Computer Engineering (CESE4130) 2025/26
Lab 1 Manual

Ignacio García Ezquerro
Pantazis Anagnostou
Zacharia Rudge
Shadid Hassan
Amin Yaldagard
Mahmood Naderan-Tahan
Mottaqiallah Taouil
Georgi Gaydadjiev

TU DELFT – EECMS Faculty – QCE Department

Table of Contents

1. Lab 1

1.1. Setup

In this lab, you will be putting your Verilog skills to practice. You will write hardware descriptions for different applications and synthesize and implement these designs using Xilinx Vivado. Then, after synthesis and implementation, you will generate a bitstream of your design to upload to an FPGA board. On the FPGA board, you will test your designs.

During the lab, you will be using Vivado software. You need to install it before the lab. Installation instructions are available in Vivado Setup If you absolutely cannot install Vivado, it will be possible to use one of the lab desktops..

If you are new to Verilog or Hardware description languages, it is recommended that you go through the Verilog Tutorial before the lab. This will help you understand the workflow of how a hardware description program is written and the main building blocks of a Verilog code.

1.2. Content

Download the Vivado project templates used in this exercise here: Vivado_Lab1

This lab will be organized as follows:

Part 1: Asynchronous Encoder & Decoder

In this lab, you will use a binary rotary switch and a gray code rotary switch to perform some basic asynchronous decoders.

Learning Goals of the Lab

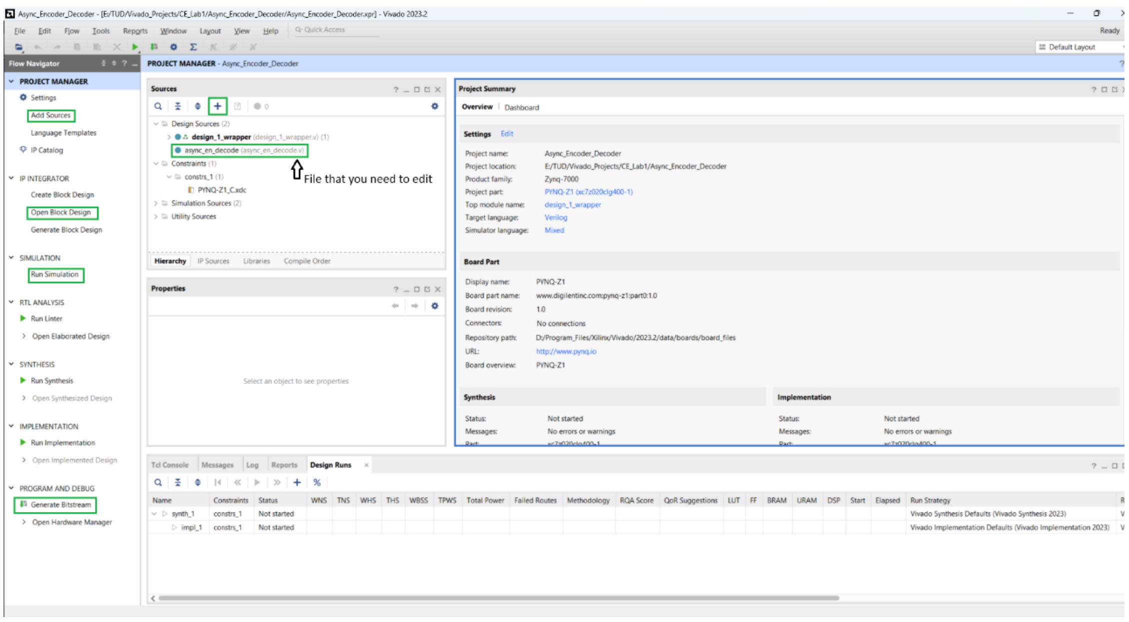
1. Introduction to Vivado and Vivado toolchain

2. Understanding block diagrams, inputs, and outputs of the FPGA
3. Implementing Asynchronous logic

Lab Template:

For all the exercises in this lab, you will receive a basic template for you to work on, except the *bonus* exercise. For this lab, we describe the components in the template. From the next exercises, you are expected to explore the template on your own.

To start the lab, launch the `Vivado` application. Open the project `Async_Encoder_Decoder` in Vivado. It will open the following screen.



File that you need to edit

Sources: Design Sources (2): design_1_wrapper (design_1_wrapper.v) (1); async_en_decode (async_en_decode.v); Constraints (1): constrs_1 (1): PYNQ-Z1_C.xdc; Simulation Sources (2); Utility Sources

Project Summary — Settings

Project name:	Async_Encoder_Decoder
Project location:	E:/TUD/Vivado_Projects/CE_Lab1/Async_Encoder_Decoder
Product family:	Zynq-7000
Project part:	PYNQ-Z1 (xc7z020clg400-1)
Top module name:	design_1_wrapper
Target language:	Verilog
Simulator language:	Mixed

Board Part

Display name:	PYNQ-Z1
Board part name:	www.digilentinc.com:pynq-z1:part0:1.0
Board revision:	1.0
Connectors:	No connections
Repository path:	D:/Program_Files/Xilinx/Vivado/2023.2/data/boards/board_files
URL:	http://www.pynq.io
Board overview:	PYNQ-Z1

Synthesis — Status: Not started; Messages: No errors or warnings

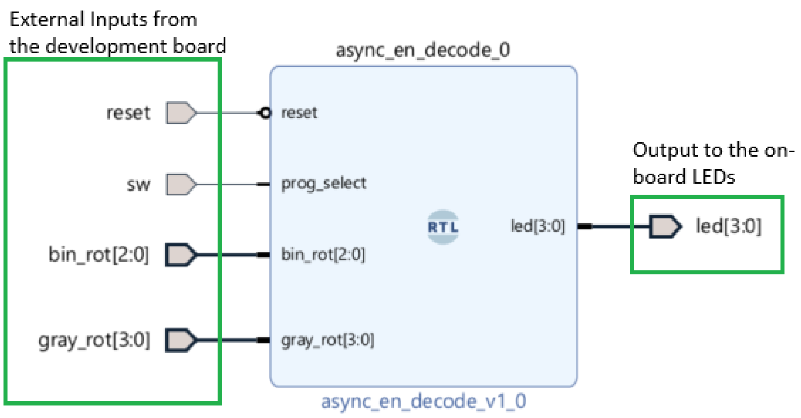
Implementation — Status: Not started; Messages: No errors or warnings

Design Runs

Name	Constraints	Status	Run Strategy
synth_1	constrs_1	Not started	Vivado Synthesis Defaults (Vivado Synthesis 2023)
impl_1	constrs_1	Not started	Vivado Implementation Defaults (Vivado Implementation 2023)

Above you can see the file `async_en_decode.v`, which needs to be edited to complete the assignment. Other functions that you will be using in the lab are also highlighted.

You can now Open the Block Diagram. A block diagram is a visual representation of all the interconnected IPs and Verilog blocks with each other and with external components. The block diagram for this project is shown below.



There are 3 main parts to the above block diagram. 1. Inputs coming into the block diagram from the board. These include: - Reset switch: Mapped to SW1 - SW switch: Mapped to SW0 - Binary Rotary Encoder: Mapped to PMODB[0:2] - Gray Encoder Switch: Mapped to PMOD[4:7] 2. In the middle is the RTL code you will write in the Verilog file. 3. Finally, the output LEDs are mapped to the on-board LEDs

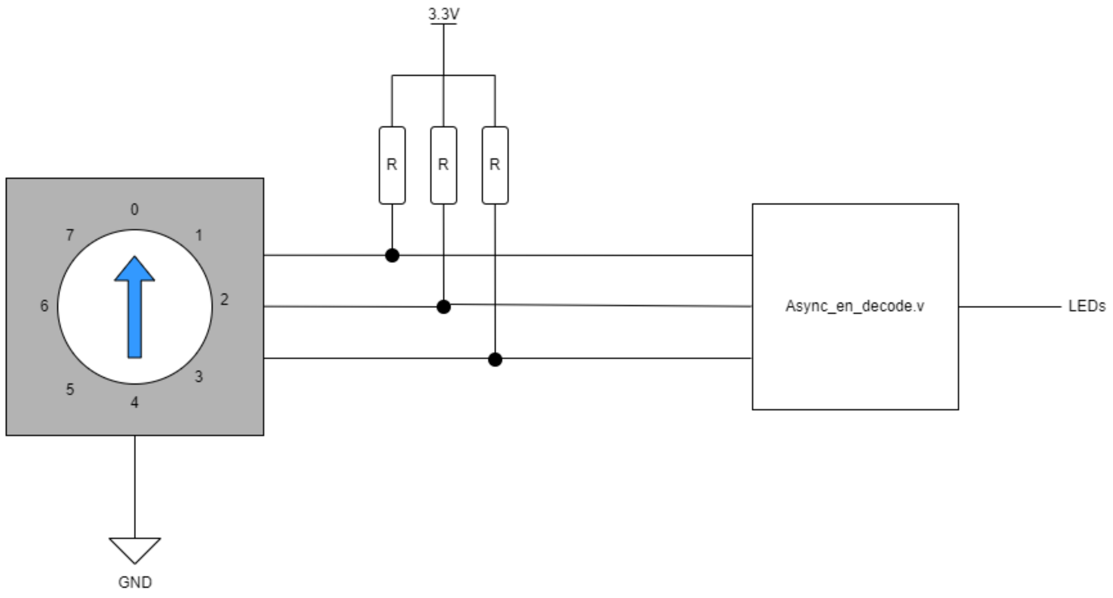
The above-mentioned component mapping is done in the constraints file. This file is located in `Sources -> Constraints -> constrs_1 -> PYMQ-Z1_C.xdc`. You can open and explore how the physical pins on the FPGA are mapped to various external components. These files are different for every development board, based on the on-board components.

Step 1: Binary Display

Now that you are a bit more familiar with the Vivado tool. Open the `async_en_decode.v` file. In this exercise, you will be mapping the binary input from the binary rotary switch to the on-board LEDs.

Follow the following steps to complete the assignment:

1. Make an asynchronous `always@(*)` block in the module
2. Implement the reset switch logic:
   - If the `reset` signal is on, your code should turn off all the LEDs
   - Only if `reset` is off, the rest of the code should be executed
3. Map the `bin_rot` input to the `led[2:0]` output. Refer to the circuit diagram below.
   - Note: The rotary switch is connected with pull-up logic

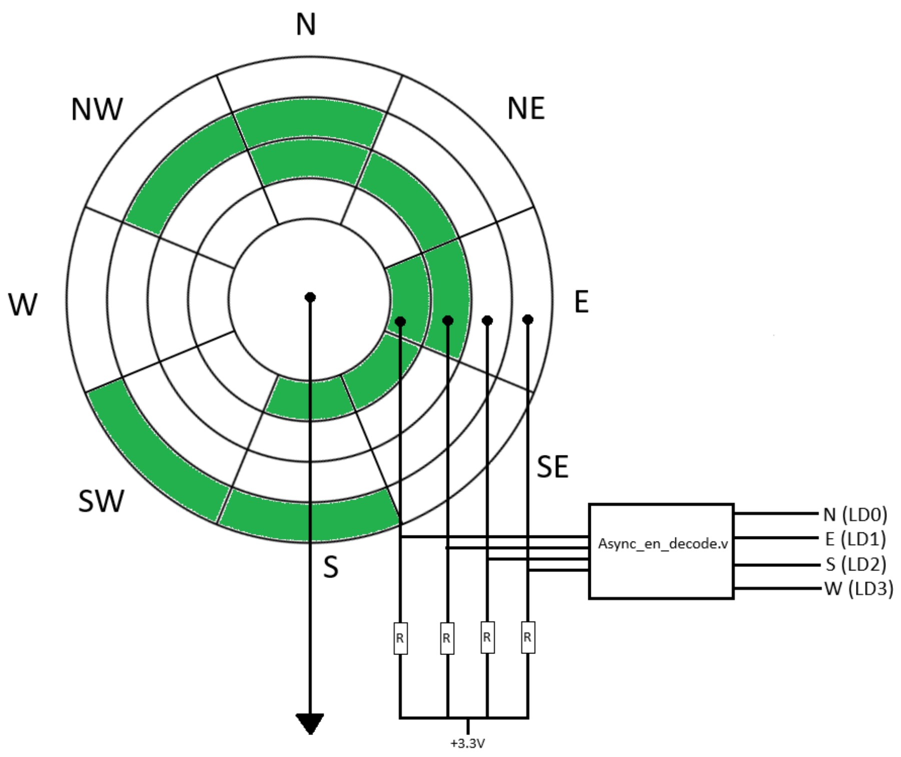


4. Make a testbench for your project and simulate.
   - You can find the steps to making your testbench in Simulation & Synthesis.
   - Once you have the testbench working as expected you can move on to **Step 2**

The data sheet for the binary rotary switch being used: [Binary Rotary Switch (S-8010)]

Step 2: Cardinal Direction Decoder

In this exercise, you will interface with the gray rotary switch and convert the inputs from the gray rotary switch into Cardinal directions.



As you can see from the above image, you have to translate the rotary switch output into cardinal directions. For example, if the wheel is in the `North` position, only `LD0` should be on and if the wheel is in the `South-West` direction, then both LD2 and LD3 should be on.

Follow the following steps to complete the assignment:

1. Since you are going to use the same project for 2 programs, you need to use the `prog_select` switch
   - When the switch is UP the binary display from *Step 1* should be executed
   - When the switch is DOWN the Cardinal Direction Decoder should be executed
2. Implement cardinal direction decoder logic in `async_en_decode.v`
3. Modify the testbench to test cardinal direction decoder logic
4. Once successfully tested, generate bitstream, and program the FPGA to test binary display
5. Call a TA to test your final design and test your cardinal direction decoder on hardware

The data-sheet for the gray rotary switch being used: [Grey Rotary Switch]

Part 2: Debouncing of electrical signals

In this lab, you will work with the on-board buttons and make a binary and gray-encoded counter.

Learning Goals of the Lab

1. Interface with buttons on the board
2. Use Debouncing and understand its need
3. Implement synchronous logic
4. Programming and testing the FPGA

Step 1: Binary Counter

In this step, you should implement a synchronous binary counter, and display the value using the on-board LEDs. When `BTN0` is pressed, the counter should increment, and `BTN1` is pressed the counter should decrement.

Note: Do not forget to implement the reset logic, similar to **Part 1**

1. Open the `Debouncer` project in Vivado
2. Implement synchronous logic to increment and decrement the counter and display it on the LEDs
   - DO NOT modify the block diagram yet
3. Write a testbench for the module and simulate it using the simulation tool in Vivado
4. Generate bitstream and upload it to the FPGA

Did the counter work as expected? Read more about what is debouncing and why do we need it: Debouncing

Step 2: Adding Debouncing

Here, we introduce debouncing logic in our project. This should remove all issues with your program getting multiple button presses.

1. Open the Block-Diagram
2. Disconnect the *encoder* A & B inputs from the buttons
3. Connect the inputs to the `btn_out` of the debouncer block
4. Simulate to verify that you have made the connections correctly
5. Generate the bitstream and program the FPGA

Step 3: Converting to Gray Encoding

In this step, you will convert your binary value into a gray encoded value and display it on the LEDs

1. Add an additional asynchronous block to convert the binary value into a gray encoded value
   - Use an internal register to store your binary value
   - You can use a look-up table for the conversion or use the gray encoding equations for 4-bit numbers

2. Simulate to verify that you have made the encoding correctly
3. Generate the bitstream and program the FPGA

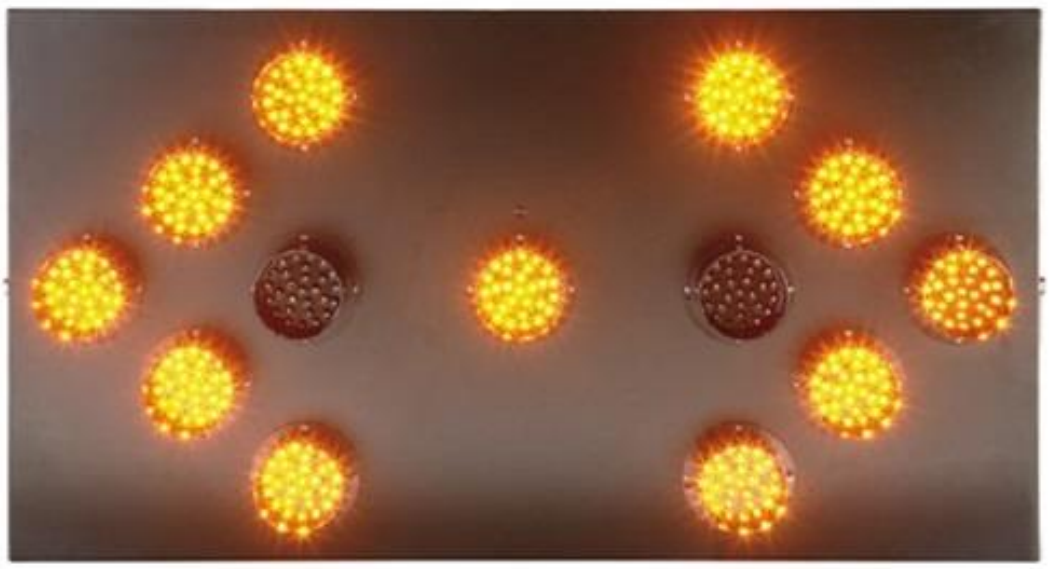
   Note: You can refer to Gray Encoding.

Once completed, please call the TAs who will approve your work and you can continue with the rest of the lab.

Part 3: Finite State Machine (FSM)

In this part, you will be implementing an FSM for a Road Safety Sign. The particular sign that you need to implement has the following modes:

1. Arrow Point Left
2. Arrow Point Right
3. Blinking Lights Warning Mode
4. Safe Mode



Learning Goals of the Lab

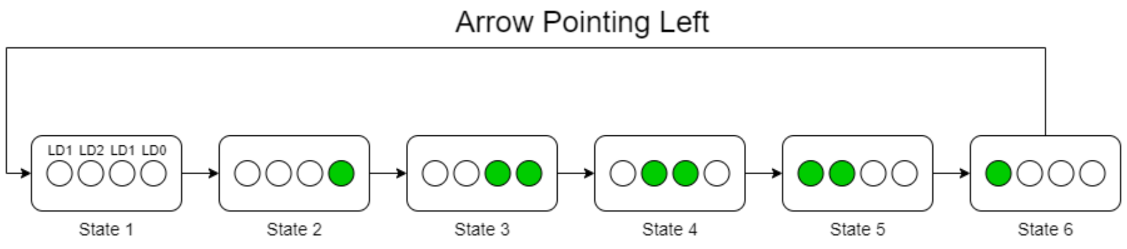
1. Understanding and Implementing a Finite state machine
2. Understanding different components and types of FSMs

To complete this exercise, use the `Road_Sign` provided in the template folder. Open it using Vivado. You will need to modify `Road_Sign\Road_Sign.srcs\sources_1\new\road_sign.v` to implement the encoder.

Step 1: Circular FSM

First, you will implement only the `Arrow Pointing Left` Mode. In this mode, the LEDs light up in a sequence moving towards the left. This is used to let the people know to divert to the left side of the road.

Since we have only 4 LEDs on the board, a pattern of running left LEDs is used to point left. For this, you will implement a 6-state circular FSM that has the following states:

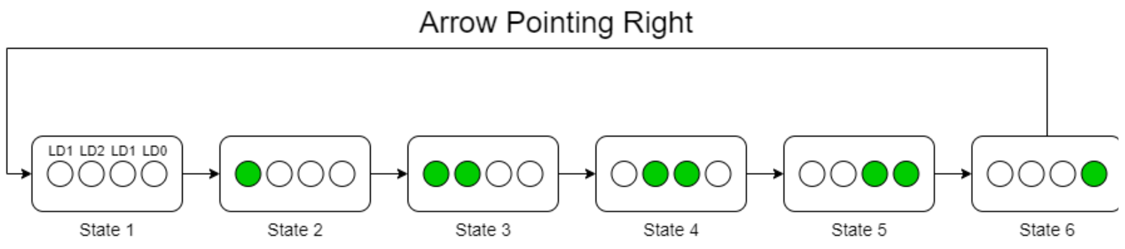


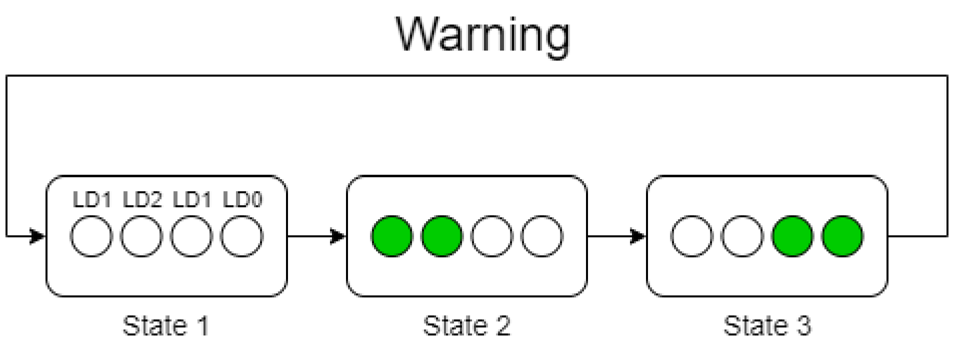
For the FSM to switch between the states, we have a time constraint. Every state transition should take place after `500ms`. To achieve this, you can use the provided clock which is of 125MHz, and divide it appropriately so that the state update happens when expected.

Step 2: High-Level FSM

Once you have Point Left working correctly, you need to implement a higher-level FSM to control which mode the Road Sign is in.

To complete this step follow the following steps: 1. Make an independent 4-state FSM - The 4 states are Point-Left; Point-Right; Warning; Idle 2. The FSM state change is triggered by the 4 on-board buttons. When the corresponding button is pressed the FSM should enter that mode - Point-Left: BTN0 - Point-Right: BTN1 - Warning: BTN2 - Idle: BTN3 3. Before implementing the functionality of these states, verify that the high-level FSM is working as expected - Use the RGB_LED1 to verify your functionality (Map 1 color to each state) 4. Duplicate the circular FSM used before and use it for Arrow Point-Right and Warning. As shown in the below figures



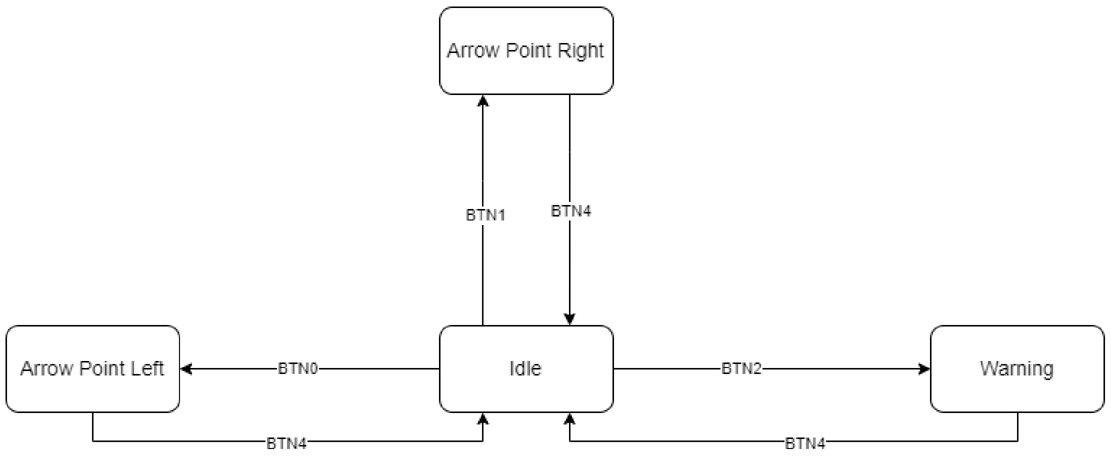


5. In the Idle state ensure that all the LEDs are turned OFF.

Step 3: Introducing Safe State of Execution

In the High-Level FSM implemented in the last step, any state can enter any other state. Which is a very unsafe way to transition between states. Since the road sign is deployed in a hazardous outdoor environment like a road-work site, it is important to have an additional safety measure. This can prevent accidental changes in the state of the sign, which can be hazardous on an active highway for example.

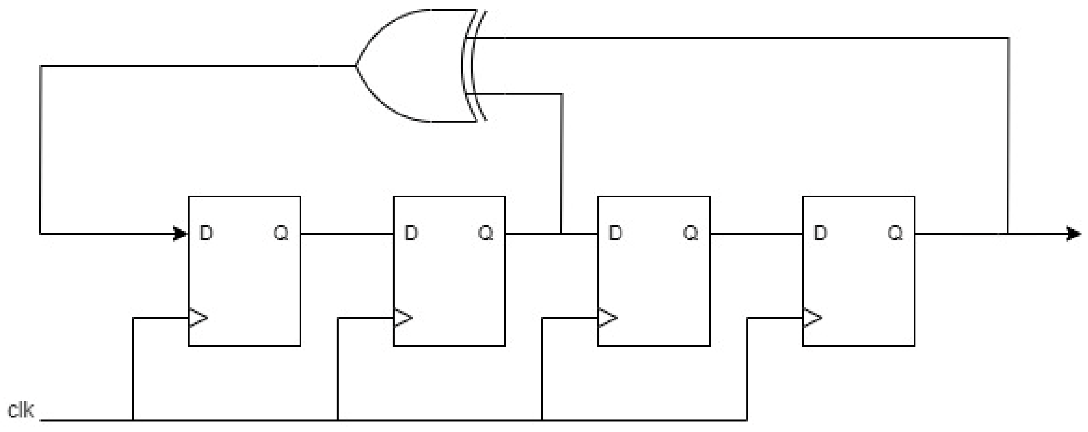
Thus, you need to add a constraint to the high-level FSM. You need to ensure, that if the FSM is in Point-Left, Point-Right, or Warning State it can only go to Idle state. And when in the Idle state it can enter any other state. A basic FSM diagram is presented below:



BONUS: Linear Feedback Shift Register Random Number Generator

What are LFSRs:

Linear Feedback Shift Register(LFSR) is a commonly used method to generate a set of pseudo-random numbers since it theoretically will be able to cover all the possible inputs for any n-bit circuits. A schematic of a typical LFSR is demonstrated in the figure below. A characteristic polynomial demonstrates the representation of the structure of the LFSR.



In this example, the characteristic polynomial is \(P(x)=x^{4+x}2+1\), which indicates that the first bit will be the XOR result of the 4th and 2nd bit. Repetition distance stands for the number of cycles that the two same generated pseudo-random numbers appear twice. To get the cycle counter, you will implement an internal counter to record the cycle. Seed it as the initial value of the internal register after you press the “reset” button.

Lab Procedure

Implement a 6-bit LFSR with the polynomial of \(P(x)=x^{6+x}5+x^3+1\), with three seeds, different seeds.

Seeds Stored
6’b00_0000
6’b01_1001
6’b10_1001

Follow the steps below to complete this lab: 1. Open Vivado and make a new project (no template is provided for this part) - Instructions on making a new Vivado project can be found in Vivado Project Setup. 2. Implement a LFSR logic with the polynomial \(P(x)=x^{6+x}5+x^3+1\) in the project 3. Add logic to check when the LFSR loops back and starts repeating the sequence - We recommend having a repetition signal as an output, it can go high when the LFSR value is the same as the seed value 4. Write a testbench to test the LFSR Implementation 5. Simulate the LFSR using the 3 seeds mentioned above - Record the repetition distances for all three seeds

Once you have finished the simulation, please call a TA for implementation verification.

2. Appendix

2.1. Setup

Please ensure that you have Lab Setup completed **before** the lab

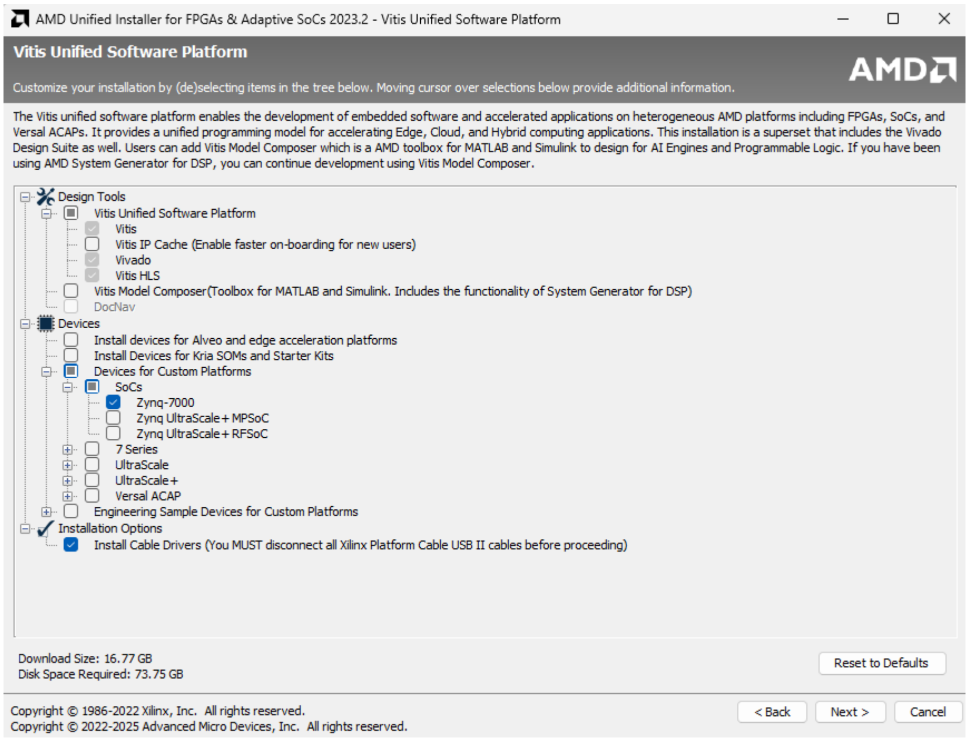
Vivado Installation

To run the labs it is recommended to install Vivado and Vitis 2023.2, but if you absolutely cannot install them, it will be possible to run everything in the lab desktops. The entire installation requires between 50 to 60 GB of final disk space (a bit more during install ~70 GB).

Installers are available for Linux and Windows: link. The coming lab was tested in version 2023.2. After you click on download, you need to create an account or log in if you already have one. On Linux do not forget to allow running chmod a+x FPGAs_AdaptiveSoCs_Unified_2023.2_1013_2256_Lin64.bin and ./FPGAs_AdaptiveSoCs_Unified_2023.2_1013_2256_Lin64.bin

Once you run the installer:

1. Select **Vitis**
2. Select the minimal installation components as shown in the image below (include SoCs Zynq-7000 family). Windows: leave the last option (Cable drivers) checked.



AMD Unified Installer for FPGAs & Adaptive SoCs 2023.2 - Vitis Unified Software Platform

Vitis Unified Software Platform

Customize your installation by (de)selecting items in the tree below. Moving cursor over selections below provide additional information.

The Vitis unified software platform enables the development of embedded software and accelerated applications on heterogeneous AMD platforms including FPGAs, SoCs, and Versal ACAPs. It provides a unified programming model for accelerating Edge, Cloud, and Hybrid computing applications. This installation is a superset that includes the Vivado Design Suite as well. Users can add Vitis Model Composer which is a AMD toolbox for MATLAB and Simulink to design for AI Engines and Programmable Logic. If you have been using AMD System Generator for DSP, you can continue development using Vitis Model Composer.

- Design Tools
  - ■ Vitis Unified Software Platform
    - ✓ Vitis
    - ☐ Vitis IP Cache (Enable faster on-boarding for new users)
    - ✓ Vivado
    - ✓ Vitis HLS
  - ☐ Vitis Model Composer(Toolbox for MATLAB and Simulink. Includes the functionality of System Generator for DSP)
  - ☐ DocNav
- Devices
  - ☐ Install devices for Alveo and edge acceleration platforms
  - ☐ Install Devices for Kria SOMs and Starter Kits
  - ■ Devices for Custom Platforms
    - ■ SoCs
      - ☑ Zynq-7000
      - ☐ Zynq UltraScale+ MPSoC
      - ☐ Zynq UltraScale+ RFSoC
    - ☐ 7 Series
    - ☐ UltraScale
    - ☐ UltraScale+
    - ☐ Versal ACAP
  - ☐ Engineering Sample Devices for Custom Platforms
- Installation Options
  - ☑ Install Cable Drivers (You MUST disconnect all Xilinx Platform Cable USB II cables before proceeding)

Download Size: 16.77 GB
Disk Space Required: 73.75 GB

Copyright © 1986-2022 Xilinx, Inc. All rights reserved.
Copyright © 2022-2025 Advanced Micro Devices, Inc. All rights reserved.

3. Linux: install cable drivers by running:

```
cd ${vivado_install_dir}/data/xicom/cable_drivers/lin64/install_script/

install_drivers/

sudo ./install_drivers
```

Read more about Cable Drivers: https://docs.amd.com/r/en-US/ug973-vivado-release-notes-install-license/Install-Cable-Drivers.

After the installation is complete, ensure that you have Vivado 2023.2 and Vitis Classic 2023.2 installed.

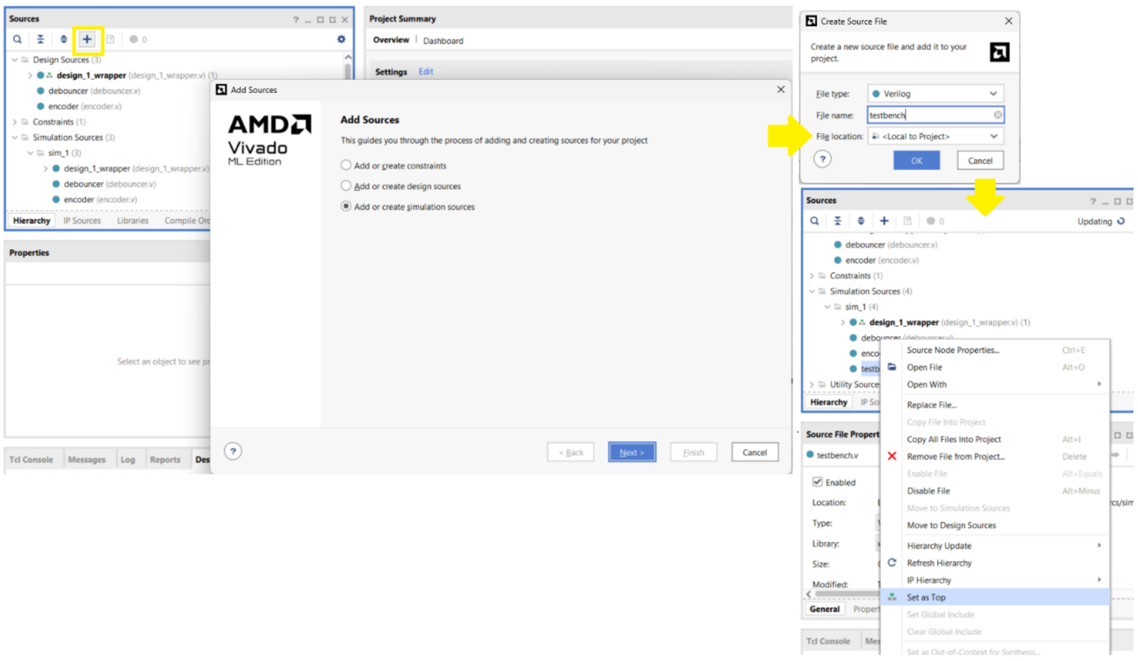
If you face any issues with the installations please reach out to the TAs. You will not be able to start working on Lab 1 without the software completely installed. So you must install it before the lab so you don't lose time during the lab.

2.2. Simulation and Synthesis

Simulation

Simulation is an important step when working on any FPGA project. It lets you test your code with custom inputs and access all the variables within your code. Additionally, for bigger projects, simulation has a much faster turn-around time rather and synthesizing your design and running it on the FPGA.

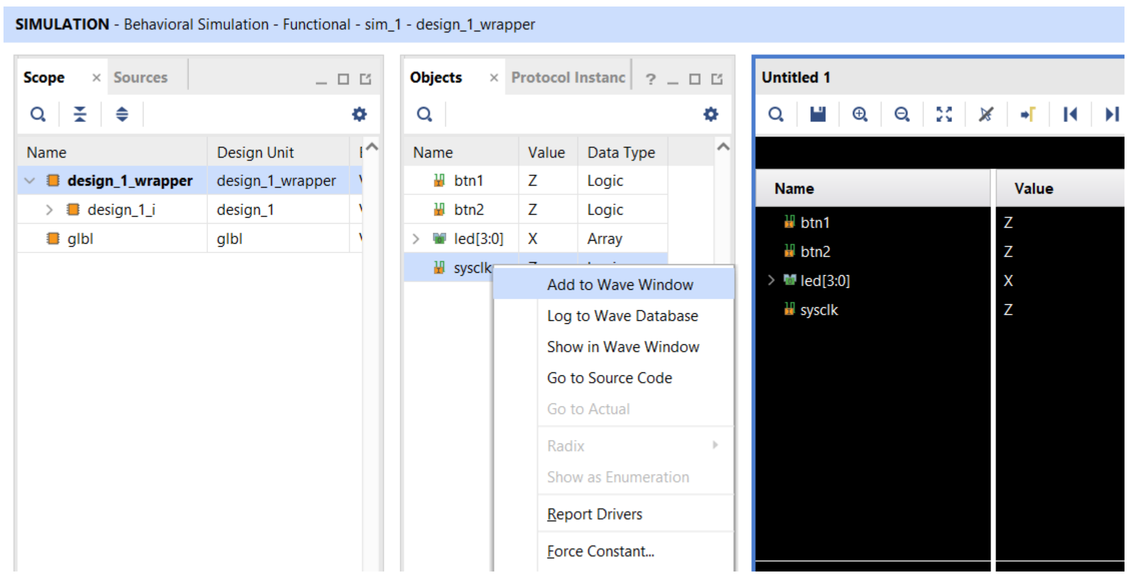
To simulate your project: 1. Create a new testbench file - With the project Open, select `Add Sources` or press `Alt + A`; select `Add or create simulation sources` - Select `Create File`, name the file `testbench`, then `Finish` - Expand `Simulation Sources` in the `Sources` window. - *Right-click* on your testbench file and select `Set as Top`. Your testbench file name should be written in **bold**

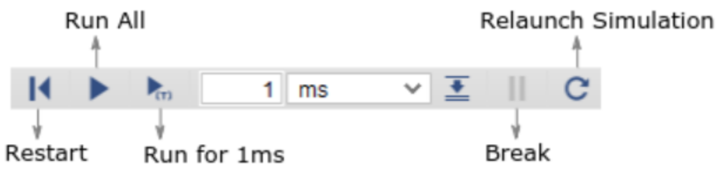


2. Write your test-bench
   - Create an Instance of the module you want to test.
   - Add code to modify the inputs to check various cases
   - If the module under test has synchronous logic, simulate a clock signal in your testbench

```
// General testbench example
`timescale 1ns / 1ps

module testbench;
    reg clk;    // Inputs as reg and outputs as wire
    reg reset;
    reg [3:0] input_signal;
    wire [3:0] output_signal;

    your_module uut (   // Instantiate the unit under test
        .clk(clk),
        .reset(reset),
        .input_signal(input_signal),
        .output_signal(output_signal)
    );

    initial begin   // Clock generation block
        clk = 0;
        forever #5 clk = ~clk;    // Invert clock signal every 5
time units
    end

    initial begin   // Stimulus block
        reset = 1;  // Initialize inputs
        input_signal = 0;

        #10;        // Wait to un-toggle reset
        reset = 0;

        #10 input_signal = 4'b0001; // Test inputs
        #10 input_signal = 4'b0010;
        #10 input_signal = 4'b0100;
        #10 input_signal = 4'b1000;

        #50;    // Wait and finish
        $finish;
    end
endmodule
```

3. Simulating your design
   - Launch the simulation by selecting, `Flow Navigator -> SIMULATION -> Run Simulation -> Run Behavioral Simulation`.
   - When the simulation you defined in your testbench is complete, open the waveform viewer to view your signals.
   - You can add more signals to the `Wave Window`. You can find more by expanding the `Sources` on the left side of the window. After you add a new signal you must reload the simulation to see the new signal changing.
   - Use the controls on the top taskbar to start/stop the simulation
   - Do not forget to resize your simulation window, so you can see the signals for the entire duration of the simulation



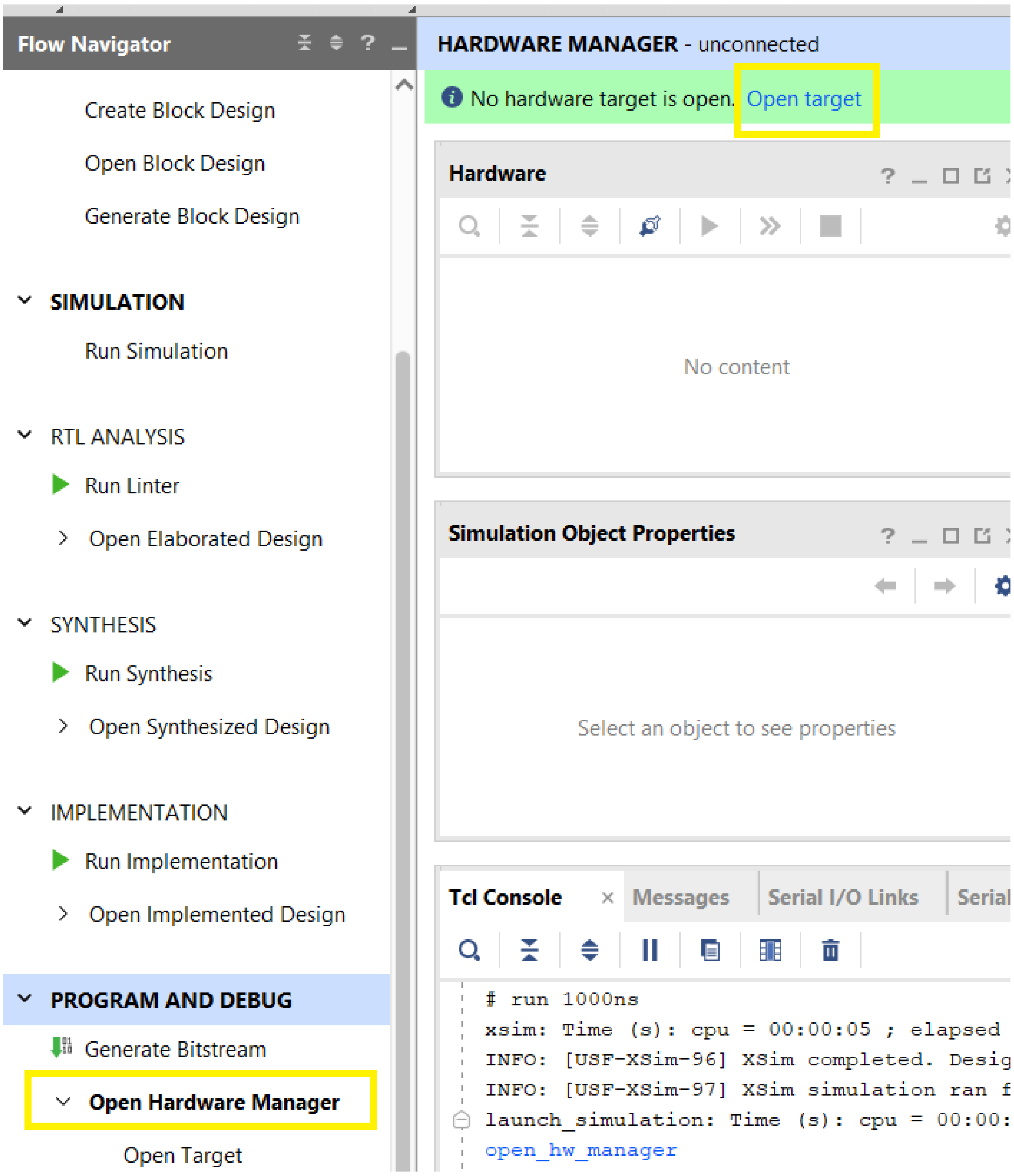


Running Project on FPGA

Once the project is working as expected in the simulation, you need to generate the bitstream to upload to the FPGA.

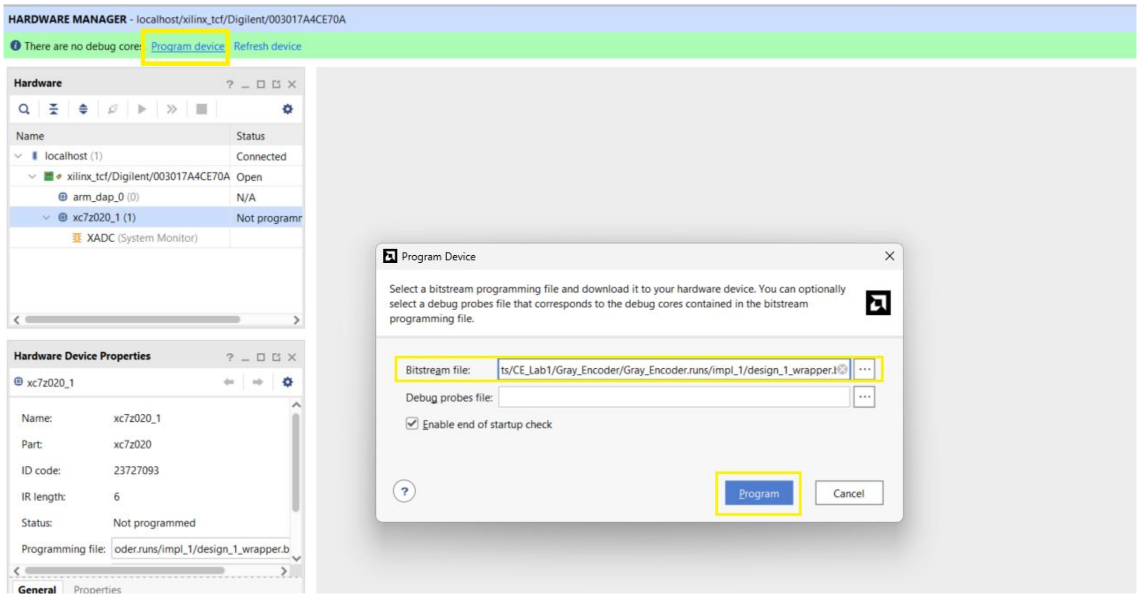
Run Program on the FPGA:

1. Generate Bitstream
    1. Select `Flow Navigator -> PROGRAM AND DEBUG -> Generate Bitstream`
    2. This will generate the bitstream for your project. This step may take up to 5 minutes to complete. So prefer the simulation to verify the correctness of your code.
2. Connecting to the FPGA
    1. Connect the FPGA to the Laptop using the USB Cable and turn ON the FPGA switch if it is OFF
    2. In Vivado select `Flow Navigator -> PROGRAM AND DEBUG -> Open Hardware Manager` - Select `Open target` at the top of the window, then select `Auto Connect`



```
# run 1000ns
xsim: Time (s): cpu = 00:00:05 ; elapsed
INFO: [USF-XSim-96] XSim completed. Desig
INFO: [USF-XSim-97] XSim simulation ran f
launch_simulation: Time (s): cpu = 00:00:
open_hw_manager
```

3. Program the FPGA
   - *Right-click* on the FPGA, select `Program Device`
   - Select the bitstream that was generated in the previous step. It can be found in `<project_name>\<project_name>.runs\impl_1`.
   - Program the FPGA


HARDWARE MANAGER - localhost/xilinx_tcf/Digilent/003017A4CE70A
There are no debug cores. Program device Refresh device
Hardware
Name
Status
localhost (1)
Connected
xilinx_tcf/Digilent/003017A4CE70A
Open
arm_dap_0 (0)
N/A
xc7z020_1 (1)
Not programr
XADC (System Monitor)
Hardware Device Properties
xc7z020_1
Program Device
Program
Cancel
General
Properties

Select a bitstream programming file and download it to your hardware device. You can optionally select a debug probes file that corresponds to the debug cores contained in the bitstream programming file.

Bitstream file: ts/CE_Lab1/Gray_Encoder/Gray_Encoder.runs/impl_1/design_1_wrapper.b

Debug probes file:

☑ Enable end of startup check

Name: xc7z020_1
Part: xc7z020
ID code: 23727093
IR length: 6
Status: Not programmed
Programming file: oder.runs/impl_1/design_1_wrapper.b

Your FPGA is now programmed and running the Verilog code that you wrote!

2.3. Vivado Project Setup

Creating a Project

To generate a new project in Vivado follow the steps:

1. Download the Board files (both Z1 and Z2) from here and paste them into `{installation_location}\Xilinx\Vivado\2023.2\data\xhub\boards\XilinxBoardStore\boards\Xilinx\`
2. Start Vivado, Select `Create Project`
3. Name your project and select `Next`
4. Select `RTL Project` and ensure, `Do not specify sources at this time` is selected
5. Select `Boards` and search for `PYNQ-Z1 (depending on your board Z1/Z2)`; Select the board from the list and click `Next`
6. Click `Finish` and your project is **completed**

Setting up a Project

First you need to add the constraints file for the selected board. For the PYNQ-Z1 board follow the following steps:

7. Add the constraints file for the board from the template repo, `PYNQ-Z1_Constraints.xdc`. Press the + button in the Sources Tab or Press ALT+A
8. Select `Add or create constraints`, add the file downloaded in the previous step, and select `Finish`
9. The constraint file has all the available GPIOs, LEDs, Buttons, and Switches on the board. You need to uncomment the ones you will need for the project.
10. PYNQ-Z1 also provides a 125MHz clock, enable it and use it in your design.

Second, you will have to either add or create a new source file. This can done similarly to the constraint file but by selecting, `Add or create sources`. A source file is an Verilog file, in which you write your code. Ensure that the inputs and outputs of the Verilog module are set correctly.

Third, you will have to create a new block diagram. Block Diagram is used to stitch different IPs and components together in your project. To create a block diagram select `Flow Navigator -> IP Integrator -> Create Block Diagram`. You can now add your Verilog source to this block diagram by right-clicking on the file and selecting `Add Module to Block Diagram`. If you want to connect any pins from your block to external GPIOs, you can right-click on them and select `Make External` or press `Cntr + T`.

> Ensure that the names of the resources in the constraint file and the block diagram are the same.

Fourth, and finally, once your block diagram is completed *Right-click* on it and select `Create HDL Wrapper...`. This will make an additional Verilog file which will act as the top layer of your project connecting all the IPs and sources in the block diagram together.

Now the project can be simulated or bitstream can be generated.