ECE 4156/6156 Hardware-Oriented Security and Trust

Spring 2025

Assoc. Prof. Vincent John Mooney III
Georgia Institute of Technology

Lab 3, 100 pts.

Due Tuesday, March 25 prior to 11:55pm

(please turn in homework electronically on Canvas)

In this lab, you will generate VHDL code for and synthesize several versions of a hardware-oriented (hardware-efficient) cryptographic primitive onto the Intel DE10-Standard architecture. For each synthesis run, you will document the FPGA utilization and maximum clock frequency of the design, generating a series of plots as deliverables.

This lab is based on the following paper: https://cic.iacr.org/p/1/4/19

You are not expected to read the entire paper to be able to complete the assignment. However, you may find it helpful to refer to the paper to better understand some of the terminology used in this assignment, such as feedback registers, Mersenne Product Registers, Composite Mersenne Product Registers, and update polynomials.

I. Setup

Download the lab3.zip file and place in a new directory. Unzip the archive.

Clone the GitHub repository https://github.com/gt-hwswcosec/cmprs2025. Note that for this lab, we will only use the ProductRegisters directory from this repository.

Ensure that you are on a system that has **Python3.12** (or newer) and **pip** installed.

1. From unzipping lab3.zip, you should see two directories labeled “vhdl” and “PythonCode.”
2. In PythonCode, there is a Python script called “vhdl_generator.py.”
    - The “ProductRegisters” folder you downloaded from GitHub is a Python library, and “vhdl_generator.py” uses this Python library
    - The Python library depends on the following Python packages: `memoization, numba, numpy, galois, python-sat`
    - First, install the above dependencies using pip (`pip install memoization, numba, numpy, galois, python-sat`)

- Once the dependencies are installed, move the ProductRegisters folder to inside the PythonCode folder

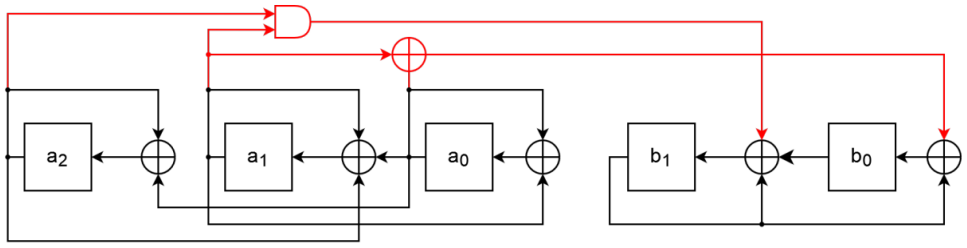
II. Running vhdl_generator.py

1. The file vhdl_generator.py uses the ProductRegisters Python library to automatically generate VHDL descriptions for a feedback register structure that can generate nonlinear, pseudorandom binary sequences. These sequences are very useful in cryptographic applications; they can be used as a starting point for constructing encryption schemes, hash functions, and MACs.

   The information below will help familiarize you with what exactly you are being asked to do in this lab:

   - This feedback register structure is termed a **Composite Mersenne Product Register** (CMPR)
   - A CMPR is constructed out of interconnected **Mersenne Product Registers** (MPRs)

     A gate-level diagram of a 5-bit CMPR constructed from a 3-bit MPR connected to a 2-bit MPR is shown below:



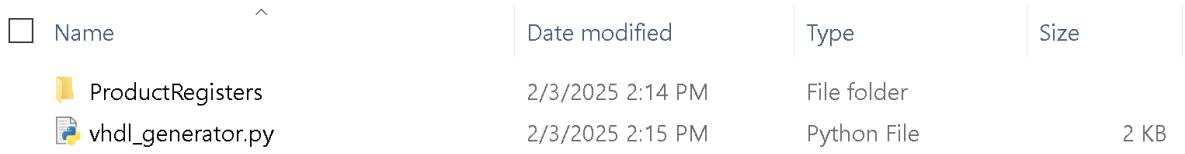
     Note that the portions of the diagram highlighted in red represent the logic used to connect the two MPRs together. This is called a chaining function and is discussed in Section 3 of the linked paper from the introduction.

   - MPRs are registers whose size (number of bits) is a **Mersenne exponent**, or **a prime number n such that $2^n - 1$ is also a prime number**
   - In this assignment, the CMPR size is 170 bits, and is constructed from MPRs of sizes 127 bits, 19 bits, 17 bits, 5 bits, and 2 bits (observe that 127 + 19 + 17 + 5 + 2 = 170, observe that all of the numbers that sum to 170 are Mersenne exponents)
   - Each MPR has two tweakable parameters: **its primitive polynomial**, and its **update polynomial** (refer to the paper if you want to gain a better understanding of what purpose these polynomials serve/what mathematical

properties they have). Changing these parameters results in significant changes to the nonlinear binary sequences generated by the CMPR, because a CMPR (more-or-less) behaves like a **keyed pseudorandom function**

- For an n-bit MPR, its primitive polynomial P(x) is a degree n binary polynomial, and its update polynomial U(x) is a degree n – 1 binary polynomial. A binary polynomial is a polynomial whose coefficients are either 0 or 1
- **Theory:** an MPR performs the recursive mathematical **operation A[t+1] = U(x)A[t] mod P(x)**, where A[t] is the current state of the MPR and A[t+1] is the next state of the MPR
- Here is an example of how an MPR is instantiated in vhdl_generator.py, for the case of a 2-bit MPR: `M2 = MPR(2, poly("1 + x + x^2"), poly("x"))`
- In the code snippet shown above, the primitive polynomial is $1 + x + x^2$ and the update polynomial is x
- We have used this 170-bit CMPR to develop VHDL code for a cipher (encryption scheme)
- For this lab, you will be changing the values of the update polynomial, generating VHDL, replace the feedback register logic in the cipher with the generated VHDL and synthesizing the cipher

2. The first update polynomial we will use is "x." In this case, leave the code in vhdl_generator.py as-is, and run the Python script (`python vhdl_generator.py`). **Note that vhdl_generator.py and ProductRegisters must be co-located in the same directory as mentioned in the setup steps of this assignment:**



3. Once the script is finished running, you should see a VHDL file C170.vhd that has been generated in the same directory as vhdl_generator.py.
    - This VHDL file contains the hardware description for the 170-bit CMPR that was originally instantiated in the Python script
    - For now, set aside C170.vhd (store it somewhere on your computer) and continue with the next steps
    - When storing C170.vhd elsewhere on your computer, you may find it useful to change the filename to somehow indicate the update polynomial (for example, C170_x.vhd)
4. Edit vhdl_generator.py. **We will now vary the update polynomial for only the 127-bit MPR**, which corresponds to variable M127 in the Python code. Now, we want to use an update polynomial of x^2.
    - To change the update polynomial from x to x^2, replace `poly("x")` with `poly("x^2")` in the variable declaration for M127

5. After changing the update polynomial, re-run the Python script (which again generates a VHDL file), and set aside this VHDL file
6. Repeat the process described in steps 4 and 5 for update polynomials of x^3, x^4, x^5, x^{10}, and x^{15}
7. You should now have 7 different VHDL files, each corresponding to the 170-bit CMPR but with a different update polynomial

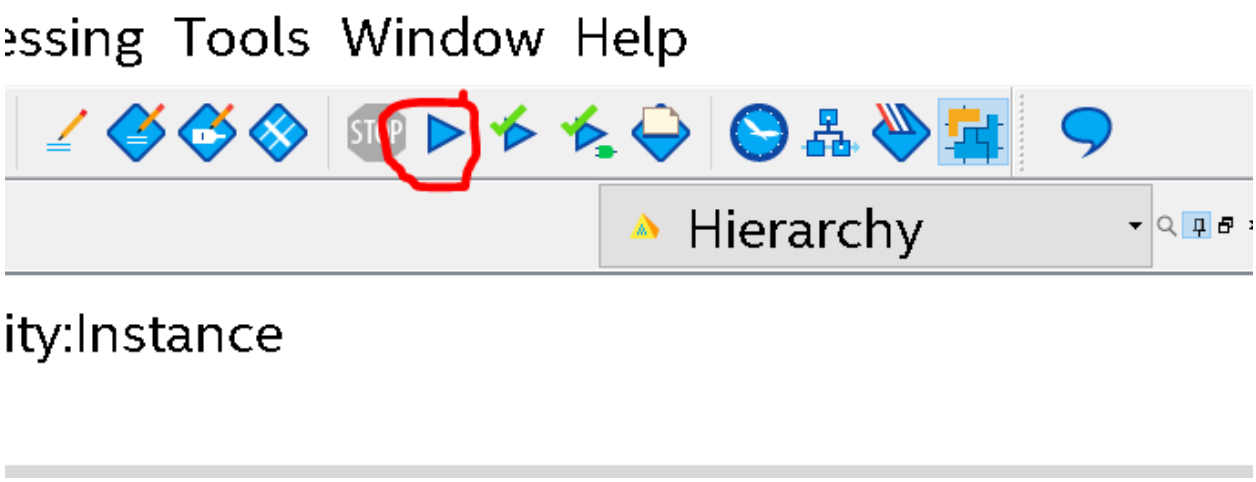
III. Synthesis using Quartus

1. Make a new project in Quartus (same as steps Lab 1, p. 25)
    - For this lab, the **top-level entity name** is "cmprcipherv3"
    - When the New Project Wizard asks if you would like to add a file to the Quartus project, add in the VHDL file in the "vhdl" directory of the lab assignment ZIP file. This VHDL file is called cmprcipherv3.vhd
    - When the New Project Wizard prompts you to specify family, device, and board settings, use the same settings as in Lab 1 (p. 29 of Lab 1)
    - When the New Project Wizard prompts you to specify EDA tool settings, use the same settings as in Lab 1 (p. 30 of Lab 1)
2. Now that you have created your Quartus project with the VHDL file included in the "vhdl" directory, you need to replace the feedback register logic in your project VHDL file with the feedback register logic you generated in the previous section
    - Open cmprcipherv3.vhd (the version of the file that is being used in your Quartus project)
    - The feedback register logic is described immediately after the begin statement for the architecture (lines 26-238 of the VHDL code), with variable names s and s_reg
    - Replace this feedback register logic with the logic from C170_x.vhd (the generated VHDL for update polynomial x). In the generated VHDL, the feedback register logic is described immediately after the "statereg" process (lines 29-241 of the generated VHDL)
    - Note that in cmprcipherv3.vhd, the variable names s and s_reg are used, but in the generated VHDL, the variable names nextstate and currstate are used. When replacing the feedback register logic in cmprcipherv3.vhd, you will need to change the variable names (nextstate => s, currstate => s_reg), which you should be able to do straightforwardly using any text editor with a "replace all" option

For your reference, here is where the feedback register logic should be inserted:

```
14  architecture encrypt of cmprcipherv3 is
15
16  type state_type is (setup, run);
17  signal keystream_bit : std_logic;
18  signal state : state_type;
19  signal s_reg : std_logic_vector(169 downto 0);
20  signal s: std_logic_vector(169 downto 0);
21  signal count : integer;
22
23  begin
24
25      -- CMPR LOGIC DESCRIPTION: Replace below based on the generated VHDL
26
27      -- FEEDBACK REGISTER LOGIC GOES HERE!!!
28
29      -- END CMPR LOGIC DESCRIPTION
30
31  process(rst, clk)
32  variable temp : std_logic_vector(84 downto 0);
33  begin
34  if (rst = '1') then
35      state <= setup;
36      count <= 0;
37      o_vld <= '0';
38      s_reg(169 downto 86) <= key(83 downto 0);
39      s_reg(85 downto 2) <= IV(83 downto 0);
40      s_reg(1 downto 0) <= "11";
```

3. Now that you have replaced the feedback register logic, you may compile the project (the compile button is circled in the screenshot below for convenience)



4. Wait for the project to finish compiling (~3-5 minutes). Once the project has compiled, **take screenshots of the Flow Summary and of Timing Analyzer -> Slow 1100Mv 85C Model -> Fmax summary. Note down the values for Logic Utilization (in ALMs) and Fmax (in MHz).** These are two important metrics for analyzing the FPGA utilization of a

design: ALMs are the basic building blocks of Intel FPGAs, and Fmax provides an estimate of the maximum speed at which the design can be clocked.

5. Repeat steps 2-4 for all of the generated VHDL files. That is, replace the feedback register logic in cmprcipherv3.vhd with the generated feedback register logic for all of the remaining update polynomials x^2, x^3, x^4, x^5, x^{10}, x^{15}, take the screenshots required in step 4 and note down the values for ALMs and Fmax.
    - **You do not need to make a new Quartus project each time, you can just edit the VHDL in the same project each time**

IV. Figure Generation

In this section, you will generate plots for the synthesis data you collected in the previous section.

You may use any plotting software of your choice (ex: Python, MATLAB). You will be generating two plots. For both plots, the x-axis should indicate the update polynomial used, in order of increasing degree (x, x^2, x^3, x^4, x^5, x^{10}, x^{15}).

For the first plot, the y-axis should indicate ALM count. For the second plot, the y-axis should indicate the value of F_{max} (in MHz).

It is recommended that the first plot be a bar graph and the second plot be a scatter plot to make the visualization easier.

What to Turn In

Please type all your answers **and include a cover sheet with your first and last name, GT ID number, and GT username. In the report, include:**

1. All screenshots from Section III.
2. All plots from Section IV.

Submit your Lab 3 report on Canvas as a PDF or .docx file.