ENIGMA on an FPGA

6.111 Final Report

bhanly, sabina22

December 11, 2021

Contents

1 Overview

For our project, we attempted to simulate an Enigma machine on the Nexys DDR 4.

2 Background

The Enigma machine was an encryption device used extensively by the Germans in the Second World War. It was considered to be very secure and thus was used to encrypt top-secret messages. The machines were regularly updated and the added complexity made them progressively more cryptographically secure. Here are some of the most important versions.

1. **Rotor Machine:** This initial version had 4 cipher wheels and a cog-wheel driven stepping mechanism. There were knobs for setting the initial position of the wheels and for switching between encoding and decoding.
2. **Enigma C:** This version replaced one of the cipher wheels with a reflector, aka UKW, which connected the output letters in pairs, thus making the machine symmetric. The reflector could be set in different positions (more options available in further models).
3. **Enigma I** This is the first model only available to the German Army. It evolved from the Reichwehr Enigma D, which had an extra plugboard (aka Steckerbrett) at the front. Some of the advanced versions included two extra rotors, increasing the security 10 times.
4. **Schlüsselgerät 39 (Feburary 1943)** This new machine coupled each rotor with a Hagelin pinwheel in order to allow for variable stepping of the rotors.
5. **Enigma Uhr (July 1944)** This version had a new Steckerbrett, which allowed for 40 positions, and removed the constraint that the plug connections be pair-wise.

3 Setup

The Nexys DDR 4 is connected to a keyboard via the USB slot, to a monitor via the VGA port, and to a second FPGA through the JA and JB ports as shown in Figure 1.

The keyboard is used to send an unencrypted message to the FPGA which then encrypts the message and displays it on both the monitor and the computer. This ciphertext is also sent to the other FPGA which then encrypts the message again and displays it. If the second FPGA has the same starting settings as the first one this second encryption decrypts the message back into readable plaintext.

Our inputs from the board and their uses are as follows:

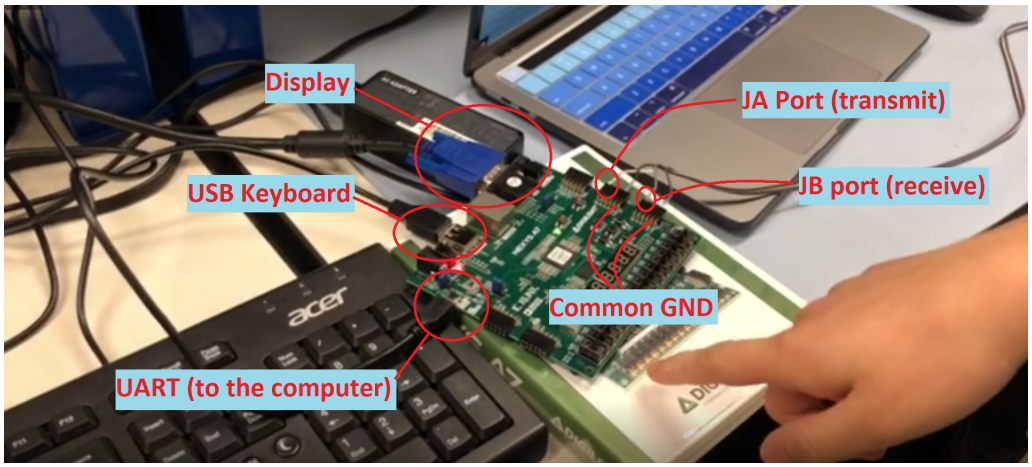


Figure 1: FPGA connections

1. **Center Button (btnc)** - resets everything
2. **Right Button (btnr)** - starts the settings changing process
3. **Upper Button (btnu)** - while changing settings, allows the user to enter the current value for the current feature and move to the next feature to be changed
4. **Down Button (btnd)** - resets just the enigma module
5. **sw[15]** - the leftmost switch establishes the mode of operation, off means the FPGA is transmitting and taking input from the keyboard, on means it's receiving and taking input from the other FPGA.
6. **sw[14]** - the next switch determines if we're using the steckerbrett or not
7. **sw[4:0]** - the rightmost switches determine the day (i.e. which settings we should use
8. **sw[9:5]** - these switches are used for changing settings

All the button inputs are debounced before use.

3.1 Changing Settings

The `top_level` module had a simple FSM to set new settings. If the user presses the btnr button then the FSM starts in the state ROTOR1, if the user then hits the btnu button the FPGA takes in the switches sw[7:5] as an input for what the rotor number of the first rotor should be the FSM then moves to the POS1 state. If the user then hits the button btnu again the FPGA takes in the switches sw[9:5] as an input for what position the first rotor should start in and

then the FSM moves to the ROTOR2 state. This acts just just like the ROTOR1 state except its for the second rotor, the FSM then cycles through the POS2, ROTOR3, POS3 states. After you hit btnu in the POS3 state and the FPGA takes in the starting position based on switches it moves to the INPUT_SET state where it inputs this new state into the enigma. It then switches to the DONE state and the enigma will now be encrypting with the given settings.

4 System Design

The system had four major subsystems: enigma, comms, display, and keyboard.

Figure 2 shows the overall structure of the code, while Figure 3 details the Enigma module structure.

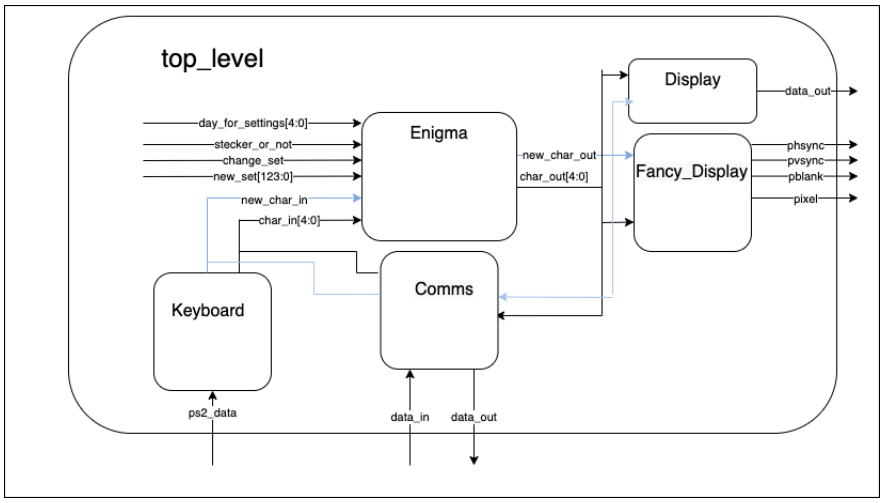


Figure 2: Overall Block Diagram

The main FSM, shown in Figure 4, illustrates the different states the machine is in, and how it moves between them. The leftmost switch decides where we take our input from; this can only be changed when we're not currently encrypting anything. If our input is from keyboard, whenever we have a new letter available, we encrypt it; once that's done, we send the it out through comms, and return back to wait for other letters. If we take our input from comms, when there's a new letter available, we decrypt it, then go back to wait for others.

5 Goals

We split this section into three parts, detailing our baseline goals, stretch goals, and extreme stretch goal.

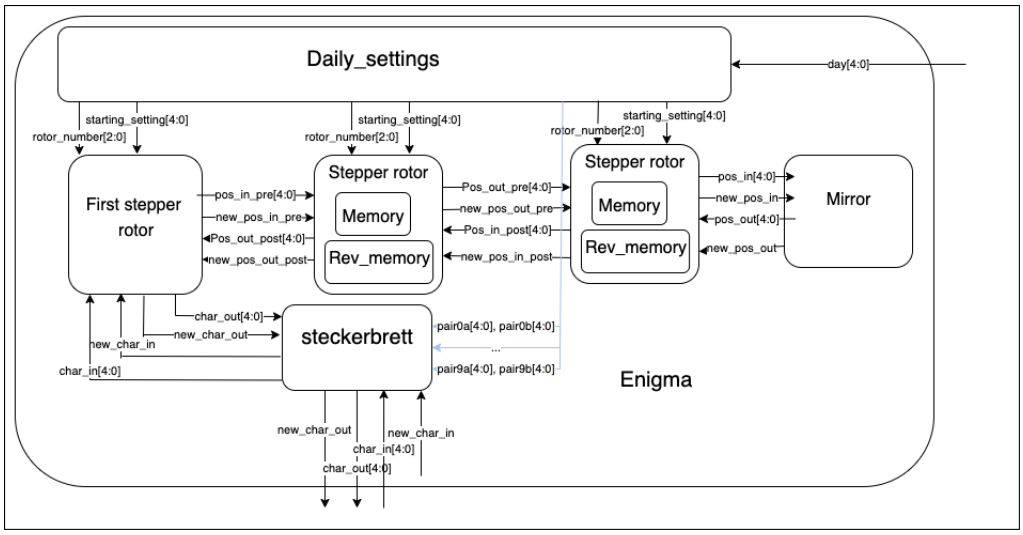


Figure 3: Enigma Module Block Diagram

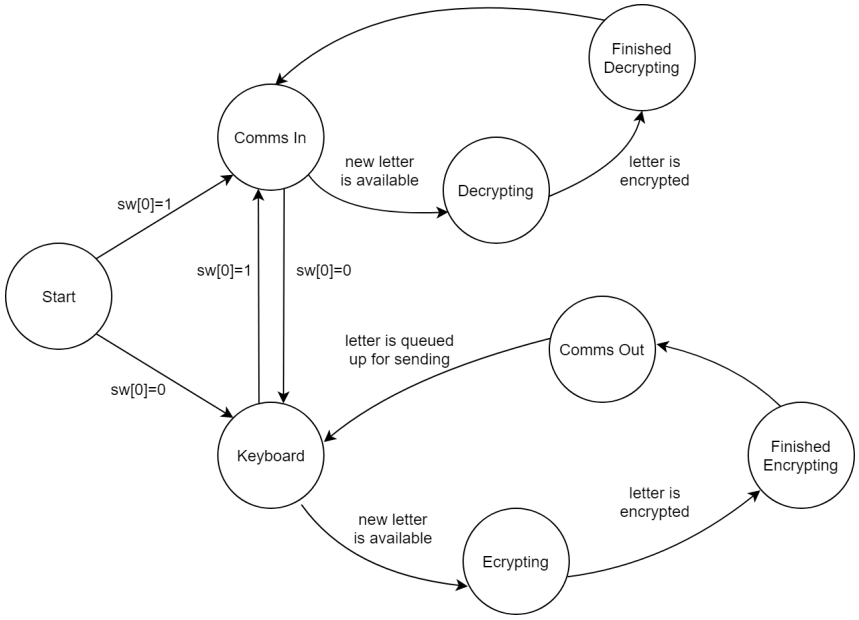


Figure 4: Enigma Finite State Machine

5.1 Baseline Goals

Our baseline goals were to have a fully functional basic system, specifically:

1. a functional enigma module, including 3 rotors and a mirror module
2. a simple display module, to see what the outcome of the encryption is
3. a communication module, so we can pass encrypted messages between two FPGAs
4. a way to take input from the keyboard
5. have all the above integrated into a single system

In addition, we wanted to have tested our system extensively, which we accomplished with a mix of testbenches and simulations for any module for which that would be feasible, as well as manual testing.

5.2 Stretch Goals

Our stretch goals which were fully implemented were to

1. create a fancier display through VGA
2. add a steckerbrett(plugboard) to the encryption process

Another stretch goal was to have four rotors and a variable stepping pattern, this was not done on the board but the code is modularized in such a way that lends itself to the easy addition of as many motors as desired and the ability to change the stepping pattern of each rotor.

5.3 Extreme Stretch Goal

Our extreme stretch goal was to crack Enigma (as in, decode messages knowing the internal structure of the machine, but not knowing any specific settings). For implementation details, check out the "Eve's System Modules" section.

6 Modules

6.1 enigma

Overall encryption module which takes in a character and passes it through a steckerbrett and multiple rotors and then reflects off a mirror and back through the rotors and steckerbrett in order to encrypt the character. It also sets the initial starting settings of the rotors and steckerbrett through a call to the daily_settings module.

In terms of the difficulty and process of making enigma. The rotor modules were the most difficult modules to make in enigma due to having to thoroughly understand and read up on the enigmas inner workings in order to get the math right. The memory, mirror, steckerbrett, and daily_settings modules were mostly a matter of figuring out how to connect them to the other modules and

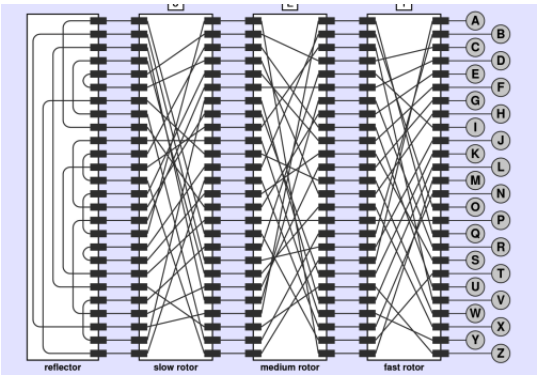


Figure 5: How the internal pieces are wired together on the Enigma

then a lot of time copying settings in and translating characters to the numbers used to represent them. At one point I even made a python script to convert the settings from the keysheet we used in to usable verilog code since there were so many.

6.1.1 rotors [Bianca]

In an actual enigma a signal passes through each rotor twice, once going towards the mirror and once after it passes through and heads back out. To model this, both sets of rotor modules have two sets of character inputs and outputs, one to be used for signals before and one for signals after the mirror. The rotor encrypts a character by having an internal wiring or mapping which sends one character to another. This is accomplished in the rotor modules by having it call sub-modules memory and rev_memory with memory being the mapping for when characters go through prior to the mirror and rev_memory being the reverse for when they pass back through the other way after the mirror. The rotor module will pass characters coming in into the respective module and take the mapping out in order to encrypt the character. Additionally, each rotor will rotate or step after a certain number of characters have passed through it. This number is determined by the rotors position in the enigma and the type of enigma being used. In order to simulate this rotation, there was a logic called rotation which keeps track of how rotated the rotor should be. This way when a character comes in, the module offsets the character by the amount the rotor is rotated to before sending it to memory and then offsetting it back when it is passed out again in order to keep the relative positioning real rotors would have.

The rotors also have different individual starting settings: the walzenlage, choice and order of wheels, and the ringstellung, starting position. The order part of the walzenlage is taken care of in the enigma module as the rotors are connected there in order. The choice of rotor is passed into the rotor module as rotor_number and used when calling memory so that it passes back the correct

internal wiring according to the chosen rotor. The ringstellung is taken care of through the starting_setting input. This way when the rotor is reset, the logic rotation is set to that input so the rotor begins in that rotated position.

first_rotor The first rotor on an enigma always steps every time so as soon as a new character comes in from before the mirror it steps the rotor and then encrypts the character.

stepper_rotor This module is very similar to first_rotor with the main difference being the stepping. Because rotors in positions other than the first had different stepping patterns this module has a parameter, steps, which denotes how many characters it should have pass through it before it steps/rotates. This makes it easy to place as many rotors as desired with variable stepping patterns into the enigma.

6.1.2 mirror [Bianca]

The mirror takes in a character and sends it back out as a different character based on a preset mapping.

6.1.3 steckerbrett [Bianca]

Also known as a plugboard. Takes in as an input 20 characters which denote the ten pairs of letters. When a character is then passed in to it, if it is a part of a pair, it is mapped to its pair and if it isnt it passes through unchanged.

6.1.4 memory [Sabina]

This is mainly used as storage - the internal registers are hard-coded with the internal wirings of all the 8 rotors. Whenever a request come in with a letter and a rotor number, it returns the corresponding output letter at the next clock cycle.

6.1.5 rev_memory [Sabina]

Similar to the `memory` module, it has the same mappings, just in the opposite direction.

6.1.6 daily_settings [Sabina]

This module is a simulation of the monthly sheets and stores the settings of the machine. It outputs which rotors are being used, in what order, what their letter offsets are in the starting position, and the steckerboard pairs.

Since we wanted the settings to be changeable, we decided to have the hard-coded settings at reset (instead of having them combinatorial like in the previous two modules). It allows settings for a certain day to be changed through the `read` and `new_set` inputs.

6.2 Displays [Bianca]

The FPGA displayed the encrypted characters through two displays, one on the monitor and one through the serial communicator to the computer.

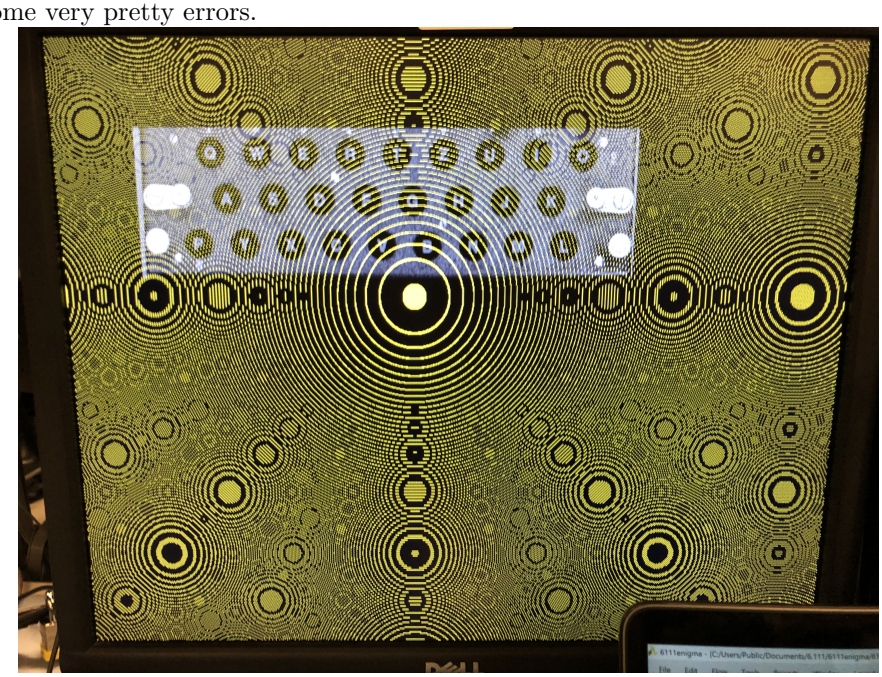
6.2.1 display

The display module used the serial reporter protocol from lab 2 to send the ASCII for the encrypted character to the laptop programming the FPGA. This was done by having an always_comb block to map the incoming character to its appropriate ascii value and then initiating the sending over serial when new_char_in the indicator for a new character coming in was high

6.2.2 fancy_display

The fancy display module uses VGA to display the glowlamp part of the enigma on a monitor and light up the letter that the enigma module encrypted. This was done by modifying the animation code from lab 3. The glowlamp image was rendered using the picture_blob module with a greyscale color map. The lit indicator on the letter which was encrypted was done by modifying the blob module to be circular instead and having the position of it change based on a mapping of letters to pixel position on the screen.

The more tedious part of this code was figuring out how to create the circular blob since there ended up being a lot of timing issues so the module needed to be pipelined to fit timing constraints. However in the process of this there were some very pretty errors.



Additionally, in order to figure out the mapping of the lit indicators position for every letter I had to find the pixel position of the center of every letter on the glowlamp image and put that into the verilog which ended up being very tedious.

6.3 comms [Sabina]

This module allows two FPGAs to communicate with each other. While we wanted to connect the devices via ethernet, we decided against that as we realized that we only needed one data line. In the end, we settled on using the JA and JB ports for transmitting and receiving data, respectively.

These modules are also used to communicate with the computer through python scripts (both for sending display data, and for sending/receiving data in the breaking enigma section). The UART ports are used for that purpose.

It integrates a transmitting and a receiving module.

6.3.1 serial_tx

This module was taken form the Lab 3 submission. It takes in an 8-bit value to be transmitted, appends a 0 bit at the start and a 1 bit at the end, then sets data_out to the next bit to be transmitted with a frequency that matches the baud rate used by the python script.

6.3.2 serial_rx

`serial_rx` is the receiving module. It samples the data line at every clock cycle, and when it stops being high over a longer period of time (corresponding to nothing being sent), it starts reading the transmitted bits, ignoring the first and the last bits in the 10-bit transmission (always 0 and always 1 respectively).

6.4 keyboard [Sabina]

We used the code from this Github project to get the data from the keyboard to the FPGA. From there we had to parse keycodes. When you press a key, an 8-bit scancode (e.g. 1B) corresponding to that key is sent out; when you release that key, the scancode sent out is 16 bits, "F0" and the code of the released key (e.g. F01B). We used that knowledge to select the keycode from right before the key was released as the one of interest, and we used the diagram from here to map the keycodes to letters.

6.4.1 PS2Reciever

This is very close to the Keyboard Demo code, we added an output signal so we know when the data was parsed.

6.4.2 keyboard_debouncer

We changed the name of the module, as it was similar to the module that we used to debounce the buttons.

7 Cracking Enigma

In addition to the Enigma main project, our last stretch goal was cracking it. Decrypting original Enigma messages was done by knowing some words that were used a lot (e.g. weather report). This assumption was used to figure out the day's settings, after which you could figure out what any of the messages on that day.

To break our version of ENIGMA, we assumed that we have access to the same source files as the ENIGMA project except for the daily settings (i.e. we know the internal system structure). We also assumed that the first 12 characters sent after reset are always "SIXONEONEONE".

We decided to do this in two stages:

1. Run the algorithm for all the possible settings (done once, compiled in cribsheet.txt)
2. At each reset (e.g. every "day"), take the first 12 received letters and compare them with the cribsheet; then find the settings it corresponds to and use those settings for the rest of the incoming letters until the next reset.

Therefore, we have created 2 extra Vivado projects: `eve_bf` for bruteforcing all the possible solutions, and `eve` for using those pre-calculated values to decode incoming messages.

7.1 Choice of Approach

We realize this is not the most feasible way to decrypt, and obviously wouldn't be perfectly-implementable in real life. However, given that we want to have a fully automated process with no human input (as opposed to manually checking for real words, as was done to decode messages while Enigmas were actively used), and given the tight schedule that we were on for the stretch goals, we added these extra constraints to make it more manageable for now.

Since this part of the project was started when there was no steckerbrett, and because introducing the steckerbrett will multiply the number of possibilities by a factor bigger than 1000, leading to a 62+ GB file and 5000 hours cribsheet.txt generation, we're going to assume that encryption does not use a steckerbrett in this case.

7.2 Memory

Since the first part is a bruteforcing algorithm, the results were going to take up a lot of memory. For each possible setting that we want to account for in

the cribsheet, we need 3 bytes for storing the settings and 7.5 bytes for storing the encoded 12-letter preamble. Assuming a choice of 3 rotors out of 8, we have 336 possible rotor groups; each of those can have 26 starting positions, therefore we get $336 * 26^3$ (almost 6 million) possibilities. Therefore, the total amount of storage needed is 62MB.

The generation of the file for the roughly $100,000$ initial entries took less than 5 minutes, and we expect the generation of the complete file to take less than 5 hours. If this time would pose an issue (e.g. if we were to remove the no-steckerbrett constraint), the code can be optimized to run multiple instances of the enigma module in parallel (as currently we only have one).

7.3 Modules

We named our other projects `eve` (which is the name usually given to the eavesdropper in cryptography literature) and `eve_bf` (the bf comes from Brute Force).

7.3.1 eve_bf top_level

This is the main module of the bruteforcing algorithm. We go through all the possible settings combinations, and for each of them, we try it (by calling the `trying_set` module), then send the outcome to the computer through the the UART port.

7.3.2 eve_bf trying_set

The helper method for `top_level`, it takes the new settings as an input and uses them to encrypt "SIXONEONEONE".

7.3.3 eve_bf bruteforce.py

The script listens for data from the FPGA, then parses it in chunks of 11 bytes, and then adds the resulted values in a file (`cribsheet.txt`).

7.3.4 eve top_level

The `top_level` module for the eavesdropper is similar to the main project `top_level`, but it's missing the keyboard module.

7.3.5 eve daily_settings

One of our goals with this was to have as much as possible compatible with the main enigma project source files as possible. Therefore, this module (which is called by the enigma module both in Eve's project and the main one), had to have the same inputs and outputs, but fulfill the job of providing the daily settings that were infered by `find_ds`.

7.3.6 eve find_ds

This module is responsible for finding the right settings. It sends a request to the computer with the encoded preamble, and waits for it to find the corresponding settings. Then, once the computer is sending data back, it parses it and outputs the settings.

7.3.7 interface.py

This is the python script that interfaces with the FPGA, by receiving the request, looking it up, and sending back the settings.

7.4 Points of failure

There are *a lot* of bugs (observed and potential) with the current implementation. We'd like to discuss some of them here.

At some point between trying the first versions of the breaking enigma code and the time of this writing, some bugs have appeared in the enigma module that only manifest themselves in the `eve` project. So far, the hours of trying to find them have yielded no success. Therefore, we have been unable to further generate, or even recreate, `cribsheet.txt`.

Something else we found after this broke: while `cribsheet.txt` was being generated, the python script had a bug that, if the numerical value of a byte it received was smaller than 128, it would crop all the leading zeros when stringing it together with the other received bytes. Since the condition we implemented to check for the arrival of one full 10.5 byte settings result was based on the length of the string (88 bits), we assume that most of these data points we got were in fact a mixture of multiple data points we wanted. This also means that the current file is useless. While we think we fixed the bug after the fact, there's no way for us to tell if it worked in practice until we fix the first bug.

The file is also larger than expected. At the time of its generation, it only had 3 rotor orders to go through, so only $3 * 26^3 = 52,728$ settings. However, it quite obviously did not stop writing to the file. I had to interrupt it at some point, even though I was very curious to see where it would stop, but it looked like it just wouldn't. Combined with the knowledge of the previous bug, the file should have been shorter, which is even more concerning. It seems like the FPGA didn't stop sending data once it was supposed to be done going through all the options. There might be some kind of infinite loop in there, although not one we could spot at a first glance. And since, as mentioned before, the simulation is broken for other reasons, it's hard to debug that.

There are other potential pitfalls that we would need a somewhat working project to check for them.

We realized after the fact that txt files likely encode 0s and 1s as more than 1 bit each, therefore if we store it as a txt file, its size would be much larger (presumably 8 times bigger, ignoring the semicolons and spaces). Indeed, the current file with almost 100,000 entries is 7.87MB, as opposed to $10.5 * 93,873$

which amounts to slightly less than 1MB. A change we would like to make is to change it to a .pickle file, so it hopefully takes less storage, especially as we add in all the rotor order options.

Also, it is possible that among the 6 million options, the preamble has the same encoding with more than one option, so when we search for that, they collide (and e.g. it returns the wrong settings, therefore the rest of the message is still gibberish). We don't know if it will be a problem, and if it is, we haven't found a good solution of how to avoid it (except for, of course, making the preamble longer and thus decreasing the probability that any of them collide). That said, there are currently $25^12 = 59,604,644,775,390,625$ possible mappings for our 12-letter preamble, so the probability of collisions is quite low, even if we add using the plugboard as an option.

Once we get `eve_bf` to work, we'll be able to generate a good cribsheet file and test the other project, `eve`. Getting both of them to work would accomplish our last goal, so we would like to keep working on that.

8 Conclusion

All in all, this was a successful project, since we accomplished all of our baseline goals of having a working Enigma simulation, as well as all of our stretch goals except for cracking Enigma.

9 Code

You can find our public GitHub repository here.

10 References

Glowlamp picture used in fancy_display:
https://cs.carleton.edu/faculty/awb/cs111/w20/lab7_images/enigma-top-view.png

Technical details of the enigma and rotor and mirror wirings:
http://users.telenet.be/d.rijmenants/en/enigmatech.htm
https://www.cryptomuseum.com/crypto/enigma/working.htm

Message sheet used to have preset settings: http://users.telenet.be/d.rijmenants/en/enigmaproc.htm

Keyboard tutorial and GitHub:
https://digilent.com/reference/learn/programmable-logic/tutorials/nexys-4-ddr-keyboard-demo/start
https://github.com/Digilent/Nexys-4-DDR-Keyboard