

27-11-2021

## Lab#4



– Digital Electronics –

## Building and programming a simple NIOS system

### Students

Masson Nicolas s293826

Callegarin Demangeat Martin s294064

Wojciech Paluszkiewicz s293958

Muhammad Arshad s288059

The idea of this Lab is to exploit the nios system creator in order to simplify the complex connexions between the Nios processor and the peripherals we, the users, will manipulate on the DE1 FPGA.

For a start we had to follow the tutorial on how to create a Nios system to familiarize ourselves with the tools. The comments about following a step-by-step tutorial being irrelevant for this report, we won't be discussing minor problems we've encountered. Instead, we present you on figure 1 the system we've created that will be used during this lab project.

Platform Designer - nios\_system2.qsys (D:\download\quartus\projects\lab4\nios\_system2.qsys)  
 Edit System Generate View Tools Help

System Contents Address Map Interconnect Requirements

System: nios\_system2 Path: dk\_0

Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Tags
		clk_in	Clock Input	clk	exported				
		clk_in_reset	Reset Input	reset					
		clk	Clock Output	Double-click to export	clk_0				
		clk_reset	Reset Output	Double-click to export					
<input checked="" type="checkbox"/>		nios2_qsys_0	Nios II (Classic) Processor						
		clk	Clock Input	Double-click to export	clk_0				
		reset_in	Reset Input	Double-click to export	[ck]				
		data_master	Avalon Memory Mapped Master	Double-click to export	[ck]				
		instruction_master	Avalon Memory Mapped Master	Double-click to export	[ck]				
		d_irq	Interrupt Receiver	Double-click to export	[ck]			IRQ 0	IRQ 31
		jtag_debug_module_r...	Reset Output	Double-click to export	[ck]				
		jtag_debug_module	Avalon Memory Mapped Slave	Double-click to export	[ck]				
		custom_instruction_m...	Custom Instruction Master	Double-click to export	[ck]				
<input checked="" type="checkbox"/>		onchip_memory2_0	On-Chip Memory (RAM or ROM) Intel ...						
		clk1	Clock Input	Double-click to export	clk_0				
		s1	Avalon Memory Mapped Slave	Double-click to export	[ck1]	0x0000	0x0fff		
		reset1	Reset Input	Double-click to export					
<input checked="" type="checkbox"/>		keys	P10 (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	Double-click to export	clk_0				
		reset	Reset Input	Double-click to export	[ck]				
		s1	Avalon Memory Mapped Slave	Double-click to export	[ck]	0x2040	0x204f		
		external_connection	Conduit						
<input checked="" type="checkbox"/>		switches	P10 (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	Double-click to export	clk_0				
		reset	Reset Input	Double-click to export	[ck]				
		s1	Avalon Memory Mapped Slave	Double-click to export	[ck]	0x2000	0x200f		
		external_connection	Conduit						
<input checked="" type="checkbox"/>		leds	P10 (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	Double-click to export	clk_0				
		reset	Reset Input	Double-click to export	[ck]				
		s1	Avalon Memory Mapped Slave	Double-click to export	[ck]	0x2010	0x201f		
		external_connection	Conduit						
<input checked="" type="checkbox"/>		hex0	P10 (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	Double-click to export	clk_0				
		reset	Reset Input	Double-click to export	[ck]				
		s1	Avalon Memory Mapped Slave	Double-click to export	[ck]	0x2020	0x202f		
		external_connection	Conduit						
<input checked="" type="checkbox"/>		hex1	P10 (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	Double-click to export	clk_0				
		reset	Reset Input	Double-click to export	[ck]				
		s1	Avalon Memory Mapped Slave	Double-click to export	[ck]	0x2030	0x203f		
		external_connection	Conduit						
<input checked="" type="checkbox"/>		jtag_uart_0	JTAG UART Intel FPGA IP						
		clk	Clock Input	Double-click to export	clk_0				
		reset	Reset Input	Double-click to export	[ck]				
		avalon_jtag_slave	Avalon Memory Mapped Slave	Double-click to export	[ck]	0x2050	0x205f		
		irq	Interrupt Sender	Double-click to export	[ck]				

Current filter:

ors, 1 Warning

Figure 1: Nios system configuration

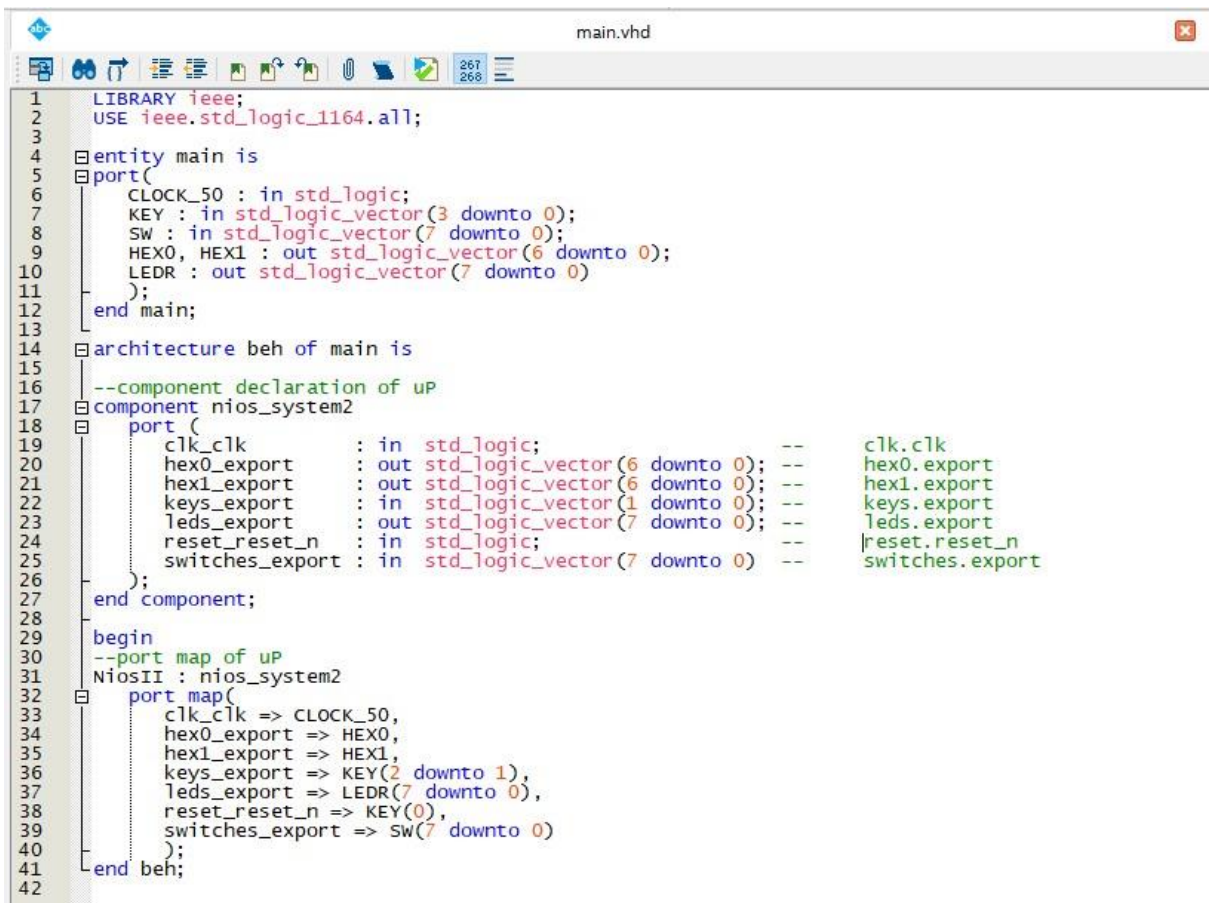
- Building a NIOS system

After instantiated the processors with its memory-on-chip, it's peripherals and the Jtag to manage the interruptions, we proceeded to connect the different "components" altogether to make the system works properly.

Afterward the system being only conscious about himself, we had to plug its entities with the one we'll be using which are: the FPGA peripherals and clocks. This part is done within the Quartus main entity project as it is mostly hardwiring in VHDL. In it we declare the Nios system as a component with its inputs and outputs. The inside of this component being written by the Qsys software that we've created.

Then we connect the peripherals and the clock with the different buttons, switches and displayers of the DE1 board. The clock being the 50 MHz one of the FPGA, it will also be the Nios clock.

It is relevant to notice that the asynchronous reset of the Nios system is directly connected to the KEY(0) of our board. Thus, while being ignorant about what all the other peripherals will be doing in the c programs we'll be creating, we know for sure that the key 0 of the DE1 board will act as the program reset as it is directly wired to the Nios system. It's also relevant to notice that it's not the FPGA reset, therefore we will still have the wiring of the Nios after a key0 reset. While the reset present on the FPGA would discard our whole program and wirings.



```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3
4  entity main is
5  port(
6      CLOCK_50 : in std_logic;
7      KEY : in std_logic_vector(3 downto 0);
8      SW : in std_logic_vector(7 downto 0);
9      HEX0, HEX1 : out std_logic_vector(6 downto 0);
10     LEDR : out std_logic_vector(7 downto 0)
11 );
12 end main;
13
14 architecture beh of main is
15
16     --component declaration of uP
17     component nios_system2
18     port (
19         clk_clk      : in  std_logic;           -- clk.clk
20         hex0_export  : out std_logic_vector(6 downto 0); -- hex0.export
21         hex1_export  : out std_logic_vector(6 downto 0); -- hex1.export
22         keys_export  : in  std_logic_vector(1 downto 0); -- keys.export
23         leds_export  : out std_logic_vector(7 downto 0); -- leds.export
24         reset_reset_n : in  std_logic;           -- |reset.reset_n
25         switches_export : in  std_logic_vector(7 downto 0) -- switches.export
26     );
27 end component;
28
29 begin
30     --port map of uP
31     NiosII : nios_system2
32     port map(
33         clk_clk => CLOCK_50,
34         hex0_export => HEX0,
35         hex1_export => HEX1,
36         keys_export => KEY(2 downto 1),
37         leds_export => LEDR(7 downto 0),
38         reset_reset_n => KEY(0),
39         switches_export => Sw(7 downto 0)
40     );
41 end beh;
```

Figure 2 : VHDL wiring of the Nios system and DE1 board.

After compiling and sending data bits to the board, the Nios system is comfortably set, and we can start to program on c with the Eclipse software.

- Do it yourself

- Connecting the switches to the LEDs

This exercise is the same as the one taken in appendix to run our first code with using a NIOS processor. It uses 2 main functions: IORD and IOWR which are Macros generated by io.h. IORD implies a reading peripheral register and IOWR a writing one. Their syntax is the following (taken from the lecture of NIOS II and Qsys) :

$$e_i = \sum_{n=0}^i \frac{1}{n!}$$

IORD\_ALTERA\_AVALON\_PIO\_DATA(base)

So here, as the switches are the input of our code, we use their exact name in 'base'.

IOWR\_ALTERA\_AVALON\_PIO\_DATA(base,val)

As the LEDs are our output, we put their exact name in 'base'. Coding the integer & 0xFF leaves only the last 8 significant bits, that is called 'masking'. In this code it's not necessary but we'll see it's utility later on.

Alt\_putstr acts like a print and while(1) is just an infinite loop.

```
#include "system.h"
#include "sys/alt_stdio.h"
#include "altera_avalon_pio_regs.h"

int main ()
{
    int flag;

    alt_putstr("Led and switch test\n");

    while (1)
    {
        flag = IORD_ALTERA_AVALON_PIO_DATA(SWITCHES_BASE);
        IOWR_ALTERA_AVALON_PIO_DATA(LED0_BASE, flag &
0xff);
    }
}
```

Figure 3 : code for first exercise

- Displaying on HEX0 the hexadecimal value from 0 to F. The value is set according to the position of the switches and their corresponding binary value.

```

matteoisgoodlooking.c  altera_avalon_pio_regs.h
1  #include "system.h"
2  #include "sys/alt_stdio.h"
3  #include "altera_avalon_pio_regs.h"
4
5  int main ()
6  {
7      int flag=0;
8      alt_putstr("Led and switch test\n");
9      while (1)
10     {
11         flag = IORD_ALTERA_AVALON_PIO_DATA(SWITCHES_BASE);
12         int seg=(flag & 0x0f);
13         switch (seg)
14         {
15             case 0:
16                 flag=0xc0;
17                 break;
18             case 1:
19                 flag=0xf9;
20                 break;
21             case 2:
22                 flag=0xa4;
23                 break;
24             case 3:
25                 flag=0xb0;
26                 break;
27             case 4:
28                 flag=0x99;
29                 break;
30             case 5:
31                 flag=0x92;
32                 break;
33             case 6:
34                 flag=0x82;
35                 break;
36             case 7:
37                 flag=0xf8;
38                 break;
39             case 8:
40                 flag=0x80;
41                 break;
42             case 9:
43                 flag=0x90;
44                 break;
45             case 10:
46                 flag=0x88;
47                 break;
48             case 11:
49                 flag=0x83;
50                 break;
51             case 12:
52

```

Figure 4 : first part of the code of the second exercise

```

51     break;
52 case 12:
53     flag=0xc6;
54     break;
55 case 13:
56     flag=0xa1;
57     break;
58 case 14:
59     flag=0x86;
60     break;
61 case 15:
62     flag=0x8e;
63     break;
64 }
65 IOWR_ALTERA_AVALON_PIO_DATA(HEX0_BASE, flag);
66 IOWR_ALTERA_AVALON_PIO_DATA(HEX1_BASE, 0xff);
67 }
68 }
69

```

Figure 5 : Second part of the code of the second exercise

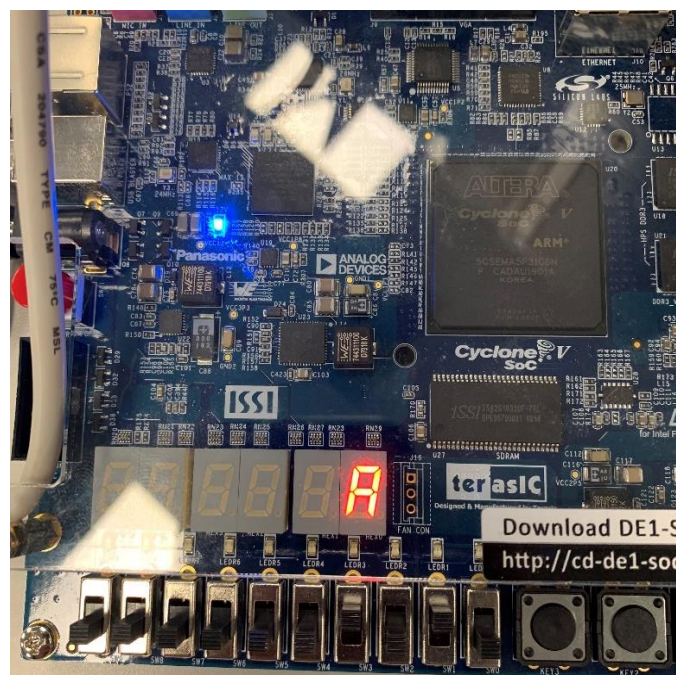
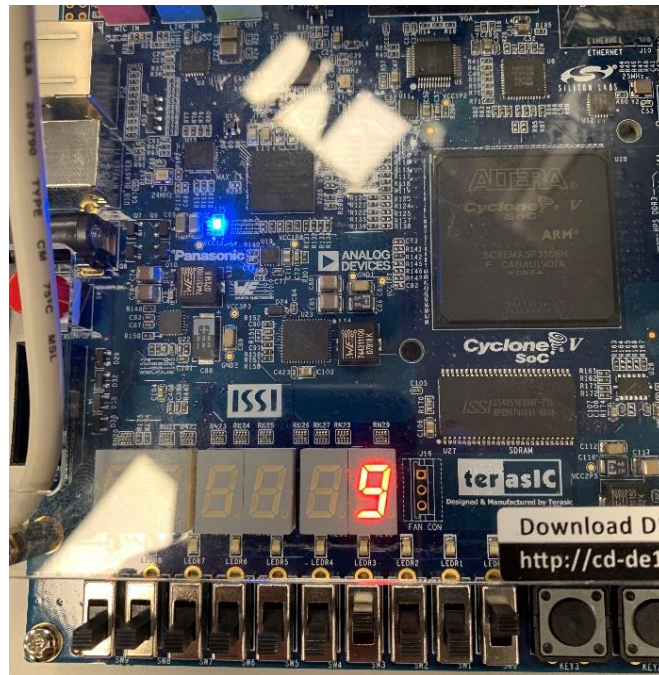
We keep the switches\_base as an input for the reading peripheral and we set HEX0 and HEX1 as outputs. First, we put a mask on the last 4 switches we don't use (& 0x0f) because to count to F we only need 4 bits.

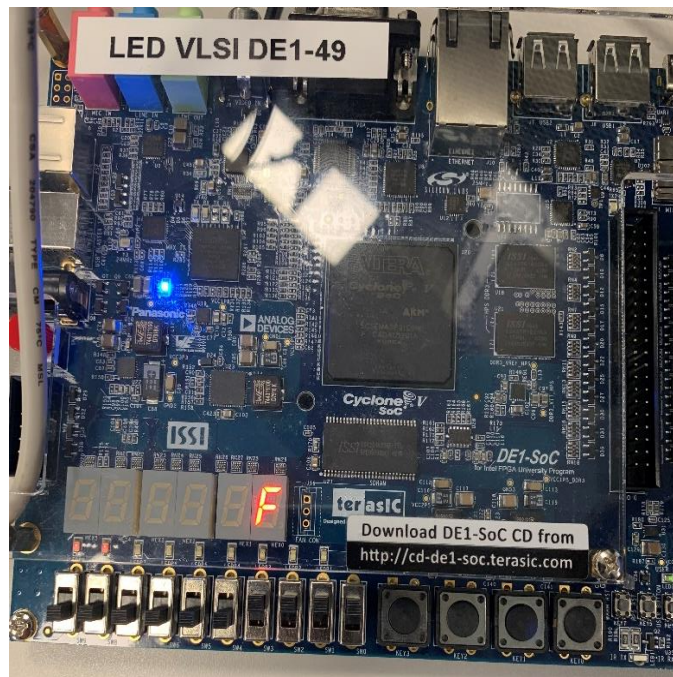
Then, we created a switch case which act as a decoder so we put every possible pattern in 4 bits to count from 0 to F. As a 0 turns the segment on, to display a 7 we need to have 11111000 which corresponds to F8 in hexadecimal. We did the same for every hexadecimal value.

Finally, we put a mask on HEX1 by setting all his segments to 1, meaning turned off.

The following pictures shows the effective functioning of our code :









- Counting the rising transitions of the key(1) button.

The first thing we did for this problem was making the previous hexa to 7seg display decoder into a c function. We kept the switch cases and the function is the exact same configuration (with the masking on the 4 first bits of the switches input) . As depicted on figure 6, the function takes a 4bits word and return an 8bits one (which we coded on hexadecimal)

```

50 //decoding 7 seg function
51 int display(int cbool)//takes a four bit, return an 8bit
52 {
53     int flag=0;
54     switch(cbool)
55     {
56     case 0:
57         flag=0xc0;
58         break;

```

Figure 6 : the decoder (is longer than 8 lines of course).

Let's base our comments with the figure 7 being the code of the first counter we've sent which, as we'll explain, had some bugs.

We must still use a polling method by using the while(1) loop. We use the reading function of altera to get the keys buttons states on the flag integer. From line 17 to 21 we explain the idea on how to not overflow the counter each time we press the button. Since the clock frequency is really high compared to human consciousness. Pushing the button even 0.25seconds would increase the counter by 12.5 millions with the 50 MHz clock of the system. As we only want to increase it by one for each pushes, we had to implement the "state" boolean. (which is an integer as we code on C) The coding of the idea is done from line 26 to line 38.

Next to that we have the reset by the key(2) which as to be coded. Our first idea as beginners was to mask ou the only bit we want from the flag reading which was the 2<sup>nd</sup> one from the keys reading. It was not working since on line 39, we code the two conditions for resetting the counter as either the reset key is pressed or the counter as gone beyond the readable bits and must comme back to 0. This overflow isn't actually a problem since we work with an integer on 32 bits and its 8 first bits would behave the same way wether the counter value is 12 or 563 but we dislike having bits that are wandering around.

Line 25 is masking the key(1) information. Same reason as the previous exercise. When sending te integer to the decoder function, if we don't know the value of the others bits of the flag integer, we can't know for sure the boolean keyup would be exactly equal to 1 or 0 without the masking since it's c language.

Line 42 and 43 is initialising integer with the value from the decoded 4 by 4 bits of the counter. (we could have and we should have given others names than "flag" and "flag2" for comprehension reading, still, it is what it is). Since we must send a 4bits word to the display function, we mask the counter 4 first bits. For the bits 4 to 7 of the counter, we had to mask them like before but also shift the bits in order to not send a logic "0000" everytime to the decoding function. Afterward, we let the Altera write function does it's wonder on the seven segments displays. This time we don't want to hide the 2<sup>nd</sup> displayer.

```

1  #include "system.h"
2  #include "sys/alt_stdio.h"
3  #include "altera_avalon_pio_regs.h"
4
5  int main ()
6  {
7      int flag=0;
8      alt_putstr("Led and switch test\n");
9      int state=0; //to have the state of the key1. Used as boolean
10     while (1)
11     {
12         while (1)
13         {
14             flag = IORD_ALTERA_AVALON_PIO_DATA(KEYS_BASE);
15             /*
16              * either the state is 0 and button is pushing for first time
17              * then we give 1 to state and increment counter
18              * or state is already 1 meaning button was already pushed
19              * then we dont increment and keep state to 1
20              * if button is not pushed and state is 1 we must give 0 back to state
21              */
22
23             int keyreset = flag & 0x02;
24             int keyup = flag & 0x01;
25             if(state == 0 & keyup == 1) //button was not pushed
26             {
27                 state = 1;
28                 cnt++;
29             }
30             else if(state==1 & keyup==1) //button is still pushed but not on rising
31             {
32                 state = 1;
33             }
34             else if(state==1 & keyup==0) //button is not pushed anymore
35             {
36                 state = 0;
37             }
38             if(keyreset==1 | cnt==0x100)
39                 cnt = 0;
40
41             flag = display(cnt & 0x0f);
42             flag2=display((cnt & 0xf0)>>4);
43             IOWR_ALTERA_AVALON_PIO_DATA(HEX0_BASE, flag);
44             IOWR_ALTERA_AVALON_PIO_DATA(HEX1_BASE, flag2);
45         }
46     }
47 }

```

Figure 7 : coding of a bugged counter.

Problems were: The reset from key(2) was not working and our counter started at decimal value "1". We could increment the displayer and it's worked well on both the 7seg, however when using the system reset with key(0), the counter would never come back to 0 but still began on 1.

This last issue (from what we understood of what Mister Valpreda explained us) is due from a bug of the machine which send a 1 on and from all peripherals after a reset. Thus, since we are not capable of overthrowing this problem, we must embrace it. As for the reset issue, we realised that without a proper shifting of the keyreset integer, it would never be equal to  $1_2$  or  $0_2$  but  $10_2$  or  $00_2$ .

We can see on figure 8 the code of the working counter and displayer.

```

1  #include "system.h"
2  #include "sys/alt_stdio.h"
3  #include "altera_avalon_pio_regs.h"
4
5  int main ()
6  {
7      int flag=0;
8      int cnt=256-1;
9      alt_putstr("Led and switch test\n");
10     int state=0;//to have the state of the key. Used as boolean
11     while (1)
12     {
13         flag = IORD_ALTERA_AVALON_PIO_DATA(KEYS_BASE);
14
15         /* either the state is 0 and button is pushing for first time
16          * then we give 1 to state and increment counter
17          * or state is already 1 meaning button was already pushed
18          * then we don't increment and keep state to 1
19          * if button is not pushed and state is 1 we must give 0 back to state */
20
21         int keyreset = (flag & 0b10)>>1;//10>>01
22         int keyup = flag & 0b01;
23
24         if((keyreset==0) || (cnt>=0x100))
25         {
26             cnt = 0;
27         }
28         else if(state == 0 && keyup == 1)//button was not pushed
29         {
30             state = 1;
31             cnt++;
32         }
33         else if(state==1 && keyup==1)//button is still pushed but not on rising
34         {
35             state = 1;
36         }
37         else if(state==1 && keyup==0)//button is not pushed anymore
38         {
39             state = 0;
40         }
41
42         int disphex0 = display(cnt & 0x0f);
43         int disphex1 = display((cnt & 0xf0)>>4);
44         IOWR_ALTERA_AVALON_PIO_DATA(HEX0_BASE, disphex0);
45         IOWR_ALTERA_AVALON_PIO_DATA(HEX1_BASE, disphex1);
46     }
47 }

```

Figure 8 : Working code

Initializing the counter at  $256_{10}-1$  which is equal to  $0x0100_{16}-1$  will make it go to  $0x0100$  at the start and after the reset of the code. Therefore, the resetting of the counter at line 38 will always gives the La of the counting operations.

Following pictures are the proofs of working

