

Enigma Machine Decoder

Noah Sullivan

March 12, 2021

Introduction

In this lab I will be designing and implementing a circuit to break the code for a variant of the Enigma Machine. Given a plain text message, the machine will substitute letters with other letters according to a specific algorithm or mapping, to produce the cipher text. The Enigma Machine used the same algorithm and key to both encode and decode a message. In this project I will be using the following modules...

- A state machine to detect the crib
- An overall state machine that controls the counters
- A module given that provides the characters to decode
- A module given that decodes the fed character based on a key input

Along with this I will be reusing the seven seg converter from the previous lab, as well as the logic for the counters, the ring counter, and the selector I used in previous labs. I will also be using a variety of counters, and while I will touch on them in the report they are all 20 bit counters, which was expanded upon from the 15 bit counter I have used in a previous lab, I added one more 5 bit counter to the party.

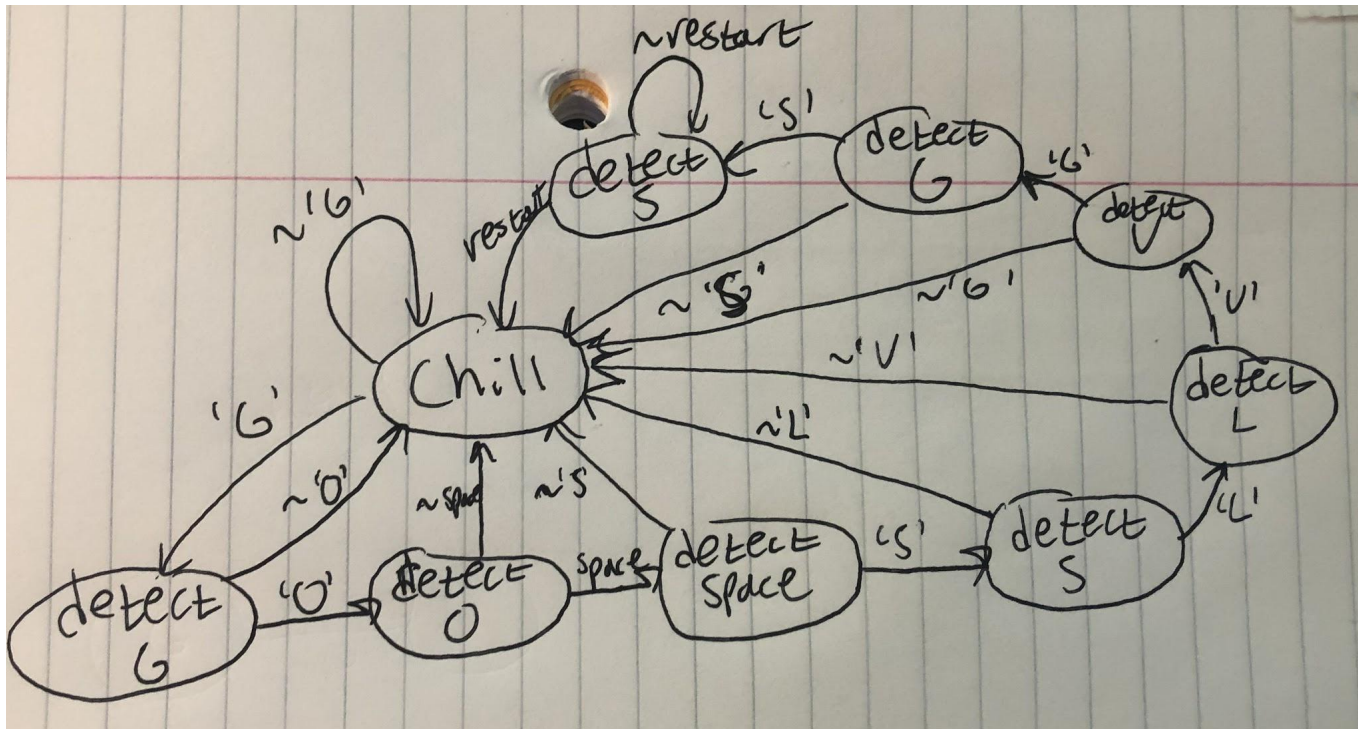
Part 1: Crib Detection

I began by designing a state machine that will be used to detect the crib message. The crib message for this lab was GO SLUGS so I needed to design a state machine that could tell when that exact string of characters was fed into it. Therefore I gave the state machine the following inputs and outputs

- input [4:0]char
- input clock
- input restart

- output crib_found

The input char holds the current character to be read, the clock is obviously the clock. Restart goes high after the entire message is processed with a single key, and needs to be done again. Crib_found will go high after the crib message has been detected. Shown below is the state machine I used to figure out how to write this crib detection module.



The output crib_found will go high in the second detect S state, meaning that the crib has indeed been found. However if at any point the in detecting the crib there is a letter that is not supposed to be there it will be returned to the starting state where it will await another G. I will also include the logic equations along with the comments specifying which state is which.

```
assign d[0] = restart | (s[1] & ~G & ~O) | (s[2] & ~O & ~space) | (s[3] & ~space & ~S) | (s[4] & ~S & ~L)
```

```
| (s[5] & ~L & ~U) | (s[6] & ~U & ~G) | (s[7] & ~G & ~S) | (s[0] & ~G);
```


```
// Detect "G"
```

```
assign d[1] = (s[0] & G) | (s[1] & G);
```

```
// Detect "GO"
```

```
assign d[2] = (s[1] & O) | (s[2] & O);
```

```
// Detect "GO "
```



```
assign d[3] = (s[2] & space) | (s[3] & space);  
  
// Detect "GO S"  
  
assign d[4] = (s[3] & S) | (s[4] & S);  
  
// Detect "GO SL"  
  
assign d[5] = (s[4] & L) | (s[5] & L);  
  
// Detect "GO SLU"  
  
assign d[6] = (s[5] & U) | (s[6] & U);  
  
// Detect "GO SLUG"  
  
assign d[7] = (s[6] & G) | (s[7] & G);  
  
// Detect "GO SLUGS"  
  
assign d[8] = ((s[7] & S) | (s[8] & ~restart)) & ~s[0];
```


Part 2: Counters

This lab will require three counters, all of them can be implemented using a 20 bit counter, however only two out of the three will really use the full 20 bits. I will explain how I use them in the top level section of the report, but to make them I simply turned a 15 bit counter into a 20 bit counter.

Part 3: State Machine

The process of searching for keys, and stopping when the right key is found is controlled via another state machine with the following inputs...

- crib

- 
- btnC
 - clock
 - restart
 - idle

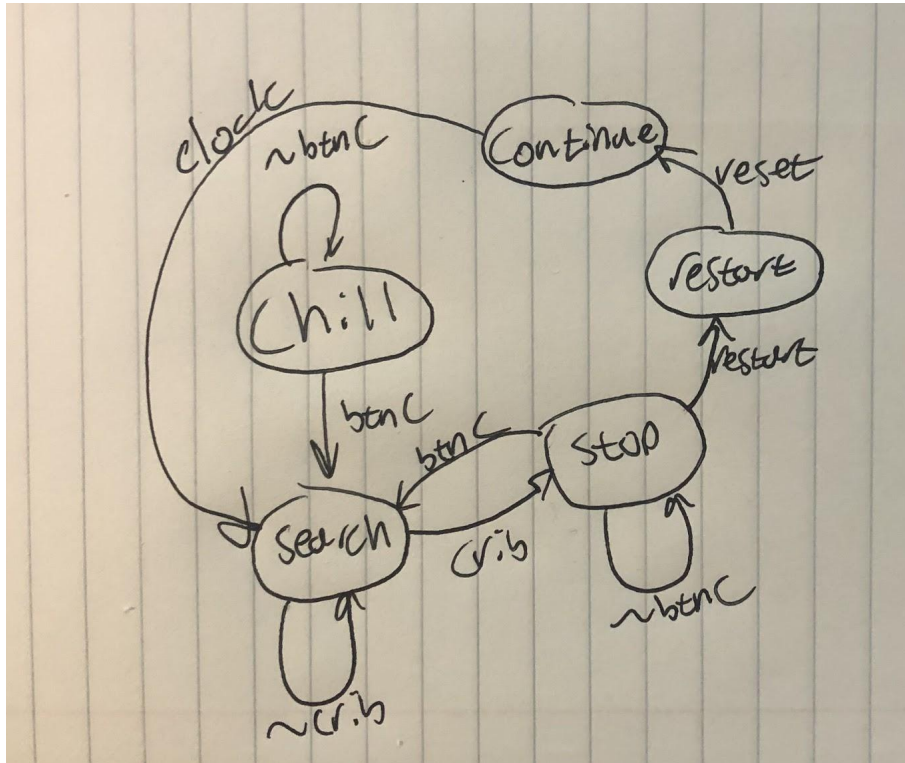
BtnC will be used to begin the search as well as continue it if a crib is found using the wrong key. The crib will go high when a crib message is detected. Restart is the same as in the other state machine, going high when the entire message has been processed. Idle is high when every key has been processed. These inputs will enjoy the following outputs...

- search
- Stop
- Chill
- reset
- continue
- done

Search will go high when the state machine is in the “search” state, this will allow for the counters to increment and continue processing the characters. Stop will go high when the crib is found, and will be what stops the counters from counting and allows for viewing of the current key.

Chill is high when it's in the starting state and, and won't be high after the search process begins. Reset goes high one clock edge after restart goes high. This is to help the counters load in the correct order, which I will explain later. Continue goes high a clock edge after reset and it's done

for the same reason. The state machine diagram for this is shown below.



I will also include the logic equations for the above states.

// begin state

```
assign d[0] = (s[0] & ~btnC);
```

// Searching For Crib

```
assign d[1] = (s[0] & btnC) | (s[1] & ~crib) | (s[2] & btnC) | (s[4]) | (s[2] & ~crib);
```

// Found Crib

```
assign d[2] = ((s[1] & crib) | (s[2] & ~btnC));
```

// chill state

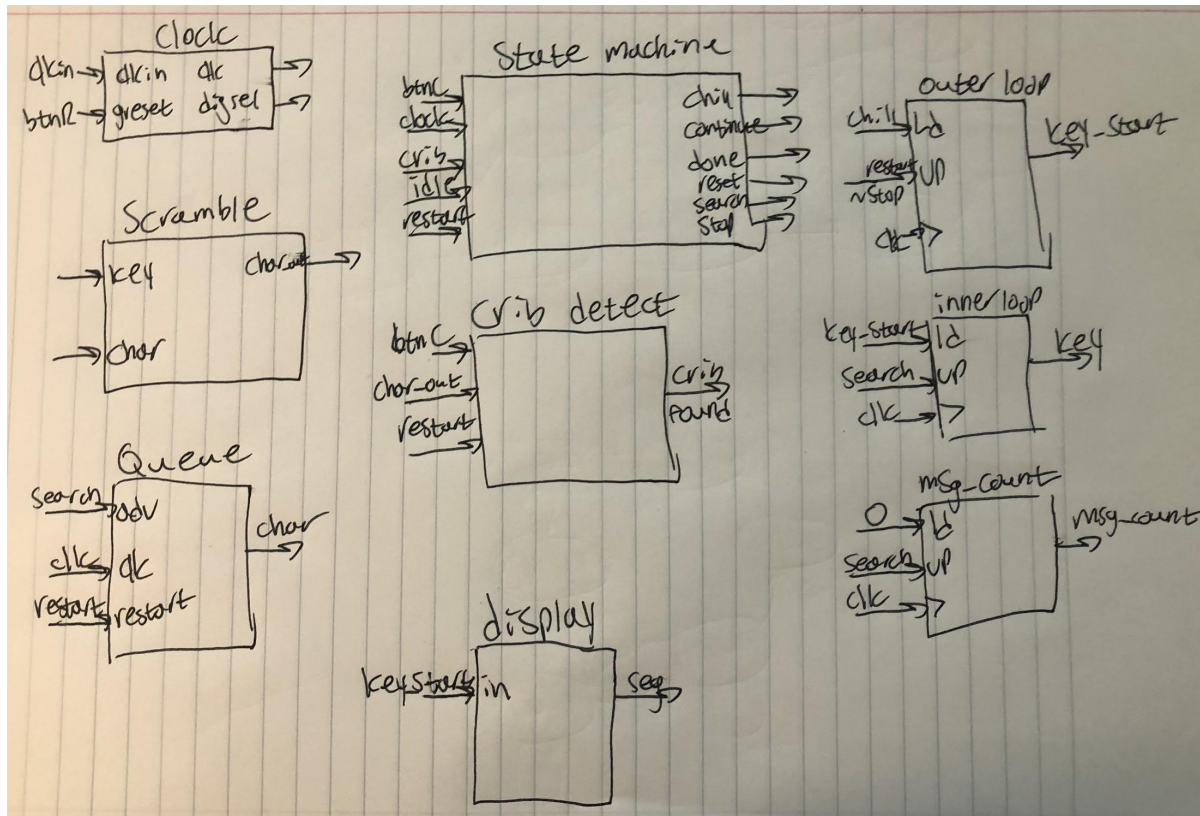
```
assign d[3] = restart;
```

```
assign d[4] = reset;
```

```
assign d[5] = idle | s[5];
```

Part 6: Top Level

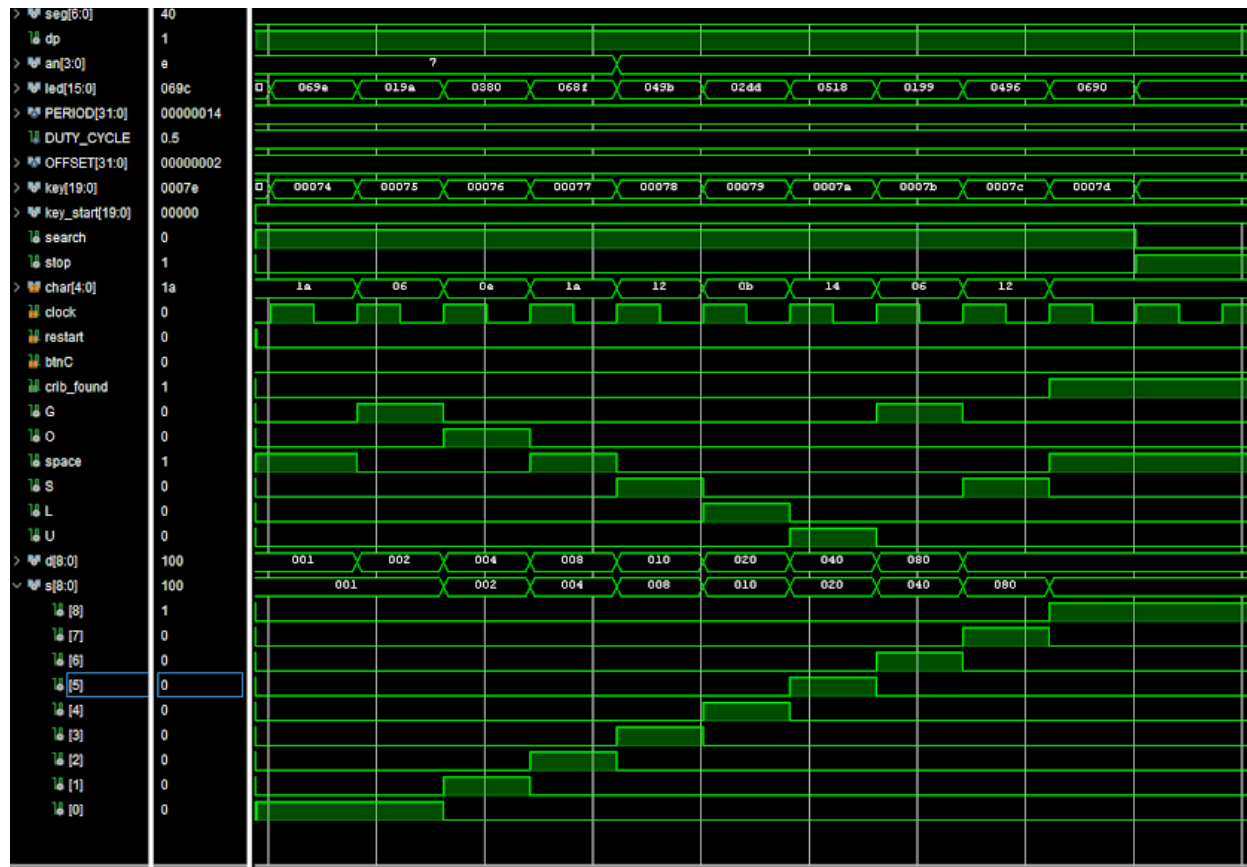
Here I will go into detail about how my entire system works and how the modules interact with each other. The board begins in an idle state, with dashes showing on all the displays. When the btnC is pressed it begins the searching process. We are given a 128 character message that we must decode with every key from 00000-FFFFF. However each key must also be incremented from starting key \rightarrow starting key + 128. Meaning for the key 00000 the first character would use 00000 and then the second character would use 00001 and so on and so forth. After the key has incremented to the end of the message the starting key should increment by 1, and the process of starting key \rightarrow starting key + 128 should start over. In order to accomplish this I implemented three counters. One I call the “outer loop” that is the main key counter, holding the value for the current key being tested. The second counter is called the “inner loop” and will load in the value of the outer loop counter every restart after it increments. This is the reason why I have multiple reset variables. If I load the inner loop counter at the same time I increment the outer loop the inner loop will load in the previous value, making the outer loop representing 1 and the inner loop representing 0, which will obviously cause issues. To solve this I just make the inner loop load a clock edge after the outer loop increments. The third counter counts from 0-127 so that we know when an entire message has been processed. This is also what makes the first restart wire go high. Using the search output from the state machine as a constant increment, the inner loop and msg length counter continue to increment along with the module queue that spits out the characters to decode. These characters are fed into a module titled scramble that decodes the character. Once it is decoded it can be sent to the crib detect module to see if it will advance the state machine. Once a crib is found the counters will stop and the current key will be displayed on the seven segment display. If the current key does not give the desired message then another press of btnC will continue the search. The entire schematic is shown below.



It is also worth noting that despite using a 20 bit key, we only have 4 seven segment displays at our disposal, meaning we can really only show 16 bits. To get around this I assigned the 5th bit of each display to the led below, if it's on that signifies that the 5th bit is one. After quite a lot of trial and error involving the edge cases (such as separating the reset variables to deal with the crib being at the end of the message) I was able to get my machine working completely.

Part 7: Simulation

Below is the waveform from a simulation in which a crib is located. Notice how it goes through all of the states in my crib state machine, then makes crib go high which in turn makes stop go high, waiting for a btnC press to continue or for the global reset to be hit.



As requested I will also display the intra-clocks timing report.

Clock	Edges (WNS)	WNS (ns)	TNS (ns)	Failing Endpoints (TNS)	Total Endpoints (TNS)	Edges (WHS)	WHS (ns)	THS (ns)	Failing Endpoints (THS)
clk	rise - rise	614.368	0.000	0	1415	rise - rise	0.126	0.000	0
sys_clk_pin									
clk_out1_clk_wiz_0	rise - rise	37.822	0.000	0	4	rise - rise	0.173	0.000	0
clkfbout_clk_wiz_0									

Results

In the end my board was able to run all of the test cases, even the really weird edge cases. The maximum clock frequency is 38.46 Mhz. This makes the time it will take about 3.4 seconds to go through all the keys at the max frequency.



Conclusion

This lab was a monster. I went into it with absolutely no idea as to how I was gonna accomplish it. But I slowly chipped away at it day by day, working out the main logic until I had the skeleton down. After that I probably spent at least twice as long figuring out how to get every little edge case to work, all of that to be checked off after showing two tests that I had finished three days before completing the entire project. But I digress, I'm proud to have gotten this working, and I feel as though this lab taught me more about logic design than the rest of the class.