Lab 2: Hardware-based Simplified Data Encryption Standard (S-DES)

Assigned: Tuesday 9/28; Due **Monday 10/15** (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 logic. Security is becoming a major item for all devices. It also essential to devices 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], 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 version of DES 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.

The DES uses a symmetric-key algorithm for the encryption of data. Symmetric cryptography just means that the keys can be used both for encryption and decryption. Although symmetric-key algorithms can use the keys both ways, they sometimes are different using some simple transformations between the two. The original DES uses a 56-bit key and a 64-bit block size. It also computes the cipher or translated message in 16 rounds. Although we could easily create this logic within our Field Programmable Gate Array (FPGA), it would take a significant amount of work to "crack" or break the message given the time we have in this class. So, we will use something called a Simplified DES (S-DES) algorithm which is almost identical to DES albeit smaller in size [3]. The S-DES encryption algorithm takes an 8-bit block of plaintext (e.g., `1011_1101`) and a 10-bit key as input and produces an 8-bit block of ciphertext as output. Similarly, the S-DES decryption algorithm takes an 8-bit block of ciphertext and the **same** 10-bit key used to produce that ciphertext as input and produces the original 8-bit block of plaintext.

The S-DES encryption algorithm involves five functions: an initial permutation (IP); a complex function labeled f_K, which involves both permutation and substitution operations and depends on a key input; a simple permutation function that switches (SW) the two halves of the data; the function f_K again; and finally a permutation function that is the inverse of the initial permutation (IP^{-1}). Although this sounds complicated, it is just simple blocks where the inputs are used to produce outputs (all in bits).

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

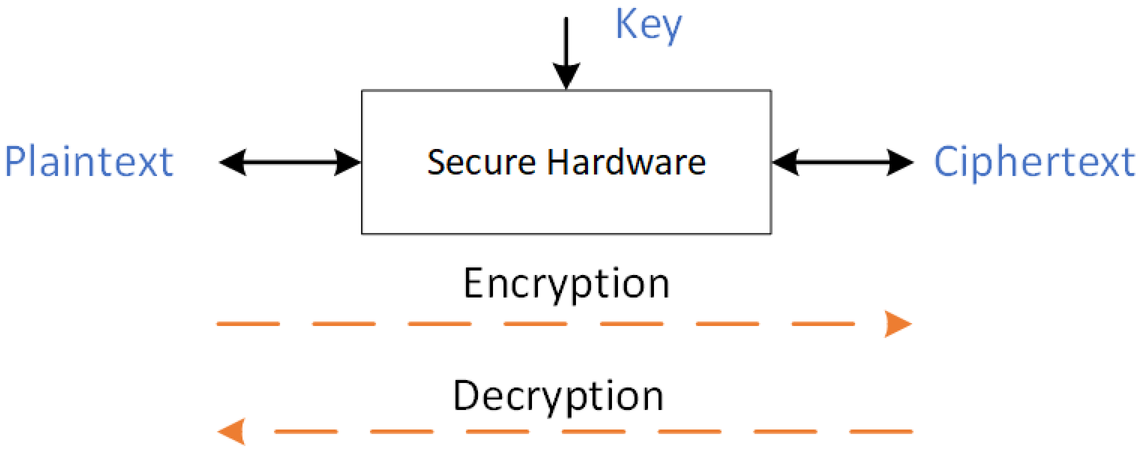


Figure 1: Basic Symmetric Cryptographic Hardware Block Diagram

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.

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 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. Simplified DES

Simplified 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 this great text [4].

The symmetric-based cryptographic algorithm of Simplified or S-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 key generation (i.e., K_1 and K_2) and the other for the basic symmetric cryptographic algorithm.

2.1 Key Generation

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

1. P10 : 10-bit permutation
2. Rotation

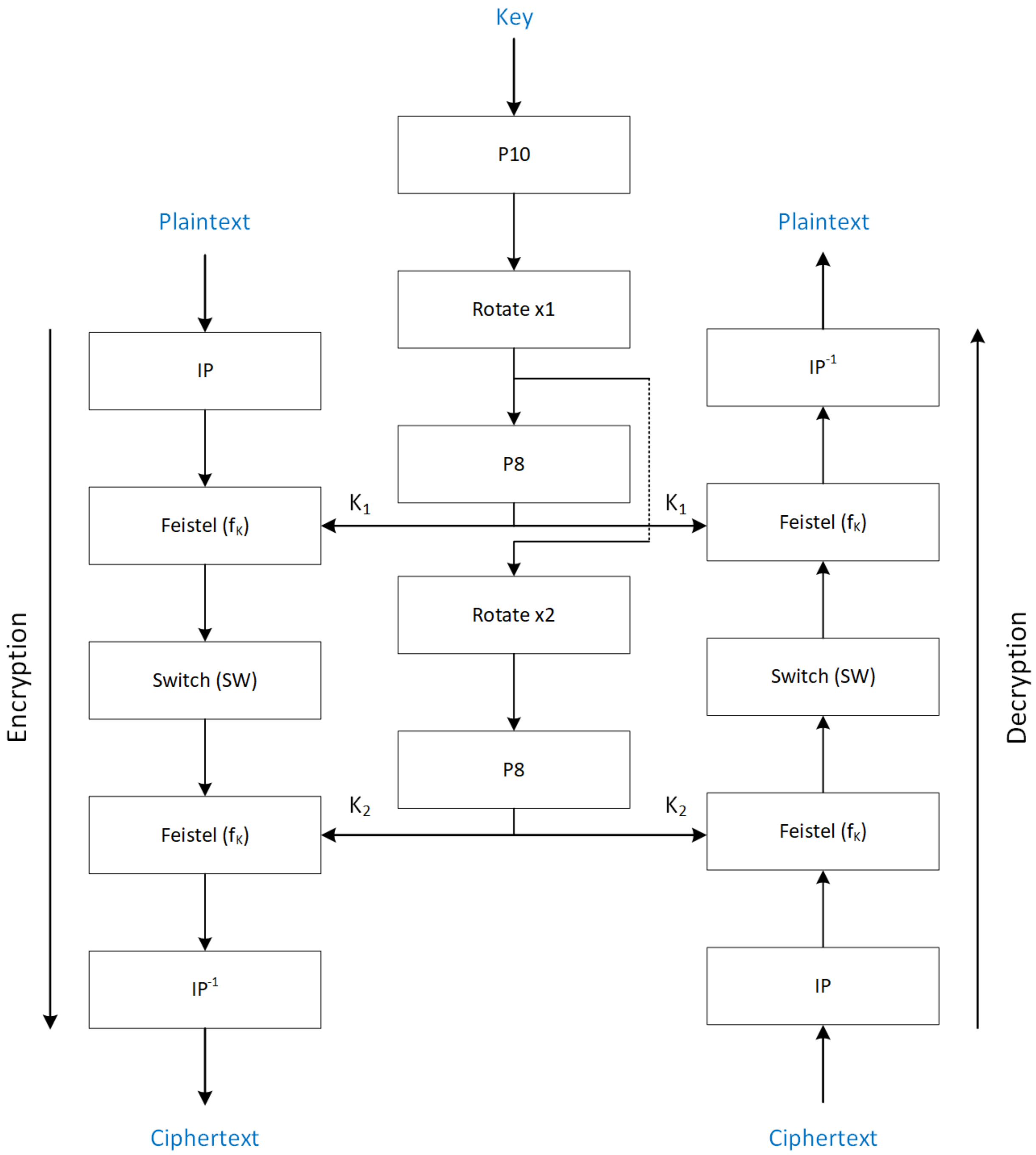


Figure 2: S-DES Block Diagram

3. P8 : 8-bit permutation

Each block produces two keys: K_1 and K_2 that get used during encryption and decryption. More advanced algorithms, for example for those found in [4], use the same option but typically have more keys to produce more security.

To understand the permuations, let's start with a vector called `A[9:0]`. Therefore, P10 is the following:

```
{A[7], A[5], A[8], A[3], A[6], A[0], A[9], A[1], A[2], A[4]}
```

And, P8 is the following:

```
{A[4], A[7], A[3], A[6], A[2], A[5], A[0], A[1]}
```

It is important to note that P8 actually does not use all of the 10-bit vector but only utilizes part of the vector to produce an 8-bit output. Again, it is not important to understand why these permutations are done this way, but just that you need to follow the procedure correctly. Later on, we will discuss how to verify each section is working correctly.

Rotation is performed as explained earlier except that the first block is rotated once and the second block rotates twice. The only caveat here is that **each rotation** should be done by separating each 10-bit block into 5-bits or broken into two groups. 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`.

2.2 S-DES encryption or decryption

As stated earlier, S-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})

Assuming that the input to this sequence is `PT[7:0]` The Initial Permutation (IP) block is easily created by:

```
IP = {PT[6], PT[2], PT[5], PT[7], PT[4], PT[0], PT[3], PT[1]}
```

Swapping is just swapping each 8-bit blocks by 4-bits. For example, if the input is `FB[7:0]` the swap would be

```
SW = {FB[3:0], FB[7:4]}
```

Finally, the inverse permutation or IP^{-1} is created similarly to the IP block:

```
IP^{-1} = {SW[4], SW[7], SW[5], SW[3], SW[1], SW[6], SW[0], SW[2]}
```

2.2.1 Feistel Block

The main part of this section is called the Feistel block that is named after the German-American cryptographer Horst Feistel. Horst Feistel main work was in developing ciphers for 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 8-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 Feistel block correctly. The order of operations in the Feistel block are as follows:

1. Split the 8-bit IP output into two separate blocks of 4-bits: `F[3:0]` and `S[3:0]`.
2. Create the expansion/permutation (EP) operation by replicating the second block only as:

   ```
   EP[7:0] = {S[0], S[3], S[2], S[1], S[2], S[1], S[0], S[3]}
   ```
3. XOR the EP block with the key (K_i)
4. Take the output of the XOR block and go into two S-boxes where the first S-box uses `XOR_out[7:4]` and the second S-box uses `XOR_out[3:0]` producing `S0[1:0]` and `S1[1:0]`, respectively.
5. The next block in the Feistel block is called the P4 block or 4-bit permutation and it takes the output of the S-boxes and groups it into 4-bits : `P4_in[3:0] = S0[1:0], S1[1:0]` and produces the output

   ```
   P4_out[3:0] = {P4_in[2], P4_in[0], P4_in[1], P4_in[3]}
   ```
6. Finally, the P4 output is XORed with `F[3:0]` as `xor_out = P4_out[3:0] xor F[3:0]` and output as `xor_out`, `S[3:0]`.

Figure 3 shows the basic block diagram of what the Feistel algorithm is doing. Again, this does look like a lot of work but it is quite straightforward. Also, the Feistel block operates with two keys. Therefore, pay attention to Figure 2 for each key K_1 and K_2 and its use for either encryption or decryption. Please also note in Figure 2 does two Feistel computations, so you will have to use this block twice per operation (e.g., encryption).

2.3 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). Your top diagram should have an input that tells it whether to encrypt or decrypt your block.

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 by Professor Rob Beezer at the University of Puget Sound in Washington State. He has graciously allowed us to use this software for the project and it is great because it allows you to put vectors in and test them at each stage. I have slightly modified the User Interface (UI) shown in Figure 4 to have names that match the block diagrams on Figure 2. 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.

Verification is hard 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 SDES` at the command prompt.

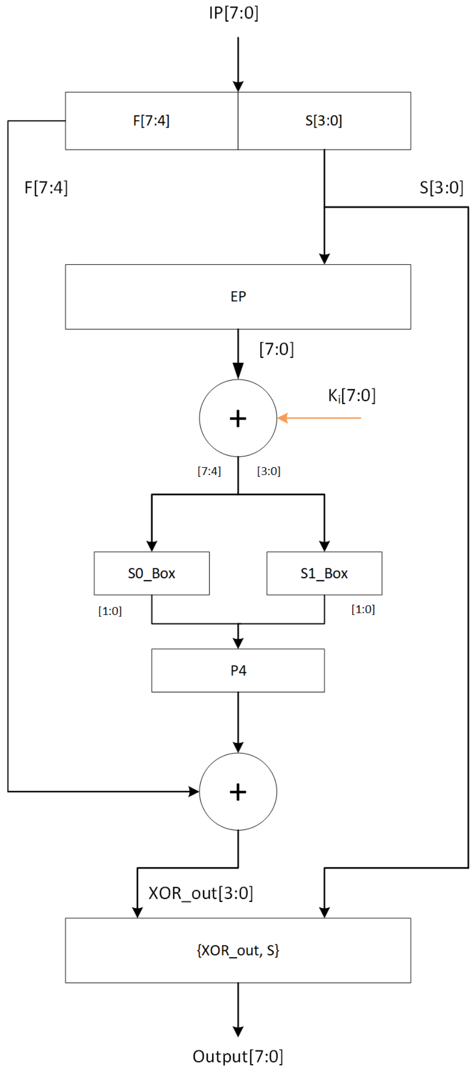


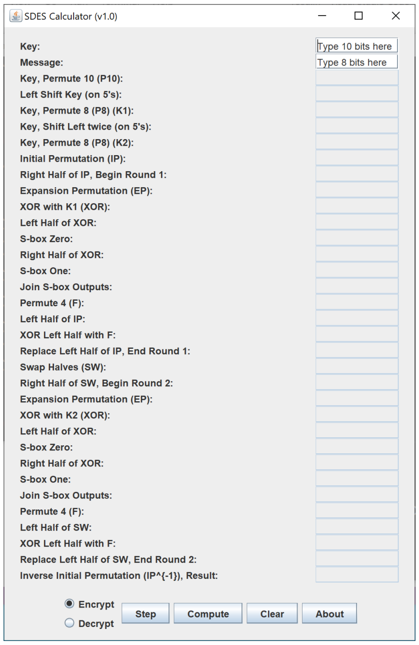
Figure 3: Feistel (f_K) Block

```
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.

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

1. Design the S-DES combinatianal block for both encryption and decryption in SystemVerilog and simulate with ModelSim.
2. Use the Java verification tool to help you with verifying the correct operation within ModelSim.
3. Test at least 10 random messages (i.e., plaintext) using 2 random keys for both encryption and decryption.
4. 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.
5. 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.



Key: Type 10 bits here
Message: Type 8 bits here
Key, Permute 10 (P10):
Left Shift Key (on 5's):
Key, Permute 8 (P8) (K1):
Key, Shift Left twice (on 5's):
Key, Permute 8 (P8) (K2):
Initial Permutation (IP):
Right Half of IP, Begin Round 1:
Expansion Permutation (EP):
XOR with K1 (XOR):
Left Half of XOR:
S-box Zero:
Right Half of XOR:
S-box One:
Join S-box Outputs:
Permute 4 (F):
Left Half of IP:
XOR Left Half with F:
Replace Left Half of IP, End Round 1:
Swap Halves (SW):
Right Half of SW, Begin Round 2:
Expansion Permutation (EP):
XOR with K2 (XOR):
Left Half of XOR:
S-box Zero:
Right Half of XOR:
S-box One:
Join S-box Outputs:
Permute 4 (F):
Left Half of SW:
XOR Left Half with F:
Replace Left Half of SW, End Round 2:
Inverse Initial Permutation (IP^{-1}), Result:

● Encrypt
○ Decrypt

Figure 4: Java Verification Tool Written by Professor Rob Beezer

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

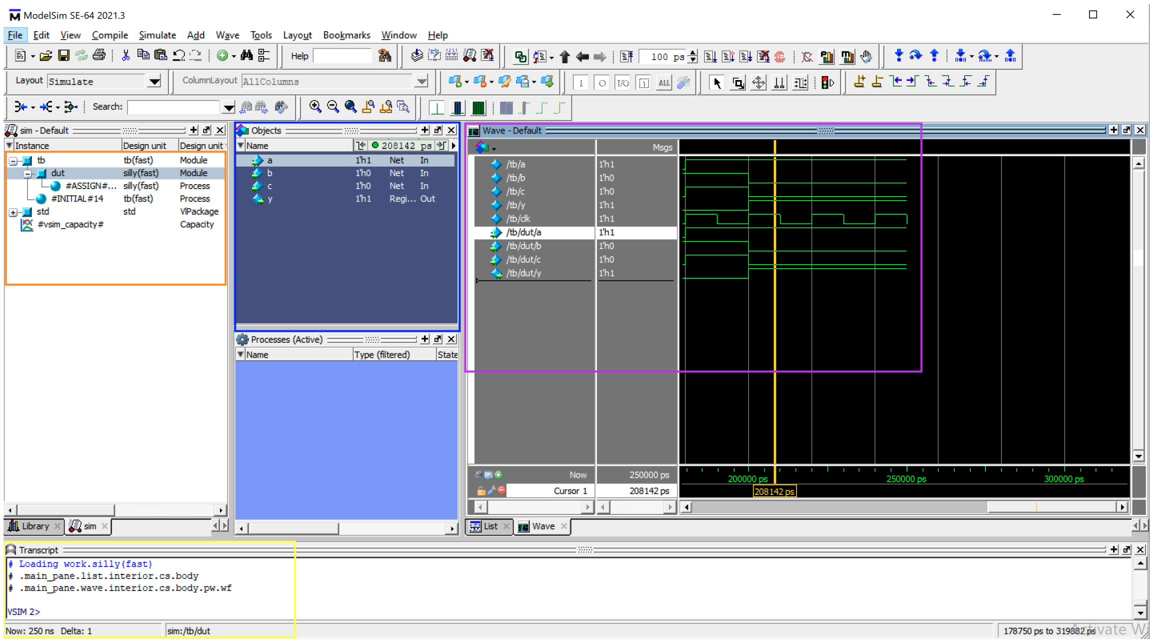
You should use the testbenches you utilized for Lab0 and Lab1 to help you test your design. The design is completely combinational and should not be any different in terms of structure than both of these labs. To get full credit, you should demonstrate that your design works for both encryption and decryption 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. Inside the SV, I have also included the `S0_box` and `S1_box` which are the two substitution boxes you will use for this laboratory. Both of the S-boxes work by giving them 4-bits and they produce 2-bits as indicated previously.

3.2 Getting to know ModelSim and Debugging

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.

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 5 when activated. Both of these windows are utilized with the Wave window.



Instance	Design unit	Design unit
tb	tb(fast)	Module
dut	silly(fast)	Module
#ASSIGN#...	silly(fast)	Process
#INITIAL#14	tb(fast)	Process
std	std	VlPackage
#vsim_capacity#		Capacity

Name	208142 ps		
a	1'h1	Net	In
b	1'h0	Net	In
c	1'h0	Net	In
y	1'h1	Regi...	Out

Name	Type (filtered)	State

	Msgs
/tb/a	1'h1
/tb/b	1'h0
/tb/c	1'h0
/tb/y	1'h1
/tb/clk	1'h1
/tb/dut/a	1'h1
/tb/dut/b	1'h0
/tb/dut/c	1'h0
/tb/dut/y	1'h1
Now	250000 ps
Cursor 1	208142 ps

```
# Loading work.silly(fast)
# .main_pane.list.interior.cs.body
# .main_pane.wave.interior.cs.body.pw.wf

VSIM 2>
```

Now: 250 ns Delta: 1 sim:/tb/dut 178750 ps to 319882 ps

Figure 5: ModelSim Sim and Objects Window

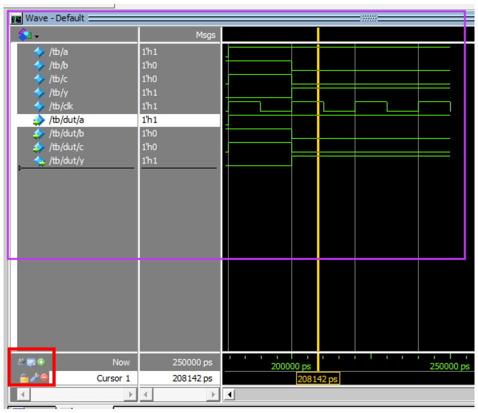
To use the two windows effectively, you should use the *Wave* to see the data at a certain time. First, move your cursor to the time you wish to investigate something - you should see a yellow line indicating the time you are observing the data. Next, you should navigate to the hierarchy of the module 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.

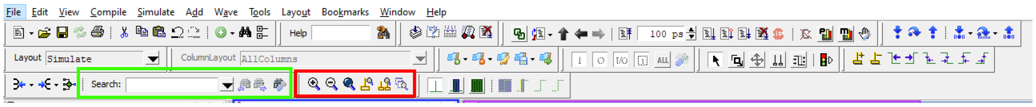
- If the toolbar gets disorderly, right click in the toolbar and select reset.
- Signals in the wave by default show the full path name. This can be changed to just the lowest level of hierarchy by clicking the “toggle leafs name” button in the lower left of the wave shown in Figure 6
- Zoom buttons 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 7 in red.
- 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.



/tb/a 1'h1
/tb/b 1'h0
/tb/c 1'h0
/tb/y 1'h1
/tb/clk 1'h1
/tb/dut/a 1'h1
/tb/dut/b 1'h0
/tb/dut/c 1'h0
/tb/dut/y 1'h1

Now 250000 ps
Cursor 1 208142 ps

Figure 6: ModelSim toggle leafs. In the red box, the left-most box is the “Now” row.


File Edit View Compile Simulate Add Wave Tools Layout Bookmarks Window Help
Help
Search:

100 ps

Layout Simulate

ColumnLayout AllColumns

Figure 7: Search inside the green box and zoom controls in the red box.

- 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 7 in green.

At the risk of complicating things the “data flow” window can be very helpful when debuging 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 option is to adapt the design to perform the complete DES - this is a big undertaking and only for serious students who find this topic very interesting and have extra time.

Another possible improvement is to work on optimizing the verification of your design. I have included a Java program that I wrote that outputs multiple values for the S-DES implementation in Java. You could use these vectors to self-verify your implementation as in Lab 1.

Yet another piece of extra credit is analyzing how fast you can run DES through. This will involve using Vivado to analyze how fast your design can be and doing some calculations on how fast you can encrypt and decrypt your data.

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: ARM Edition*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2015.

[2] National Institute of Standards and Technology, “Data Encryption Standard (DES),” 1999.

[3] Edward F Schaefer, “A simplified Data Encryption Standard algorithm,” *Cryptologia*, vol. 20, no. 1, pp. 77–84, 1996.

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