Class Project

Brendan Bovenschen A20323473, Landon Fox A20319340

Section 1: Introduction

This project has been the culmination of our work in this class bringing together every aspect of digital logic that we've learned in these labs. Here we used a finite state machine, an encryption system (specifically DES), registered data, and a testbench that makes sure all of these things work together. Similar to one of our previous labs (Lab 2), we are using DES encryption. Our final goal in this lab is to determine the key for the provided plaintext and ciphertext using the logic tools provided to us. These systems include a counter (UDL counter), a DES encryptor, and a dynamic size comparator and flip flop. Although we were given a state diagram, we were required to develop our own state machine to control our larger combinational block. We completed both an original design containing one counting system as well as an alternative consisting of 16 parallel crackers. This was done to limit our max cracking time from 2E64 searches down to 2E60. A significant decrease in time could be found by adding more of these blocks in parallel, but 16 is substantial enough to demonstrate the effectiveness of this method. Since this alternative system mainly acts to reduce the chance of keys containing larger values, it sees little advantage when presented with the smaller keys used in this lab.

Section 2: Baseline Design

The project mainly depends on the DES system and the FSM we have created combined with the other logic it will provide us with a key when given the plaintext and ciphertext. The FSM of 4 states, being initial, up-count, store, and found key (as shown in figure 1) act as the sole control logic for our system. To begin our process, our system stays idle until our start signal is activated. This then begins our up count state. First, our 56-bit counter is iterated by 1. This value is piped into our parity module to properly generate our test key. Each set of 8 bits within our key is outfitted with a parity bit in the least significant position. This key is then used by our DES module to generate a new stream of ciphertext which is separate from the entering ciphertext. Both the ciphertext and key are registered through 64-bit flip flops, each of which is controlled by en1 and en2. This is done so a key is not sent to our final output until the proper key is found and the ciphertext is not stored during the up-count state. The generated ciphertext from this permutation is then compared with the original through the comparator module which determines if the key has been found. All of our logic is encapsulated into modules to keep our design human-readable and readily modifiable.

Within our waveform diagram (figure 5) our state transition process can be visualized. While in the up-count state, our count is iterated and during the store state, the ciphertext is generated and registered Once the key is found, our system enables the temporary key to be registered and piped out of our system.

Section 3: Detailed Design

By registering our key and our ciphertext, no incomplete or incorrect values are computed and displayed by the system. If no registered data were included in our design, a stream of incorrect keys would be exported potentially causing errors in a larger project utilizing this design. Alternatively, this entire system can also be encapsulated and repeated among multiple instances. Doing so results in a faster search process when done in parallel by covering different areas of the counted space. First, the starting number of each UDL counter must search a different part of the total 56 bits of counting (most significant hexadecimal digits 0-f). To achieve this, each UDL counter's "in" was hardcoded to one of these values. The final key is controlled by each key generated from the set of modules. These are controlled by a 16:1 multiplexer. The "set" input is determined through a combinational system which sets the key-out of the multiplexer to the value of the key whose module has an active "FoundKeys". To properly determine the larger "FoundKey", separate from the "FoundKeys" array, a unary-or operator was used on the multidimensional array containing all 16 smaller "FoundKeys". This is done so the control logic is able to transition to the FK state when multiple crackers are in use.

For our implementation, we controlled the 7-segment display with two switches (two bits) to switch between the hexadecimal digits of our key. In figures 4.2-4.5, this can be seen as we switched through to determine

if our key had generated as expected. Our start input was also controlled by a switch and the reset was implemented to a button.

Section 4: Testing Strategy

To test our design, we hard wired our plaintext and ciphertext in the testbench and created a module from our top file which contains the engine for the cracker, i.e. the counter, flip flops, the DES, comparator, and FSM all with their variables properly assigned. We first decided to see if the comparison of keys was functioning as expected. With our testbench from lab 2, we were able to input the key "0101010101010101" to receive an initial ciphertext. This was done so we could complete all our tests on the first clock cycle without having to wait minutes to determine if the system would compare the ciphertexts correctly. Once this test generates the expected key, we could now determine the final key for this lab. Figure 3 contains the key found in our system and the count from which this key was generated. Going back to lab 2, we were able to prove that our key would generate the same ciphertext provided to us. In figure 2, a screenshot of our tests using lab 2's testbench shows that our key and the ciphertext generated from it match what was given to us. Finally, we transported our design onto our board to determine if the key would display as expected when computed through our implementation. As shown in figures 4.2-4.5, each segment of our display matches the key in our simulation.

Section 5: Evaluation

In this lab, we have gained experience in putting many facets of digital logic together into a machine that can find a key when given plaintext and ciphertext. We've begun to create an understanding of encryption systems through our work in this project by exploiting the flaws of DES. Developing around this encryption standard relates to the field of cybersecurity and by cracking a known system we can begin to understand ways to better these designs. Since cybersecurity is such a broad field, this specific project can harbor skills in many future studies and professions. Although we have worked with flip-flops in previous projects, this is our first implementation of registered data. To keep incorrect or incomplete information from transferring to other modules of computational logic, this data can be registered to output on a specific clock cycle. Also, by using many pieces of logic that were already designed and functioning independently, it presented us with an opportunity to understand how we can implement larger pieces of logic without having to develop all of them ourselves. This is important for team environments where labor on a project would be split among members and these parts would be aggregated to create a higher abstract product. This experience of workflow is crucial for developing alongside others and communicating information within our written design through comments and documentation.

Figure 1 (FSM Diagram)

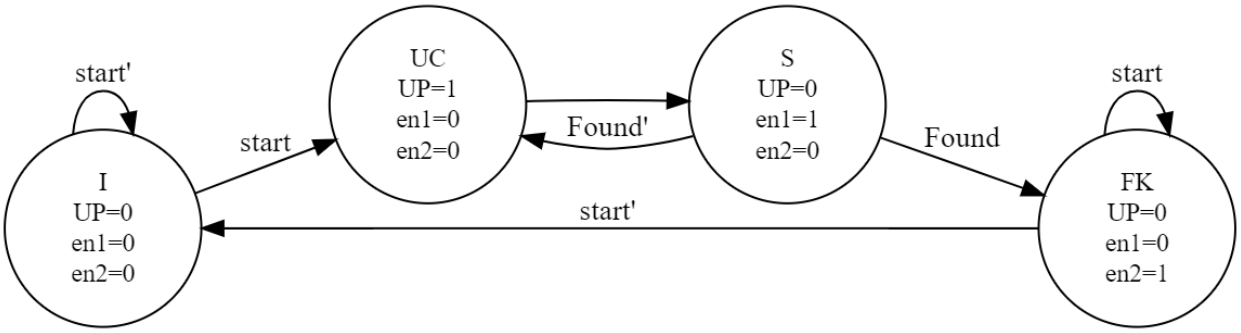

Figure 2 (Encryption Data and Testing Screenshot)

Plaintext	Key	Ciphertext
5eb98cbc40c4b52f	0101010101085b9b	a35e08ca3783f128

```
5eb98cbc40c4b52f 0101010101085b9b 1 || a35e08ca3783f128 || a35e08ca3783f128 1
```

Figure 3 (Screenshots from Simulation);

```
# Found Key1!
# ** Note: $stop    : tb_class.sv(45)
#    Time: 713875 ns  Iteration: 1  Instance: /stimulus_class
# Break in Module stimulus_class at tb_class.sv line 45
# Stopped at tb_class.sv line 45
```

```
0101010101085b9b 1 | 000000000116cd
```

Figure 4.1 (State “Idle”)

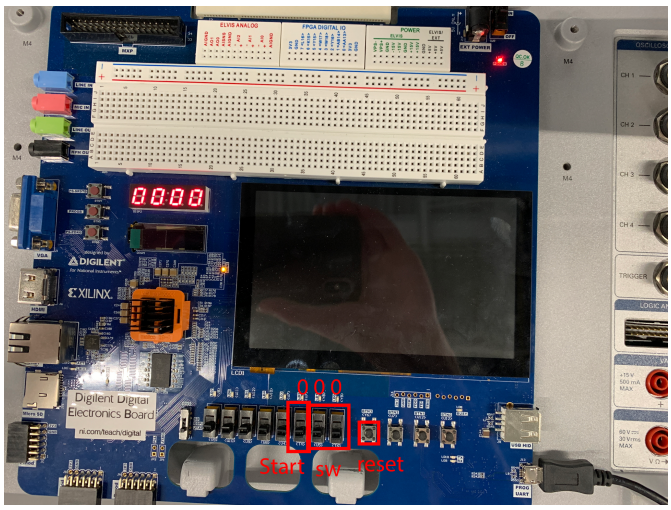

Figure 4.2 (Digits 0-15)

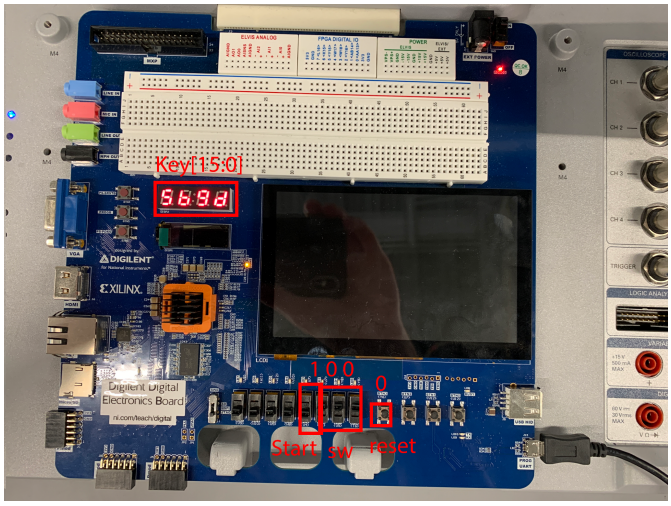

Figure 4.3 (Digits 16-31)

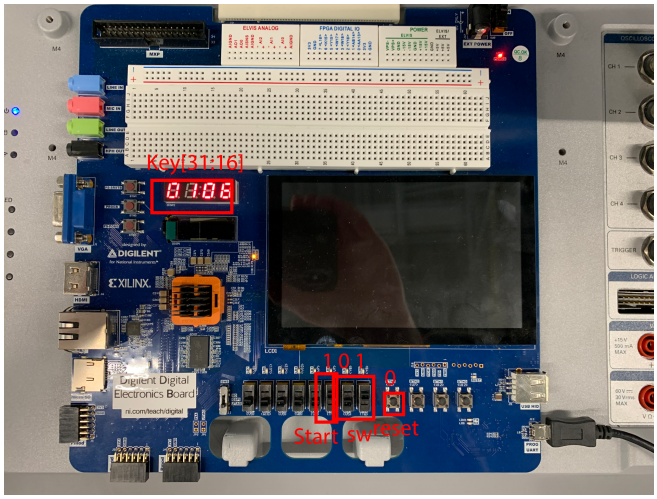

Figure 4.4 (Digits 32-47)

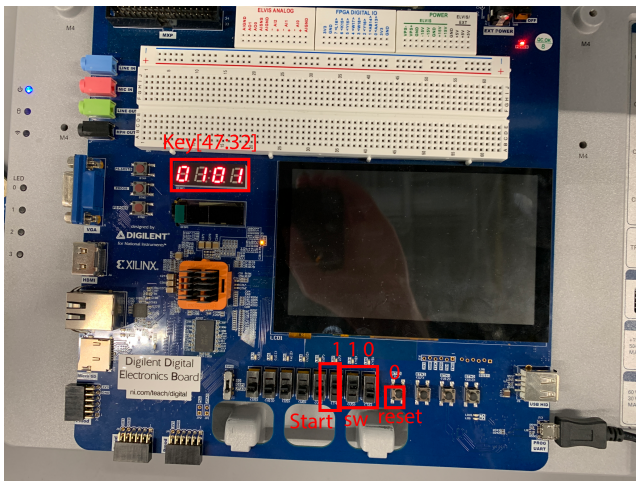

Figure 4.5 (Digits 63-48)

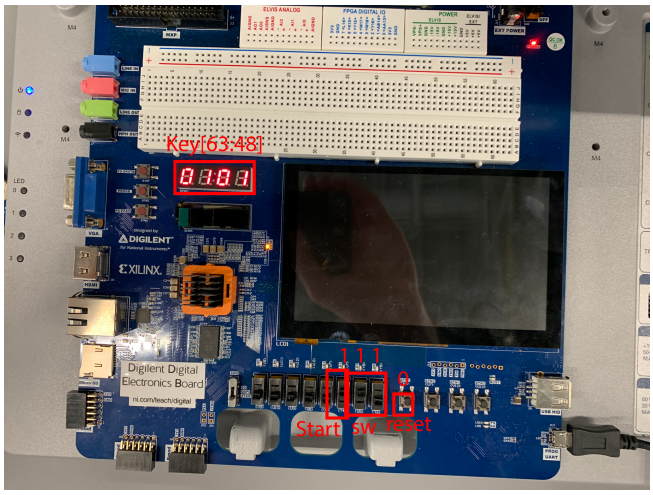

Figure 5 (Waveform)