# Digital Electronics

# Laboratory 4: Building and programming a simple NIOS system

Group <15>:

Vacchetto Edoardo 223218

Aga Andi 281743

Zhang Zhifan 287876

Tong Lin 287649

15/12/2020

# Index

## Part 1) Configuration of the Platform Designer (QSYS/SOPC) tool

### 1.1 Description

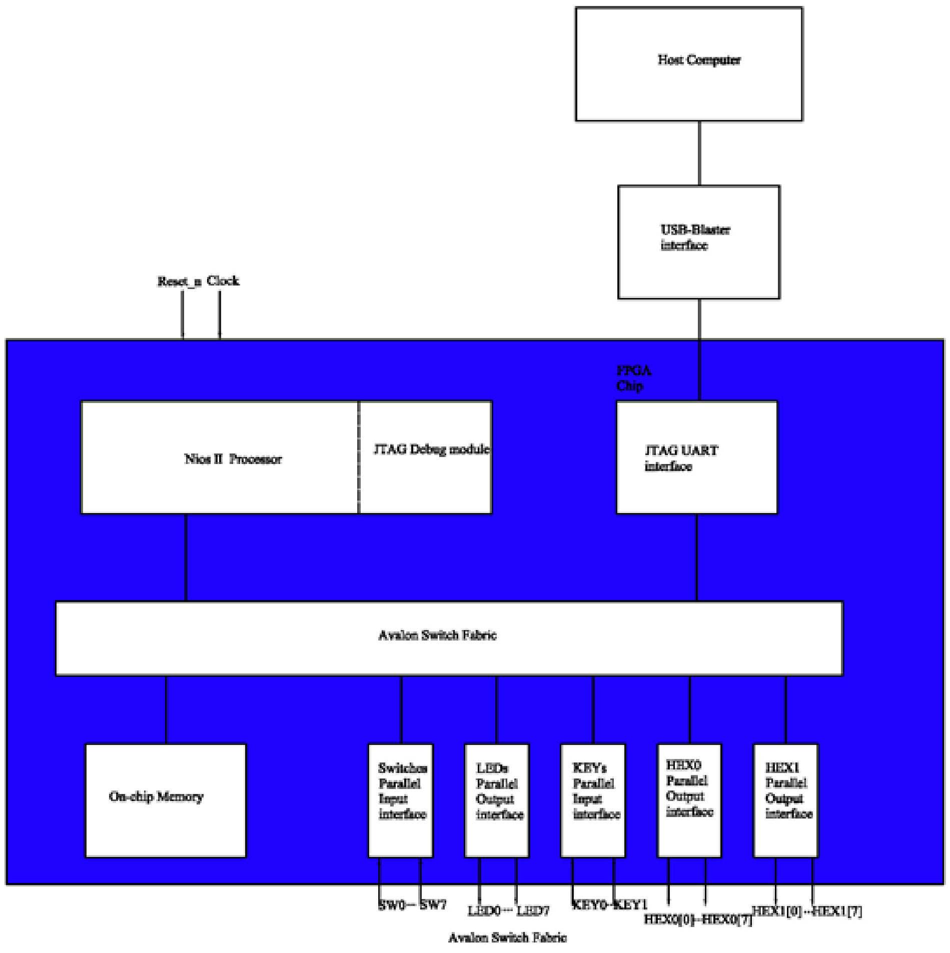
During this part, a NIOS processor system has to be implemented. This implementation has been performed using the Platform Designer(QSYS/SOPC) Tool of Quartus Altera. This processor system will be named *“lab2_system.qsys”*.

### 1.2 Discussion of option(s) considered for implementation

The processor system is made of:

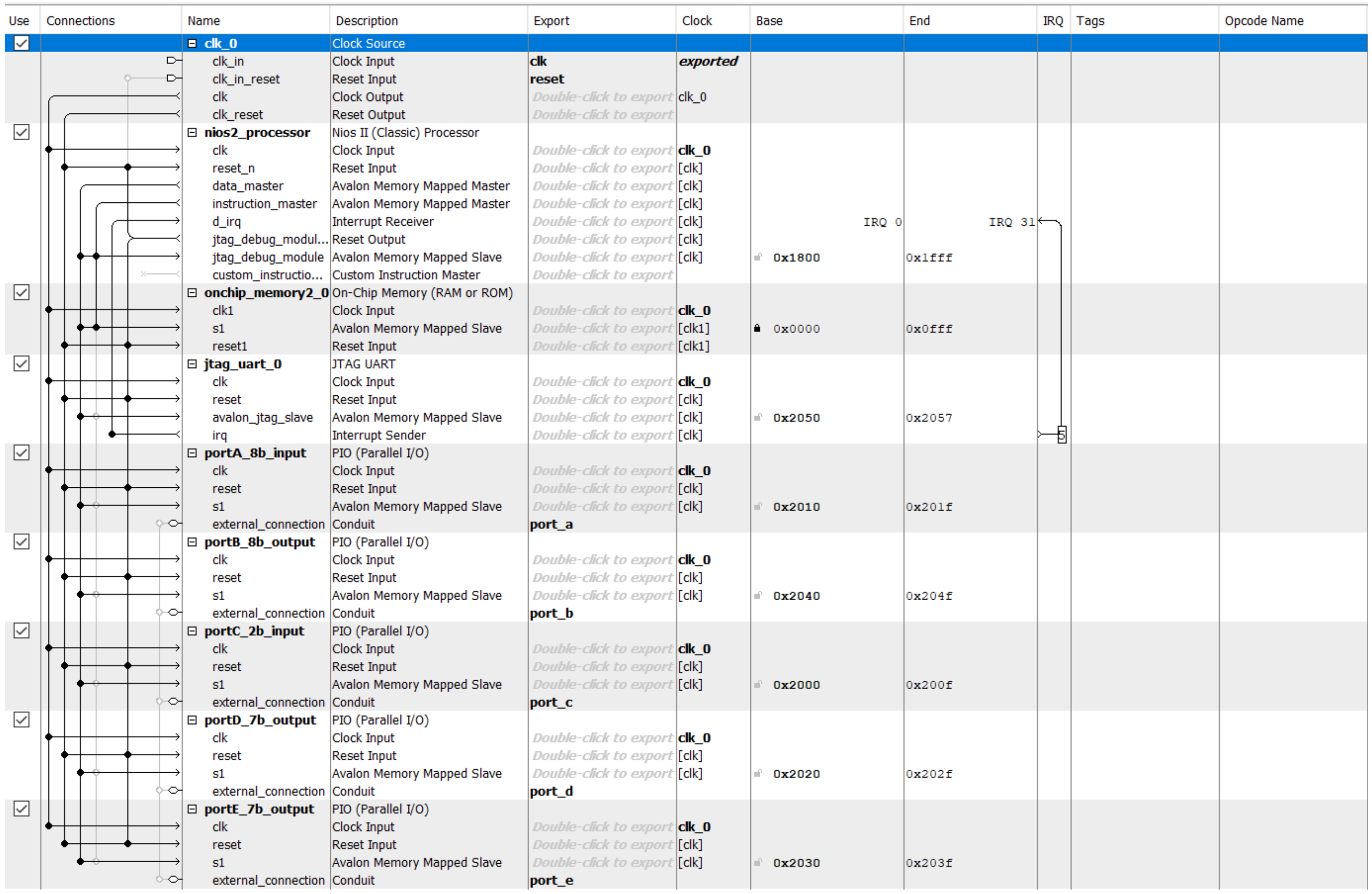
- one NIOS II/e processor
- a 4 kB one chip memory
- one 2-bit input parallel port
- one 8-bit output parallel port
- one 8-bit input parallel port
- two 7-bit output parallel port
- a JTAG UART

o The Hardware design by Qsys tool is based on the sketch showing below. A great use of the tutorial provided on the assignment has been made to implement the system.



### 1.3 Results and Conclusions

The following figure shows the resulting configuration.



| Use | Connections | Name | Description | Export | Clock | Base | End | IRQ | Tags | Opcode Name |
|---|---|---|---|---|---|---|---|---|---|---|
| ✓ | | ⊟ **clk_0** | Clock Source | | | | | | | |
| | | 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* | | | | | | |
| ✓ | | ⊟ **nios2_processor** | Nios II (Classic) Processor | | | | | | | |
| | | clk | Clock Input | *Double-click to export* | **clk_0** | | | | | |
| | | reset_n | Reset Input | *Double-click to export* | [clk] | | | | | |
| | | data_master | Avalon Memory Mapped Master | *Double-click to export* | [clk] | | | | | |
| | | instruction_master | Avalon Memory Mapped Master | *Double-click to export* | [clk] | | | | | |
| | | d_irq | Interrupt Receiver | *Double-click to export* | [clk] | IRQ 0 | IRQ 31 | | | |
| | | jtag_debug_modul... | Reset Output | *Double-click to export* | [clk] | | | | | |
| | | jtag_debug_module | Avalon Memory Mapped Slave | *Double-click to export* | [clk] | 0x1800 | 0x1fff | | | |
| | | custom_instructio... | Custom Instruction Master | *Double-click to export* | | | | | | |
| ✓ | | ⊟ **onchip_memory2_0** | On-Chip Memory (RAM or ROM) | | | | | | | |
| | | clk1 | Clock Input | *Double-click to export* | **clk_0** | | | | | |
| | | s1 | Avalon Memory Mapped Slave | *Double-click to export* | [clk1] | 🔒 0x0000 | 0x0fff | | | |
| | | reset1 | Reset Input | *Double-click to export* | [clk1] | | | | | |
| ✓ | | ⊟ **jtag_uart_0** | JTAG UART | | | | | | | |
| | | clk | Clock Input | *Double-click to export* | **clk_0** | | | | | |
| | | reset | Reset Input | *Double-click to export* | [clk] | | | | | |
| | | avalon_jtag_slave | Avalon Memory Mapped Slave | *Double-click to export* | [clk] | 0x2050 | 0x2057 | | | |
| | | irq | Interrupt Sender | *Double-click to export* | [clk] | | | 0 | | |
| ✓ | | ⊟ **portA_8b_input** | PIO (Parallel I/O) | | | | | | | |
| | | clk | Clock Input | *Double-click to export* | **clk_0** | | | | | |
| | | reset | Reset Input | *Double-click to export* | [clk] | | | | | |
| | | s1 | Avalon Memory Mapped Slave | *Double-click to export* | [clk] | 0x2010 | 0x201f | | | |
| | | external_connection | Conduit | **port_a** | | | | | | |
| ✓ | | ⊟ **portB_8b_output** | PIO (Parallel I/O) | | | | | | | |
| | | clk | Clock Input | *Double-click to export* | **clk_0** | | | | | |
| | | reset | Reset Input | *Double-click to export* | [clk] | | | | | |
| | | s1 | Avalon Memory Mapped Slave | *Double-click to export* | [clk] | 0x2040 | 0x204f | | | |
| | | external_connection | Conduit | **port_b** | | | | | | |
| ✓ | | ⊟ **portC_2b_input** | PIO (Parallel I/O) | | | | | | | |
| | | clk | Clock Input | *Double-click to export* | **clk_0** | | | | | |
| | | reset | Reset Input | *Double-click to export* | [clk] | | | | | |
| | | s1 | Avalon Memory Mapped Slave | *Double-click to export* | [clk] | 0x2000 | 0x200f | | | |
| | | external_connection | Conduit | **port_c** | | | | | | |
| ✓ | | ⊟ **portD_7b_output** | PIO (Parallel I/O) | | | | | | | |
| | | clk | Clock Input | *Double-click to export* | **clk_0** | | | | | |
| | | reset | Reset Input | *Double-click to export* | [clk] | | | | | |
| | | s1 | Avalon Memory Mapped Slave | *Double-click to export* | [clk] | 0x2020 | 0x202f | | | |
| | | external_connection | Conduit | **port_d** | | | | | | |
| ✓ | | ⊟ **portE_7b_output** | PIO (Parallel I/O) | | | | | | | |
| | | clk | Clock Input | *Double-click to export* | **clk_0** | | | | | |
| | | reset | Reset Input | *Double-click to export* | [clk] | | | | | |
| | | s1 | Avalon Memory Mapped Slave | *Double-click to export* | [clk] | 0x2030 | 0x203f | | | |
| | | external_connection | Conduit | **port_e** | | | | | | |

*Figure 1:The ports connection*

For simplicity, the PIO ports have been named based on their number of bits. For example, portA_8b_input is an input port of 8 bits. Moreover, the on-chip memory has been assigned to have a base address of zeros, whereas the other components' addresses have been assigned automatically. After this, the HDL can be generated and used in the coming steps of the project.

## Part 2) Quartus Project

### 1.1 Description

A Quartus project has to be implemented, in which the code generated in the previous point will be involved. More in detail, the component lab2_processor will be used in the general project.

### 1.2 Discussion of option(s) considered for implementation

The implementation is very straightforward and there are not many details. It can be noted that the I/O ports of the project are named corresponding to the "DE1_SoC.qsf" file, so an import of that file is needed to automatically assign the pins. On the other hand, using the *TimeQuest Timing Analyzer Tool*, a 50MHz clock constrain is generated for the system operation.

### 1.3 Discussion of VHDL code

The main VHDL code is shown below:

```
--  CLOCK_50 => processor clock
--  KEY(0) => processor reset
--  KEY(2 downto 1) => processor 2 bit input port
--  LEDR => processor 8 bit output port
--  HEX0 => processor 7 bit output port
--  HEX1 => processor 7 bit output port
--  SW => processor 8 bit input

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity lab2_system is
    port(CLOCK_50 : in std_logic;
          KEY : in std_logic_vector(3 downto 0);
          SW : in std_logic_vector(7 downto 0);    -- only 3 used
          HEX0 : out std_logic_vector(6 downto 0);
          HEX1 : out std_logic_vector(6 downto 0);
          LEDR : out std_logic_vector(7 downto 0));
end lab2_system;


Architecture behaviour of lab2_system is

    component lab2_processor is
        port (
            clk_clk       : in  std_logic                    := 'X';             -- clk
            port_a_export : in  std_logic_vector(7 downto 0) := (others => 'X'); -
- export
            port_b_export : out std_logic_vector(7 downto 0);                    -
- export
            port_c_export : in  std_logic_vector(1 downto 0) := (others => 'X'); -
- export
            port_d_export : out std_logic_vector(6 downto 0);                    -
- export
            port_e_export : out std_logic_vector(6 downto 0);                    -
- export
            reset_reset_n : in  std_logic                    := 'X'              -
- reset_n
        );
    end component lab2_processor;

Begin

SYSTEM: lab2_processor
                port map(clk_clk => CLOCK_50,
                        reset_reset_n => KEY(0),
                        port_a_export => SW,
                        port_b_export => LEDR,
                        port_c_export => KEY(2 downto 1),
                        port_d_export => HEX0,
                        port_e_export => HEX1
                        );

end behaviour;
```

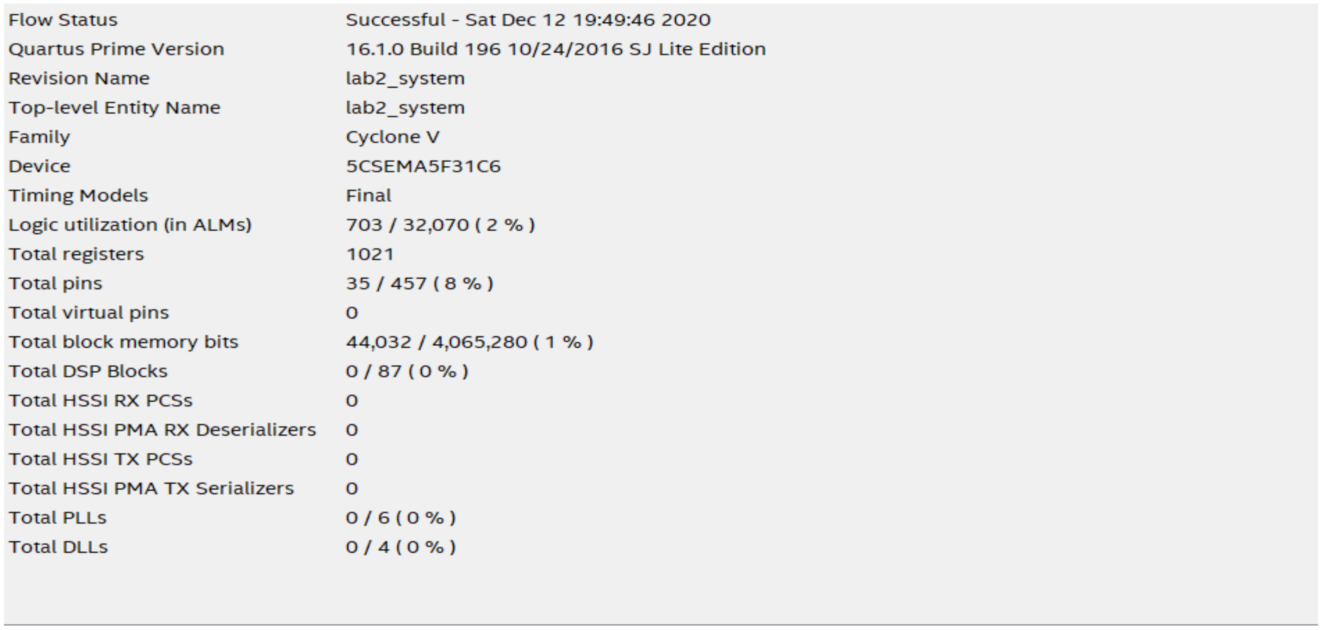
As it can be seen from the code, the I/O ports have been defined and then, also the lab2_processor component has been called. A suitable port mapping, according to the requirements, is then performed.

The component lab2_processor is taken from the autogenerated file called “lab2_processor_inst.vhd”, whose code is shown below:

```
    component lab2_processor is
        port (
            clk_clk       : in  std_logic                    := 'X';             -- clk
            port_a_export : in  std_logic_vector(7 downto 0) := (others => 'X'); -
- export
            port_b_export : out std_logic_vector(7 downto 0);                    -
- export
            port_c_export : in  std_logic_vector(1 downto 0) := (others => 'X'); -
- export
            port_d_export : out std_logic_vector(6 downto 0);                    -
- export
            port_e_export : out std_logic_vector(6 downto 0);                    -
- export
            reset_reset_n : in  std_logic                    := 'X'              -
- reset_n
        );
    end component lab2_processor;

    u0 : component lab2_processor
        port map (
            clk_clk       => CONNECTED_TO_clk_clk,       --    clk.clk
            port_a_export => CONNECTED_TO_port_a_export, -- port_a.export
            port_b_export => CONNECTED_TO_port_b_export, -- port_b.export
            port_c_export => CONNECTED_TO_port_c_export, -- port_c.export
            port_d_export => CONNECTED_TO_port_d_export, -- port_d.export
            port_e_export => CONNECTED_TO_port_e_export, -- port_e.export
            reset_reset_n => CONNECTED_TO_reset_reset_n  --  reset.reset_n
        );
```

### 1.4 Results and Conclusions

The project is compiled successfully:



```
Flow Status                     Successful - Sat Dec 12 19:49:46 2020
Quartus Prime Version           16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name                   lab2_system
Top-level Entity Name           lab2_system
Family                          Cyclone V
Device                          5CSEMA5F31C6
Timing Models                   Final
Logic utilization (in ALMs)     703 / 32,070 ( 2 % )
Total registers                 1021
Total pins                      35 / 457 ( 8 % )
Total virtual pins              0
Total block memory bits         44,032 / 4,065,280 ( 1 % )
Total DSP Blocks                0 / 87 ( 0 % )
Total HSSI RX PCSs              0
Total HSSI PMA RX Deserializers 0
Total HSSI TX PCSs              0
Total HSSI PMA TX Serializers   0
Total PLLs                      0 / 6 ( 0 % )
Total DLLs                      0 / 4 ( 0 % )
```

*Figure 2: The result of compilation*

## Part 3) Programming the system

### 2.1 Description

During this part, some programming has to be performed. There are three different programs, which have to do with the switches, LED displays, pushbuttons and the 7-segment displays. The programs consist of:

- connecting the switches to the LED displays.
- encoding the switches numbered SW(3), SW(2), SW(1) and SW(0) into the 7-segment display HEX0. The switches form a 4-bit binary number which has to be displayed in the 7-segment display using its corresponding hexadecimal representation.
- counting the rising transitions of the signal coming from the pushbutton KEY(1), with maximum count equal to 255 and then displaying the count as a hexadecimal into the 7-segment displays HEX1 and HEX0. When the pushbutton KEY(2) is pressed, the counter should be reset.

It should be noted that the programming should be done both with and without HAL.

## 2.2 Discussion of option(s) considered for implementation

As for the stimulation using CPUlator (without HAL), the process is pretty straightforward. The first thing to do is to define some pointers to the addresses of the devices.

For the first program, a single operation is enough to connect the switches to the LED displays.

On the other hand, in the second program a kind of look up table has to be created, mapping the values of the switches to the value which has to be displayed on the 7-segment display HEX0. It should be noted each bit of a 7-bit number, corresponds to one of the segments of the 7-segment display. The LSB corresponds to segment ‘0’, whereas the MSB - 1 corresponds to segment ‘6’ as shown in the figure below. Moreover, a segment is on when its corresponding bit is set to a high logic value.

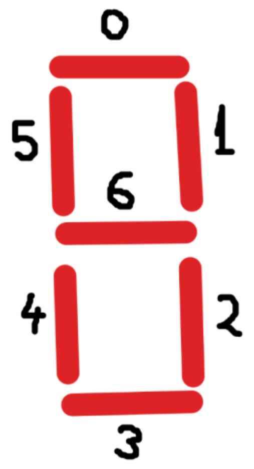


*Figure 3: The sketch map of HEX0*

The following table shows the mappings and the values which have to be given to 7-segment display HEX0:

| SW(3) | SW(2) | SW(1) | SW(0) | HEX | Value to give to HEX0 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | **0** | 00111111 = 0x3F |
| 0 | 0 | 0 | 1 | **1** | 00000110 = 0x06 |
| 0 | 0 | 1 | 0 | **2** | 01011011 = 0x5B |
| 0 | 0 | 1 | 1 | **3** | 01001111 = 0x4F |
| 0 | 1 | 0 | 0 | **4** | 01100110 = 0x66 |
| 0 | 1 | 0 | 1 | **5** | 01101101 = 0x6D |
| 0 | 1 | 1 | 0 | **6** | 01111101 = 0x7D |
| 0 | 1 | 1 | 1 | **7** | 00000110 = 0x07 |
| 1 | 0 | 0 | 1 | **8** | 01111111 = 0x7F |

|  |  |  |  |  |  |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | **9** | 01101111 = 0x6F |
| 1 | 0 | 1 | 0 | **A** | 01110111 = 0x77 |
| 1 | 0 | 1 | 1 | **B** | 01111100 = 0x7C |
| 1 | 1 | 0 | 0 | **C** | 00111001 = 0x58 |
| 1 | 1 | 0 | 1 | **D** | 01011110 = 0x5E |
| 1 | 1 | 1 | 0 | **E** | 01111001 = 0x79 |
| 1 | 1 | 1 | 1 | **F** | 01110001 = 0x71 |

The final hexadecimal values to give to the 7-segment display have been stored in vector “dec_lut”, whose index corresponds to the decimal value we want to represent.

For the third program, a counter should be initialized, which increments when KEY(1) is on and resets to 0 when KEY(2) is on.  The counter was easily implemented using a variable that is incremented on the rising edge of KEY(1) and reset on the rising edge of KEY(2).

The detection of the rising edge was done using two variables, one for storing the past state and one for the present state. So, when the past state is logic 0 and the present state is logic 1, a rising edge occurs.

Since the counter can go up to 255, it has to be represented with two 7-segment displays. To represent the value of the counter correctly, the variable *cnt* has been divided into two-part. The 4 lower bits drive HEX0 while the upper 4 bits drive HEX1. The 4-bit binary value are decoded thanks to the lookup table (dec_lut) to provide the right values to the hexadecimal displays.

Since the NIOS II processor uses memory-mapped peripherals, reading the value of a port or writing the value to a port is the same as reading or writing data in a variable. For this reason, switches, LEDs and hexadecimal displays are considered as pointers, containing the base address of the corresponding port.

The two codes implemented are completely equivalent. In fact, if in the code used with CPUlator pointers are used to read/write the memory areas dedicated to GPIOs, in the second code the Hal libraries are used to abstract the interaction with the memory, making it an even simpler operation.

## 2.3 Discussion of code

**The CPUlator code is shown below:**

```
// inputs
volatile unsigned int *SW = (unsigned int*) 0xff200040;
volatile unsigned int *KEY = (unsigned int*) 0xff200050;

// outputs
```

```c
volatile unsigned int *LED = (unsigned int*) 0xff200000;
volatile unsigned int *HEX0 = (unsigned int*) 0xff200020;

// data
 char cnt = 0;
 unsigned int dec_lut[16];
 unsigned int sw_x_hex = 0, sw_x_hex2 = 0;
// consumable
long int k,i, prev_state, prev_state1;

int main()
{
    // decoder lut initialization
    dec_lut[0] =  0x3F; // 0
    dec_lut[1] =  0x06; // 1
    dec_lut[2] =  0x5B; // 2
    dec_lut[3] =  0x4F; // 3
    dec_lut[4] =  0x66; // 4
    dec_lut[5] =  0x6D; // 5
    dec_lut[6] =  0x7D; // 6
    dec_lut[7] =  0x07; // 7
    dec_lut[8] =  0x7F; // 8
    dec_lut[9] =  0x6F; // 9
    dec_lut[10] =  0x77; // A
    dec_lut[11] =  0x7C; // B
    dec_lut[12] =  0x58; // C
    dec_lut[13] =  0x5E; // D
    dec_lut[14] =  0x79; // E
    dec_lut[15] =  0x71; // F


    while(1)
    {
        unsigned int key1 = (*KEY & 0x02) >> 1;
        unsigned int key2 = (*KEY & 0x04) >> 2;

        //           -----------> Reset Counter <-------------
        if(key2)
        {
            if(!prev_state1)
                cnt = 0;
        }
        prev_state1 = key2;

        //         -----------> Key1 rising end detection and counter  updating <------------
---
        if( key1 )
        {
            if(!prev_state)
                cnt++;
        }
        prev_state = key1;

        //                 ------------>  HEX display driven by the switches  <--------------
----
        /*sw_x_hex = ((*SW & 0x08) >> 3) << 3 | ((*SW & 0x04) >> 2) << 2 |
                     ((*SW & 0x02) >> 1) << 1 | (*SW & 0x01);
        sw_x_hex2 = ((*SW & 0x80) >> 7) << 3 | ((*SW & 0x40) >> 6) << 2 |
                    ((*SW & 0x20) >> 5) << 1 | (*SW & 0x10) >> 4;*/

        //                 ------------> HEX display driven by the counter <-----------------
--
        sw_x_hex = ((cnt & 0x08) >> 3) << 3 | ((cnt & 0x04) >> 2) << 2 |
                    ((cnt & 0x02) >> 1) << 1 | (cnt & 0x01);
        sw_x_hex2 = ((cnt & 0x80) >> 7) << 3 | ((cnt & 0x40) >> 6) << 2 |
                    ((cnt & 0x20) >> 5) << 1 | (cnt & 0x10) >> 4;

        *HEX0 = 0x3F << 24 | 0x3F << 16| dec_lut[sw_x_hex2] << 8 | dec_lut[sw_x_hex];

        // connect the switch to the led
        *LED = *SW;

    }
}
```

This code is used to accomplish the function of counting the rising edge of the signal by clicking the pushbutton KEY(1) and reset by pushbutton KEY(2). Moreover, it displays the counting result on HEX(1) and HEX(0). When KEY(1) is not equal to the previous state, the count number increments, whereas when KEY(2) is not equal to the previous state, then counting number will be reset to 0. As the maximum counting number is 255, its binary representation is of a maximum 8 bits. The lower 4 bits and higher 4 bits are separately extracted, and the vector “dec_lut” can be used to convert them into hexadecimal numbers to display on the 7-segment displays HEX(1) and HEX(0). Then, those displays can also be combined with two 0s on the HEX(3) and HEX(2) and shown together as the final display result. Here two 0s on HEX(3) and HEX(2) must be also included, since HEX(0), HEX(1) HEX(2) and HEX(3) are all linked to the same base address. For this reason, to avoid the instability or errors of HEX(3) and HEX(2), they are set to display 0s.

Moreover, the function of connecting the Switches and LEDs is realized in this code.

The code commented with the title of “HEX display driven by the switches” can realize the hexadecimal number display from ‘0’ to ‘F’ on the 7-segment display by using the switches SW(0) to SW(3), with the same method to transfer the acts of switches to the display of HEX(1) and HEX(0) as the method used in the HEX display driven by the counter function. However, only one of those functions can be done at a time since they are written in the same code, so one of them must be masked when performing the other.

**CODE with HAL:**

```
/*
 *    - Use ALT versions of stdio routines:
 *
 *          Function                  Description
 *        ===============  ======================================
 *        alt_printf       Only supports %s, %x, and %c ( < 1 Kbyte)
 *        alt_putstr       Smaller overhead than puts with direct drivers
 *                         Note this function doesn't add a newline.
 *        alt_putchar      Smaller overhead than putchar with direct drivers
 *        alt_getchar      Smaller overhead than getchar with direct drivers
 *
 */

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

unsigned int SW_status, sw_x_hex, sw_x_hex2, hex_tot, key1, key2;
unsigned int key1_prev, key2_prev, cnt;

unsigned int dec_lut[16];

int main()
{
    //   ----------------------> LUT Initialization <----------------------
    dec_lut[0] =  0x3F; // 0
    dec_lut[1] =  0x06; // 1
    dec_lut[2] =  0x5B; // 2
    dec_lut[3] =  0x4F; // 3
    dec_lut[4] =  0x66; // 4
    dec_lut[5] =  0x6D; // 5
    dec_lut[6] =  0x7D; // 6
    dec_lut[7] =  0x07; // 7
    dec_lut[8] =  0x7F; // 8
    dec_lut[9] =  0x6F; // 9
    dec_lut[10] =  0x77; // A
    dec_lut[11] =  0x7C; // B
    dec_lut[12] =  0x58; // C
    dec_lut[13] =  0x5E; // D
    dec_lut[14] =  0x79; // E
    dec_lut[15] =  0x71; // F
```

```c
  /* Event loop never exits. */
  while (1)
  {
      //   --------------------> Connect SW to LEDR <--------------------
      SW_status = IORD_ALTERA_AVALON_PIO_DATA(PORTA_8B_INPUT_BASE);  // read switch value
      IOWR_ALTERA_AVALON_PIO_DATA(PORTB_8B_OUTPUT_BASE, SW_status & 0xFF);  // put switches v
alue inside the led

      // ----------------------> LUT index evaluation for HEX0 and HEX1 <----------------
      sw_x_hex = ((SW_status & 0x08) >> 3) << 3 | ((SW_status & 0x04) >> 2) << 2 |
                    ((SW_status & 0x02) >> 1) << 1 | (SW_status & 0x01);
      sw_x_hex2 = ((SW_status & 0x80) >> 7) << 3 | ((SW_status & 0x40) >> 6) << 2 |
                    ((SW_status & 0x20) >> 5) << 1 | (SW_status & 0x10) >> 4;

      //   -----------> Connection from switches to hex display <------------------------
      IOWR_ALTERA_AVALON_PIO_DATA(PORTD_7B_OUTPUT_BASE, dec_lut[sw_x_hex] & 0xFF); // write d
ata to HEX0
      IOWR_ALTERA_AVALON_PIO_DATA(PORTE_7B_OUTPUT_BASE, dec_lut[sw_x_hex2] & 0xFF); // write
data to HEX1

      key1 = (IORD_ALTERA_AVALON_PIO_DATA(PORTC_2B_INPUT_BASE) & 0x01) >> 1;
      key2 = (IORD_ALTERA_AVALON_PIO_DATA(PORTC_2B_INPUT_BASE) & 0x02) >> 2;

      if(key1)   // key1 rising edge detection
      {
          if(!key1_prev)
              cnt++;
      }
      key1_prev = key1;

      if(key2)   // reset counter on rising edge of key2
      {
          if(!key2_prev)
              cnt = 0;
      }
      key2_prev = key2;
  }

  return 0;
}
```

The code above behaves in the same way as the code above. However, it has been written using the HAL libraries provided by the NIOS II software build tool. The use of this level of abstraction simplifies the use of and interaction with the previously designed system.

All the addresses of the peripherals in the system have been collected in the file "system.h" and associated to an easy-to-use mnemonic.

In this case, working exclusively with GPIO, two functions were used:

- IORD_ALTERA_AVALON_PIO_DATA to read the value of the inputs (push buttons, switches)
- IOWR_ALTERA_AVALON_PIO_DATA to write the desired value in the output registers of the GPIOs considered (LEDs, 7-segment display)

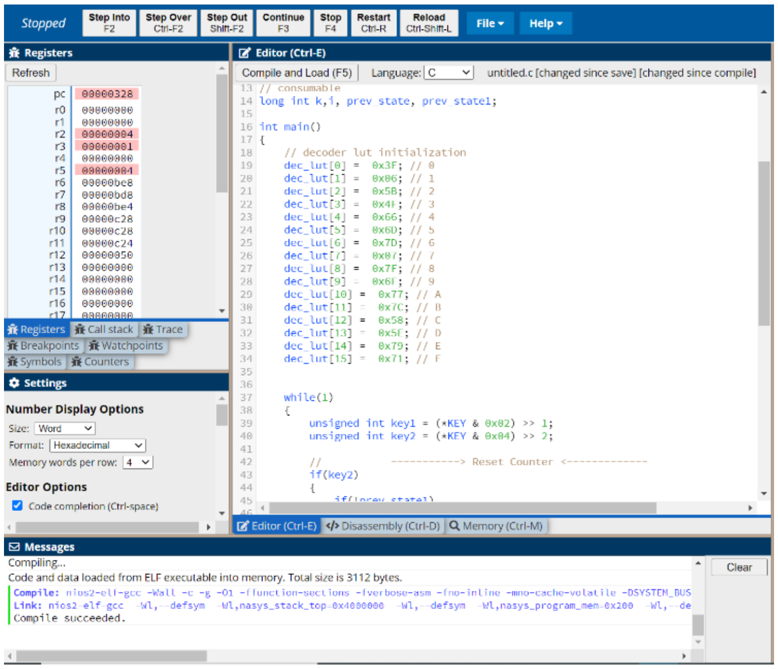
The starting addresses of the peripherals used are called, for example:

- PORTA_8B_INPUT_BASE, input port
- PORTB_8B_OUTPUT_BASE, output port

During the execution, the program first evaluates the switches or push buttons value, then creates some variables that will be used as an index for the lookup table useful for the generation of the outputs needed to control the 7-segment displays.

## 2.4 Simulation results

**The result of CPUlator is shown in the figure below:**



```
pc   00000328
r0   00000000
r1   00000000
r2   00000004
r3   00000001
r4   00000000
r5   00000004
r6   00000be8
r7   00000bd8
r8   00000be4
r9   00000c28
r10  00000c28
r11  00000c24
r12  00000050
r13  00000000
r14  00000000
r15  00000000
r16  00000000
r17  00000000
```

Number Display Options
Size: Word
Format: Hexadecimal
Memory words per row: 4

Editor Options
☑ Code completion (Ctrl-space)

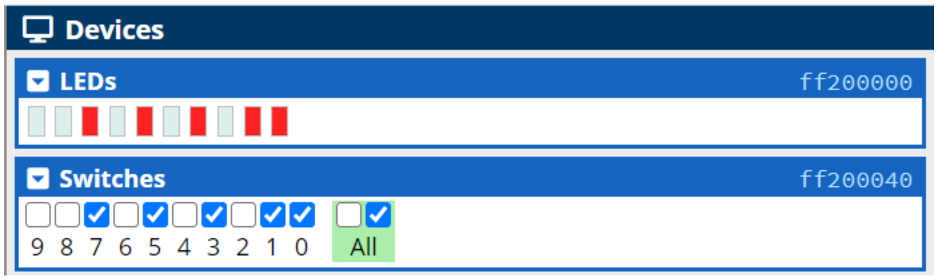
Language: C    untitled.c [changed since save] [changed since compile]

```c
// consumable
long int k,i, prev_state, prev_state1;

int main()
{
    // decoder lut initialization
    dec_lut[0] =  0x3F; // 0
    dec_lut[1] =  0x06; // 1
    dec_lut[2] =  0x5B; // 2
    dec_lut[3] =  0x4F; // 3
    dec_lut[4] =  0x66; // 4
    dec_lut[5] =  0x6D; // 5
    dec_lut[6] =  0x7D; // 6
    dec_lut[7] =  0x07; // 7
    dec_lut[8] =  0x7F; // 8
    dec_lut[9] =  0x6F; // 9
    dec_lut[10] =  0x77; // A
    dec_lut[11] =  0x7C; // B
    dec_lut[12] =  0x58; // C
    dec_lut[13] =  0x5E; // D
    dec_lut[14] =  0x79; // E
    dec_lut[15] =  0x71; // F


    while(1)
    {
        unsigned int key1 = (*KEY & 0x02) >> 1;
        unsigned int key2 = (*KEY & 0x04) >> 2;

        //          -----------> Reset Counter <------------
        if(key2)
        {
            if(!prev_state1)
```

```
Compiling...
Code and data loaded from ELF executable into memory. Total size is 3112 bytes.
Compile: nios2-elf-gcc -Wall -c -g -O1 -ffunction-sections -fverbose-asm -fno-inline -mno-cache-volatile -DSYSTEM_BUS
Link: nios2-elf-gcc  -Wl,--defsym  -Wl,nasys_stack_top=0x4000000  -Wl,--defsym  -Wl,nasys_program_mem=0x200  -Wl,--de
Compile succeeded.
```

*Figure: The result of compilation*

*Figure: The result of the connection of the SWs and LEDs*



The switches and the LEDs have been connected, when clicking the random number of switches, the related LEDs will be delighted.

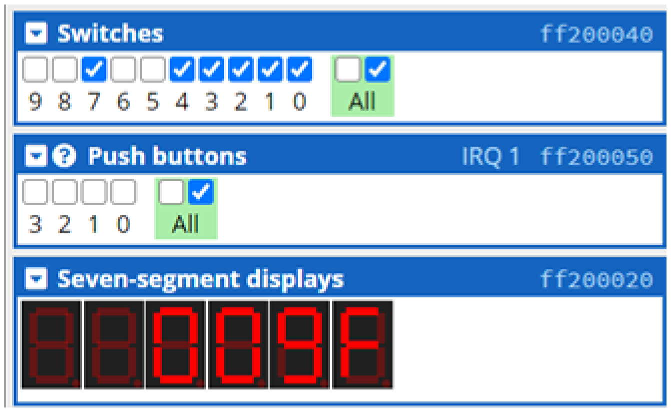


*Figure: correct encoding of the switches to the 7 segment displays*

The switches SW(3), SW(2), SW(1), SW(0) are encoded with the 7-segment display HEX0. When clicking the switches, they will be saved as the number in binary from '0000' to '1111', then the numbers are transferred into hexadecimal and displayed on the 7-segment display from '0' to 'F'. Using the same method HEX1 can also display the hexadecimal numbers driven by SW(7), SW(6), SW(5) and SW(4).

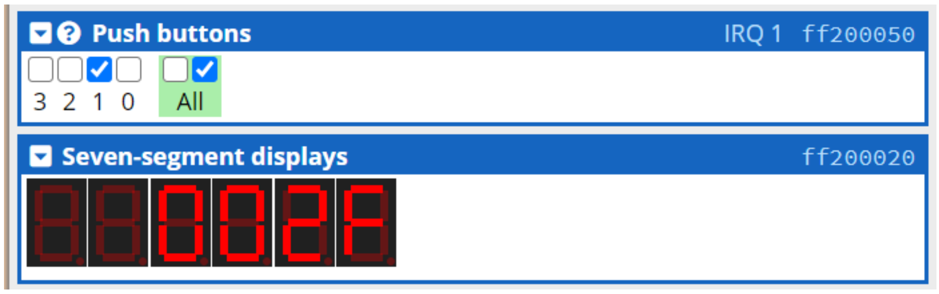
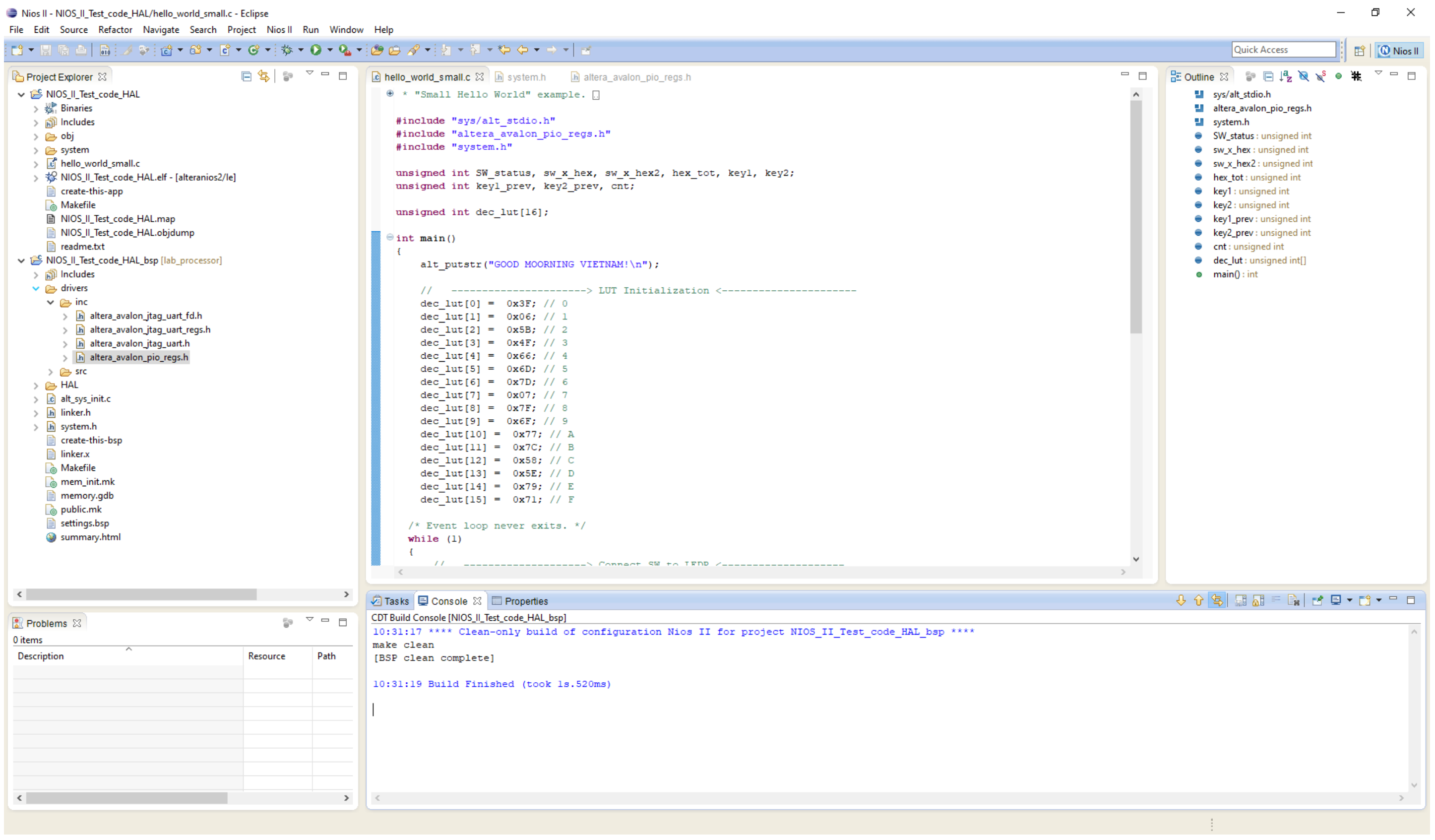


*Figure: Counting the rising transitions of the signal coming from pushbutton KEY(1)*

The pushing buttons KEY(1) and KEY(2) have been connected to the 7-segment display. When clicking KEY(1) the number of the display will be added, the maximum counting number is 255. When clicking KEY(2) the number displayed on the 7-segment displays will be reset.

**The result of the code with HAL is shown in the figure below:**



```c
 * "Small Hello World" example.

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

unsigned int SW_status, sw_x_hex, sw_x_hex2, hex_tot, key1, key2;
unsigned int key1_prev, key2_prev, cnt;

unsigned int dec_lut[16];

int main()
{
    alt_putstr("GOOD MOORNING VIETNAM!\n");

    //   ---------------------> LUT Initialization <---------------------
    dec_lut[0] =  0x3F; // 0
    dec_lut[1] =  0x06; // 1
    dec_lut[2] =  0x5B; // 2
    dec_lut[3] =  0x4F; // 3
    dec_lut[4] =  0x66; // 4
    dec_lut[5] =  0x6D; // 5
    dec_lut[6] =  0x7D; // 6
    dec_lut[7] =  0x07; // 7
    dec_lut[8] =  0x7F; // 8
    dec_lut[9] =  0x6F; // 9
    dec_lut[10] =  0x77; // A
    dec_lut[11] =  0x7C; // B
    dec_lut[12] =  0x58; // C
    dec_lut[13] =  0x5E; // D
    dec_lut[14] =  0x79; // E
    dec_lut[15] =  0x71; // F

  /* Event loop never exits. */
  while (1)
  {
      //   --------------------> Connect SW to LEDR <--------------------
```

CDT Build Console [NIOS_II_Test_code_HAL_bsp]

```
10:31:17 **** Clean-only build of configuration Nios II for project NIOS_II_Test_code_HAL_bsp ****
make clean
[BSP clean complete]

10:31:19 Build Finished (took 1s.520ms)
```

*Figure: Correct compilation of the code with HAL*

## 2.5 Conclusions

As the design in VHDL is very complex, it was impossible to simulate using Modelsim. The only way to test the code was CPUlator which does not support the Hal libraries. However, the C code behaves as expected.