Lab 2: Complex Combinational Logic and Debugging : Hardware-based Data Encryption Standard (DES-ECB/DES-CBC)

Assigned: Monday 9/25; Due **Monday 10/16** (midnight)

Instructor: James E. Stine, Jr.

1. Introduction

Digital systems are important in all areas of society and using combinational logic is a key element to this development [1]. This laboratory will give you more experience with combinational logic for digital systems. Security is a major design concern for all devices, including those we use every day, such as cellular phones and computers. This laboratory will deal with a security cipher that was important in the 1990s. However, this security encryption standard, called Data Encryption Standard (DES) [2, 3], fell out of favor because we could use digital logic to help break into these devices.

For this laboratory, we are going to develop a DES crypto-based hardware implementation in two parts. The first part will involve designing the DES encryption and decryption units found in this laboratory, and second laboratory which we will use for our project will be a cracker or a device that can break into the device via hardware. Security is not only important but many people feel that its one of the most important topics that engineers need to learn in the 21st century. Therefore, I believe this laboratory will be a great experience in learning some security and the basics related to making sure someone does not have unwanted guests within their systems. The ideas can also be translated easily into more advanced cryptographic systems, such as Advanced Encryption Standard (AES) and SHA 256 hash function that is commonly used in bitcoin and web-based authentication.

DES is an example of a block cipher, which takes as its input a fixed length of plaintext and converts it to a fixed length ciphertext. The plaintext and ciphertexts are both 64-bit in DES. DES also takes an additional 56-bit key as it input. The key scrambles the plaintext such that different keys with the same plaintext produce different ciphertexts. In addition to supporting encryption, DES decrypts the ciphertext into the orignal plaintext using the **same** key. This is called symmetric encryption as the same key is used for both encryption and decryption.

The DES encryption algorithm involves two permutations (P-boxes), which we call initial and final permutations, and sixteen Feistel rounds. Each round uses a different 48-bit round key generated from the key according to a predefined algorithm.

1.1 Security Basics

Encryption security can be broken down into the basic idea of using a password or a key to grant access to information. The message that we want to encrypt is known as the *plaintext* and the resulting encrypted message is known as the *ciphertext*. DES is what is referred to as a symmetric-key algorithm, where the same key is used to encrypt and decrypt information. Symmetric-key algorithms also benefit from straightforward decryption operations: decryption is either the exact same as encryption or all the steps from encryption simply performed in reverse-order. DES fits into the latter category. The block diagram of cryptographic hardware operations is shown in Figure 1.

1.2 Rotation

One of the most common operations in cryptography is called rotation. Rotation is similar to shifting except anything that is shifted out of a block gets put back into the block on the other side. In other words, a rotation or sometimes called a circular shift is an operation similar to shift except that the bits that fall off at one end are put back to the other end. It is easy to see this as an example.

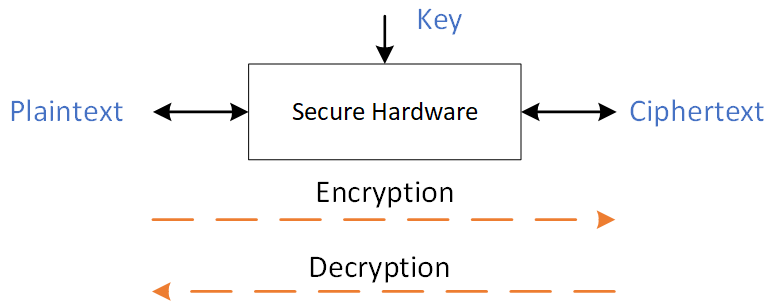

Figure 1: Basic Symmetric Cryptographic Hardware Block Diagram

If we have n that is stored using 8 bits. A left rotation of `n = 1110_0101` by 3 makes `n = 0010_1111` (Left shifted by 3 and first 3 bits are put back in least-significant positions. Fortunately, SystemVerilog (SV) makes rotation and shifting easy to create with bit-swizzling.

Bit swizzling is SV is achieved with the curly braces { and }. Using an example from our textbook [1], where y is given as a 9-bit value $c_2c_1d_0d_0d_0c_0101$ using bit swizzling operations. This can be created in SV by the following statement.

```
assign y = {c[2:1], {3{d[0]}}, c[0], 3'b101};
```

In reality, the {} operator is used to concatenate busses. The `{3{d[0]}}` indicates three copies of `d[0]`. As stated in our textbook do not confuse the 3-bit binary constant `3'b101` with a bus named b. It is important to note that it is critical to specify the length of 3 bits in the constant; otherwise, it would have had an unknown number of leading zeros that might appear in the middle of y. If y were wider than 9 bits, zeros would be placed in the most significant bits.

2. Data Encryption Standard (DES)

The Data Encryption Standard (DES) algorithm, adopted by the U.S. government in 1977, is a block cipher that transforms 64-bit data blocks under a 56-bit secret key, by means of permutation and substitution. It is officially described in FIPS PUB 46 [2]. The DES algorithm is used for many applications within the government and in the private sector.

DES uses lots of rotations, swapping or sometimes called switching, and the use of the exclusive OR operation. The exclusive OR or xor is useful in that it can be utilized for numbers that have properties that are sometimes called modular arithmetic. We will not go into the theory behind this idea, but you are welcome to explore more of cryptography in these great texts [4, 5] [1].

The symmetric-based cryptographic algorithm of DES block diagram can be seen in Figure 2. Although Figure 2 looks intimidating, it is just simple combinational logic for each block. This will be excellent practice in trying to create some combinational logic from scratch as well as honing some excellent debugging skills. In order to break down each block, we will break this into two sections : one for subkey generation (e.g., K_1) and the other for the basic symmetric cryptographic algorithm.

2.1 SubKey Generation

In order to start, the best place to start is producing the subkey generation. The key generation involves two basic blocks:

1. Rotation by either 1 or 2
2. Two Permutation Choice Box

[1] The reference [4] is available electronically in Edmon Low library at `http://library.okstate.edu`. It an extremely good reference for those interested in cryptography

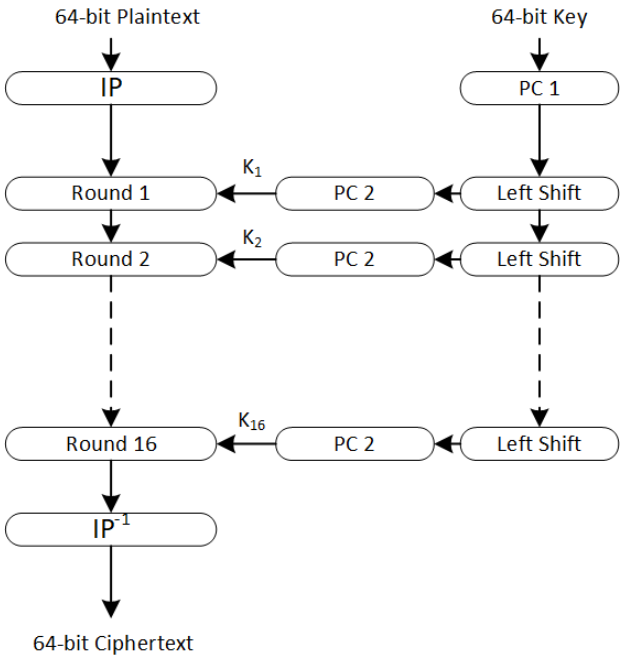

Figure 2: DES Block Diagram

K_{63}	K_{62}	K_{61}	K_{60}	K_{59}	K_{58}	K_{57}	K_{56}	...	K_7	K_6	K_5	K_4	K_3	K_2	K_1	K_0
X	X	X	X	X	X	X	P_7	...	Y	Y	Y	Y	Y	Y	Y	P_0

Table 1: Parity Locations within 64-bit key

The central element to this block is that it produces all sixteen (16) subkeys that get used during encryption and decryption. More advanced algorithms, for example for those found in [4], use the same idea with either the more or less keys to produce more security.

Rotation is performed as explained earlier except that the some blocks are rotated once and the others block rotates twice. The only caveat here is that **each rotation** should be done by separating each block of the key into 28-bits or broken into two groups. This kind of confusing as the key is 64 bits. This is because the key has something called parity bits that protect it from potential errors.

2.1.1 Parity Bits on the Key

A parity bit is a bit added to a string of binary code. Parity bits are a simple form of information to detect any errors. Parity bits are generally applied to the smallest units of a communication protocol, typically 8-bits (bytes), although they can also be applied separately to an entire message string of bits.

The parity bits on the key are done on each group of 8 bits of the 56 bit key making the total numbers of bits 64. The locations of the parity bit are shown in Table 1. There are many examples in engineering for parity but the key is to detect any difference in the original or transmitted message from the received one. The parity within the key is computed simply by XOR'ing of the previous 7 bits. For example, in Table 1, P_7 is found by XOR'ing the 7 X bits and P_0 is found by XOR'ing the 7 Y bits. This form of parity is sometimes called odd parity where the occurrences of bits whose value is 1 are counted. If that count is even, the parity bit value is set to 1, making the total count of occurrences of 1s in the whole set (including the parity bit) an odd number.

Iteration #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Left Shifts	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Table 2: Left Rotation within SubKey Generation

2.1.2 Rotations within each SubKey

As indicated previously, the key is rotated or circularly shifted by 2. Rotating is central to the idea in most cryptographic algorithms as it mixes up the original message similar to how a deck of cards is rotated. For the key, rotation happens based on the round that it is in. As indicated previously, DES uses 16 iterations or rounds and the rotation happens on different iterations, as indicated in Table 2. During each iteration, either one or two circular left shifts on both $C[i-1]$ and $D[i-1]$ to get $C[i]$ and $D[i]$, respectively.

Let's try an example to illustrate this by showing a rotation by two (2) on `10_1011_1000`. To accomplish this rotation, first break the 10-bits into two groups of 5-bits or `1_0101` and `1_1000`. Then, the rotation is done on each block separately or: `1_0110` and `0_0011` That is, the final rotation will be the concatenation of these two items or `10_1100_0011`.

2.1.3 Permuted Choice

Rotations within cryptographic algorithms are sometimes done with specific mathematics in mind. There are several of these ongoing within DES and they are typically called permutations. Inside the key logic, several permutations are computed to generate the subkey (i.e., K_i). They are called Permutation Choice 1 and Permutation Choice 2 or PC-1 and PC-2 as shown in Figure 2, respectively. These permutations do not use the parity bits discussed earlier as the parity bits are utilized to detect any errors with the key. Figure 3 illustrates how each of the key bits are allocated for each Permuted Choice (PC) function. The PC-1 operates on 56 bits of the key where C_i and D_i represent the left and right side of key logic, respectively. The PC-1 permutations for the left and right hand side (i.e., 32 bits for each) are routed as shown in Table 3 for each position. For example, for the left side of the input, position 0 output is mapped to position 36 of the input key. At each round, PC-2 takes each block from PC-1 left and right hand side to produce the necessary subkeys K_i for the encryption and decryption. The PC-2 permutations for the subkey are shown in Table 4

Left						
57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
Right						
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

Table 3: Permutation Choice 1 (PC-1) Function[2]

14	17	11	24	1	5	3	28
15	6	21	10	23	19	12	4
26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40
51	45	33	48	44	49	39	56
34	53	46	42	50	36	29	32

Table 4: Permutation Choice 2 (PC-2) Bit Function

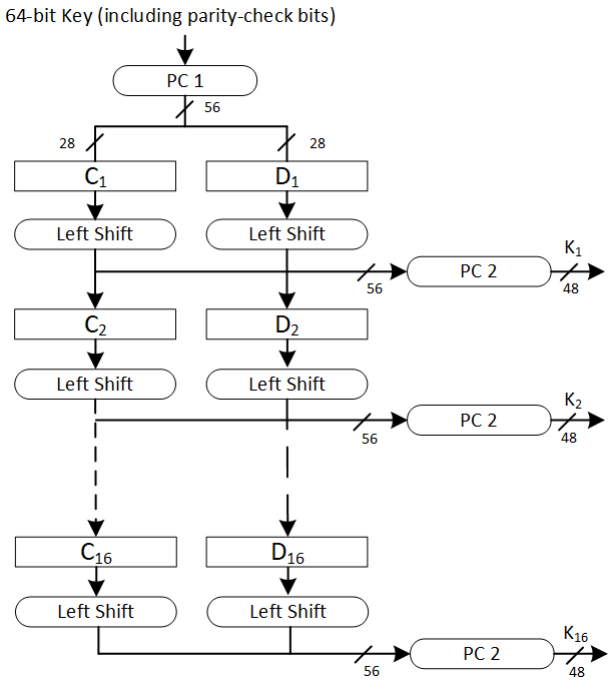

Figure 3: DES SubKey Permutation Diagram

It is important to note that the parity-check bits (namely, bits 4, 8, 16, 32, 40, 48, 56, 64) are not chosen, and they do not appear in PC-1. Also, PC-2 selects the 48-bit subkey for each round from the 56-bit key-schedule state from each rotation. Also, one of the more difficult elements in electrical and computer engineering digital applications, in my opinion, is what constitutes the Least Significant Bit (LSB). Some programs, such as MATLAB, have the first entry represented by vector 1; however, computers typically represent the first entry by vector 0. All of the tables, similar to Table 3, will signify the first entry as 1 when they really mean bit 0. Therefore, please be wary of this conundrum when implementing the permutations throughout this laboratory. .

2.2 DES Encryption or Decryption Function

As stated earlier, DES is a symmetric cryptographic algorithm in that it can be done either way for encryption or decryption. It is important to understand the order as shown in Figure 2 making the correct direction is utilized for encryption or decryption. Again, this block is another group of simple combinational logic blocks with it broken down into four basic blocks:

1. Initial Permutation (IP)
2. Feistel Block (f_K)
3. Switch or Swapping (SW)
4. Inverse Permutation (IP^{-1})

The Initial Permutation (IP) and the Inverse or Final Permutation (IP^{-1}) blocks are basically moving data around before each round is processed. These permutation blocks are similar to the previous permutation sequences within the subkey generation, but have different positions for their output. Both blocks process 64 bits of data as inputs and outputs and each position notation is shown in Table 5 and Table 6.

[2]As discussed in the text, for this and other permutation tables maps the inputs to the outputs (i.e., bits1 through bits64 which should map to position 0 to 63)

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Table 5: Initial Permutation (IP) Function

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

Table 6: Inverse or Final Permutation (IP^{-1}) Function

One of the more challenging elements is when designers and those that develop these standards refer to specific bits. Although Electrical and Computer Engineers are careful with these bits, I find that the security professionals are sometimes not very specific. This is definitely the case for DES as well as other cryptographic algorithms. If you look at Table 5 and all the other permutations, they refer to each block starting from the MSB and move towards the LSB. Unfortunately, this makes no sense what bit the standard is referring to and also makes things very confusing. I have modified the SystemVerilog to make the permutations easier to follow. As an example for Table 5, you can see the position is delineated from the MSB with some minor arithmetic shown in Figure 4. I left several examples inside the SystemVerilog files that come with the laboratory to help you follow along – please use this coding to help you make sure you reference the correct bit for any of the permutation tables that you need to create.

2.2.1 Round Block

The main part of this section is called the round block that utilizes a Feistel function that is named after the German-American cryptographer Horst Feistel. Horst Feistel main work was in developing ciphers for

```
// Initial Permutation
module IP (inp_block, out_block);

   input logic [63:0]  inp_block;
   output logic [63:0] out_block;

   assign out_block[63] = inp_block[64-58];
   assign out_block[62] = inp_block[64-50];
   assign out_block[61] = inp_block[64-42];
   assign out_block[60] = inp_block[64-34];
   assign out_block[59] = inp_block[64-26];
  ...
```

Figure 4: Sample SystemVerilog snippet showing position notation for Initial Permutation (IP) function

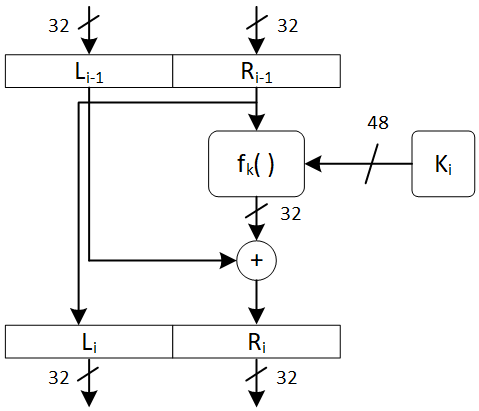

Figure 5: DES Round Block Diagram

IBM and we call this block the Feistel block after his pioneering work.

The Feistel block or f_K basically performs the exclusive OR'ing or XORing on blocks of 32-bits. Inside this Feistel block are key elements of symmetric cryptographic algorithms called substitution boxes or sometimes abbreviated as S-box. S-boxes are utilized in block ciphers to to obscure the relationship between the key and the ciphertext, thus ensuring Shannon's property of confusion [4]. For this lab, I will provide the S-boxes for you so you just have to compute the round block correctly. The order of operations in the round block are as follows:

1. Split the 64-bit IP output into two separate blocks of 32-bits: `L[31:0]` and `R[31:0]`.
2. Use the Feistel block to compute a value on the `R[31:0]` and $K_i[47:0]$ that can be used to XOR with `L[31:0]`.
3. XOR the output of the Feistel block with the `L[31:0]` and outputs into the next `R[31:0]` block.
4. The `L[31:0]` output is taken from the previous `R[31:0]` effectively swapping the two 32-bit values.

Figure 5 shows the basic block diagram of what is happening during each 16 rounds. If you look at Figure 5 closely, you will see that that each left and right sub-block is swapped during each of the 16 rounds.

2.2.2 Feistel Block

The main Feistel (f_k) block utilizes the S-boxes previously mentioned. The Feistel block also processes some data before it enters the S-box as well as when it exits the S-box using an Expansion Function or Permutation and a Straight D-box Function or Permutation.

The Expansion Permutation (EP) follows a predetermined rule. For each section, input bits 1, 2, 3, and 4 are copied to output bits 2, 3, 4, and 5, respectively. Output bit 1 comes from bit 4 of the previous section; output bit 6 comes from bit 1 of the next section. Please note that the number of output ports is 48, but the input range is only 1 to 32. That is, some of the inputs go to more than one output. For example, the value of input bit 5 becomes the value of output bits 6 and 8. This mapping is shown in Table 7.

After the expansion permutation, DES uses the XOR operation on the expanded right section and the round key. Note that both the right section and the key are 48-bits in length. Also note that the round key or subkey is used only in this operation. Finally, the straight Diffusion block takes in 32 bits and outputs 32-bits, as well. This mapping or permutation is shown in Table 8. Visually, the Feistel (f_K) block is shown in Figure 6.

The S-boxes do the real mixing (confusion). DES uses eight (8) S-boxes, each with a 6-bit input and a 4-bit output. The 48-bit data from the second operation is divided into eight 6-bit chunks, and each chunk is fed into a box. The result of each box is a 4-bit chunk; when these are combined the result is a 32-bit text. The substitution in each box follows a pre-determined rule based on a 4-row by 16-column table. Again, to avoid confusion this S-box is given to you in SystemVerilog. Also, as stated previously, the Feistel block operates with each of the 16 subkeys (i.e., it is done 16 times).

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

Table 7: Expansion Function or Permutation (EP) Function

16	7	20	21	29	12	28	17
1	15	23	26	5	18	31	10
2	8	24	14	32	27	3	9
19	13	30	6	22	11	4	25

Table 8: Straight Diffusion-box (D-box) Function

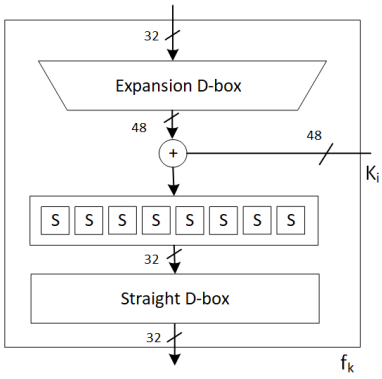

Figure 6: Feistel (f_K) Block Diagram

2.3 DES in CBC Mode (DES-CBC)

For this laboratory, I want you to implement both modes of operation for the Data Encryption Standard (DES) and have a switch that selects which version. These are called Cipher Block Chaining (CBC) and Electronic Code Book (ECB). The DES CBC encryption process requires a 64-bit cryptographic key similar to the ECB mode which is the default mode. There are other modes, but we will only implement CBC and ECB for this laboratory. The DES CBC encryption process also requires a 64-bit Initialization Vector (IV).

The process in implementing CBC is similar to ECB except that it requires an IV input, as well. The first step to initiating a CBC is to XOR the first of many plaintext blocks with an IV – a unique, fixed-length conversion function – to create a random, or pseudorandom, output. That is, it is simply that the plaintext is XOR'ed with the Iniatialization Vector (IV) right in the beginning of the process.

$$Plaintext_{new} \quad = \quad IV \oplus Plaintext_{input}$$

2.4 Encryption or Decryption

Encryption or decryption can be produced from the same block as shown in Figure 2. This is where the name symmetrical cryptographical algorithm comes from. The only difference is the order of operations and what Key is being used (i.e., K_1 or K_2). That is, decryption uses the same algorithm as encryption, except that the subkeys $K_1K_2, \ldots, K_{16}$ are applied in reversed order. Your design should have an input that tells it whether to encrypt or decrypt your block.

2.5 Summary of Algorithm

To give a summary of the algorithm, we will show the pseudo-code of what is happening. Pseudo-coding is done to give a very high-level view of an algorithm. As stated previously, the algorithm is not difficult, it just has many steps. If you know what has to happen at each step, it become easy to implement, especially with SystemVerilog. Algorithms 1, 2, 3 show the pseudo-code for each.

Algorithm 1 SubKey Schedule

Require: *key*
Ensure: `key[63:0]` with odd parity
  $\{C[0], D[0]\} = PC1(key)$
  for $1 \leq i \leq 16$ **do**
    $C[i] = LS[i](C[i-1])$
    $D[i] = LS[i](D[i-1])$
    $K[i] = PC2(\{C[i], D[i]\})$
  end for

Algorithm 2 Encipherment

Require: plain block, `K[1]` through `K[16]`
  $\{L[0], R[0]\} = IP$(plain block)
  for $1 \leq i \leq 16$ **do**
    $L[i] = R[i-1]$
    $R[i] = L[i-1] \oplus f_k(\{R[i-1], K[i]\})$
  end for
  cipher block $= FP(\{R[16], L[16]\})$

Algorithm 3 Decipherment

Require: cipher block, `K[1]` through `K[16]`
  $\{R[16], L[16]\} = IP$(cipher block)
  for $1 \leq i \leq 16$ **do**
    $R[i-i] = L[i-1]$
    $L[i] = R[i-1] \oplus f_k(\{L[i-1], K[i]\})$
  end for
  plain block $= FP(\{L[0], R[0]\})$

2.6 Power, Performance and Area (PPA)

For this laboratory, we are going to analyze the design with better PPA. That is, you should analyze your design for Power, Performance and Area. As opposed to previous laboratories, this procedure that will be documented here is more robust and gives better numbers that you can use to assess whether your design is credible or not. As with any digital design, engineers use PPA to assess the level of difficulty, challenge, and effort needed for a design.

To assess your PPA for this design, you should determine its PPA after implementation. This is because some of the PPA results (e.g., timing) are not adjusted properly until the Implementation phase. The Implementation phase typically places and routes the design onto the FPGA by connecting all the logic blocks that we read about in the article that we looked at in Lab 0 [6].

To obtain the PPA results, you first have to run through your design making sure that it is implemented correctly. Then, you need to add the following reports after the route stage (i.e., during Implementation):

1. `report_utilization` : Area
2. `report_timing` : Performance

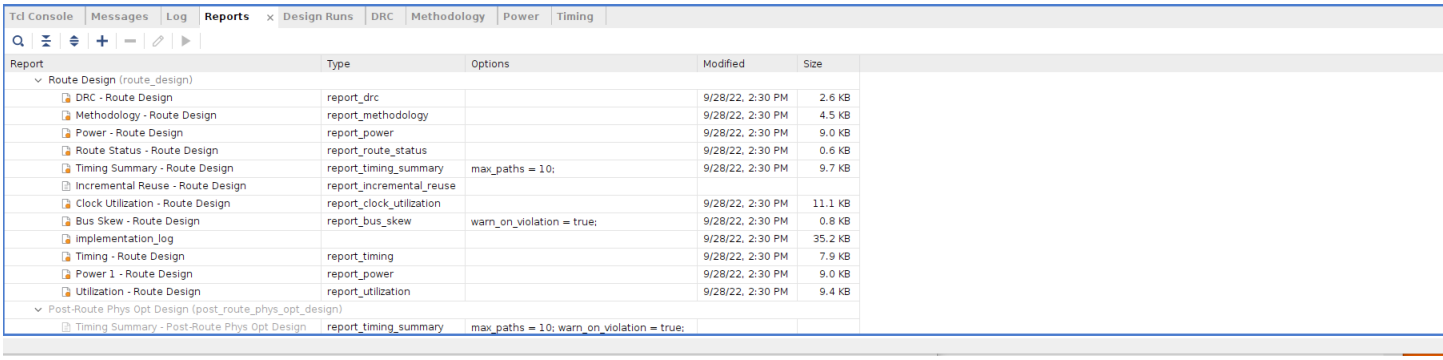

Figure 7: Reports Window within Xilinx Vivado

3. `report_power` : Power

You can the reports you need by clicking on the reports tab, right mouse clicking, and then adding the report you need, as shown in Figure 7. Once you add the report, it is easiest to re-run the implementation to get the report. Clicking on the option gets you specific report which you can save.

3. Tasks

Most of the blocks and their operation have been given to you to help you understand the problem better. For those that are interested in more about cryptography and how hardware can impact the future, I encourage you to read more about it through searching on the Internet as well as this great reference [4]. One of the hard parts of any engineering problem is to understand what is going on and making sure you are correct. Therefore, digital designers rely heavily on getting good data to make sure they are right. Typically, this is done either on paper and pencil or through software.

We will use software for this approach and use a piece of software written in Java. If you need to install Java on your machine at home or laptop, go to `https://www.oracle.com/java/technologies/downloads/#java16` and download the appropriate version. The main part of the encryption output looks like the following in Figure 8 after typing `java DES`. The plaintext and key are inside the `DES.java` program and can be easily modified, however, I wrote a method in Java that checks the parity so make sure you have a good key. For example, in round 1, `L_1 = 0x8C13_B66C`, `R_1 = 0xF3EF_C169` and `K_1 = 0x2080_66A2_53BA`. As seen by the output in Figure 8, the plaintext `0x2579_DB86_6C0F_528C` with a key of `433E_4529_462A_4A62` produces the correct ciphertext of `ECB5_4739_A183_2EC5`. This could be checked by taking the ciphertext and decrypting it through the algorithm. Since the algorithm is symmetric, it utilizes the same procedure for encryption or decryption except that the keys are reversed. There are several DES calculators available online through Google search if you wish to validate the result this way, as well.

Verification is extremely difficult because there are so many moving parts. Use the Java program to verify each block out of the HDL. Although the Java works based on bytecodes that are interpreted, I have found that some machines have problems reading the Java bytecodes. I am still not quite sure why this is the case, however, there is an easy fix. Therefore, I included a Makefile that I wrote that allows you to compile the Java correctly. Please type the following if you are having problems running the code. To run the tool, type `java DES` at the command prompt.

```
make clean
make
```

If you cannot run `make` on your Windows box, just type the `javac` commands found within the Makefile on each Java file (i.e., `javac -d . -classpath . DES.java`).

The main tasks for this laboratory will be the following elements:

1. Design the DES combinatianal block for both encryption and decryption in SystemVerilog and simulate with ModelSim.

```
Original plain Text: 2579DB866C0F528C
Key: 433E4529462A4A62

Encryption:

After initial permutation: 5646B9278C13B66C
After splitting: L0=5646B927 R0=8C13B66C

Round 1  8C13B66C F3EFC169 208066A253BA
Round 2  F3EFC169 25DAF255 C0B6508F6DC2
Round 3  25DAF255 1890CFBF 44D6422CC355
Round 4  1890CFBF AFB98FA0 62D142D3C4C6
Round 5  AFB98FA0 8F76DBD7 28C143CC8789
Round 6  8F76DBD7 C176D0E5 21411B9A764D
Round 7  C176D0E5 C7401A8C 2501917AD3A0
Round 8  C7401A8C B748825A 170891906D2B
Round 9  B748825A 61239171 084949255DD5
Round 10  61239171 FE28B577 01690D8B80F3
Round 11  FE28B577 CDB650DE 012D81C7CF05
Round 12  CDB650DE 8B8270E5 512CA11A07DC
Round 13  8B8270E5 DDDBEE19 D1A480D9D185
Round 14  DDDBEE19 5F82D63F 5086864266A9
Round 15  5F82D63F B35B4964 709006FA390D
Round 16  B35B4964 850AC7BE C03E202F8437

Cipher Text: ECB54739A1832EC5
```

Figure 8: Java output for encryption from DES.java

2. Use the Java verification tool to help you with verifying the correct operation within ModelSim. There is also a decent online DES calculator available at `https://emvlab.org/descalc/` that shows a simplified input/output value from either encryption or decryption.

3. Implement a switch that indicates ECB or CBC modes and processes everything accordingly.

4. Test at least 10 random messages (i.e., plaintext) using 2 random keys for both encryption and decryption.

5. After verifying your design with a testbench in ModelSim, implement your design on the DSDB board and use the 7-segment display to display your plaintext and ciphertext. Since you only have four 7-segment displays, you will not be able to show the entire plaintext, ciphertext or key, so you will have to figure a way to verify the operation.

6. You should also design an option that displays a LED if the key is correct (i.e., that parity is correct or that it is odd). Your hardware should work regardless of parity because it does not use these bits when computing the subkeys, but a LED should be lit up if the key is bad - i.e., it does not have odd parity. The java code comes with a method to help check whether the parity is odd to help you validate the parity.

7. Use the push buttons, switches, and LEDs to help you input your plaintext as well as debug operation and prove that your design works on your DSDB board.

8. You should also analzye the PPA impact on your design.

This laboratory should involve **only combinational logic** and be straight forward in creating Boolean logic with SystemVerilog. Again, there are many parts to this design and based on experience, I believe it will be easier to debug the key generation first and then once this works, debug the encryption/decryption next. The key generation is slightly easier than operations like the Feistel block, so it will optimize your design process if you focus on this block first. However, I would use the strategy that works the best for you.

3.1 Testing and Stubbing Code

You should use the testbenches you utilized for Lab0 and Lab1 to help you test your design. The design is completely combinational and should not be any different in terms of structure than both of these labs. To get full credit, you should demonstrate that your design works for both encryption and decryption by testing at least 10 plaintext messages using at least 2 different keys. This is basically testing 20 vectors - the more vectors tested and the methodology you use could possibly earn you extra credit on this laboratory.

I have also given you some freebies to help you with this lab. When writing HDL or software, it is sometimes useful to *stub* your code. A stubbed piece of code is a blank piece of software that has most of your functions you believe will work for your design. Fortunately, I have stubbed out your SV for you and you can use this as a guide. I also put some comments in the SV to help you know where to instantiate certain items. Inside the SV, I have also included the complete S-boxes which are the substitution boxes you will use for this laboratory. All of the S-boxes work by giving them 6-bits and they produce 4-bits as an output, as indicated previously.

I have also utilized a more advanced testbench that reads your key, plaintext, and ciphertext from a file. These are included in the `des.tv` files and 4 examples are given. The testbench should read in the values on each edge of the clock as in Lab 1. Although this testbench outputs data to a file, you will find more information can be found through debugging in ModelSim as documented in the next subsection.

3.2 Getting to know ModelSim and Debugging more in depth

ModelSim is a professional Hardware Descriptive Language tool for simulation and verification. It has many neat features to help you with debugging. Although testbenches are the main vehicle for understanding how to test a digital system, using ModelSim can save you hours and days in debugging a design. Therefore, we are also going to introduce some new features of ModelSim that you should use to help you with this laboratory. I also encourage you to use the testbench skills you learned from Lab 1.

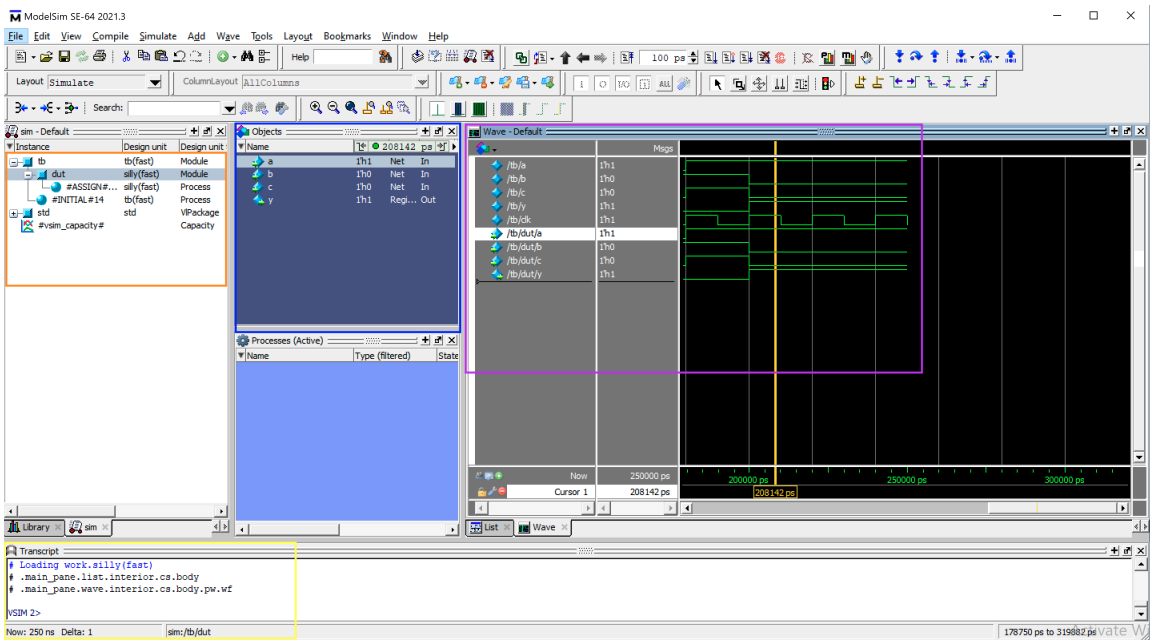

Figure 9: ModelSim Sim and Objects Window

The features you will use in ModelSim are the *Sim* and *Objects* window. Normally both of these windows are present when running a DO file, however, sometimes I find that they do not open properly. You may need to activate them in the View menu at the top of ModelSim. They should look like Figure 9 when activated. Both of these windows are utilized with the Wave window.

To use the two windows effectively, you should use the *Wave* to see the data at a certain time. First, move your cursor to the time you wish to investigate something - you should see a yellow line indicating the time you are observing the data. Next, you should navigate to the hierarchy of the module you wish to verify in the *Sim* window and the *Objects* window will display all signals and values that for that instance at a given time. You might need to play around with using these two windows together with the *Wave* window, but once you do you will find that its easy to debug what each block is producing at a given time.

The "sim" window (orange) contains the hierarchy of the design. The top level shows the test bench (tb) with a expandable button to the left. By clicking the "+" it opens the hierarchy for all modules instantiated in tb. Clicking on the name of the instance changes which objects (blue) are visible in the "objects" window. You can also add an object to the wave by right clicking on the name of the object in the "objects" window "Add Wave". Your testbench and modules may use different names but the same process applies to add signals to the wave (purple).

You can save the wave by clicking in the wave window then clicking the brown colored floppy disk icon in the toolbar. (Third icon from the left) The saved file only contains the configuration of the wave not the actual data. This allows you to recall the wave if you restart modelsim at a later time. To recall the wave you can type "do <name of wave file> in the transcript (yellow). You can also add this to the do file so it always pulls up your wave every time the simulation is run.

Modelsim has many extra features which can greatly aid in your debugging. First let's discuss some tips and tricks.

- If the toolbar gets disorderly, right click in the toolbar and select reset.
- Signals in the wave by default show the full path name. This can be changed to just the lowest level of hierarchy by clicking the "toggle leafs name" button in the lower left of the wave shown in Figure 10
- Zoom buttons are confusing. The "+" zoom in is mostly useless. Use the yellow upside down "T" with magnifying glass to zoom in at the cursor, as shown in Figure 11 in red.

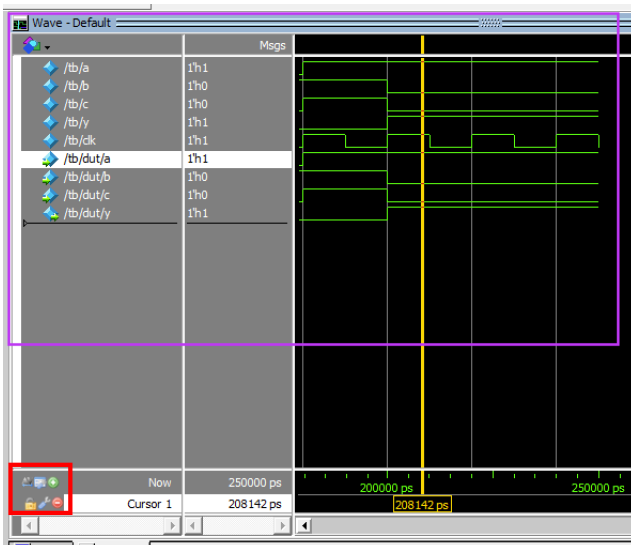

Figure 10: ModelSim toggle leafs. In the red box, the left-most box is the "Now" row.

Figure 11: Search inside the green box and zoom controls in the red box.

- The "-" zoom button works as expected.
- If you select a signal in the wave viewer, "Tab" and "Shift + Tab" will move the cursor to the next transition.
- A multibit bus can be search for a specific value using the "Search" buttons in the toolbar. The blue left and right allows to the right of the "Search" button will search backwards (left) or forwards (right) in time, as shown in Figure 11 in green.

At the risk of complicating things the "data flow" window can be very helpful when debugging red X's. Either in the objects window or the wave window right click a signal and select "Add to dataflow". This opens a new window where you can right click and select "ChaseX" or "TraceX". These allow you to quickly find the source of an X. If this does not make sense you can skip.

3.3 Extra Credit

If you get done early, you can attempt some extra credit. However, I would only try this option if you get everything verified within your design. One possible improvement is to work on optimizing the verification of your design. You can do any other modification (e.g., re-writing the Java code) or the DES implementation in Java, as well.

You also have the opportunity to enhance the DES and creating 3DES which is sometimes called triple DES. Triple DES is still used in some application and even used for some Microsoft keys that were used when purchasing software; however, I think they stopped using these several years ago. Interestingly, four out of 2^{56} possible keys are called weak keys. A weak key is the one that, after parity removal operation, consists either of all 0s, all 1s, or half 0s and half 1s. These keys are shown in Table 9 and I encourage you to try these keys in your hardware but do not use them in real life ;).

Keys with parity (64 bits)	Actual key (56 bits)
0x0101_0101_0101_0101	0x00_0000_0000_0000
0x1F1F_1F1F_0E0E_0E0E	0x00_0000_0FFF_FFFF
0xE0E0_E0E0_F1F1_F1F1	0xFF_FFFF_F000_0000
0xFEFE_FEFE_FEFE_FEFE	0xFF_FFFF_FFFF_FFFF

Table 9: DES Weak Keys

4. Submission

You should electronically hand in your HDL (all files that you want us to see) into Canvas. You should also take a printout of your waveform from your ModelSim simulation. Only one of your team members should upload the files and/or lab report. Please contact James Stine (james.stine@okstate.edu) for more help. Your code should be readable and well-documented. In addition, please turn in additional test cases or any other added item that you used. Please also remember to document everything in your Lab Report using the information found in the Grading Rubric.

References

[1] Sarah Harris and David Harris, *Digital Design and Computer Architecture: RISC-V Edition*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2021.

[2] National Institute of Standards and Technology, "Data Encryption Standard (DES)," FIPS Publication 46-3, October 1999.

[3] Alex Biryukov and Christophe De Cannière, *Data Encryption Standard (DES)*, pp. 129–135, Springer US, Boston, MA, 2005.

[4] Christof Paar and Jan Pelzl, *Understanding Cryptography: A Textbook for Students and Practitioners*, Springer Publishing Company, Incorporated, 1st edition, 2009.

[5] William Stallings, *Cryptography and Network Security: Principles and Practice*, Prentice Hall Press, USA, 6th edition, 2013.

[6] Stephen M. Trimberger, "Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 318–331, 2015.