Romain COCOGNE
Camille BRUNIN

ELEC4
Groupe 1

Compte rendu TP3

Introduction

Dans ce TP, nous allons voir comment programmer un multiplicateur binaire hardware sous forme séquentielle.

I) Préparation

A) Le script

Tout d'abord, nous avons écrit un script RTL permettant d'automatiser le processus de vérification du système.

run_file.tcl

```
vsim -t 1ns work.test(bench)
#ajout des signaux dans la fenêtre de simulation
add wave -noupdate -divider {entree du bloc multiplication}
add wave -noupdate test/UUT/clk
add wave -noupdate -radix unsigned -radixshowbase 0 test/UUT/multiplieur
add wave -noupdate -radix unsigned -radixshowbase 0 test/UUT/multiplicande
add wave -noupdate test/UUT/go

add wave -noupdate -divider {sorties du bloc multiplication}
add wave -noupdate -radix unsigned -radixshowbase 0 test/UUT/s
add wave -noupdate -radixshowbase 0 test/UUT/fin

add wave -noupdate -divider {signaux internes}
add wave -noupdate -radixshowbase 0 test/UUT/rst
add wave -noupdate -radixshowbase 0 test/UUT/C
add wave -noupdate -radix binary -radixshowbase 0 test/UUT/Q
add wave -noupdate -radix binary -radixshowbase 0 test/UUT/A
add wave -noupdate -radixshowbase 0 test/UUT/I
add wave -noupdate -radixshowbase 0 test/UUT/etat

#simulation de 200ns
run 200 ns
```

B) L'entité

Nous avons défini une entité *mult* dont les entrées sont définies sur 8 bits et la sortie sur 16 bits. Il ne faut pas oublier d'inclure les packages *std_logic_1164* et *std_logic_unsigned* de la librairie ieee.

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

entity mult is
        port(multiplieur, multiplicande: in std_logic_vector(7 downto 0);
             go,clk,rst: in std_logic;
             s: out std_logic_vector(15 downto 0);
             fin: out std_logic);
end entity;
```

C) L'architecture

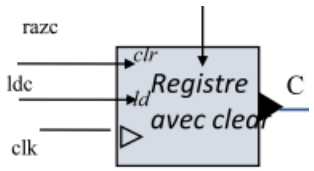
Nous avons ensuite donné une description des différents blocs du chemin de données dans une architecture *rtl* à l'aide d'instructions d'affectation concurrente.

Nous avons décomposé le schéma de la solution RTL du circuit en plusieurs signaux. En effet, on observe 3 registres différents.

- Ainsi, le signal C correspond à la sortie du registre avec clear.

Ld	clr	operation
0	0	aucune
0	1	Remise à 0
1	0	chargement
1	1	interdit

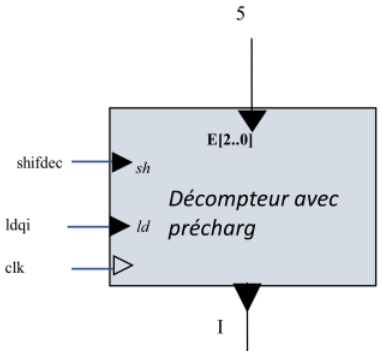
Specification registre avec clear



- Le signal I correspond à la sortie du registre décompteur à préchargement.

Ld	dec	operation
0	0	aucune
0	1	decomptage
1	0	chargement
1	1	interdit

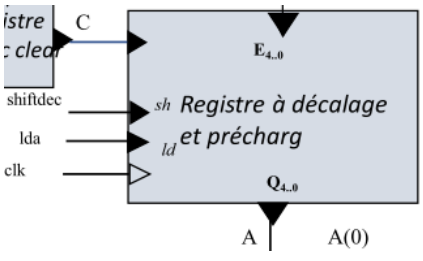
Specification registre decompteurà precharg

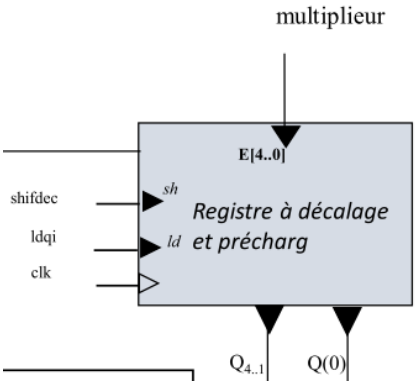


- Enfin, les signaux A et Q sont les sorties des registres à décalage et préchargement, où E et multiplieur sont les entrées.

Ld	sh	operation
0	0	aucune
0	1	decal droit
1	0	chargement
1	1	interdit

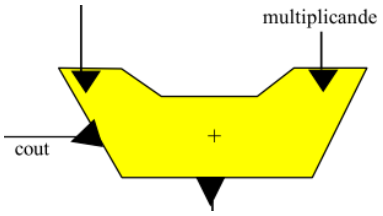
Specification registre decalage et precharg





Après les registres, nous avons dû coder :

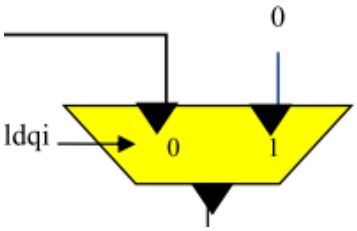
- L'additionneur :



Entrées : A, multiplicande

Sorties : Cout (retenue), add = A+B

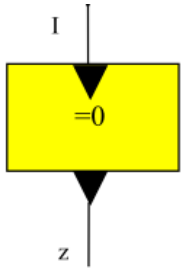
- Le multiplexeur :



Entrées : add, '0', ldqi

Sortie : E = add ou 0 en fonction de ldqi

- Le bloc booléen :



Entrée : I

Sortie : z == 0 ?

C'est ainsi que nous avons pu implémenter tous les signaux.
Ci-dessous le code final :

```
architecture rtl of mult is
signal razc, ldc, shiftdec, lda, ldqi,C,z,f, cout: std_logic := '0';
signal A, Q, E: std_logic_vector(7 downto 0) := "00000000";
signal I: std_logic_vector(7 downto 0);
signal add: std_logic_vector(8 downto 0) := "000000000";
type defetat is (E0,E1,E2);
signal etat, netat: defetat;
begin
        c <= '0' when (ldc='0' and razc='1') and (clk='1' and clk'event)
                else cout when (ldc='1' and razc='0') and (clk='1' and clk'event);

        E <= add(7 downto 0) when ldqi = '0' else "00000000";

        Q <= multiplieur when (ldqi='1' and shiftdec = '0') and (clk='1' and clk'event)
                        else A(0) & Q(7 downto 1) when (ldqi='0' and shiftdec='1') and (clk ='1' and clk'event);

        A <= E when (lda='1' and shiftdec = '0') and (clk='1' and clk'event)
                else C & A(7 downto 1) when (lda='0' and shiftdec='1') and (clk ='1' and clk'event);

        I <= (I-1) when (ldqi='0' and shiftdec='1') and (clk='1' and clk'event)
                   else "00001000" when (ldqi='1' and shiftdec='0') and (clk='1' and clk'event);

        add <= ('0' & A) + ('0' & multiplicande); --on fait un and avec un 0 pour prévenir d'un dépassement
        cout <= add(8);

        z <= '1' when I = "00000000" else '0';

        s(15 downto 8) <= A;
        s(7 downto 1) <= Q(7 downto 1);
        s(0) <= Q(0);
        fin <= f;
```

Remarquons que tous les signaux sont sur 8 ou 16 bits, à part le signal add. En effet, ce signal traduit la sortie de l'additionneur. Or, lors d'une addition, il est possible qu'il y ait un dépassement (c'est-à-dire que le résultat de l'addition de deux nombres binaires sur 8 bits soit un nombre binaire sur 9 bits).

D) Description séquenceur à 2 processus

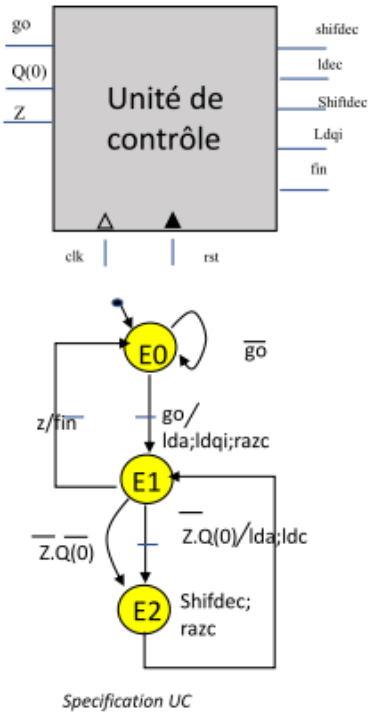
Dans cette partie, nous avons effectué une description du séquenceur à l'aide de deux processus :

- Un processus séquentiel de mémorisation de l'état suivant

```
memorisation:process(clk, rst)
begin
        if(rst='0') then
                etat <= E0;
        elsif(clk='1' and clk'event) then
                etat <= netat;
        end if;
end process;
```

Dans ce processus, nous mémorisons l'état suivant à chaque front montant d'horloge.

- Un processus combinatoire pour la génération des sorties et de l'état suivant



Specification UC

```
combinatoire:process(etat, go, Q(0),z)
begin
        netat <= etat;
        lda <= '0';
        ldc <= '0';
        ldqi <= '0';
        razc <= '0';
        shiftdec <= '0';
        case etat is
                when E0 =>
                        if(go = '1') then
                                lda <= '1';
                                ldqi <= '1';
                                razc <= '1';
                                netat <= E1;
                        end if;
                when E1 =>
                        if(z = '0') then
                                if(Q(0) = '1') then
                                        lda <= '1';
                                        ldc <= '1';
                                        netat <= E2;
                                else
                                        netat <= E2;
                                end if;
                        else
                                f <= '1';
                                netat <= E0;
                        end if;
                when E2 =>
                        shiftdec <= '1';
                        razc <= '1';
                        netat <= E1;
        end case;
end process;
```

Dans ce processus, nous définissions les sorties de l'unité de contrôle et l'état suivant en fonction de l'état courant et des entrées, à l'aide du diagramme de spécification de l'unité de contrôle.

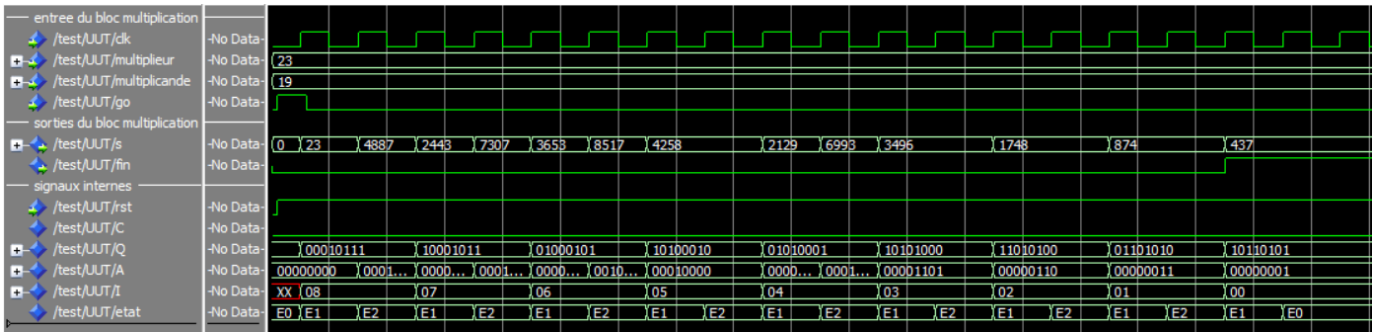
E) Le test

Nous définissons à présent l'architecture de test inspirée de celle de l'énoncé, qui couvre toutes les possibilités.

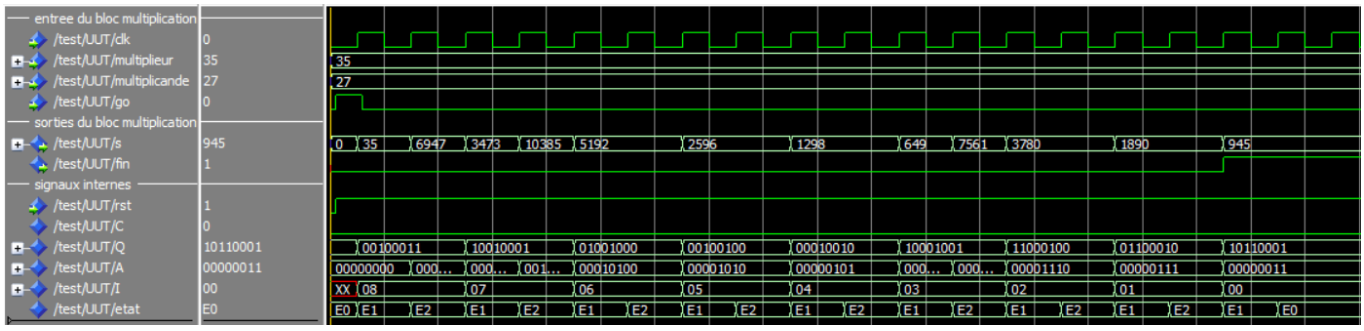
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity test is
end test;

architecture bench of test is
component mult
        port(multiplieur, multiplicande: in std_logic_vector(7 downto 0);
             go, clk, rst: in std_logic;
             s: out std_logic_vector(15 downto 0);
             fin: out std_logic);
end component;
for UUT:mult use entity work.mult(rtl);
signal multiplieur, multiplicande: std_logic_vector(7 downto 0);
signal go, clk, rst, fin: std_logic := '0';
signal s: std_logic_vector(15 downto 0);
begin
        UUT:mult port map(multiplieur=>multiplieur, multiplicande=>multiplicande, go=>go, clk=>clk, rst=>rst, fin=>fin, s=>s);
        clk <= not clk after 5 ns;
        multiplieur <= "00010111";
        multiplicande <= "00010011";
        go <= '1' after 1 ns, '0' after 6 ns;
        rst <= '1' after 1 ns;
end bench;
```

II) Manipulations

Nous arrivons à présent à la simulation, que nous effectuons à l'aide de notre script. Nous pouvons écrire directement dans la console de modelsim : *do run_file.tcl*



Nous remarquons que la simulation nous donne le bon résultat de la multiplication en sortie (signal s). De plus, si nous changeons les deux opérandes en entrées, nous avons également un bon résultat du produit. Ci-dessous un exemple avec 35x27 :



Conclusion

Au travers de la réalisation d’un multiplicateur binaire, ce TP nous a appris à utiliser les scripts dans nos projets de VHDL. Cela sera grandement utile pour améliorer notre productivité et réaliser des projets plus complexes.