UQAC
Université du Québec
à Chicoutimi

Rapport de laboratoire No 4
CPU
Hiver 2023

Conception de systèmes digitaux
6GEI367

Département des Sciences Appliquées
Module d'ingénierie

Travail d'équipe

Maxime Simard
SIMM26050001

Samuel Gaudreault
GAUS09109500

Date de remise : 30 03 2023

Table des matières

Introduction 3
Partie 1 : Processeur simple 3
Processeur 3
AddSub 5
Registre 5
FSM 6
Partie 2 : Testbench 8
Processeur 8
AddSub 9
Registre 10
FSM 12
Partie 3 : Interfacer le circuit 14
Interface 14
Partie 4 : Utilisation de la mémoire 15
Interface mémoire 15
Mémoire 16
Conclusion 17
Références A

Liste des figures

Figure 1 - Code VHDL Processeur 3
Figure 2 - Code VHDL AddSub 5
Figure 3 - Code VHDL Registre 5
Figure 4 - Code VHDL FSM #1 6
Figure 5 - Code VHDL FSM #2 7
Figure 6 - Code VHDL du testbench Processeur 8
Figure 7 - Résultat du testbench Processeur 9
Figure 8 - Code VHDL du testbench AddSub 9
Figure 9 - Résultat du testbench AddSub 10
Figure 10 - Code VHDL du testbench Registre 10
Figure 11 - Résultat du testbench Registre 11
Figure 12 - Code VHDL du testbench FSM 12
Figure 13 - Résultat du testbench FSM 13
Figure 14 - Code VHDL Interface 14
Figure 15 - Code VHDL Interface Mémoire 15
Figure 16 - Mémoire du fichier « inst_mem.mif » 16

Introduction

Dans ce laboratoire, nous devrons concevoir et mettre en place un processeur simple. Après avoir implémenter et tester à l'aide de « testbenchs » les instructions « MV », « MVI », « ADD » et « SUB », nous devrons interfacer ce processeur sur le FPGA. Pour finir, nous devrons ajouter une mémoire au processeur.

Partie 1 : Processeur simple

Processeur

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.fsm_constants.ALL;

PACKAGE processeur_constants IS
    COMPONENT processeur IS
        PORT (
            run, clk, rst : IN STD_LOGIC;
            din : IN STD_LOGIC_VECTOR(8 DOWNTO 0);
            buswire : BUFFER STD_LOGIC_VECTOR(8 DOWNTO 0);
            done : OUT STD_LOGIC
        );
    END COMPONENT;
END PACKAGE;
-------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.ff.ALL;
USE work.ch8.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE work.registre_constants.ALL;
USE work.addsub_constants.ALL;
USE work.muxb10_constants.ALL;
USE work.fsm_constants.ALL;

ENTITY processeur IS
    PORT (
        run, clk, rst : IN STD_LOGIC;
        din : IN STD_LOGIC_VECTOR(8 DOWNTO 0);
        buswire : BUFFER STD_LOGIC_VECTOR(8 DOWNTO 0);
        done : OUT STD_LOGIC
    );
END processeur;

ARCHITECTURE arch OF processeur IS
    SIGNAL r0in, r1in, r2in, r3in, r4in, r5in, r6in, r7in, ain, gin, irin, mode : STD_LOGIC := '0';
    SIGNAL inControl : STD_LOGIC_VECTOR(10 DOWNTO 0);
    SIGNAL outControl : STD_LOGIC_VECTOR(9 DOWNTO 0);
    SIGNAL r0out, r1out, r2out, r3out, r4out, r5out, r6out, r7out, aout, gout, irout, addsubout : STD_LOGIC_VECTOR(8 DOWNTO 0);
    SIGNAL sb : STD_LOGIC_VECTOR(1 DOWNTO 0);
    SIGNAL i : STD_LOGIC_VECTOR(2 DOWNTO 0);
    SIGNAL xReg, yReg : STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
    r0 : registre PORT MAP(buswire, clk, inControl(0), r0out);
    r1 : registre PORT MAP(buswire, clk, inControl(1), r1out);
    r2 : registre PORT MAP(buswire, clk, inControl(2), r2out);
    r3 : registre PORT MAP(buswire, clk, inControl(3), r3out);
    r4 : registre PORT MAP(buswire, clk, inControl(4), r4out);
    r5 : registre PORT MAP(buswire, clk, inControl(5), r5out);
    r6 : registre PORT MAP(buswire, clk, inControl(6), r6out);
    r7 : registre PORT MAP(buswire, clk, inControl(7), r7out);

    a : registre PORT MAP(buswire, clk, inControl(8), aout);
    addsub0 : addsub PORT MAP(aout, buswire, mode, addsubout);
    g : registre PORT MAP(addsubout, clk, inControl(9), gout);

    ir : registre PORT MAP(din, clk, inControl(10), irout);

    i <= irout(8 DOWNTO 6);
    decX : Dec GENERIC MAP(3, 8) PORT MAP(irout(5 DOWNTO 3), xReg);
    decY : Dec GENERIC MAP(3, 8) PORT MAP(irout(2 DOWNTO 0), yReg);

    fsmProc : fsm PORT MAP(i, xReg, yReg, run, clk, rst, inControl, outControl, done, mode);

    enc : Enc164 PORT MAP("000000" & outControl, sb);
    mux : Muxb10 PORT MAP(gout, "000000" & irout(2 DOWNTO 0), r7out, r6out, r5out, r4out, r3out, r2out, r1out, r0out, sb, buswire);
END arch;
```

Figure 1 - Code VHDL Processeur

Ce code VHDL prend en entrée 3 bits (run, clk et rst) et un vecteur de 9 bits permettant d'envoyer des commandes dans le format IIIXXXYYY, où III est l'instruction, XXX est l'adresse du registre X et YYY est l'adresse du registre Y. En buffer, on retrouve un vecteur de 9 bits représentant le bus de données principal du processeur. C'est par ce bus que les différents registres et composantes du processeur communiquent ensemble et partagent des données. Finalement, il y a en sortie l'état « done » du processeur. Si ce bit est à « 1 », cela signifie que le processeur n'est pas en train d'effectuer une opération, et donc qu'il est prêt à recevoir une nouvelle instruction.

Dans l'architecture du processeur, on peut voir qu'il instancie huit registres (r0 à r7). Ces registres reçoivent en entrée le bus de données, et leur activation est contrôlée par un signal « inControl » géré par la FSM du processeur.

Il y a un registre « a » et « g » pour l'ALU du processeur ainsi qu'une entité « addsub » permettant l'addition et la soustraction entre les registres. Le registre « a » permet de garder la valeur du premier registre envoyé sur le bus de données, puis la deuxième valeur sera envoyée par le bus directement dans l'entité « addsub ». Finalement, on sort la valeur de « addsub » dans le registre « g » pour garder en mémoire le résultat de l'opération. Le mode addition et soustraction est géré par un bit « mode ».

Une FSM gère les différents états du processeur et contrôle les signaux selon les instructions données. Les entrées de la FSM sont prétraitées par un décodeur qui transforme les 3 bits d'adresse X et Y en « one hot ». La sortie « inControl » active les registres à modifier, et le « outControl » permet de sélectionner d'où proviendront les données du bus en « one hot ».

Le mux prend entrée une valeur prétraitée par un encodeur qui transforme le « one hot » sortant de la FSM en vecteur de 3 bits pouvant être utilisé pour indiquer quel registre devrait être sur le bus de données.

AddSub

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3
4  PACKAGE addsub_constants IS
5     COMPONENT addsub IS
6        GENERIC (n : INTEGER := 9);
7        PORT (
8           a, b : IN STD_LOGIC_VECTOR(n - 1 DOWNTO 0);
9           mode : IN STD_LOGIC;
10          c : OUT STD_LOGIC_VECTOR(8 DOWNTO 0)
11       );
12    END COMPONENT;
13 END PACKAGE;
14 ------------------------------------------------------
15 LIBRARY ieee;
16 USE ieee.std_logic_1164.ALL;
17 USE work.ch8.ALL;
18 USE IEEE.STD_LOGIC_UNSIGNED.ALL;
19
20 ENTITY addsub IS
21    GENERIC (n : INTEGER := 9);
22    PORT (
23       a, b : IN STD_LOGIC_VECTOR(n - 1 DOWNTO 0);
24       mode : IN STD_LOGIC;
25       c : OUT STD_LOGIC_VECTOR(n - 1 DOWNTO 0)
26    );
27 END addsub;
28
29 ARCHITECTURE arch OF addsub IS
30 BEGIN
31    c <= a + b WHEN mode ELSE
32       a - b;
33 END arch;
```

Figure 2 - Code VHDL AddSub

L'entité « addsub » prend en entrée un vecteur a et b, ainsi que le mode (addition ou soustraction). Son architecture consiste à faire une simple addition ou soustraction de vecteur de « n » longueur, puis de sortir ce résultat dans le signal c.

Registre

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3
4  PACKAGE registre_constants IS
5     COMPONENT registre IS
6        GENERIC (n : INTEGER := 9);
7        PORT (
8           rin : IN STD_LOGIC_VECTOR(n - 1 DOWNTO 0);
9           clk, activated : IN STD_LOGIC;
10          rout : OUT STD_LOGIC_VECTOR(n - 1 DOWNTO 0)
11       );
12    END COMPONENT;
13 END PACKAGE;
14 ------------------------------------------------------
15 LIBRARY ieee;
16 USE ieee.std_logic_1164.ALL;
17 USE work.ff.ALL;
18 USE work.sseg_constants.ALL;
19 USE IEEE.STD_LOGIC_UNSIGNED.ALL;
20
21 ENTITY registre IS
22    GENERIC (n : INTEGER := 9);
23    PORT (
24       rin : IN STD_LOGIC_VECTOR(n - 1 DOWNTO 0);
25       clk, activated : IN STD_LOGIC;
26       rout : OUT STD_LOGIC_VECTOR(n - 1 DOWNTO 0)
27    );
28 END registre;
29
30 ARCHITECTURE arch OF registre IS
31 BEGIN
32    PROCESS (ALL) BEGIN
33       IF rising_edge(clk) THEN
34          IF activated THEN
35             rout <= rin;
36          END IF;
37       END IF;
38    END PROCESS;
39 END arch;
```

Figure 3 - Code VHDL Registre

L'entité de registre prend en entrée un vecteur « rin » qui représente les nouvelles données, un bit « clk » et « activated » qui représente l'horloge du processeur et l'activation du registre permettant son écriture, et

sort un vecteur « rout » représentant la valeur courante du registre. Lors d'un front montant de l'horloge, si le registre est activé, on remplace la valeur actuelle du registre par la nouvelle valeur.

FSM

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3
4  PACKAGE fsm_constants IS
5     TYPE State_type IS (T0, T1, T2, T3);
6     COMPONENT fsm IS
7        PORT (
8           i : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
9           xReg, yReg : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
10          run, clk, rst : IN STD_LOGIC;
11          inControl : OUT STD_LOGIC_VECTOR(10 DOWNTO 0);
12          outControl : OUT STD_LOGIC_VECTOR(9 DOWNTO 0);
13          done : OUT STD_LOGIC;
14          mode : OUT STD_LOGIC;
15          stateValue : OUT State_type
16       );
17    END COMPONENT;
18 END PACKAGE;
19
20 ----------------------------------------------------
21
22 LIBRARY ieee;
23 USE ieee.std_logic_1164.ALL;
24 USE work.ch8.ALL;
25 USE work.ff.ALL;
26 USE ieee.numeric_std.ALL;
27 USE work.fsm_constants.ALL;
28 ENTITY fsm IS
29    PORT (
30       i : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
31       xReg, yReg : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
32       run, clk, rst : IN STD_LOGIC;
33       inControl : OUT STD_LOGIC_VECTOR(10 DOWNTO 0);
34       outControl : OUT STD_LOGIC_VECTOR(9 DOWNTO 0);
35       done : OUT STD_LOGIC;
36       mode : OUT STD_LOGIC;
37       stateValue : OUT work.fsm_constants.State_type
38    );
39 END fsm;
40
41 ARCHITECTURE arch OF fsm IS
42    SIGNAL Tstep_Q, Tstep_D : State_type;
43 BEGIN
44    statetable : PROCESS (Tstep_Q, i, run, rst)
45    BEGIN
46       CASE Tstep_Q IS
47          WHEN T0 =>
48             IF run = '0' THEN
49                Tstep_D <= T0;
50             ELSE
51                Tstep_D <= T1;
52             END IF;
53
54          WHEN T1 =>
55             IF run = '0' THEN
56                Tstep_D <= T1;
57             ELSIF i = "000" OR i = "001" THEN
58                Tstep_D <= T0;
59             ELSIF i = "010" OR i = "011" THEN
60                Tstep_D <= T2;
61             END IF;
62
63          WHEN T2 =>
64             IF run = '0' THEN
65                Tstep_D <= T2;
66             ELSIF i = "010" OR i = "011" THEN
67                Tstep_D <= T3;
68             END IF;
69
70          WHEN T3 =>
71             IF run = '0' THEN
72                Tstep_D <= T3;
73             ELSE
74                Tstep_D <= T0;
75             END IF;
76       END CASE;
77    END PROCESS;
```

Figure 4 - Code VHDL FSM #1

La FSM prend en entrée un vecteur de 3 bits « i » représentant l'instruction actuelle, deux vecteurs de 8 bits « xReg » et « yReg » représentant les adresses de registres en « one hot ». Il y a aussi 3 bits (run, clk, rst) permettant d'activer le processeur, de faire un avancé l'état d'une étape et de réinitialiser la FSM. La FSM sort finalement le « inControl » contrôlant en quels registres doivent être activés, et le « outControl » contrôlant quel registre sera sur le bus. Le mode permet de contrôler le mode de l'ALU dans le processeur.

Dans le « package », on définit le « State_type » qui contient « T0 », « T1 », « T2 » et « T3 ». Ces temps sont les états possibles de la FSM, et donc du processeur. Dans le processus « statetable », on détermine le prochain état de la FSM selon le bit « run » et « rst », mais aussi l'état et l'instruction actuel.

```
78
79  controlsignals : PROCESS (Tstep_Q, i, xReg, yReg)
80  BEGIN
81      CASE Tstep_Q IS
82          WHEN T0 =>
83              done <= '1';
84              inControl <= "10000000000";
85              outControl <= "0100000000";
86          WHEN T1 =>
87              done <= '0';
88
89              CASE i IS
90                  WHEN "000" =>
91                      inControl <= "000" & xReg;
92                      outControl <= "00" & yReg;
93                  WHEN "001" =>
94                      inControl <= "000" & xReg;
95                      outControl <= "0100000000";
96                  WHEN "010" | "011" =>
97                      inControl <= "00100000000";
98                      outControl <= "00" & xReg;
99                      IF i = "010" THEN
100                         mode <= '1';
101                     ELSE
102                         mode <= '0';
103                     END IF;
104                 WHEN OTHERS =>
105                     inControl <= (OTHERS => '0');
106                     outControl <= (OTHERS => '0');
107             END CASE;
108         WHEN T2 =>
109             CASE i IS
110                 WHEN "010" | "011" =>
111                     inControl <= "01000000000";
112                     outControl <= "00" & yReg;
113                 WHEN OTHERS =>
114                     inControl <= (OTHERS => '0');
115                     outControl <= (OTHERS => '0');
116             END CASE;
117         WHEN T3 =>
118             CASE i IS
119                 WHEN "010" | "011" =>
120                     inControl <= "000" & xReg;
121                     outControl <= "1000000000";
122                 WHEN OTHERS =>
123                     inControl <= (OTHERS => '0');
124                     outControl <= (OTHERS => '0');
125             END CASE;
126     END CASE;
127 END PROCESS;
128
129 fsmflipflops : PROCESS (clk, rst, Tstep_D)
130 BEGIN
131     IF rising_edge(clk) THEN
132         IF rst = '1' THEN
133             Tstep_Q <= T0;
134         ELSE
135             Tstep_Q <= Tstep_D;
136         END IF;
137     END IF;
138 END PROCESS;
139
140 stateValue <= Tstep_Q;
141
142 END arch;
```

Figure 5 - Code VHDL FSM #2

Dans le processus « controlsignals », on décide quels registres activés, quel registre doit-on mettre sur le bus, mais aussi quelle sera la valeur du bit « mode ». Cette décision est prise selon l'état et l'instruction actuel. On utilise la représentation « one hot » de l'adresse des registres X et Y afin de faciliter l'assignation du « inControl » et « outControl ».

Finalement, dans le processus « fsmflipflops », on change l'état de la FSM selon l'horloge et le bouton « reset », mais aussi le prochain état qui a été déterminé auparavant.

Partie 2 : Testbench

Processeur

```
15    DUT : processeur PORT MAP(run, clk, rst, din, buswire, done);
16
17    PROCESS
18    BEGIN
19       REPORT "Testbench starting...";
20       run <= '1';
21       rst <= '0';
22       clk <= '0';
23       din <= "000000000";
24       WAIT FOR 10 ns;
25       rst <= '1';
26       clk <= '1';
27       WAIT FOR 10 ns;
28       rst <= '0';
29       clk <= '0';
30       WAIT FOR 10 ns;
31       -- mvi
32       din <= "001001001";
33       WAIT FOR 5 ns;
34       clk <= '1';
35       WAIT FOR 10 ns;
36       clk <= '0';
37       WAIT FOR 10 ns;
38       clk <= '1';
39       WAIT FOR 10 ns;
40       clk <= '0';
41       WAIT FOR 10 ns;
42       -- mv
43       din <= "000000001";
44       WAIT FOR 5 ns;
45       clk <= '1';
46       WAIT FOR 10 ns;
47       clk <= '0';
48       WAIT FOR 10 ns;
49       clk <= '1';
50       WAIT FOR 10 ns;
51       clk <= '0';
52       WAIT FOR 10 ns;
53       -- add
54       din <= "010000001";
55       WAIT FOR 5 ns;
56       clk <= '1';
57       WAIT FOR 10 ns;
58       clk <= '0';
59       WAIT FOR 10 ns;
60       clk <= '1';
61       WAIT FOR 10 ns;
62       clk <= '0';
63       WAIT FOR 10 ns;
64       clk <= '1';
65       WAIT FOR 10 ns;
66       clk <= '0';
67       WAIT FOR 10 ns;
68       clk <= '1';
69       WAIT FOR 10 ns;
70       clk <= '0';
71       WAIT FOR 10 ns;
72       -- sub
73       din <= "011000001";
74       WAIT FOR 5 ns;
75       clk <= '1';
76       WAIT FOR 10 ns;
77       clk <= '0';
78       WAIT FOR 10 ns;
79       clk <= '1';
80       WAIT FOR 10 ns;
81       clk <= '0';
82       WAIT FOR 10 ns;
83       clk <= '1';
84       WAIT FOR 10 ns;
85       clk <= '0';
86       WAIT FOR 10 ns;
87       clk <= '1';
88       WAIT FOR 10 ns;
89       clk <= '0';
90       WAIT FOR 10 ns;
91       WAIT;
92    END PROCESS;
```

Figure 6 - Code VHDL du testbench Processeur

Ce testbench vérifie effectue un déplacement de constante, un déplacement de registre à registre, une addition et une soustraction.

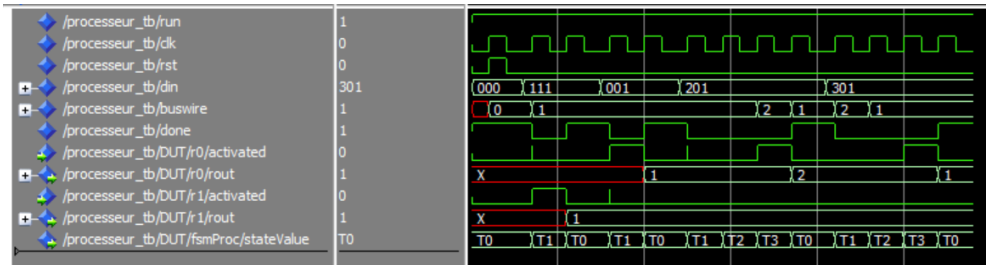

Figure 7 - Résultat du testbench Processeur

Comme on peut le voir dans la figure ci-dessus, on :

1. Réinitialise la FSM
2. MVI une valeur de 1 dans le reg1
3. MV le reg1 dans le reg0
4. Additionne le reg0 avec le reg1
5. Enregistre la valeur dans le reg0
6. Soustrait le reg1 du reg0
7. Enregistre la valeur dans le reg0

AddSub

```
13    DUT : addsub PORT MAP(a, b, mode, c);
14
15    PROCESS
16    BEGIN
17      REPORT "Testbench starting...";
18
19      a <= "000000000";
20      b <= "000000000";
21      mode <= '0';
22      WAIT FOR 10 ns;
23
24      a <= "000000001";
25      b <= "000000001";
26      mode <= '0';
27      WAIT FOR 10 ns;
28
29      a <= "111111111";
30      b <= "000000001";
31      mode <= '0';
32      WAIT FOR 10 ns;
33
34      a <= "000000001";
35      b <= "111111111";
36      mode <= '1';
37      WAIT FOR 10 ns;
38
39      a <= "111111111";
40      b <= "000000001";
41      mode <= '1';
42      WAIT FOR 10 ns;
43
44      a <= "111111111";
45      b <= "111111111";
46      mode <= '1';
47      WAIT FOR 10 ns;
48
49      a <= "000000001";
50      b <= "000000001";
51      mode <= '1';
52      WAIT FOR 10 ns;
53
54      WAIT;
55    END PROCESS;
```

Figure 8 - Code VHDL du testbench AddSub

Ce testbench effectue trois soustractions et quatre additions, tout en testant des valeurs normales et des exceptions comme les « overflows ».

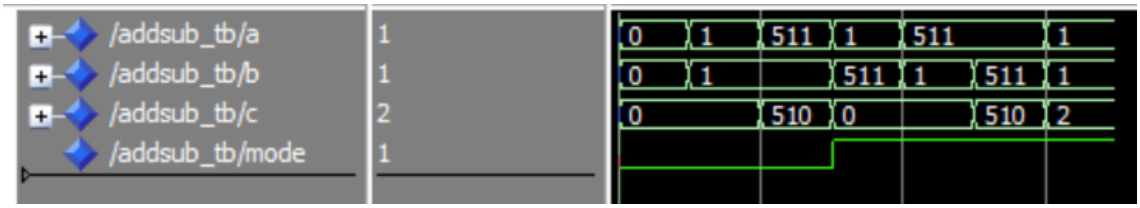

Figure 9 - Résultat du testbench AddSub

Comme on peut le voir dans la figure ci-dessus, on :

1. Initialise les valeurs
2. Soustrait
    a. 0-0=0
    b. 1-1=0
    c. 511-1=510
3. Additionne
    a. 1+511=0
    b. 511+1=0
    c. 511+511=510
    d. 1+1=2

Registre

```
15    DUT : registre PORT MAP(rin, clk, activated, rout);
16    PROCESS BEGIN
17       REPORT "Testbench starting...";
18
19       rin <= 9d"0";
20       clk <= '0';
21       activated <= '0';
22       WAIT FOR 10 ns;
23       clk <= '1';
24       WAIT FOR 10 ns;
25       clk <= '0';
26       WAIT FOR 10 ns;
27       -- test sans activated
28       rin <= 9d"3";
29       clk <= '1';
30       WAIT FOR 10 ns;
31       clk <= '0';
32       WAIT FOR 10 ns;
33       -- test avec activated
34       activated <= '1';
35       clk <= '1';
36       WAIT FOR 10 ns;
37       clk <= '0';
38       WAIT;
39    END PROCESS;
```

Figure 10 - Code VHDL du testbench Registre

Ce testbench essaie d'assigner une valeur sans que le registre soit activé, puis réessaie avec le registre activé.

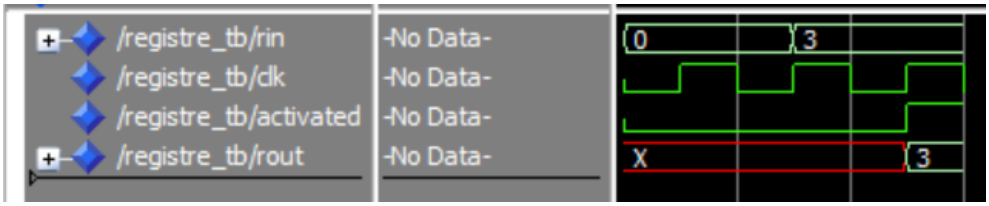

Figure 11 - Résultat du testbench Registre

Comme on peut le voir dans la figure ci-dessus, on :

1. Initialise les valeurs
2. Tente d'assigner la valeur 3 au registre sans l'activer
3. Le « rout » reste à « undefined » (X)
4. Réessaie avec l'activation
5. Le « rout » devient 3

FSM

```
20    process begin
21        report "Testbench starting...";
22        run <= '1';
23        clk <= '0';
24        wait for 10 ns;
25        -- mv
26        i <= "000";
27        xReg <= "00000001";
28        yReg <= "00000001";
29        clk <= '1';
30        wait for 10 ns;
31        clk <= '0';
32        wait for 10 ns;
33        clk <= '1';
34        wait for 10 ns;
35        clk <= '0';
36        wait for 10 ns;
37        -- mvi
38        i <= "001";
39        clk <= '1';
40        wait for 10 ns;
41        clk <= '0';
42        wait for 10 ns;
43        clk <= '1';
44        wait for 10 ns;
45        clk <= '0';
46        wait for 10 ns;
47        -- add
48        i <= "010";
49        clk <= '1';
50        wait for 10 ns;
51        clk <= '0';
52        wait for 10 ns;
53        clk <= '1';
54        wait for 10 ns;
55        clk <= '0';
56        wait for 10 ns;
57        clk <= '1';
58        wait for 10 ns;
59        clk <= '0';
60        wait for 10 ns;
61        clk <= '1';
62        wait for 10 ns;
63        clk <= '0';
64        wait for 10 ns;
65        -- add
66        i <= "011";
67        clk <= '1';
68        wait for 10 ns;
69        clk <= '0';
70        wait for 10 ns;
71        clk <= '1';
72        wait for 10 ns;
73        clk <= '0';
74        wait for 10 ns;
75        clk <= '1';
76        wait for 10 ns;
77        clk <= '0';
78        wait for 10 ns;
79        clk <= '1';
80        wait for 10 ns;
81        clk <= '0';
82        wait for 10 ns;
83        -- test reset
84        i <= "010";
85        clk <= '1';
86        wait for 10 ns;
```

Figure 12 - Code VHDL du testbench FSM

Ce testbench vérifie les étapes et le contrôle des signaux de la FSM pour le MV, MVI, ADD et SUB, puis teste le RESET.

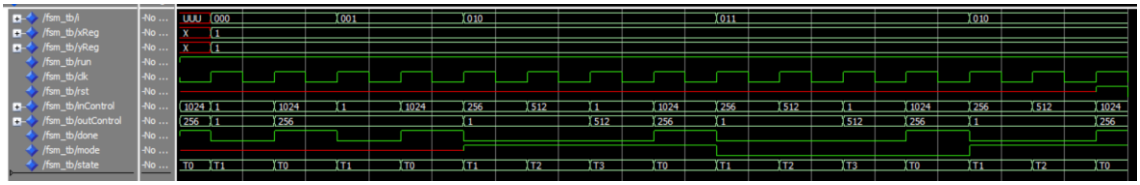

Figure 13 - Résultat du testbench FSM

Afin de réduire la grosseur de l'image, les « one hot » sont affichés en décimales. Comme on peut le voir dans la figure ci-dessus, on :

1. Initialise les valeurs
2. Fait un MV (000)
    a. Passe au T1 et retourne au T0
3. Fait un MVI (001)
    a. Passe au T1 et retourne au T0
4. Fait un ADD (010)
    a. Passe au T1, T2, T3 et retourne au T0
5. Fait un SUB (011)
    a. Passe au T1, T2, T3 et retourne au T0
6. Fait un ADD (010) pour tester le RESET
    a. Passe au T1, T2, puis on active le RESET et retourne au T0
7. À chaque T0, le bit « done » est à 1.
8. DIN est utilisé en outControl et IR en inControl au T0

Partie 3 : Interfacer le circuit

Interface

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.sseg_constants.ALL;
USE work.processeur_constants.ALL;
USE work.ff.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY processeur_interface IS
    PORT (
        SW : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
        KEY : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        CLOCK_50 : IN STD_LOGIC;
        LEDR : OUT STD_LOGIC_VECTOR(9 DOWNTO 0);
        HEX0, HEX1, HEX2, HEX3, HEX4, HEX5 : OUT STD_LOGIC_VECTOR(6 DOWNTO 0)
    );
END processeur_interface;

ARCHITECTURE arch OF processeur_interface IS
    SIGNAL clk, rst : STD_LOGIC;
    SIGNAL buswire : STD_LOGIC_VECTOR(8 DOWNTO 0);
    SIGNAL done : STD_LOGIC;
BEGIN
    d_key0 : ENTITY work.debounce PORT MAP(CLOCK_50, KEY(0), clk);
    d_key1 : ENTITY work.debounce PORT MAP(CLOCK_50, KEY(1), rst);

    processeur0 : processeur PORT MAP(SW(0), clk, rst, SW(9 DOWNTO 1), buswire, done);

    LEDR <= buswire & done;

    sseg0 : sseg PORT MAP(buswire(3 DOWNTO 0), HEX0);
    sseg1 : sseg PORT MAP(buswire(7 DOWNTO 4), HEX1);
    sseg2 : sseg PORT MAP("000" & buswire(8), HEX2);
    sseg3 : sseg PORT MAP("0" & SW(3 DOWNTO 1), HEX3);
    sseg4 : sseg PORT MAP("0" & SW(6 DOWNTO 4), HEX4);
    sseg5 : sseg PORT MAP("0" & SW(9 DOWNTO 7), HEX5);
END arch;
```

Figure 14 - Code VHDL Interface

Afin d'interfacer le circuit, nous avons créé une entité « processeur_interface ». Cette entité utilise en entrée et en sortie les différents contrôles du FPGA, notamment les interrupteurs, les boutons, l'horloge, les témoins lumineux et les afficheurs 7-segments. Les neuf premiers interrupteurs représentent le DIN (IIIXXXYYY) et le dernier est le bit « run ». Le bouton 0 sert à l'horloge et le bouton 1 sert au RESET. Les boutons sont « debouncer » afin de s'assurer qu'il n'y a qu'une activation par enfoncement. Les témoins lumineux affichent les valeurs du bus de données, ainsi que l'état « done » du processeur. Finalement, les trois premiers 7-segments affichent les valeurs des interrupteurs en octale, puis les trois derniers affichent la valeur du bus sous forme hexadécimale.

Partie 4 : Utilisation de la mémoire

Interface mémoire

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE work.sseg_constants.ALL;
4  USE work.processeur_constants.ALL;
5  USE work.ff.ALL;
6  USE IEEE.STD_LOGIC_UNSIGNED.ALL;
7  USE work.inst_mem;
8  USE work.registre_constants.ALL;
9
10 ENTITY processeur_mem_interface IS
11    PORT (
12       SW : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
13       KEY : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
14       CLOCK_50 : IN STD_LOGIC;
15       LEDR : OUT STD_LOGIC_VECTOR(9 DOWNTO 0);
16       HEX0, HEX1, HEX2, HEX3, HEX4, HEX5 : OUT STD_LOGIC_VECTOR(6 DOWNTO 0)
17    );
18 END processeur_mem_interface;
19
20 ARCHITECTURE arch OF processeur_mem_interface IS
21    SIGNAL run, clk, rst : STD_LOGIC;
22    SIGNAL buswire, din : STD_LOGIC_VECTOR(8 DOWNTO 0);
23    SIGNAL done : STD_LOGIC;
24    SIGNAL address, nextAddress : STD_LOGIC_VECTOR(4 DOWNTO 0) := "00000";
25 BEGIN
26    run <= SW(0);
27    d_key0 : ENTITY work.debounce PORT MAP(CLOCK_50, KEY(0), clk);
28    d_key1 : ENTITY work.debounce PORT MAP(CLOCK_50, KEY(1), rst);
29
30    processeur0 : processeur PORT MAP(run, clk, rst, din, buswire, done);
31
32    LEDR <= address & "0000" & done;
33
34    sseg0 : sseg PORT MAP(buswire(3 DOWNTO 0), HEX0);
35    sseg1 : sseg PORT MAP(buswire(7 DOWNTO 4), HEX1);
36    sseg2 : sseg PORT MAP("000" & buswire(8), HEX2);
37    sseg3 : sseg PORT MAP("0" & din(2 DOWNTO 0), HEX3);
38    sseg4 : sseg PORT MAP("0" & din(5 DOWNTO 3), HEX4);
39    sseg5 : sseg PORT MAP("0" & din(8 DOWNTO 6), HEX5);
40
41    nextAddress <= address + "00001";
42    counter : registre GENERIC MAP(5) PORT MAP(nextAddress, clk, done and run, address);
43    mem : inst_mem PORT MAP(address, clk, din);
44 END arch;
```

Figure 15 - Code VHDL Interface Mémoire

Pour interfacer la mémoire, on remplace les interrupteurs 9 à 1 par un signal de 9 bits « din ».Afin de savoir l'adresse actuelle, on utilise un registre qui prend en entrée l'adresse suivante (adresse actuelle plus un), l'horloge, le bit « done » et « run » comme bit d'activation, et qui sort l'adresse actuelle. L'adresse actuelle change donc pour l'adresse suivante chaque fois que le processeur retourne à l'état T0. On donne finalement en entrée à l'instance mémoire cette adresse, l'horloge et on sort la valeur à l'adresse dans le vecteur « din ». On affiche aussi l'adresse actuel sur les témoins lumineux 9 à 5. Les trois premiers 7-segments, eux, affichent la valeur sortant de la mémoire.

Mémoire

```
DEPTH = 32;
WIDTH = 9;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN

00 : 001001001; -- MVI REG1 <= 1
01 : 001000010; -- MVI REG0 <= 2
02 : 010000001; -- ADD REG0 <= REG0 + REG1 (2 + 1 = 3)
03 : 000010000; -- MV  REG2 <= REG0 (3)
04 : 011010001; -- SUB REG2 <= REG2 - REG1 (3 - 1 = 2)
05 : 000011010; -- MV  REG3 <= REG2 (2)
06 : 000000000;
07 : 000000000;
08 : 000000000;
09 : 000000000;
0A : 000000000;
0B : 000000000;
0C : 000000000;
0D : 000000000;
0E : 000000000;
0F : 000000000;
10 : 000000000;
11 : 000000000;
12 : 000000000;
13 : 000000000;
14 : 000000000;
15 : 000000000;
16 : 000000000;
17 : 000000000;
18 : 000000000;
19 : 000000000;
1A : 000000000;
1B : 000000000;
1C : 000000000;
1D : 000000000;
1E : 000000000;
1F : 000000000;

END;
```

Figure 16 - Mémoire du fichier « inst_mem.mif »

Dans la mémoire, on utilise seulement les 6 premières adresses, puisqu'elles sont suffisantes pour tester toutes les fonctionnalités du processeur. Voici les instructions en ordre :

1. MVI : On déplace la valeur 1 dans le registre 1
2. MVI : On déplace la valeur 2 dans le registre 0
3. ADD : On additionne le registre 0 avec le registre 1, puis on enregistre la valeur dans le registre 0 (2+1=3)
4. MV : On déplace le registre 0 dans le registre 2 (3)
5. SUB : On soustrait le registre 1 du registre 2, puis on enregistre la valeur dans le registre 2 (3-1=2)
6. MV : On déplace le registre 2 dans le registre 3 (2)

Conclusion

En conclusion, ce laboratoire nous a permis de concevoir un processeur implémentant des opérations de bases, mais aussi de mieux comprendre son fonctionnement et l'utilisation des machines à états. Il a aussi permis de bien mettre en application les connaissances accumulés jusqu'à présent au travers des précédents laboratoires, et de voir comment ses connaissances pourraient être utilisées dans des problèmes plus réels.

Références

Aucune référence.