UMONS
Université de Mons

Faculté Polytechnique

Project 2020

Board Altera DE1: Driver Temperature Sensor and 7-segment display

MA1: Hardware/Software Platforms

Maxime Destrait, Thomas Godesiabois-Galand

POLYTECH MONS

Under the direction of VALDERRAMA

Academic year 2019-2020

ACADÉMIE UNIVERSITAIRE WALLONIE-BRUXELLES

Contents

List of Figures

Introduction

The Hardware and Software course is a second master course whose objective is the construction of a VHDL driver coding project. We decided to work with an Altera DE1 Board and the latest version of Quartus II and Modelsim.

To make this project, we have divided this tutorial into different parts:

The first part illustrates the correct use of the Quartus II and Modelsim programs, introducing the I2C protocol at the same time.

Then, a part will be dedicated to the management of the link between the state machine and the temperature sensor (resolution of the measurements).

We will then discuss the use of the seven-segment readers incorporated in our FPGA board. We will take a particular interest in the format used, in particular with a BCD algorithm.

Finally, we will build a global application described as follows: A temperature sensor displaying its results on the seven-segment readers of an Altera DE1 board.

Chapter 1

Driver Temperature Sensor

In this chapter we will describe the I2C protocol in detail. We will begin with a brief theoretical overview of this protocol. Then we will present a global tutorial on how to implement VHDL code in a Quartus project and how to make a Test bench.

1.1 I2C

1.1.1 Introduction

I2C is a half-duplex bidirectional synchronous serial bus, where multiple devices can be connected to a data bus. Data exchange is always from a single master to one or more slaves, always at the initiative of the master. However, nothing prevents a component from changing from master to slave and vice versa.

The connection is made via two lines :

- SDA (Serial Data Line): bidirectional data line.
- SCL (Serial Clock Line): bidirectional synchronization clock line.

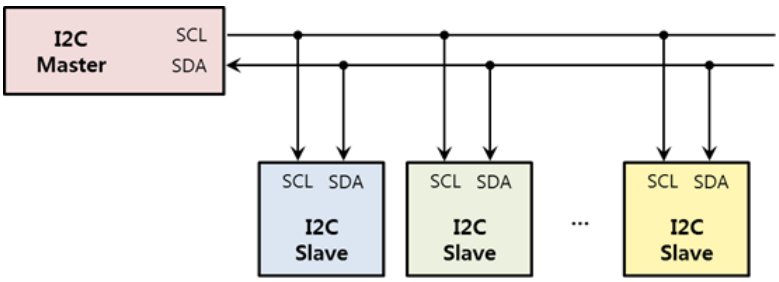


Figure 1.1: I2C Master-Slave

We will use a special machine state representation for this project: The component always starts in the ready state and waits for the ena signal to lock into a control. The slv_ack1 state captures and then checks the slave acknowledgement. The rw command determines whether the component writes data to the slave (wr state) or receives data from the slave (rd state). Once completed, the master captures and verifies the slave's response (slv_ack2 state).

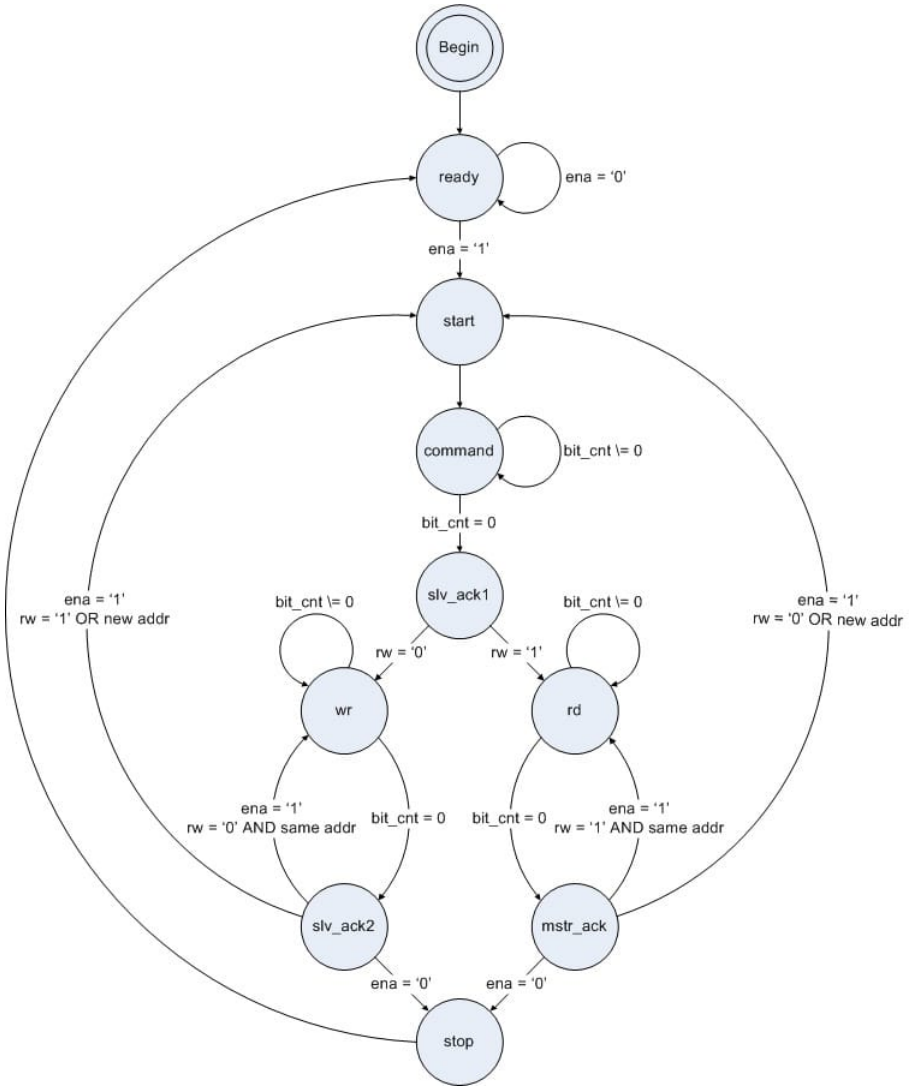


Figure 1.2: I2C Master State Machine

Once the master has completed a read or write operation and the ena signal does not trigger a new command, the master generates the stop condition (stop state) and returns to the ready state.

1.1.2 I2C driver

In this section, we will present a tutorial on how to install a VHDL code:

- Launch Quartus II-64bits
- Create a VHDL projet: File → New Project Wizard → Click "Next" until the new project is created
- Create a new file in the project
- Paste the VHDL code
- Save the file and the project
- Compile (use the shortcut represented with a play symbol)

From that moment, we incorporated a single I2C master component for a single master bus, written in VHDL by Scott Larson for use in FPGAs.

1.1.3 I2C test bench

The next step is to perform a Test Bench. A Test Bench is a VHDL module with which the behavior of a code is tested. In order to do it properly, here are the few important steps:

- Create the VHDL Test Bench
- Go to Assignement → Settings → General: Make sure that file in top-level entity correspond with the Test Bench
- Go to Assignement → Settings → EDA Tool Settings → Simualtion : Make sure that the Tool name is Modelsim-Altera
- Compile: Make sure that there aren't any errors
- Go to Tools → Run Simulation Tools → RTL simulation
- Openning of Modelsim
- Go to Compile → Compile → Check his own Test Bench
- Go to Library Windows → work → our Test Bench → Architecture of our Test Bench
- Objects Appearance
- Select all the objects and add to a new wave
- Select the simulation time and run

This is the test bench we have built for I2C data bus application:

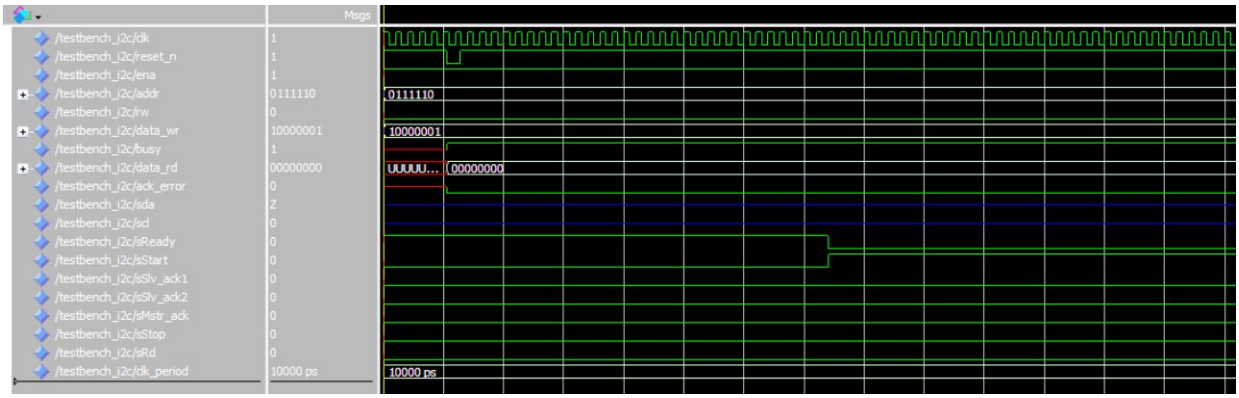


Figure 1.3: I2C Master test bench

Chapter 2

Path

The purpose of the application is to manage the state machine linked to the temperature sensors. To begin with, it takes care of choosing the resolution of the measurement. In our case, we set it to 10 bits to have an accuracy to within 0.25C. Then, the application reads the data sent by the temperature sensor.

2.1 Entity

```
ENTITY Path IS
  PORT(
    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);
END Path;
```

Figure 2.1: Inputs and Outputs of the Path

2.2 Architecture

2.2.1 Reset

By default, we initialize all logical variables using the reset. The reset is defined as a condition on the state of the reset_n bit. The condition is true, when reset_n is equal to 1 and false otherwise. It is therefore an asynchronous function which is independent of the clock. Either the condition is not respected, then we stay on the state machine. This state machine

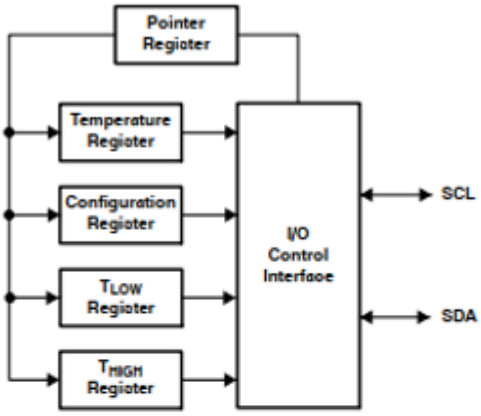
runs synchronously on the rising edge of the clock. Either the condition is satisfied, then you leave the state machine. From now on, the value 0 is assigned to ena and state 1 to state.

```
IF(reset_n = '0') THEN
        ena<='0';
        state<=etat1;
```

Figure 2.2: Reset of the Path

2.2.2 State Machine

Resolution Adjustment: The resolution is not automatically adjusted. First, with the state1 of the state machine, we have to access the configuration register. The configuration register (figure[] table 6) is an 8-bit read/write register used to store the bits that supervise the operating methods of the temperature sensor.



. Internal Register Structure of TMP100 and TMP101

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

. Pointer Register Byte

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)

Pointer Addresses of the TMP100 Registers

Figure 2.3: Setting the resolution

hat's why we start by giving the Register Bits (P1 and P0) of the Reg-

ister pointer the values zero and one to go to the Register configuration (Figure 2.3). The register pointer corresponds to the data_wr in our code. The other bits of the register pointer remain zero by default. During this step, we also define the device address in addr. We apply the value '0' to rw which means that we are in write state. When all this is done, we set the ena to one to say that we are ready.

When the register address is sent to step 1 we wait for an ack from the slave to go to step 2.

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

Configuration Register Format

Figure 2.4: Configuration Register

When we are at step 2 in the configuration register, we will change bits R1 and R0 which are related to the resolution. As previously announced, we wish to have a resolution of 0.25C with 10 bits. So we set R1 and R0 to 0 and 1 respectively. The configuration register now coincides with data_wr.

When the resolution is set, we are still waiting for an ack from the slave to go to step 3.

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

Resolution of the TMP100

Figure 2.5: Resolution

Step 3, we change the state of the ena to '0' to indicate that we have finished the modification. By sending ena=0, we find ourselves in the stop state of the IC2 which returns us busy to '0' and thus allows us to go to step 4 which corresponds to the temperature reading.

```
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 2.6: Code of Resolution Adjustement, state: 1,2, 2bis and 3

Temperature Reading: When we are at step 4, we will modify the registers bits (P1 and P0) to switch from configuration register mode to temperature register mode. The value to be assigned to P1/P0 is "0/0" respectively. At this point, the sensor will send the measured temperatures. We reset rw to '0' to be in write mode and change the ena to 1. We are still waiting for the slave ack to continue in the state machine.

Step 5 starts after receipt of the acknowledgement, the sensor will transmit the data equivalent to the temperature encoded in 2 bytes. To prevent the IC2 from passing through the stop state which would cancel the read request, we set R/W to 1 in read mode. The driver will then record the data sent by the sensor. The master will then send an acknowledgement once this is done. This allows the passage to state 6.

Step 6 is there to ensure correct reception of the 16 bits. For this reason, it waits for a second acknowledgement from the master. This is followed in step 7 by a zero assignment to the ena to signal the end of the reading and the beginning of the stop state of the IC2.

The last state of the status machine is waiting for iStop='1' reception. It closes the state machine and we can start the reading process again. In the event of a problem, we always return to step 1.

```
WHEN etat4 =>
    data_wr<="00000000";
    rw<='0';
    ena<='1';
    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;
WHEN OTHERS =>
    ena<='0';
    state<=etat1;
```

Figure 2.7: Code of Temperature Reading, state: 4, 5, 6, 7, 8 and default state

Chapter 3

7-segment display

For this project, we decided to display the temperature on 7-segment displays. To achieve this, we have to go through two preliminary steps that are the BCD and the encoding for the display.

3.1 BCD

IC2 returns the data read by the sensors in the form of a byte. The byte is unsigned so we can read temperatures from 0 to 255C. The 8 information bits of the register are accessible on the data_rd output of the IC2.

The problem is that the display only shows numbers from 0 to 9. So we have to transform a number between 0-255 into three separate digits that represent hundreds, tens and units. Each of these digits will be encoded in BCD (Binary Coded Decimal) and then used by the encoder. The BCD converts the decimal digits into a predetermined number of bits. For our encoder, we need each digit to be represented in 4 bits. It makes sense to encode on 4 bits because it gives us the possibility to cover the numbers from 0-15.

However, we are already working with a binary number. There is a way to go directly from a binary number to a BCD notation of that byte. This is the "double dabble" algorithm. The algorithm gives us the means to transform an 8-bit number into three 4-bit digits each.

3.1.1 Double Dabble Algorithm

We start by creating an 8-bit variable "temp" that stores the binary number "BinIn" to be converted. Next, we build a BCD vector to store the result of the algorithm. This vector must have a size of $4\times$ ceil[1] $(BinIn/3)$ bits.

[1]ceil is a mathematical function that rounds the number up to the next integer.

Consequently, BDC is composed of 12 bits with: bits 0-3 for units, 4-7 for tens and 8-11 for hundreds. All bits of BDC are initialized to zero.

```
-- temporary variable
variable temp : STD_LOGIC_VECTOR (7 downto 0);

-- variable to store the output BCD number
-- organised as follows

-- hundreds = bcd(11 downto 8)
-- tens = bcd(7 downto 4)
-- units = bcd(3 downto 0)
variable bcd : UNSIGNED (11 downto 0) := (others => '0');

-- by
-- https://en.wikipedia.org/wiki/Double_dabble

begin
  -- zero the bcd variable
  bcd := (others => '0');

  -- read input into temp variable
  temp(7 downto 0) := binIN;
```

Figure 3.1: Variables for Double Dabble Algorithm

The algorithm (Figure 4.3) is a loop that runs as many times as the length of BinIn (8 times here). At each iteration, we shift one bit to the left in tmp and BDC. The value of the bit with the highest "temp" weight is then placed on bit 0 of BDC. When a group of 4 bits (unit, ten, hundred) is greater than or equal to 5, then 3 is added to the result of the 4 bits and the whole is shifted to the left.

	BDC		BinIn=temp	
centaine	dizaine	unité	178	
0000	0000	0000	10110010	initialisation
0000	0000	0001	01100100	décalage à gauche
0000	0000	0010	11001000	décalage à gauche
0000	0000	0101	10010000	décalage à gauche
0000	0000	1000	10010000	comme unités>=5 on a ajouté 3 à unités
0000	0001	0001	00100000	décalage à gauche imposé par la condition
0000	0010	0010	01000000	décalage à gauche
0000	0100	0100	10000000	décalage à gauche
0000	1000	1001	00000000	décalage à gauche
0000	1011	1100	00000000	comme unités>=5 on a ajouté 3 à unités, centaines>=5 on a ajouté 3 à centaines
0001	0111	1000	00000000	décalage à gauche imposé par la condition
				total de 8 décalages donc tous les bits sont passé, on a finit
1	7	8		=valeur

Figure 3.2: Example of how the algorithm works

The addition of three followed by the lag is difficult to understand. In fact, when you add 3 to the value 5, you make the value 5 match 8. Then we double the value by shifting left. We finish by making base 10 and base 16 correspond.

```
-- cycle 8 times as we have 8 input bits
-- this could be optimised, we do not need to check and add 3 for the
-- first 3 iterations as the number can never be >4
for i in 0 to 7 loop

  if bcd(3 downto 0) > 4 then
    bcd(3 downto 0) := bcd(3 downto 0) + 3;
  end if;

  if bcd(7 downto 4) > 4 then
    bcd(7 downto 4) := bcd(7 downto 4) + 3;
  end if;



  -- hundreds can't be >4 for a 8-bit input number
  -- so don't need to do anything to upper 4 bits of bcd

  -- shift bcd left by 1 bit, copy MSB of temp into LSB of bcd
  bcd := bcd(10 downto 0) & temp(7);

  -- shift temp left by 1 bit
  temp := temp(6 downto 0) & '0';

end loop;
```

Figure 3.3: Code Double Dabble Algorithm

3.1.2 BCD test bench

To make the test bench work, we use the routine (section 1.1.3, page 6) by selecting "*TestBench_BCD.vhd*". We're going to vary the binIN input (255, 0 et 178).

/	BinIN	hundred	ten	unit
decimal	255	2	5	5
binary	11111111	0010	0101	0101
decimal	0	0	0	0
binary	00000000	0000	0000	0000
decimal	178	1	7	8
binary	10110010	0001	0111	1000

Table 3.1: Expected result with BCD

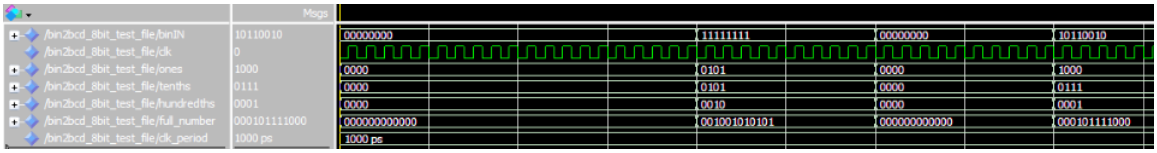


Figure 3.4: Test bench BCD

We observe that the numbers are well separated into three digits hundred, ten and unit. It matches with the result we predicted.

3.2 Encoder7

The Encode 7 program simply maps a 4-bit BDC number to a binary code that controls the 7 segments.

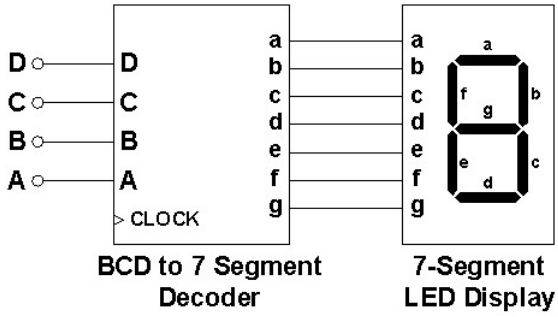


Figure 3.5: Schematic representation of the operation of the encoder

This binary code consists of 7 bits. Each bit controls the status of a 7-segment LED. The leds are generally cathodes. The logic levels of the cathode are '0' when the led is on and 1 when the led is off. A table allows to easily understand the conversion.

Nombre	Cathode_a	Cathode_b	Cathode_c	Cathode_d	Cathode_e	Cathode_f	Cathode_g	code binaire
0	bas	bas	bas	bas	bas	bas	haut	0000001 7'b
1	haut	bas	bas	haut	haut	haut	haut	1001111 7'b
2	bas	bas	haut	bas	bas	haut	bas	0010010 7'b
3	bas	bas	bas	bas	haut	haut	bas	0000110 7'b
4	haut	bas	bas	haut	haut	bas	bas	1001100 7'b
5	bas	haut	bas	bas	haut	bas	bas	0100100 7'b
6	bas	haut	bas	bas	bas	bas	bas	0100000 7'b
7	bas	bas	bas	haut	haut	haut	haut	0001111 7'b
8	bas	bas	bas	bas	bas	bas	bas	0000000 7'b
9	bas	bas	bas	bas	haut	bas	bas	0000100 7'b

Figure 3.6: Summary table of binary codes

```
case BCDin is
when "0000" =>
Seven_Segment <= "0000001"; ---0
when "0001" =>
Seven_Segment <= "1001111"; ---1
when "0010" =>
Seven_Segment <= "0010010"; ---2
when "0011" =>
Seven_Segment <= "0000110"; ---3
when "0100" =>
Seven_Segment <= "1001100"; ---4
when "0101" =>
Seven_Segment <= "0100100"; ---5
when "0110" =>
Seven_Segment <= "0100000"; ---6
when "0111" =>
Seven_Segment <= "0001111"; ---7
when "1000" =>
Seven_Segment <= "0000000"; ---8
when "1001" =>
Seven_Segment <= "0000100"; ---9
when others =>
Seven_Segment <= "1111111"; ---null
end case;
```

Figure 3.7: Encoding code

3.2.1 Encoder 7 test bench

To run the test bench, we use the routine (section 1.1.3, page 6) by selecting "*TestBench_Encoder7.vhd*". Then, we test the numbers from 0 to 9 to verify that all the numbers are correctly coded for the 7 segment.

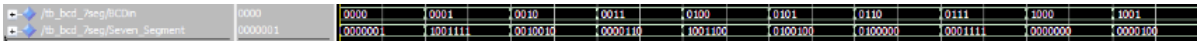


Figure 3.8: Encoding code

Chapter 4

Applications

The last part of this project consists in assembling the different drivers in order to build an accomplished project. However, it's not as complicated as it sounds, you just have to be careful to take all the input and output signals from each of the different VHDL files. This section will talk about two versions of final projects:

- Projet that takes into account the I2C communication
- Projetsim when we fix the input data

4.1 Entity

```
entity Projet is
port
    (
-- Place ports here
      iclk         : in      STD_LOGIC;
      reset_n      : in      STD_LOGIC;
        HEX1         : out std_logic_vector(6 downto 0);
        HEX10        : out std_logic_vector(6 downto 0);
        HEX100   : out std_logic_vector(6 downto 0); --M
        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

    );
end entity;
```

Figure 4.1: Inputs and Outputs of the Projet

4.2 Projet

As you can see on the I2C comunication work properly, the value of ena remains at zero because we don't have the sensor.And we notice that it works well because we don't have random values.

So if we had the opportunity to validate the code in the laboratory, we would notice a variation in ena as well as the data received.

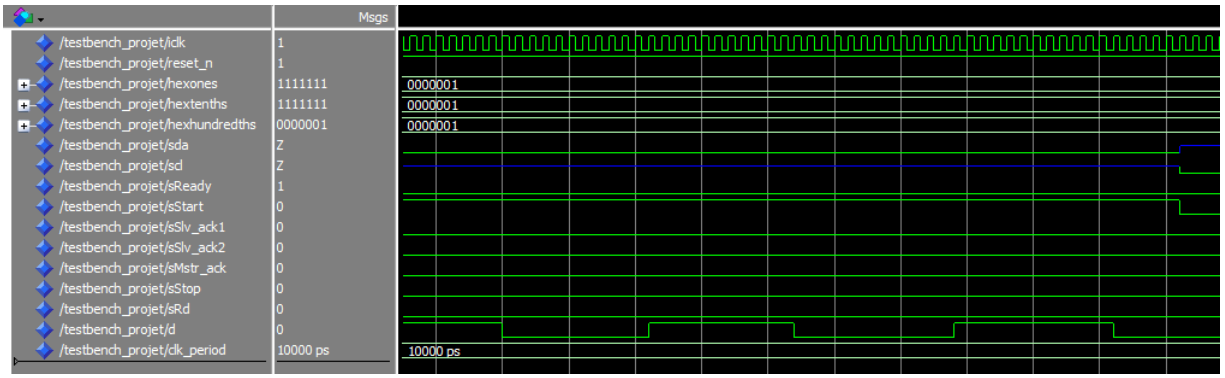


Figure 4.2: Test bench Projet

4.3 ProjetSim

However, in this test bench we arbitrarily set the read measurement in order to observe in output if the same thing is displayed on the seven-segment.

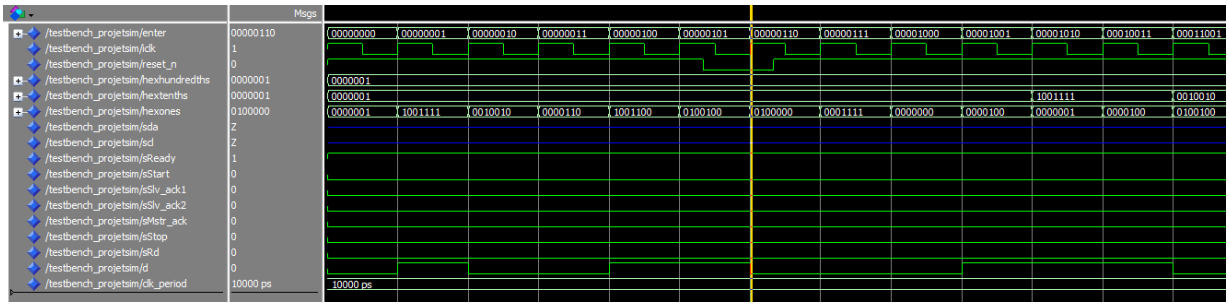


Figure 4.3: Test bench ProjetSim

Conclusion

This tutorial project allowed us to become familiar with VHDL coding but also to become familiar with the use of test bench.

The difficulty of this work lies in the fact that we didn't have access to the lab because of the lockdown. This required a lot of validation work and despite this, our various test benches show us encouraging results for future applications.

Now we just have to put everythings in our card and see if it works perfectly!

Bibliography

[1] Double dabble. https://en.wikipedia.org/wiki/Double_dabble.

[2] Tmp100ep digital temperature sensor with i2c interface. july 2005- revised october 2013.

[3] Adam Blake Bell. Alu-design-in-vhdl-using-de1-soc-fpga-kit. https://github.com/adambb2/ALU-design-in-VHDL-using-DE1-SOC-FPGA-kit, 2018.

[4] Scott Larson. I2c master (vhdl). https://www.digikey.com/eewiki/pages/viewpage.action?pageId=10125324, 2020.

[5] Van Loi Le. Vhdl code for seven-segment display on basys 3 fpga. https://www.fpga4student.com/2017/09/vhdl-code-for-seven-segment-display.html, 2020.