# CSE/CEN 598 Hardware Security and Trust
# Project 3: Physical Unclonable Functions
# Due Date: Oct. 18, 2023

## Part 1 - Arbiter PUF (40 Points)

In Part 1, you will design an Arbiter PUF and simulate it using an RTL simulation. The arbiter PUF uses delay differences between multiplexers (mux) to create unique responses for different challenges and devices. Figure 1 shows a simple arbiter PUF design. Challenge bits control the select signals on multiplexers. When a response is needed, the enable signal is raised from zero to one. The enable signal propagates through the muxes along a path controlled by the challenge signals. A register's data and clock input is connected to the outputs of the last muxes, creating a race condition. If the data signal arrives before the clock, the register will store a "1" value when the clock arrives. If the clock arrives before the data, the register will store a "0".

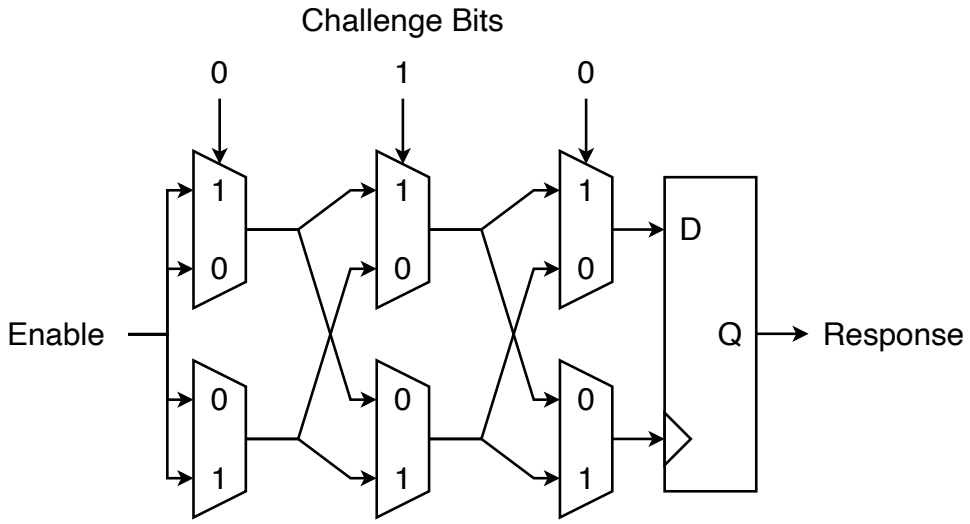


Figure 1: Arbiter PUF

To model a PUF in simulation, we have provided a Verilog module that implements a multiplexer with different propagation delays for different inputs (see project3˙part1/mux.v). The length of delays are controlled by a 4-bit `DELAY` parameter. Setting the `DELAY` parameter for each mux at the top level test bench allows you to model multiple pieces of hardware with different propagation delays.

### Required Tools - Modelsim

Modelsim is an RTL simulation tool. In this project, we use it to simulate our test benches. We have provided helper scripts to compile your verilog modules and simulate your test benches.

To simulate a test bench, execute the `run_test` script with the test bench module name as the only argument:

```
cd  ./project3/project3_part1/modelsim
./run_test tb_mux
```

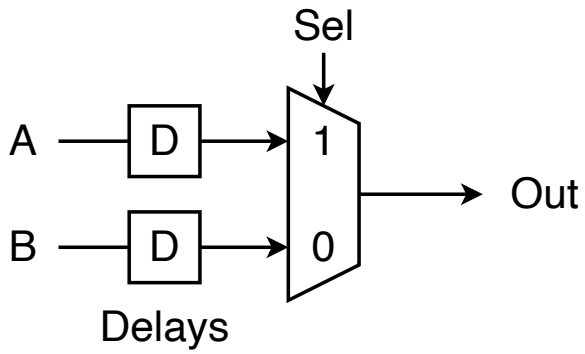


Figure 2: Multiplexer Module with Propagation Delays

The `tb_mux` and `tb_process_delay_model` test benches in the `project3_part1/tb/` directory will run out of the box, but you may add more test cases to them.

The Modelsim GUI is useful for observing signal waveforms. To open the Modelsim GUI, run the following:

```
cd  ./project3/project3_part1/modelsim
# Running the run_test script re-compiles the Verilog files
./run_test
# Open The Modelsim GUI
vsim &
```

If you are simulating the top level `tb_arbiter_puf` module, the following commands will add waveforms for the PUF inputs and outputs. **Note: you will have to finish the `tb_arbiter_puf` test before it runs.** In the Modelsim TCL command prompt, run:

```
vsim -voptargs=+acc work.tb_arbiter_puf
source wave.do
```

To simulate a different test bench, such as `tb_mux`, in the Modelsim TCL command prompt, run:

```
vsim -voptargs=+acc work.<your tb module name>
```

From there, you can use the Modelsim GUI to add signals to the waveform viewer.

## Task 1

Use the `mux` module provided in the src directory to build an arbiter PUF. Your PUF module must use the module template provided in `src/arbiter_puf.v` The PUF module must be parameterized and support an arbitrary number of challenge and response bits. The 4-bit delay values for each multiplexer must be set with the top level `DELAY` parameter. The `DELAY` parameter concatenates 4-bit delay values for each mux into a single N-bit parameter, where N is 4x the number of muxes in the PUF. You are encouraged to use additional modules, such as a module for a PUF with 1 response bit or a single round (2 multiplexors, 1 challenge bit) of the arbiter PUF design.

In your report:

- Include a block diagram of your PUF design and sub-module hierarchy.
- Describe the PUF and the design decisions you made.

## Task 2

Verification is typically 50-80% of a hardware development project. Every module must have a test bench. Write a test bench for every module. In your report, show the test bench output that gives you confidence that the module

is functioning properly. Note that the `tb_mux` and `tb_process_delay_model` test benches have already been written. You do not need to add more to these test benches, but you may to understand the module functionality better.

**Hint:** The `delay.py` in the `modelsim` directory will create a file with delay parameters that can be included in your `tb_arbiter_puf` test bench.

## Task 3

Write a test bench in `tb_harvest_crp.v` to generate all 16 challenge response pairs (CRPs) for a PUF with 4 challenge bits and 32 response bits. Re-run the test bench with multiple process delay configurations. Generate CRPs for at least 16 PUF configurations. Include a table of the results in your report.

## Task 4

Compute the Single-Chip Hamming Distance (HD) for each PUF used in Task 3 and report the results in a table. If you use a script/spreadsheet/etc. include it in your submission. What do the results say about the quality of your PUF? What is the ideal Single-Chip Hamming Distance for your Arbiter PUF design?

Next compute the Multi-Chip Hamming Distance for the 16 PUF designs tested in Part 3. What does the result say about the quality of your PUF? What is the ideal Multi-Chip Hamming Distance for your Arbiter PUF design?

If your PUF design performs poorly, what changes could be made to improve it? Is a 4-bit challenge with 32-bit responses enough for unique device ID or key generation? How many CRPs do you think are needed to uniquely id a device and why? Support your conclusions with evidence. Generate CRPs for different PUF configurations if necessary.

# Part 2 - Ring Oscillator (RO) PUF (60 Points + Bonus)

In Part 2, you will design a Ring Oscillator (RO) PUF, synthesize it for an FPGA, and harvest Challenge-Response Pairs. The RO PUF uses multiple Ring Oscillators to increment counters. Variations in fabrication mean that each RO will have a unique frequency. When two counters use different RO outputs as a clock, one counter will finish before the other. Checking which counter finished first provides a source of randomness unique to each device. Multiple PUF challenges are supported by providing several ROs to choose from when clocking each counter. Multiple PUF response bits are generated with multiple pairs of counters racing to the max count value.

Figure 3 presents a simple block diagram of a RO PUF. **Directions for obtaining an FPGA to complete Part 2 will be discussed in class. We will also have an FPGA available during office hours for this part of the lab.**

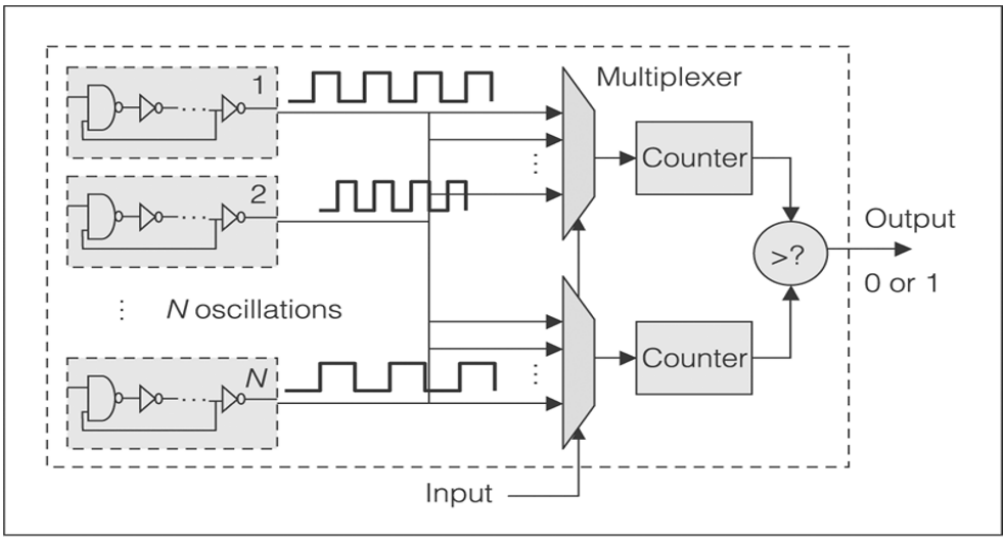


Figure 3: A Ring Oscillator PUF Block Diagram.

## Required Tools - Quartus Prime

The Quartus Prime tool synthesizes a Verilog design into an FPGA bit-stream. Project 3 includes a template project with the necessary materials to synthesize your Ring Oscillator and PUF designs.

To open the Quartus Prime tool, run the following in your VM terminal:

```
cd  ./project3/project3_part2/quartus/
quartus &
```

A template project targeting the Cyclone V FPGAs is provided. To open the template project, select Open Project, then select the Project3.qpf file. Once the project is open, use the Synthesize button to generate a bitstream from your PUF design. Note, you will need to fill in the Verilog template files before synthesis will complete successfully.

Figure 4: The Synthesize button in the Quartus Prime Toolbar.

To configure an FPGA with a given bitstream, click the Programmer button in Figure 5. Then use the ”Add File” button in the popup dialog box to add a “.sof” bitstream file. Use the “Hardware Setup...” button to make sure the correct hardware device is selected. Then, click the “Start” button to configure the FPGA. Figure 6 shows a successful FPGA configuration.

Figure 5: The Programmer button in the Quartus Prime Toolbar.

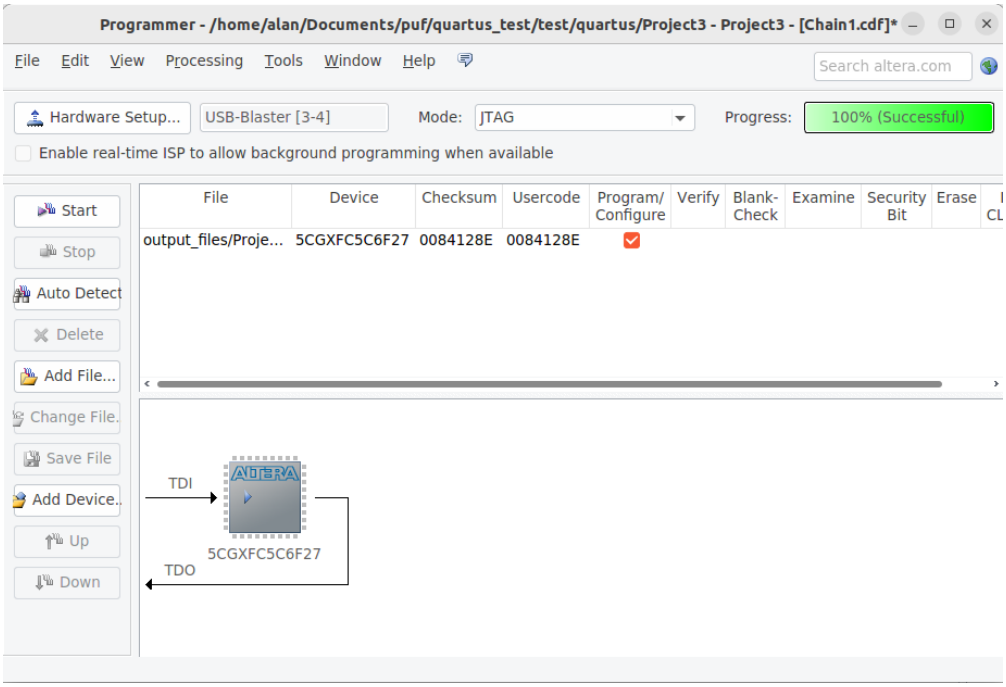


Figure 6: The Quartus Programmer window.

## Chip Planner

An FPGA is a collection of Look-up-Tables (LUT) and other configurable hardware such as memory or multiplier circuits. The Chip Planner shows us how each configurable element is used in a given design. Once you have opened your Quartus project, click the Chip Planner button shown in Figure 7 to open the Chip Planner tool.

Figure 7: The Chip Planner button in the Quartus Prime Toolbar.

When the tool opens, the default view shows the FPGA with each configurable element. The side toolbar includes view and zoom controls as well as buttons to see fan-in and fan-out connections between the different configurable components. The properties windows present detailed views and settings of each element. Figure 8 shows the default Chip Planner view. The dark blue squares in Figure 8 represent resources used by the current design. When you manually place your design, you should see the corresponding resources marked in the Chip planner.

The Cyclone V FPGA used in this lab groups LUTs with additional logic, including registers, multiplexers, and configurable adders into blocks called Adaptive Logic Modules (ALM). ALMs are organized into larger Logic Array Block (LAB) Cells. Each LAB Cell includes 10 ALMs. Figure 9 zooms in on the FPGA resources in Figure 8 to show a single LAB Cell and its ALMs. We will use these LAB Cells to create our Ring Oscillators.

LAB Cells and other FPGA resources are organized in a 2D grid of X and Y coordinates. Selecting a LAB cell in the chip planner will show its coordinates in the properties window. A third dimension “N” is used to select individual

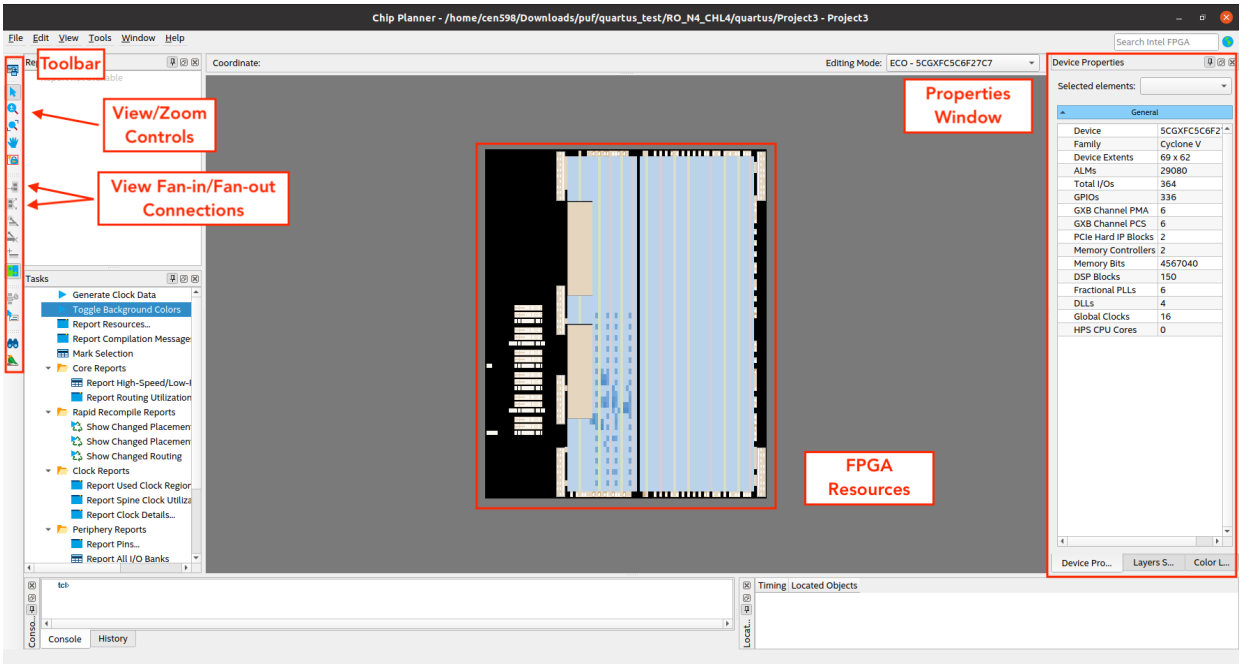


Figure 8: Quartus Prime Chip Planner Tool.

ALMs within a LAB cell. You will need the X/Y/N coordinates to constrain placement of your RO circuit to a specific LAB cell. Note that the N dimension for ALMs increases by a multiple of six for the ALMs due to other configurable resources available for selection. **To minimize the impact of routing on the RO circuit, you must constrain it to fit within a single LAB Cell.**

## Location Assignments

Configuring FPGA logic with a PUF requires careful placement and routing of the PUF design to ensure propagation delays are not dominated by different paths through the FPGA fabric. The Chip Planner can be used to see how the synthesis tool has placed a circuit on the FPGA resources, but to control the placement assignment constraints must be used.

Placement is controlled with the "Location Assignment" constraint. Assignments can be created in the Quartus Assignment Editor GUI or with TCL commands in a script. This lab will use TCL scripts due to the large number of assignments required.

For example, to place the NOT gate output `not_out0` from the Verilog in Figure 10 to the first ALM in the LAB at (11,2), use the following TCL command assignment (also provided in `example.tcl`):

```
set_location_assignment LABCELL_X11_Y2_N0 \
-to "not_wrapper:not_wrapper_inst|not_out0"
```

To run the TCL script:

- In the Quartus menu bar, select Tools → TCL Scripts...
- Select your script in the popup window and click Run.
- Check that your assignments were created by opening the assignments editor. In the Quartus menu bar, select Assignments → Assignment Editor.

The assignment editor in Figure 11 shows the valid assignment constraint was added to the project.

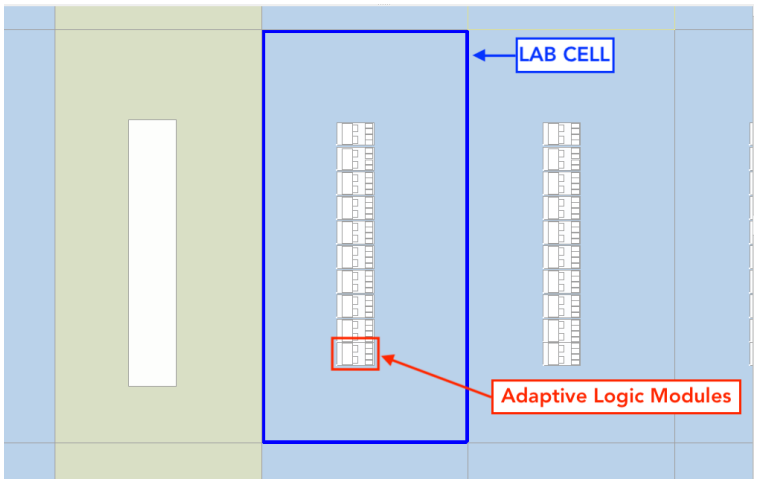


Figure 9: A LAB Cell with 10 ALMs in the Chip Planner tool.

```
1 module cyclone_v_top (
2   input SW,
3   output LED
4 );
5   not_wrapper not_wrapper_inst(LED, SW);
6 endmodule
7
8 module not_wrapper(
9  input in,
10  output out
11 );
12   wire not_out0;
13   (* keep *) not not0 (not_out0, SW);
14   assign out = not_out0;
15 endmodule
```

Figure 10: Example Verilog for the not_out0 location assignment.

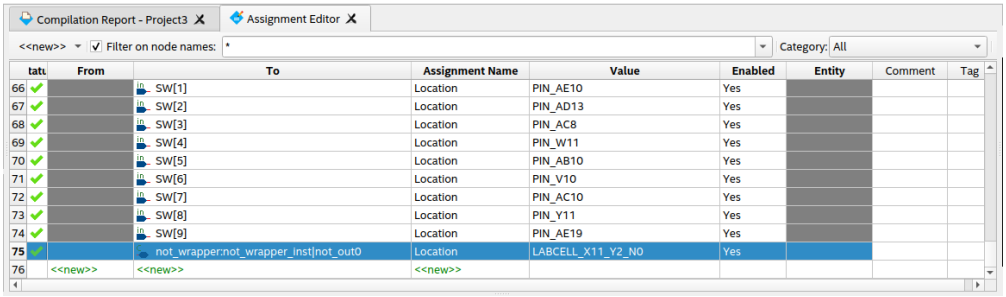


Figure 11: The Assignment Editor window with the added location assignment.

## FPGA Setup

Before configuring an FPGA from the VM, you must setup USB passthrough in Virtual Box to make sure the FPGA USB-Blaster configuration interface is visible to the VM. On Linux, run the following command to enable USB passthrough if USB devices are not already visible to your VM:

```
sudo usermod -aG vboxusers $USER
```

```
sudo reboot
```

You must reboot your host machine for the changes to take effect. After a reboot, you should see USB devices available for passthrough to the VM. Once you have the FPGA connected to your host machine, make sure it is selected as a USB device for passthrough to the VM. Figure 12 shows the USB passthorugh menu with the FPGA selected for passthorugh.

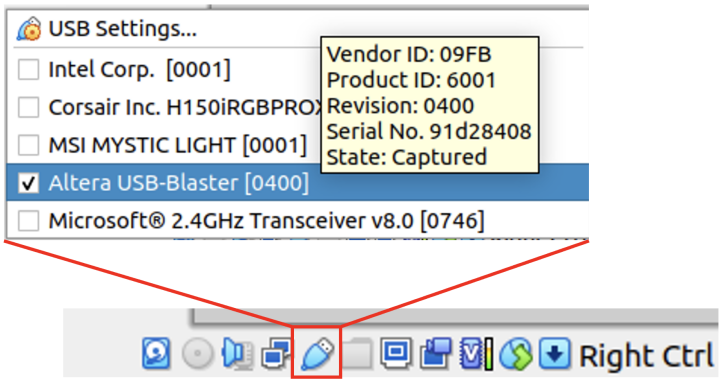


Figure 12: The FPGA selected for USB passthrough in the VM Toolbar.

Now the FPGA is visible as a USB device in the VM, but the Quartus Prime tool does not have permission to access it. To enable FPGA configuration by the Quartus Prime tool, run the following commands to install udev rules for the USB device:

```
cd ./project3/setup
sudo cp ./51-usbblaster.rules /etc/udev/rules.d/
sudo chown root:root /etc/udev/rules.d/51-usbblaster.rules
sudo reboot
```

After rebooting the VM, test that everything is working by running the following command

```
quartus_pgm -l
```

You should see an output similar to Figure 13.

## Task 1

Use the `RO_basic` Verilog module to create a ring oscillator with an Enable input and an oscillator output (`RO_out`). Constrain the placement of the RO to use a single LAB Cell X/Y Coordinate.

**Some helpful Hints:**

- You may need to use the "keep" synthesis attribute to synthesize the RO correctly.
- The Ring Oscilator creates a combinational loop which prevents a correct RTL simulation. To test your RO you may need to create a modified version that can be simulated in a test bench.

Include in your report the following:

- A circuit schematic of the RO module.
- The assignments you used to constrain the module placement
- A screenshot of the synthesized RO placement in the Chip Planner tool. Include the Fan-out of each RO signal in your screenshot.

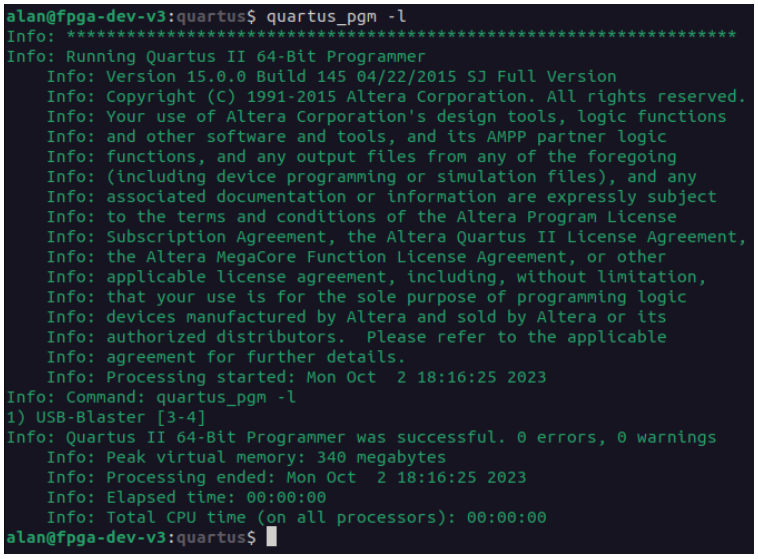


Figure 13: Quartus Programmer Command Line Output.

- Explain the placement in the Chip Planner tool. Is all of the RO Logic present? How does the Verilog map to the different ALMs in your placement?
- Explain how you tested your RO. Include the results that give you confidence the module functions correctly.
- Explain what the keep synthesis attribute is useful for. Did you use it in your design?

## Task 2

Fill in the contents of the `Counter_group` module with two counters that count up based on the frequency of RO outputs selected by the challenge bits. The table below describes the Counter_group module inputs and outputs.

| Name | Direction | Bits | Description |
|---|---|---|---|
| **reset** | Input | 1 | Reset the counters to zero. |
| **RO_out** | Input | 16 | Output from your Ring Oscillator Modules |
| **Cha0** | Input | 4 | Challenge Bits for the first counter. The Challenge should select a RO output signal from the RO_out signal. |
| **Cha1** | Input | 4 | Challenge Bits for the second counter. The Challenge should select a RO output signal from the RO_out signal. |
| **Response** | Output | 1 | A signal indicating which counter completed first. If done correctly, this will be a random signal that is dependent on the frequency of the RO_out signals selected. |

Write a test bench to verify that your counter modules function correctly. Include in your report the test bench output that gives you confidence the module is working. How many bits does your counter use? Why did you select this value?

## Task 3

Use the PUF template to combine your RO modules and counter modules into a single module. Your PUF must have 4 challenge bits for each counter group (8 bits total) and 4 response bits.

You will have to create a large number of location assignment constraints to correctly place each RO. Use a script to generate assignments for your design. Include the script and its assignment outputs in your submission.

**Hint:** The `example.tcl` file includes a sample assignment.

Include a block diagram of your RO PUF showing each module and input/output signal in your report. Explain your design decisions.

### Task 4

The `cyconle_v_top` module has been provided to make the buttons, switches, and LEDs of the Cyclone V GX FPGA easily accessible. Instantiate your PUF module in the `cyclone_v_top` module with the following connections:

1. Reset - Use KEY 0 as the reset button. The device should enter reset when the button is pushed.
2. Cha0 - Use Switches 0 - 3 for the first 4-bit challenge signal.
3. Cha1 - Use Switches 4 - 7 for the second 4-bit challenge signal.
4. Response - Use LEDG 0 - 3 for the PUF Response output.

**Hint**: This module should contain very little logic. It is mainly to connect the FPGA specific inputs/outputs with the PUF inputs/outputs.

Include in your report the following items and answers:

- A summary of synthesis results. How many ALMs does your design use? Are any other FPGA resources used?
- A screenshots of the Chip Planner showing the placement of the RO modules. Annotate the picture and describe the placement.
- Explain why you chose the given placement. Include at least one screenshot with enough detail to see each NAND and NOT gate in a single RO. Show the Fan-in and Fan-out of each ALM in the screenshot.
- Approximately how long does the PUF take to generate a response? Use this information to estimate the frequency of the Ring Oscillator.

### Task 5

Configure an FPGA with your PUF design and harvest at least 16 CRPs. Compute the Single-Chip Hamming Distance for your PUF. Explain the performance of your PUF. Is the Hamming Distance close to the ideal value? Does the PUF produce repeatable results?

**Bonus (20 Points):** Configure another FPGA with your PUF design and harvest the same CRPs as before. You may share your bitstream with someone else and generate CRPs for each other. Use the new CRPs to compute the Multi-Chip Hamming Distance for your PUF design. Explain how your PUF performs for the two devices.

## Deliverables and Submission Requirements

Write a report answering the questions for each task in Part 1 and Part 2. Double check that you have addressed each question in the task. Save your report in the CSE598_Project3_Handout directory as a PDF. Zip your CSE598_Project3_Handout with the file name **your_asurite_id**_project3.zip, for example aehret1_project3.zip. Be sure to return any hardware you have borrowed. Your grade for this project will not be posted until your hardware has been returned.