UMONS
Université de Mons

Faculté Polytechnique

Tutorial: How to Drive an Accelerometer Sensor with an Altera Cyclone V SoC Board

Course: Hardware and Software Platforms

Master (first year) in electrical engineering, specialism 'Multimedia and Telecommunications'

Dieudonné KASONGO KABASELE
dieudonne.kasongokabasele@student.umons.ac.be

POLYTECH MONS

Supervised by:
Professor Mr. Carlos Alberto VALDERRAMA SAKUYAMA

June 2019

Content

Introduction

This tutorial is providing and teaching how to drive and control an accelerometer sensor, which is an I2C peripheral device, with an Altera cyclone V. The board and the accelerometer sensor model you will use are respectively DE1-SoC, which is built on a FPGA, and LIS3LV02DL sensor.

To perform this project, we will divide this tutorial in two parts. The first part "Before the lab" will help you to understand the I2C serial interface and get you familiar with an accelerometer C code. On this C code you will base the beginning of your work. The second part "During the lab" will drive you step by step to the end of the project.

It's important to separate the driver, the application and the combination of the two. Because you have to simulate these three part individually in order to see if what you are doing is correct. These separations will appear in the second part "During the lab". This part describe step by step all the steps you will need to follow to complete the project. For this project it's asked to read the different axis direction (X, Y, and Z) of detectable acceleration in bytes. We will not read it in decimal numbers. Indeed, we will retrieve and display the accelerometer detectable acceleration (X, Y, and Z) axis values on the board's leds.

Chapiter 1

Before the Lab

Herein are some important information you need to know at the beginning of this tutorial. In this chapter, we will present an overview on the I2C bus protocol, features and we will study the way it is used in the C PIC program.

1.1 What is the I2C bus?

The Inter-Integrated Circuit, abbreviated as I2C is a serial bus short distance communication protocol developed by Philips Semiconductor (now NXP Semiconductors [5]) that permits various electronic components to communicate with only three connections [8]. These connections are the data signal SDA, the clock signal SCL and the electrical reference signal (ground). It has the particularities that there is one single address for a device and that it is a multi-master with collision detection and arbitration.

To take the control of the bus, the SDA and the SCL have to be at 1. For data transmission, we have to monitor the start and the stop condition:

- Start condition: SCL at one and SDA goes to zero
- Stop condition: SCL and SDA at one

After checking that the bus is free, then taking control of it, the circuit becomes the bus's master: it's him who generates the clock signal. Hereunder a picture which illustrates this well (Cf. Figure 1.1)

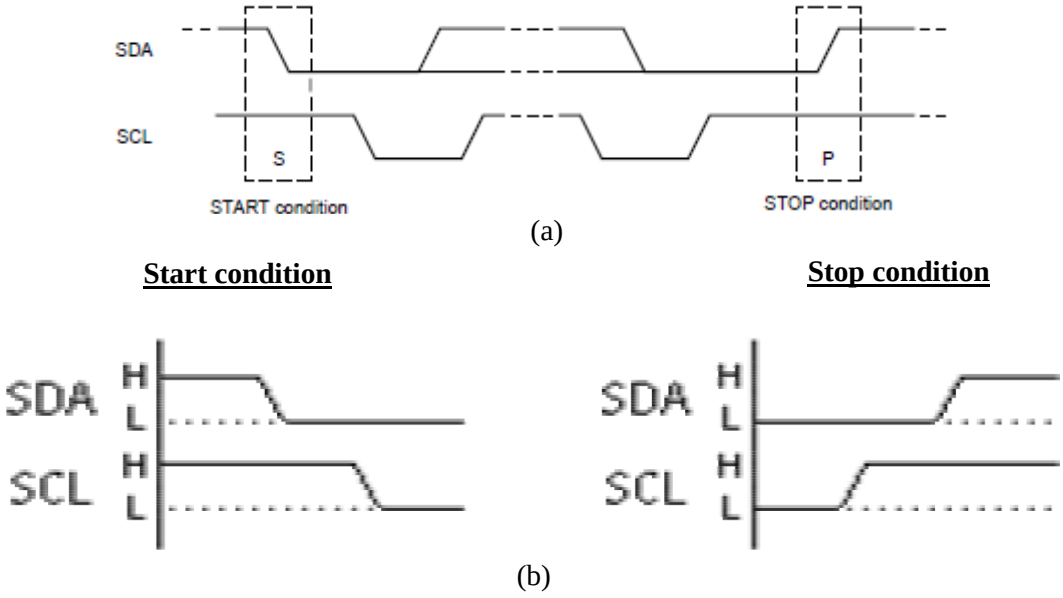

Figure 1.1: Image representing the bit start and the bit stop conditions during the take around of the bus (a) [8] & (b) [5].

1.1.1 Byte format and transmission

Every byte put on the SDA line must be eight bits long. The number of bytes that can be transmitted per transfer is unrestricted. Each byte must be followed by an Acknowledge bit.

To transmit a byte, after start condition, the master sends first the most significant bit (MSB) on the SDA. The data is approved by the master when he sends a ’1’ on the SCL. Then, he transmits the second most significant bit only when SCL goes back to ’0’ and so on until the entire byte is transmitted. The master sends an ACK at ’1’ when he sees that SDA has received the byte. The slave sends an ACK at ’0’ to say that the transmission is fine. When the master sees this low state, he can pursue. This approach is shown on Figure 1.2 and Figure 1.3.

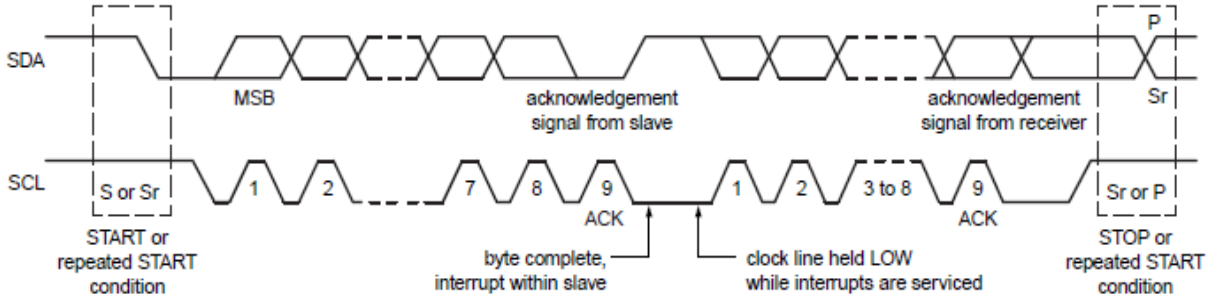

Figure 1.2: Data transfer on the I2C-bus [5].

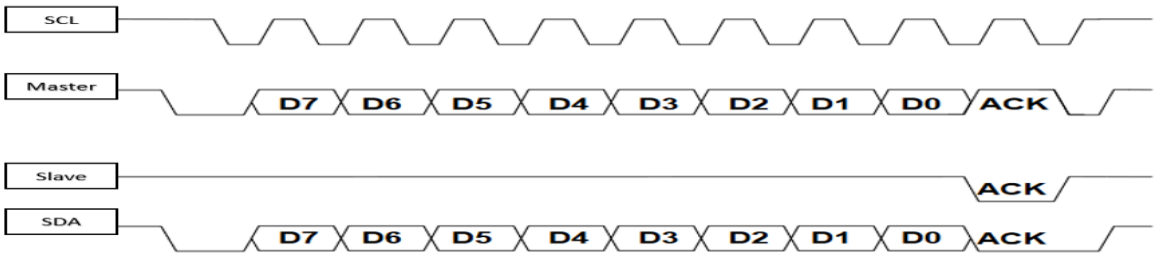

Figure 1.3: Byte transmission Illustration [8].

1.1.2 Writing a data

To write a data with I2C, we need to specify several data. First of all, after start condition, we need to send the slave device address. This address is 7 bits long and is concatenated with a R/W bit which specifies if we want to read (’1’) or to write (’0’). So, we send the slave device address and the R/W bit (here, it is equal to 0). Then, after the slave ACK, we send the data that we need to be written (cf. Figure 1.4).

1.1.3 Reading a data

To read a data, at first, we need to send some data and then we can read. The master sends the slave device address and the R/W bit (here, equals to 1). Then, after the ACK of the slave has been received, its sends the data to the SDA. Eventually, the master sets the ACK to 0 to continue the reading and to 1 to stop it (cf. Figure 1.4).

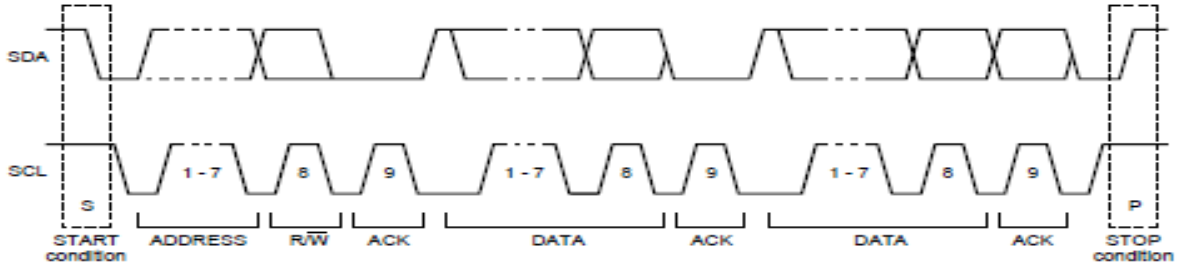

Figure 1.4: A complete data transfer [5].

1.2 PIC program Analysis

In this section, we are going to analyze the PIC code that you will receive with this document. First, we can see that there are several functions called by the main program. There is a function to write 1 byte, a function to read 1 bytes, another function to read 2 byte. We can also see that there are two functions called a lot of times: *i2c_start* and *i2c_stop*. These functions represent respectively the start and the stop sequences.

1.2.1 Writing a byte

The function *ecr_i2c* is very similar to what we saw in the I2C protocol, i.e. the function sends the device address, the register address and the data to be written (cf. Figure 1.5).

```
//-------------Writing of  I2C------------------
void ecr_i2c(byte device, byte address, byte data) {
    i2c_start();
    i2c_write(device);           //We write the device
    i2c_write(address);          //We write the address
    i2c_write(data);             //We write the data
    i2c_stop();
}
```

Figure 1.5: Code used in the function *ecr_i2c*

1.2.2 Reading one byte

The below code is used to read a data which is 8 bits long (cf. Figure 1.6). We can see that like with the I2C, we begin with the device address specification and the register address. Then, we give the device address once again and we wait for the data sent by the sensor.

```
//-------------Read of one byte-----------------
signed byte lec_i2c(byte device, byte address) {
   signed BYTE data;

   i2c_start();
   i2c_write(device);       //Specification of the Device
   i2c_write(address);      //Specification of the register
   i2c_start();             //We loop. This way we will be able to read the data
   i2c_write(device | 1);  // | is a OR logic bit by bit and writing |1 is like copying again the device
   data=i2c_read(0);
   i2c_stop();
   return(data);
}
```

Figure 1.6: Code used in the function *lec_i2c*

1.2.3 Reading two byte

The function called *lecdb_i2c* allows to read data that are 16 bits long. We can see on the Figure 1.7 that the function begins with sending the device address and the register address. Then, it sends the device address again and waits for the slave to send the data.

```
//-------------Read of 2 bytes -----------------
signed int16 lecdb_i2c(byte device, byte address) {
  BYTE dataM,dataL;
  int16 data;

  i2c_start();
  i2c_write(device);
  i2c_write(address);
  i2c_start();
  i2c_write(device | 1);
  dataM = i2c_read(1);            // Read of MSB (8th most significant bit), we want to read 16 bits but we can only transfer 8 bits at a time
  dataL = i2c_read(0);            // Read of LSB (8th less significant bit)
  i2c_stop();
  data=((dataM*256)+dataL);       // We collect de final value (the eight most significant bit, we multiply by 256.
  lcd_gotoxy(1,1);                // To go on location (1,1) gives the place X and Y we want to be on the lcd
  printf(lcd_char,"MSB:%d  LSB:%d  ",dataM,dataL);  // Display the data on the lcd
  return(data);
}
```

Figure 1.7: Code used in the function *lecdb_i2c*

1.2.4 Accelerometer reading

In the code Hereunder, we use every function defined in the previous sections to do a complete reading of different axis data on the accelerometer sensor to know X, Y, Z axis (cf. Figure 1.8). The comments in the code are detailing what is the purpose of each line.

```
//-------------Reading Accelerometer I2C-----------------
void lecture_ACC() {
    signed long byte axe_x, axe_y, axe_z = 0;
    byte tmp=0;
    signed int16 datax, datay, dataz=0;


    ecr_i2c(ACC,0x20,0b01000111);/*Writing in the register Ctrl_Reg1 (0x20), allows the activation of the 3 axes x, y and z. WARNING full scale to 2g!!*/
    tmp=lec_i2c(ACC,0x27); /*Contains the Status_Reg 0x27 register which says if there is new information on the x, y, z axis, */


    if ((tmp&0x0C)==0x0C) axe_z=lec_i2c(ACC,0x2d);/*If new data is present on the z axis, we read*/
    if ((tmp&0x0A)==0x0A) axe_y=lec_i2c(ACC,0x2b);/*If new data is present on the y axis, we read*/
    if ((tmp&0x09)==0x09) axe_x=lec_i2c(ACC,0x29);/*If new data is present on the x axis, we read*/

    datax = axe_x*18; /*multiplication by 18 (because FS byte to 0, ie full scale to 2g) to have it in g, 1 corresponds to 18mg*/
    datay = axe_y*18;
    dataz = axe_z*18;


    lcd_gotoxy(1,2);//Location of the lcd where we will display (2nd line) (x = column y = line)
    printf(lcd_char,"X%ld Y%ld Z%ld  ",datax,datay,dataz);//Display on the lcd

}

void main () {
```

Figure 1.8: Code used in the function *lecture_ACC*

Chapiter 2

During the Lab

In this chapter, we will describe, step by step, what was done in this project. From the beginning until the time we sent the code on the chip.

2.1 I2C driver

2.1.1 Code

Some research was made to find a first code which will help to start the project. Indeed, to have a consistent base to begin this project was very important and wanted. An I2C VHDL code [2] was found on the internet. This tutorial has been made using Intel Quartus Prime software version 18.1.0 Lite Edition. The Author of this tutorial cannot guarantee that other versions of Quartus will work with the provided files.

Once you have the VHDL code mentioned above:

- Create and name a folder on your computer in which your project files will be saved to
- Launch Quartus – 64bits installed on your computed, if not installed download it using [1]

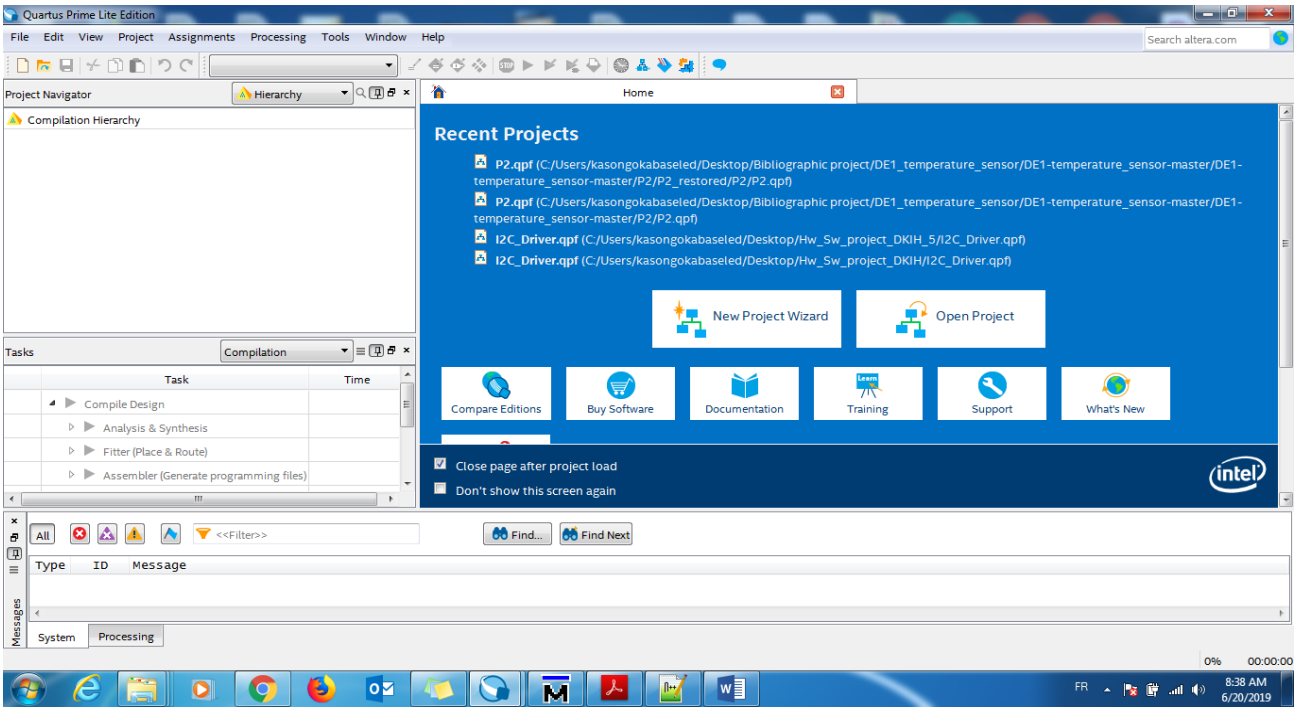

Figure 2.1: Quartus new project creation step 1

- Create a project: File → New Project Wizard → Click "Next" until it's created.

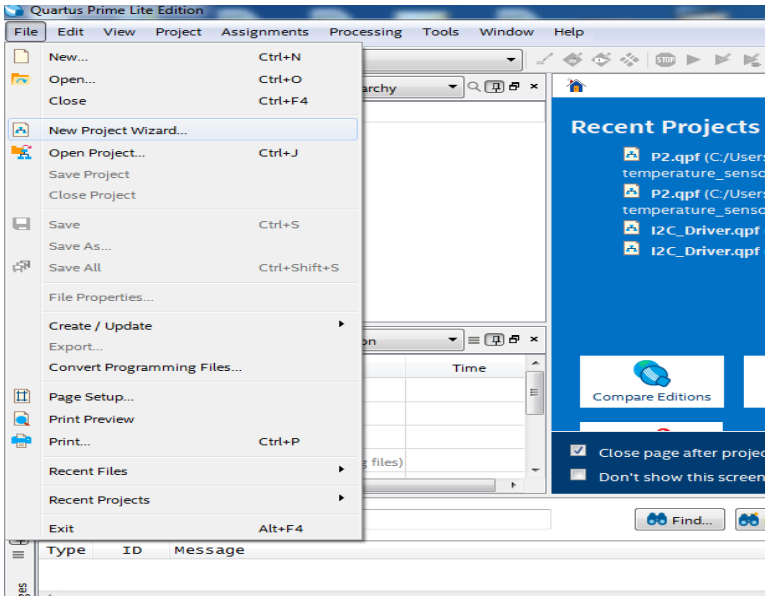

Figure 2.2: Quartus new project creation step 2

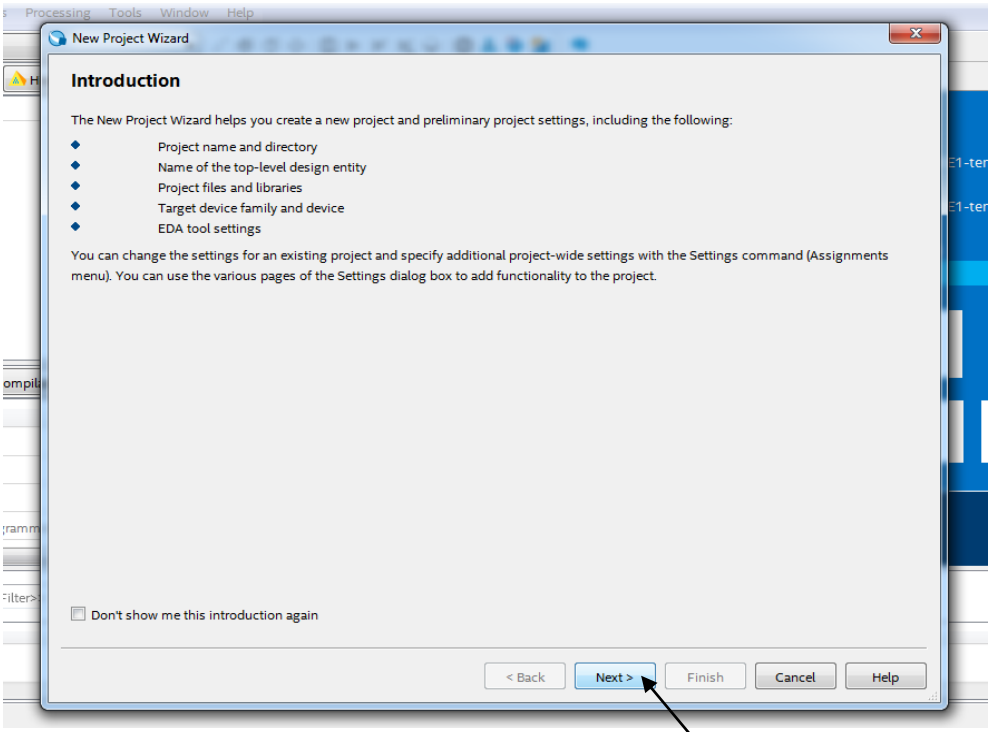

Figure 2.3: Quartus new project creation step 3

→ Click "Next"

- Choose the directory where you created the project folder → Name the project and the top level entity in my case it is "Accelerometer"

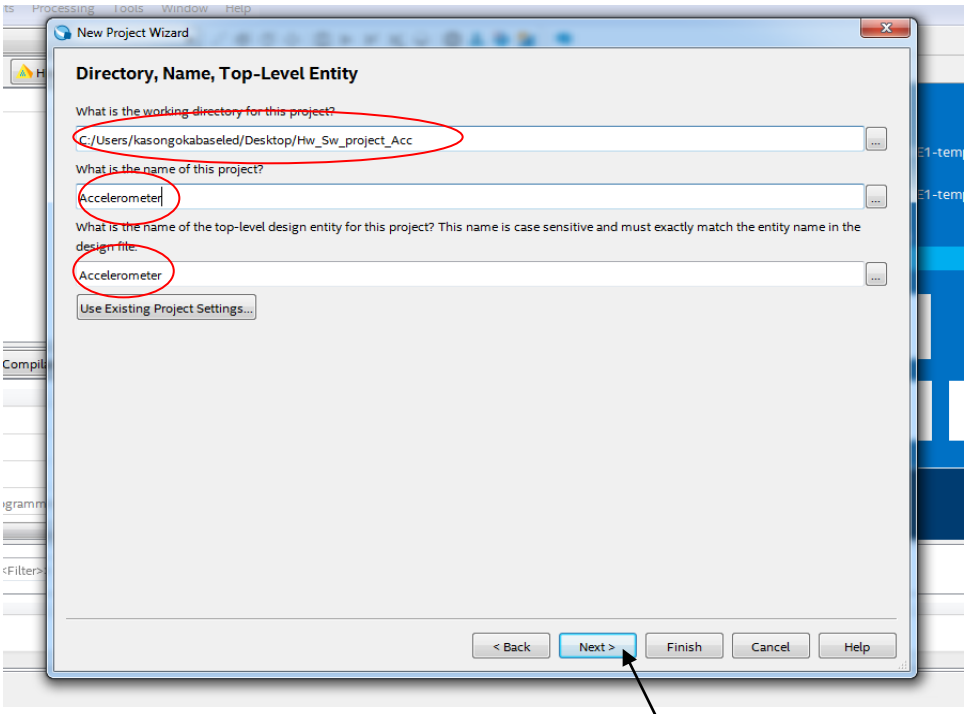

Figure 2.4: Quartus new project creation step 4

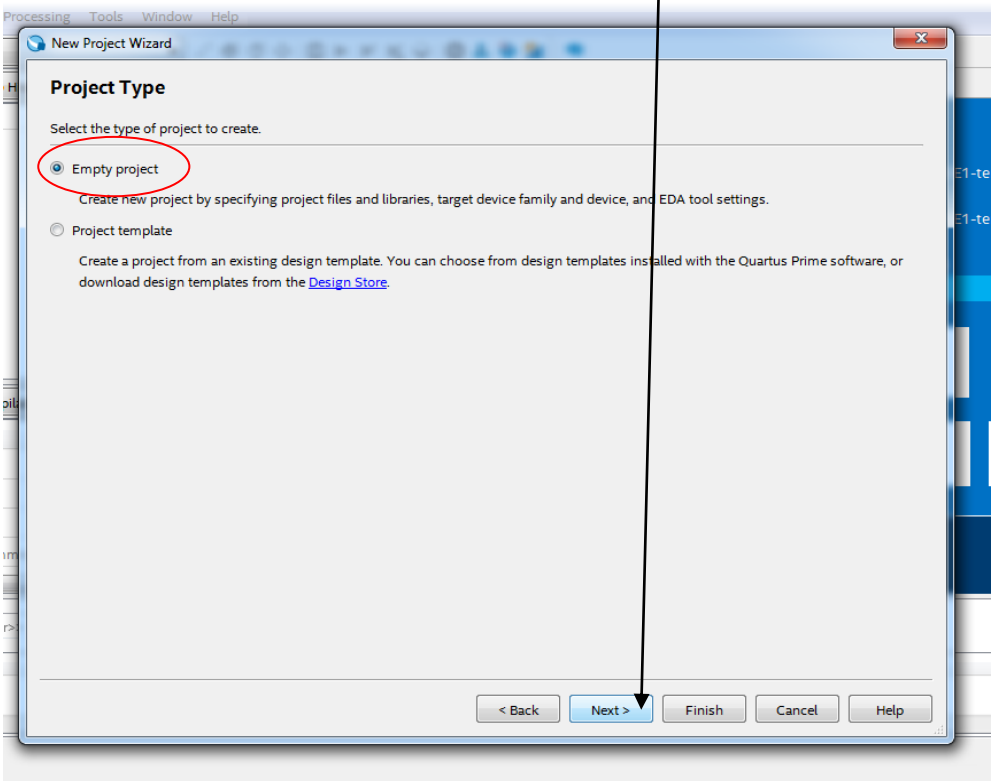

Figure 2.5: Quartus new project creation step 5

- Choose the device family → click on next → then finish

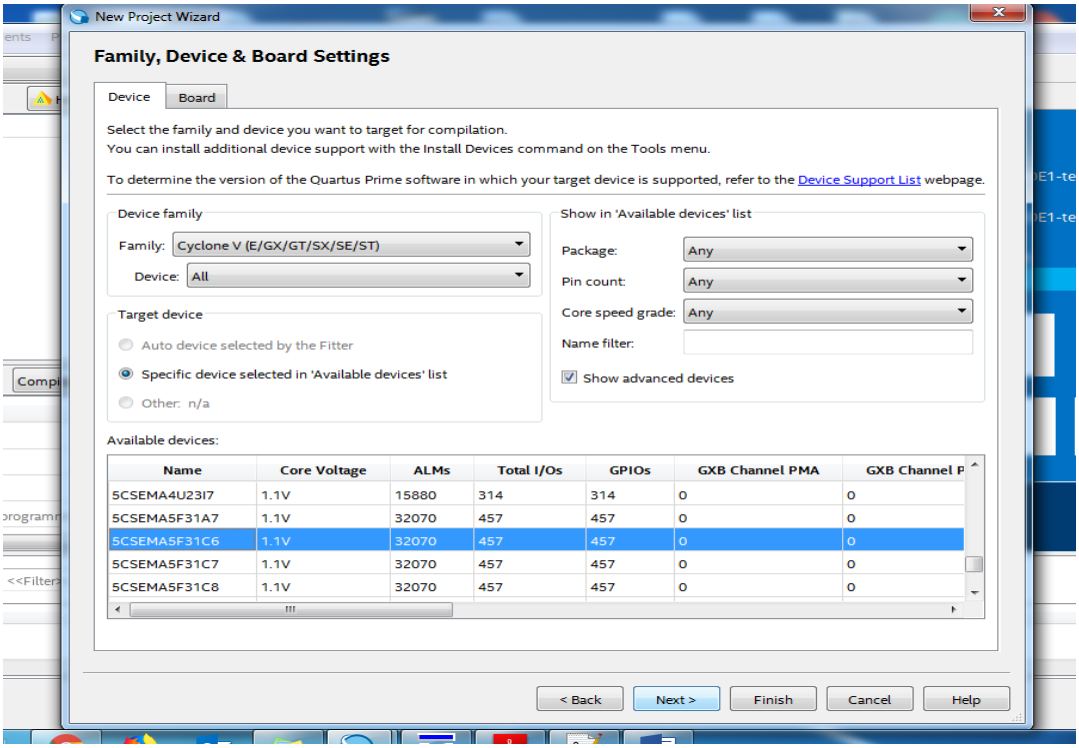

Figure 2.6: Quartus new project creation step 6

- Create a new file in the project

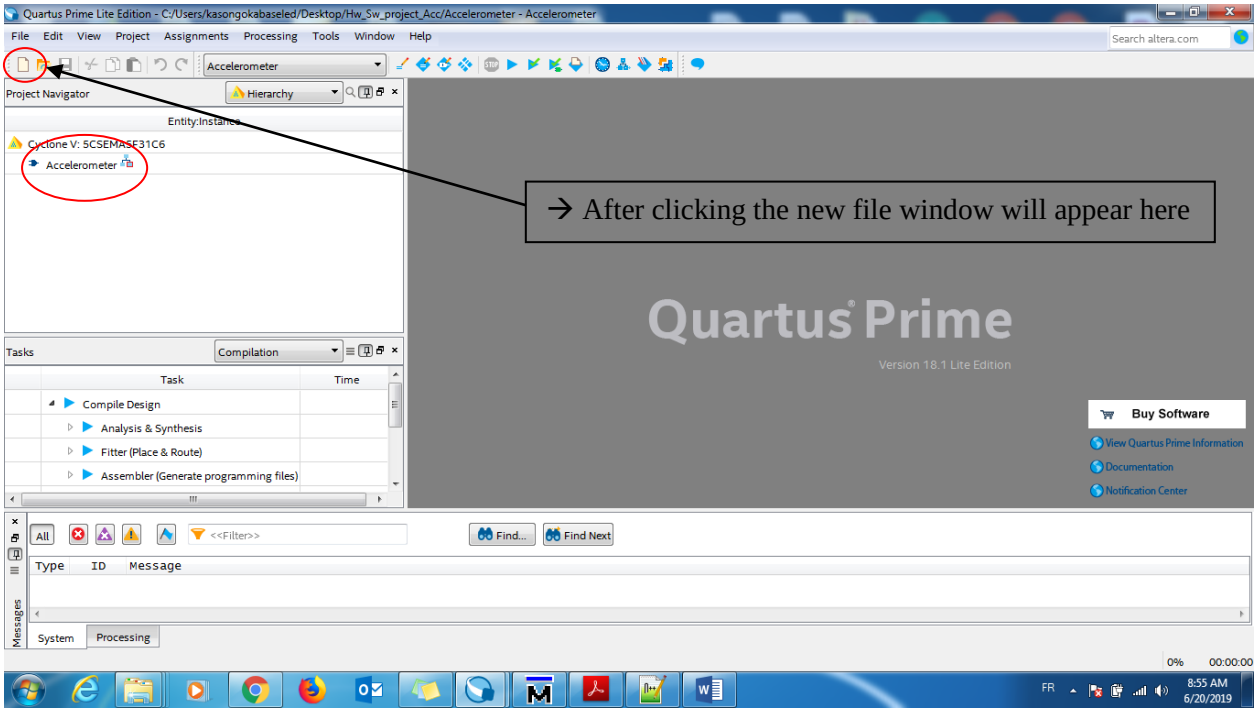

Figure 2.7: Quartus new file creation in the project step 7

- Paste the VHDL code [2] in the new file window

- Save the file in the project folder path with the name I2C_Driver

- Compile the I2C_Driver.vhd file (use the shortcut represented with a play symbol) as shown in the below print screen (Cf. Figure 2.8)

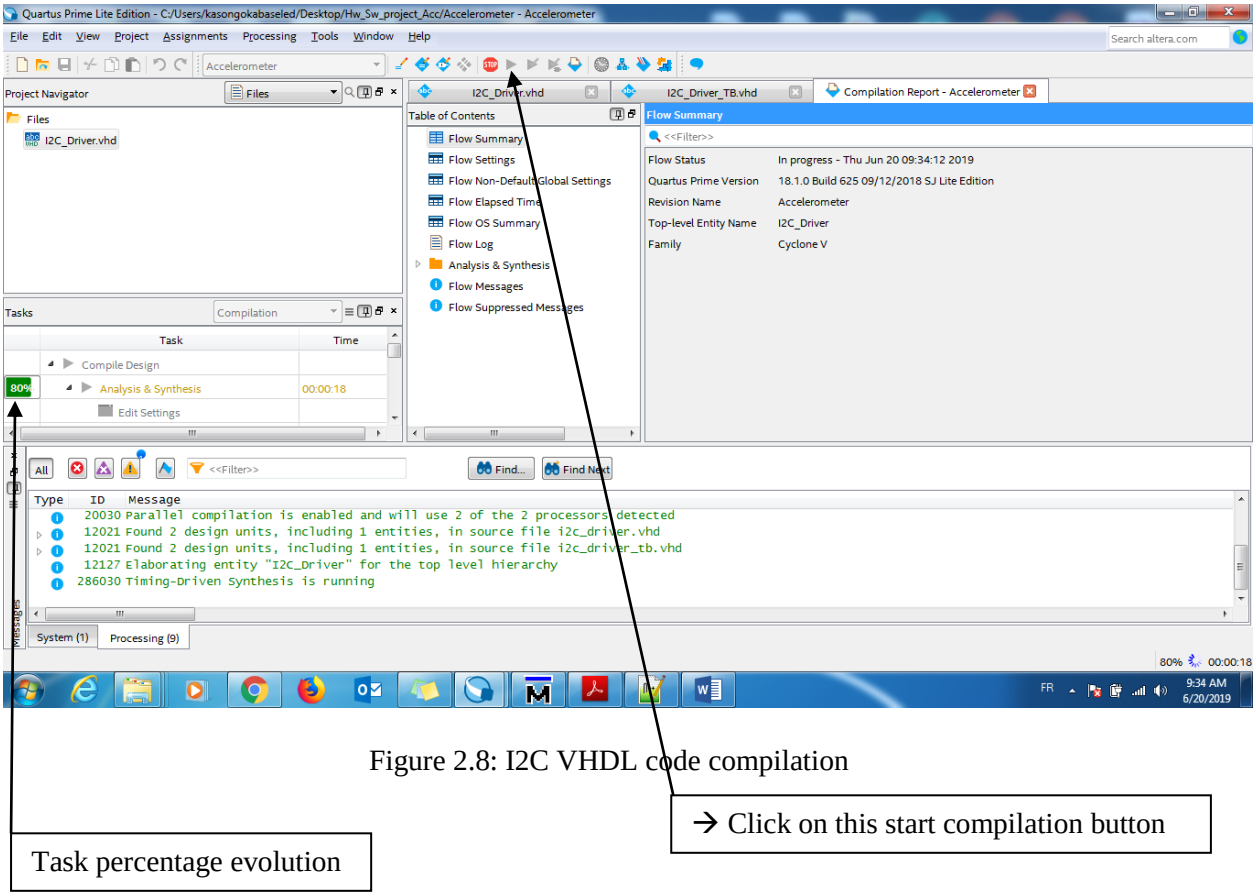

Figure 2.8: I2C VHDL code compilation

Once compiled, Quartus shows us the I2C driver state machine Figure 2.10 but the clearer representation was the one on the website [2] Figure 2.9. On Figure 2.9, we have an overview of this state machine with key signals conditions. Once you have understood the state machine, you will have to create a Test Bench.

To create a Test Bench, you have to create another new file in the project (name it as you want; in my case it is named I2C_Driver_TB). A test Bench is a VHDL module which tests the behavior of another module. He produces the signals to send to the system we want to put under test and check that the outputs of the circuit are correct. Hereunder is a part of the code you need to implement (Cf. Figure 2.11). In the part port map we associate our PORT to the corresponding signals.

Then, we have some process:

- One for the clock (clk_process)

- The SDA_process is there to generate fake data read from the slave (as there is nothing connected yet at the SDA's other side). Indeed if we don't provide input stimulus with fake data we can't see the simulation working properly we will face signals with unknown values.
- The last process called stimulus describe the writing of the device address, the register address (data configuration) and the reading mode activation for data coming from the slave.

The driver also need input and outputs that you have to define. You can find an illustration below (Cf. Figure 2.29). To create this code, we looked at the state machine on Figure 2.9 and translated it.

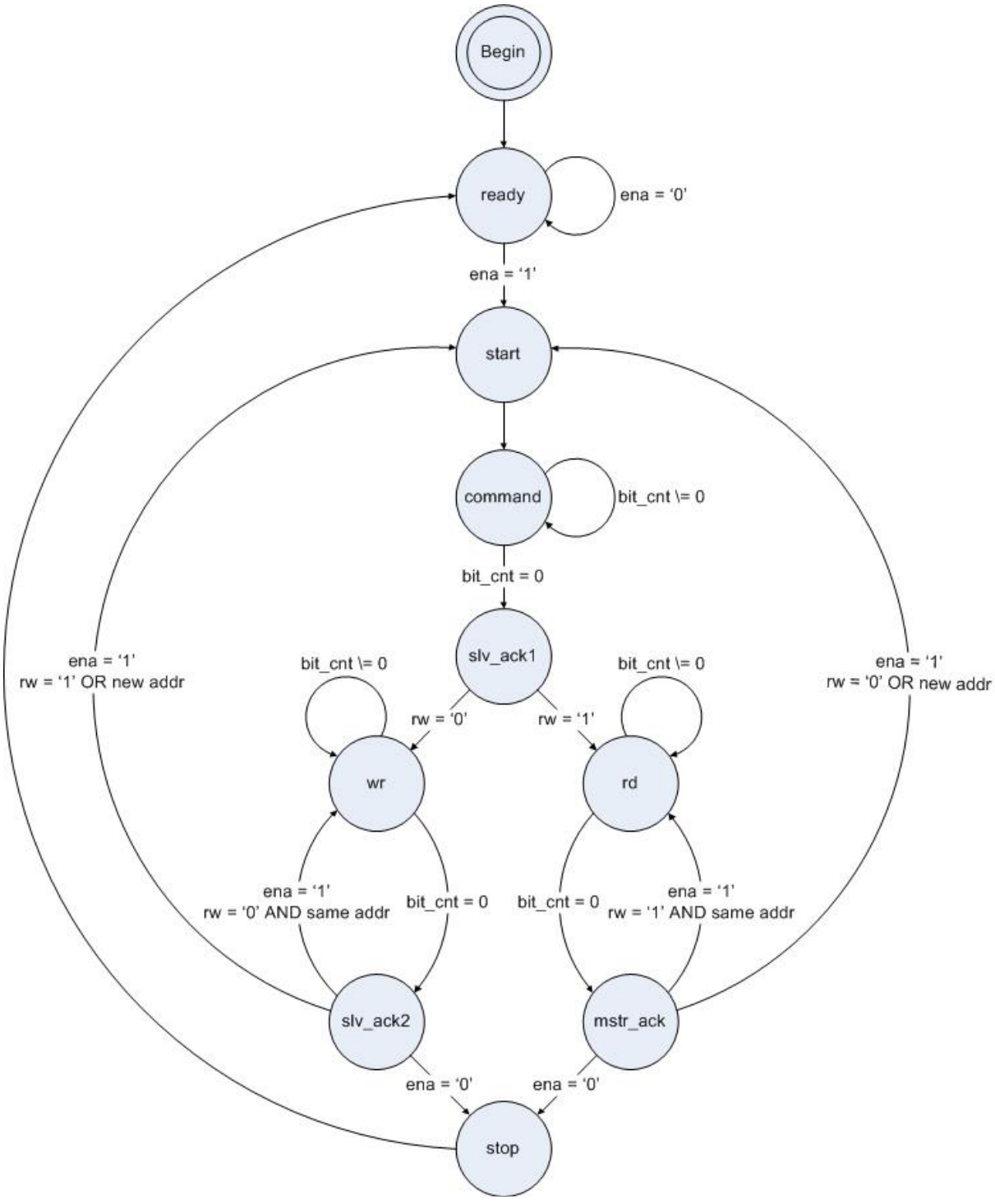

Figure 2.9: I2C Driver State Machine from [2]

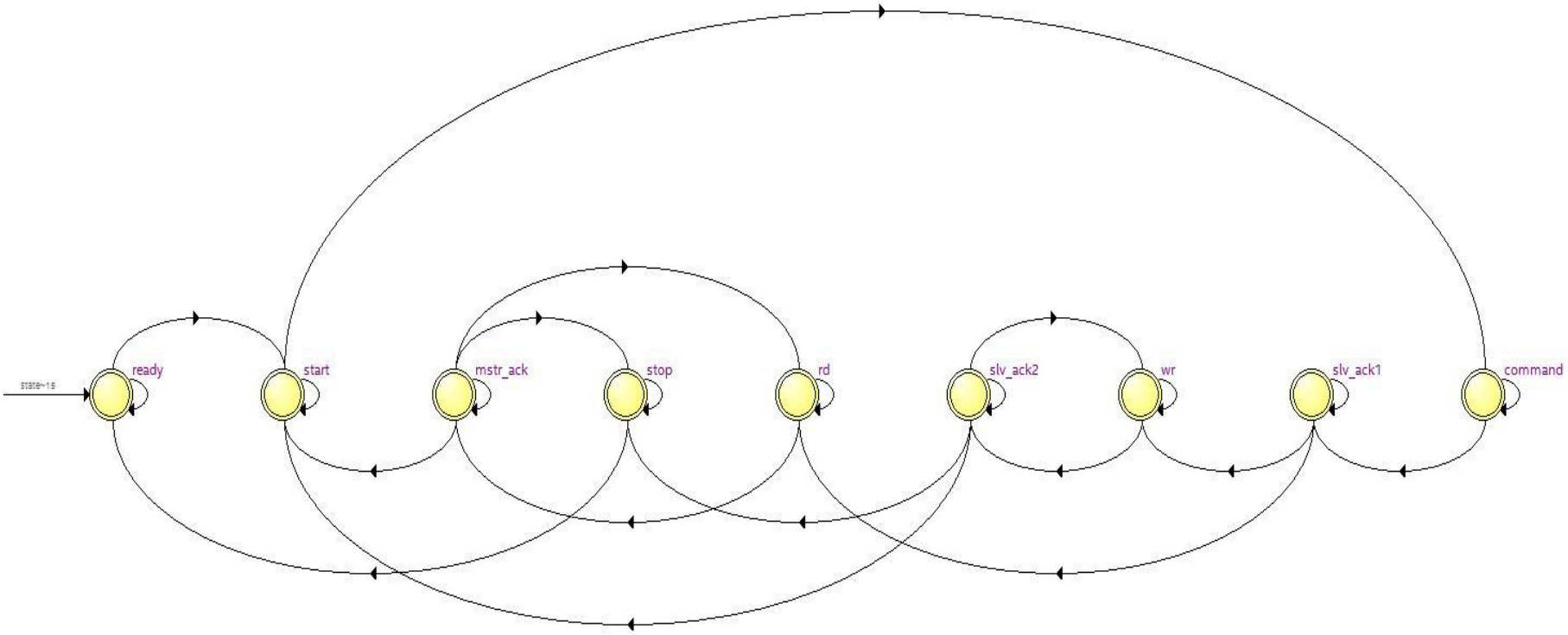

Figure 2.10: Driver State Machine after the I2C_Driver.vhd file compilation on Quartus

```
    constant clk_period : time := 10 ns;
BEGIN

 UUT: entity work.i2c_driver(logic)
    port map(
          clk        => sclk        ,
          reset_n    => sreset_n    ,
          ena        => sena        ,
          addr       => saddr       ,
          rw         => srw         ,
          data_wr    => sdata_wr    ,
          busy       => sbusy       ,
          data_rd    => sdata_rd    ,
          ack_error  => sack_error  ,
          O_rdon     => sO_rdon     ,
          O_ack1on   => sO_ack1on   ,
          O_ack2on   => sO_ack2on   ,
          O_ackmon   => sO_ackmon   ,
          sda        => ssda        ,
          scl        => sscl
    );
```

```
clk_process : PROCESS

    BEGIN
       sclk <= '1';
           wait for clk_period/2;
       sclk <= '0';
           wait for clk_period/2;
    END PROCESS;
```

```
  SDA_process: PROCESS

          constant din : STD_LOGIC_VECTOR(0 to 15):= "1100111010001001";
          variable idx : natural range 0 to 15 := 0;

       BEGIN

           ssdard <= din(idx);
           wait until sO_rdon = '1'  ;
           while sO_rdon = '1' loop
               wait until sscl = '0';
               if (idx +1) = 16  then
                   idx := 0;
               else
                   idx := idx +1;
               end if;
               ssdard <= din(idx);
           end loop;
       end process;

 ssda <= ssdard    WHEN sO_rdon = '1'   ELSE
               '0' WHEN sO_ack1on = '1' ELSE
               '0' WHEN sO_ack2on = '1' ELSE
               '0' WHEN sO_ackmon = '1' ELSE 'Z';
```

```
stimulus : PROCESS

   BEGIN
      -- code executes for every event on sensitivity list

         sena <='0';
         srw <='0';
         saddr    <= "0111010" ;       -- device address
         sdata_wr <= "00100000";       -- register
         wait for clk_period*50;
         sena <='1';                   -- enable set to high for start condition
         WAIT UNTIL sO_ack1on <= '1';  -- Slave Acknowledgement 1
         WAIT UNTIL sO_ack2on <= '1';  -- Slave Acknowledgement 2
         sdata_wr <= "01000111";       -- config data during the writing mode
         WAIT UNTIL sO_ack2on <= '1';
         srw   <='1';                  -- reading mode activated
         WAIT UNTIL sO_ackmon = '1';
         WAIT UNTIL sO_ackmon = '1';
         sena <='0';                   -- enable set low for the stop condition and go to ready state
         WAIT ;

    END PROCESS stimulus;

END Behavior;
```

Figure 2.11: Test Bench Code

2.1.2 Simulation

Once you have done your Test Bench, you'll have to simulate it to see if the code of your driver works properly i.e. you don't get errors in your different signals (U->unknown values, X-> short-circuit,).

To do so, you'll have to follow below steps to set your test bench for simulation if it is not done:

- Set the I2C_Driver_TB file as Top – Level Entity (right click on the file)
- Go into the tab Assignments → then Settings and verify that in the section simulation, the compile test bench is I2C_Driver_TB

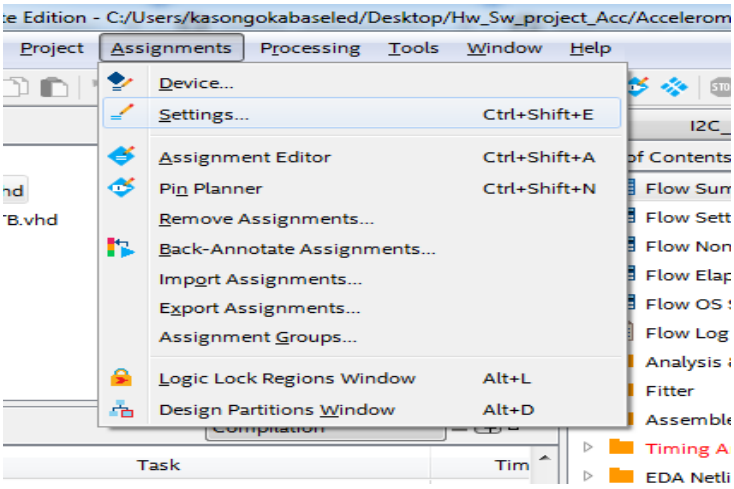

Figure 2.12: Setting the test bench for simulation step 1

- → If it is not the case like in Figure 2.13 hereunder, follow the below steps to set it. Otherwise, if the compile test bench is I2C_Driver_TB, then go to the tab tools steps to simulate directly.
- → Go to Test Benches

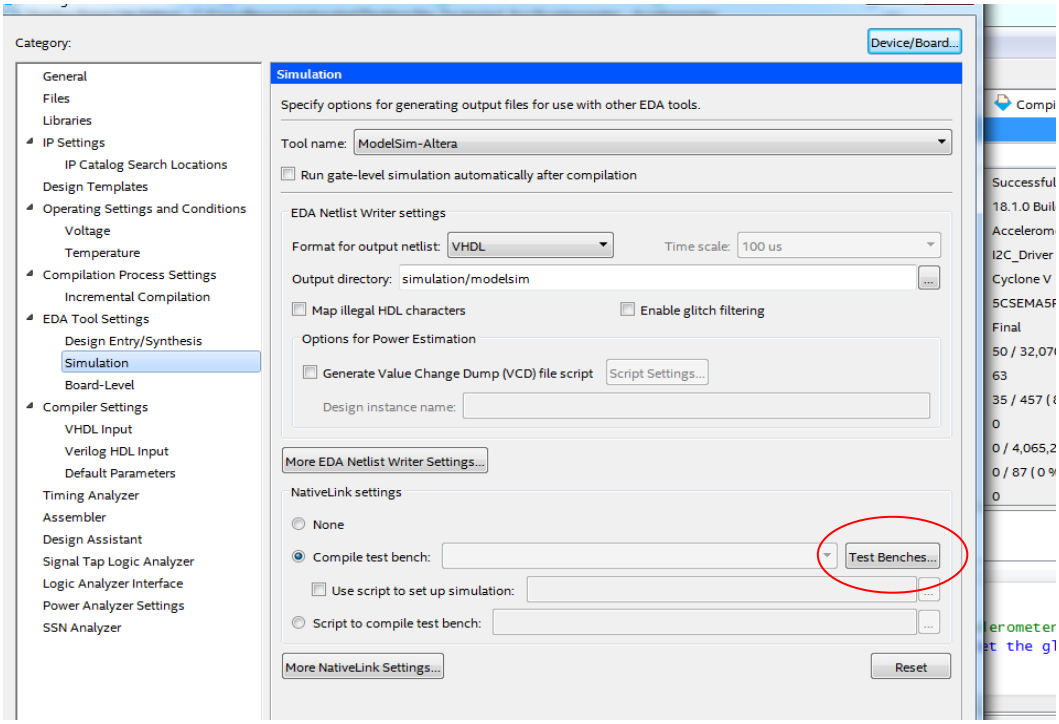

Figure 2.13: Setting test bench for simulation step 2

- → Click on new →

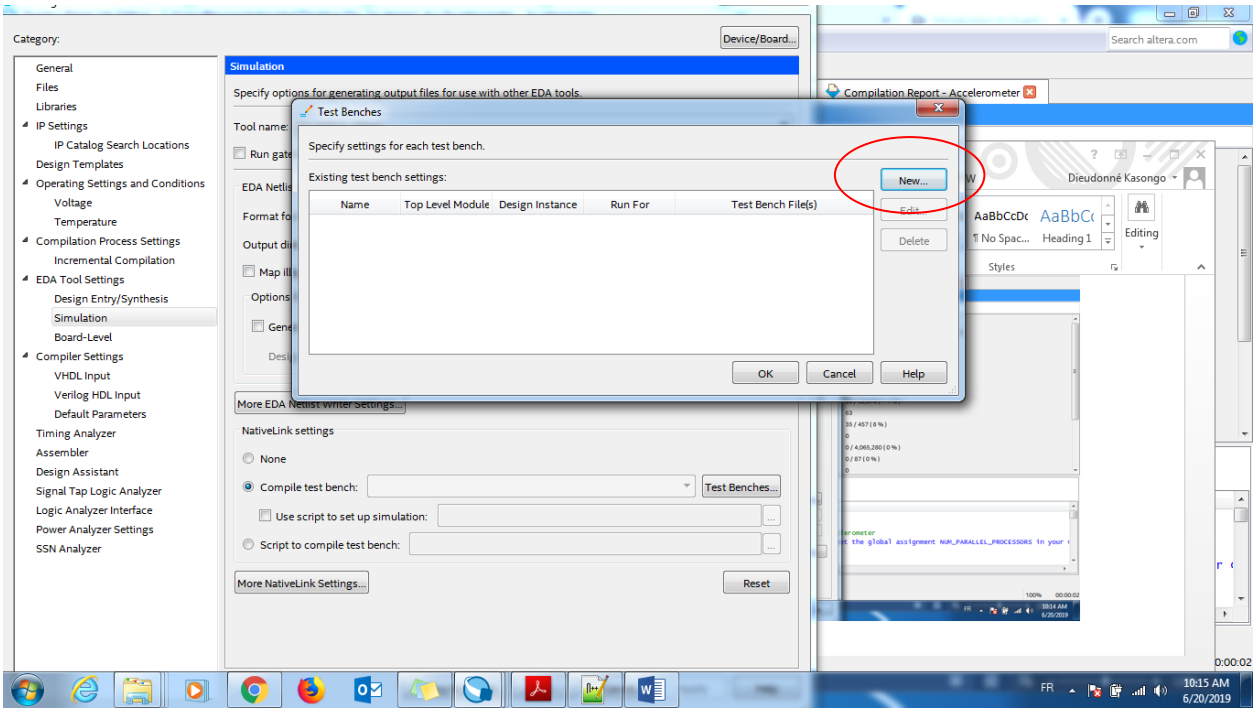

Figure 2.14: Setting the test bench for simulation step 3

→ Then name your Test Bench as below and the top level module in test bench → click to file name to choose your test bench file

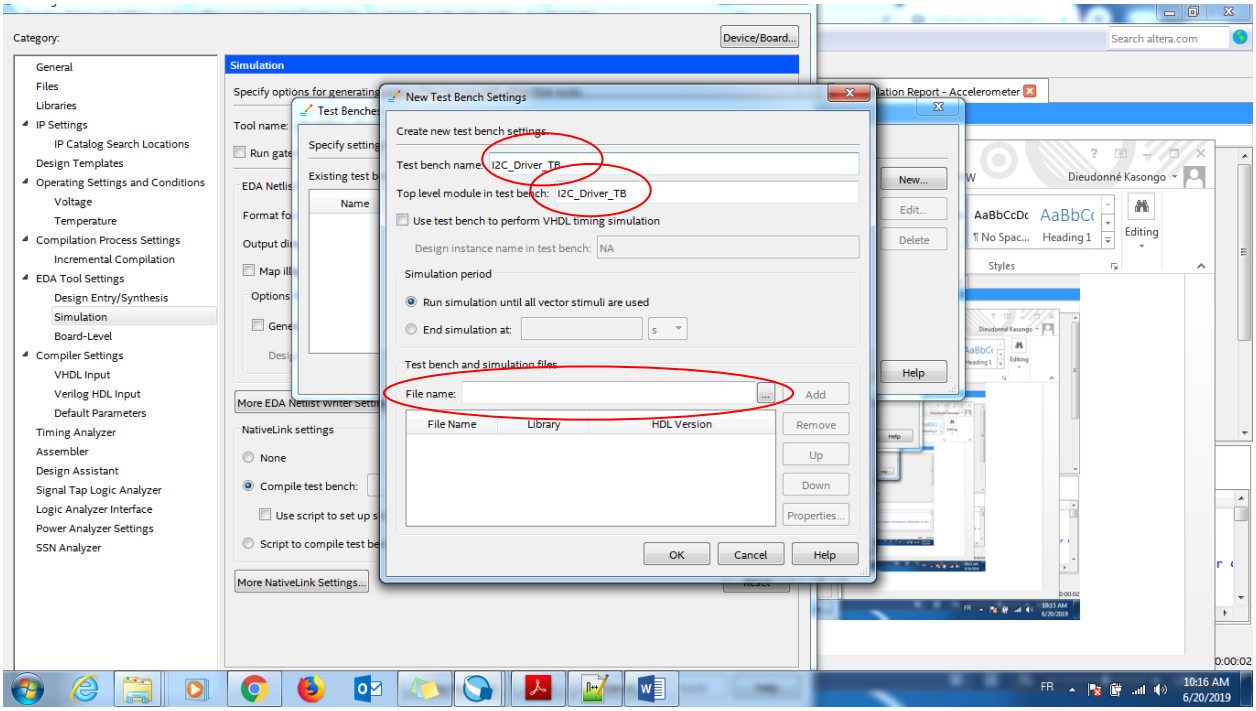

Figure 2.15: Setting the test bench for simulation step 4

- In my case I2C_Driver_TB.vhd → click to Open

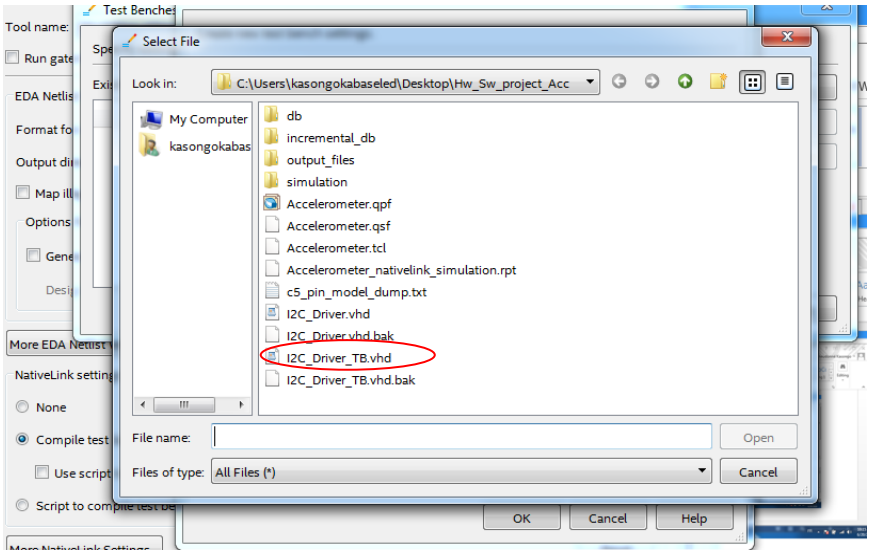

Figure 2.16: Setting the test bench for simulation step 5

- → click to Add

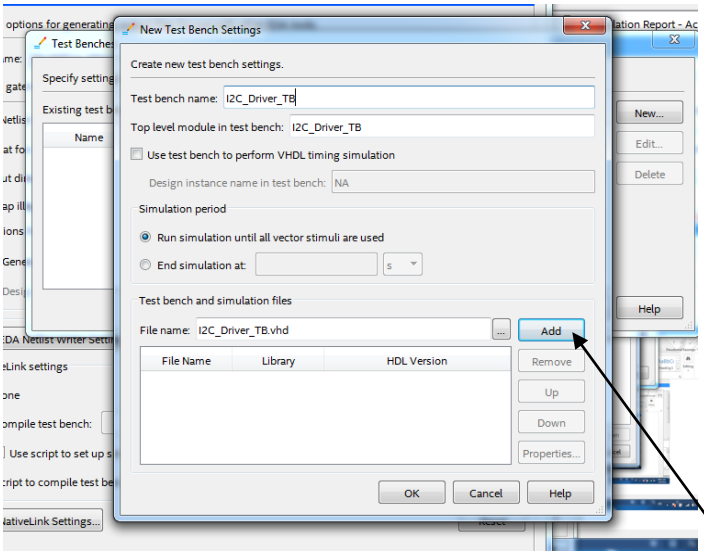

Figure 2.17: Setting the test bench for simulation step 6

- → click on Ok

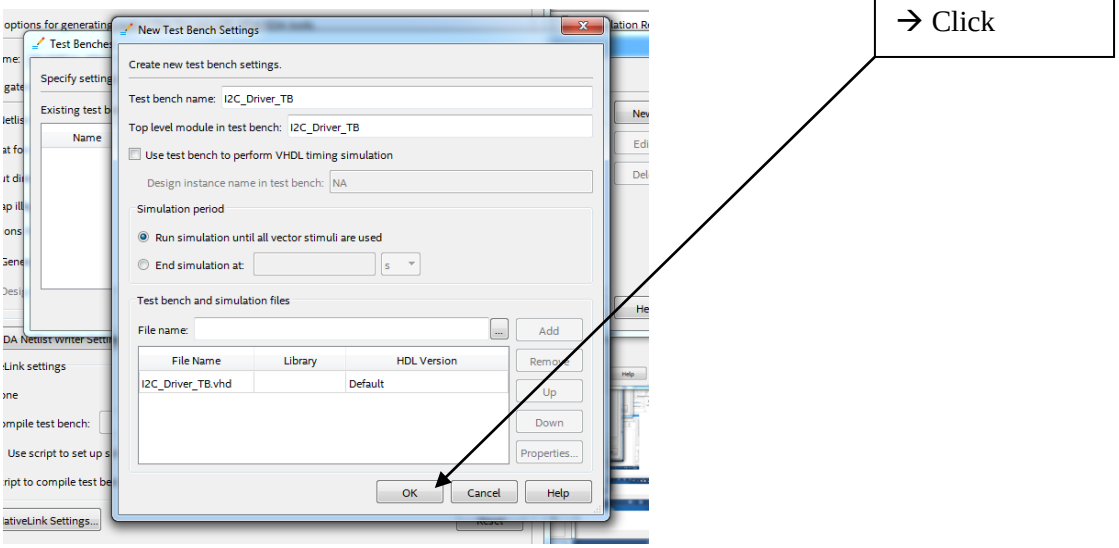

Figure 2.18: Setting the test bench for simulation step 7

- → click on Ok

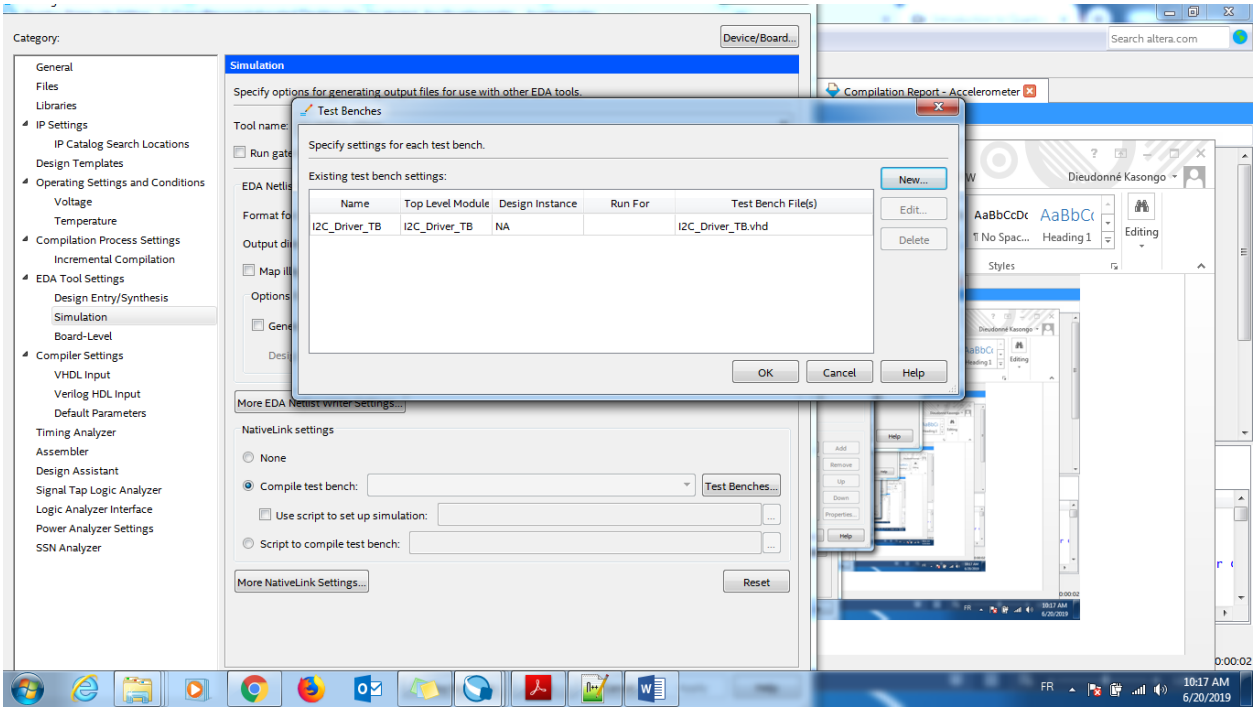

Figure 2.19: Setting the test bench for simulation step 8

- → Close the setting window

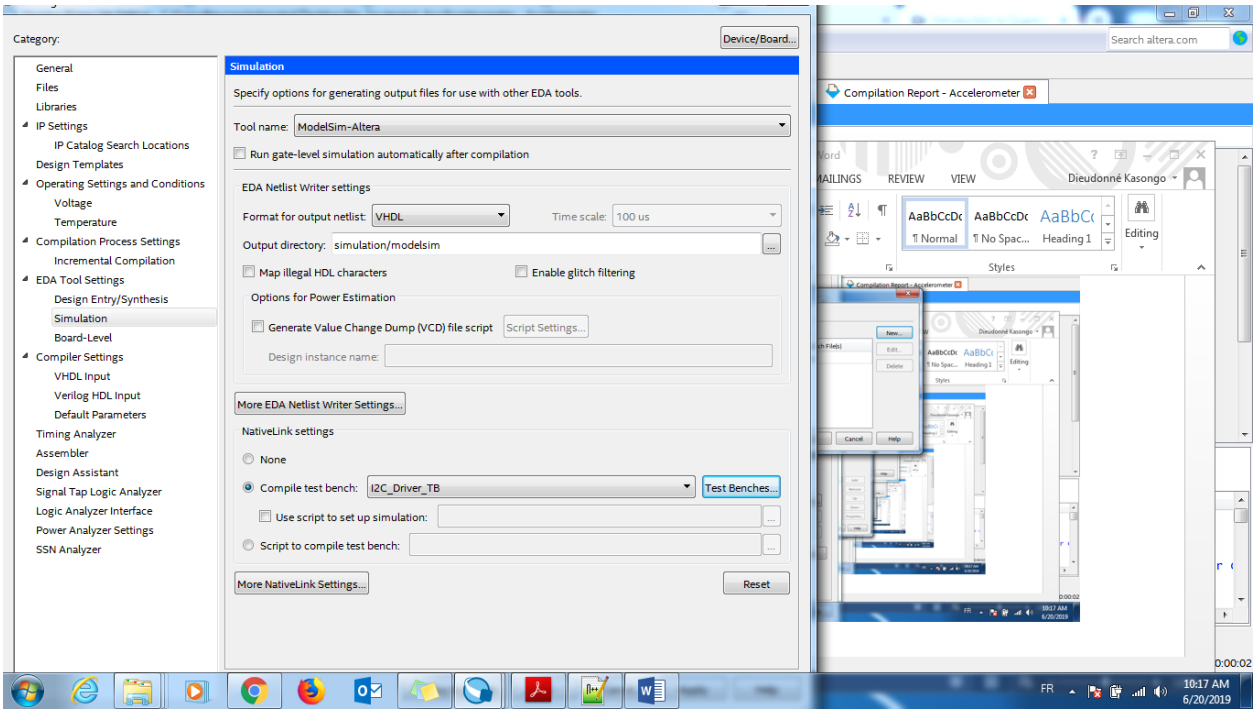

Figure 2.20: Setting the test bench for simulation step 9

- Go into the tab Tools → run simulation Tool → then RTL simulation

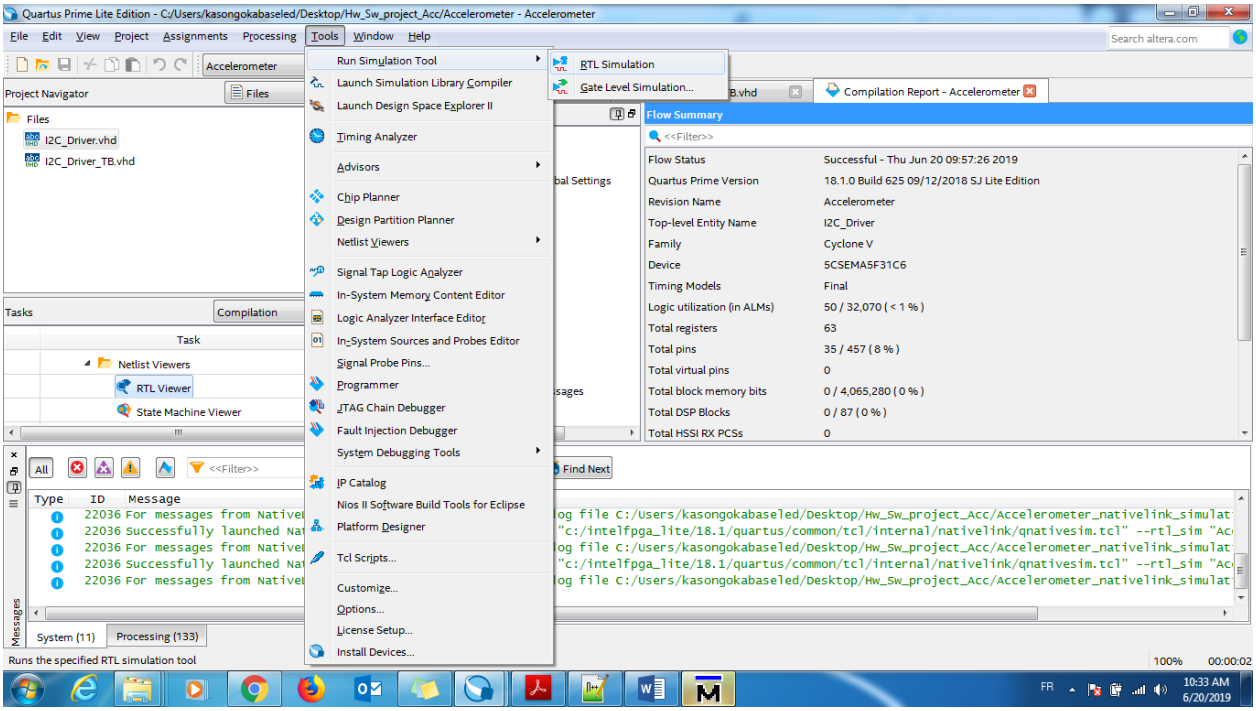

Figure 2.21: Launching the simulation step 1

- The Modelsim launches itself

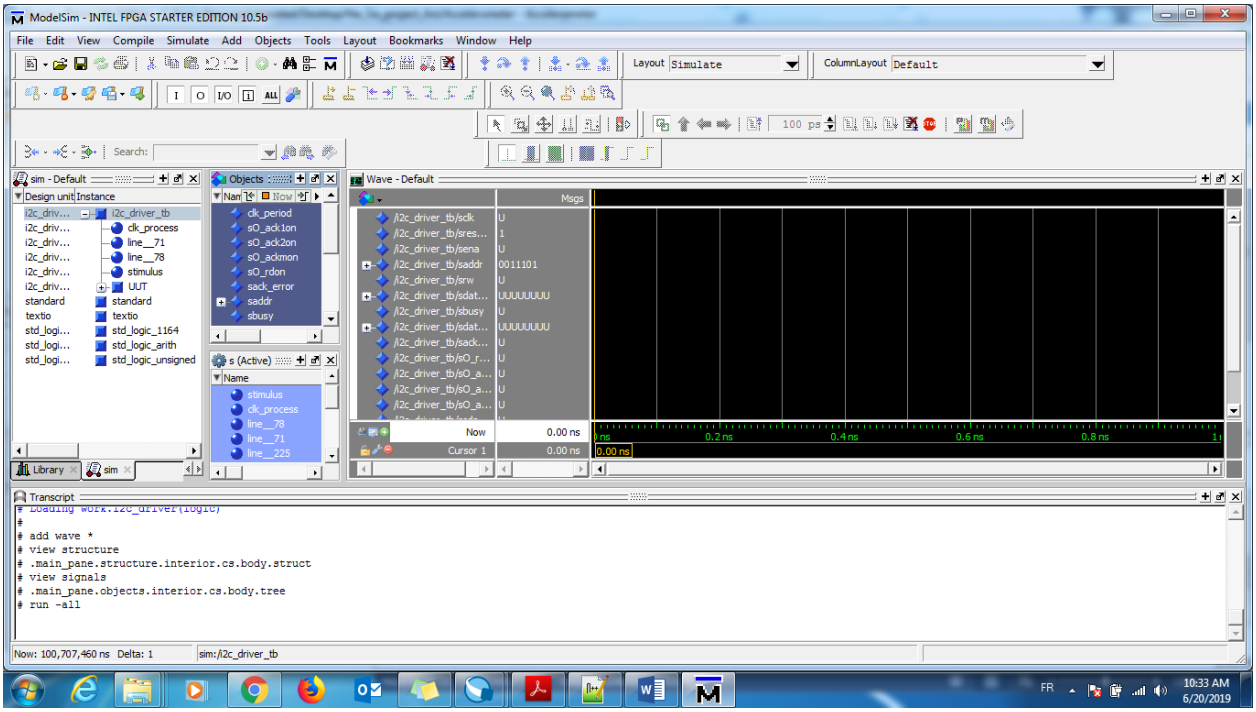

Figure 2.22: Modelsim window

- Go to the compile Tab on Modelsim window

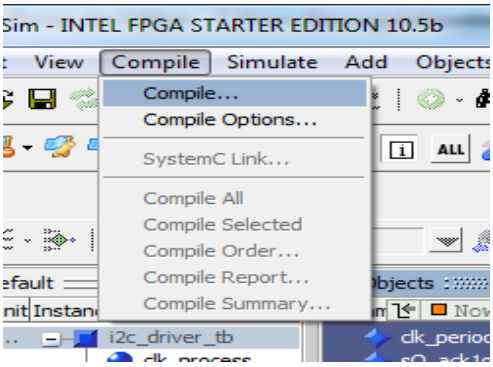

Figure 2.23: Launching the simulation step 2

- Select the source file to be compiled as shown below

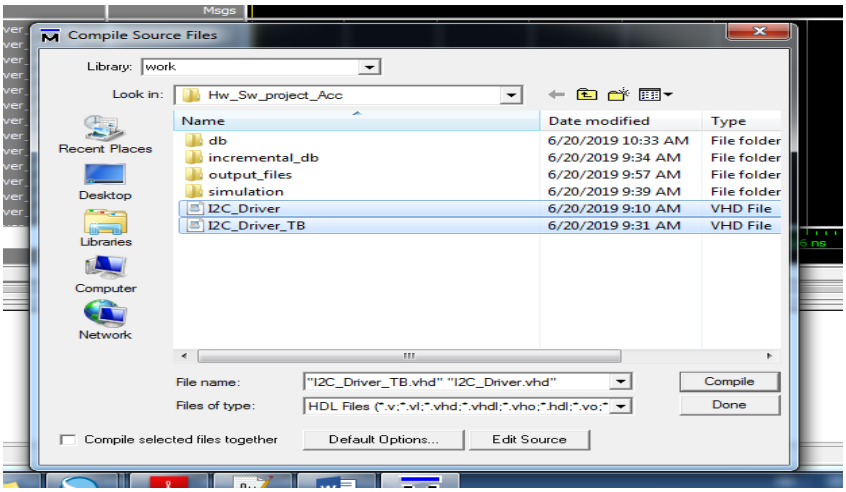

Figure 2.24: Launching the simulation step 3

- Go to the simulate Tab → Start Simulation

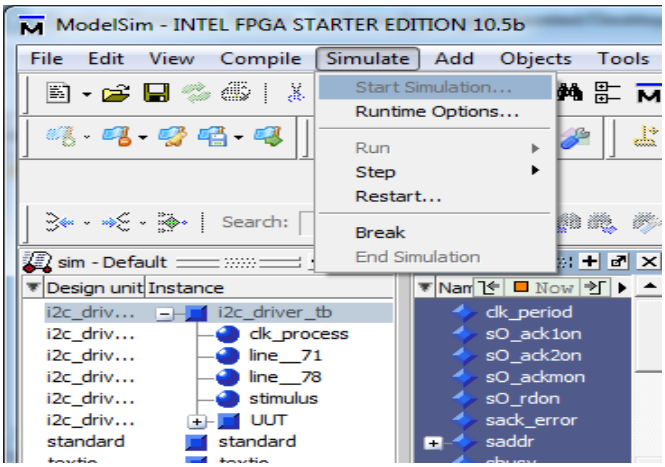

Figure 2.25: Launching the simulation step 4

- Select the I2C_Driver_TB test bench file to be simulated as shown below → then Click OK

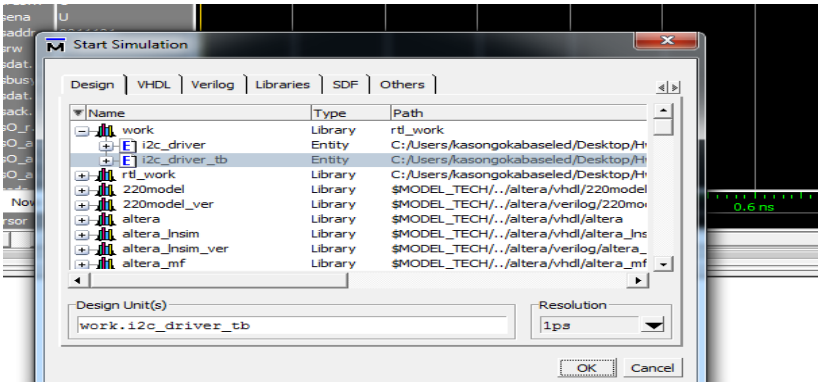

Figure 2.26: Launching the simulation step 5

- Drag and drop from objects window to waveform default windows all signals that you want to see waveform simulations

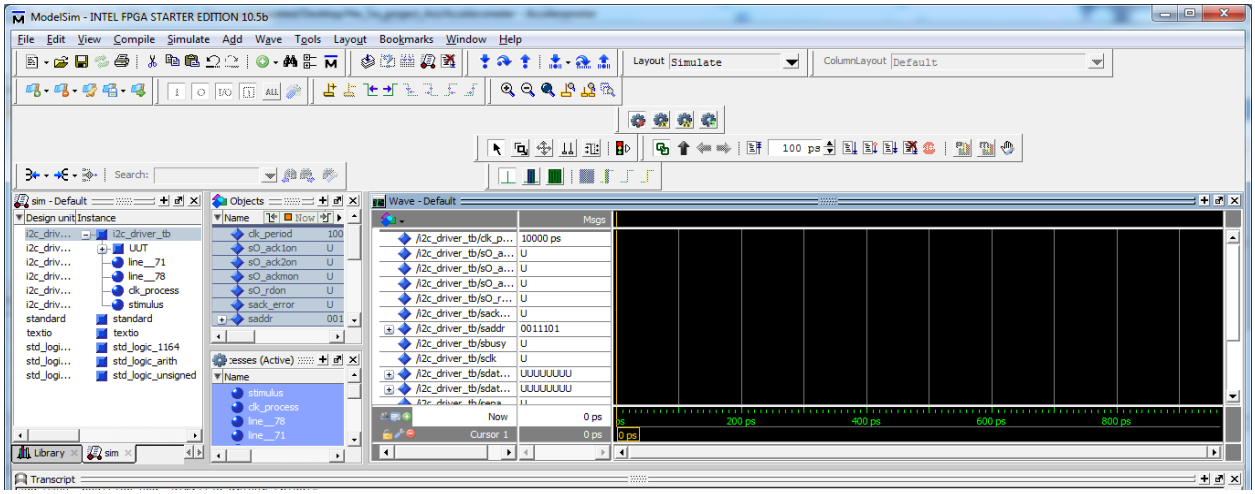

Figure 2.27: Launching the simulation step 6

Once all the above steps are well performed, check your results. You must have the same results as shown below (Cf. Figure 2.28).

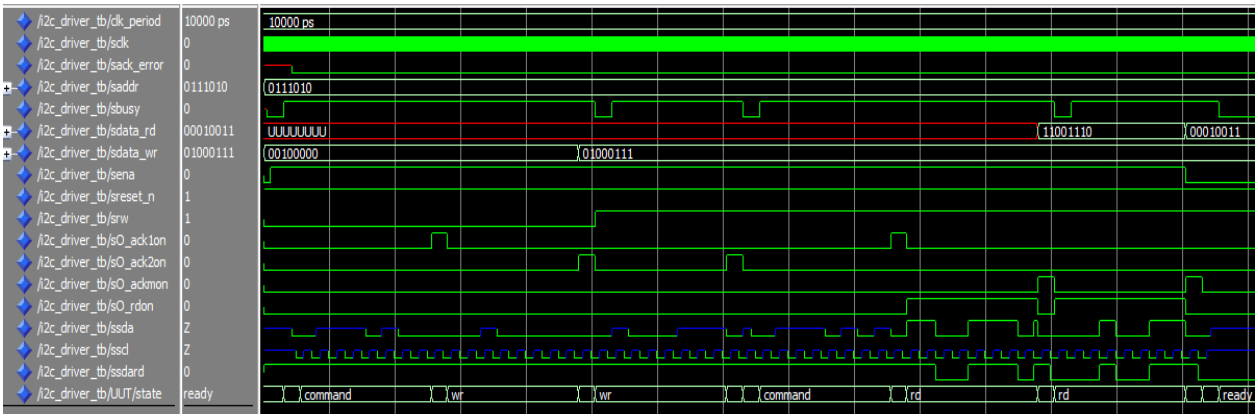

Figure 2.28: I2C Driver Simulation Result

The signals that are important to look at are the ssdard, ssda, sscl, sdata_rd, sdata_rw, sena, srw, sO_ack1, sO_ack2 and the sO_ackmon. When we look at the sdata_wr, we see well the writing of register address. We also see that we receive a Master acknowledgments sO_ackmon at the end of each read states, Slave acknowledgments sO_ack1 and sO_ack2 after each writing state. We also can see the ssdard fake waveform generated.

2.2 Application

The application is the part of the project that will handle the accelerator sensor state machine. It will do the same thing than the test bench seen previously but with exact conditions to change from a state to another. So what we need to do is to follow up the right path in the state machine of the driver to, in a first part, set the accelerometer configuration and, in a second part, to write the register address (for X, Y and Z axis) from which we will read the data corresponding to a selected axis direction. The application will need to have the inputs and outputs as shown on Figure 2.29 below.

```
ENTITY Appli IS
   PORT(
        LEDR      : OUT   STD_LOGIC_VECTOR(9 DOWNTO 0);    --display when data received from the I2C_Driver
        clk       : IN    STD_LOGIC;                       --system clock
        ena       : OUT   STD_LOGIC;                       --latch in command
        addr      : OUT   STD_LOGIC_VECTOR(6 DOWNTO 0);    --address of target slave
        rw        : OUT   STD_LOGIC;                       --'0' is write, '1' is read
        data_wr   : OUT   STD_LOGIC_VECTOR(7 DOWNTO 0);    --data to write to slave
        data_rd   : IN    STD_LOGIC_VECTOR(7 DOWNTO 0);    --data read from slave
        reset_n   : IN    STD_LOGIC;                       --active low reset resetn(0)
        button    : IN    STD_LOGIC;                       --active high button(1)
        busy      : IN    STD_LOGIC;                       --indicates transaction in progress
        I_rdon    : IN    STD_LOGIC;                       --flag when reading from slave
        I_ack1on  : IN    STD_LOGIC;                       --flag for the first acknowledgement from slave
        I_ack2on  : IN    STD_LOGIC;                       --flag for the second acknowledgement from slave
        I_ackmon  : IN    STD_LOGIC);                      --flag for the acknowledgement from Materslave
END Appli;
```

Figure 2.29: Application's Inputs and Outputs

2.2.1 Counter

We have implemented a counter in our application with a button as sensitivity signal (Cf. Figure 2.30) of our process in order to allow the operator to select between incoming read X, Y and Z accelerations direction bits values we want to display on the leds for every button rising edge through the sSW signal (Cf. Figure 2.31).

```
PROCESS(button, reset_n)
   VARIABLE   cnt       : natural range 0 to 3;
BEGIN
   IF(rising_edge(button)) then
      IF reset_n = '0' then
         -- Reset the counter to 0
         cnt := 0;
      ELSIF sena = '1' then
         -- Increment the counter if counting is enabled
         cnt := (cnt + 1) mod 4;
      END IF;
   END IF;

   -- Output the current count
   --vhdl type conversion integer standard_logic_vector
   sSW <= std_logic_vector(to_unsigned(cnt,2));
END PROCESS;
```

Figure 2.30: Counter with the button process

```
Selector1: LEDR(5 downto 0) <= sLEDR_Z(7 downto 2) when  sSW ="01" else
                               sLEDR_Y(7 downto 2) when  sSW ="10" else
                               sLEDR_X(7 downto 2) when  sSW ="11" else "000000";

           LEDR(6) <='0';

Selector2: LEDR(9 downto 7) <= "001" when sSW ="01" else
                               "010" when sSW ="10" else
                               "100" when sSW ="11" else "000";
```

Figure 2.31: The signal sSW controlling the selector

2.2.2 Reset

We also realize an asynchronous reset (cf. Figure 2.32). This is done by setting a condition that does not rely on the clock. We check the state of a bit named reset_n. If it is a 1, we set the ena (enable) to 0 and the state to "ready" which is the first state. Otherwise, we simply go to the state machine at each clock's rising edge.

```
PROCESS(clk, reset_n, state)

BEGIN
   IF(reset_n = '0') THEN                    --reset asserted
      sena <='0';                            --no command latch in
      state <= ready;
   ELSIF(clk'EVENT AND clk = '1') THEN
      CASE state IS
```

Figure 2.32: The Reset

2.2.3 Configuration Setting

To configure the accelerometer sensor we need, after the start condition, to write first as said above the device address "0011101b" with b the low significant bit (LSB) representing the Read/Write bit activation, during the configuration it is set at '0', then after the slave ACK we have to write the CTRL_REG1 "00100000" (20h) which is the slave control register address and finally after the second slave ACK received we will write the data output register "01000111" (47h) then we wait for the stop condition. In the below table the sensors registers addresses we used in this project.

Name	Type	Register Address		Default	Comment
		Hex	Binary		
Reserved (Do not modify)		00-0E			Reserved
Who_Am_I	r	0F	00 1111	00111011	Dummy register
Reserved (Do not modify)		10-1F			Reserved
Ctrl_Reg1	rw	20	10 0000	00000111	
Ctrl_Reg2	rw	21	10 0001	00000000	
Ctrl_Reg3	rw	22	10 0010	00000000	
HP_filter_reset	r	23	10 0011	dummy	Dummy register
Reserved (Do not modify)		24-26			Reserved
Status_Reg	r	27	10 0111	00000000	
--	r	28	10 1000		Not Used
Out_X	r	29	10 1001	output	
--	r	2A	10 1010		Not Used
Out_Y	r	2B	10 1011	output	
--	r	2C	10 1100		Not Used
Out_Z	r	2D	10 1101	output	

Figure 2.33: The LIS302DL Registers Address Map

2.2.4 Reading X, Y and Z detected acceleration

To read the acceleration detected for different axis, we can refer to the register address map in Figure 2.33 because to read X, Y and Z output we need to write their respective address register and this each time after the slave register address's ACK. The process of reading will be realized by the states named respectively, "*state_writeread1_0x3b*", "*state_writeread2_0x3b*" and "*state_writeread3_0x3b*".

After accelerometer configuration setting stop condition we set a loop routine process starting from the state "*state_writeread1*" as follow: sena sets to '1' and R/W to '0' we write the slave address, after first slave ACK, we write the Output_Z register address "00101101" (2Dh) while waiting to receive the second slave ACK we keep sena to '1' and switch R/W to '1' for a restart and then reading mode activated, the slave address is sent and after slave ACK, we go to the state "*state_writeread1_0x3b*" to read the Output_Z (data_rd) value available and then after master ACK ("*I_ackmon"*) we set the sena to '0' for stop condition then we go to the "*state_writeread2"* to perform the same routine than "*state_writeread1*" and this time to write the Output_Y register address "00101011" (2Bh) and after the stop condition of this state we will go to "state_writeread3" to do again the same routine but this time we will write the Output_X register address "00101001" (29h) and after the stop condition we will come again to "*state_writeread1*" state and loop the process.

```
WHEN state_writeread1 =>
      sena    <='1';
      rw      <='0';
      data_wr <= "00101101";       -- x2Dh config Z-axis output register
   IF(I_ack1on = '1') THEN
      state <= state_writeread1_waitsack2on ;
   END IF;
WHEN state_writeread1_waitsack2on =>
      sena    <='1';
      rw      <='1';
   IF(I_ack2on = '1') THEN
      state <= state_writeread1_0x3b ;
   END IF;
WHEN state_writeread1_0x3b =>
      rw      <='1';
   IF(I_ackmon = '1') THEN
      sLEDR_Z <= data_rd;
      state   <= wait_stopR1;
   END IF;
WHEN wait_stopR1 =>
      sena    <='0';
   IF(I_ackmon = '0') THEN
      state <= state_rdyR1;
   END IF;
WHEN state_rdyR1 =>
      sena    <='0';
      rw      <='0';
   IF(busy = '0') THEN
      state <= state_writeread2 ;
   END IF;
```

.

Figure 2.34: The routine code to read Z

```
WHEN state_writeread2 =>
      sena    <='1';
      rw      <='0';
      data_wr <= "00101011";          -- x2Bh config Y-axis output register
   IF(I_ack1on = '1') THEN
      state <= state_writeread2_waitsack2on ;
   END IF;
WHEN state_writeread2_waitsack2on =>
      sena    <='1';
      rw      <='1';
   IF(I_ack2on = '1') THEN
      state <= state_writeread2_0x3b ;
   END IF;
WHEN state_writeread2_0x3b =>
      rw      <='1';
   IF(I_ackmon = '1') THEN
      sLEDR_Y <= data_rd;
      state <= wait_stopR2;
   END IF;
WHEN wait_stopR2 =>
      sena    <='0';
   IF(I_ackmon = '0') THEN
      state <= state_rdyR2;
   END IF;
WHEN state_rdyR2 =>
      sena    <='0';
   IF(busy = '0') THEN
      state <= state_writeread3 ;
   END IF;
WHEN state_writeread3 =>
```

Figure 2.35: The routine code to read Y

```
    WHEN state_writeread3 =>
          sena    <='1';
          rw      <='0';
          data_wr <= "00101001";          -- x29h config X-axis output register
       IF(I_ack1on = '1') THEN
          state <= state_writeread3_waitsack2on ;
       END IF;
    WHEN state_writeread3_waitsack2on =>
          sena    <='1';
          rw      <='1';
       IF(I_ack2on = '1') THEN
          state <= state_writeread3_0x3b ;
       END IF;
    WHEN state_writeread3_0x3b =>
          rw      <='1';
       IF(I_ackmon = '1') THEN
          sLEDR_X <= data_rd;
          state <= wait_stopR3;
       END IF;
    WHEN wait_stopR3 =>
          sena    <='0';
       IF(I_ackmon = '0') THEN
          state <= state_rdyR3;
       END IF;
    WHEN state_rdyR3 =>
          sena    <='0';
          rw      <='0';
       IF(busy = '0') THEN
          state <= state_writeread1 ;
       END IF;
    WHEN others =>
       sena    <='0';
       state <= ready;                  --go to idle state
END CASE;
```

Figure 2.36: The routine code to read X

2.2.5 Simulation

We can now create a test bench to check if our application is working. To do so, we perform the steps mentioned in section 2.1.2. In this application test bench the signals we control are the acknowledgements, the busy and I_rdon. The routine, that is the major concern here, is shown on Figure 2.37. This routine can of course, be adapted. We particularly focused on making the state machine work normally by changing from one state to another in the right order. As earlier, we also simulate a clock and a reset in the beginning. The results of the simulations are shown on Figure 2.38 and Figure 2.39. We can see that they are satisfying because the states are changing in the right order.

```
144     stimulus : PROCESS
145
146      BEGIN
147           sdata_rd <= "00000000";
148           wait for clk_period;
149           busy <='1';
150           wait for clk_period;
151           wait for clk_period;
152           busy <='0';
153           wait for clk_period;
154           I_ack1on <='1';
155           wait for clk_period;
156           I_ack1on <= '0';
157           wait for clk_period;
158           I_ack2on <='1';
159           wait for clk_period;
160           I_ack2on <='0';
161           wait for clk_period;
162           I_ack2on <='1';
163           wait for clk_period;
164           I_ack2on <= '0';
165           wait for clk_period;
166           I_ack1on <='1';
167           wait for clk_period;
168           I_ack1on <='0';
169           wait for clk_period;
170           I_ack2on <='1';
171           wait for clk_period;
172           I_ack2on <='0';
173           wait for clk_period;
```

```
174           I_rdon <= '1';
175           wait for clk_period;
176           sdata_rd <= "10100100";
177           sLEDR_Z <= sdata_rd;
178           I_ackmon <= '1';
179           button <= '1';
180           I_rdon <= '0';
181           wait for clk_period;
182           I_ackmon <='0';
183           button <= '0';
184           wait for clk_period;
185           I_ack1on <='1';
186           wait for clk_period;
187           I_ack1on <='0';
188           wait for clk_period;
189           I_ack2on <='1';
190           wait for clk_period;
191           I_ack2on <='0';
192           wait for clk_period;
193           I_rdon <= '1';
194           wait for clk_period;
195           sdata_rd <= "00111100";
196           sLEDR_Y <= sdata_rd;
197           I_ackmon <='1';
198           button <= '1';
199           I_rdon <= '0';
200           wait for clk_period;
201           I_ackmon <='0';
202           button <= '0';
203           wait for clk_period;
204           I_ack1on <='1';
205           wait for clk_period;
206           I_ack1on <='0';
207           wait for clk_period;
208           I_ack2on <='1';
209           wait for clk_period;
210           I_ack2on <='0';
211           wait for clk_period;
212           I_rdon <= '1';
213           wait for clk_period;
214           sdata_rd <= "00100111";
215           sLEDR_X <= sdata_rd;
216           I_ackmon <='1';
217           button <= '1';
218           I_rdon <= '0';
219           wait for clk_period;
220           I_ackmon <='0';
221           button <= '0';
222           wait for clk_period;
223      END PROCESS stimulus;
```

Figure 2.37: Test bench Routine

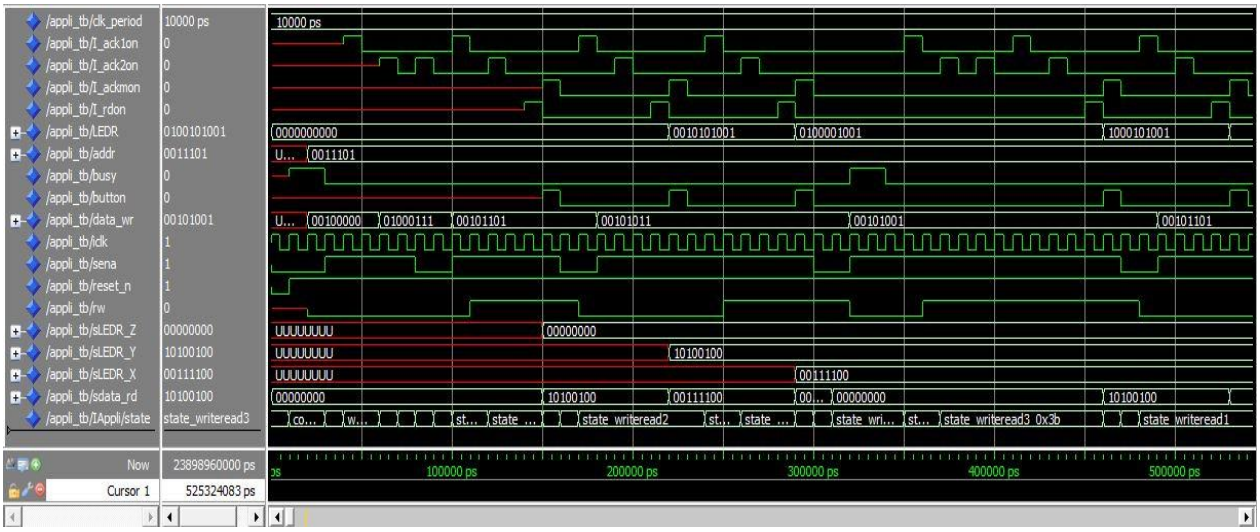

Figure 2.38: Application test bench simulation from 0s to 50µs

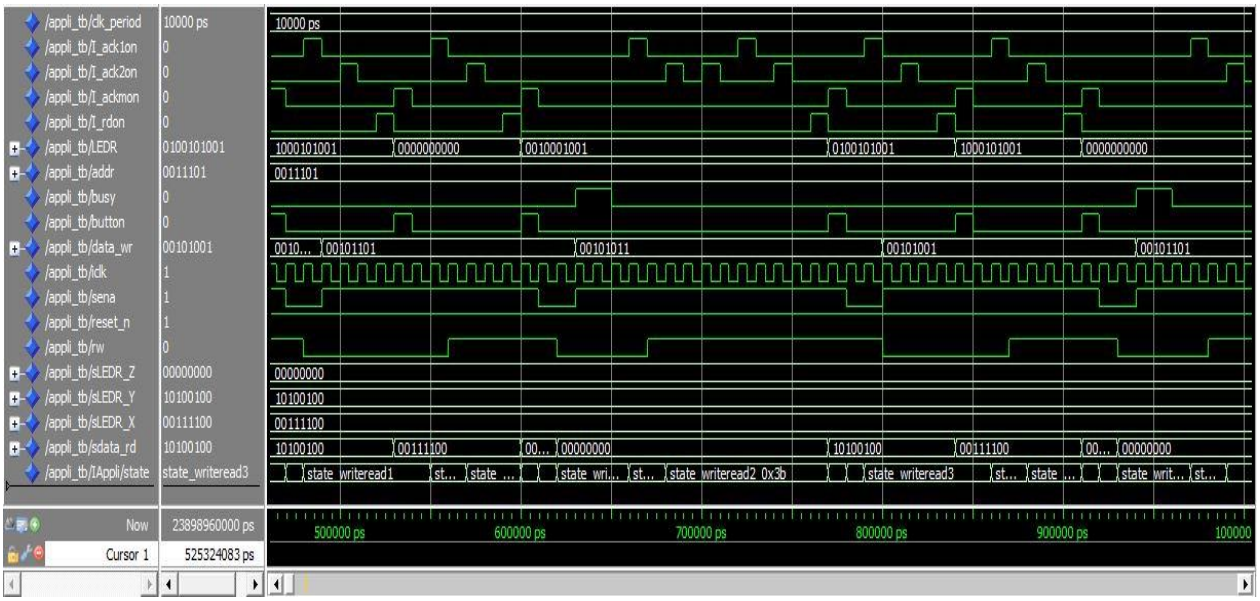

Figure 2.39: Application test bench simulation from 50µs to 100µs

2.3 I2C Driver and Application Assembly

2.3.1 Working of the assembly

The last and main ".vhd" file will be the assembly of the driver and the application. This is not difficult. Indeed, you only need to declare and instantiate the I2C "driver" component and instantiate the "application" component then make connections between them. The inputs and outputs need to be as shown on Figure 2.40. Indeed, our main program needs to have the clock, the button and the reset as inputs and the I2C ports SDA and SCL as inputs and outputs. The leds on which we will show the different X, Y and Z axis values are also outputs. An important thing is that we need to connect all the signals related to the states on common signals from the assembly.

```
ENTITY Appli_Driver IS
    PORT(
    clk       : IN     STD_LOGIC;                     --system clock
    reset_n   : IN     STD_LOGIC;                     --active low reset
    button    : IN     STD_logic;                     --active low buttor
    O_rdon    : OUT    STD_LOGIC;                     --flag when reading from  slave
    O_ack1on  : OUT    STD_LOGIC;                     --flag when acknowledgement from slave
    O_ack2on  : OUT    STD_LOGIC;
    O_ackmon  : OUT    STD_LOGIC;                     --flag when acknowledgement from master
    LEDR      : OUT    STD_LOGIC_VECTOR(9 DOWNTO 0); --display when data received from the I2C_Driver
    sda       : INOUT  STD_LOGIC;                     --serial data output of i2c bus
    scl       : INOUT  STD_LOGIC);                    --serial clock output of i2c bus
END Appli_Driver;
```

Figure 2.40: Inputs and outputs of the assembly

2.3.2 Simulation of the assembly

As previously, we need to test the assembly prior sending it on the card. To do so, we realize a test bench in which we control the SDA by making fake data when we are on the read state (Cf. Figure 2.41). We get the simulation results as shown on Figure 2.42 and 2.43. These results are fine because we can see that the states are changing in the right order, the leds show the value "sent" by the accelerometer sensor and the acknowledgement error (at the bottom of the simulation) is always 0, which means there are no errors. As the simulation works correctly, we can now consider sending it to the card.

```
ssda <= ssdard  WHEN srdon = '1'   ELSE
                '0' WHEN sack1on = '1' ELSE
                '0' WHEN sack2on = '1' ELSE
                '0' WHEN sackmon = '1' ELSE 'Z';


SDA_process:|   process

         constant din : STD_LOGIC_VECTOR(0 to 15):= "1100111010001001";
         variable idx : natural range 0 to 15 := 0;

      begin
         ssdard <= din(idx);
         wait until srdon = '1'  ;
         while srdon = '1' loop
            wait until sscl = '0';
            if (idx +1) = 16  then
               idx := 0;
            else
               idx := idx +1;
            end if;
            ssdard <= din(idx);
         end loop;
      end process;
```

Figure 2.41: Control of the SDA

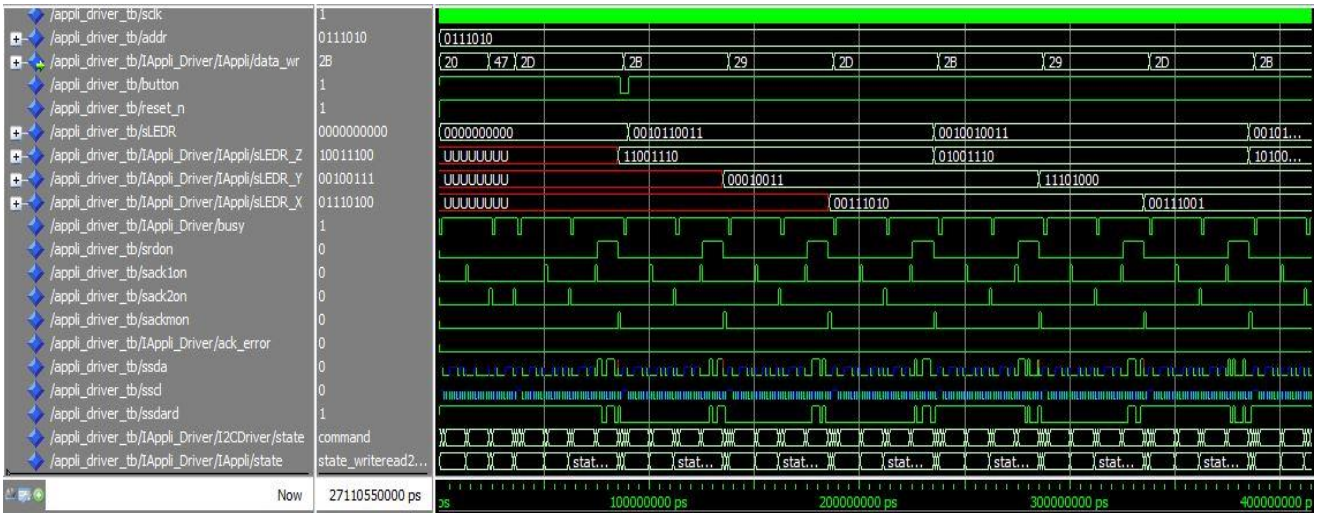

Figure 2.42: Assembly test bench simulation from 0s to 0.4s

Figure 2.43: Assembly test bench simulation from 0.4s to 0.8s

2.4 Pin assignment

The names of the pins we have on the software aren't always relevant. It's sometimes difficult to see the corresponding function of the pin. To solve this little problem we had to find a new file which gives another name to the pins (easier and corresponding to their functions). To do so:

- Find and download a .qsf file for DE1 [6]
- Go to the tab assignments then to import assignments
- Import the file in the project
- If you want to see what it gives, you can go in Assignments editor and you will see all names, the old ones ("Value" column) and the new ones ("To" column).

Giving a new name to pins will help us for the port mapping.

2.5 Port Mapping

Once you have tested the driver, the application and the driver with the application, you are almost ready to make the project work. You just have to create a last ".vhd" file as before to associate the PORT to the pins. It's also the code you will have to compile to put on the chip card. Here's a part of the code (Cf. Figure 2.44) We can see that we associate the clock to CLOCK-50 and the reset to the button KEY(0) and the button signal to KEY(1). Then we have some signals we don't associate, that's why we can see the word open. We set the scl to GPIO-1(1) and sda to GPIO-1(3). We can see these pins on the picture below (Cf. Figure 2.45). Before flashing the code into the card, you must connect the accelerometer sensor to the card based on Figures 2.45 and 2.46. To connect the card and the accelerometer sensor:

- Plug the SDA on the GPIO-1(1)
- Plug the SCL on the GPIO-1(3)
- Plug VDD on VCC3V3 of GPIO-1
- Plug GND on GND of GPIO-1

```
BEGIN

IAppli_Driver: entity work.appli_driver(structure)

    PORT MAP(
            clk         => CLOCK_50,
            reset_n     => KEY(0),
            button      => KEY(1),
            o_rdon      => open,
            o_ack1on    => open,
            o_ack2on    => open,
            o_ackmon    => open,
            LEDR        => LEDR,
            sda         => GPIO_1(1),
            scl         => GPIO_1(3)
           );
```

Figure 2.44: Pin-Port Association

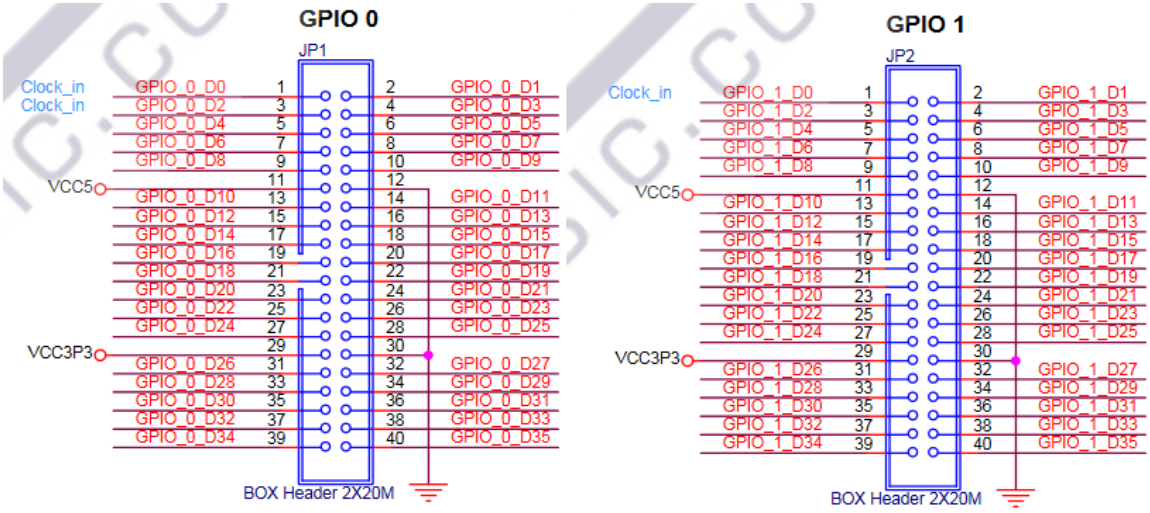

Figure 2.45: Pin GPIO on the DE1-SoC

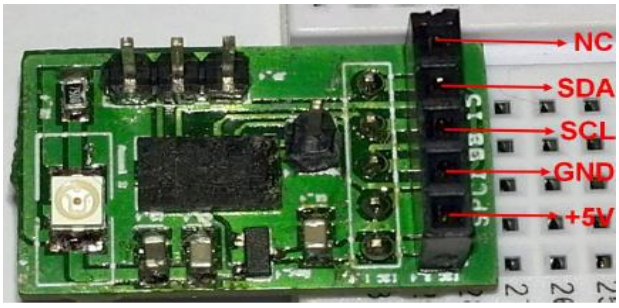

Figure 2.46: LIS3LV02DL accelerometer sensor pins [1]

You have to pay attention that SDA and SCL are in common collector so you have to polarize them. For that, you have to add two resistors RP of 4,7k Ohm each: one between VDD and SCL and one between VDD and SDA as show in the Figure 2.47.

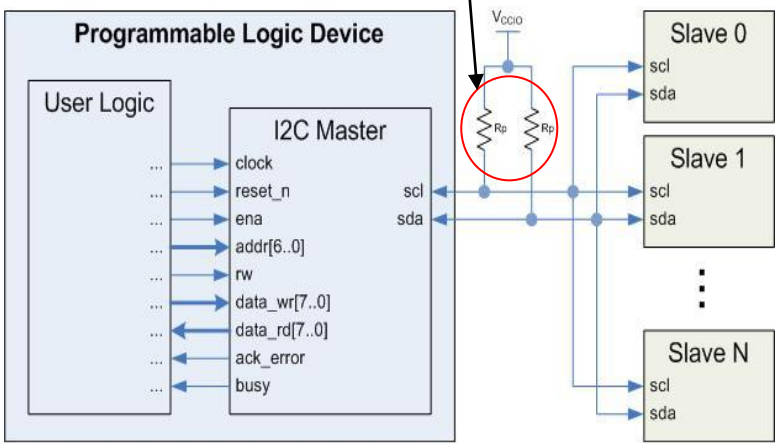

Figure 2.47: Implementation Example

Hereunder a picture that illustrate this assembly (Cf. Figure 2.48):

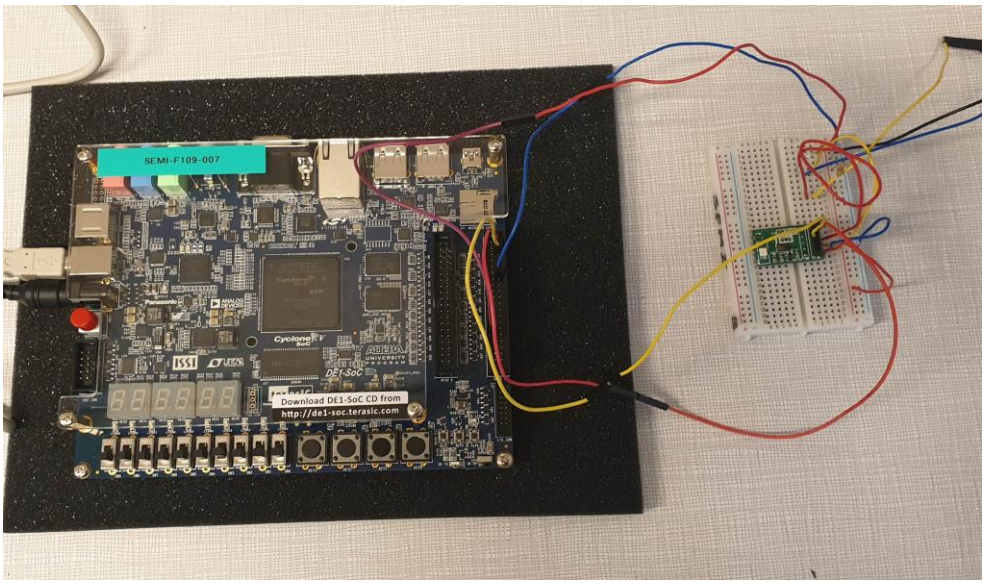

Figure 2.48: Assembly of the accelerometer sensor and the card

Once you have done the assembly, you are ready to upload the code on the DE1-SoC. To do that, follow the below steps:

- Compile the last code you wrote
- Go in the tab Program Device
- Switch the card on. When it appears, select it
- Select the file you can find in the output files
- You can now click start

Also for your information, configuring the FPGA in JTAG Mode is more detailed in [7], it explain also how to convert the ".sof" file into ".jic" for JTAG mode configuration. SOF stands for "SRAM Object File" and it is a binary file that comes from the compilation of the driver and the application. You can usually find this file in the folder "output_files".

Conclusion

During this project we learned to drive the LIS3LV02DL accelerometer sensor on a FPGA. For that, we familiarized with the I2C bus protocol to implement our codes properly. We have done different simulation for the driver, the application and the combination of the two to see their behavior and to check if what we did was right. After our simulations we concluded that we did things well. Everything worked properly. So we flashed the code on the card. This way, when we move the sensor to the direction (X, Y, Z) of a detectable acceleration, we could see the acquisition value for one direction changing with the leds. The button allow us to choose the axis direction channel from Z, Y to X. We looked the I2C tram on an oscilloscope and we saw well that we write three times before reading axis direction from the sensor.

Quartus signal tap analyzer was used as well to analyze all important signal and this tool help us to see the LEDR signal value in hexadecimal corresponding to the direction acceleration displayed on led of the DE1-SoC board.

References

[1] Free download URL: http://fpgasoftware.intel.com/?edition=lite.

[2] I2C Master (VHDL): https://www.digikey.com/eewiki/pages/viewpage.action?pageId=10125324 by Scott Larson.

[3] AN2335 Application note: LIS302DL.

[4] Capteur accéléromètre LIS3LV02DL en I2C – SEMI.

[5] UM10204, I2C-bus specification and user manual, NXP Semiconductors N.V..

[6] http://www.ecs.umass.edu/ece354/ECE354HomePageFiles/Labs_files/DE1_SoC.qsf.

[7] http://www.ee.ic.ac.uk/pcheung/teaching/ee2_digital/DE1-SoC_User_manual.pdf.

[8] I2C Introduction au bus I2C, Camille Diou URL: https://www.les electroniciens.com/sites/default/files/cours/cours_i2c.pdf.