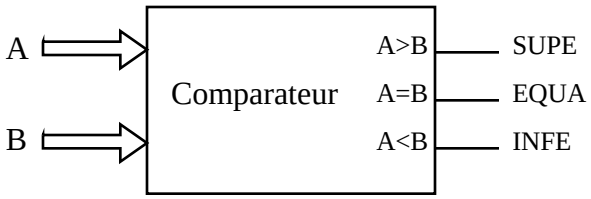
- réveillé sur un front montant de l'horloge
- qui recopie ensuite Data sur le bit S(n-1)
- qui utilise une boucle **for ...loop.....endloop** pour recopier le bit i+1 sur le bit i

• Ecrire cette nouvelle architecture, nommée **beh**, dans le fichier **registre.vhd**
• Changer dans le fichier de test la configuration de registre qui doit maintenant utilisé l'architecture **beh**.
• Compiler et tester.

Exo 4 : Modélisation d'un comparateur 8bits

On veut modéliser un circuit permettant de comparer 2 bus de données de 8 bits, A et B, et de générer 3 signaux de sortie :

EQUA si les données sur les 2 bus sont égales
SUPE si la donnée du bus A est supérieure à la donnée du bus B
INFE si la donnée du bas A est inférieure à la donnée du bus B

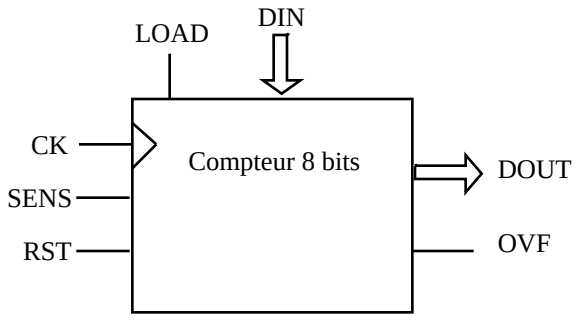


A et B sont du type **std_logic_vector**
EQUA, SUPE et INFE sont du type **std_logic**

• Ecrire l'entité **comp8bit** et l'architecture **concur** de ce modèle en utilisant une description de type flux de données.
• Ecrire le modèle de test **test_comp8bit** comportant une valeur de A et des valeurs de B décrivant tous les codes possibles (0 à 255).
• Compiler et Simuler.

Exo 5 : Modélisation d'un compteur programmable 8bits

On veut modéliser le compteur synchrone suivant :



Description des signaux :

CK : horloge

SENS : à l'état bas, le circuit décompte à chaque transition montante de l'horloge,
à l'état haut, le circuit compte.

RST : signal de remise à zéro du compteur synchrone avec l'horloge (actif haut)

LOAD : signal de commande synchrone de chargement du compteur (actif haut)

DIN : donnée à charger dans le compteur quand la commande LOAD est active.

OVF : signal de dépassement actif haut quand la sortie du compteur est égale à 128

DOUT : sortie sur 8 bits du compteur

➥ Les signaux CK,SENS,RST,LOAD et OVF sont du type **std_logic**

➥ DIN et DOUT sont des entiers contraints de 0 à 255 (Integer range 0 to 255)

• Ecrire l'entité **compteur** et l'architecture **beh** de ce modèle en utilisant une description comportementale. Un seul process doit être utilisé, l'affectation de OVF pourra être décrite par une instruction coucurrante à ce process.

• Ecrire le modèle de test **test_compteur**.

• Compiler et Simuler.

Exo 6 : Synthèse du comparateur 8 bits

Nous allons dans cet exercice effectuer la synthèse du comparateur 8 bits modélisé dans l'exercice 4.

• Dans la fenêtre où vous avez lancé ***LaunchTool*** taper : **synopsys**

• Lancer alors l'outil de synthèse ***design_analyzer &***

• Lancer la commande **READ** en choisissant le mode VHDL et en donnant comme paramètre votre fichier **comp8bit.vhd**

• Effectuer la synthèse . Pour cela choisier dans le menu **Tools** la commande **Design Optimization**.

• Explorer les différents options du logiciel en regardant le résultat de simulation

• Sauvegarder le résultat en choisissant dans la commande **Save As** l'option VHDL (donner comme nom au fichier ***synthese.vhd***).

Si vous voulez plus d'informations sur la synthèse consultez le support de cours sur la synthèse logique sur le serveur Web de l'AIME :

http://www.aime.insa-tlse.fr/home/tournier/Enseignements/TP_synthese/Directives.pdf

Simulation du modèle synthétisé

Il est possible de simuler le résultat de la synthèse de votre modèle en utilisant NC VHDL. On peut de cettte manière s'assurer que l'outil de synthèse a bien fonctionné, et évaluer les temps de propagations dans le circuit.

Repasser dans l'environnement NC VHDL

Compiler le fichier synthese.vhd

➠ Synergy a du créer dans votre librairie WORK une architecture dénommée :

nom_de_l'entité:nom_architecture **syn_concur**

➥ Vérifier

• Changer dans votre fichier de test le nom de l'architecture du comparateur.

• Compiler le fichier test et Simuler