Lab 2
Brendan Bovenschen A20323473, Landon Fox A20319340

Section 1: Introduction

This lab focuses on the design and implementation of a block cipher (DES in our case) and then utilizes this to encrypt and decrypt plaintext and ciphertext using a set key. The absence of sequential logic in this project allows us to work solely with combinational logic. Many steps exist within the main block which makes testing imperative between each stage as the complexity of our design grows. This also required us to incrementally develop the large set of tools needed to complete our device. Each step required us to understand principles of combinational logic, as well as keeping track of our data since large sums of information were passed throughout each part of the circuit. Both the ciphertext and key contain 64-bits of information which greatly increases the difficulty of manually tracking each bit as this information passes through each part of the system. As we furthered this complexity, we were able to work at a higher level of abstraction through the instancing of our various modules. Specifically, each round of encryption contained sub-processes (straight function, expansion function, and the Feistel block) which could be hidden in abstraction near the end of our work. This was also the case with each round of subkey generation. However, we began our design process with the development of these tools to begin work on the final encryption operation.

Section 2: Baseline Design

Our DES module consists of 3 main phases: key generation, a 1:16 multiplexer, and 16 rounds of encryption (along with the initial and final permutation). This will take a 64-bit key, 64-bit plaintext, and a signal to switch between encryption and decryption. From here, we will determine whether or not the system will encrypt or decrypt the plaintext using the earlier mentioned 1-bit signal. The application of a block cipher in this system allows the process of encryption and decryption to act identically apart from the reverse use of subkeys for each round. From this knowledge, we can either pass our subkeys in the order generated (for our encryption cycle) or pass them in reverse to decrypt the entered ciphertext which is controlled by our multiplexer. Next are the main rounds of encryption. Each of these modules contains the same logic, but each is chained into the other to increase the cipher complexity of our output. The initial permutation module scrambles the 64-bit plaintext as per its designated swaps of bits before the rounds of encryption. After the plaintext has been enciphered through the 16 rounds, it is then altered by the final permutation module to produce the final output. For our board implementation, we were able to view the 16-digit hexadecimal characters through four 7-segment displays.

Because most of the main block exists in a high level of abstraction as the lower-level functions are run within each task, it is difficult to visualize how long this combinational logic would take. Although our entire system was completed within an extremely small amount of time, if utilized in a state system where it must be synced to a clock device, this complicated process could take more time than the clock's duty cycle causing problems with other components that rely on this circuit. Even though this system doesn't require any sequential logic, it could cause these issues while being implemented in another system. Syncing different components would solve these proposed issues, but it's not necessary for the scope of this project.

Section 3: Detailed Design

Within each of the larger modules inside the DES system are smaller logic operations to execute the low level math of the encryption process. For the subkey generation, the initial 64-bit key is split into the most significant (32-bit left side) and least significant (32-bit right side) half. These are piped into the PC1 module which switches specific data elements within the two halves around into two 28-bit data blocks. Next, the left and right bit arrays proceed through 16 rounds of subkey generation to be used in each of the round modules. Each of these rounds of generation includes a bit shift followed by a PC2 module which acts similar to that of PC1 by switching around explicit parts of the left and right subkey. The bit shift is conducted in SystemVerilog through a bit swizzle, in this case to the left. The PC2 module acts similar to the PC1 module, instead of producing a 48-bit subkey. This process is repeated until each of the subkeys has been produced. After the subkeys have been sent through the

previously mentioned 1:16 multiplexer, the round operation can proceed. The encrypt input in our DES module acts as they select for the multiplexer. This determines the orders of the subkeys with a high signal running encryption and low running decryption.

The round block functions to pass the previous round of ciphertext along with that round's subkey to produce a 64-bit ciphertext output. While the outputted left block is written to the previous right block, the right block must undergo a series of operations to encipher half of the full block. The entered right block is first piped into the Feistel block. In the Feistel block, the 32-bit input is first expanded through a series of duplications and bit swaps to create a 48-bit array that is then XOR'd that round's subkey. The output from this is then split for 8 different SBox modules which, through cutting each section of 6 bits to 4, reduce the size back to 32-bits. Finally, the left input of the round is XOR'd with the data from the Feistel block to create the final right half of the round. After the final switching of bits in the final permutation, the data is finally fully enciphered.

To display each part of our 64-bit, 16 character hexadecimal ciphertext, we utilized a 2:4 multiplexer to change between sections of 4 character text on 7 segment displays. This is controlled by two switches representing the binary position of each starting bit divided by 4. With this, we could view sections of the final ciphertext. For example, in figure 3, the 64-bit hexadecimal output of ecb54739a1832ec5 can be split into 4 sections with the control of 2 switches. Since the button is pressed (high signal), this runs the encryption cycle.

Section 4: Testing Strategy

Since this system has a 64-bit plaintext input, key, and a 64-bit ciphertext output, the mass amount of data required us to conduct rigorous testing to ensure that our system was void of small errors early in our design process. To test if our key generation and encryption system was working as expected, we used the provided Java program. When given the plaintext and key, it will output all of the 16 rounds of encryption as well as the final ciphertext as well as the same for the decryption. Using the same plaintext and key for both the Java program and HDL program, we first tested each round of subkey generation and compared both systems outputs. This was done to avoid tracking down small errors that would greatly affect the final data. The same process was used when developing the rounds of encryption. Next, once we had one valid test case, we could generate more known cases using the same Java program. This was done 5 times for encryption and 5 times for decryption to ensure that both functions of the system would match our known. To compare these cases without analyzing each hex digit manually, we utilized the tools provided for us in the DES testbench. By matching these cases within SystemVerilog and sending their equivalence value to an "equals" signal, the performance of the system can be shown as a 1-bit signal. This can be seen in figure 1 as the final value of the table for every case. The waveform (figure 2) is another way of visualizing this information as a sequence of operation rather than a text file. Then, in our implementation, we output the ciphertext in the seven segment display and once again compare them to the program to ensure our design's accuracy. As shown in figure 3, the ciphertext can be compared to line 3 in figure 1.

Section 5: Evaluation

From the work done in this lab, we gain experience in principles of encryption, complex combinational processes, and the abstraction of modules. By designing a system that encrypts a 64-bit plaintext and shows each of 16 rounds of encryption we were able to better observe the process that takes place when encrypting text, therefore giving us a better understanding of how encryption and decryption work with this data encryption standard. Since we were working with a large portion of data, we were able to reuse concepts learned in lab 1 using a testbench and incrementally develop our system. By separating different functions into modules, we were able to produce a user-friendly design for each process and subprocess.

Although we had little variation in design, we could have copied each block from each module with different variables. However, this would not effectively utilize the benefits of our HDL and significantly decrease the order and readability of our system while maintaining relatively identical functionality. With the guidelines and structure set in this lab, it was difficult to deviate a great amount from the expected design while still processing the data correctly.

Figure 1

plaintext	key	encrypt	ciphertext	expected-ciphertext
123456abcd132536	133457799bbcdff1	1	f77bcd7dfe57e119	f77bcd7dfe57e119 1
2579db866c0f528c	433e4529462a4a62	1	ecb54739a1832ec5	ecb54739a1832ec5 1
ed7bc587a26f8c67	3b3898371520f75e	1	ea37231a9ad2e5d9	ea37231a9ad2e5d9 1
318101b45f32078d	0e329232ea6d0d73	1	7f0ec241ebbdcf2b	7f0ec241ebbdcf2b 1
19e947f93a938dc9	2940ecd38901a89c	1	939df8ee14d8376f	939df8ee14d8376f 1
f77bcd7dfe57e119	133457799bbcdff1	0	123456abcd132536	123456abcd132536 1
ecb54739a1832ec5	433e4529462a4a62	0	2579db866c0f528c	2579db866c0f528c 1
ea37231a9ad2e5d9	3b3898371520f75e	0	ed7bc587a26f8c67	ed7bc587a26f8c67 1
7f0ec241ebbdcf2b	0e329232ea6d0d73	0	318101b45f32078d	318101b45f32078d 1
939df8ee14d8376f	2940ecd38901a89c	0	19e947f93a938dc9	19e947f93a938dc9 1

Figure 2

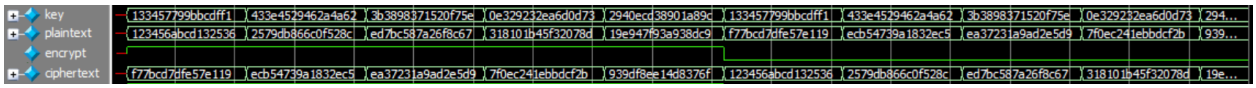

Figure 3.1

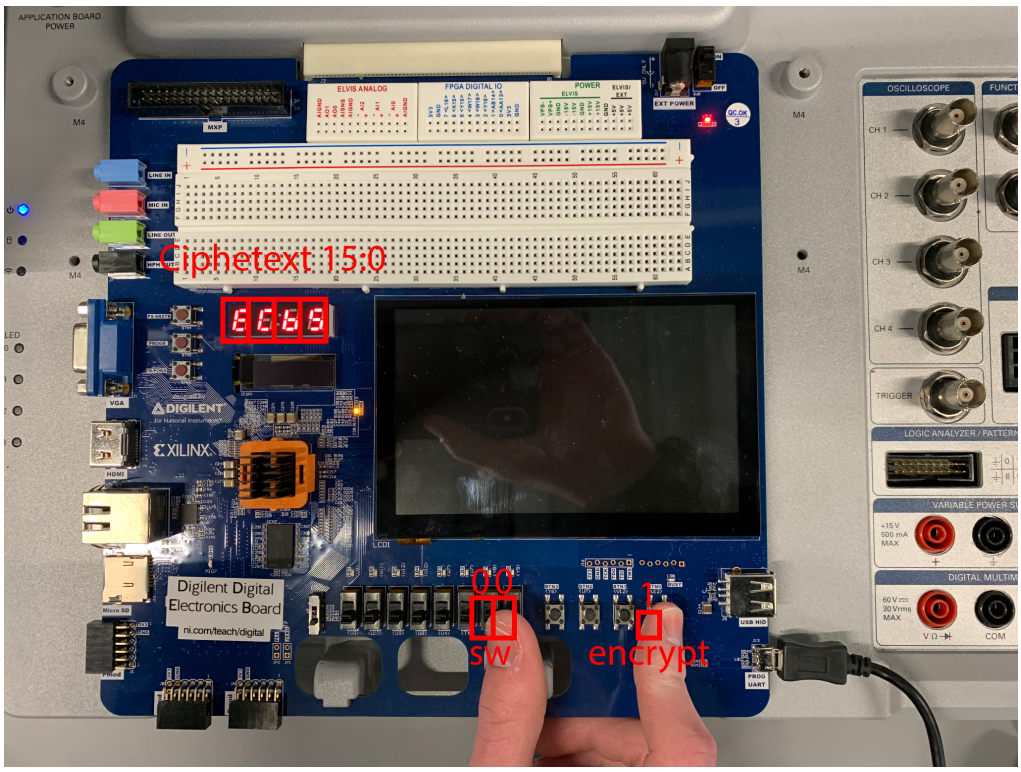

Figure 3.2

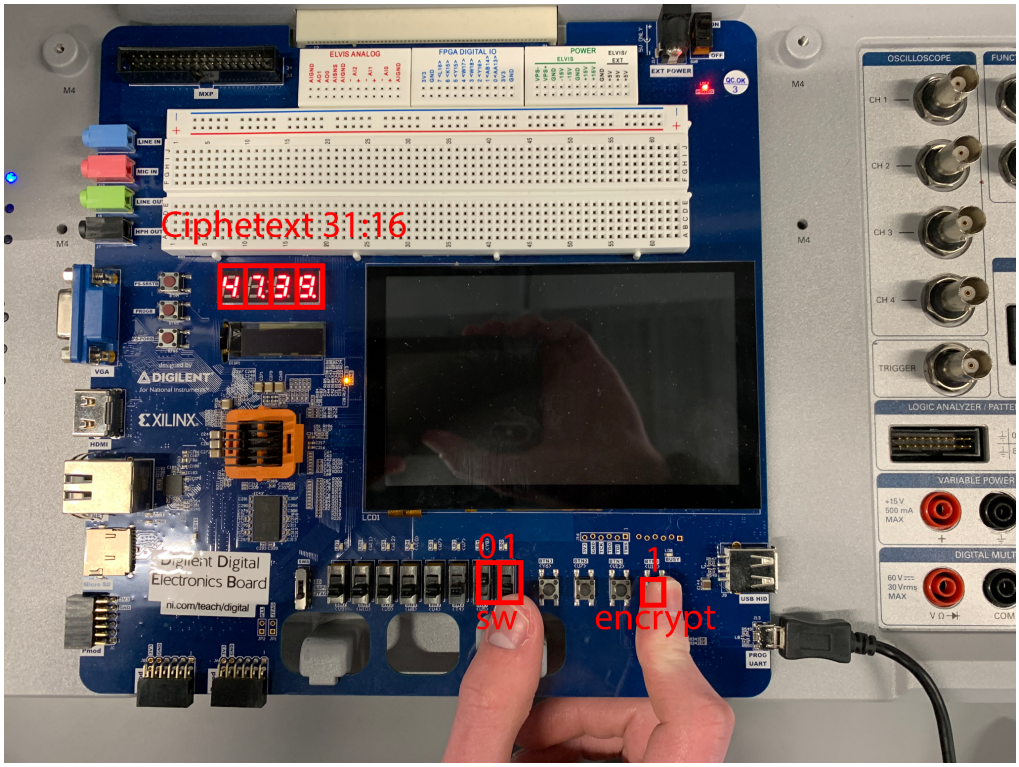

Figure 3.3

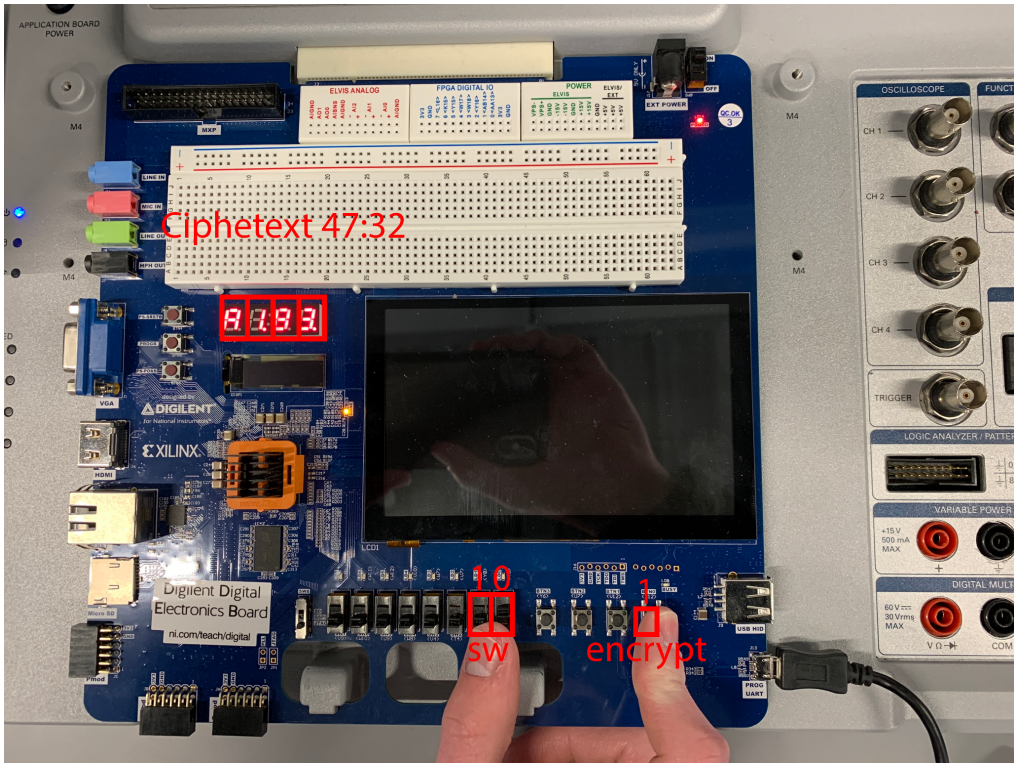

Figure 3.4