Ecole Centrale de Nantes
Master of Science - Control Robotics
Embedded Electronics

Implementing a Calculator

CPLD Xilinx CoolRunner II

Erwin Lejeune & Jean-Maxime Le Carpentier
ERTS SIP

Supervisors:
Joumana LAGHA

First version

Nantes, OCTOBER 2019

Table of Contents

1 Introduction **1**

1.1 Presentation of the Hardware 1

1.2 Purpose 1

2 Experiments **2**

2.1 Specifications 2

2.1.1 Functions 2

2.2 Tasks 3

2.2.1 Introduction to 7-segment Display 3

2.2.2 Displaying a chosen number 4

2.2.3 Displaying ”2018” 6

2.2.4 Push-Buttons 8

2.2.5 Calculator 9

3 Results and discussion **11**

3.1 Failures and Fixes 11

4 Conclusion **12**

1. Introduction

1.1 Presentation of the Hardware

A CPLD or Complex Programmable Logic Device is a programmable logic device similar to a FPGA (Field-Programmable Gate Array). Its complexity is inferior to an FPGAs, but it contains a large amount of architectural features from it. The main building block of the CPLD is a macrocell, which contains logic implementing disjunctive normal form expressions and more specialized logic operations.

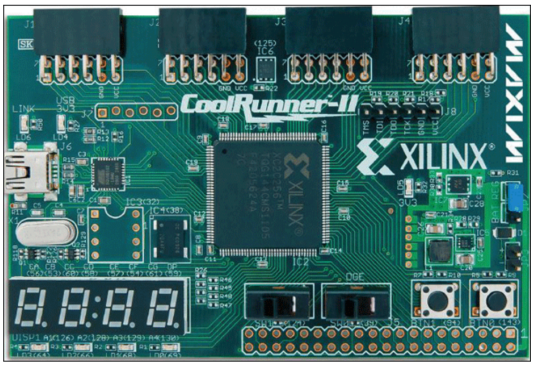

Figure 1.1: Xilinx CoolRunner II, used during labs

CPLDs were an evolutionary step from even smaller devices that preceded them, PLAs (first shipped by Signetics), and PALs.

CPLDs were a direct evolution from even smaller devices that came prior to them : PLAs (programmable logic array) and PALs (programmable array logic). The main difference between CPLD and FPGA architecture is internal : the prior is forming its logic functions with sea-of-gates and the later on look-up tables (LUT). A lookup table is an array that replaces run-time computation with a simpler array indexing operation. Processing time can be reduced significantly by applying this method.

1.2 Purpose

Ultimately, the goal is to display how to make a calculator with a 7-segments display showing:

- The result of the function applied on the two left digits ;
- The two numbers being computed on the two right digits.

It is specified to use switch buttons to choose the function and push buttons to choose which numbers to compute. A more in-depth description of the functions will be done in section 2.1.

2. Experiments

2.1 Specifications

As specified in the introduction, the purpose of this lab is to implement a calculator with different functions in a CPLD using VHDL description language. The architecture, as shown below in Figure 2.1 is divided in four blocks.

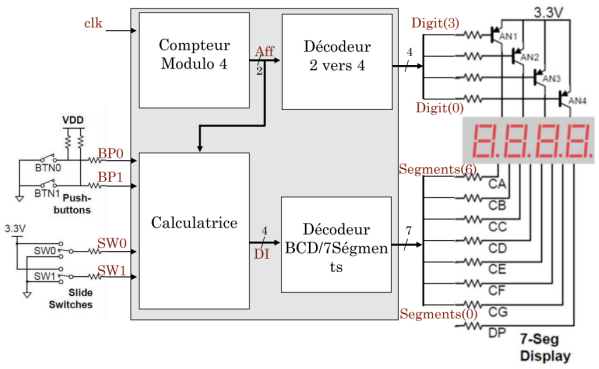

Figure 2.1: Architecture of the full Calculator

It counts five inputs : a clock, two push buttons and two switch buttons. It counts two outputs : one to choose the segment to be lit up, the other to choose the digit that should display the number chosen. The counter allows to choose the digit on which a number should be displayed. It counts from zero to four then resets, allowing to select the Digit one is writing on as well as what one is writing in the said digit.

2.1.1 Functions

The calculator should be able to perform an addition, a subtraction and a product. The two numbers we operate on are defined by the push buttons. Each time one presses a button, both numbers increments by 1 from 0 to 9. The functions are selected thanks to the switches :

- Addition : not(SW0) and not(SW1) ;
- Subtraction : SW0 and not(SW1) ;
- Multiplication : SW1.

2.2 Tasks

2.2.1 Introduction to 7-segment Display

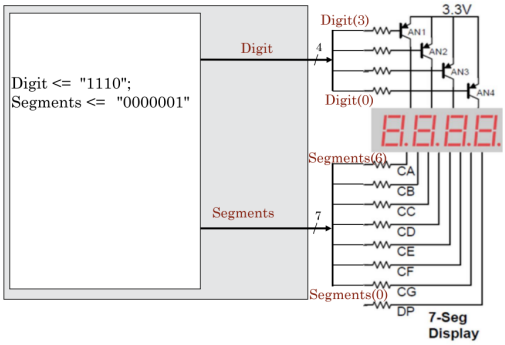

Figure 2.2: Architecture of Task 1

The first task specified the 7-segments display should show the value 0 on the digit 0. No need for a counter yet, no need for inputs. Just two outputs Digit and Segments that are linked to the right pins in the netlist.

```
34  entity command_display is
35      Port ( Digit : out  STD_LOGIC_VECTOR (3 downto 0);
36             Segments : out  STD_LOGIC_VECTOR (6 downto 0));
37  end command_display;
38
39  architecture Behavioral of command_display is
40
41  begin
42
43  Digit <= "1110";
44  Segments <= "0000001";
45
46  end Behavioral;
```

Figure 2.3: VHDL Description of Task 1

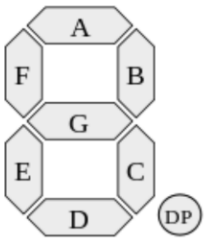

Figure 2.4: 7-Segment Help

Writing "1110" in Digit chooses the first digit as seen in the schematics and after testing. According to the figure 2.4 above, and considering writing '0' is actually turning the segment on, we deduced that writing "0000001" would write 0.

The netlist below links the outputs to the corresponding pins according to the documentation provided.

```
1  NET "Digit<0>" LOC = "p126";
2  NET "Digit<1>" LOC = "p128";
3  NET "Digit<2>" LOC = "p129";
4  NET "Digit<3>" LOC = "p130";
5
6  NET "Segments<6>" LOC = "p56";
7  NET "Segments<5>" LOC = "p53";
8  NET "Segments<4>" LOC = "p60";
9  NET "Segments<3>" LOC = "p58";
10 NET "Segments<2>" LOC = "p57";
11 NET "Segments<1>" LOC = "p54";
12 NET "Segments<0>" LOC = "p61";
```

Figure 2.5: Netlist of Task 1

The result after implementing the description in the CPLD :

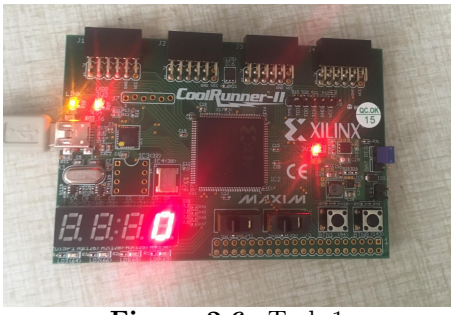

Figure 2.6: Task 1

2.2.2 Displaying a chosen number

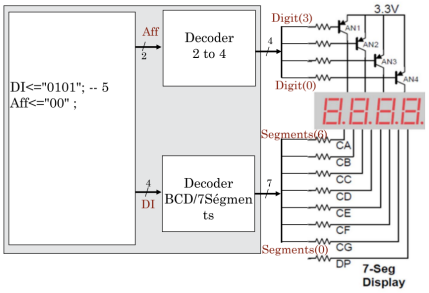

Figure 2.7: Architecture of Task 2

Here, the purpose was to use a "DI" signal that could take a value from 0 to 9 and display that value in the 7-Segment without having to manually choose which segment one would want to turn on every time. To do this, it was asked to describe a decoder 2 to 4 to choose the digit one was to display a number on, as well as a decoder 4 to 7 to go from a chosen number between 0 and 9 to the associated series of 1 and 0 we should write in the Segments bit vector. To do this, we just select Segments according to the value of DI, and we choose Digit according to the value of Disp, as shown below in our entity.

```
34  entity command_display is
35      Port ( Digit : out  STD_LOGIC_VECTOR (3 downto 0);
36             Segments : out  STD_LOGIC_VECTOR (6 downto 0));
37  end command_display;
38
39  architecture Behavioral of command_display is
40
41  signal Disp : std_logic_vector(1 DOWNTO 0);
42  signal DI : std_logic_vector(3 DOWNTO 0);
43
44  begin
45
46  DI <= "0101";
47  Disp <= "00";
48
49  with DI select Segments <=
50     "0000001" when "0000",
51     "1001111" when "0001",
52     "0010010" when "0010",
53     "0000110" when "0011",
54     "1001100" when "0100",
55     "0100100" when "0101",
56     "0100000" when "0110",
57     "0001111" when "0111",
58     "0000000" when "1000",
59     "0000100" when "1001",
60     "1111111" when others;
61
62  with Disp select Digit <=
63     "1110" when "00",
64     "1101" when "01",
65     "1011" when "10",
66     "0111" when "11",
67     "0000" when others;
68
69  end Behavioral;
```

Figure 2.8: VHDL Description of Task 2

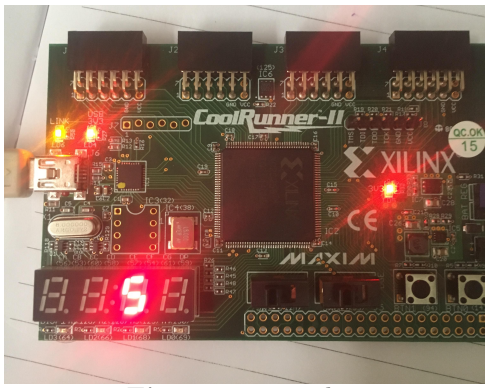

Figure 2.9: Task 2

It was an arbitrary decision to keep all the blocks in one architecture instead of using components, and that's what has been consistently used throughout the tasks instead of segmenting every blocks as one component. The next step is to use every digit and display a 4 digits number.

2.2.3 Displaying "2018"

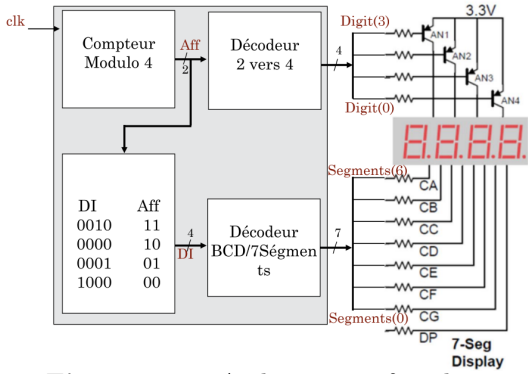

Figure 2.10: Architecture of Task 3

During this task, difficulties were encountered as to displaying on 4 digits the right number. Indeed, the counter block was added to the architecture. It counts from 0 to 3 and its purpose is to select the digit one wants to display a chosen number on. For this, a basic counter was realized, and coincidentally with the value of the signal "Disp" that is incremented, the number one wants to select is chosen.

```
34  entity command_display is
35      Port ( CLK : in STD_LOGIC ;
36             Digit : out  STD_LOGIC_VECTOR (3 downto 0);
37             Segments : out  STD_LOGIC_VECTOR (6 downto 0));
38  end command_display;
39
40  architecture Behavioral of command_display is
41
42  signal Disp : std_logic_vector(1 DOWNTO 0);
43  signal DI : std_logic_vector(3 DOWNTO 0);
44  signal divFreq : std_logic_vector(6 downto 0);
45
46  begin
47
48  process(CLK)
49
50  begin
51
52     if (CLK'event and CLK = '1') then
53        if (divFreq = 99) then
54           Disp <= Disp + 1;
55           divFreq <= "0000000";
56        else
57           divFreq <= divFreq + 1;
58        end if;
59     end if;
60
61  end process;
```

Figure 2.11: VHDL Description of Task 3

The counter is processed with the clock input that is going at the hardware frequency. This caused issues with the display. The counter was going too fast to be processed and seen with our eyes. To counter this problem, the Pulse-Width Modulation introduction helped. A clock divider was used to reduce the frequency as shown in figure 2.11. The signal "Disp" is only incremented when the "divFreq" signal has reached 99, which slows down the process and actually allow the right value to be displayed.

After the counter started to show good results, "Disp" had to change the number defined by the signal "DI" that is to be displayed. For this we just added another "with select" statement as shown in the figure below.

```
63  with Disp select DI <=
64     "1000" when "00",
65     "0001" when "01",
66     "0000" when "10",
67     "0010" when "11",
68     "0000" when others;
69
70  with DI select Segments <=
71     "0000001" when "0000",
72     "1001111" when "0001",
73     "0010010" when "0010",
74     "0000110" when "0011",
75     "1001100" when "0100",
76     "0100100" when "0101",
77     "0100000" when "0110",
78     "0001111" when "0111",
79     "0000000" when "1000",
80     "0000100" when "1001",
81     "1111111" when others;
82
83  with Disp select Digit <=
84     "1110" when "00",
85     "1101" when "01",
86     "1011" when "10",
87     "0111" when "11",
88     "0000" when others;
89
90  end Behavioral;
```

Figure 2.12: VHDL Description of Task 3 (bis)

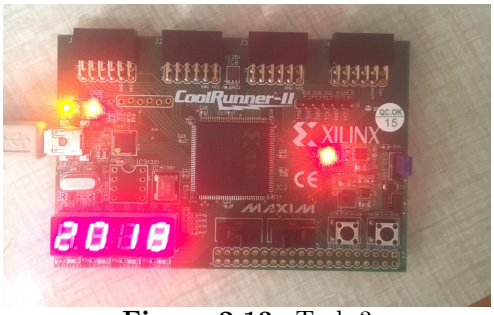

Figure 2.13: Task 3

2.2.4 Push-Buttons

Adding the push-buttons was actually one of the easiest task since it was merely just adding two other signals "x" and "y" and two inputs "BP0" and "BP1". In a basic counter fashion, x and y are incremented by 1 each time a button is pushed. The pushed detection means a button changed state, characterised by a falling edge or a rising edge. Here a falling edge was used since the pushed state is believed to be a digital 0 and the unpushed state a digital 1.

```
74  process (Disp)
75
76  variable K : integer range 0 to 10;
77
78  begin
79
80     -- Increment left digit every time we push BP0
81     if (falling_edge(BP0)) then
82        x <= x+1;
83        if ( x = "1001" ) then
84           x <= "0000";
85        end if;
86     end if;
87
88     -- Increment right digit every time we push BP1
89     if (falling_edge(BP1)) then
90        y <= y + 1;
91        if ( y = "1001" ) then
92           y <= "0000";
93        end if;
94     end if;
```

Figure 2.14: VHDL Description of Task 4

The "K" variable will be used in section 2.2.5 and has no incidence on this task. It was chosen to deal with those in the "Disp" process because of the way the Architecture is described in figure 2.10. Indeed, Decoder and the DI chosen are incidental to the "Disp" signal. The other option was to process on both buttons, but it's a bad idea to make them sensitive to themselves. After incrementing x and y, the last step was to put them in the select shown in figure 2.11 in the select of "DI". Unfortunately it wasn't possible to just change it like this. Since everything had to be in the "Disp" process with our recent choice, the use of a "with select" statement was not possible, so we switched to a "case is".

The results of the fourth task can be seen in the video attached below :

Click here to access the push-button results video...

2.2.5 Calculator

The thought process for the calculator was quite simple at first. The problem did not seem that hard. Basically implement addition, subtraction and multiplication with the integer values of the bit-vectors depending on the states of the switches and filter the units and tens of the returned number by using the "mod" and "division" operators. The units would be the remaining of the numbers mod 10. The tens would be the integer part of the division by ten. Unfortunately, VHDL doesn't allow to divide or use mod/remainder by numbers that are not multiple of two. A way to bypass that problem is to use shifts this way :

n = a/(2^y) <= ($n = a >> y$)

However, it is not the way used in this project. Instead of having the result as an integer variable, it was a choice at first to have it as a bit vector with a size superior to 81.

81 ¡ $2^6 + 2^5$

As seen right above, 6 bits seemed enough at first for our result variable. Unfortunately with this size as soon as the result was superior to 64, the calculation started to be wrong. It was only logical that the problem came from this, so the size of the bit vector was increased to 7 bits. To complete an addition with two 4 bits vectors in a result of 7 bits, it is necessary to perform a logical "and 000" on both operands so that they are the right size before computation. That way results is a result of two 7 bits operands with three 0 put at the most significant bits.

```
 96    -- Calculator
 97    if (SW0 = '0' and SW1 = '0') then
 98       -- ADDITION
 99       result <= ("000" & x) + ("000" & y);
100
101    elsif (SW0 = '1' and SW1 = '0') then
102       -- SUBSTRACTION
103       if (x >= y) then
104          z1 <= x - y;
105          z10 <= "1111";
106       else
107          z1 <= y - x;
108          z10 <= "1101";
109       end if;
110
111    elsif (SW1 = '1') then
112       -- MULTIPLICATION
113       result <= x * y;
114
115    else
116       -- ELSE ( shouldn't happen ) put everything to zero
117       z1 <= "0000";
118       z10 <= "0000";
119    end if;
```

Figure 2.15: VHDL Description of Calculator Functions

After the operations were performed, the most complicated task was to separate units and tens in order to have the correct result displayed. As said before, the first thing tried was to use mod and div operators. Unfortunately this process was not conclusive and the mutual decision was to move away from it.

```
121     -- Only differentiate units and tens when we're not in SUBSTRACTION case
122     if ((SW1 = '1') or (SW0 = '0' and SW1 = '0')) then
123        -- Increment from 1 to 9 ( 81 is the max value and it's inferior to 90)
124        for K in 1 to 9 loop
125           if ( 10*K = result ) then
126              -- If multiple of 10 then units is 0 and tens is K
127              z1 <= "0000";
128              z10 <= conv_std_logic_vector(K, 4);
129              exit;
130           elsif ( 10*K > result ) then
131              -- As soon as 10*K passes by the result, then we can compute the value
132              z1 <= result - 10*(K-1);
133              z10 <= conv_std_logic_vector(K-1, 4);
134              exit;
135           end if;
136        end loop;
137     end if;
```

Figure 2.16: VHDL Description of the Units/Tens Display

First step was to exclude the subtraction from this part, since its display was performed in the operation part : indeed, a subtraction by two units has a unit as a result, meaning result is the unit, and the tens is to be chosen : it was a choice to put a "-" instead of only perform subtraction of the greater number by the smaller. If the first number is smaller, we display a "-" as a minus sign. This required to add another line to the segments select where we turn off everything but the dash (the inverse of displaying 0).

For every other cases, an integer is incremented from 1 to 9. If the result of ten multiplied by this integer is the same as the result of the operation, then the tens is the counting integer and the units is zero. The loop breaks when this condition is met. Otherwise, as soon as ten multiplied by the integer exceeds the result of the operation, then the operation in figure 2.16 is performed. The integer has to be converted into a bit vector of size 4 before being assigned to our signals.

The results of the operation can be seen below :

Click here to access the full calculator results video...

3. Results and discussion

3.1 Failures and Fixes

Here is a photo of the display problem that occurred because of the too high frequency of the counter :

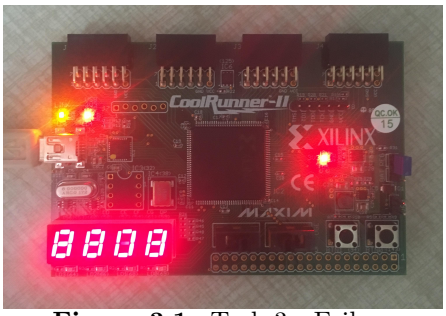

Figure 3.1: Task 3 - Failure

Some mistakes were very time consuming. The frequency divider problem was fixed in no time, but at the push-button tasked a lot of time was spent trying to tweak the description code because the numbers were jumping at times, as if the button was pressed twice in one real press. After wasting quite some time on it, the decision of changing the chip (at that point hardware was the only option possible for the issue) was made and was conclusive. Indeed, a new chip stopped the buttons from jumping. It varied from chips to chips.

As said above, the first method to differentiate units from tens in the digits was the remainder and the division. It was also stated that this method was not used in the end and was excluded from every options. To avoid time consumption, personal work was done about this between the second and the third lab, to think about a way to actually succeed at using this method. When in the lab, it was a choice from both members to only spend thirty minutes to one hour trying to use this one, and if not conclusive, switch back to the loop method.

4. Conclusion

Overall, the project allowed us to comprehend the ISE suite as well as deepen our knowledge and abilities with the VHDL description language and the use of CPLDs. Both of worked on FPGA before, using a different IDE, a different brand (Altera) and one of us actually worked on both FPGA and CPLD, using Quartus and the ISE suite. Even though we went through some projects before with VHDL, it was the first time we stumbled on the issue with the division and the remainder, helping us comprehend even more the functioning of Digital Design and the importance of knowing those things.

Going too fast is definitely not effective but not having a global vision of the project could cause software architecture problems towards the end of it. Without having that vision, we might have gone with a component approach which apparently was not the right way, according to the architecture given. It is always a challenge to keep working on small tasks while having that global idea of where we're going, which supposedly went well with our group, even though we still have room for improvement.