Lab 2: Complex Combinational Logic and Debugging : Hardware-based Secure Hash Algorithm

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 hardware-based Secure Hash Algorith (SHA) implementation in two parts. The primary part of this laboratory will involve designing the SHA algorithm found in this laboratory. 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) function that is commonly used in bitcoin and web-based authentication.

The most widely used cryptographic operations are encryption and decryption for secrecy, hashing for integrity, and signatures for authenticity. Pure software implementations are slow, power-hungry, and vulnerable to timing attacks that can be exploited remotely. Modern instruction sets provide dedicated cryptography instructions that are faster, simpler, and provide better performance than pure software implementations. Moreover, having cryptographic instructions promotes standardized software and reduced code size, which helps reduce the risk of inadvertent security flaws.

The Secure Hash Algorithm 2 (SHA-2) is the hash function used in most internet protocols, such as TLS, SSL, PGP, etc., as well as to verify transactions in Bitcoin and other cryptocurrencies. It was designed by the US National Security Agency (NSA) and was first published in 2001. It is now a standard maintained by the National Institute of Standards and Technology [**?**]. SHA-2 generates 224-, 256-, 384-, or 512-bit message digests, replacing the SHA-1, MD4, and MD5 algorithms that produced shorter digests and are no longer considered secure. Other flavors of SHA-2 are similar but truncate the digest to fewer bits after it is computed, trading compactness for reduced security.

1.1 Security Basics

Cryptography is the science of hiding the meaning of messages. Although it has gained interest in recent decades for computer security, it has been around at least since Julius Caeser wrote B in place of A to prevent the Gauls from reading his messages to his generals. For our field, Claude Shannon (1916-2001) originally thought of applying these ideas related to software and hardare in terms of their confusion and diffusion [4] and later expanded this into his communication theory of secrecy systems [5]. Cryptography uses many primatives, including symmetric ciphers, asymmetric (also called public key) ciphers, hash functions, and cryptographic protocols.

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*. Symmetric ciphers use the same secret key for both encryption and decryption. It is an efficient way to encrypt bulk data. These 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. Figure 1 shows a symmetric-key encryption system that encrypts a plaintext message with a key to produce the ciphertext. In a good system, deducing the plaintext from the ciphertext without the key is impractically difficult.

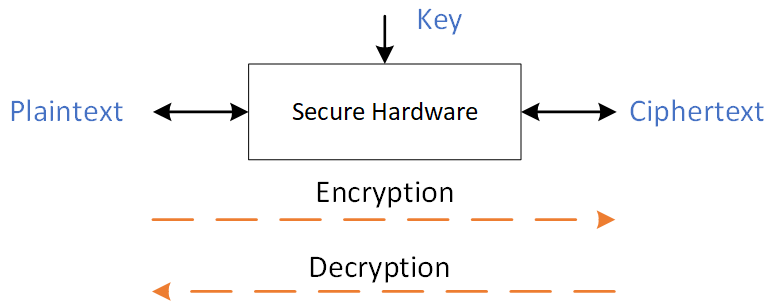

Figure 1: Basic Symmetric Cryptographic Hardware Block Diagram

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.

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. Hash Functions

Cryptographic hash functions are important elements of cryptography. They transform a variable-length message into a short numerical fingerprint called a message digest. Hashes are used to verify data integrity and as a building block for digital signatures. Any alteration to a message will corrupt the message digest. Passwords are usually stored in hashed form instead of plaintext, so stealing the password file will not reveal the password itself. A good hash function has several properties:

- Avalanche Effect - Any change in the message will, with very high probability, change many bits of the digest
- Pre-Image Resistant - Given a message digest, it is computationally infeasible to find the original message.
- Collision Resistant - Given a message digest, it is computationally infeasible to find another message that produces the same digest
- Fast and easy to compute

One of the most commmon uses of hash functions are in use with our use of the GitHub repository which is a SHA-1 digest. The hash function is used to differentiate which modification is made for a given repository. This can be seen in Figure 2. But to make these hashes or ids easier to handle it also supports

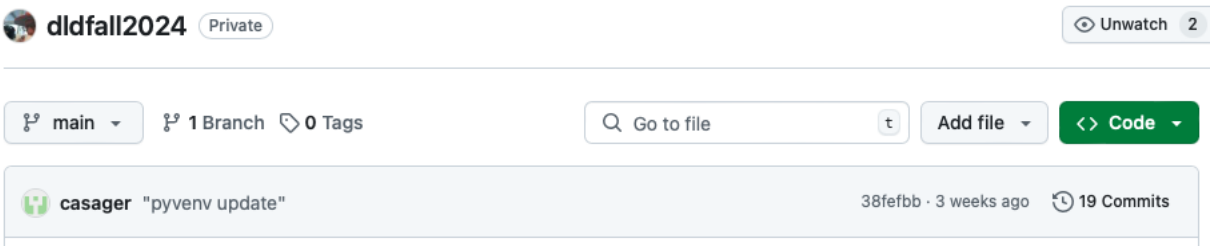

Figure 2: Example GitHub repository showing Hash of `0x38fefbb`

SHA-2 Algorithm	Variable	SHA-256	SHA-512
Message Digest Size (bits)	d	256	512
Block Size (bits)	m	512	1024
Word Size (bits)	w	32	64
Rounds	r	64	80
$\Sigma_0^d(x)$		$(x\ ror\ 2) \oplus (x\ ror\ 13) \oplus (x\ ror\ 22)$	$(x\ ror\ 28) \oplus (x\ ror\ 34) \oplus (x\ ror\ 39)$
$\Sigma_1^d(x)$		$(x\ ror\ 6) \oplus (x\ ror\ 11) \oplus (x\ ror\ 25)$	$(x\ ror\ 14) \oplus (x\ ror\ 18) \oplus (x\ ror\ 41)$
$\sigma_0^d(x)$		$(x\ ror\ 7) \oplus (x\ ror\ 18) \oplus (x >> 3)$	$(x\ ror\ 1) \oplus (x\ ror\ 8) \oplus (x >> 7)$
$\sigma_1^d(x)$		$(x\ ror\ 17) \oplus (x\ ror\ 19) \oplus (x >> 10)$	$(x\ ror\ 19) \oplus (x\ ror\ 61) \oplus (x >> 6)$

Table 1: SHA-2 structure and sigma operations

using a short version of the id. The short commit id can actually be any number of characters as long as it's unique for a commit within the same repo. To conserve space, GitHub actually shortens the hash even though its 40 characters or 160-bits in length (i.e., `38fefbbd46d62f394949b0448707c4f24cb60a3a`).

For this laboratory, we will implement SHA-256 which is the most popular form that people are most familiar with. Most Linux distributions come with programs to compute the SHA-256 hash function to verify data integrity. For example, the following produces the hash for "Hello World!" using SHA-256: echo -n "Hello World!" | sha256sum. If somebody changed the exclamation point to a question mark, sha256sum would give a different hash, revealing that the message had been corrupted.

```
7f83b1657ff1fc53b92dc18148a1d65dfc2d4b1fa3d677284addd200126d9069  -
```

You can also interactively watch SHA-256 run at `https://sha256algorithm.com`.

Table summarizes the structure of SHA-256 and SHA-512. The hash operates on a message M comprising N m-bit blocks; it is padded if necessary to be an integral number of blocks. Each block is formed from 16 w-bit words. Word j of block i is denoted M_j^i. The message digest (also called the hash) H is formed from 8 w-bit words. Each block goes through r rounds of hashing, which involve applying some shifts, rotates, and logical and addition operations. The hashing steps involve sigma (Σ/σ) functions that are expressed in terms of right rotations (ror) and right shifts (») of the words. The hash is initialized with 8 w-bit constants H_j^0 and uses r w-bit round constants K_t tabulated in the SHA-2 specification.

2.1 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
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 3. 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 [7]. 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 4 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 4, 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.

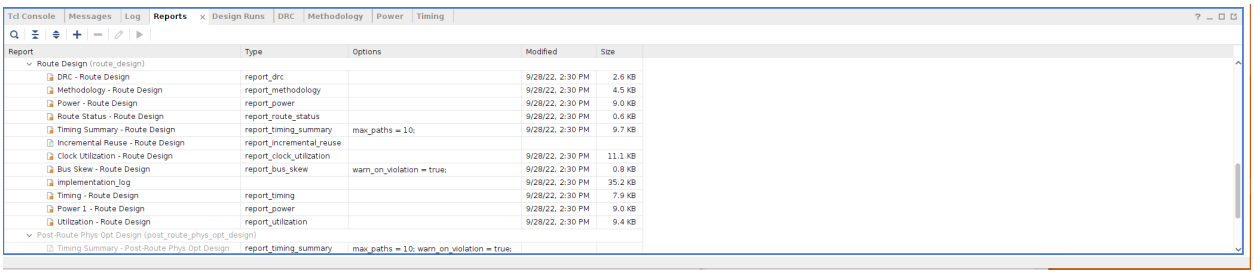

Figure 3: Reports Window within Xilinx Vivado

```
Original plain Text:    2579DB866C0F528C
Key:                    433E4529462A4A62
IV (for CBC mode):      0000000000000000

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 4: 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.

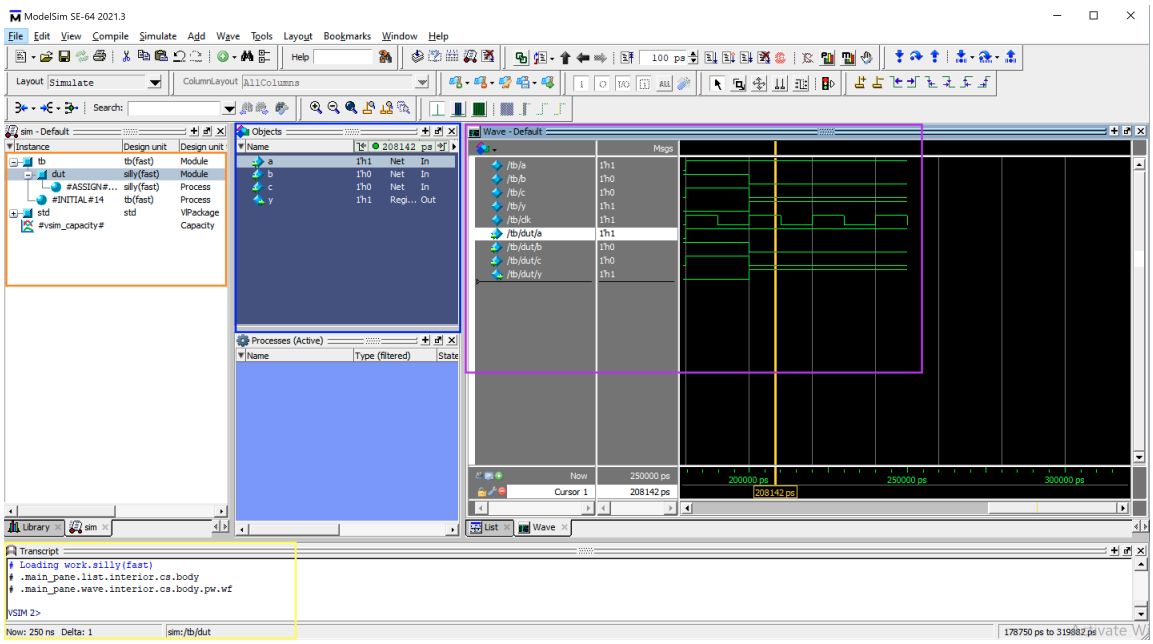

Figure 5: ModelSim Sim and Objects Window

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 5 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 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.

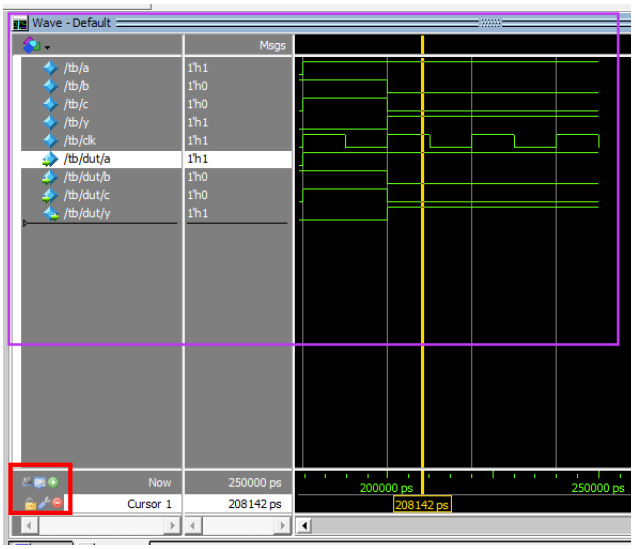

Figure 6: ModelSim toggle leafs. In the red box, the left-most box is the "Now" row.

Figure 7: 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 7 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 2 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 2: 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] C. E. Shannon, "A mathematical theory of cryptography," Classified report, Bell Laboratories, Murray Hill, NJ, USA, Sept. 1945.

[5] C. E. Shannon, "Communication theory of secrecy systems," *The Bell System Technical Journal*, vol. 28, no. 4, pp. 656–715, 1949.

[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.

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