Final Project:

Collin Bovenschen - A20323474
Katilynn Mar - A20312851

Section 1: Introduction

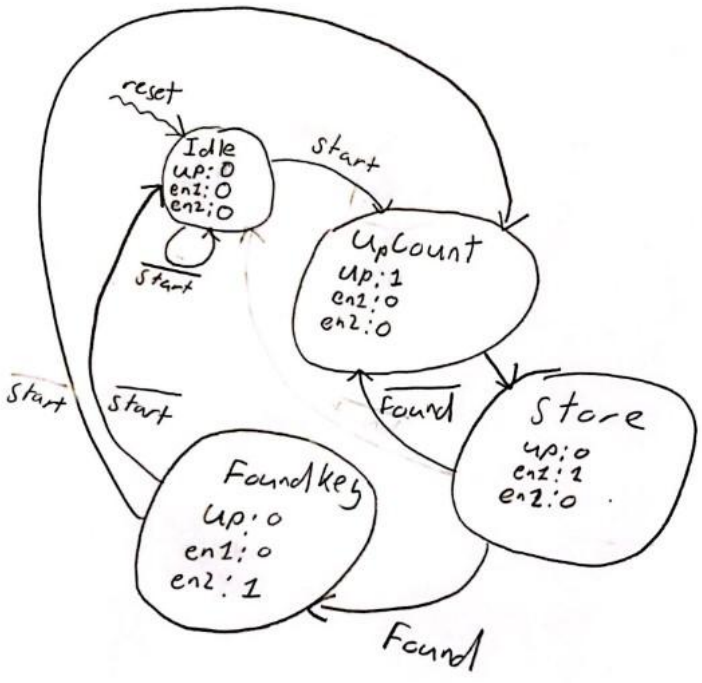
This Final Project tasked our team to create a cryptographic cracker, and we were employed to find a key when given a plaintext and ciphertext. Our key and ciphertext was only 64-bits of hexadecimal characters, which led to easier production of the unknown key. With this lab being our last, many different areas of the class are utilized. Required items from previous labs and lectures include a finite state machine, DES standard encryptor/decryptor, and combinational logic. All of these elements culminated into a functional system that was able to correctly obtain and produce an output.

Section 2: Baseline Design

For this project, we were provided with the DES module that is used to encrypt/decrypt the plaintext provided. From there we had two main goals that needed to be completed. We were provided a figure to follow for the design of the Top Module. Then we needed to create logic that would find the key within a reasonable time frame. The logic being a simple FSM or finite state machine that will act as the brain or the control for the whole project. With these main objectives completed we start the program and it begien to use the UDL counter to count up until it reaches the key for the plaintext and ciphertext we were assigned. Once a key is found it goes through the DES block with the plaintext where it is encrypted and outputs into the ciphertext and the key goes to a register. We then compare the ciphertext with the original to see if they are equal. If true, the bit is returned under the "Found" output.

Section 3: Detailed Design

Our SystemVerilog files were structured so that they could all be called by a single file. This file (top.sv) ran all necessary instructions written within multiple different files. With the large amount of processes required, this structure allowed for easier readability, as well as better calling of all functions and variables from the separate files. One of these called processes was our Finite State Machine. The finite state machine is depended upon by a large amount of other processes in this system.



The state machine initially begins within its "Idle" state, where no outputs are drawn. If the "start" statement is true, the "UpCount" state is accessed. Start is always true throughout our design, therefore idle is simply a default state to account for any mismanagement of bits. "UpCount" determines what current bit process the FSM is at, or its position along the ciphertext and key. "Up" is high within this state, and is called specifically by the UDL Counter. This counter determines the current bit the system is checking at that specific clock cycle. "Up" is the only external variable that affects its current location along the key. After UpCount, the FSM will always progress to the "Store"

state. The outputs “en1” and “en2” generally serve the same purpose, except “en1” is used to write an experimental value, while “en2” is used to write a known value within the key.

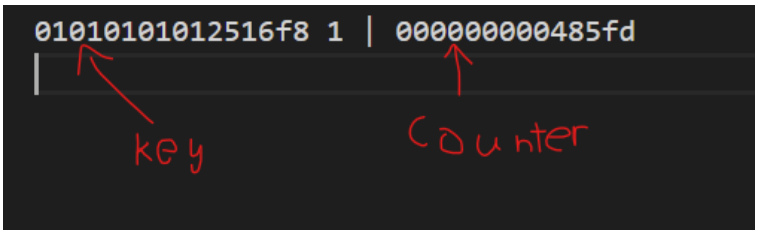
At the “Store” state, the DES encryption module is called, and the current counter location bit is called and tested. This encrypts the bit. After the plaintext is entered with a guessed key, its output is stored to a flip-flop register using “en1.” Finally, the ciphertext at a specific bit value is compared using a logic statement determining if they are equal. If equal, the “Found” Identifier is true, determining the next state entered. If high, the “FoundKey” state will be entered. If not, the system will return back to the “UpCount” state, incrementing the UDL Counter once again.

“FoundKey’s” function is derived solely within the storing of the correct key bit to a register. “en2” is high within this state, causing a second register to store the correct key value to its corresponding placement within the 64-bit key. When complete, the state will recurse back to either “Idle” or “UpCount” if the start boolean is set to low or high, respectively.

Section 4: Testing Strategy

The most efficient way to test the correctness of our program was to run it through ModelSim. Through ModelSim, an out file would be generated that would store the correct key after the program ran. Due to this program brute-force running through all possible key combinations at each bit, ModelSim took 5,929,035 nanoseconds to successfully encounter the key, or approximately 12 minutes. When generated in the GUI environment of ModelSim, each process can be viewed individually at every point along its timeline. All of these processes are saved to an extremely large .wlf file, spanning a size of approximately 7 gigabytes.

Through the testbench, the final key is exported to an out file, and can be viewed along with the amount of necessary attempts committed on each bit to obtain the final key. When uploading to the FPGA, similar steps to Lab 2 were taken. By 5importing our top.sv function into a vivado environment, we were able to assign values, inputs, and outputs to all needed values. Our found key was set to display to our four 7-segment displays, and we assigned two switches to alternate through all hexadecimal values. To begin the cracking sequence, our “Start” value was set to a switch, as well as the “Reset” input. When run, the board displayed the key as it was being determined, so the hexadecimal values incremented and changed as the instructions ran. When completed, the board displayed the correct key, as it matched the key within the previous statement.




```
Key:                    01010101012516F8

Encryption:

After initial permutation: 8E4646A8DF033D38
After splitting: L0=8E4646A8 R0=DF033D38

Round 1         DF033D38 91D62280 110060100802
Round 2         91D62280 154A7583 508818100104
Round 3         154A7583 124796AD 04A002800080
Round 4         124796AD 0921E722 220C06402201
Round 5         0921E722 39687A6E 682000320008
Round 6         39687A6E 69331F46 008438001102
Round 7         69331F46 E51A9BDD C40012042020
Round 8         E51A9BDD 891E21FD 268A00600840
Round 9         891E21FD C5CAB376 A01101428000
Round 10        C5CAB376 2700C0D3 010245800508
Round 11        2700C0D3 D553697B 015090081200
Round 12        D553697B ACD24D49 1401E0504020
Round 13        ACD24D49 9DEC4B43 924001000808
Round 14        9DEC4B43 D4C9B552 090304803010
Round 15        D4C9B552 403072AD 00108D210220
Round 16        403072AD 8424F683 480480024200

Cipher Text: 030D56022C3E8C47
```

To determine if the key generated by both the FPGA board and the ModelSim program was correct, the java DES program employed in Lab 2 can be reused. By entering the original plaintext along with the found key, a ciphertext will be generated by the DES module. This ciphertext perfectly matched our original given ciphertext, confirming that our system had performed its designated function correctly.

(Below depicts our given ciphertext, while the left image displays the generated ciphertext by the DES java program)

```
ciphertext_known = 64'h030d56022c3e8c47;
```

Section 5: Evaluation

Through designing, modeling, and testing , our team was able to successfully find the corresponding key with the given plaintext and ciphertext. To arrive at this conclusion, multiple design iterations were utilized. Initially, our design did not correctly instantiate parity bit modules to store the key values in their correct location. This caused unexpected key values, and even caused the program to run endlessly at some stages. Implementing these modules generated the expected key value, and allowed all bit locations from the key register to be stored within their respective location along the key. Another issue came with the testbench. Initially, our clock was not properly instantiated within our multiplexer that displayed our key through the 7-segment display. This resulted in only one of the displays generating and showing the key. After adding the proper clock, (smol_clock) the displays functioned as expected.