

---

**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***

## Table des matières

Introduction	3
Modification du processeur simple	4
Processeur	4
ALU	5
FSM	6
Registre	8
Testbench des modifications	8
Processeur	8
ALU	12
FSM	14
Affichage VGA	18
Résultat	18
Processeur	19
Affichage	19
Digit Converter	23
Testbench VGA	24
Digit Converter	24
Communication UART	25
Interface	25
UART	26
Conclusion	26
Références	A

## Liste des figures

Figure 1 - Code VHDL Processeur	4
Figure 2 - Code VHDL ALU	5
Figure 3 - Code VHDL FSM #1	6
Figure 4 - Code VHDL FSM #2	7
Figure 5 - Code VHDL du testbench Processeur #1	9
Figure 6 - Code VHDL du testbench Processeur #2	9
Figure 7 - Code VHDL du testbench Processeur #3	10
Figure 8 - Résultat du testbench Processeur	11
Figure 9 - Code VHDL du testbench ALU	12
Figure 10 - Résultat du testbench ALU	12

Figure 11 - Code VHDL du testbench FSM #1	14
Figure 12 - Code VHDL du testbench FSM #2	15
Figure 13 - Code VHDL du testbench FSM #3	16
Figure 14 - Résultat du testbench FSM	17
Figure 15 - Affichage VGA	18
Figure 16 - Code VHDL Processeur	19
Figure 17 - Code VHDL VGA #1	19
Figure 18 - Code VHDL VGA #2	20
Figure 19 - Code VHDL VGA #3	20
Figure 20 - Code VHDL VGA #4	21
Figure 21 - Code VHDL VGA #5	21
Figure 22 - Code VHDL VGA #6	22
Figure 23 - Code VHDL Digit Converter	23
Figure 24 - Code VHDL du testbench Digit Converter	24
Figure 25 - Résultat du testbench Digit Converter	24
Figure 26 - Code VHDL Interface	25
Figure 27 - Code VHDL UART	26

## Introduction

Dans ce projet de conception, nous devons 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 devons modifier l'interface 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 IIXXXXYYY).

Les instructions déjà présentes 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).

## Modification du processeur simple

### Processeur

```

32
33 ENTITY processeur IS
34   PORT (
35     run, clk, rst : in std_logic;
36     din : in std_logic_vector(8 downto 0);
37     buswire : buffer std_logic_vector(8 downto 0);
38     done : out std_logic;
39
40     CLOCK_50 : in std_logic;
41     KEY : in std_logic_vector(1 downto 0);
42     VGA_R, VGA_G, VGA_B : out std_logic_vector(3 downto 0);
43     VGA_HS : out std_logic;
44     VGA_VS : out std_logic;
45   );
46 END processeur;
47
48 ARCHITECTURE arch OF processeur IS
49   signal r0in, r1in, r2in, r3in, r4in, r5in, r6in, r7in, ain, gin, irin : std_logic := '0';
50   signal incontrol : std_logic_vector(10 downto 0);
51   signal outcontrol : std_logic_vector(10 downto 0);
52   signal r0out, r1out, r2out, r3out, r4out, r5out, r6out, r7out, aout, gout, irout, aluout : std_logic_vector(8 downto 0);
53   signal sb : std_logic_vector(3 downto 0);
54   signal i : std_logic_vector(2 downto 0);
55   signal xReg, yReg : std_logic_vector(7 downto 0);
56 BEGIN
57   r0 : registre port map(buswire, clk, incontrol(0), r0out);
58   r1 : registre port map(buswire, clk, incontrol(1), r1out);
59   r2 : registre port map(buswire, clk, incontrol(2), r2out);
60   r3 : registre port map(buswire, clk, incontrol(3), r3out);
61   r4 : registre port map(buswire, clk, incontrol(4), r4out);
62   r5 : registre port map(buswire, clk, incontrol(5), r5out);
63   r6 : registre port map(buswire, clk, incontrol(6), r6out);
64   r7 : registre port map(buswire, clk, incontrol(7), r7out);
65
66   a : registre port map(buswire, clk, incontrol(8), aout);
67   alu0 : alu port map(aout, buswire, i, aluout);
68   g : registre port map(aluout, clk, incontrol(9), gout);
69
70   ir : registre port map(din, clk, incontrol(10), irout);
71
72   i <= irout(8 downto 6);
73   decX : Dec generic map(3, 8) port map(irout(5 downto 3), xReg);
74   decY : Dec generic map(3, 8) port map(irout(2 downto 0), yReg);
75
76   fsmProc : fsm port map(i, xReg, yReg, run, clk, rst, incontrol, outcontrol, done);
77
78   enc : Enc164 port map("00000" & outcontrol, sb);
79   mux : Muxb10 port map(gout, "000000" & irout(2 downto 0), r7out, r6out, r5out, r4out, r3out, r2out, r1out, r0out, sb, buswire);
80
81   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);
82 END arch;

```

Figure 1 - 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 ARCHITECTURE arch OF alu IS
31 BEGIN
32 calculate : process(a, b, mode)
33 begin
34 case mode is
35 when "010" => c <= std_logic_vector(unsigned(a) + unsigned(b));
36 when "011" => c <= std_logic_vector(unsigned(a) - unsigned(b));
37 when "100" => c <= std_logic_vector(resize(unsigned(a) * unsigned(b), a'length));
38 when "101" => c <= std_logic_vector(unsigned(a) / unsigned(b));
39 when "110" => c <= std_logic_vector(to_unsigned(to_integer(unsigned(a)) ** 2, n));
40 when others => c <= std_logic_vector(unsigned(a) + unsigned(b));
41 end case;
42 end process calculate;
43 END arch;

```

Figure 2 - 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'entière 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 3 - Code VHDL FSM #1**

Les premières modifications de la FSM se trouvent 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 ajouté 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 ajoutée où l'instruction MV et MVI se trouvent puisqu'elle n'a besoin qu'un coup d'horloge.

```

69
70
71 controlsignals: PROCESS (Tstep_Q, i, xReg, yReg)
72 BEGIN
73 CASE Tstep_Q IS
74 WHEN T0 =>
75     done <= '1';
76     inControl <= "1000000000";
77     outControl <= "0010000000";
78 WHEN T1 =>
79     done <= '0';
80
81 CASE i IS
82     when "000" =>
83         inControl <= "000" & xReg;
84         outControl <= "000" & yReg;
85     when "001" =>
86         inControl <= "000" & xReg;
87         outControl <= "0010000000";
88     when "010" | "011" | "100" | "101" | "110" =>
89         inControl <= "0010000000";
90         outControl <= "000" & xReg;
91     when "111" =>
92         inControl <= "0001111111";
93         outControl <= "1000000000";
94     when others =>
95         inControl <= (others => '0');
96         outControl <= (others => '0');
97 END CASE;
98 WHEN T2 =>
99     done <= '0';
100 CASE i IS
101     when "010" | "011" | "100" | "101" | "110" =>
102         inControl <= "0100000000";
103         outControl <= "000" & yReg;
104     when others =>
105         inControl <= (others => '0');
106         outControl <= (others => '0');
107 END CASE;
108 WHEN T3 =>
109     done <= '0';
110 CASE i IS
111     when "010" | "011" | "100" | "101" | "110" =>
112         inControl <= "000" & xReg;
113         outControl <= "0100000000";
114     when others =>
115         inControl <= (others => '0');
116         outControl <= (others => '0');
117 END CASE;
118 END CASE;
119 END PROCESS;

```

**Figure 4 - 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.

```

119 |
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 ».

## Testbench des modifications

### Processeur



```

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  -- mv1
32  din <= "001001010";
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;

```

Figure 5 - 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 6 - Code VHDL du testbench Processeur #2

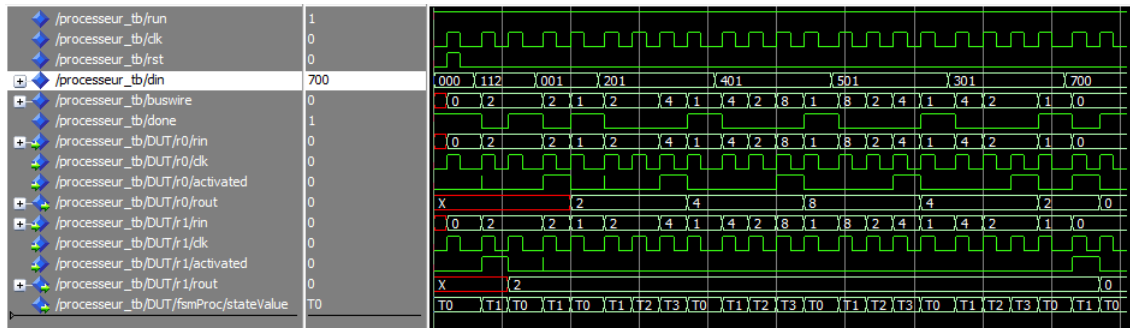
```

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 7 - Code VHDL du testbench Processeur #3**

Ce testbench vérifie effectuée 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 8 - 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 9 - Code VHDL du testbench ALU

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

/alu_tb/mode	-No Data-	0	2		3		4	5	6
/alu_tb/a	-No Data-	0	1	511	1	511	1	10	
/alu_tb/b	-No Data-	0	1		511	1	511	1	2
/alu_tb/c	-No Data-	0	2	0		510	0	20	5 100

Figure 10 - 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 11 - 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;
120     -- end

```

Figure 12 - Code VHDL du testbench FSM #2

```

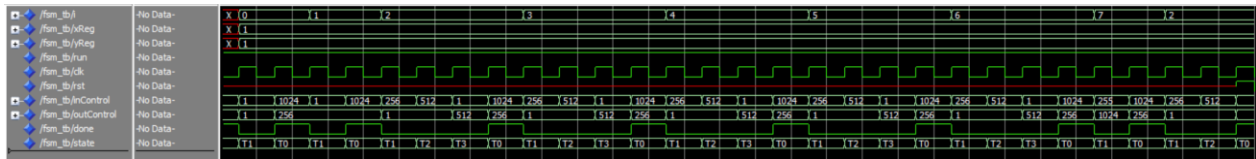
120      -- pow
121      i <= "110";
122      clk <= '1';
123      wait for 10 ns;
124      clk <= '0';
125      wait for 10 ns;
126      clk <= '1';
127      wait for 10 ns;
128      clk <= '0';
129      wait for 10 ns;
130      clk <= '1';
131      wait for 10 ns;
132      clk <= '0';
133      wait 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 13 - 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.





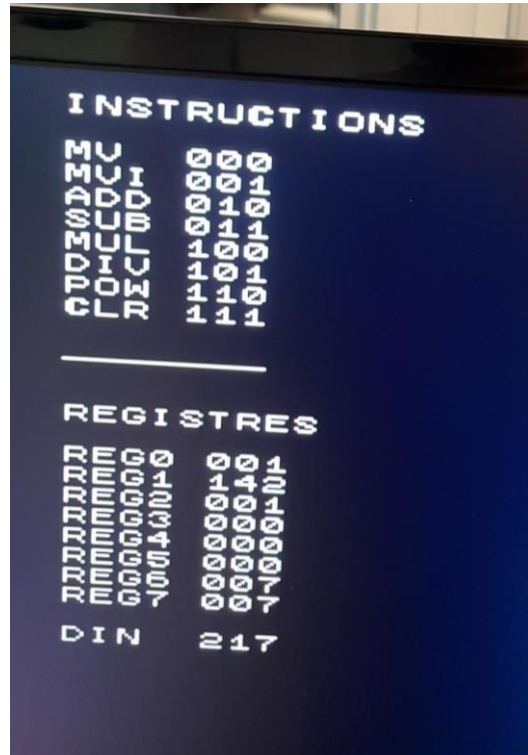
### Figure 14 - Résultat du testbench FSM

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 15 - 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

```

32
33
34 ENTITY processeur IS
35 PORT (
36     run, clk, rst : IN std_logic;
37     din : IN std_logic_vector(8 downto 0);
38     buswire : BUFFER std_logic_vector(8 downto 0);
39     done : OUT std_logic;
40
41     CLOCK_50 : IN std_logic;
42     KEY : IN std_logic_vector(1 DOWNTO 0);
43     VGA_R, VGA_G, VGA_B : OUT std_logic_vector(3 DOWNTO 0);
44     VGA_HS : OUT std_logic;
45     VGA_VS : OUT std_logic;
46 );
47 END processeur;
48
49 ARCHITECTURE arch OF processeur IS
50     signal r0in, r1in, r2in, r3in, r4in, r5in, r6in, r7in, ain, gin, irin : std_logic := '0';
51     signal outcontrol : std_logic_vector(10 downto 0);
52     signal r0out, r1out, r2out, r3out, r4out, r5out, r6out, r7out, aout, gout, irout, aluout : std_logic_vector(8 downto 0);
53     signal sb : std_logic_vector(3 downto 0);
54     signal i : std_logic_vector(2 downto 0);
55     signal xreg, yreg : std_logic_vector(7 downto 0);
56 BEGIN
57     r0 : registre port map(buswire, clk, incontrol(0), r0out);
58     r1 : registre port map(buswire, clk, incontrol(1), r1out);
59     r2 : registre port map(buswire, clk, incontrol(2), r2out);
60     r3 : registre port map(buswire, clk, incontrol(3), r3out);
61     r4 : registre port map(buswire, clk, incontrol(4), r4out);
62     r5 : registre port map(buswire, clk, incontrol(5), r5out);
63     r6 : registre port map(buswire, clk, incontrol(6), r6out);
64     r7 : registre port map(buswire, clk, incontrol(7), r7out);
65
66     a : registre port map(buswire, clk, incontrol(8), aout);
67     alu : alu port map(aluout, buswire, i, aluout);
68     g : registre port map(aluout, clk, incontrol(9), gout);
69     ir : registre port map(din, clk, incontrol(10), irout);
70
71     i <= irout(8 downto 6);
72     decx : Dec generic map(3, 8) port map(irout(5 downto 3), xreg);
73     decy : Dec generic map(3, 8) port map(irout(2 downto 0), yreg);
74
75     fsmProc : fsm port map(i, xreg, yreg, run, clk, rst, incontrol, outcontrol, done);
76
77     enc164 : enc164 port map("00000" & outcontrol, sb);
78     mux10 : mux10 port map(gout, "000000" & irout(2 downto 0), r7out, r6out, r5out, r4out, r3out, r2out, r1out, r0out, sb, buswire);
79
80     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);
81
82 END arch;

```

Figure 16 - 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

```

27
28 ENTITY text_screen IS
29 PORT (
30     CLOCK_50 : IN std_logic;
31     KEY : IN std_logic_vector(1 DOWNTO 0);
32     VGA_R, VGA_G, VGA_B : OUT std_logic_vector(3 DOWNTO 0);
33     VGA_HS : OUT std_logic;
34     VGA_VS : OUT std_logic;
35
36     cpu_clk : IN std_logic;
37     reg0, reg1, reg2, reg3, reg4, reg5, reg6, reg7, din : IN std_logic_vector(8 DOWNTO 0)
38 );
39 END text_screen;
40
41 ARCHITECTURE DE1 OF text_screen IS
42     FUNCTION ascii_to_vector(input_num : INTEGER) RETURN std_logic_vector IS
43     BEGIN
44         RETURN std_logic_vector(to_unsigned(input_num, 7));
45     END FUNCTION ascii_to_vector;
46
47     FUNCTION ternary_to_ascii(input_vec : std_logic_vector(2 DOWNTO 0)) RETURN INTEGER IS
48     BEGIN
49         RETURN to_integer(to_unsigned(to_integer(unsigned(input_vec)), 7) + 48);
50     END FUNCTION ternary_to_ascii;
51
52     FUNCTION nibble_to_ascii(input_vec : std_logic_vector(3 DOWNTO 0)) RETURN INTEGER IS
53     BEGIN
54         RETURN to_integer(to_unsigned(to_integer(unsigned(input_vec)), 7) + 48);
55     END FUNCTION nibble_to_ascii;
56
57 END DE1;

```

Figure 17 - 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. Dans cette entité, nous avons créer 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 18 - 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\_convertter » 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_wir <= '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 19 - 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 écrit les caractères ligne par ligne, on vérifie d'abord la ligne actuelle, puis la colonne. Le premier cas écrit « Instructions » et le deuxième écrit la première instruction disponible (MV). Le reste des instructions est écrit de la même manière.

```

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

```

Figure 20 - 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
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
WHEN to_unsigned(26, 6) =>
CASE x IS
WHEN to_unsigned(1, 7) =>
curr_char <= ascii_to_vector(68);
WHEN to_unsigned(2, 7) =>
curr_char <= ascii_to_vector(73);
WHEN to_unsigned(3, 7) =>
curr_char <= ascii_to_vector(78);
WHEN to_unsigned(4, 7) =>
curr_char <= ascii_to_vector(32);
WHEN to_unsigned(5, 7) =>
curr_char <= ascii_to_vector(32);
WHEN to_unsigned(6, 7) =>
curr_char <= ascii_to_vector(ternary_to_ascii(din(8 DOWNTO 6)));
WHEN to_unsigned(7, 7) =>
curr_char <= ascii_to_vector(ternary_to_ascii(din(5 DOWNTO 3)));
WHEN to_unsigned(8, 7) =>
curr_char <= ascii_to_vector(ternary_to_ascii(din(2 DOWNTO 0)));
WHEN OTHERS => curr_char <= ascii_to_vector(32);
END CASE;
WHEN OTHERS => curr_char <= ascii_to_vector(32);

```

Figure 21 - 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_tc
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 22 - Code VHDL VGA #6**

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

## Digit Converter

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3
4 package digit_converter_constants is
5     component digit_converter is
6     port (
7         data : in std_logic_vector(8 downto 0);
8         hundred, ten, unit : out std_logic_vector(3 downto 0)
9     );
10    end component;
11 end package;
12
13 -----
14
15 LIBRARY ieee;
16 USE ieee.std_logic_1164.all;
17 use ieee.numeric_std.all;
18 use IEEE.STD_LOGIC_UNSIGNED.ALL;
19
20
21 ENTITY digit_converter IS
22     PORT (
23         data : in std_logic_vector(8 downto 0);
24         hundred, ten, unit : out std_logic_vector(3 downto 0)
25     );
26 END digit_converter;
27
28 ARCHITECTURE arch OF digit_converter IS
29     signal data_u : unsigned(8 downto 0);
30     signal mod_h, mod_t, mod_u, mod_diff_ht, mod_diff_tu : unsigned(8 downto 0);
31 BEGIN
32     data_u <= unsigned(data);
33
34     mod_h <= data_u mod 1000;
35     mod_t <= data_u mod 100;
36     mod_u <= data_u mod 10;
37
38     mod_diff_tu <= mod_t - mod_u;
39     mod_diff_ht <= mod_h - mod_t;
40
41     unit <= std_logic_vector(resize(mod_u,4));
42     ten <= std_logic_vector(resize(mod_diff_tu / to_unsigned(10, 9),4));
43     hundred <= std_logic_vector(resize(mod_diff_ht / to_unsigned(100, 9),4));
44 end arch;

```

**Figure 23 - 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 modulus 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

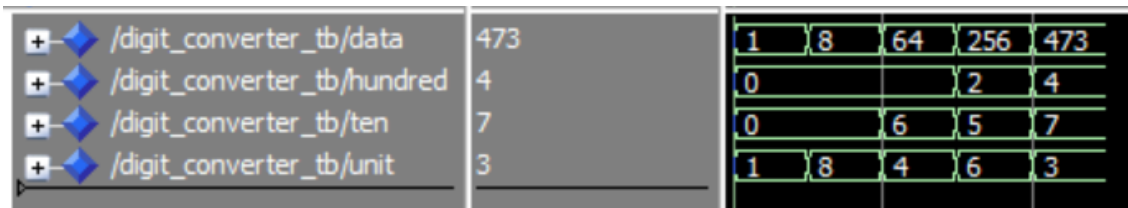
```

1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  use ieee.numeric_std.all;
4  use work.digit_converter_constants.all;
5
6  entity digit_converter_tb is
7  end digit_converter_tb;
8
9  architecture test of digit_converter_tb is
10     signal data : std_logic_vector(8 downto 0);
11     signal hundred, ten, unit : std_logic_vector(3 downto 0);
12 begin
13     DUT: digit_converter port map (data, hundred, ten, unit);
14
15     process
16     begin
17         report "Testbench starting..";
18
19         data <= "000000001";
20         wait for 10 ns;
21
22         data <= "000001000";
23         wait for 10 ns;
24
25         data <= "001000000";
26         wait for 10 ns;
27
28         data <= "100000000";
29         wait for 10 ns;
30
31         data <= "111011001";
32         wait for 10 ns;
33         wait;
34     end process;
35 end architecture test;
36

```

**Figure 24 - Code VHDL du testbench Digit Converter**

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



**Figure 25 - 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

### Interface

```

53  uart0 : Uart port map (CLOCK_50, rx, tx, done_receiving, uart_ascii, count);
54
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 26 - 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 deuxième 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

```

36 begin
37   process(clock_50, count)
38     variable data_receiving: STD_LOGIC := '0';
39     variable counter: integer range 0 to 50000 := 0;
40     variable counter_2: integer range 0 to 50000 := 0;
41     variable tx_flag: std_LOGIC := '0';
42     begin
43       if clock_50' event and clock_50='1' then --équivalent du VDDF (code du livre)
44         --detection du bit start du message
45         if (rx='0' and counter=0) then
46           done_receiving <= '0';
47           data_receiving := '1';
48           elsif (counter=2604 and data_receiving='1') then
49             UART_buffer(0) <= rx;
50           elsif (counter=7812 and data_receiving='1') then
51             UART_buffer(1) <= rx;
52           elsif (counter=13020 and data_receiving='1') then
53             UART_buffer(2) <= rx;
54           elsif (counter=18228 and data_receiving='1') then
55             UART_buffer(3) <= rx;
56           elsif (counter=23436 and data_receiving='1') then
57             UART_buffer(4) <= rx;
58           elsif (counter=28644 and data_receiving='1') then
59             UART_buffer(5) <= rx;
60           elsif (counter=33852 and data_receiving='1') then
61             UART_buffer(6) <= rx;
62           elsif (counter=39060 and data_receiving='1') then
63             UART_buffer(7) <= rx;
64           elsif (counter=44268 and data_receiving='1') then
65             UART_buffer(8) <= rx;
66           elsif (rx='1' and data_receiving='1' and counter=49476) then
67             done_receiving <= '1';
68             counter:=0;
69             data_receiving := '0';
70             tx_flag:= '1';
71             UART_buffer(9) <= rx;
72             if not(UART_buffer = "00000000") then
73               if count = "10" then
74                 count <= "00";
75               else
76                 count <= count + 1;
77               end if;
78             end if;
79           end if;
80         end if;
81       end if;
82     end process;
83   end if;
84 end;

```

Figure 27 - 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  $\text{reg0} + \text{reg1}$  ou  $\text{reg0} = 2$ ).



## Références

Aucune référence.