UMONS
Université de Mons

Faculté Polytechnique

# How to control a temperature sensor with an Altera Cyclone V SoC

MA1 - Hardware/Software Platforms

Charlotte LUYPAERT, Julien VANVILTHOVEN

POLYTECH MONS

May 2019

ACADÉMIE
UNIVERSITAIRE
WALLONIE-
BRUXELLES

# Contents

# Introduction

For this lab, you will learn to control a temperature sensor with an Altera cyclone V. The card you will use is a DE1-SoC which is built on a FPGA.

To do this projet, we will divide this tutorial in two parts.

The part "Before the lab" will help you to understanding the i2c protocol and get you familiar with a temperature C code. This is the code you will base on the beginning of your work.

The part "During the lab" that 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. You will have to simulate these three part individually to see if what you do is correct. These separations will appear in the part "During the lab". This part describe step by step all the steps you will have to follow to complete the project For this project it's asked to read the temperature in bytes. You don't have to read it with decimal numbers. In fact, you will have to display the temperature on leds.

Chapter 1

# Before the lab

Here are some crucial informations you need to know before the beginning of this tutorial. In this chapter, we will see an overview of what is the I2C protocol and we will study the way used in the PIC program.

## 1.1 What is the I2C protocol ?

The I2C (standing for Inter-Integrated circuit) is a communication protocol developped by Philips that permits various electronic components to communicate with only three connections. These signals are the data signal SDA, the clock signal SCL and the ground. It has the particularities that there is one single address for a device and that it is a multi-master.

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

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

Once we have checked that the bus was available and taken control, the circuit becomes the master and it's it that generates the clock signal.
Here's a picture wich illustrates this well (Cf. Figure 1.1)

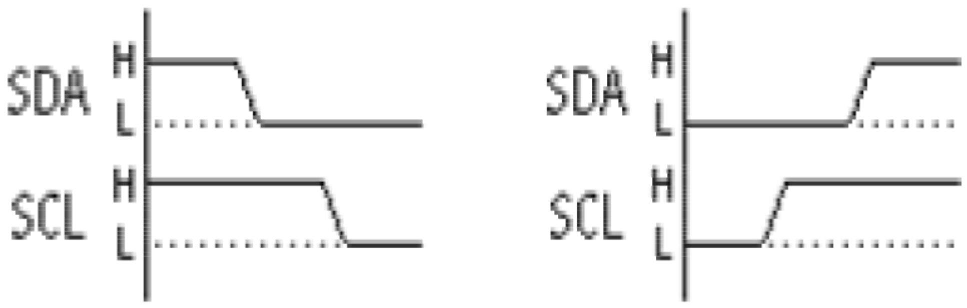


Figure 1.1: Image representing the bit start (on the left) and the bit stop (on the right) during the takearound of the bus []

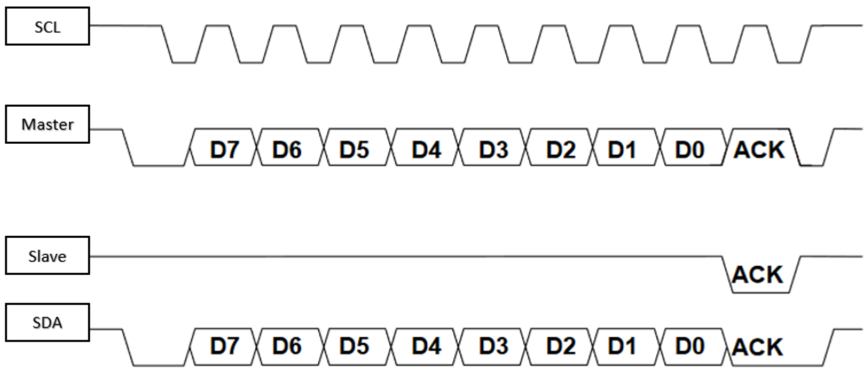


Figure 1.2: Illustration of a byte transmission []

### 1.1.1 Byte transmission

To transmit a byte, the master sends first the most significant bit 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.

### 1.1.2 Writing a data

To write a data with I2C, we need to specify several datas. First, we need to send the register 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 register address and the R/W (here, equals to 0). Then, we send the data that we need to be written.

### 1.1.3 Reading a data

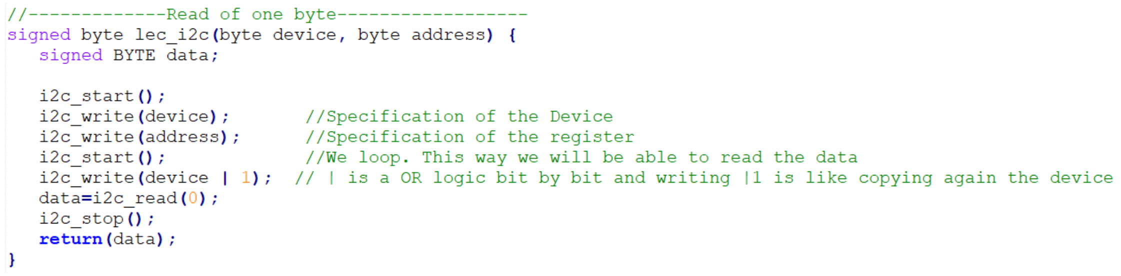
To read a data, at first, we need to send some datas and then we can read. The master sends the register address and the R/W (here, equals to 1). Then, after the ACK of the slave has been received, its sends the datas to the SDA. Eventually, the master sets the ACK to 0 to continue the reading and to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

## 1.2 Analysis of the PIC program

In this section, we are going to analyse the PIC code that you received with this document. First, we can see that there are several functions called by the main program. There is a function to read 1 byte, a function to read 2 bytes, a function to write 1 byte. We can also see that there are 2 functions called a lot of times : *i2cstart* and *i2cstop*. These represents respectively the start and the stop sequences.

### 1.2.1 Reading one byte

This code is used to read a data that is 8 bits long (cf. Figure 1.3). We can see that like with the I2C, we begin with specifying the device address and the register address. Then, we give the device address once again and we wait for the data sent by the sensor.

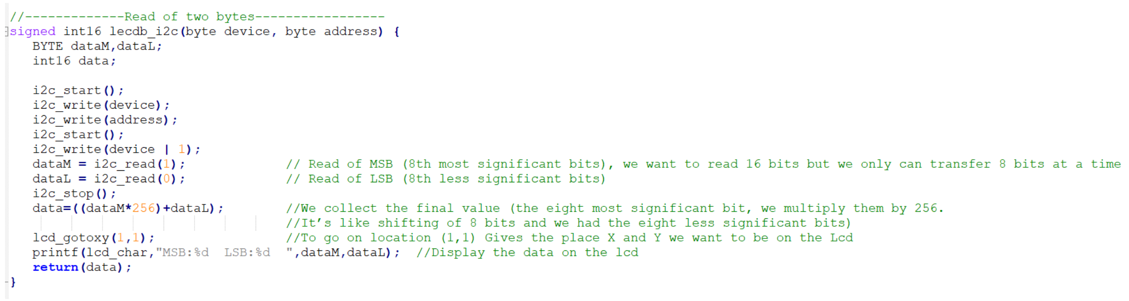


```
//-------------Read of one byte------------------
signed byte lec_i2c(byte device, byte address) {
   signed BYTE data;

   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 1.3: Code used in the function *leci2c*

### 1.2.2 Reading two bytes

This function called *lecdbi2c* allows to read datas that are 16 bits long. We can see on the Figure 1.4 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 datas.

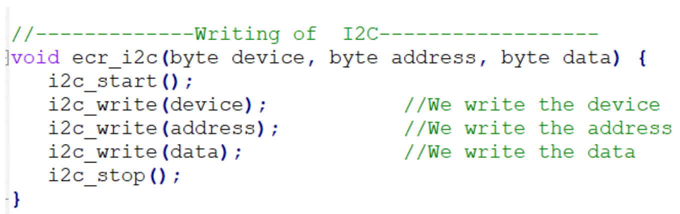


```
//-------------Read of two bytes-----------------
signed int16 lecdb_i2c(byte device, byte address) {
   BYTE dataM,dataL;
   int16 data;

   i2c_start();
   i2c_write(device);
   i2c_write(address);
   i2c_start();
   i2c_write(device | 1);
   dataM = i2c_read(1);             // Read of MSB (8th most significant bits), we want to read 16 bits but we only can transfer 8 bits at a time
   dataL = i2c_read(0);             // Read of LSB (8th less significant bits)
   i2c_stop();
   data=((dataM*256)+dataL);        //We collect the final value (the eight most significant bit, we multiply them by 256.
                                    //It's like shifting of 8 bits and we had the eight less significant bits)
   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 1.4: Code used in function *lecdbi2c*

### 1.2.3 Writing a byte

The function *ecri2c* 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).

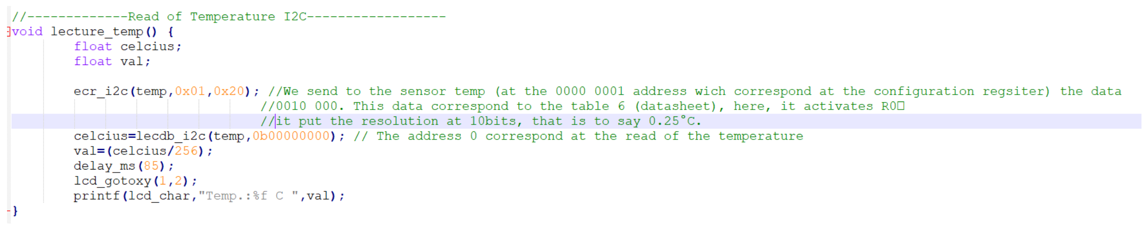


```
//-------------Writing of  I2C------------------
void ecr_i2c(byte device, byte address, byte data) {
   i2c_start();
   i2c_write(device);           //We write the device
   i2c_write(address);          //We write the address
   i2c_write(data);             //We write the data
   i2c_stop();
}
```

Figure 1.5: Code used in function *ecri2c*

### 1.2.4 Temperature reading

Here, we use everything we defined before to do a complete temperature reading on the sensor (cf. Figure 1.6). The comments in the code are detailing what is the purpose of each line.



```
//-------------Read of Temperature I2C------------------
void lecture_temp() {
        float celcius;
        float val;

        ecr_i2c(temp,0x01,0x20); //We send to the sensor temp (at the 0000 0001 address wich correspond at the configuration regsiter) the data
                                //0010 000. This data correspond to the table 6 (datasheet), here, it activates R0
                                //it put the resolution at 10bits, that is to say 0.25°C.
        celcius=lecdb_i2c(temp,0b00000000); // The address 0 correspond at the read of the temperature
        val=(celcius/256);
        delay_ms(85);
        lcd_gotoxy(1,2);
        printf(lcd_char,"Temp.:%f C ",val);
}
```

Figure 1.6: Code used in the function *lecturetemp*

# Chapter 2

# During the lab

In this chapter, we will describe, step by step, all the things we did for this project. From the beginning until the time we sent the code on the chip.

## 2.1 I2C driver

### Code

We have done some research to find a first code to help us in the project. Indeed, we wanted a base to begin our project. We found an I2C VHDL code [] on the internet.
Once you have this code:

- Launch QuartusII-64bits
- Create a projet: File → New Project Wizard → Click "Next" until it's created.
- Create a new file in the project
- Paste the VHDL code [4]
- Save the project as a vhdl file with the name *i2c_master*
- Compile (use the shortcut represented with a play symbol)

Once compiled, Quartus shows us the state machine but the representation was clearer on the website. On Figure 2.1, we can see an overview of this state machine. Once you have understood the state machine, you will have to create a Test Bench. (It's easier to understand the VHDL code with the state machine).

To create a Test Bench, you have to create a new file in the project (Name it i2cmasterTB). A test Bench is a VHDL module wich tests the behaviour of another module. He produces the signals to send to the system we want to test and check that the outputs of the circuit are correct. Here is a part of the code you need to implement (Cf. Figure 2.2 and Figure 2.3). In the part **port map** we associate our **PORT** to the corresponding **signals**.

Then, we have some **process**:

- One for the clock

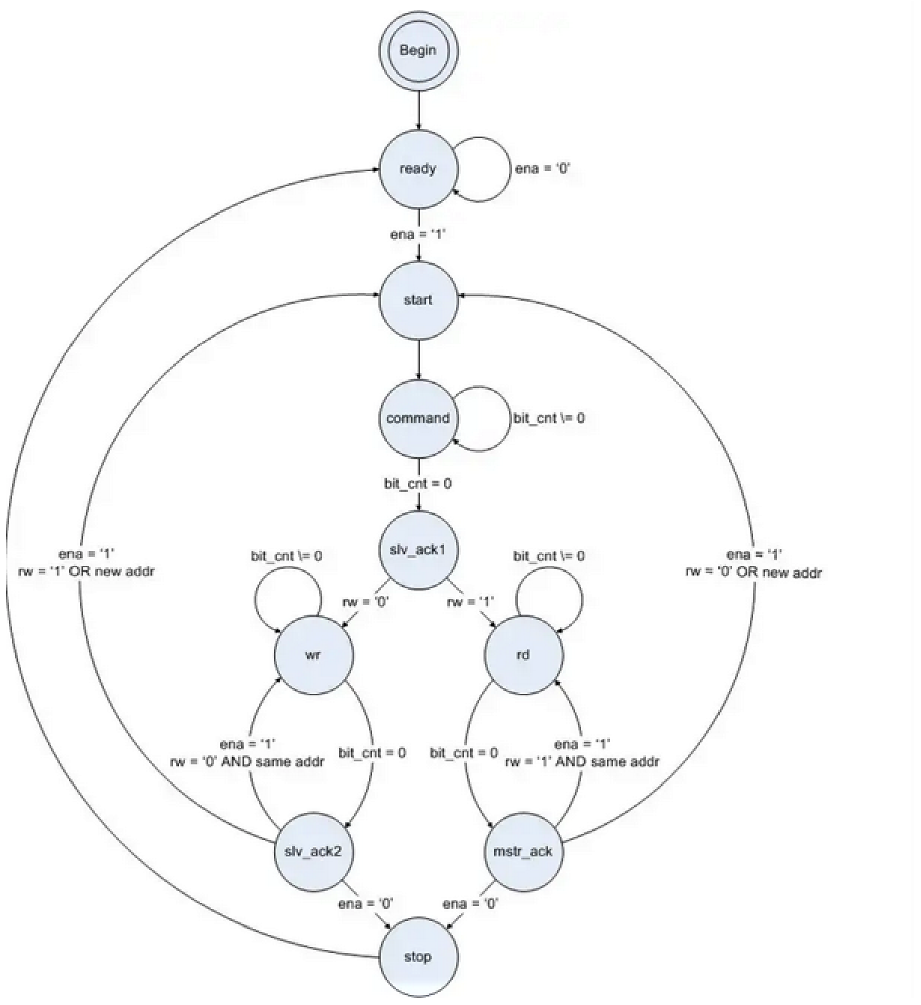


Figure 2.1: State machine of the driver [4]

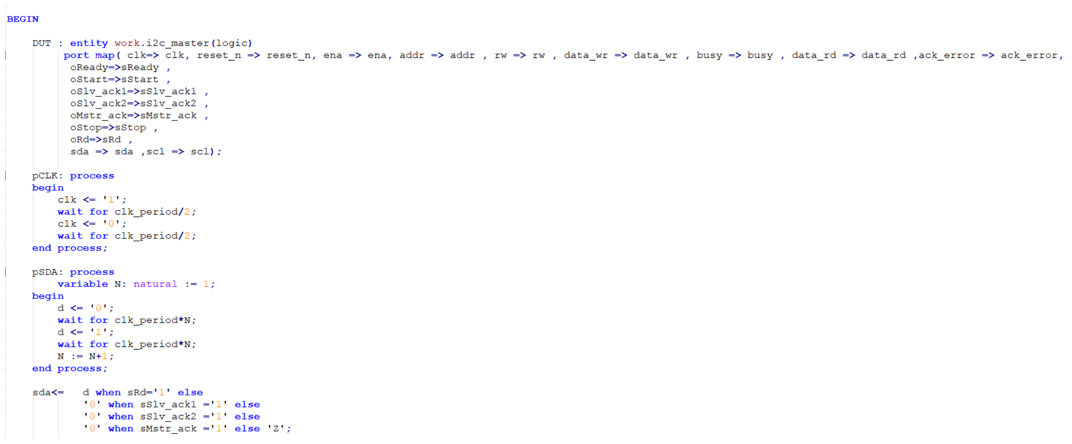
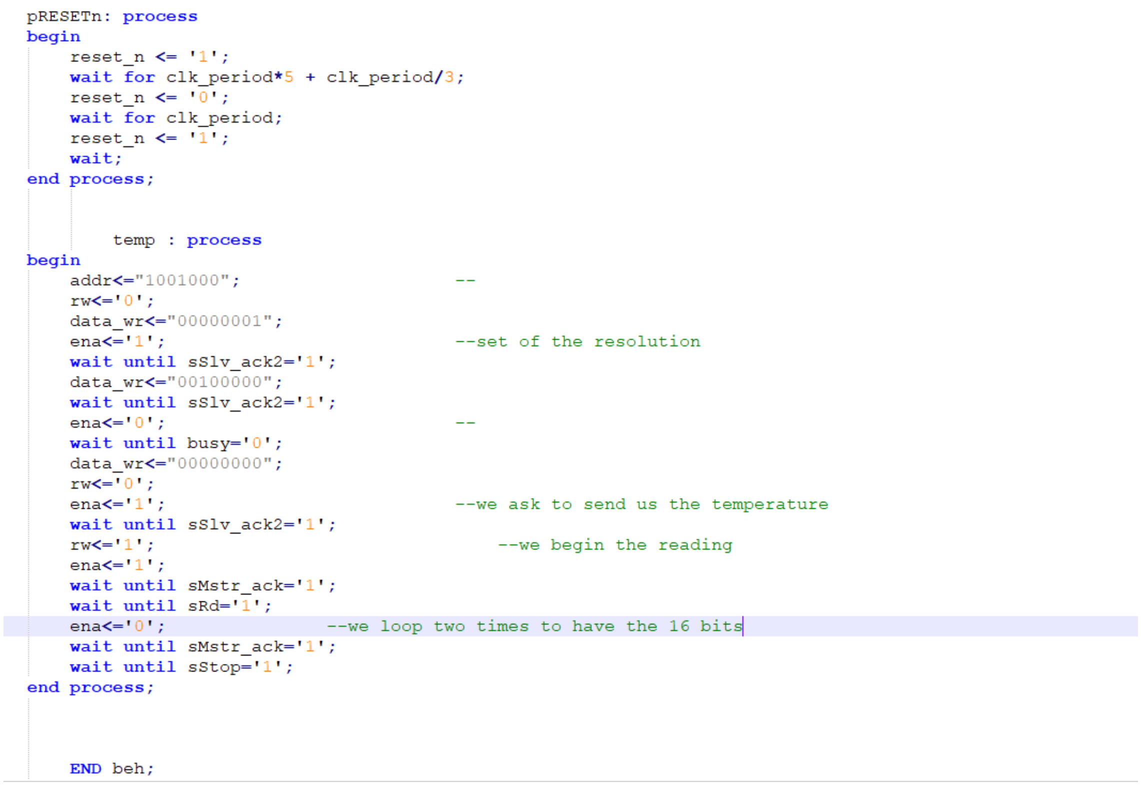

```
BEGIN

    DUT : entity work.i2c_master(logic)
          port map( clk=> clk, reset_n => reset_n, ena => ena, addr => addr , rw => rw , data_wr => data_wr , busy => busy , data_rd => data_rd ,ack_error => ack_error,
           oReady=>sReady ,
           oStart=>sStart ,
           oSlv_ack1=>sSlv_ack1 ,
           oSlv_ack2=>sSlv_ack2 ,
           oMstr_ack=>sMstr_ack ,
           oStop=>sStop ,
           oRd=>sRd ,
           sda => sda ,scl => scl);

    pCLK: process
    begin
        clk <= '1';
        wait for clk_period/2;
        clk <= '0';
        wait for clk_period/2;
    end process;

    pSDA: process
        variable N: natural := 1;
    begin
        d <= '0';
        wait for clk_period*N;
        d <= '1';
        wait for clk_period*N;
        N := N+1;
    end process;

    sda<=   d when sRd='1' else
            '0' when sSlv_ack1 ='1' else
            '0' when sSlv_ack2 ='1' else
            '0' when sMstr_ack ='1' else 'Z';
```

Figure 2.2: First part of the code



```
pRESETn: process
begin
    reset_n <= '1';
    wait for clk_period*5 + clk_period/3;
    reset_n <= '0';
    wait for clk_period;
    reset_n <= '1';
    wait;
end process;


        temp : process
begin
    addr<="1001000";                        --
    rw<='0';
    data_wr<="00000001";
    ena<='1';                               --set of the resolution
    wait until sSlv_ack2='1';
    data_wr<="00100000";
    wait until sSlv_ack2='1';
    ena<='0';                               --
    wait until busy='0';
    data_wr<="00000000";
    rw<='0';
    ena<='1';                               --we ask to send us the temperature
    wait until sSlv_ack2='1';
    rw<='1';                                    --we begin the reading
    ena<='1';
    wait until sMstr_ack='1';
    wait until sRd='1';
    ena<='0';                   --we loop two times to have the 16 bits
    wait until sMstr_ack='1';
    wait until sStop='1';
end process;



    END beh;
```

Figure 2.3: Second part of the code

- the pSDA process is there to create a fake gate. In fact if we don't simulate a fake gate, we can't see the simulation working properly
- There is also a process for the reset but it's not very important in our case
- The last process describe the setting of the resolution and the reading of the temperature as commented in the code.

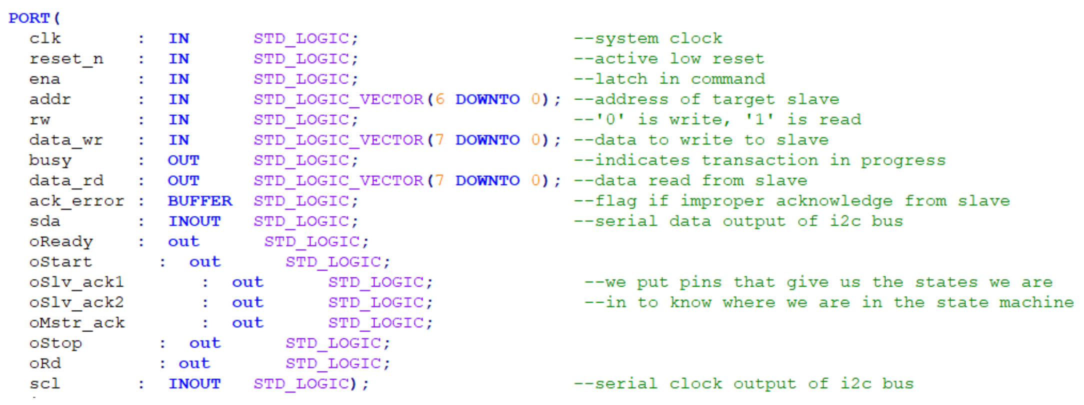
The driver also need input and outputs that you have to define. You can find them below (Cf. Figure 2.6).

To create this code, we looked at the state machine and translated it.

### Simulation

Once you have done your Test Bench, you'll have to simulate it to see if the code of your driver works properly. To this end, you'll have to do these steps:

- Put the file i2cmaster as set top (right click on the file)
- Go into the tab Assignements, then Settings and verify that in the section simulation, the compile test bench is i2cmasterTB
- Go into the tab Tools then RTL Viewer
- The simulation launches itself, you just have to click on the Stop button to stop it



```
PORT(
  clk        :  IN      STD_LOGIC;                      --system clock
  reset_n    :  IN      STD_LOGIC;                      --active low reset
  ena        :  IN      STD_LOGIC;                      --latch in command
  addr       :  IN      STD_LOGIC_VECTOR(6 DOWNTO 0);  --address of target slave
  rw         :  IN      STD_LOGIC;                      --'0' is write, '1' is read
  data_wr    :  IN      STD_LOGIC_VECTOR(7 DOWNTO 0);  --data to write to slave
  busy       :  OUT     STD_LOGIC;                      --indicates transaction in progress
  data_rd    :  OUT     STD_LOGIC_VECTOR(7 DOWNTO 0);  --data read from slave
  ack_error  :  BUFFER  STD_LOGIC;                      --flag if improper acknowledge from slave
  sda        :  INOUT   STD_LOGIC;                      --serial data output of i2c bus
  oReady     :  out      STD_LOGIC;
  oStart       :  out      STD_LOGIC;
  oSlv_ack1        :  out      STD_LOGIC;                --we put pins that give us the states we are
  oSlv_ack2        :  out      STD_LOGIC;                --in to know where we are in the state machine
  oMstr_ack        :  out      STD_LOGIC;
  oStop        :  out      STD_LOGIC;
  oRd          : out      STD_LOGIC;
  scl        :  INOUT   STD_LOGIC);                     --serial clock output of i2c bus
```

Figure 2.4: Inputs and outputs of the application

- Go to the beginning of the simulation and zoom out to see the the signals

Once all is done, check your results. You must have the same results as shown below (Cf. Figure 2.5).

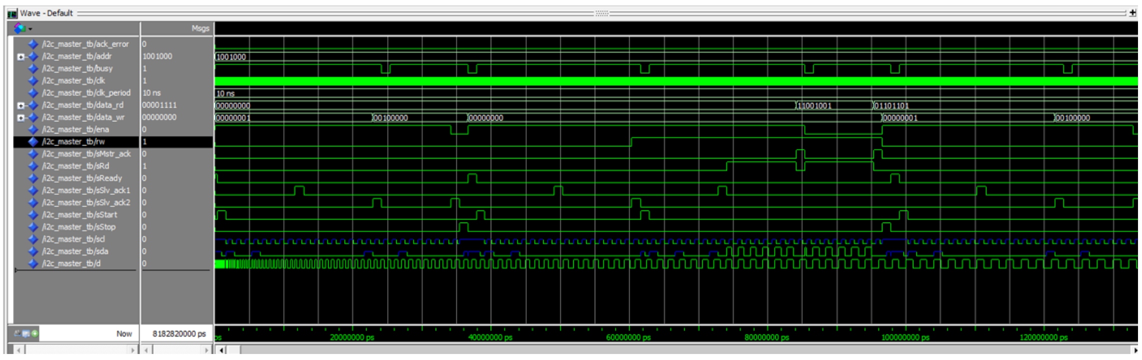
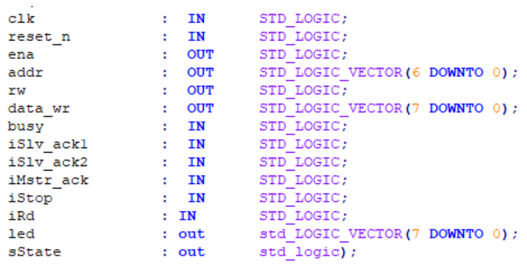


Figure 2.5: Simulation of the driver

The signals that are important to look at are the d, the data-rd, the data-rw and the Mstr-ack. When we look at the data-wr, we see well the resolution. We also see we receive a Master ack after it finishes to read. We also can see the d fake gate.

## 2.2 Application

The application is the part of the project that will handle the temperature sensor state machine. It will do the same thing that the test bench previously seen but with exact conditions to change from a state to another. So what we need to do is to follow the right path in the state machine of the driver to, in a first part, set the resolution to the one we want (here, 10 bits) and, in a second part, read the data corresponding to the temperature. The application will need to have the inputs and outputs as shown on Figure 2.6.

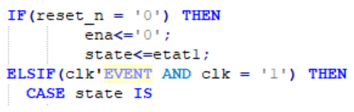


```
clk             :  IN     STD_LOGIC;
reset_n         :  IN     STD_LOGIC;
ena             :  OUT    STD_LOGIC;
addr            :  OUT    STD_LOGIC_VECTOR(6 DOWNTO 0);
rw              :  OUT    STD_LOGIC;
data_wr         :  OUT    STD_LOGIC_VECTOR(7 DOWNTO 0);
busy            :  IN     STD_LOGIC;
iSlv_ack1       :  IN     STD_LOGIC;
iSlv_ack2       :  IN     STD_LOGIC;
iMstr_ack       :  IN     STD_LOGIC;
iStop           :  IN     STD_LOGIC;
iRd             : IN      STD_LOGIC;
led             : out     std_LOGIC_VECTOR(7 DOWNTO 0);
sState          : out     std_logic);
```

Figure 2.6: Inputs and outputs of the application

### 2.2.1 Reset

First, we realize an asynchronous reset (cf. Figure 2.7). 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* to 0 and the state to "etat1" which is the first state. Otherwise, we simply go to the state machine at each rising edge of the clock.



```
IF(reset_n = '0') THEN
        ena<='0';
        state<=etat1;
ELSIF(clk'EVENT AND clk = '1') THEN
  CASE state IS
```

Figure 2.7: Reset

### 2.2.2 Setting the resolution

To set the resolution, we will need to make our way into the settings of the sensor. These are ruled like described on Figure 2.8 [2]. We see that we, first, need to go to the configuration

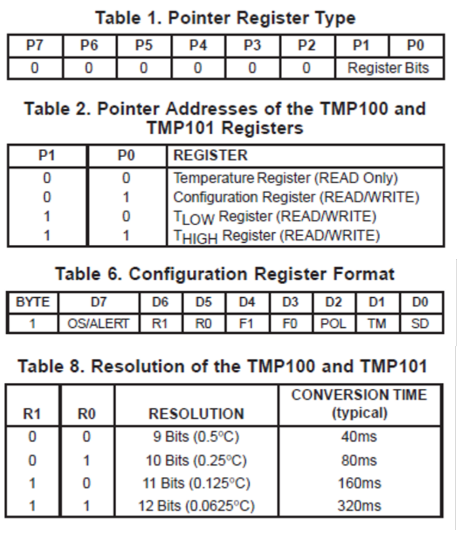


**Table 1. Pointer Register Type**

| P7 | P6 | P5 | P4 | P3 | P2 | P1 | P0 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | Register Bits | |

**Table 2. Pointer Addresses of the TMP100 and TMP101 Registers**

| P1 | P0 | REGISTER |
|---|---|---|
| 0 | 0 | Temperature Register (READ Only) |
| 0 | 1 | Configuration Register (READ/WRITE) |
| 1 | 0 | $T_{LOW}$ Register (READ/WRITE) |
| 1 | 1 | $T_{HIGH}$ Register (READ/WRITE) |

**Table 6. Configuration Register Format**

| BYTE | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 1 | OS/ALERT | R1 | R0 | F1 | F0 | POL | TM | SD |

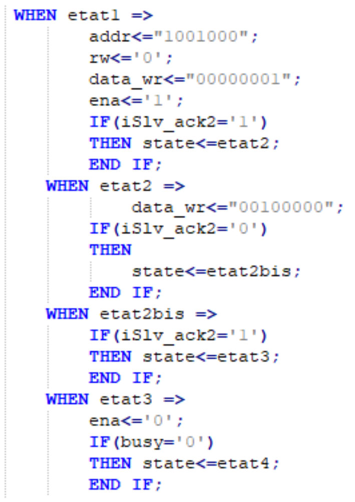
**Table 8. Resolution of the TMP100 and TMP101**

| R1 | R0 | RESOLUTION | CONVERSION TIME (typical) |
|---|---|---|---|
| 0 | 0 | 9 Bits (0.5°C) | 40ms |
| 0 | 1 | 10 Bits (0.25°C) | 80ms |
| 1 | 0 | 11 Bits (0.125°C) | 160ms |
| 1 | 1 | 12 Bits (0.0625°C) | 320ms |

Figure 2.8: Registers of the sensor

register, which is done by setting *P0* to 1 and *P1* to 0 in the pointer register, *i.e.* by writing

0x01. Then, we arrive in the configuration register and we can see that, here, we need to set the *R0* to 1 and *R1* to 0, so 0x20, to choose a 10 bits resolution. But the special thing here is that we need to send those two datas without passing through the stop state. That is why we will need to ensure that the state machine loops one time in the write state and that there is a change in the data to be written during this loop.
The state machine is implemented in the program with a "case". The process of the setting of the resolution will be realized by the states named "etat1", "etat2", "etat2bis" and "etat3" (the names could be changed for an easier reading of the program) and the code associated is shown on Figure 2.9. To set the resolution, it is basically a writing. We need to specify the device address (0x90), the pointer register address (0x01) and the configuration register address (0x20). To do so, we need to set the device address (0x90) in *addr*, the *R/W* (0) and



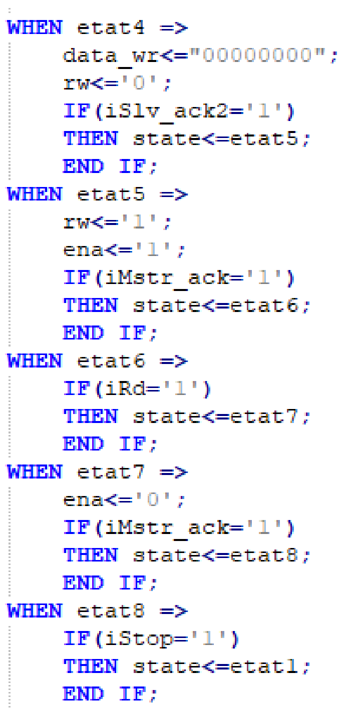
```
WHEN etat1 =>
        addr<="1001000";
        rw<='0';
        data_wr<="00000001";
        ena<='1';
        IF(iSlv_ack2='1')
        THEN state<=etat2;
        END IF;
    WHEN etat2 =>
            data_wr<="00100000";
        IF(iSlv_ack2='0')
        THEN
            state<=etat2bis;
        END IF;
    WHEN etat2bis =>
        IF(iSlv_ack2='1')
        THEN state<=etat3;
        END IF;
    WHEN etat3 =>
        ena<='0';
        IF(busy='0')
        THEN state<=etat4;
        END IF;
```

Figure 2.9: Setting the resolution

the pointer register address (0x01) in *data_wr*. As we begin in the ready state, we switch to the start state by setting the *ena* to 1 (etat1). The *R/W* is equal to 0, so we go to the writing state and when the register address is sent, we arrive in the slaveack2 state. Once we received this acknowledgement, we change the *data_wr* to 0x20 to set the resolution on 10 bits (etat2) and when we, once again, go to the slaveack2 state, we know that the resolution is defined (etat2bis). We, then, switch the *ena* to 0 to go to the stop state and then to return in the ready state (etat3). We are now ready for the reading.

### 2.2.3 Reading the temperature

To read the temperature, we can see on Figure 2.8 that we only need to write a data corresponding to the temperature register in the pointer register. It is to say setting *P0* and *P1* to 0, so we need to write 0x00. The sensor will then send the datas corresponding to the temperature. The process of reading will be realized by the states named "etat4", "etat5", "etat6", "etat7" and "etat8" (cf. Figure 2.10). So, as for the setting of the resolution, we specify the device address (0x90) in *addr*, the *R/W* (0) and the pointer register address (0x00) in *data_wr*. We begin in the ready state, we switch to the start state by setting the *ena* to 1 (etat4). The *R/W* is equal to 0, so we go to the writing state and when the register address is



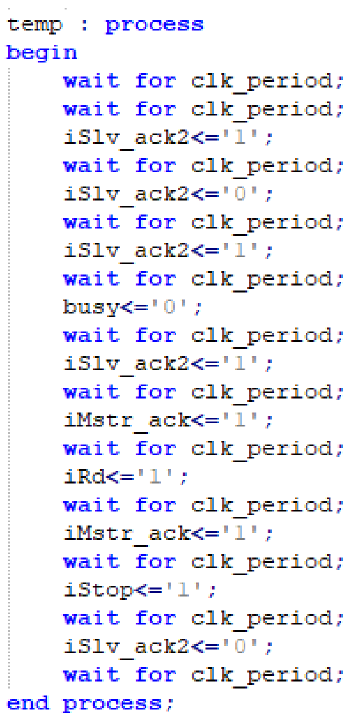
```
WHEN etat4 =>
        data_wr<="00000000";
        rw<='0';
        IF(iSlv_ack2='1')
        THEN state<=etat5;
        END IF;
WHEN etat5 =>
        rw<='1';
        ena<='1';
        IF(iMstr_ack='1')
        THEN state<=etat6;
        END IF;
WHEN etat6 =>
        IF(iRd='1')
        THEN state<=etat7;
        END IF;
WHEN etat7 =>
        ena<='0';
        IF(iMstr_ack='1')
        THEN state<=etat8;
        END IF;
WHEN etat8 =>
        IF(iStop='1')
        THEN state<=etat1;
        END IF;
```

Figure 2.10: Reading the temperature

sent, we go to the slaveack2 state. Once we received this acknowledgement, the sensor is going to send the data corresponding to the temperature which is coded on 16 bits. We dont want to go to the stop state because it will cancel the demand of reading, so, we only switch the $R/W$ to 1 (etat5). The driver will then save the datas sent by the sensor on a variable. Once this is done, we receive a master acknowledgement. To make sure to read the entire 2 bytes, we do not change anything before receiving a second master acknowledgement. To check that, we wait until the moment the card is reading the second time with *iRd* (etat6) which is on 1 when the card is reading. We can then set the *ena* to 0 (etat7) so that we switch to the stop state as soon as the second master acknowledgement is received (etat8) thanks to the *iStop* which is 1 when we are in the stop state. We are now ready for a new process of reading. In the event of a case value different from our defined states, we go back to the first state.

### 2.2.4 Simulation

We can now create a test bench to check that our application is working well. To that end, we do as mentioned in section 2.1. In this test bench the signals we control are the acknowledgements and the *iRd* and *iStop*. The routine, that is the major concern here, is shown on Figure 2.11. 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 2.12. We can see that they are satisfying because the states are changing in the right order which was the goal of our routine.



```
temp : process
begin
    wait for clk_period;
    wait for clk_period;
    iSlv_ack2<='1';
    wait for clk_period;
    iSlv_ack2<='0';
    wait for clk_period;
    iSlv_ack2<='1';
    wait for clk_period;
    busy<='0';
    wait for clk_period;
    iSlv_ack2<='1';
    wait for clk_period;
    iMstr_ack<='1';
    wait for clk_period;
    iRd<='1';
    wait for clk_period;
    iMstr_ack<='1';
    wait for clk_period;
    iStop<='1';
    wait for clk_period;
    iSlv_ack2<='0';
    wait for clk_period;
end process;
```

Figure 2.11: Routine of the test bench

## 2.3 Assembly of the driver and the application

### 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 the hardest though. Indeed, you only need to declare an instance of a component "driver" and an instance of a component "application" and make the connections. The inputs and outputs will need to be as shown on Figure 2.13. Indeed, our main program needs to have the clock and the reset as inputs and the i2c ports SDA and SCL as inputs and outputs. The leds on which we will show the temperature 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. We also need to assign the value read by the sensor to the led vector.

### 2.3.2 Simulation of the assembly

As before, we need to test the assembly before sending it on the card. To do so, we realize a test bench in which we control the SDA by making fake acknowledgements when needed and fake datas when we are on the read state (cf. Figure 2.14). We get the results shown on Figure 2.15. 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 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.

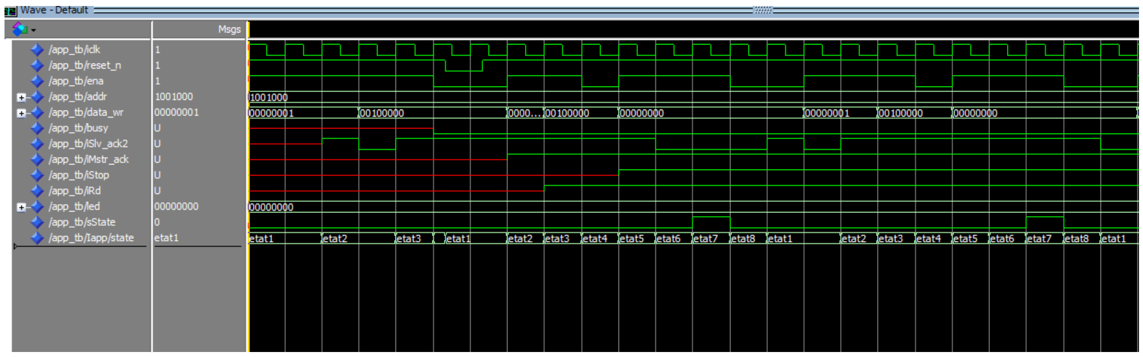
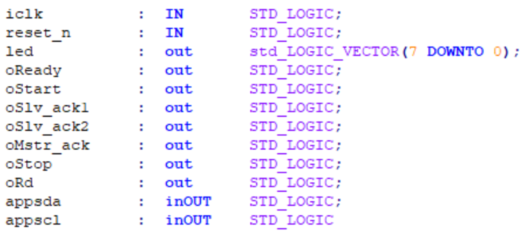


Figure 2.12: Application test bench simulation



```
iclk         :  IN     STD_LOGIC;
reset_n      :  IN     STD_LOGIC;
led          :  out    std_LOGIC_VECTOR(7 DOWNTO 0);
oReady       :  out    STD_LOGIC;
oStart       :  out    STD_LOGIC;
oSlv_ack1    :  out    STD_LOGIC;
oSlv_ack2    :  out    STD_LOGIC;
oMstr_ack    :  out    STD_LOGIC;
oStop        :  out    STD_LOGIC;
oRd          :  out    STD_LOGIC;
appsda       :  inOUT  STD_LOGIC;
appscl       :  inOUT  STD_LOGIC
```

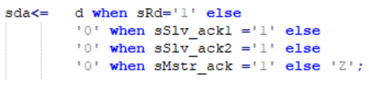
Figure 2.13: Inputs and outputs of the assembly

## 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 [3]
- Go to the tab assignements then to import assignements
- Import the file in the project
- If you want to see what it gives, you can go in Assignments editor and you see all the names, the old ones ("Value" column) and the new ones ("To" column).

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



```
sda<=   d when sRd='1' else
        '0' when sSlv_ack1 ='1' else
        '0' when sSlv_ack2 ='1' else
        '0' when sMstr_ack ='1' else 'Z';
```

Figure 2.14: Control of the SDA

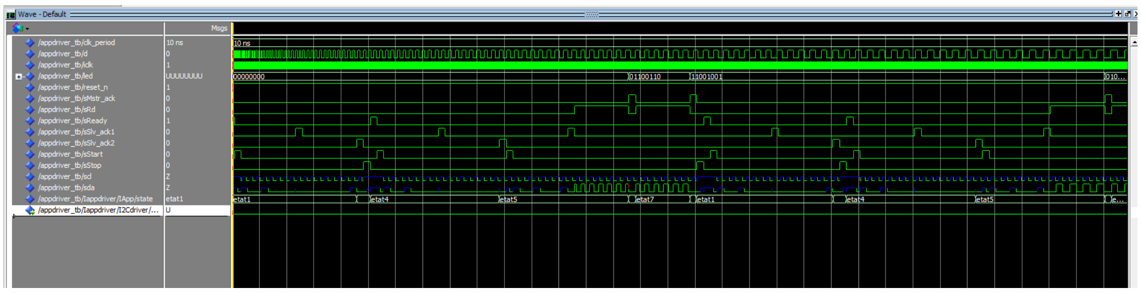
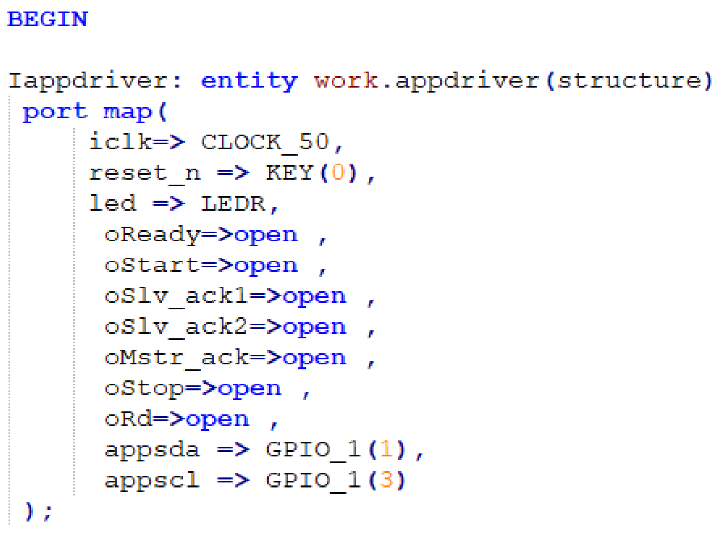

Wave - Default
Msgs
/appdriver_tb/clk_period 10 ns
/appdriver_tb/d 0
/appdriver_tb/iclk 1
/appdriver_tb/led UUUUUUUU
/appdriver_tb/reset_n 1
/appdriver_tb/sMstr_ack 0
/appdriver_tb/sRd 0
/appdriver_tb/sReady 1
/appdriver_tb/sSlv_ack1 0
/appdriver_tb/sSlv_ack2 0
/appdriver_tb/sStart 0
/appdriver_tb/sStop 0
/appdriver_tb/scl Z
/appdriver_tb/sda Z
/appdriver_tb/Iappdriver/IApp/state etat1
/appdriver_tb/Iappdriver/I2Cdriver/... U

Figure 2.15: Assembly test bench simulation



```
BEGIN

Iappdriver: entity work.appdriver(structure)
 port map(
      iclk=> CLOCK_50,
      reset_n => KEY(0),
      led => LEDR,
       oReady=>open ,
       oStart=>open ,
       oSlv_ack1=>open ,
       oSlv_ack2=>open ,
       oMstr_ack=>open ,
       oStop=>open ,
       oRd=>open ,
       appsda => GPIO_1(1),
       appscl => GPIO_1(3)
 );
```

Figure 2.16: Pin-Port Association

## 2.5 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 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 2.16) We can see that we associate the clock to CLOCK-50 and the reset to the button KEY(0). Then we have some signals we don't associate, that's why we can see the word *open*. We set the appscl to GPIO-1(1) and appscl to GPIO-1(3). We can see these pins on the picture below (Cf. Figure 2.17)

Before sending the code to the card, you must connect the temperature sensor to the card based on figures 2.17 and 2.18. To connect the card and the temperature sensor:

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

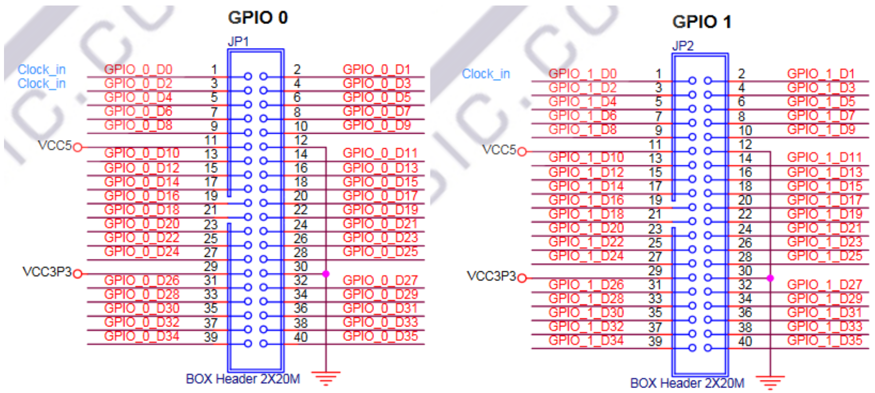


Figure 2.17: Pin GPIO on the DE1-SoC

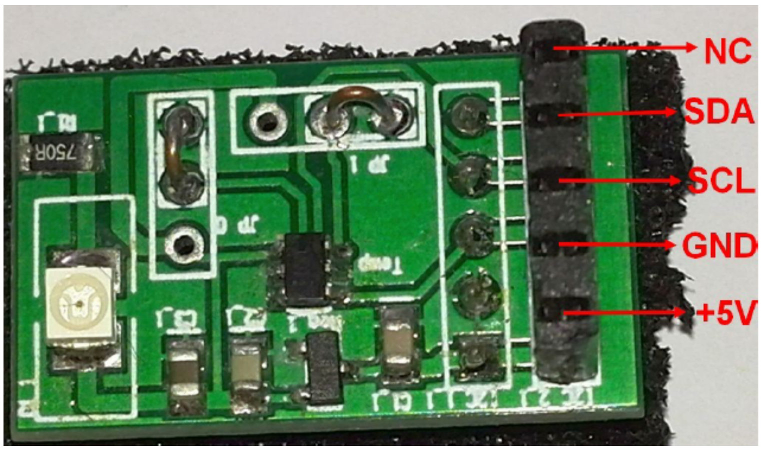


Figure 2.18: Temperature pins [1]

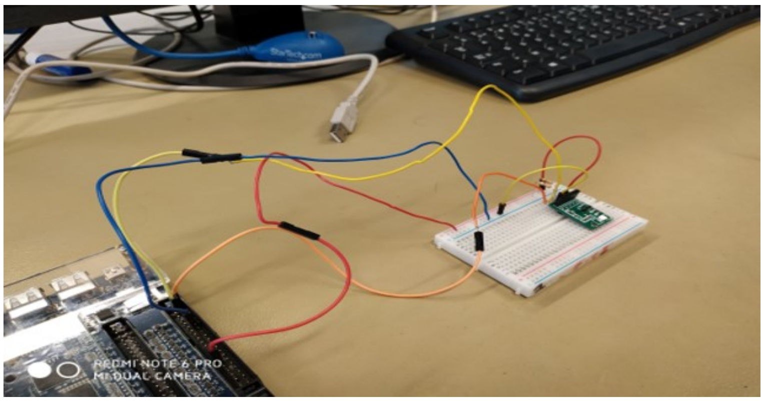


Figure 2.19: Assembly of the temperature sensor and the card

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 of $4.7k\Omega$ each : one between VDD and SCL and one between VDD and SDA.

Here's a picture that illustrate this assembly (Cf. Figure 2.19):
Once you have done the assembly, you are ready to upload the code on the DE1-SoC. To do that, follow the following steps:

- Compile the last code you wrote
- Go in the tab Programme 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

# Conclusion

During this project we have learned to control a temperature sensor on a FPGA. For that, we familiarized with the i2c 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 behaviour and to check if what we did wasn't wrong.

After our simulations we concluded we did things well. Everything worked properly. So we implemented the code on the card. This way, when we touched the sensor, we could see the temperature changing with the leds. We even looked the i2c trame on an oscilloscope and we saw well that we write three times before reading.

# Bibliography

[1] *Capteur de température TMP100 en i2c-SEMI.*

[2] *Digital temperature sensor with i2c interface-Texas Instrument.*

[3] Walker Jason. *DE1.qsf.* URL: `https://github.com/ungood/fpga/blob/master/DE1.qsf`.

[4] Scott Larson. *I2C Master (VHDL)/Digikey.* 2019. URL: `https://www.digikey.com/eewiki/pages/viewpage.action?pageId=10125324`.