UMONS
Université de Mons

Faculté Polytechnique

Hardware and software platforms

Tutorial :How to drive ab accelometer with an altera cyclone v

Lumba Nsimba Minor
Leumalieu Nguenkam Christian
Master en Sciences de l'Ingénieur, Orientation Électricité

POLYTECH MONS

Sous la direction de :
Prof Carlos VALDERRAMA

Année académique 2020-2021

ACADÉMIE UNIVERSITAIRE WALLONIE-BRUXELLES

TABLE DES MATIÈRES

TABLE DES FIGURES

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.
This tutorial has two parts. The first part chapiter 1 for helping you to understand the I2C serial bus 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 Chapiter 2 will drive you step by step to the end of the project.

We will separate the implementation of the driver, the application and the combination of the two. The simulation of these three part is individually in order to see if the code is correct.We will explain all in the chapiter 2. 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 detectable acceleration of axis direction (X, Y, and Z) in bytes. We will not read it in decimal numbers. Indeed, we will retrieve and display the detectable acceleration (X, Y, and Z) axis values of the sensor on the LED's board.

CHAPITRE 1

I2C SERIAL BUS INTERFACE AND ACCELEROMETER

In this chapter, we will present an overview on the I2C bus protocol and features ; and we will study the way the I2C is used in the $CPIC$ program.

1.1 the I2C bus

The Inter-Integrated Circuit, abbreviated as I2C is a serial bus short distance communication protocol developed that permits various electronic components to communicate with only three connections :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.

We can take the control of the bus, the SDA and the SCL have to be equal to 1. For data transmission, we have to monitor two conditions :

— Start condition : SCL at one and SDA pass to zero

— Stop condition : SCL and SDA at one

When the bus is free, we will take the control of it, the circuit becomes the bus's master : who generates the clock signal. This is the vieuw of our accelerometer we will use for this project.

Capteur accéléromètre LIS3LV02DL en I2C – SEMI

Capteurs SEMI :

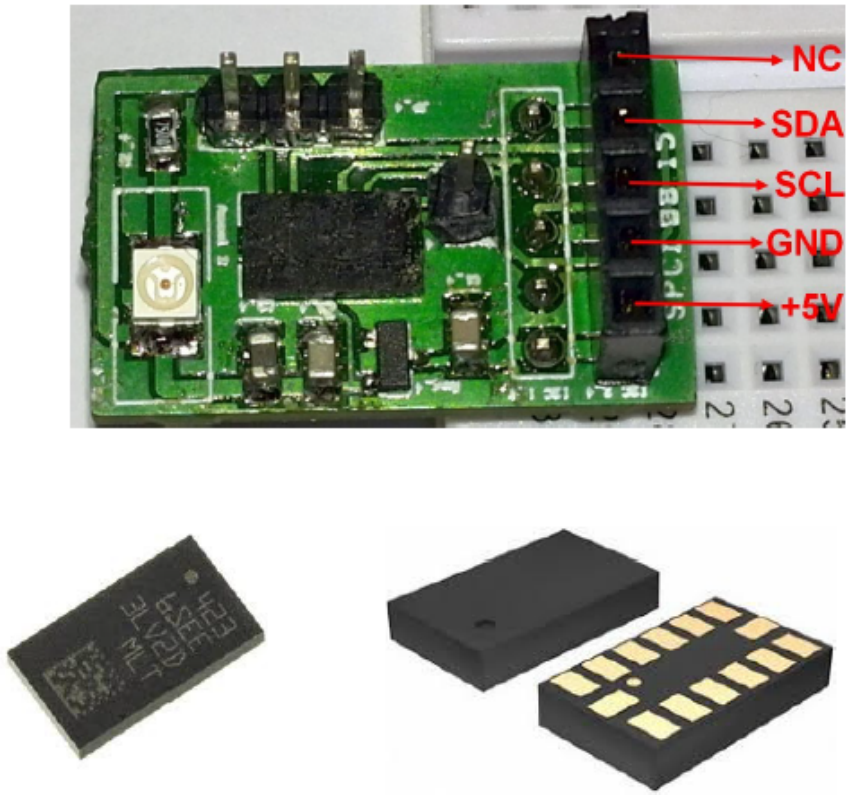
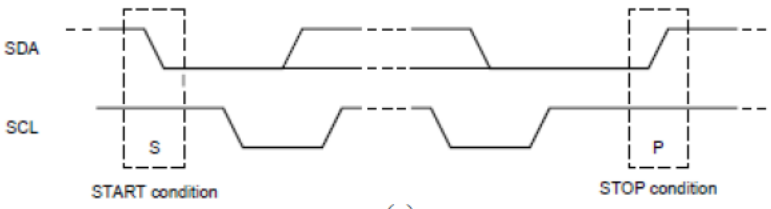
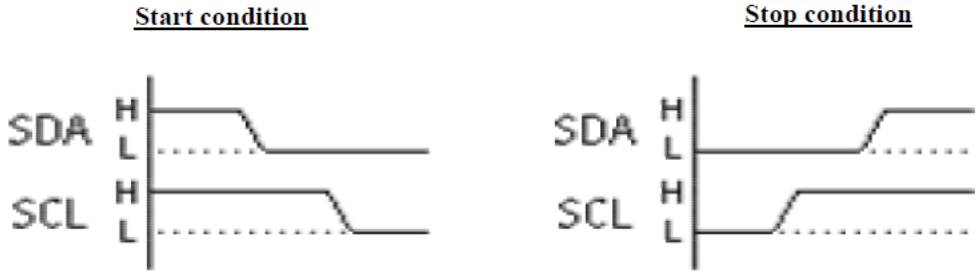


FIGURE 1.1 – Accelerometer



(a)



(b)

FIGURE 1.2 – Start condition and Stop condition

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 (ACK). 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.

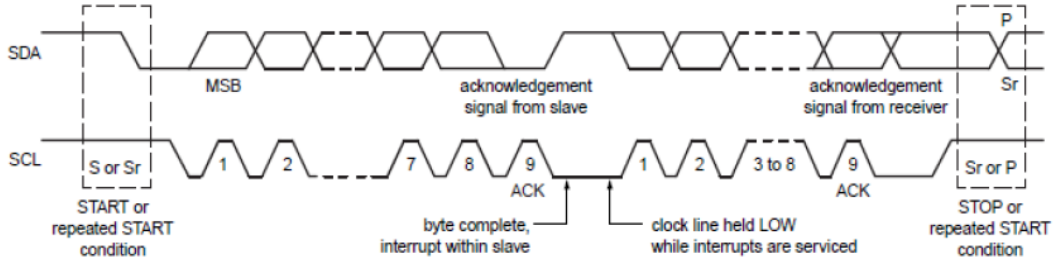


FIGURE 1.3 – Data transfer

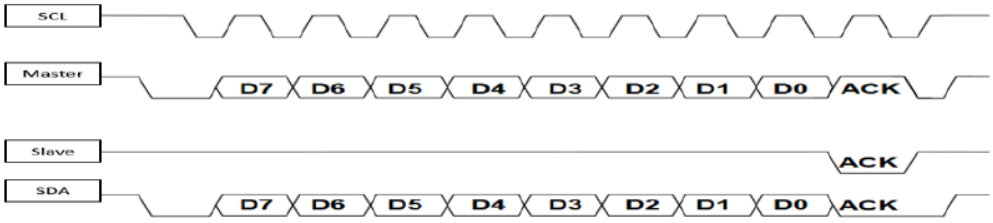


FIGURE 1.4 – Byte transmission

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.

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, it sends the data to the SDA. Eventually, the master sets the ACK to 0 to continue the reading and to 1 to stop it.

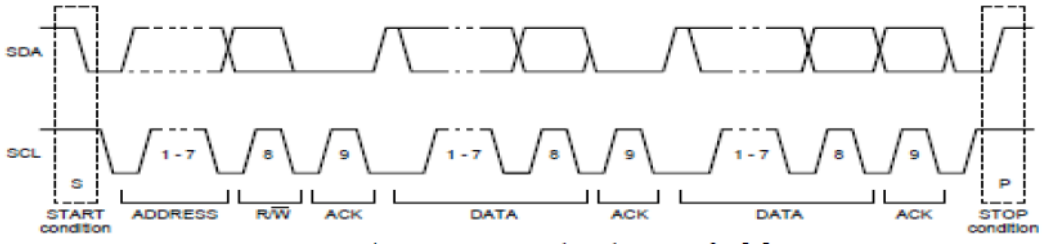


FIGURE 1.5 – Start condition and Stop condition

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.

```
//-------------Ecriture  I2C------------------
void ecr_i2c(byte device, byte address, byte data) {
   i2c_start();
   i2c_write(device);//device=adresse du boitier
   i2c_write(address);//address=adresse du registre qui se trouve dans le boitier
   i2c_write(data);//data=infos que l'on vient stocker
   i2c_stop();//indique que l'on a plus rien à faire au niveau du bus
}
```

FIGURE 1.6 – Writing

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 to be read.

```
//-------------Lecture  I2C------------------
signed byte lec_i2c(byte device, byte address) {
   signed BYTE data;

   i2c_start();
   i2c_write(device);//device=adresse du device (accéléromètre)
   i2c_write(address);//address=adresse du registre qui se trouve dans le device
   i2c_start();
   i2c_write(device | 1);//bit le + à droite mis à 1 pour qu'on puisse écrire sur le bus (signale au device qu'il peut répondre)
   data=i2c_read(0);
   i2c_stop();//indique que l'on a plus rien à faire au niveau du bus
   return(data);
}
```

FIGURE 1.7 – reading

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 data to be read.

```
//-------------Read of 2 bytes I2C-----------------
 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 1.8 – reading two byte

1.2.4 Accelerometer reading

In the code Here under, 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. The comments in the code are detailing what is the purpose of each line.

```
//-------------Lecture Accéléromètre 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);/*Ecriture dans le registre Ctrl_Reg1 (0x20), permet l'activation des 3 axes. ATTENTION full scale a 2g!!*/
        tmp=lec_i2c(ACC,0x27); /*Contient le registre Status_Reg 0x27 qui dit si il y a des nouvelles infos sur l'axe x,y,z, */


        if ((tmp&0x0C)==0x0C) axe_z=lec_i2c(ACC,0x2d);/*Si des nouvelles données sont présentes sur l'axe z, on lit*/
        if ((tmp&0x0A)==0x0A) axe_y=lec_i2c(ACC,0x2b);/*Si des nouvelles données sont présentes sur l'axe y, on lit*/
        if ((tmp&0x09)==0x09) axe_x=lec_i2c(ACC,0x29);/*Si des nouvelles données sont présentes sur l'axe x, on lit*/

        datax = axe_x*18; /*multiplication par 18 (car FS byte à 0, càd full scale à 2g) pour l'avoir en g, 1 correspond à 18mg*/
        datay = axe_y*18;
        dataz = axe_z*18;


        lcd_gotoxy(1,2);//Endroit du lcd où on va afficher (2e ligne) (x=colonne y=ligne)
        printf(lcd_char,"X%ld Y%ld Z%ld  ",datax,datay,dataz);//Affichage sur le lcd

}
```

FIGURE 1.9 – reading accelerometer

CHAPITRE 2

IMPLEMENTATION

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 Coding

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 all your project files will be saved to

— Launch Quartus - 64bits installed on your computed, if not yet installed download it using.

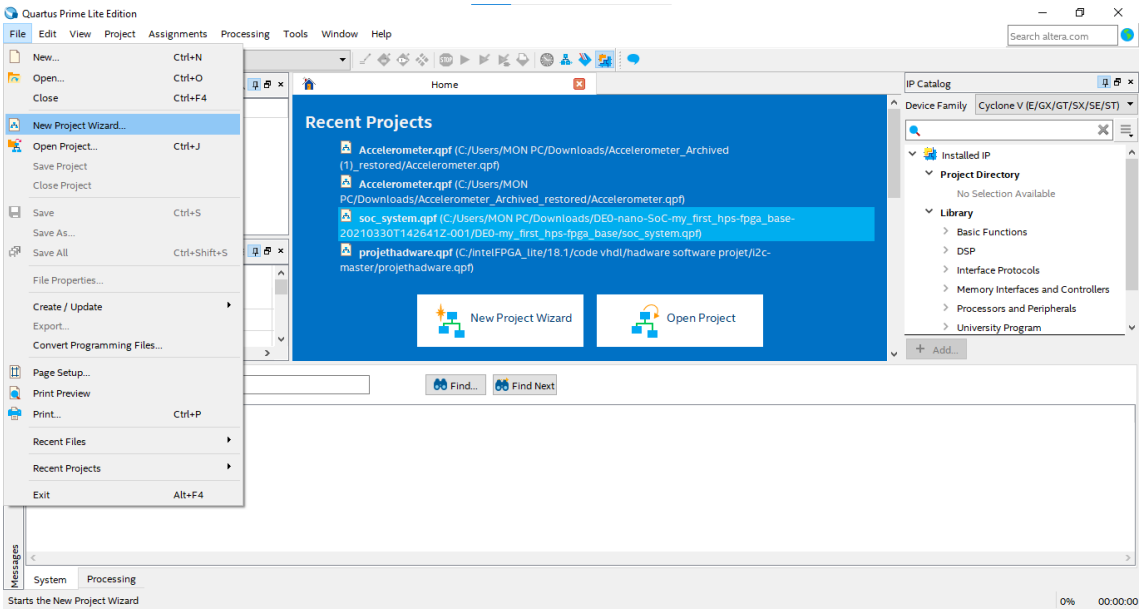


FIGURE 2.1 – opening of Quartus

— go in File , create a new project Wizard and click on next

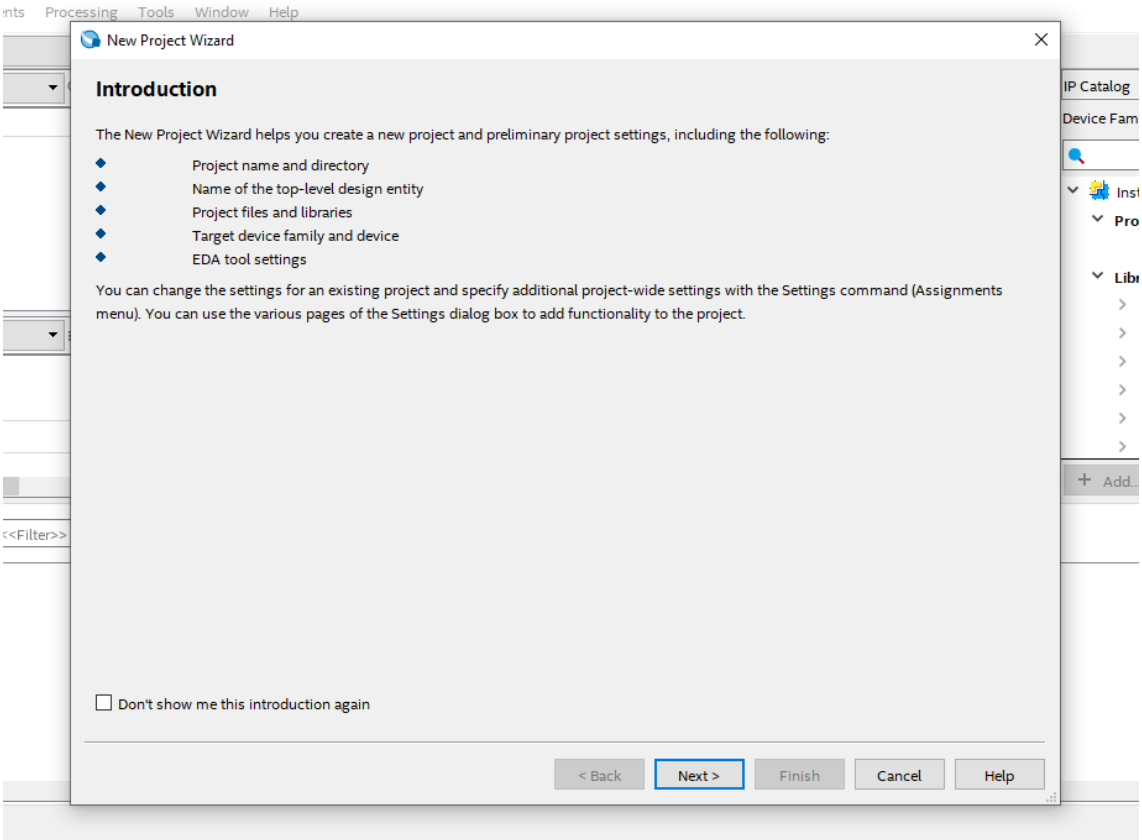


FIGURE 2.2 – new project

— Choose the directory of the project folder and give a name of your project.In this case,we give accelerometer

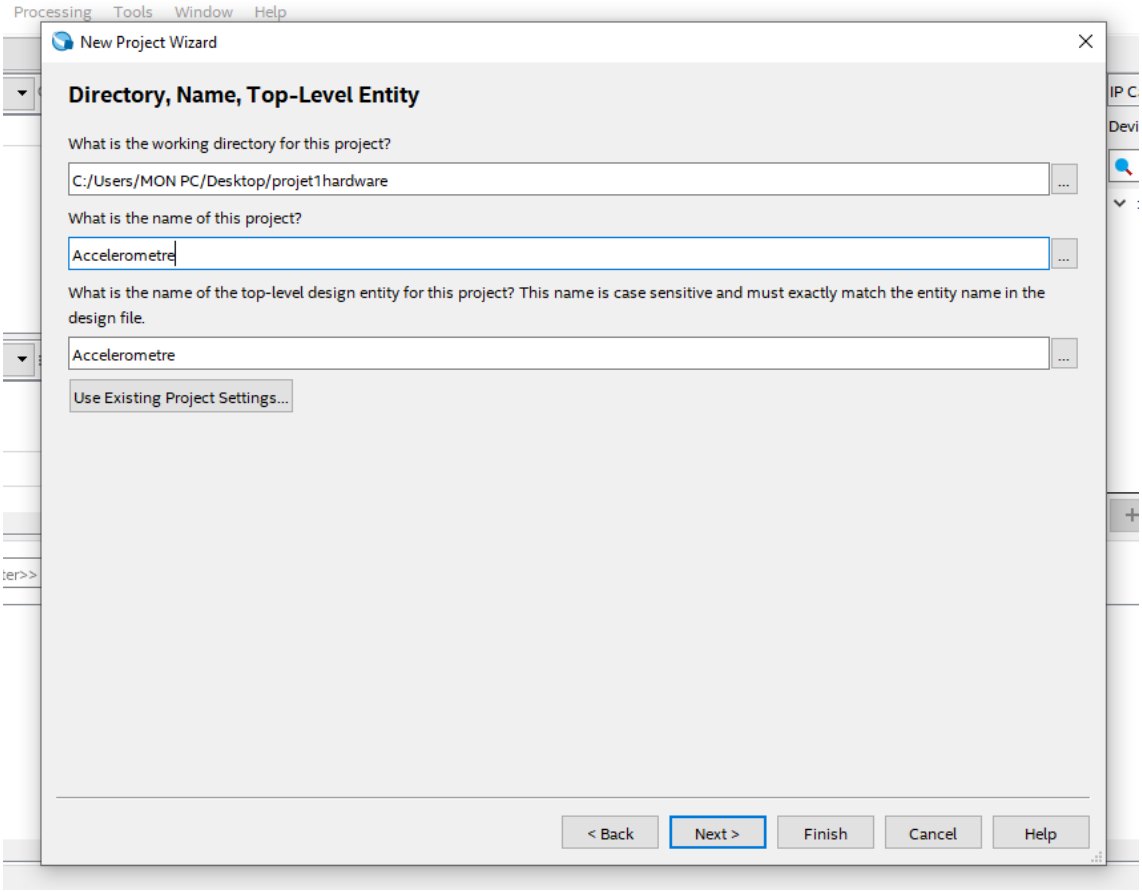


FIGURE 2.3 – project folder

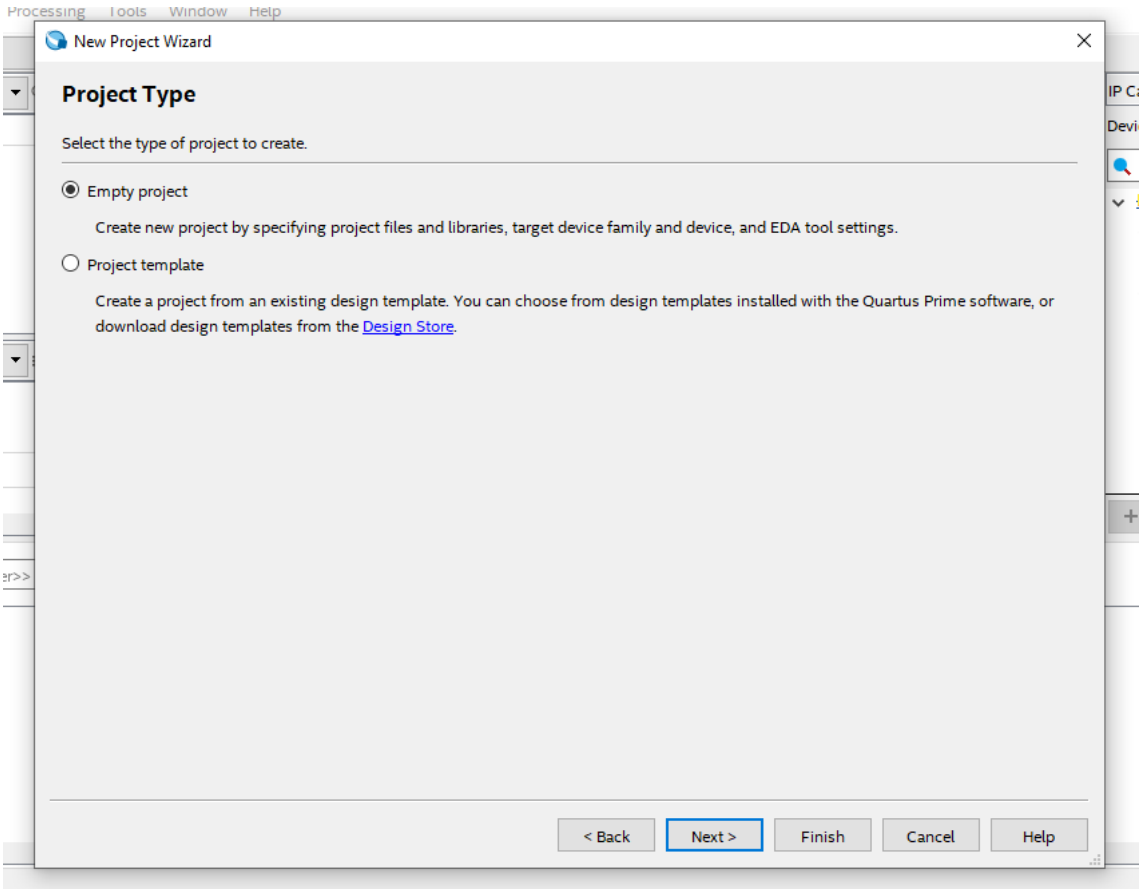


FIGURE 2.4 – reading

— Choose the devise family and Finish.

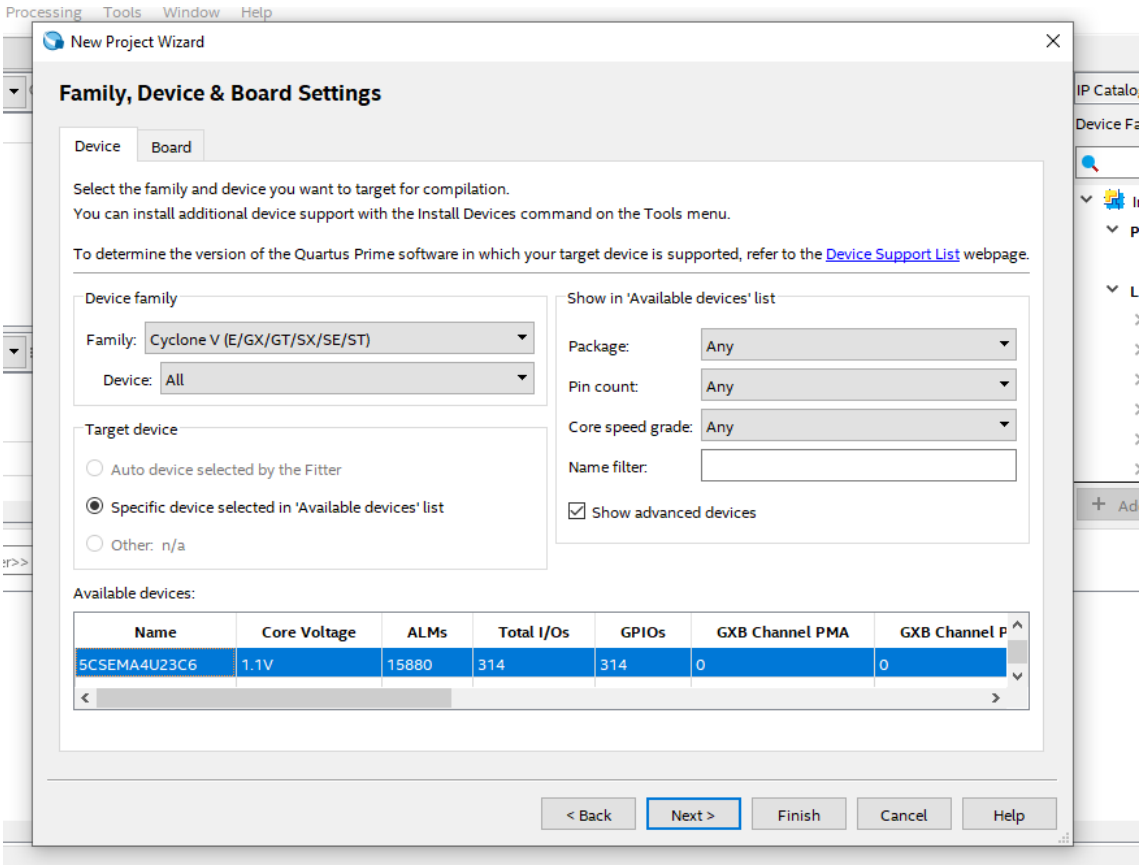


FIGURE 2.5 – family device

— create a new file in the project ,choose VHDL file

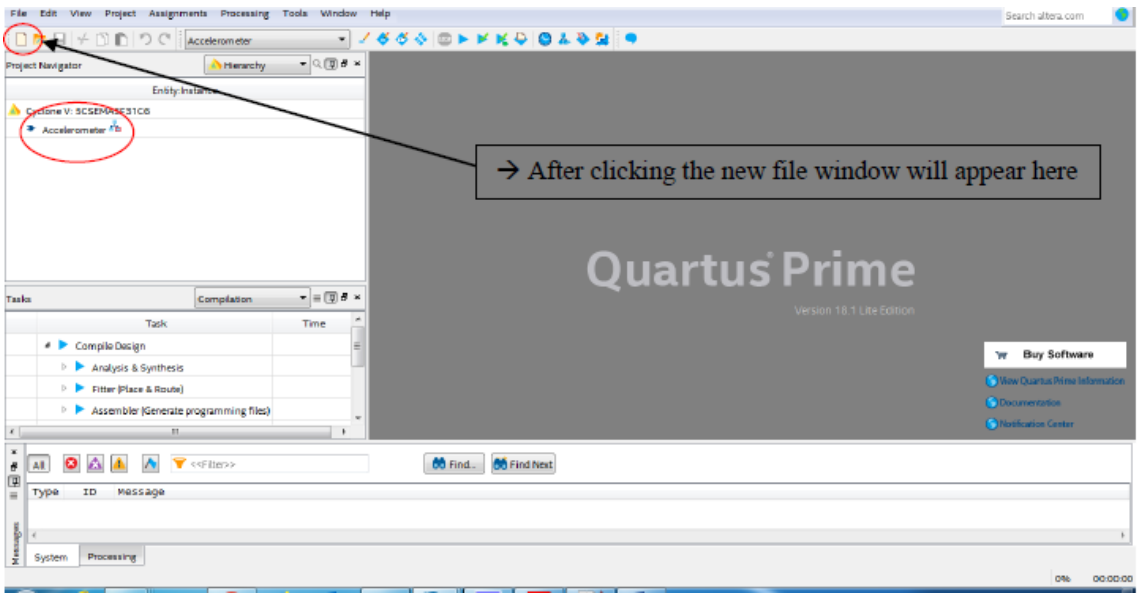


FIGURE 2.6 – Vhd code

— Paste the code of I2C inn the new window file.

— Save the project folder path and give of the name of I2C-Driver .

— Compile I2C-Driver.vhd file

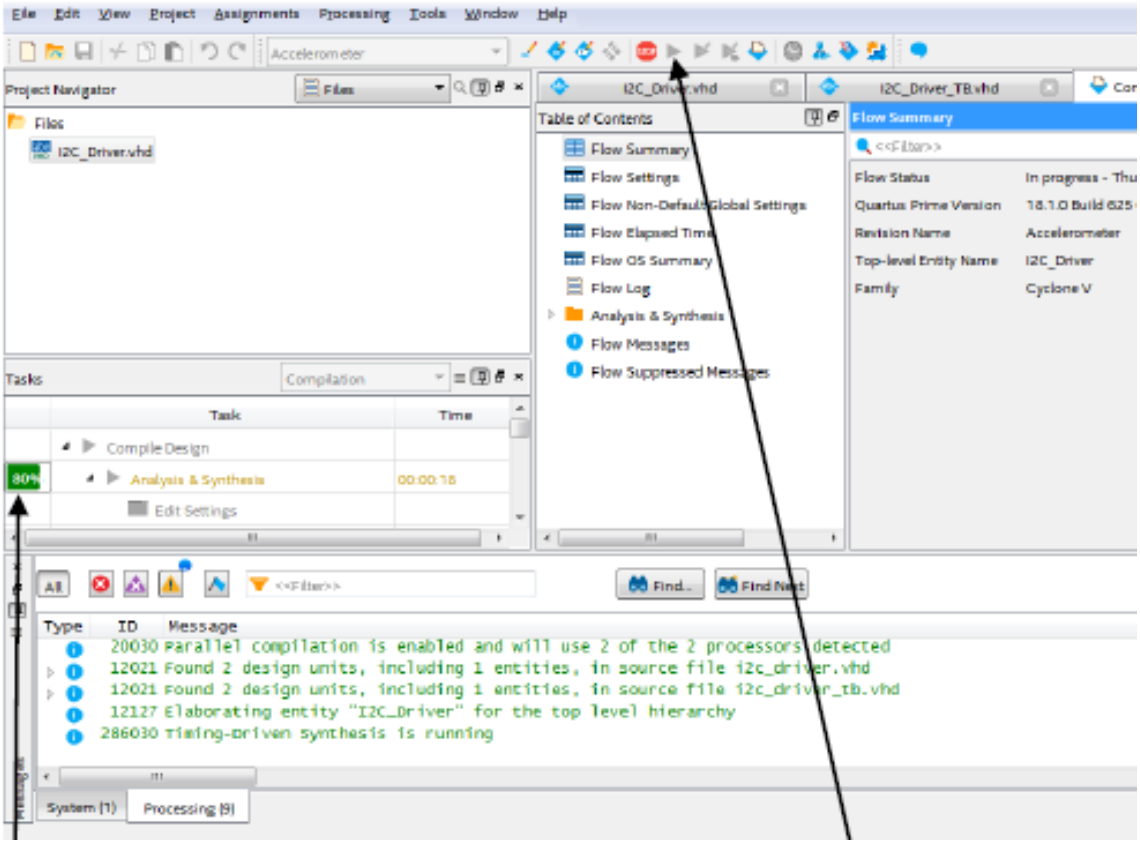


FIGURE 2.7 – Compile

Once compiled, Quartus shows us the I2C driver state machine but the clearer representation was the one on the website.We have an overview of this state machine with key signals conditions. Once you have understood this 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. It produces the signals to send to the system we want to put under test and check that the outputs of the circuit are correct. Here under is a part of the code you need to implement. 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 (U)

— 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.

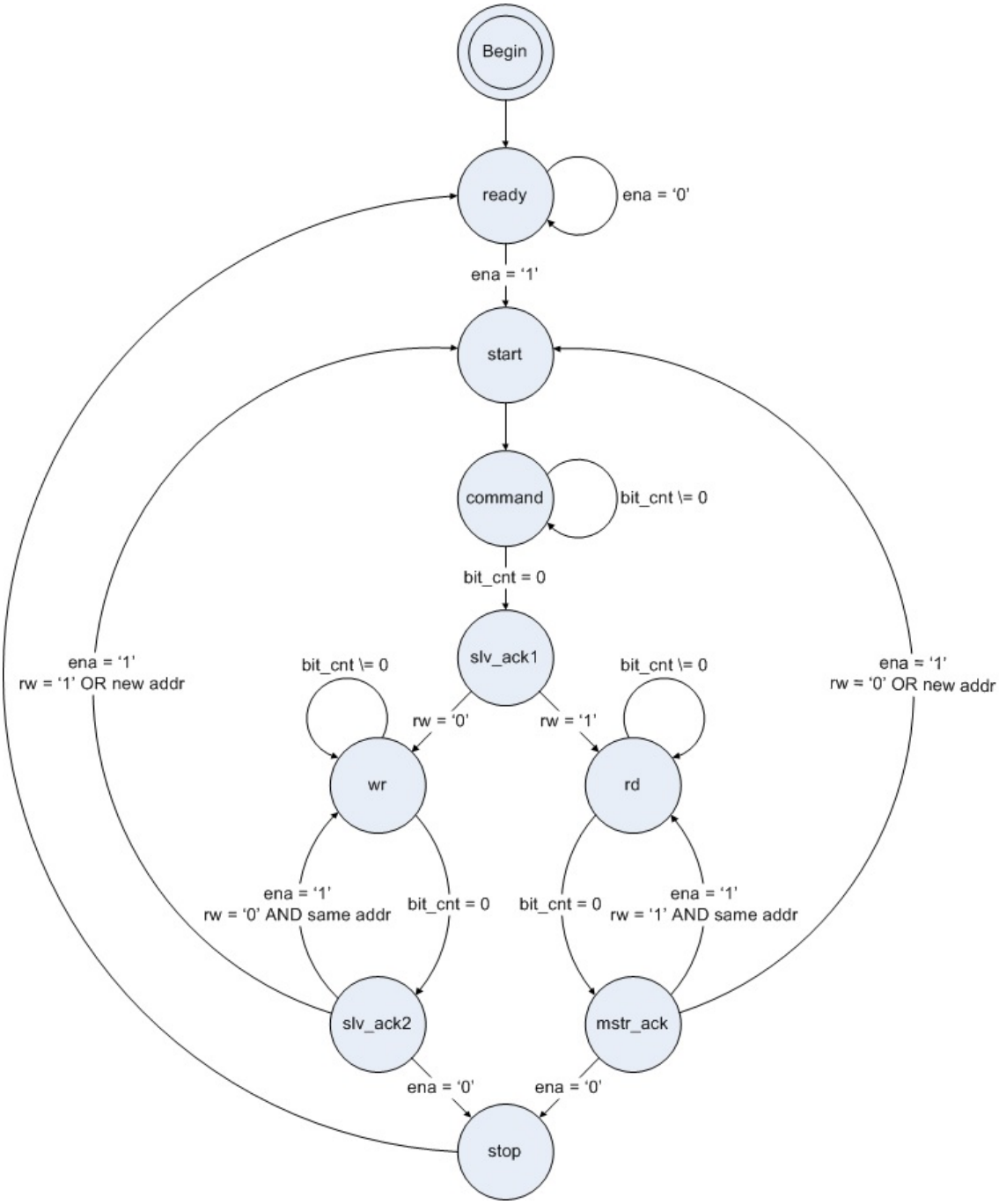


FIGURE 2.8 – state diagram

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

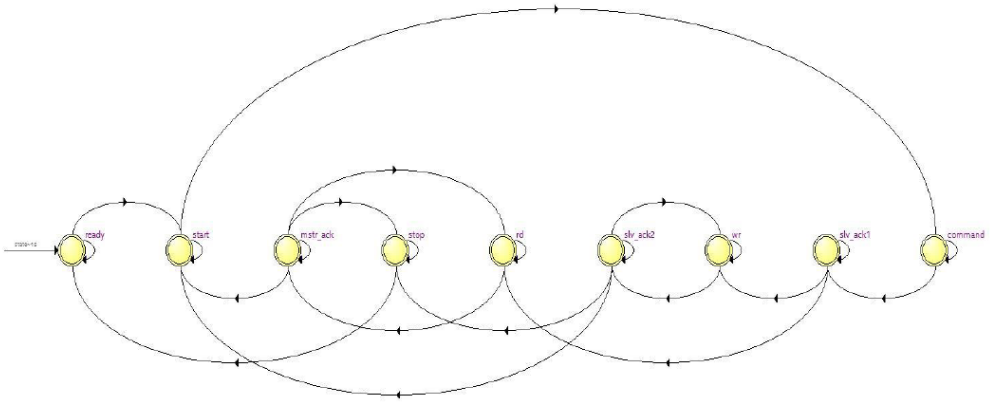


FIGURE 2.9 – Driver state machine

2.1.2 Simulation

Once we have done our Test Bench, we'll have to simulate it to see if the code of our driver works properly i.e. we don't get errors in our different signals (U->unknown values, X-> short-circuit). To do so, we have to follow below steps to set our test bench for simulation if it is not done :

- Set the I2CDriverTB file as Top – Level Entity

- Go into the tab Assignments then Settings and verify that in the section simulation, the compile test bench field is I2CDriverTB

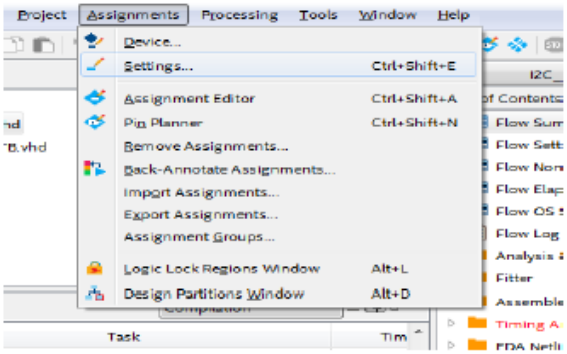


FIGURE 2.10 – Setting the test bench for simulation

If it is not set, like it is the case in Figure hereunder, follow the below steps to set it. Otherwise, if the compile test bench is I2CDriverTB, then go to the tab tools steps to simulate directly and Go to Test Benches.

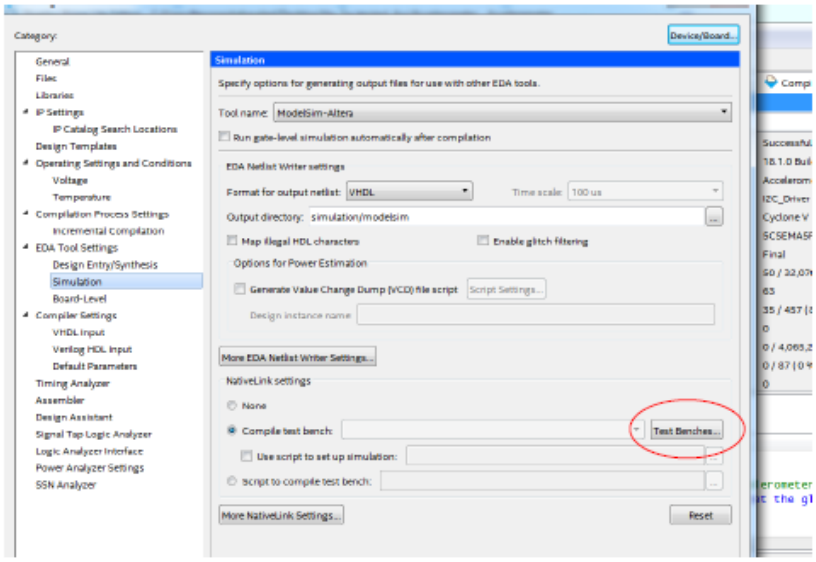


FIGURE 2.11 – Step 2 for the simulation of test bench

- Then click on new->

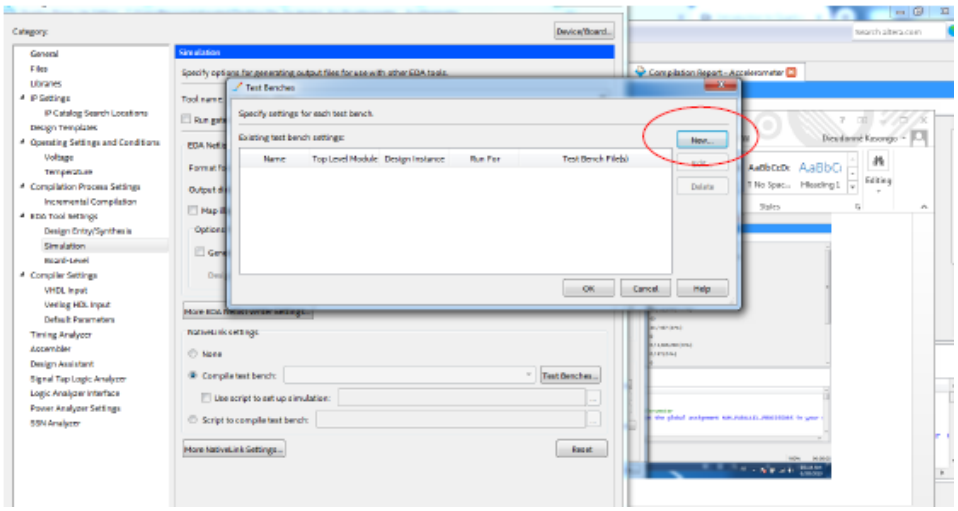


FIGURE 2.12 – Step 3 for the simulation of test bench

- After that name your Test Bench and the top level module in test bench as below -> click to file name to choose your test bench file

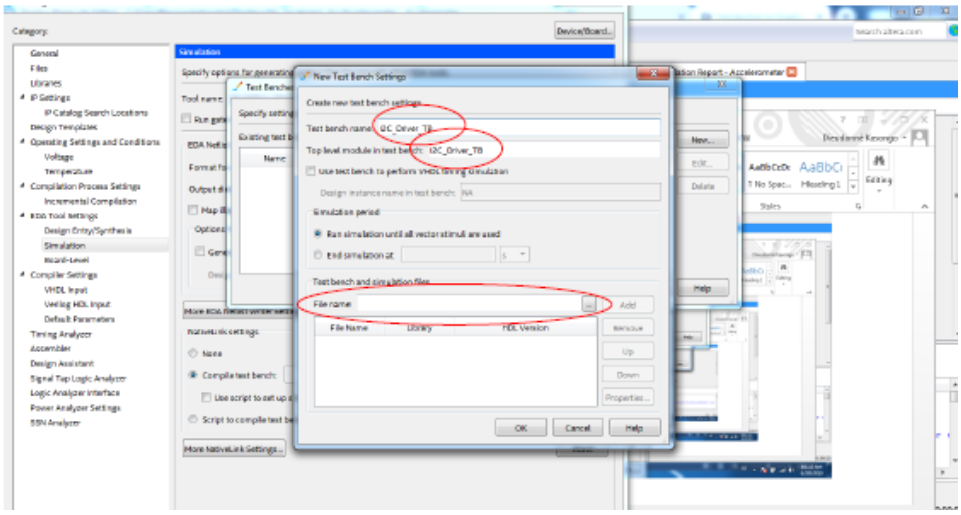


FIGURE 2.13 – Step 4 for the simulation of test bench

- Then choose your driver test bench name -> click to open

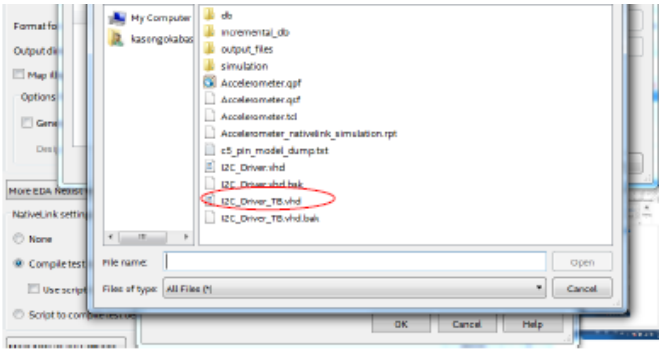


FIGURE 2.14 – Step 5 for the simulation of test bench

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 through a selector button signal. The application will need to have the inputs and outputs as shown on Figure 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 2.15 – Application's Inputs and Outputs

2.2.1 Counter

We will need to implemented a counter in our application with a button as sensitivity signal 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. The sSW will allow the two selectors management in Figure below.

```
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 2.16 – 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 2.17 – The signal sSW controlling the selectors

2.2.2 Reset

You will also realize an asynchronous reset. This is done by setting a condition that doesn't really on the clock. We check the state of a bit named $reset_n$. If it is a 1, we will set the sena (enable) to '0' and the state to "ready" which is the first state. Otherwise, we will 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 2.18 – The Reset

2.2.3 Configuration Setting

To configure the accelerometer sensor you will need, after the start condition, to write first 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 you will write the data output register "01000111" (47h) then you will wait for the stop condition. In the below table the sensors registers addresses you 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 2.19 – The LIS302DL Registers Address Map

2.2.4 Reading X, Y and Z detected acceleration

To read the acceleration detected for different axis, you can refer you to the register address map because to read X, Y and Z output, you 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, $\imath state_w ritereadı_0 x3b\jmath$, $\imath state_w ritereadı_0 x3b\jmath$ and $\imath state_w ritereadı_0 x3b\jmath$. After accelerometer configuration setting stop condition, you set a loop routine process starting from the state $\imath state_w ritereadı\jmath$ as follow : sena sets to '1' and R/W to '0' you write the slave address, after first slave ACK, you 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 $\imath state_w ritereadı_0 x3b\jmath$ to read the $Output_Z$ ($data_r d$) value available and then after master ACK ($\imath I_a ckmon\jmath$) you set the sena to '0' for stop condition then we go to the $\imath state_w ritereadı\jmath$ to perform the same routine than $\imath state_w ritereadı\jmath$ and this time to write the $Output_Y$ register address "00101011" (2Bh) and after the stop condition of this state we will go to $\imath state_w ritereadı\jmath$ to do again the same routine but this time you

will write the $Output_X$ register address "00101001" (29h) and after the stop condition we will come again to ı$state_w ritereadı$ state to 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 2.20 – 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 2.21 – 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 2.22 – The routine code to read X

2.2.5 Simulation

We can now create a test bench to check if our application is working. In this application the test bench of the signals we control are the acknowledgements, the busy, Button, $data_rd$, and I_rdon. The routine, that is the major concern here, is shown on Figure below. 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 process in the beginning. The results of the simulations are shown on Figure from 0s to 50us and from 50us to 100us below. you 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 2.23 – Test bench Routine

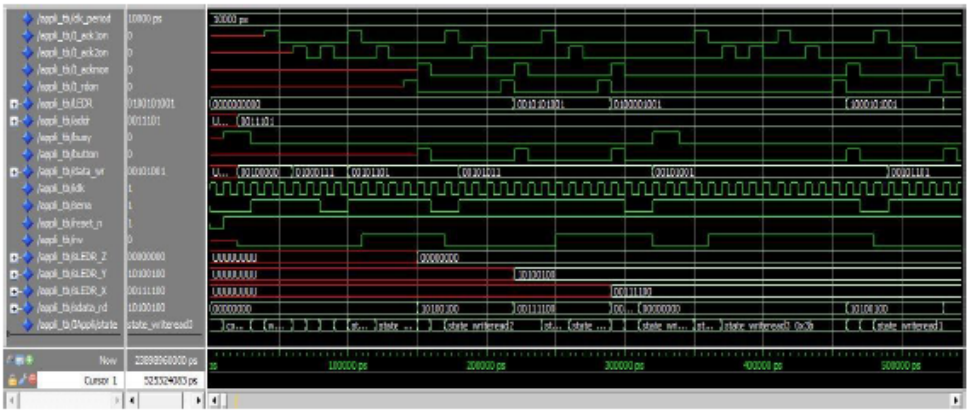


FIGURE 2.24 – Application test bench simulation from 0s to 50us

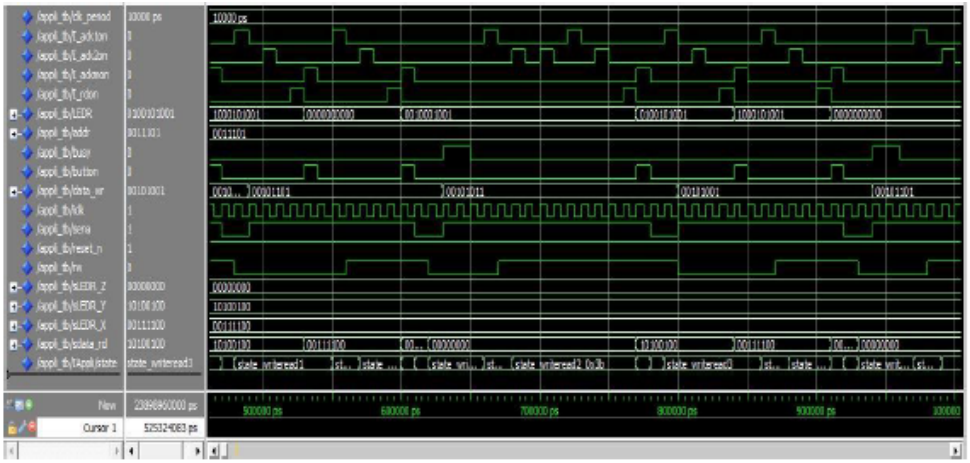


FIGURE 2.25 – Application test bench simulation from 50us to 100us

2.3 I2C Driver and Application Assembly

2.3.1 Working of the assembly

The last and main ".vhd" file of our projet will be the assembly of the driver and the application. It isn't difficult. Indeed, we 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 below. 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 button
    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 2.26 – Input and output of the assemby

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 you are on the read state. you will get the simulation results as shown on Figure below. These results are fine because you 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 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 2.27 – Control of the SDA

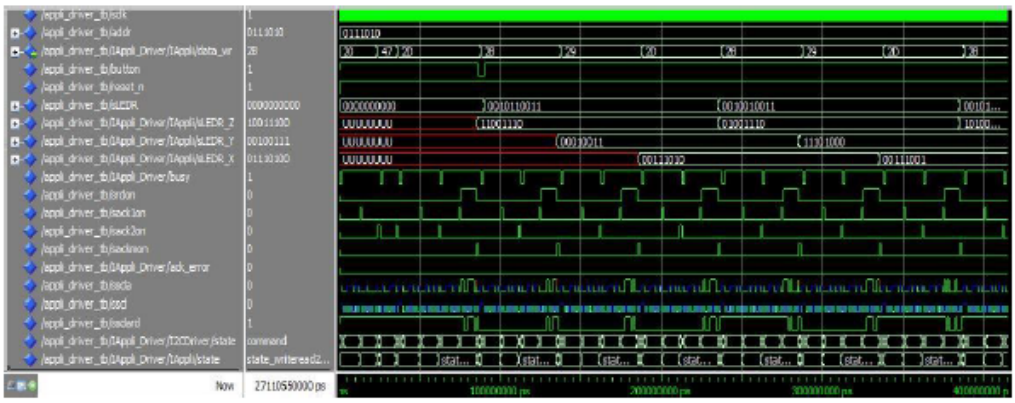


FIGURE 2.28 – Assembly test bench simulation from 0s to 0.4s

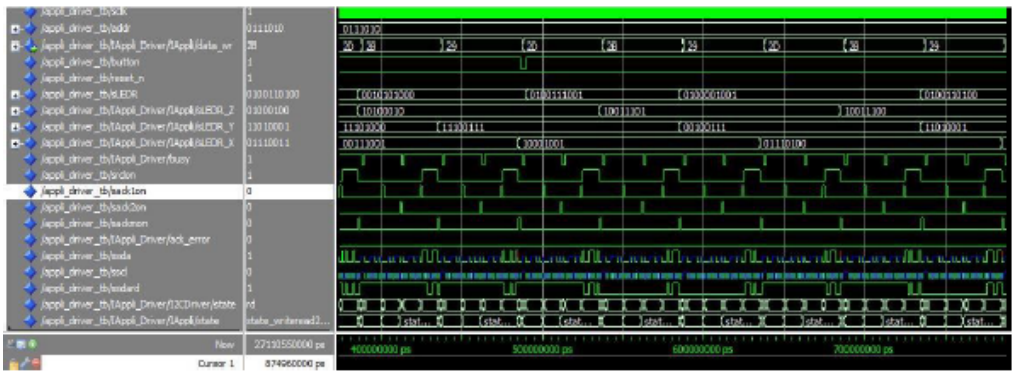


FIGURE 2.29 – Assembly test bench simulation from 0.4s to 0.8s

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

— Go to the tab assignments then to import the assignments

— Import the file in the project

— If we want to see what it gives, we can go in the Assignments editor and we will see all names, the old ones ("Value" column) and the new ones ("To" column).

CHAPITRE 3

CONCLUSION

During this laboratory project, we had learned to drive the LIS3LV02DL accelerometer sensor on a FPGA. For that, we had 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 were on the right track. After those simulations, we have concluded that we did things in order. Everything worked properly. So we flashed the code on the board. 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.

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

ANNEXE

- Free download URL : http ://fpgasoftware.intel.com/ ?edition=lite.
- I2C Master (VHDL) : https ://www.digikey.com/eewiki/pages/viewpage.action ?pageId=10125324 by Scott Larson.
- AN2335 Application note : LIS302DL.
- Capteur accéléromètre LIS3LV02DL en I2C - SEMI.
- UM10204, I2C-bus specification and user manual, NXP Semiconductors N.V.
- http ://www.ecs.umass.edu/ece354/ECE354HomePageFiles/Labs$_f iles/DE1_S oC.qsf.http$: $//www.ee.ic.ac.uk/pcheung/teaching/ee2_d igital/DE1 - SoC_U sermanual.pdf$.

— I2C Introduction au bus I2C, Camille Diou URL : https ://www.les electroniciens.com/sites/default/files/c i2c.pdf.