UQAC
Université du Québec
à Chicoutimi

Projet de conception
Calculatrice / Processeur
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 : 28 04 2023

Table des matières

Liste des figures

Introduction

Dans ce projet de conception, nous devrons modifier le processeur simple conçu lors du laboratoire 4 afin d'ajouter quatre instructions supplémentaires, mais aussi pour supporter un affichage VGA des instructions disponibles et des valeurs des registres pendant son utilisation. De plus, nous devrons modifier l'interfaçage du processeur afin d'utiliser comme entrée un clavier qui communiquera avec le processeur par protocole UART. Ce clavier remplacera les interrupteurs qui permettaient d'indiquer au processeur l'instruction à faire et les registres X et Y qui seront affectés par l'instruction (format IIIXXXYYY).

Les instructions déjà présentent sont les suivantes :

- MV (000)
- MVI (001)
- ADD (010)
- SUB (011)

Les instructions qui seront ajoutées au processeur sont les suivantes :

- MUL (100)
- DIV (101)
- POW (110)
- CLR (111)

L'instruction MUL permettra de multiplier le registre X avec le registre Y, puis d'enregistrer la valeur résultante dans le registre X. L'instruction DIV permettra de diviser le registre X par le registre Y, puis d'enregistrer la valeur résultante dans le registre X. L'instruction POW permettra de calculer le carré du registre X (donc sa valeur à la puissance 2), puis d'enregistrer la valeur résultante dans le registre X. L'instruction CLR permettra de réinitialiser tous les registres en leur assignant la valeur 0 (leur valeur initiale).

Architecture du processeur

L'architecture du processeur peut être décrite par deux schémas :

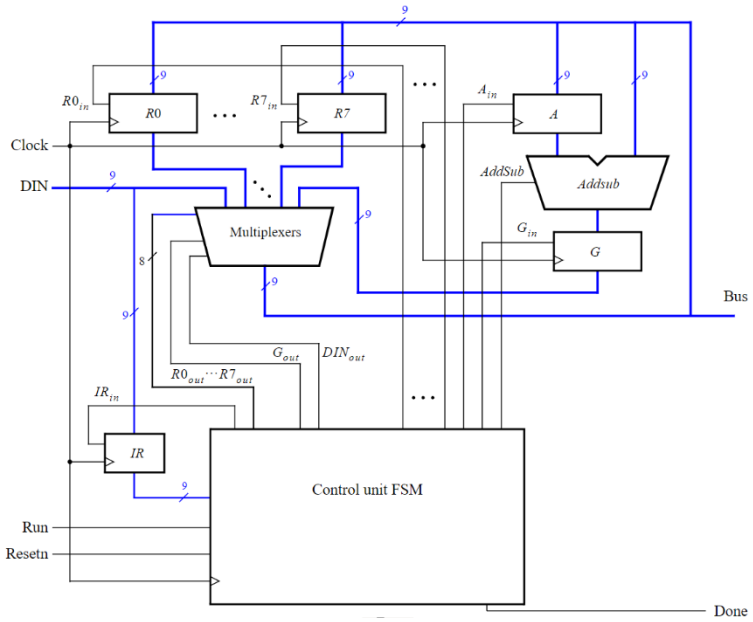


Figure 1 - Architecture du processeur

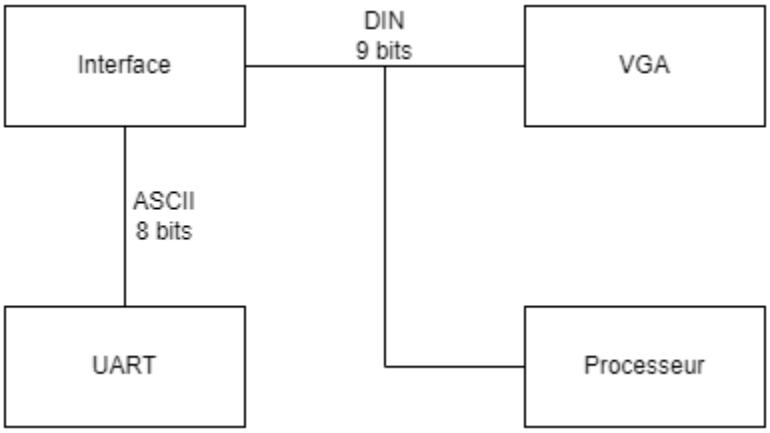


Figure 2 - Architecture de l'interface

La figure 1 représente l'architecture du processeur et les connexions de ses composants. La figure 2, elle, représente la connexion simplifiée de l'interface du processeur, c'est-à-dire ce qui permet la réception de caractères ASCII par clavier, l'affichage VGA, ainsi que la transmission d'instruction au processeur.

Modification du processeur simple

Processeur

```
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;

        CLOCK_50 : IN STD_LOGIC;
        KEY : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        VGA_R, VGA_G, VGA_B : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
        VGA_HS : OUT STD_LOGIC;
        VGA_VS : OUT STD_LOGIC
    );
END processeur;

ARCHITECTURE arch OF processeur IS
    signal r0in, r1in, r2in, r3in, r4in, r5in, r6in, r7in, ain, gin, irin : std_logic := '0';
    signal inControl : std_logic_vector(10 downto 0);
    signal outControl : std_logic_vector(10 downto 0);
    signal r0out, r1out, r2out, r3out, r4out, r5out, r6out, r7out, aout, gout, irout, aluout : std_logic_vector(8 downto 0);
    signal sb : std_logic_vector(3 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);
    alu0 : alu port map(aout, buswire, i, aluout);
    g : registre port map(aluout, 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);

    enc : Enc164 port map("00000" & outControl, sb);
    mux : Muxb10 port map(gout, "000000" & irout(2 downto 0), r7out, r6out, r5out, r4out, r3out, r2out, r1out, r0out, sb, buswire);

    affichage : text_screen port map(CLOCK_50, KEY, VGA_R, VGA_G, VGA_B, VGA_HS, VGA_VS, clk, r0out, r1out, r2out, r3out, r4out, r5out, r6out, r7out, irout);
END arch;
```

Figure 3 - Code VHDL Processeur

Afin de permettre l'ajout des nouvelles instructions mathématiques, il faut tout d'abord modifier l'entité qui permet de calculer ces instructions. Nous avons donc modifié le nom de l'entité « addsub » pour « alu » (arithmetic-logic unit). Cette entité prend en entrée un vecteur de 3 bits représentant l'instruction qu'elle doit gérer. Pour remplacer le bit « mode » qui était auparavant entrant à l'ancienne entité, nous connectons simplement le signal de 3 bits « i » (l'instruction du registre IR) à l'entrée de l'ALU. Pour permettre le fonctionnement de l'instruction CLR, nous avons dû changer la taille du vecteur « outControl » de 10 bits à 11 bits.

Le reste de l'architecture du processeur (connexion des registres au bus et à l'ALU) ne change pas puisqu'il permet déjà de faire les nouvelles instructions dans leur structure actuelle.

ALU

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4   package alu_constants is
5      component alu is
6         GENERIC  (n : INTEGER := 9);
7         PORT (
8            a, b : IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
9            mode : in std_logic_vector(2 downto 0);
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  use ieee.numeric_std.all;
20
21  ENTITY alu IS
22     GENERIC  (n : INTEGER := 9);
23     PORT (
24        a, b : IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
25        mode : in std_logic_vector(2 downto 0);
26        c : OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0)
27     );
28  END alu;
29
30
31  ARCHITECTURE arch OF alu IS
32  BEGIN
33     calculate : process(a, b, mode)
34     begin
35        case mode is
36           when "010" =>  c <= std_logic_vector(unsigned(a) + unsigned(b));
37           when "011" =>  c <= std_logic_vector(unsigned(a) - unsigned(b));
38           when "100" =>  c <= std_logic_vector(resize(unsigned(a) * unsigned(b), a'length));
39           when "101" =>  c <= std_logic_vector(unsigned(a) / unsigned(b));
40           when "110" =>  c <= std_logic_vector(to_unsigned(to_integer(unsigned(a)) ** 2, n));
41           when others =>    c <= std_logic_vector(unsigned(a) + unsigned(b));
42        end case;
43     end process calculate;
44  END arch;
```

Figure 4 - Code VHDL ALU

L'entité « alu » prend en entrée un vecteur a et b, ainsi que le mode (l'instruction actuelle). Afin de simplifier l'entièreté des opérations, on convertit les vecteurs en « unsigned » ou en « integer » au besoin afin d'avoir accès aux opérateurs mathématiques. L'addition, la soustraction et la division se font simplement avec l'opérateur. La multiplication, elle, retourne un vecteur ayant une taille correspondante à la somme de la taille du vecteur a et b. Afin de pouvoir retourner un vecteur de même taille que les entrées, on utilise la fonction « resize », qui tronque le vecteur à la taille souhaitée (la longueur du vecteur a). Le calcul de la puissance nécessite une constante, c'est pourquoi nous avons décidé de calculer la puissance de 2. Avant de pouvoir calculer cette puissance, on doit d'abord transformer le vecteur a en « integer ». Finalement, si l'instruction n'est pas une instruction en lien avec l'ALU, on retourne l'addition du vecteur a et b par défaut. L'entité sort ce résultat dans le signal c de taille « n ».

FSM

```
29  ENTITY fsm IS
30      PORT (
31          i : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
32          xReg, yReg : in std_logic_vector(7 downto 0);
33          run, clk, rst : in std_logic;
34          inControl : out std_logic_vector(10 downto 0);
35          outControl : out std_logic_vector(10 downto 0);
36          done : 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 Tstep_D <= T0;
49                  ELSE Tstep_D <= T1;
50                  END IF;
51
52              WHEN T1 =>
53                  IF run = '0' THEN Tstep_D <= T1;
54                  ELSIF i = "000" OR i = "001" OR i = "111" THEN Tstep_D <= T0;
55                  ELSIF i = "010" OR i = "011" OR i = "100" OR i = "101" OR i = "110" THEN Tstep_D <= T2;
56                  END IF;
57
58              WHEN T2 =>
59                  IF run = '0' THEN Tstep_D <= T2;
60                  ELSIF i = "010" OR i = "011" OR i = "100" OR i = "101" OR i = "110" THEN Tstep_D <= T3;
61                  END IF;
62
63              WHEN T3 =>
64                  IF run = '0' THEN Tstep_D <= T3;
65                  ELSE Tstep_D <= T0;
66                  END IF;
67          END CASE;
68      END PROCESS;
69
```

Figure 5 - Code VHDL FSM #1

Les premières modifications de la FSM se trouve dans le port de l'entité. Nous avons retiré le bit sortant « mode » puisqu'il n'était plus utilisé dans le processeur. Nous avons aussi modifié la taille du vecteur « outControl » de 10 bits à 11 bits pour permettre l'instruction CLR.

Dans le processus « statetable », nous avons ajoutés les instructions MUL (100), DIV (101), POW (110) aux mêmes endroits que l'addition et la soustraction puisque ce sont des instructions mathématiques qui utilisent eux aussi trois coups d'horloge afin d'être complétées. L'instruction CLR (111) a été ajouté où l'instruction MV et MVI se trouvent puisqu'elle n'a besoin qu'un coup d'horloge.

```
70    controlsignals: PROCESS (Tstep_Q, i, xReg, yReg)
71    BEGIN
72    CASE Tstep_Q IS
73       WHEN T0 =>
74          done <= '1';
75          inControl <= "10000000000";
76          outControl <= "00100000000";
77       WHEN T1 =>
78          done <= '0';
79
80          CASE i IS
81             when "000" =>
82                inControl <= "000" & xReg;
83                outControl <= "000" & yReg;
84             when "001" =>
85                inControl <= "000" & xReg;
86                outControl <= "00100000000";
87             when "010" | "011" | "100" | "101" | "110" =>
88                inControl <= "00100000000";
89                outControl <= "000" & xReg;
90             when "111" =>
91                inControl <= "00011111111";
92                outControl <= "10000000000";
93             when others =>
94                inControl <= (others => '0');
95                outControl <= (others => '0');
96          END CASE;
97       WHEN T2 =>
98          done <= '0';
99          CASE i IS
100            when "010" | "011" | "100" | "101" | "110" =>
101               inControl <= "01000000000";
102               outControl <= "000" & yReg;
103            when others =>
104               inControl <= (others => '0');
105               outControl <= (others => '0');
106         END CASE;
107      WHEN T3 =>
108         done <= '0';
109         CASE i IS
110            when "010" | "011" | "100" | "101" | "110" =>
111               inControl <= "000" & xReg;
112               outControl <= "01000000000";
113            when others =>
114               inControl <= (others => '0');
115               outControl <= (others => '0');
116         END CASE;
117   END CASE;
118 END PROCESS;
```

Figure 6 - Code VHDL FSM #2

Dans le processus « controlsignals », on ajoute encore les instructions MUL, DIV et POW aux mêmes endroits que l'addition et soustraction puisqu'ils utilisent les mêmes systèmes de signaux d'entrée et de sortie. L'instruction CLR cependant utilise un « outControl » différent que les autres instructions. Nous avons modifié la taille de ce vecteur afin d'avoir 1 bit de plus dans la représentation « one hot », ce qui permet de tomber dans le cas par défaut du Mux du processeur. Ce cas par défaut met sur le bus le vecteur de 9 bits « 000000000 ». On change ensuite le vecteur « inControl » pour activer l'ensemble des registres.

À la suite du coup d'horloge, l'ensemble des registres activés reçoivent la valeur de 0 en entrée, et donc sont réinitialisés à leur valeur initiale.

```
120   fsmflipflops: PROCESS (clk, rst, Tstep_D)
121   BEGIN
122      IF rising_edge(clk) THEN
123         IF rst = '1' THEN
124            Tstep_Q <= T0;
125         ELSE
126            Tstep_Q <= Tstep_D;
127         END IF;
128      END IF;
129   END PROCESS;
130
131   stateValue <= Tstep_Q;
132
133 end arch;
```

Le processus « fsmflipflops » ne change pas puisqu'il n'est pas touché par l'ajout des nouvelles instructions n'impacte pas

Registre

Aucun changement n'a été fait à l'entité « registre ».

Machine à états et tableau d'états

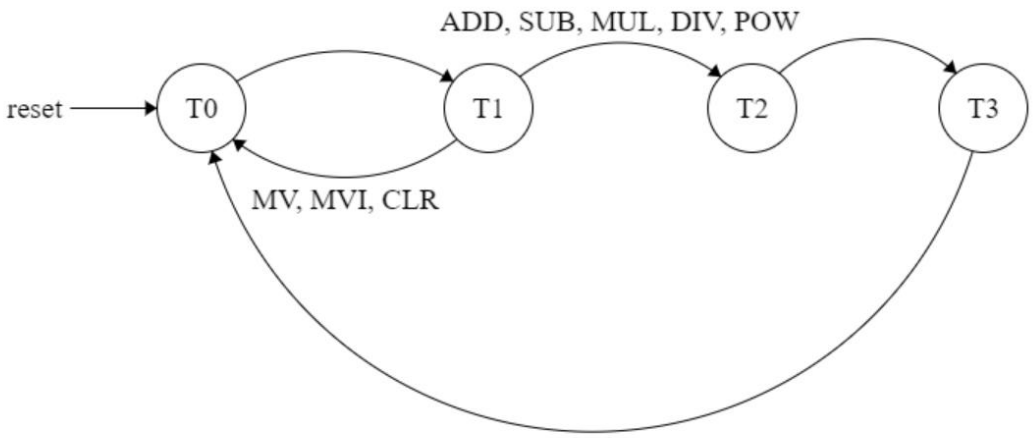


Figure 7 – Machine d'états du projet

L'état T0 est l'état initial du processeur. Après un coup d'horloge, on transitionne à l'état T1 et la FSM réagit en fonction de l'instruction de l'utilisateur. Si l'instruction est MV, MVI ou CLR, on retourne à l'état initial puisqu'elles peuvent être traitées en un coup d'horloge. Pour les instructions ADD, SUB, MUL, DIV et POW, il faut attendre trois coups d'horloge avant de retourner à l'état initial. Ces instructions nécessitent trois coups d'horloge puisque le premier permet de déplacer le registre X dans l'ALU, le deuxième déplace le registre Y sur le bus de données qui est connecté à l'ALU, puis le troisième enregistre la valeur résultante dans le registre X. Voici le tableau de transitions et d'instruction de la machine à états finis.

Tableau 1 – Transistions des différents états

Current State			**Next state**
State	**Reset**	**Instruction**	**-**
T0	0	X	T1
T0	1	X	T0
T1	0	MV | MVI | CLR	T0
T1	0	ADD | SUB | MUL | DIV | POW	T2
T1	1	X	T0
T2	0	X	T3
T2	1	X	T0
T3	X	X	T0

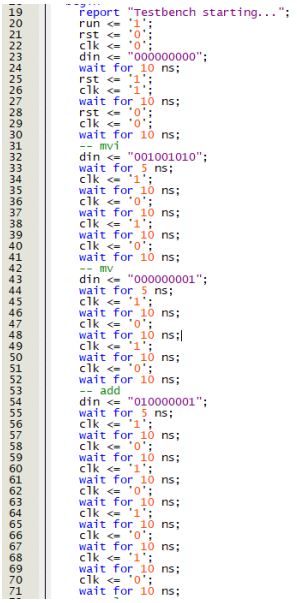
Tableau 2 – Instructions du processeur

Instructions		
Instruction	**Vecteur**	**Valeur**
MV	000	0
MVI	001	1
ADD	010	2
SUB	011	3
MUL	100	4
DIV	101	5
POW	110	6
CLR	111	7

Dans le tableau ci-dessus, il est possible de voir la combinaison de bits pour chaque instruction qui peuvent être utilisé avec la calculatrice. Afin de faciliter les entrées au clavier lors de l'utilisation de la calculatrice, nous avons assigné une valeur de 0 à 7 à la place des trois bits précédemment utilisés.

Testbench des modifications

Processeur



```
    report "Testbench starting...";
    run <= '1';
    rst <= '0';
    clk <= '0';
    din <= "000000000";
    wait for 10 ns;
    rst <= '1';
    clk <= '1';
    wait for 10 ns;
    rst <= '0';
    clk <= '0';
    wait for 10 ns;
    -- mvi
    din <= "001001010";
    wait for 5 ns;
    clk <= '1';
    wait for 10 ns;
    clk <= '0';
    wait for 10 ns;
    clk <= '1';
    wait for 10 ns;
    clk <= '0';
    wait for 10 ns;
    -- mv
    din <= "000000001";
    wait for 5 ns;
    clk <= '1';
    wait for 10 ns;
    clk <= '0';
    wait for 10 ns;
    clk <= '1';
    wait for 10 ns;
    clk <= '0';
    wait for 10 ns;
    -- add
    din <= "010000001";
    wait for 5 ns;
    clk <= '1';
    wait for 10 ns;
    clk <= '0';
    wait for 10 ns;
    clk <= '1';
    wait for 10 ns;
    clk <= '0';
    wait for 10 ns;
    clk <= '1';
    wait for 10 ns;
    clk <= '0';
    wait for 10 ns;
    clk <= '1';
    wait for 10 ns;
    clk <= '0';
    wait for 10 ns;
```

Figure 8 - Code VHDL du testbench Processeur #1

```
72      -- mul
73      din <= "100000001";
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      -- div
92      din <= "101000001";
93      wait for 5 ns;
94      clk <= '1';
95      wait for 10 ns;
96      clk <= '0';
97      wait for 10 ns;
98      clk <= '1';
99      wait for 10 ns;
100     clk <= '0';
101     wait for 10 ns;
102     clk <= '1';
103     wait for 10 ns;
104     clk <= '0';
105     wait for 10 ns;
106     clk <= '1';
107     wait for 10 ns;
108     clk <= '0';
109     wait for 10 ns;
110     wait;
111     -- sub
112     din <= "011000001";
113     wait for 5 ns;
114     clk <= '1';
115     wait for 10 ns;
116     clk <= '0';
117     wait for 10 ns;
118     clk <= '1';
119     wait for 10 ns;
120     clk <= '0';
121     wait for 10 ns;
122     clk <= '1';
123     wait for 10 ns;
124     clk <= '0';
125     wait for 10 ns;
```

Figure 9 - Code VHDL du testbench Processeur #2

```
131     -- clr
132     din <= "111000000";
133     wait for 5 ns;
134     clk <= '1';
135     wait for 10 ns;
136     clk <= '0';
137     wait for 10 ns;
138     clk <= '1';
139     wait for 10 ns;
140     clk <= '0';
141     wait for 10 ns;
142     wait;
143   end process;
144 end architecture test;
```

Figure 10 - Code VHDL du testbench Processeur #3

Ce testbench vérifie effectue un déplacement de constante, un déplacement de registre à registre, une addition, une multiplication, une division, une puissance de 2, une soustraction et une réinitialisation des registres.

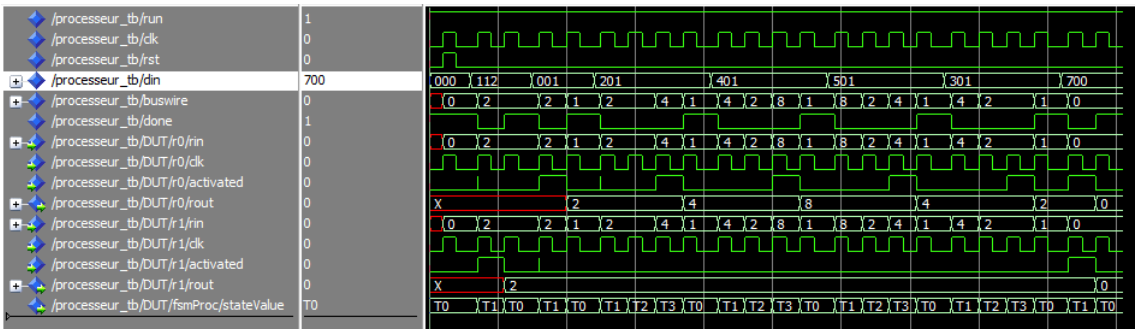
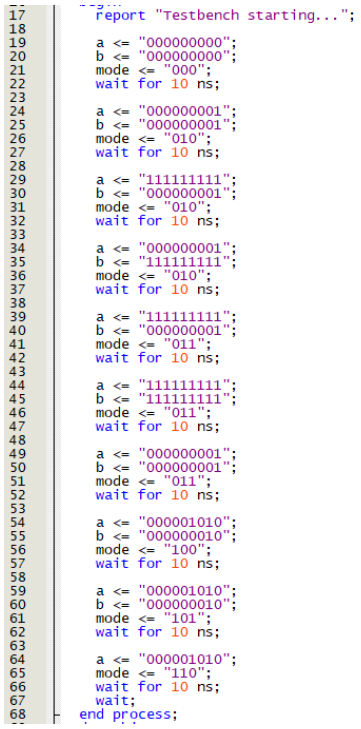


Figure 11 - Résultat du testbench Processeur

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

1. Réinitialise la FSM
2. MVI une valeur de 2 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. Multiplie le reg0 avec le reg1
7. Enregistre la valeur dans le reg0
8. Divise le reg0 par le reg1
9. Enregistre la valeur dans le reg0
10. Soustrait le reg1 du reg0
11. Enregistre la valeur dans le reg0
12. Réinitialise les registres

ALU



```
17    report "Testbench starting...";
18
19    a <= "000000000";
20    b <= "000000000";
21    mode <= "000";
22    wait for 10 ns;
23
24    a <= "000000001";
25    b <= "000000001";
26    mode <= "010";
27    wait for 10 ns;
28
29    a <= "111111111";
30    b <= "000000001";
31    mode <= "010";
32    wait for 10 ns;
33
34    a <= "000000001";
35    b <= "111111111";
36    mode <= "010";
37    wait for 10 ns;
38
39    a <= "111111111";
40    b <= "000000001";
41    mode <= "011";
42    wait for 10 ns;
43
44    a <= "111111111";
45    b <= "111111111";
46    mode <= "011";
47    wait for 10 ns;
48
49    a <= "000000001";
50    b <= "000000001";
51    mode <= "011";
52    wait for 10 ns;
53
54    a <= "000001010";
55    b <= "000000010";
56    mode <= "100";
57    wait for 10 ns;
58
59    a <= "000001010";
60    b <= "000000010";
61    mode <= "101";
62    wait for 10 ns;
63
64    a <= "000001010";
65    mode <= "110";
66    wait for 10 ns;
67    wait;
68  end process;
```

Figure 12 - Code VHDL du testbench ALU

Ce testbench effectue trois additions, trois soustractions, une multiplication, une division et une puissance de 2.

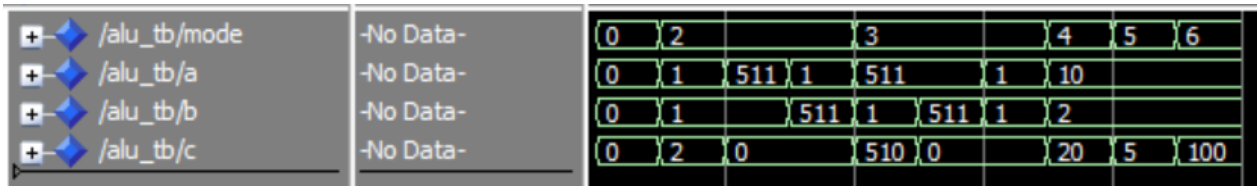


Figure 13 - Résultat du testbench ALU

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

1. Initialise les valeurs
2. Additionne
    a. 1+1=2
    b. 511+1=0
    c. 1+511=0
3. Soustrait
    a. 511-1=510
    b. 511-511=0
    c. 1-1=0
4. Multiplie
    a. 10*2=20
5. Divise
    a. 10/2=5
6. Puissance de 2
    a. 10^2=100

FSM

```
22
23          run <= '1';
24          clk <= '0';
25          wait for 10 ns;
26          -- mv
27          i <= "000";
28          xReg <= "00000001";
29          yReg <= "00000001";
30          clk <= '1';
31          wait for 10 ns;
32          clk <= '0';
33          wait for 10 ns;
34          clk <= '1';
35          wait for 10 ns;
36          clk <= '0';
37          wait for 10 ns;
38          -- mvi
39          i <= "001";
40          clk <= '1';
41          wait for 10 ns;
42          clk <= '0';
43          wait for 10 ns;
44          clk <= '1';
45          wait for 10 ns;
46          clk <= '0';
47          wait for 10 ns;
48          -- add
49          i <= "010";
50          clk <= '1';
51          wait for 10 ns;
52          clk <= '0';
53          wait for 10 ns;
54          clk <= '1';
55          wait for 10 ns;
56          clk <= '0';
57          wait for 10 ns;
58          clk <= '1';
59          wait for 10 ns;
60          clk <= '0';
61          wait for 10 ns;
62          clk <= '1';
63          wait for 10 ns;
64          clk <= '0';
65          wait for 10 ns;
66          sub
```

Figure 14 - Code VHDL du testbench FSM #1

```
66          -- sub
67          i <= "011";
68          clk <= '1';
69          wait for 10 ns;
70          clk <= '0';
71          wait for 10 ns;
72          clk <= '1';
73          wait for 10 ns;
74          clk <= '0';
75          wait for 10 ns;
76          clk <= '1';
77          wait for 10 ns;
78          clk <= '0';
79          wait for 10 ns;
80          clk <= '1';
81          wait for 10 ns;
82          clk <= '0';
83          wait for 10 ns;
84          -- mul
85          i <= "100";
86          clk <= '1';
87          wait for 10 ns;
88          clk <= '0';
89          wait for 10 ns;
90          clk <= '1';
91          wait for 10 ns;
92          clk <= '0';
93          wait for 10 ns;
94          clk <= '1';
95          wait for 10 ns;
96          clk <= '0';
97          wait for 10 ns;
98          clk <= '1';
99          wait for 10 ns;
100         clk <= '0';
101         wait for 10 ns;
102         -- div
103         i <= "101";
104         clk <= '1';
105         wait for 10 ns;
106         clk <= '0';
107         wait for 10 ns;
108         clk <= '1';
109         wait for 10 ns;
110         clk <= '0';
111         wait for 10 ns;
112         clk <= '1';
113         wait for 10 ns;
114         clk <= '0';
115         wait for 10 ns;
116         clk <= '1';
117         wait for 10 ns;
118         clk <= '0';
119         wait for 10 ns;
```

Figure 15 - Code VHDL du testbench FSM #2

```
120         -- pow
121         i <= "110";
122         clk <= '1';
123         wait for 10 ns;
124         clk <= '0';
125         wait for 10 ns;
126         clk <= '1';
127         wait for 10 ns;
128         clk <= '0';
129         wait for 10 ns;
130         clk <= '1';
131         wait for 10 ns;
132         clk <= '0';
133         wait for 10 ns;
134         clk <= '1';
135         wait for 10 ns;
136         clk <= '0';
137         wait for 10 ns;
138         -- clr
139         i <= "111";
140         clk <= '1';
141         wait for 10 ns;
142         clk <= '0';
143         wait for 10 ns;
144         clk <= '1';
145         wait for 10 ns;
146         clk <= '0';
147         wait for 10 ns;
148         -- test reset
149         i <= "010";
150         clk <= '1';
151         wait for 10 ns;
152         clk <= '0';
153         wait for 10 ns;
154         clk <= '1';
155         wait for 10 ns;
156         clk <= '0';
157         wait for 10 ns;
158         rst <= '1';
159         clk <= '1';
160         wait for 10 ns;
161         rst <= '0';
162         clk <= '0';
163         wait;
164     end process;
165 end architecture test;
```

Figure 16 - Code VHDL du testbench FSM #3

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

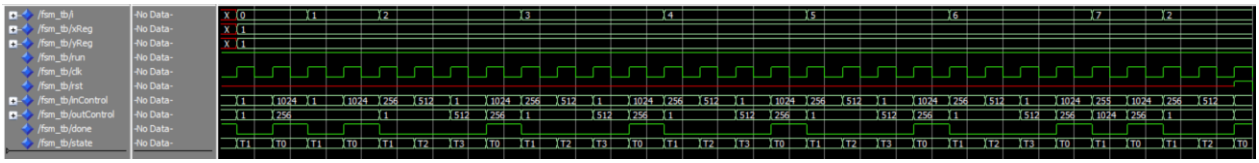


Figure 17 - Résultat du testbench FSM

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 MUL(100)
    a. Passe au T1, T2, T3 et retourne au T0
7. Fait un DIV (101)
    a. Passe au T1, T2, T3 et retourne au T0
8. Fait un POW (110)
    a. Passe au T1, T2, T3 et retourne au T0
9. Fait un CLR (111)
    a. Passe au T1, T2, T3 et retourne au T0
10. Fait un ADD (010) pour tester le RESET
    a. Passe au T1, T2, puis on active le RESET et retourne au T0
11. À chaque T0, le bit « done » est à 1.
12. DIN est utilisé en outControl et IR en inControl au T0

Affichage VGA

Résultat

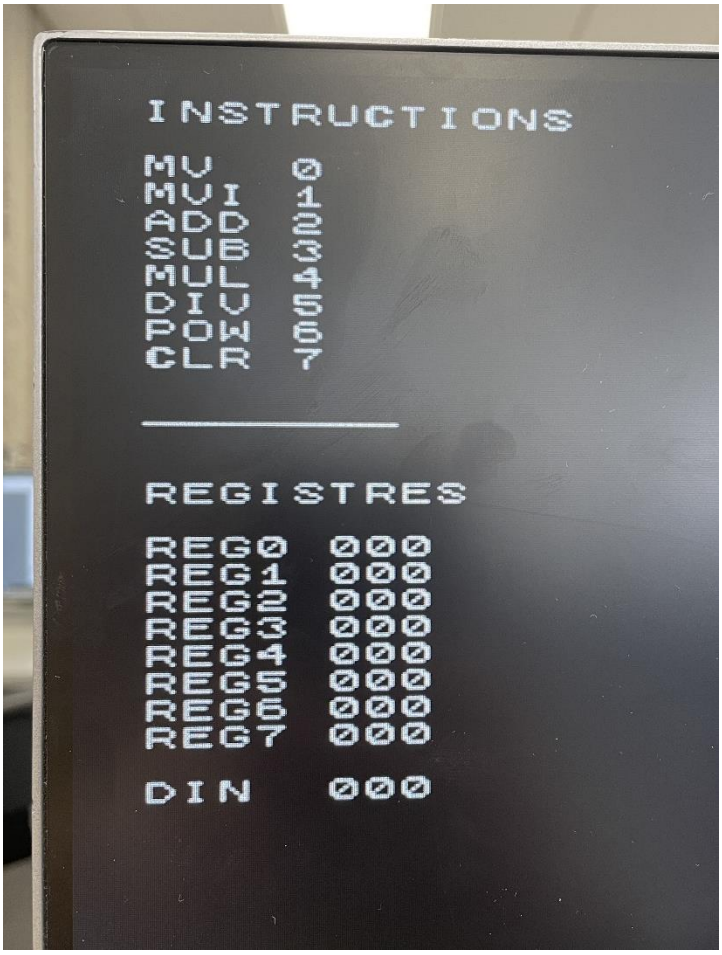
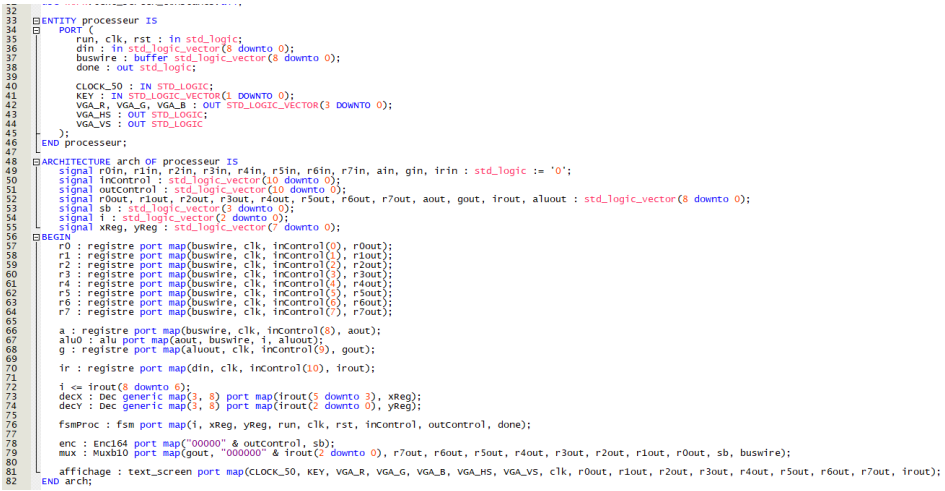


Figure 18 - Affichage VGA

Ceci est le résultat de l'affichage VGA. Comme on peut le voir, les instructions et leur valeurs correspondantes sont écrits en haut. En dessous, on voit les valeurs actuelles des registres 0 à 7 en base 10. On voit aussi la valeur du DIN en octal.

Processeur

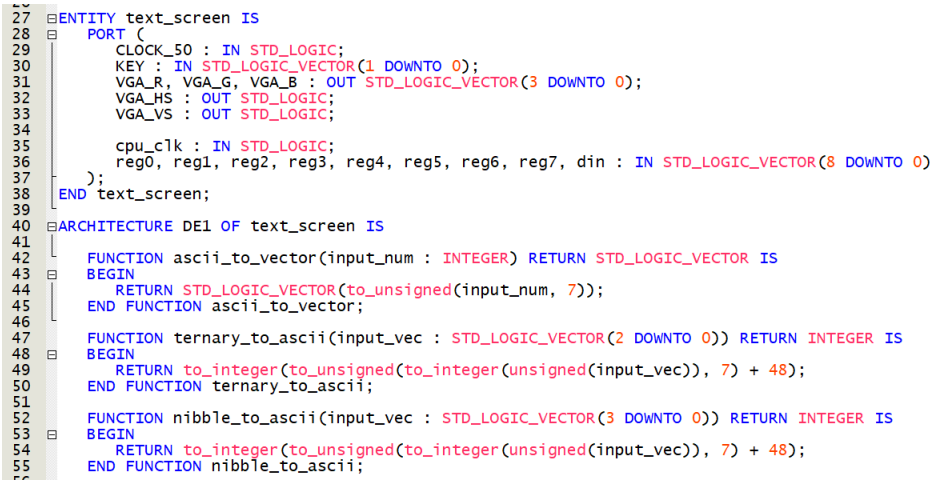


```
ENTITY processeur IS
    PORT (
        run, clk, rst : in std_logic;
        din : in std_logic_vector(8 downto 0);
        buswire : buffer std_logic_vector(8 downto 0);
        done : out std_logic;

        CLOCK_50 : IN STD_LOGIC;
        KEY : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        VGA_R, VGA_G, VGA_B : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
        VGA_HS : OUT STD_LOGIC;
        VGA_VS : OUT STD_LOGIC
    );
END processeur;

ARCHITECTURE arch OF processeur IS
    signal r0in, r1in, r2in, r3in, r4in, r5in, r6in, r7in, ain, gin, irin : std_logic := '0';
    signal inControl : std_logic_vector(10 downto 0);
    signal outControl : std_logic_vector(10 downto 0);
    signal r0out, r1out, r2out, r3out, r4out, r5out, r6out, r7out, aout, gout, irout, aluout : std_logic_vector(8 downto 0);
    signal sb : std_logic_vector(3 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);
    alu0 : alu port map(aout, buswire, i, aluout);
    g : registre port map(aluout, 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);

    enc : Enc164 port map("00000" & outControl, sb);
    mux : Muxb10 port map(gout, "000000" & irout(2 downto 0), r7out, r6out, r5out, r4out, r3out, r2out, r1out, r0out, sb, buswire);

    affichage : text_screen port map(CLOCK_50, KEY, VGA_R, VGA_G, VGA_B, VGA_HS, VGA_VS, clk, r0out, r1out, r2out, r3out, r4out, r5out, r6out, r7out, irout);
END arch;
```

Figure 19 - Code VHDL Processeur

On ajoute l'entité « text_screen » au processeur. On donne en entrée les paramètres nécessaires pour la communication VGA ainsi que les sorties des registres.

Affichage



```
ENTITY text_screen IS
    PORT (
        CLOCK_50 : IN STD_LOGIC;
        KEY : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        VGA_R, VGA_G, VGA_B : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
        VGA_HS : OUT STD_LOGIC;
        VGA_VS : OUT STD_LOGIC;

        cpu_clk : IN STD_LOGIC;
        reg0, reg1, reg2, reg3, reg4, reg5, reg6, reg7, din : IN STD_LOGIC_VECTOR(8 DOWNTO 0)
    );
END text_screen;

ARCHITECTURE DE1 OF text_screen IS

    FUNCTION ascii_to_vector(input_num : INTEGER) RETURN STD_LOGIC_VECTOR IS
    BEGIN
        RETURN STD_LOGIC_VECTOR(to_unsigned(input_num, 7));
    END FUNCTION ascii_to_vector;

    FUNCTION ternary_to_ascii(input_vec : STD_LOGIC_VECTOR(2 DOWNTO 0)) RETURN INTEGER IS
    BEGIN
        RETURN to_integer(to_unsigned(to_integer(unsigned(input_vec)), 7) + 48);
    END FUNCTION ternary_to_ascii;

    FUNCTION nibble_to_ascii(input_vec : STD_LOGIC_VECTOR(3 DOWNTO 0)) RETURN INTEGER IS
    BEGIN
        RETURN to_integer(to_unsigned(to_integer(unsigned(input_vec)), 7) + 48);
    END FUNCTION nibble_to_ascii;
```

Figure 20 - Code VHDL VGA #1

Ce code est basé sur lui fournit par le professeur en classe. Les modifications apportées sur le port est principalement l'état du CPU ainsi que les valeurs enregistrées dans chaque registre et dans le DIN. Malgré que les fonctions n'aient pas été couvertes dans le cours, nous avons décidé de prendre l'initiative de faire quelques recherches en ligne afin de simplifier la conversion de vecteurs en certains types de données. Dans

cette entité, nous avons donc créé trois fonctions permettant de simplifier des conversions de vecteurs. La fonction « ascii_to_vector » permet de convertir un « integer » représentant un caractère en vecteur 7 bits. La fonction « ternary_to_ascii » convertie un vecteur de 3 bits en sa valeur ASCII. La fonction « nibble_to_ascii » fait la même chose, mais pour un vecteur de 4 bits.

```
70
71      SIGNAL curr_char : STD_LOGIC_VECTOR(6 DOWNTO 0) := STD_LOGIC_VECTOR(to_unsigned(32, 7));
72
73      SIGNAL reg0h, reg0t, reg0u : STD_LOGIC_VECTOR(3 DOWNTO 0);
74      SIGNAL reg1h, reg1t, reg1u : STD_LOGIC_VECTOR(3 DOWNTO 0);
75      SIGNAL reg2h, reg2t, reg2u : STD_LOGIC_VECTOR(3 DOWNTO 0);
76      SIGNAL reg3h, reg3t, reg3u : STD_LOGIC_VECTOR(3 DOWNTO 0);
77      SIGNAL reg4h, reg4t, reg4u : STD_LOGIC_VECTOR(3 DOWNTO 0);
78      SIGNAL reg5h, reg5t, reg5u : STD_LOGIC_VECTOR(3 DOWNTO 0);
79      SIGNAL reg6h, reg6t, reg6u : STD_LOGIC_VECTOR(3 DOWNTO 0);
80      SIGNAL reg7h, reg7t, reg7u : STD_LOGIC_VECTOR(3 DOWNTO 0);
81
82  BEGIN
83
84      dc0 : digit_converter PORT MAP(reg0, reg0h, reg0t, reg0u);
85      dc1 : digit_converter PORT MAP(reg1, reg1h, reg1t, reg1u);
86      dc2 : digit_converter PORT MAP(reg2, reg2h, reg2t, reg2u);
87      dc3 : digit_converter PORT MAP(reg3, reg3h, reg3t, reg3u);
88      dc4 : digit_converter PORT MAP(reg4, reg4h, reg4t, reg4u);
89      dc5 : digit_converter PORT MAP(reg5, reg5h, reg5t, reg5u);
90      dc6 : digit_converter PORT MAP(reg6, reg6h, reg6t, reg6u);
91      dc7 : digit_converter PORT MAP(reg7, reg7h, reg7t, reg7u);
```

Figure 21 - Code VHDL VGA #2

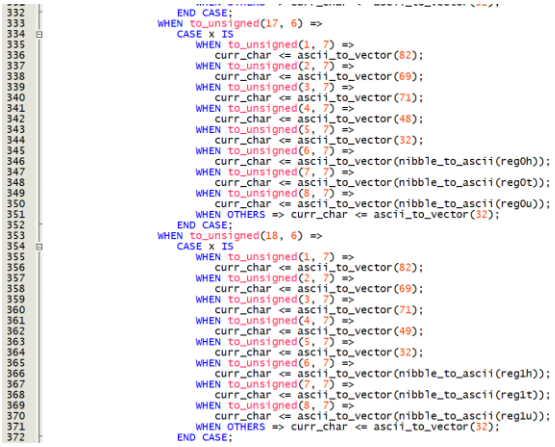
Les modifications ci-dessus ajoute un signal pour l'unité, la dizaine et la centaine de chaque registre. Cela permet de faire un affichage en base 10. Les « digit_converter » sont les entités qui permettent cette conversion d'un vecteur 9 bits à trois vecteurs 3 bits.

```
115         CASE SV IS
116             WHEN clean0 =>
117                 mem_wr <= '1';
118                 mem_in <= curr_char;
119
120                 CASE y IS
121                     WHEN to_unsigned(1, 6) =>
122                         CASE x IS
123                             WHEN to_unsigned(1, 7) =>
124                                 curr_char <= ascii_to_vector(73);
125                             WHEN to_unsigned(2, 7) =>
126                                 curr_char <= ascii_to_vector(78);
127                             WHEN to_unsigned(3, 7) =>
128                                 curr_char <= ascii_to_vector(83);
129                             WHEN to_unsigned(4, 7) =>
130                                 curr_char <= ascii_to_vector(84);
131                             WHEN to_unsigned(5, 7) =>
132                                 curr_char <= ascii_to_vector(82);
133                             WHEN to_unsigned(6, 7) =>
134                                 curr_char <= ascii_to_vector(85);
135                             WHEN to_unsigned(7, 7) =>
136                                 curr_char <= ascii_to_vector(67);
137                             WHEN to_unsigned(8, 7) =>
138                                 curr_char <= ascii_to_vector(84);
139                             WHEN to_unsigned(9, 7) =>
140                                 curr_char <= ascii_to_vector(73);
141                             WHEN to_unsigned(10, 7) =>
142                                 curr_char <= ascii_to_vector(79);
143                             WHEN to_unsigned(11, 7) =>
144                                 curr_char <= ascii_to_vector(78);
145                             WHEN to_unsigned(12, 7) =>
146                                 curr_char <= ascii_to_vector(83);
147                             WHEN OTHERS => curr_char <= ascii_to_vector(32);
148                         END CASE;
149                     WHEN to_unsigned(3, 6) =>
150                         CASE x IS
151                             WHEN to_unsigned(1, 7) =>
152                                 curr_char <= ascii_to_vector(77);
153                             WHEN to_unsigned(2, 7) =>
154                                 curr_char <= ascii_to_vector(86);
155                             WHEN to_unsigned(3, 7) =>
156                                 curr_char <= ascii_to_vector(32);
157                             WHEN to_unsigned(4, 7) =>
158                                 curr_char <= ascii_to_vector(32);
159                             WHEN to_unsigned(5, 7) =>
160                                 curr_char <= ascii_to_vector(48);
161                             WHEN to_unsigned(6, 7) =>
162                                 curr_char <= ascii_to_vector(48);
163                             WHEN to_unsigned(7, 7) =>
164                                 curr_char <= ascii_to_vector(48);
165                             WHEN OTHERS => curr_char <= ascii_to_vector(32);
166                         END CASE;
```

Figure 22 - Code VHDL VGA #3

Afin d'écrire les bons caractères aux bons endroits, on utilise un système de coordonnées X et Y. Puisque l'entité d'affichage VGA écris les caractères ligne par ligne, on vérifie d'abord la ligne actuelle, puis la

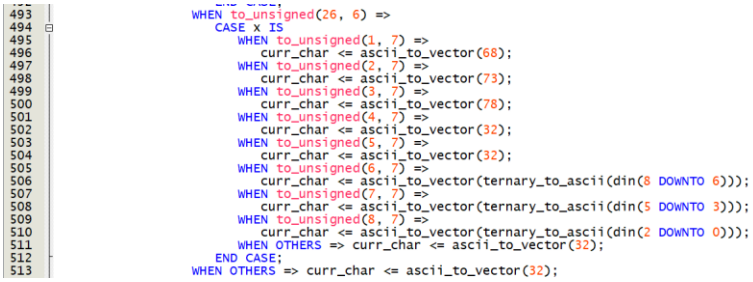
colonne. Le premier cas écrit « Instructions » et le deuxième écris la première instruction disponible (MV). Le reste des instructions est écrit de la même manière.



```
332         END CASE;
333       WHEN to_unsigned(17, 6) =>
334         CASE x IS
335           WHEN to_unsigned(1, 7) =>
336             curr_char <= ascii_to_vector(82);
337           WHEN to_unsigned(2, 7) =>
338             curr_char <= ascii_to_vector(69);
339           WHEN to_unsigned(3, 7) =>
340             curr_char <= ascii_to_vector(71);
341           WHEN to_unsigned(4, 7) =>
342             curr_char <= ascii_to_vector(48);
343           WHEN to_unsigned(5, 7) =>
344             curr_char <= ascii_to_vector(32);
345           WHEN to_unsigned(6, 7) =>
346             curr_char <= ascii_to_vector(nibble_to_ascii(reg0h));
347           WHEN to_unsigned(7, 7) =>
348             curr_char <= ascii_to_vector(nibble_to_ascii(reg0t));
349           WHEN to_unsigned(8, 7) =>
350             curr_char <= ascii_to_vector(nibble_to_ascii(reg0u));
351           WHEN OTHERS => curr_char <= ascii_to_vector(32);
352         END CASE;
353       WHEN to_unsigned(18, 6) =>
354         CASE x IS
355           WHEN to_unsigned(1, 7) =>
356             curr_char <= ascii_to_vector(82);
357           WHEN to_unsigned(2, 7) =>
358             curr_char <= ascii_to_vector(69);
359           WHEN to_unsigned(3, 7) =>
360             curr_char <= ascii_to_vector(71);
361           WHEN to_unsigned(4, 7) =>
362             curr_char <= ascii_to_vector(49);
363           WHEN to_unsigned(5, 7) =>
364             curr_char <= ascii_to_vector(32);
365           WHEN to_unsigned(6, 7) =>
366             curr_char <= ascii_to_vector(nibble_to_ascii(reg1h));
367           WHEN to_unsigned(7, 7) =>
368             curr_char <= ascii_to_vector(nibble_to_ascii(reg1t));
369           WHEN to_unsigned(8, 7) =>
370             curr_char <= ascii_to_vector(nibble_to_ascii(reg1u));
371           WHEN OTHERS => curr_char <= ascii_to_vector(32);
372         END CASE;
```

Figure 23 - Code VHDL VGA #4

La prochaine section de code permet l'écriture dynamique des registres et leurs valeurs. On utilise la fonction expliquée plus haut « nibble_to_ascii » pour transformer les vecteurs en caractères.



```
493       WHEN to_unsigned(26, 6) =>
494         CASE x IS
495           WHEN to_unsigned(1, 7) =>
496             curr_char <= ascii_to_vector(68);
497           WHEN to_unsigned(2, 7) =>
498             curr_char <= ascii_to_vector(73);
499           WHEN to_unsigned(3, 7) =>
500             curr_char <= ascii_to_vector(78);
501           WHEN to_unsigned(4, 7) =>
502             curr_char <= ascii_to_vector(32);
503           WHEN to_unsigned(5, 7) =>
504             curr_char <= ascii_to_vector(32);
505           WHEN to_unsigned(6, 7) =>
506             curr_char <= ascii_to_vector(ternary_to_ascii(din(8 DOWNTO 6)));
507           WHEN to_unsigned(7, 7) =>
508             curr_char <= ascii_to_vector(ternary_to_ascii(din(5 DOWNTO 3)));
509           WHEN to_unsigned(8, 7) =>
510             curr_char <= ascii_to_vector(ternary_to_ascii(din(2 DOWNTO 0)));
511           WHEN OTHERS => curr_char <= ascii_to_vector(32);
512         END CASE;
513       WHEN OTHERS => curr_char <= ascii_to_vector(32);
```

Figure 24 - Code VHDL VGA #5

Le dernier cas permet l'écriture dynamique du « DIN » à l'aide de la fonction « ternary_to_ascii ».

```
513                 WHEN OTHERS => curr_char <= ascii_to
514                 END CASE;
515                 SV <= clean1;
516             WHEN clean1 =>
517                 x <= x + 1;
518                 SV <= clean2;
519             WHEN clean2 =>
520                 mem_wr <= '0';
521                 IF (x > screen_width - 1) THEN
522                     x <= to_unsigned(0, 7);
523                     y <= y + 1;
524                     SV <= clean3;
525                 ELSE
526                     SV <= clean0;
527                 END IF;
528             WHEN clean3 =>
529                 IF (y > screen_height - 1) THEN
530                     SV <= done;
531                 ELSE
532                     SV <= clean0;
533                 END IF;
534             WHEN done =>
535                 IF cpu_clk = '1' THEN
536                     SV <= clean0;
537                 END IF;
538             WHEN OTHERS =>
539                 SV <= clean0;
540         END CASE;
```

Figure 25 - Code VHDL VGA #6

Après avoir passé par-dessus toutes les lignes et colonnes, on atteint l'état « done ». Une fois dans cette état, l'affichage VGA attend un coup d'horloge pour rafraichir l'affichage. Pour rafraichir, on retourne simplement à l'état « clean0 », qui recommence le parcours des lignes et des colonnes.

Digit Converter

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

package digit_converter_constants is
   component digit_converter is
      PORT (
         data : in std_logic_vector(8 downto 0);
         hundred, ten, unit : out std_logic_vector(3 downto 0)
      );
   end component;
end package;

 ----------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


ENTITY digit_converter IS
   PORT (
         data : in std_logic_vector(8 downto 0);
         hundred, ten, unit : out std_logic_vector(3 downto 0)
   );
END digit_converter;

ARCHITECTURE arch OF digit_converter IS
   signal data_u : unsigned(8 downto 0);
   signal mod_h, mod_t, mod_u, mod_diff_ht, mod_diff_tu : unsigned(8 downto 0);
BEGIN
   data_u <= unsigned(data);

   mod_h <= data_u mod 1000;
   mod_t <= data_u mod 100;
   mod_u <= data_u mod 10;

   mod_diff_tu <= mod_t - mod_u;
   mod_diff_ht <= mod_h - mod_t;

   unit <= std_logic_vector(resize(mod_u,4));
   ten <= std_logic_vector(resize(mod_diff_tu / to_unsigned(10, 9),4));
   hundred <= std_logic_vector(resize(mod_diff_ht / to_unsigned(100, 9),4));
end arch;
```

Figure 26 - Code VHDL Digit Converter

Dans cette entité, on convertit un vecteurs 9 bits en trois vecteurs 4 bits représentant l'unité, la dizaine et la centaine. Pour ce faire, on utilise les modulos de 1000, 100 et 10 afin d'avoir les restes de ces trois parties d'un nombre. Une fois chaque reste obtenu, on peut simplement diviser cette valeur par 1 pour les unités, 10 pour les dizaines et 100 pour les centaines pour avoir le chiffre à cette position.

Testbench VGA

Digit Converter

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use ieee.numeric_std.all;
use work.digit_converter_constants.all;

entity digit_converter_tb is
end digit_converter_tb;

architecture test of digit_converter_tb is
        signal data : std_logic_vector(8 downto 0);
        signal hundred, ten, unit : std_logic_vector(3 downto 0);
begin
  DUT: digit_converter port map (data, hundred, ten, unit);

  process
  begin
    report "Testbench starting...";

    data <= "000000001";
    wait for 10 ns;

    data <= "000001000";
    wait for 10 ns;

    data <= "001000000";
    wait for 10 ns;

    data <= "100000000";
    wait for 10 ns;

    data <= "111011001";
    wait for 10 ns;
    wait;
  end process;
end architecture test;

```

Figure 27 - Code VHDL du testbench Digit Converter

Ce testbench vérifie si les vecteurs se font bien convertir en unité, dizaine et centaine.

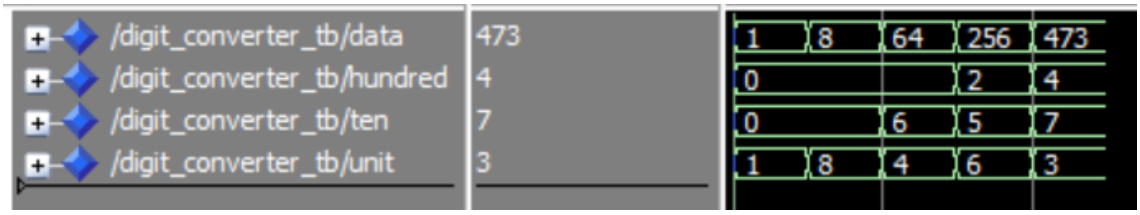


Figure 28 - Résultat du testbench Digit Converter

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

1. Initialise les valeurs
2. Pour « 1 », on a :
    a. Centaine : 0
    b. Dizaine : 0
    c. Unité : 1
3. Pour « 8 », on a :

   a. Centaine : 0
   b. Dizaine : 0
   c. Unité : 8
4. Pour « 64 », on a :
   a. Centaine : 0
   b. Dizaine : 6
   c. Unité : 4
5. Pour « 256 », on a :
   a. Centaine : 2
   b. Dizaine : 5
   c. Unité : 6
6. Pour « 1 », on a :
   a. Centaine : 0
   b. Dizaine : 0
   c. Unité : 1

Communication UART

Préparation

Afin de permettre la communication UART, il faut d'abord indiquer dans le « Pin Planner » les pins qui seront utilisées pour le RX et TX (lecture et écriture). Le GPIO 0 est utilisé comme RX et le GPIO 1 est utilisé comme TX. Voici les connexions sur le DE10-Lite :

Figure 29 - Connexion sur le DE10-Lite

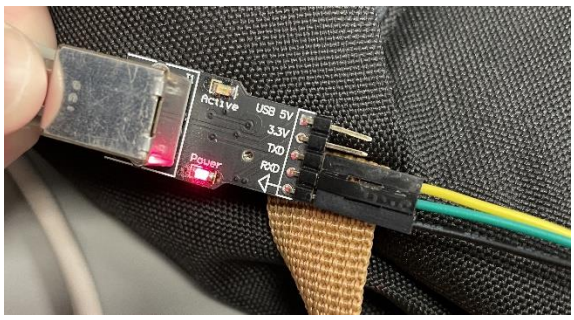


Figure 30 - Connexion sur l'adapteur USB à UART

Interface

```
53
54    uart0 : UART port map(CLOCK_50, rx, tx, done_receiving, uart_ascii, count);
55
56    uartoutput : process(count)
57    begin
58       if done_receiving = '1' then
59          if count = "00" then
60             din(6) <= ascii_to_ternary(uart_ascii)(0);
61             din(7) <= ascii_to_ternary(uart_ascii)(1);
62             din(8) <= ascii_to_ternary(uart_ascii)(2);
63          elsif count = "01" then
64             din(3) <= ascii_to_ternary(uart_ascii)(0);
65             din(4) <= ascii_to_ternary(uart_ascii)(1);
66             din(5) <= ascii_to_ternary(uart_ascii)(2);
67          elsif count = "10" then
68             din(0) <= ascii_to_ternary(uart_ascii)(0);
69             din(1) <= ascii_to_ternary(uart_ascii)(1);
70             din(2) <= ascii_to_ternary(uart_ascii)(2);
71          end if;
72       end if;
73    end process;
74 END arch;
```

Figure 31 - Code VHDL Interface

Cet ajout à l'interface du processeur permet de recevoir les entrées UART et d'insérer ses valeurs dans les bons bits du « DIN ». Dépendant de la valeur du signal « count », on inscrit les valeurs dans les trois premiers bits, trois deuxièmes bits, ou trois derniers bits. Pour ce faire, on utilise la fonction « ascii_to_ternary » pour convertir la valeur ascii du vecteur en vecteur de 3 bits.

UART

```
begin
   process(clock_50, count)
      variable data_receiving: STD_LOGIC :='0';
      variable counter:       integer range 0 to 50000 :=0;
      variable counter_2:     integer range 0 to 50000 :=0;
      variable tx_flag:       std_LOGIC :='0';
      begin
            if clock_50' event and clock_50='1' then --equivalent du VDFF (code du livre)

               --detection du bit start du message
               if (rx='0' and counter=0) then
                     done_receiving <= '0';
                     data_receiving := '1';
                  elsif (counter=2604 and data_receiving='1') then
                        UART_buffer(0)<=rx;

                  elsif (counter=7812 and data_receiving='1') then
                        UART_buffer(1)<=rx;

                  elsif (counter=13020 and data_receiving='1') then
                        UART_buffer(2)<=rx;

                  elsif (counter=18228 and data_receiving='1') then
                        UART_buffer(3)<=rx;

                  elsif (counter=23436 and data_receiving='1') then
                        UART_buffer(4)<=rx;

                  elsif (counter=28644 and data_receiving='1') then
                        UART_buffer(5)<=rx;

                  elsif (counter=33852 and data_receiving='1') then
                        UART_buffer(6)<=rx;

                  elsif (counter=39060 and data_receiving='1') then
                        UART_buffer(7)<=rx;

                  elsif (counter=44268 and data_receiving='1') then
                        UART_buffer(8)<=rx;

                  elsif (rx='1' and data_receiving='1' and counter=49476) then
                        done_receiving <= '1';

                        counter:=0;
                        data_receiving := '0';
                        tx_flag:='1';
                        UART_buffer(9)<=rx;

                        if not(UART_buffer = "000000000") then
                           if count = "10" then
                              count <= "00";
                           else
                              count <= count + 1;
                           end if;
                        end if;
                  end if;
```

Figure 32 - Code VHDL UART

Voici le code UART du cours complété. On ajoute les valeurs de « rx » dans le « UART_buffer » à chaque fois que le compteur atteint une valeur centrée sur le prochain bit. Une fois terminé, on augmente le compteur d'entrée UART qui représente quels 3 bits nous sommes en train de modifier dans le « DIN ». Ce compteur boucle les valeurs 0, 1 et 2.

Conclusion

En conclusion, ce projet de conception nous a permis d'ajouter de nouvelles instructions, un affichage graphique et une entrée clavier au processeur précédemment réaliser. Avec cette nouvelle solution, nous pouvons maintenant voir les registres changer en direct, et écrire l'entrée « DIN » sans l'utilisation d'un vecteur d'interrupteurs. Nous avons aussi eu l'occasion d'expérimenter et de mieux comprendre le fonctionnement du protocole UART et du VGA. Des pistes d'amélioration au projet seraient d'ajouter un historique des instructions, un calcul de puissance « X » à la « Y », et permettre l'écriture naturel des équations par clavier (par exemple reg0 + reg1 ou reg0 = 2).