ECE437: Sensors and Instrumentation

Lab 5: Temperature Sensor and Debugging Verilog Code

Introduction

Temperature sensors are ubiquitously used on daily basis. For example, when we check weather reports on our smartphones, data from temperature sensors placed in the city we are interested is automatically collected and upload on the cloud for easy access. We can find the weather conditions in remote parts of the Earth at an instance notice thanks to cheap and compact temperature sensors. This access to sensory information is possible due to proliferation of inexpensive temperature sensors that are easily integrated with digital platforms, such as FPGAs, microcontrollers and PCs. How do temperature sensors communicate with PCs? What is their precision and accuracy? These are some of the questions we will explore in this lab.

Our custom sensor board is equipped with several different sensors, such as temperature, humidity and others. All of these sensors communicate with the FPGA via a serial protocol called I^2C. In this lab exercise, you will get familiar with the I^2C protocol and write a finite state machine that will enable you to acquire data from this sensor and transmit it to the PC. As you develop your finite state machine for the I^2C protocol, you will also learn how to debug your Verilog code using JTAG tools.

Relevant Documents for this Lab

Required reading material for this lab:

1. The temperature sensor used in our custom board is from Analog Devices and the part number is ADT7420. This document contains all information necessary to implement the I^2C protocol and you should get familiar with it. The datasheet for this sensor is located on the course website.
2. We have provided a Verilog code for part of the finite state machine for the I^2C protocol on the course website: *JTEG_Test_File.v, I2C_Transmit.v* and *ClockGenerator.v*

Additional reference material:

1. A tutorial for the I^2C protocol can be located on the course website. This document explains the I^2C protocol and different timing requirements necessary to implement a successful finite state machine.
2. Vivado tutorial for programming and debugging.

The goals of this lab are:

1. Learn how to use the debugging tools in Vivado.
2. Design Verilog code to implement finite state machine to enable communication via I^2C protocol with temperature sensor.

Prelab Questions (30 pts):

1. How many signals does I^{2}C protocol use to send and receive data between devices? What are these signals and describe their functionality? (5 pts)
2. How do you start and end serial communication with I^{2}C protocol? Be specific about the timing relationship between Serial Data Line (SDA) and Serial Clock Line (SCLK) signal. Draw the necessary timing diagrams to clarify your point. (5 pts)
3. What is the maximum frequency for the SCL signal? You should check for this information in the temperature sensor data sheet as it is device specific. (5 pts)
4. After how many transfer bits from the master to the slave device, does the slave device sends back an acknowledgment? (5 pts)
5. How many bits is the default temperature output? (5 pts)
6. How do you set the address of the temperature sensor? How many different addresses can be used to communicate with sensor? If there are more than 1, why do you think there are multiple possible addresses? (5 pts)

Debugging Verilog Code with JTAG

Traditionally, debugging digital circuits was accomplished using benchtop logic analyzers that were connected to digital circuits through the physical I/O pins of an FPGA or a digital integrated circuit. If internal signals needed to be monitored from the FPGA, these signals had to be assigned to I/O pins, which could increase the cost and complexity of the design. Hence, debugging digital circuit was virtually impossible due to the limited number of input/output pins on the integrated circuits.

However, modern FPGAs provide an alternative way of debugging which excludes the use of benchtop logic analyzers. By incorporating specific circuits into the design which can monitor the rest of the system and by sending information from these circuits to a computer, we can debug complex designs with few additional resources. In Vivado, Integrated Logic Analyzers (ILAs) cores can be used to sample and forward any signals in a design to a computer for further analysis.

In this lab, we will be using a Xilinx Platform Cable USBII programmer/debugger along with the OpalKelly XEM7310 integration module. Before you begin, make sure that you connect the programmer/debugger to both the JTAG port on the integration module and a USB port on a computer. See the figure bellow for reference.

Why would you want to debug your code? Simulation results are not always accurate. Simulation results provide a first step understanding of the behavior of your Verilog code. However, when the code is synthetized and implemented, the behavior of the code can be different compared to your simulation results. Remember that no two implementations are the same even though your code has remained the same. The debugger will help you get a better understanding of your Verilog code and how it is implemented in the digital logic in the FPGA logic fabric. Sometimes you really don't know why your code doesn't work after you run your code. You really want something like GDB in C++ to enable you to examine the value of a certain variable. This debugger is a hardware version of GDB.

Setting Up Verilog Code for Debugging

In this tutorial, we will learn how to debug a finite state machine (FSM) implemented in Verilog. The FSM code example implements the states related to sending and receiving the first 9 bits of the I^2C protocol. Although the FSM code is generalizable to any I^2C protocol, we will be using the documentation provided for the temperature sensor AD7420 to implement this protocol. The example code also provides a good tutorial how to use modules. As your code becomes very lengthy, using modules will make your code easier to read and manage. Be careful: instantiating modules is not the same as calling a function in higher level programming languages such as C/C++ or Python. There are fundamental differences between these two that will become apparent throughout this class.

Before proceeding with the tutorial, download the following three files from the lab website:

1. JTEG_Test_File.v – This is the top Verilog module of the example code. This module will instantiate two modules: the first one will be the I^2C FSM module and the second will be an ILA module, which will enable us to debug the code.

2. I2C_Transmit.v – This is the I^2C FSM module which contains all the states necessary to send 8 bits from the FPGA to the temperature sensor and receive an acknowledgment from the sensor as the 9th bit.

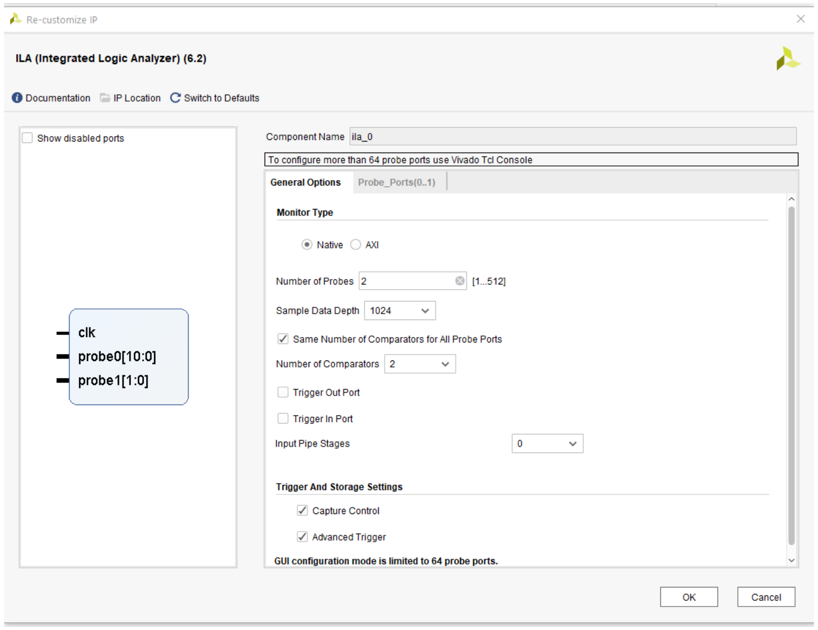
3. ClockGenerator.v – This is module which generates two clocks: one for running the FSM and the second one for running the sampling rate for the ILA.

The next step is to create a new project and add these three files to the project as described in the previous labs. You can add these three files by selecting *Add Sources* in the **Flow Navigator** window. Select *Add or create design sources* in the **Add Sources** window. In the next window, select all three Verilog files that you previously downloaded and select *Copy sources into project* option. The next step is to add the constrains file, *xem7310_v1.xdc*, in your project. You can do this by either copying this file from the previous projects or download it from the course website.

After you have created the new project, open the top module file, *JTEG_Test_File.v*. Look through the code in this file. You will notice that the code is very simple. It contains few wire declarations and two modules: *I2C_transmit* and *ila_0*. The *I2C_transmit* module contains part of the FSM for the I^{2}C protocol communication. This module can be synthetized and implemented by itself if you create a project that only contains this file. Once we have implemented the correct FSM, we do not need the JTEG debugging code in our module. In fact, JTEG debugging prevents optimization of the signals you will be observing and will render not fully optimized solution. The module you are debugging should be self-sufficient and easy to plugin (or instantiated) in the debug code. The main debug code, *JTEG_Test_File.v*. should be simple and only instantiate few different modules. Once you are done with the debugging phase and have the correct FSM, you will only synthetize and implement the *I2C_transmit* module without the *JTEG_Test_File.v* file. This will be done by creating a new project which will contain only *I2C_transmit.v, ClockGenerator.v* and *xem7410_v1.xdc* files.

The next critical step for your project is to generate the ILA core from the IP library provided in Vivado. The ILA module is instantiated in the *JTEG_Test_File.v*. However, there is no Verilog file in the project, so the synthesizer will not know where to locate this module and will give an error. To add the ILA module, follow these steps:

1. In the **Flow Navigator**, go to **PROJECT MANGER** and click on *IP Catalog*.
2. Next select *Debug & Verification -> Debug -> ILA (Integrated Logic Analyzer)*.
3. We will be monitoring two sets of signals: data signals and trigger signals. The data signals will be comprised of the signals related to the I^{2}C protocol. The trigger signals will be used to control when the ILA will collect signals from the FPGA. To monitor these two sets of signals, we will set the *Number of ports* and *Number of Comparators* to 2. Also check the *Capture control* and *Advance Trigger* box in the window. Here is how this window should look like:



ILA (Integrated Logic Analyzer) (6.2)

Component Name: ila_0

To configure more than 64 probe ports use Vivado Tcl Console

Monitor Type

[✓ Native] AXI

Number of Probes: 2 [1..512]

Sample Data Depth: 1024

✓ Same Number of Comparators for All Probe Ports

Number of Comparators: 2

☐ Trigger Out Port

☐ Trigger In Port

Input Pipe Stages: 0

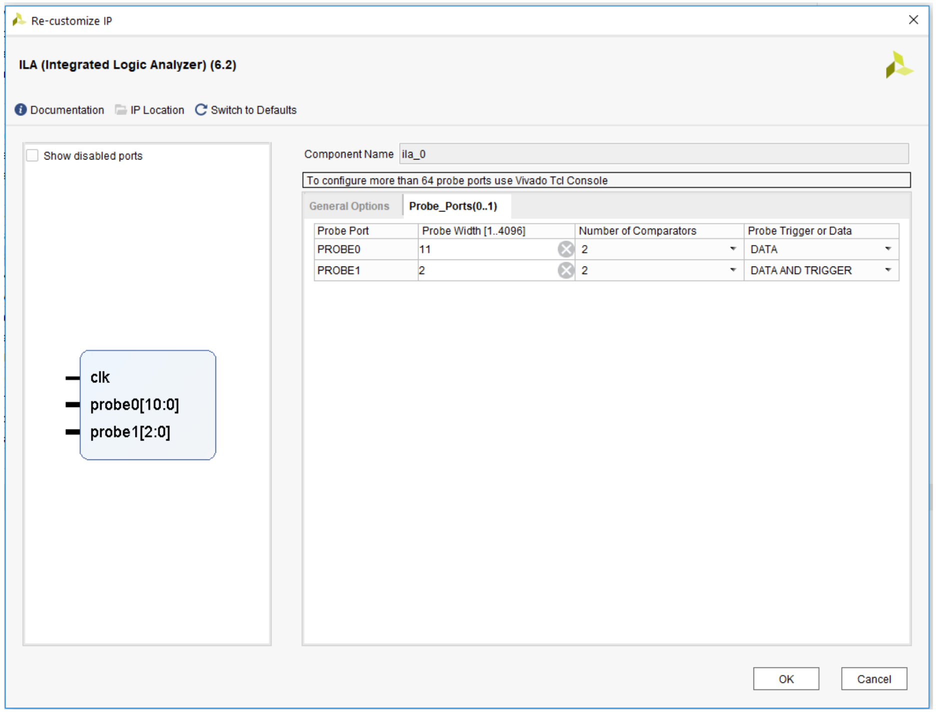
Trigger And Storage Settings

✓ Capture Control

✓ Advanced Trigger

GUI configuration mode is limited to 64 probe ports.

4. Click on the *Probe_Ports(0..1)* folder in this window. You will declare the bit depth for the two probe ports and the type of data they will collect. The first probe port will be only for data and it will be 11-bits long because we will be monitor 11 signals in total. The second probe port will be used for both data and trigger and will be 2-bits long. Here is how this window should look like


Re-customize IP

ILA (Integrated Logic Analyzer) (6.2)

Show disabled ports

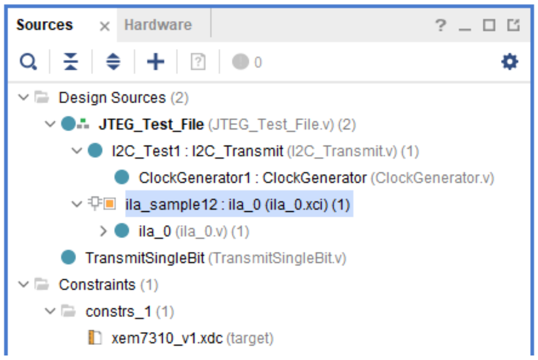
Component Name ila_0

To configure more than 64 probe ports use Vivado Tcl Console

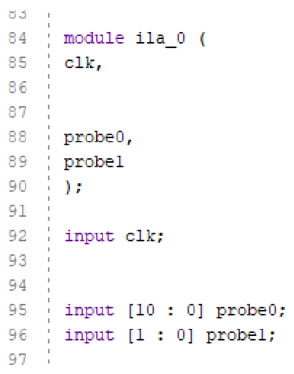
Probe Port	Probe Width [1..4096]	Number of Comparators	Probe Trigger or Data
PROBE0	11	2	DATA
PROBE1	2	2	DATA AND TRIGGER

5. Click on the *OK* button. The ILA core will start to be generated. Click on the *Generate* button in the **Generate Output Products** window and the ILA will be generated and inserted in the project.

6. Your project should look like this in the **Sources** window:


Design Sources (2)
JTEG_Test_File (JTEG_Test_File.v) (2)
I2C_Test1 : I2C_Transmit (I2C_Transmit.v) (1)
ClockGenerator1 : ClockGenerator (ClockGenerator.v)
ila_sample12 : ila_0 (ila_0.xci) (1)
ila_0 (ila_0.v) (1)
TransmitSingleBit (TransmitSingleBit.v)
Constraints (1)
constrs_1 (1)
xem7310_v1.xdc (target)

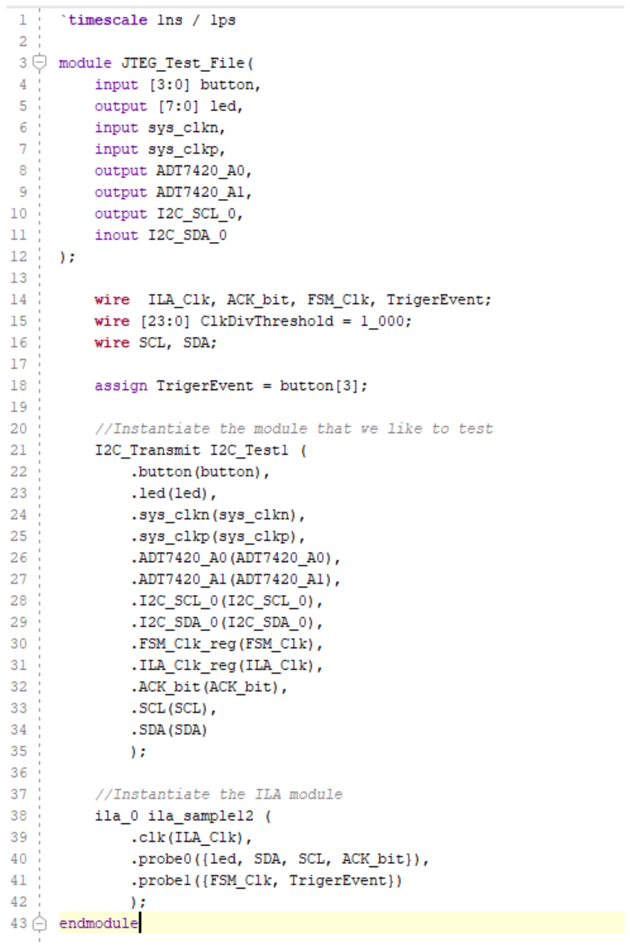
7. Since in the future you will be debugging signals from your own Verilog code, let's take a closer look at how to instantiate an ILA module. Double click on *ila_sample12: ila_0 (ila_0.xci)*. It will take some time for the *ila_0.v* file to appear in the folder. Double click on *ila_0.v* file. Scroll down through the file until you identify the declaration of the module and its variables. You should see the following lines of code:



```
module ila_0 (
clk,


probe0,
probe1
);

input clk;


input [10 : 0] probe0;
input [1 : 0] probe1;

```

The *ila_0* module expects three inputs: *clk*, *probe0* and *probe1*. The *clk* variable is used for sampling the data generated by the FPGA. This clock signal has to be between 1 MHz and ~100 MHz. If you use slower than 1 MHz sampling clocks, the debugger will timeout. Variables *probe0* and *probe1* are the two ports that you declared previously during the ILA generation. You can observe that they are 11- and 2-bit wide, respectively, as specified when we generated the ILA. If you change the port declaration in the ILA using Vivado tools, the variables in the ILA modules are automatically updated.

Let's open up *JTEG_Test_File.v* file by double clicking in the **Source** window. You should see the following code:



```
`timescale 1ns / 1ps

module JTEG_Test_File(
    input [3:0] button,
    output [7:0] led,
    input sys_clkn,
    input sys_clkp,
    output ADT7420_A0,
    output ADT7420_A1,
    output I2C_SCL_0,
    inout I2C_SDA_0
);

    wire  ILA_Clk, ACK_bit, FSM_Clk, TrigerEvent;
    wire [23:0] ClkDivThreshold = 1_000;
    wire SCL, SDA;

    assign TrigerEvent = button[3];

    //Instantiate the module that we like to test
    I2C_Transmit I2C_Test1 (
        .button(button),
        .led(led),
        .sys_clkn(sys_clkn),
        .sys_clkp(sys_clkp),
        .ADT7420_A0(ADT7420_A0),
        .ADT7420_A1(ADT7420_A1),
        .I2C_SCL_0(I2C_SCL_0),
        .I2C_SDA_0(I2C_SDA_0),
        .FSM_Clk_reg(FSM_Clk),
        .ILA_Clk_reg(ILA_Clk),
        .ACK_bit(ACK_bit),
        .SCL(SCL),
        .SDA(SDA)
        );

    //Instantiate the ILA module
    ila_0 ila_sample12 (
        .clk(ILA_Clk),
        .probe0({led, SDA, SCL, ACK_bit}),
        .probe1({FSM_Clk, TrigerEvent})
        );
endmodule
```

The ILA is instantiated at the end of this code and we named that instance *ila_sample12*. There are several variables that are passed to the ILA module. The first one is *ILA_Clk* variable, which is derived from the system clock and runs at 20 MHz. Hence, the sampling frequency of the variables in your FPGA will occur at 20 MHz. Variable *probe0* is 11-bit long and is formed by concatenating 8-bit wire *led* and three 1-bit long wires, namely *SDA*, *SCL* and *ACK_bit*. These are the four signals that will be monitored during the debugging phase.

Variable *probe1* is 2-bit long and is formed by concatenating two 1-bit long variables, namely *FSM_Clk* and *TrigerEvent*. These two variables will be used for triggering purposes. Variable *TrigerEvent* will be used to signal the start of data collection and variable *FSM_Clk* will dictate when the sampling clock, ILA_Clk, is enabled. Note that *TrigerEvent* is mapped to the fourth button (i.e. *button[3]*) on the sensor board. This means that when we press this button, the debugger will start to collect data. However, up to this point we have not specified the trigger functionality for these two variables. This will be declared once we setup the debugger in Vivado.

In the code, module *I2C_Transmit* is also instantiated. This is the module that we are interested in testing and debugging. This module is instantiated with the necessary variables. The module also returns several output variables, such as SDA, SCA and others, that we are interested in monitoring with the ILA.

8. The next step is to synthetize, implement and generate a bit file for your code. You can click sequentially on each on of these steps in Vivado or click directly on **Generate Bitstream** in the **Flow Navigator** window. If you only click on **Generate Bitstream**, Vivado will automatically go through synthesis, implementation and bit file generation.

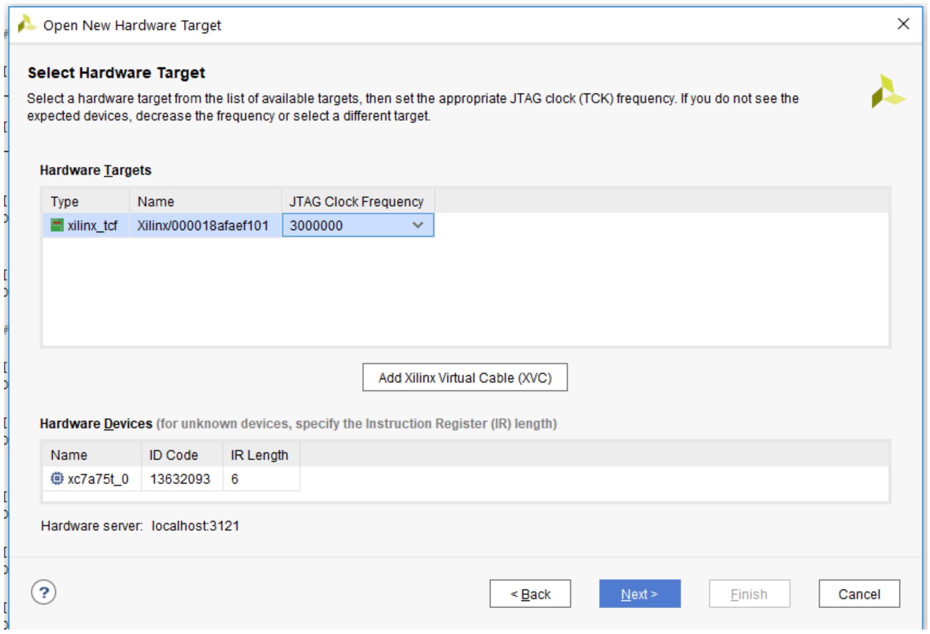
9. The next step is to start the debugger. Under **Flow Navigator** window, click on **Open Hardware Manager**.

10. Next click on **Open Target** and select **Open New Target.** Do not use the **Auto Connect** because Vivado will select communication frequency that will cause error in the data transmission.

11. Click on **Next** on the next two popup windows by leaving the default options (i.e. leave the default options in the **Open Hardware Target** and **Hardware Server Setting** windows).

12. After the JTAG communication has been established, two things will happen. First, a green LED will turn on the Xilinx Platform Cable USBII programmer/debugger. If you see an orange light, something is not connected correctly, and the programmer/debugger is not initialized properly. You will need to fix this problem and rerun these steps.

    Second, you will see the following window. Change the **JTAG Clock Frequency** to 3MHz and click on the **Next** button.


Open New Hardware Target

Select Hardware Target

Select a hardware target from the list of available targets, then set the appropriate JTAG clock (TCK) frequency. If you do not see the expected devices, decrease the frequency or select a different target.

Hardware Targets

Type	Name	JTAG Clock Frequency
xilinx_tcf	Xilinx/000018afaef101	3000000

Add Xilinx Virtual Cable (XVC)

Hardware Devices (for unknown devices, specify the Instruction Register (IR) length)

Name	ID Code	IR Length
xc7a75t_0	13632093	6

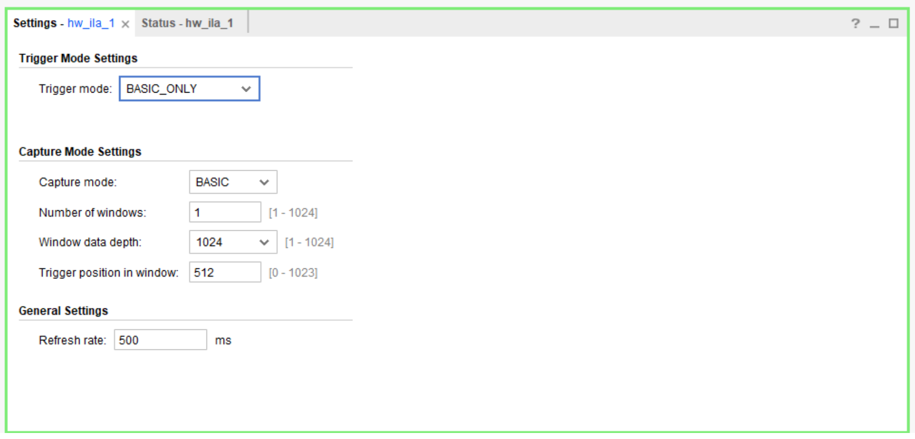
Hardware server: localhost:3121

13. Click on the **Finish** button. At this point, you have established connection between Vivado and the programmer/debugger unit. You are set to start observing signals from our code. But first, we need to program the FPGA with the bit file that we had previously generated. We will load the new bit file through Vivado.

14. Click in the **Flow Navigator** window on **Program Device**. Select the single option that is provided, which is **xc7a75t_0**. In the **Program Device** popup window, click on the **Program** button. Your device will be programmed with the bit file that you had previously generated. You don't need to change the default files that this window has selected.

15. You should see a **Waveform – hw_ila_1** window. We will setup the triggers before we debug the code.

16. To set up trigger signals, go to the **Settings – hw_ila_1** window which should be under your waveform window. Select the **BASIC_ONLY** and **BASIC** option for **Triger mode** and **Capture mode**, respectively. The window should like this:



Trigger Mode Settings

Trigger mode: BASIC_ONLY

Capture Mode Settings

Capture mode: BASIC

Number of windows: 1 [1 - 1024]

Window data depth: 1024 [1 - 1024]

Trigger position in window: 512 [0 - 1023]

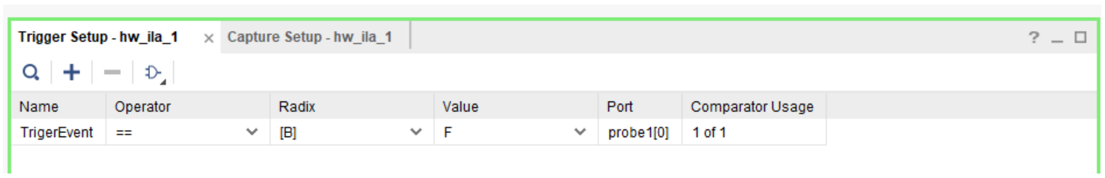
General Settings

Refresh rate: 500 ms

The **BASIC** and **BASIC_ONLY** option enable custom settings for trigger and capture mode. We will declare these conditions in the window adjacent to this one.

17. Go to **Triger Setup – hw_ila_1** window. This window enables you to specify the trigger condition when data will start to be collected by the debugger. The trigger event can be simple logic, such as signal transition. Click on the plus sign and select the *TrigerEvent* variable. Since this signals is connected to *button[3]*, which is an active low signal, the trigger will be defined as the falling edge of this signal. In other words, when *button[3]* is pressed, the debugger will start collecting data.

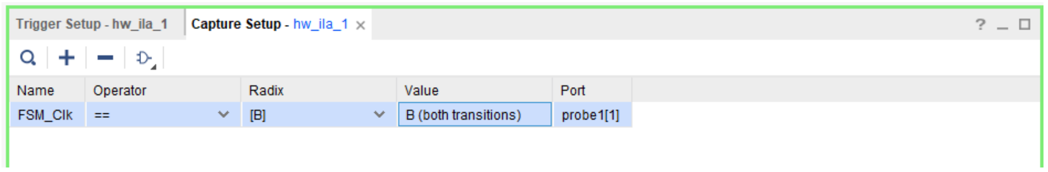
    This window should like this after these selections:


Trigger Setup - hw_ila_1 | Capture Setup - hw_ila_1

Name	Operator	Radix	Value	Port	Comparator Usage
TrigerEvent	==	[B]	F	probe1[0]	1 of 1

18. Next we need to specify when the debugger should sample data. If we don't specify this condition, then data will be collected at 10 MHz sampling rate. Since our FSM runs at much lower speed, the debugger will collect 1024 sample at high frequency and miss important state transitions. Hence, we like to sample data only when the FSM clock changes its state.

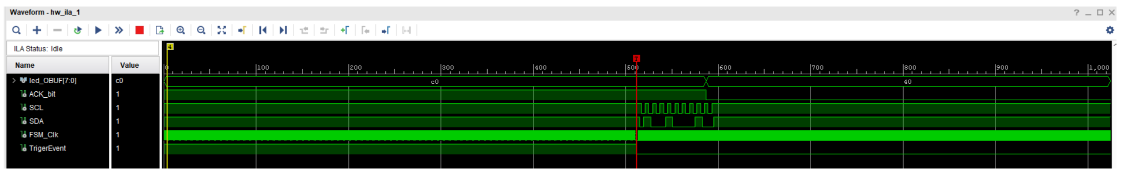
    In the **Capture Setup – hw_ila_1** window, click on the plus sign and select *FSM_Clk*. Since we like to sample the data on both falling and raising edges, we select **B (both transitions)** option. This window should like this:


Trigger Setup - hw_ila_1 | Capture Setup - hw_ila_1

Name	Operator	Radix	Value	Port
FSM_Clk	==	[B]	B (both transitions)	probe1[1]

19. You can run the debugger by clicking on the play button in the **Waveform – hw_ila_1** window. You might want to select the **Statis – hw_ila_1** window so that you can observe the status of your debugger as well. Once you press the play button, the debugger will wait for the trigger before proceeding with the data sampling.

20. Press the fourth button on the custom board. You should observe two things. First, the last LED in the custom board, i.e. *led[7]*, should turn on. Second, the waveform window should be populated with several different waveforms. You should see the following results:


Waveform - hw_ila_1

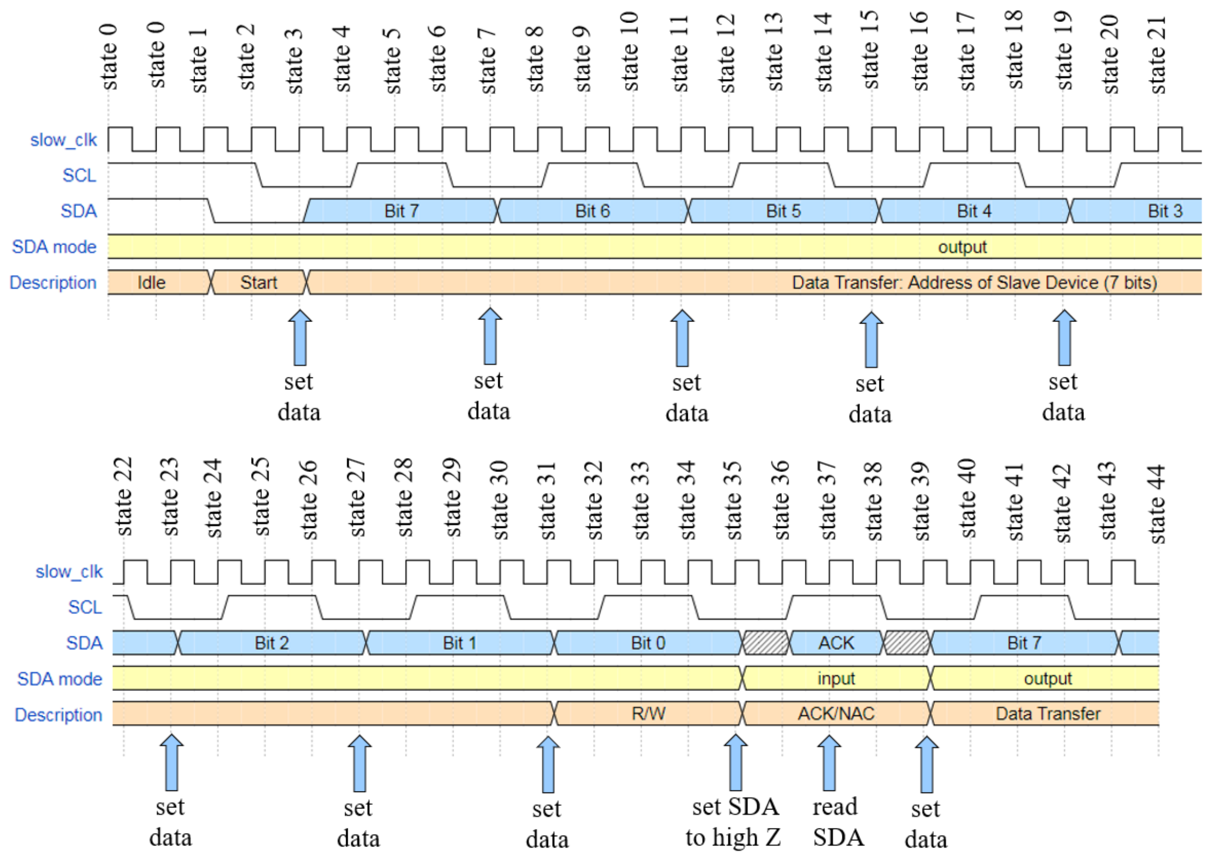
ILA Status: Idle

Name	Value
led_OBUF[7:0]	c0
ACK_bit	1
SCL	1
SDA	1
FSM_Clk	1
TrigerEvent	1

21. You have successfully used JTAG tools to observe signals from your FPGA in real-time.

Finite State Machine for I^{2}C

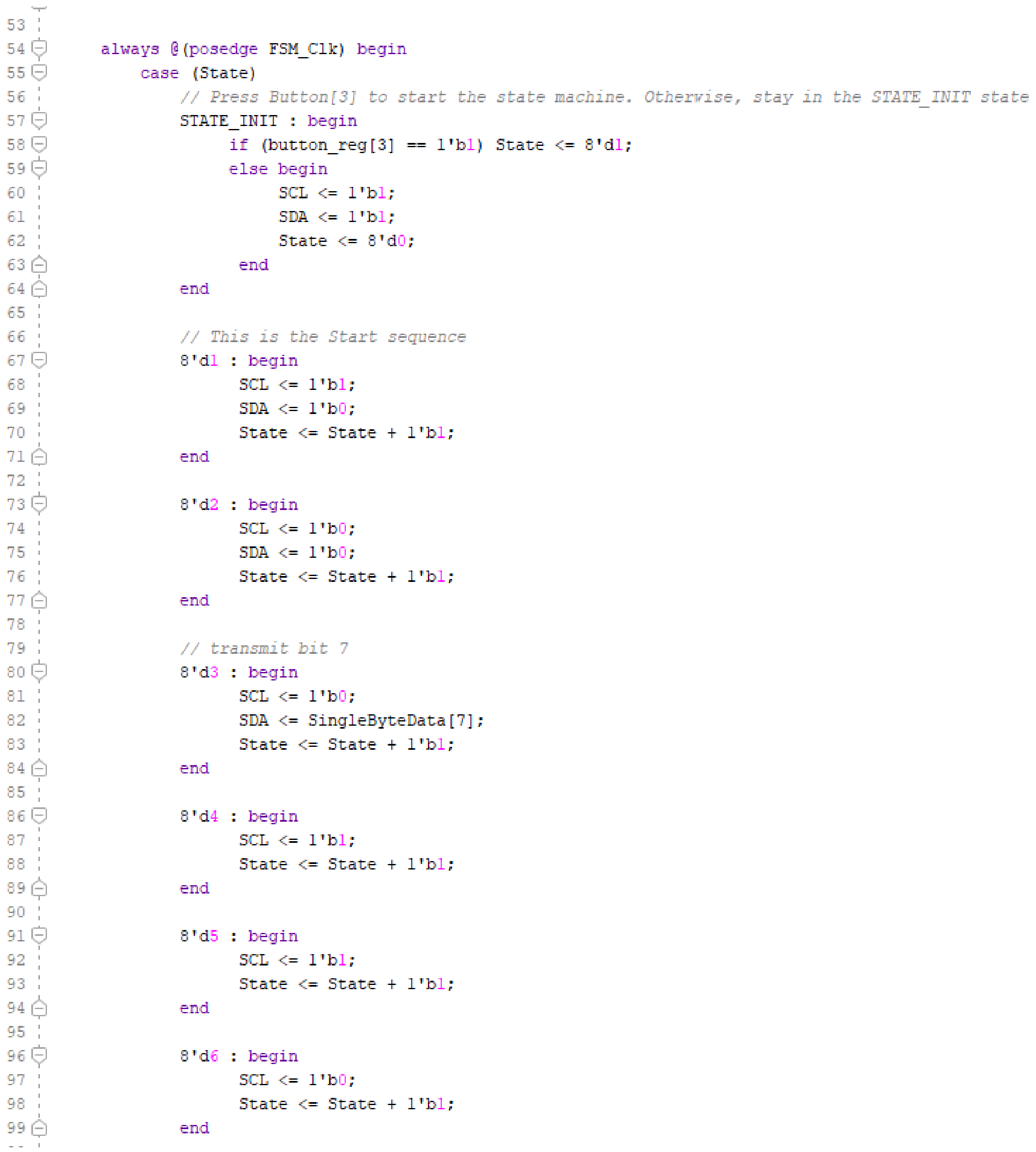
Let's take a closer look at the Verilog code that enabled part of the I^{2}C protocol. To understand the FSM Verilog code, let's examine the FSM block diagram bellow:



This block diagram shows 44 different states that the FSM has to transition through to send 8-bits of data from the FPGA to the temperature sensor and receive 1 bit of data from the temperature sensor. The FSM starts from an Idle state where both SDA and SCL clocks are high. The next state is the Start state, where the SDA and SCL clock must transition in particular sequence. Next the 8 bits of data are sent to the temperature sensor. When a read sequence is initiated, the first 8-bits of data represent the id of the sensor plus the operation mode (read or write). The key thing is to make sure that SDA does not change when SCL is high. SDA is also set only when SCL is low. Once these 8 bits are transmitted, the sensor sends an acknowledgment back to the FPGA.

Open the *I2C_Transmit.v* file by double clicking on its name. In case you cannot find you Verilog files in your project, click on **PROJECT MANGER** in the upper left corner in the **Flow Navigator** window.

As you scroll through the code, you should see the following code:

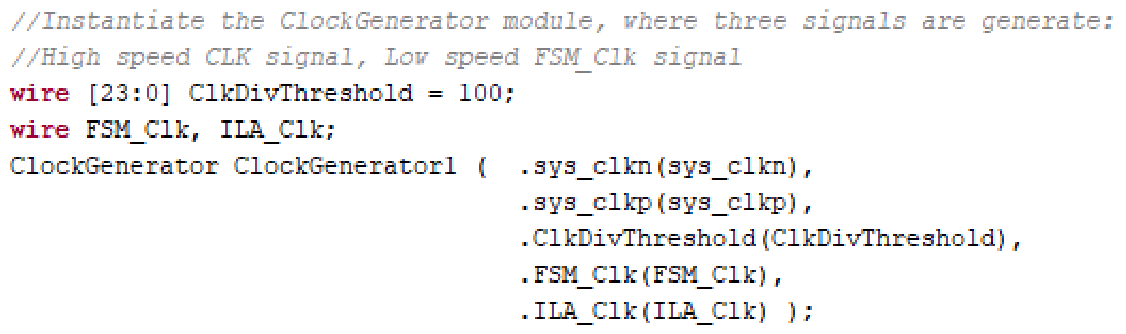


```

    always @(posedge FSM_Clk) begin
        case (State)
            // Press Button[3] to start the state machine. Otherwise, stay in the STATE_INIT state
            STATE_INIT : begin
                 if (button_reg[3] == 1'b1) State <= 8'd1;
                 else begin
                      SCL <= 1'b1;
                      SDA <= 1'b1;
                      State <= 8'd0;
                  end
            end

            // This is the Start sequence
            8'd1 : begin
                  SCL <= 1'b1;
                  SDA <= 1'b0;
                  State <= State + 1'b1;
            end

            8'd2 : begin
                  SCL <= 1'b0;
                  SDA <= 1'b0;
                  State <= State + 1'b1;
            end

            // transmit bit 7
            8'd3 : begin
                  SCL <= 1'b0;
                  SDA <= SingleByteData[7];
                  State <= State + 1'b1;
            end

            8'd4 : begin
                  SCL <= 1'b1;
                  State <= State + 1'b1;
            end

            8'd5 : begin
                  SCL <= 1'b1;
                  State <= State + 1'b1;
            end

            8'd6 : begin
                  SCL <= 1'b0;
                  State <= State + 1'b1;
            end
```

This is part of the FSM code. If you look at this code and the block diagram above, they match one to one. The FSM starts from state=0, which is the STATE_INIT. In this state, both SDA and SCL are high. Once *button[3]* is pressed, the FSM machine goes to state=1, which is the start sequence. To complete the starts sequence, the FSM has to complete state=2. The 7th bit of data is transmitted starting state=3, as shown in the block diagram.

The block diagram above helps in deriving the correct FSM and correctly implementing it in Verilog. It is a good practice to write out your FSM carefully and then code it up in Verilog. Use the simulator to make sure the behavior of the code is correct before synthetizing your code.

The FSM code also instantiates a clock generator module. Here is the code that accomplishes that task:



```
//Instantiate the ClockGenerator module, where three signals are generate:
//High speed CLK signal, Low speed FSM_Clk signal
wire [23:0] ClkDivThreshold = 100;
wire FSM_Clk, ILA_Clk;
ClockGenerator ClockGenerator1 (  .sys_clkn(sys_clkn),
                                  .sys_clkp(sys_clkp),
                                  .ClkDivThreshold(ClkDivThreshold),
                                  .FSM_Clk(FSM_Clk),
                                  .ILA_Clk(ILA_Clk) );
```

Open the *ClockGenerator.m* Verilog file. The code should be easy to follow. It generates an *ILA_Clk* signal which is 10 times slower the 200 MHz system clock. This signal is used to sample data in the ILA module for debugging purposes. The *FSM_Clk* is derived from the *ILA_Clk* and is running around 20 KHz. This is the clock that will be used to execute the finite state machine.

Note how we used modules to make the code easier to read and organize. You should try to modularize your code as much as possible. One thing to remember, instantiating a module is not the same as calling a function in C/C++. You have to careful how you invoke a module.

Checkpoint 1 (40 points)

- Make sure you go over all the steps outlined in this tutorial. Once you complete the debugger tutorial, zoom in the Waveform window from sample number 500 to sample number 600. Once you zoom in this close, you should observe the FSM_clk. Print the waveform window and include it in your report. (10 pts).
- Modify the Verilog code in your project such that you can observe the *State* variable in the waveform window. Plot the following waveforms: SCL, SDA, State ACK_bit, FSM_CLK and TriggerEvent. Zoom in at waveform window and show the first few transistor states of the FSM once the trigger is executed. Make sure you can observe the number of the *State* variable (30 pts).

Checkpoint 2 (60 points)

Following the timing diagram provided on page 19 of the sensor's data sheet, implement the complete FSM for reading temperature data from the sensor. Once you read the data from the sensor, use OK wires to send the data to the PC. Using Python, read the data and display the result on the PC. Since you will be measuring ambient temperature, the result should be around 20 ºC. To get the full credit, your temperature result should be within 10% of the ambient temperature in the lab. Also you need to read at least 10 times the value of the temperature sensor and display the results. If the results vary by more than 10%, you will lose 50% of these points.

- Simulate your FSM and show the results to the TA. Include the timing waveforms in your final report (20 pts).
- Use the JTEG debugging tools to grab the timing waveforms for the read FSM. Make sure that you can view the acknowledge bit, SCL, SDA, FSM_clk an d any other signals that you think are necessary to demonstrate that your FSM is operating correctly. (20 pts).
- Demonstrate to the TA that you can read data from the temperature register. (20 pts)

Post lab questions (20 points)

Q1 What is the maximum speed you can run the SCL signal of the temperature sensor? (5 pts)

Q2 How can you obtain multiple temperature values from the sensor? How would you modify your Verilog and Python code to do this task? You don't need to make these changes in the code. (5 pts)

Q3 What is the minimum and maximum value the temperature sensor can record? (5 pts)

Q4 What is the resolution of the temperature sensor? (5 pts)