

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 Algorithm (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 [4]. 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 Caesar 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 hardware in terms of their confusion and diffusion [5] and later expanded this into his communication theory of secrecy systems [6]. Cryptography uses many primitives, 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.

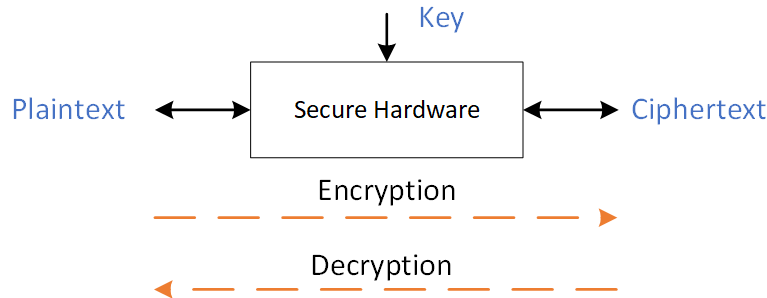


Figure 1: Basic Symmetric Cryptographic Hardware Block Diagram

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 in 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 common 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

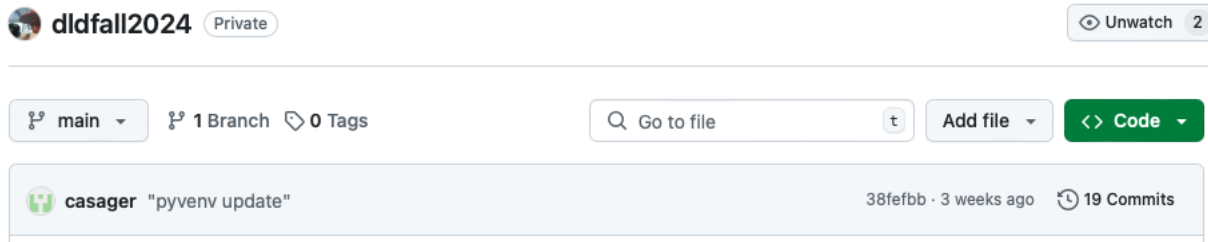


Figure 2: Example GitHub repository showing Hash of 0x38fefbb

SHA-2 Algorithm	Var	SHA-256	SHA-512
Msg Digest Size (bits/bytes)	d	256/64	512/128
Block Size (bits/bytes)	m	512/128	1024/256
Word Size (bits/bytes)	w	32/4	64/16
Rounds	r	64	80
$\Sigma_0^d(x)$		$(x \text{ ror } 2) \wedge (x \text{ ror } 13) \wedge (x \text{ ror } 22)$	$(x \text{ ror } 28) \wedge (x \text{ ror } 34) \wedge (x \text{ ror } 39)$
$\Sigma_1^d(x)$		$(x \text{ ror } 6) \wedge (x \text{ ror } 11) \wedge (x \text{ ror } 25)$	$(x \text{ ror } 14) \wedge (x \text{ ror } 18) \wedge (x \text{ ror } 41)$
$\sigma_0^d(x)$		$(x \text{ ror } 7) \wedge (x \text{ ror } 18) \wedge (x \gg 3)$	$(x \text{ ror } 1) \wedge (x \text{ ror } 8) \wedge (x \gg 7)$
$\sigma_1^d(x)$		$(x \text{ ror } 17) \wedge (x \text{ ror } 19) \wedge (x \gg 10)$	$(x \text{ ror } 19) \wedge (x \text{ ror } 61) \wedge (x \gg 6)$

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.

```
7f83b1657ff1fc53b92dc18148a1d65dfc2d4b1fa3d677284aadd200126d9069 -
```

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.

The Secure Hash Algorithm [4] is one of the most widely utilized message digest functions. We will be implementing this hardware cryptographic system in hardware on the FPGA. It will involve two basic steps:

1. Proprocessing - This sounds exactly what it sounds like; that is, the message x has to be padded to fit a size of a multiple of 512 bits.
2. Hash Computation - Each message block x_i is processed in four stages with 64 rounds each as shown in Figure 3. For this laboratory, we will handle this combinationally. Later in the semester, we will explore ways we can do this with sequential logic too.

2.1 ASCII

ASCII stands for American Standard Code for Information Interchange. Computers can only understand numbers, so an ASCII code is the numerical representation of a character such as 'a' or '@' or an action of

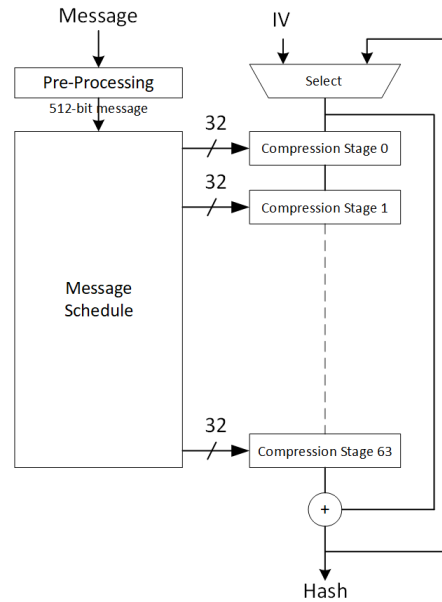


Figure 3: SHA-256 Block Diagram

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
048	0x30	0	064	0x40	@	080	0x50	P	096	0x60	'	112	0x70	p
049	0x31	1	065	0x41	A	081	0x51	Q	097	0x61	a	113	0x71	q
050	0x32	2	066	0x42	B	082	0x52	R	098	0x62	b	114	0x72	r
051	0x33	3	067	0x43	C	083	0x53	S	099	0x63	c	115	0x73	s
052	0x34	4	068	0x44	D	084	0x54	T	100	0x64	d	116	0x74	t
053	0x35	5	069	0x45	E	085	0x55	U	101	0x65	e	117	0x75	u
054	0x36	6	070	0x46	F	086	0x56	V	102	0x66	f	118	0x76	v
055	0x37	7	071	0x47	G	087	0x57	W	103	0x67	g	119	0x77	w
056	0x38	8	072	0x48	H	088	0x58	X	104	0x68	h	120	0x78	x
057	0x39	9	073	0x49	I	089	0x59	Y	105	0x69	i	121	0x79	y
058	0x3A	:	074	0x4A	J	090	0x5A	Z	106	0x6A	j	122	0x7A	z
059	0x3B	;	075	0x4B	K	091	0x5B	[107	0x6B	k	123	0x7B	{
060	0x3C	<	076	0x4C	L	092	0x5C]	108	0x6C	l	124	0x7C	
061	0x3D	=	077	0x4D	M	093	0x5D	^	109	0x6D	m	125	0x7D	}
062	0x3E	>	078	0x4E	N	094	0x5E	~	110	0x6E	n	126	0x7E	~
063	0x3F	?	079	0x4F	O	095	0x5F	_	111	0x6F	o	127	0x7F	△

Table 2: Common English ASCII Characters

some sort. ASCII was developed a long time ago and now the non-printing characters are rarely used for their original purpose. Table 2 is an abbreviated ASCII character table. ASCII was actually designed for use with teletypes, so the descriptions are somewhat obscure. ASCII has expanded for other uses, however, its use is still utilized as it is easy to use for the english alphabet (sometimes being called US-ASCII). ASCII has just 128 code points, of which only 95 are printable characters, which severely limit its scope. However, this has been expanded to handle other alphabets. More elaborate ASCII tables can be found here: <https://en.wikipedia.org/wiki/ASCII>.

You will need to convert your text that you want to create a hash for with ASCII codes. You can either look it up manually using Table 2 or use the included Python file to convert it automatically. Use these hex values to include in your SV manual that will generate the hash value.

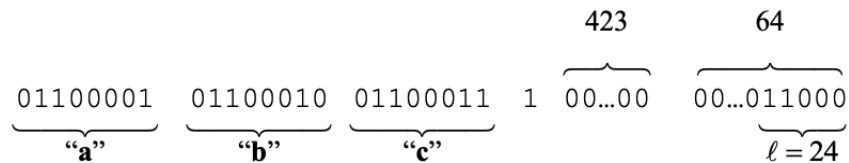


Figure 4: Padding example for “abc” for a 512-bit padded message [4]

2.2 Preprocessing

Before the actual hash computation, the message x has to be padded to fit a size of a multiple of 512 bit. For the internal processing, the padded message must then be divided into blocks. This will be dependent on the size of the message you plan on sending (e.g., SHA-256).

For example, Assume that we have a message x with a length of l bits. To obtain an overall message size of a multiple of 512 bits, we append a single “1” followed by k zero bits and the binary 64-bit representation of l . Consequently, the number of required zeros k is given by

$$k = 512 - 64 - 1 - l$$

For example, the (8-bit ASCII) message “abc” has length $8 \times 3 = 24$, so the message is padded with a one bit, then $448 - (24 + 1) = 423$ zero bits, and then the message length (i.e., 64), to become the 512-bit padded message. This is illustrated in Figure 4

This can be done easily within SystemVerilog with bitswizzling as indicated in Section 1.2. An example of this could be the following SV Hardware Descriptive Language (HDL):

```
assign padded = {message, 1'b1, {zero_width{1'b0}}, {back_0_width{1'b0}}, MSG_SIZE};
```

2.3 Hash Computation

The real advantage to using digital logic is that much of the computation can be done in parallel. That is, when one piece of logic is being computed, another part can be done at the same time reducing the total amount of time for the computation. Software, on the other hand, typically is slower as it requires waiting for previous operations to complete before it can move forward.

2.3.1 Modular Addition

The computation of the hash function is a combination of rotations, shifts, and additions. Addition is tricky as it has to be done not to exceed the size of the addition (in this case, since it is a block of 32-bits, then it should not exceed 2^{32} or modulo 32. This is called modular arithmetic and can be summarized in the following equation.

$$|X + Y|_m = \begin{cases} X + Y, & \text{if } X + Y < m \\ X + Y - m, & \text{if } X + Y \geq m \end{cases}$$

Modular arithmetic is key to many cryptographic algorithm. Fortunately, for our implementation it is quite easy as we just have to drop off the MSB.

2.3.2 constants

Before hash computation begins for each of the secure hash algorithms, the initial hash value, H^0 , must be set. For SHA-256, the size and number of words in H^0 depends on the message digest size. These words are obtained by taking the first thirty-two bits of the fractional parts of the square roots of the first eight prime numbers to ensure the numbers are random. That is, the fractional parts of 2, 3, 5, 7, 11, 13, 17, 19.

These values are different for different versions of the Secure Hash Standard [4].

$$\begin{aligned}
 H_0^0 &= 0x6a09e667 \\
 H_1^0 &= 0xbb67ae85 \\
 H_2^0 &= 0x3c6ef372 \\
 H_3^0 &= 0xa54ff53a \\
 H_4^0 &= 0x510e527f \\
 H_5^0 &= 0x9b05688c \\
 H_6^0 &= 0x1f83d9ab \\
 H_7^0 &= 0x5be0cd19
 \end{aligned}$$

There are some additional constants utilized within the main hash computation. These are computed similar based on the cube roots of the first sixty-four prime numbers and labeled $K_0^{256}, K_1^{256}, \dots, K_{63}^{256}$.

```

428a2f98 71374491 b5c0fbcf e9b5dba5 3956c25b 59f111f1 923f82a4 ab1c5ed5
d807aa98 12835b01 243185be 550c7dc3 72be5d74 80deb1fe 9bdc06a7 c19bf174
e49b69c1 efbe4786 0fc19dc6 240ca1cc 2de92c6f 4a7484aa 5cb0a9dc 76f988da
983e5152 a831c66d b00327c8 bf597fc7 c6e00bf3 d5a79147 06ca6351 14292967
27b70a85 2e1b2138 4d2c6dfc 53380d13 650a7354 766a0abb 81c2c92e 92722c85
a2bfe8a1 a81a664b c24b8b70 c76c51a3 d192e819 d6990624 f40e3585 106aa070
19a4c116 1e376c08 2748774c 34b0bcb5 391c0cb3 4ed8aa4a 5b9cca4f 682e6fff3
748f82ee 78a5636f 84c87814 8cc70208 90befffa a4506ceb bef9a3f7 c67178f2

```

2.3.3 Main SHA-256 computation

SHA-256 in hardware is typically very easy if taken systematically. For this laboratory, we will only use 1 m -bit block or 1 512-bit block. In theory, hardware may be composed of multiple blocks.

SHA-256 can be used to hash a message, M , having a length of l bits, where $0 \leq l < 2^{64}$. The algorithm uses 1) a message schedule of sixty-four 32-bit words, 2) eight working variables of 32 bits labeled a through h , and 3) a hash value of eight 32-bit words. The final result of SHA-256 is a 256-bit message digest. The words of the message schedule are labeled W_0, W_1, \dots, W_{63} . The eight working variables are labeled a, b, c, d, e, f, g , and h . The words of the hash value are labeled $H_0^i, H_1^i, \dots, H_7^i$, which will hold the initial hash value, H^0 , replaced by each successive intermediate hash value (after each message block is processed), H^i , and ending with the final hash value, H^N . For cryptographic systems, this is sometimes called a round and its similar to

The operation looks more complicated than it is but it's just a series of computations in parallel. The key is to get the order of processing correct and, of course, the number of steps. A key piece of information to remember is that addition (+) is performed modulo 2^{32} as described in Section 2.3.1.

1. Processing: set the initial hash values H^0 as previously specified as well as the padding (see Section 2.2).
2. Prepare the message:
 - Since we are only operating on 1 group or $N = 1$, we can break the computation down into blocks of 32 for each part of the message and operate on the message for 64 rounds (i.e., $0 \leq t \leq 63$). That is, Each message block, M^1, M^2, \dots, M^{64} , is processed in order.
 - For blocks $16 \leq t \leq 63$, we need to compute $W_t = \sigma_1^{256}(W_{t-2}) + W_{t-7} + \sigma_0^{256}(W_{t-15}) + W_{t-16}$ where

$$\begin{aligned}
 \sigma_0^{512}(x) &= \text{ror}^7(x) \oplus \text{ror}^{18}(x) \oplus (x \gg 3) \\
 \sigma_1^{512}(x) &= \text{ror}^{17}(x) \oplus \text{ror}^{19}(x) \oplus (x \gg 10)
 \end{aligned}$$

- This can be summarized as follows:

$$W_t = \begin{cases} M_t^i, & (0 \leq t \leq 15) \\ \sigma_1^{256}(W_{t-2}) + W_{t-7} + \sigma_0^{256}(W_{t-15}) + W_{t-16}, & (16 \leq t \leq 63) \end{cases}$$

3. Initialize the seven working variables with the (i-1)st hash:

- $a = H_0^{i-1}$
- $b = H_1^{i-1}$
- $c = H_2^{i-1}$
- $d = H_3^{i-1}$
- $e = H_4^{i-1}$
- $f = H_5^{i-1}$
- $g = H_6^{i-1}$

4. Compute the following items for $t = 0 : 63$ using $T_1 = h + \Sigma_1^{256} + \text{Choice}(e, f, g) + K_t^{63} + W_t$ and $T_2 = \Sigma_0 + \text{Majority}(a, b, c)$. These equations are broken down as for Σ_1 , Σ_0 and $\text{Choice}(x, y, z) = (x \cdot y) \oplus (\bar{x} \cdot z)$ and $\text{Majority}(x, y, z) = (x \cdot y) \oplus (x \cdot z) \oplus (y \cdot z)$. The values of Σ_1^{256} and Σ_0^{256} are similar to the lower case versions above or except that it is only composed of xor and ror (i.e., no right shifts) or the following.

$$\begin{aligned} \Sigma_0^{512}(x) &= \text{ror}^2(x) \oplus \text{ror}^{13}(x) \oplus \text{ror}^{22}(x) \\ \Sigma_1^{512}(x) &= \text{ror}^6(x) \oplus \text{ror}^{11}(x) \oplus \text{ror}^{25}(x) \end{aligned}$$

- $h = g$
- $g = f$
- $f = e$
- $e = d + T_1$
- $d = c$
- $c = b$
- $b = a$
- $a = T_1 + T_2$

5. Add the working variables above to the current hash variable to the current variables

- $H_0^1 = a + H_0^0$
- $H_1^1 = b + H_1^0$
- $H_2^1 = c + H_2^0$
- $H_3^1 = d + H_3^0$
- $H_4^1 = e + H_4^0$
- $H_5^1 = f + H_5^0$
- $H_6^1 = g + H_6^0$
- $H_7^1 = h + H_7^0$

6. Finally, concatenate or squish all the values in the previous step together forming a 512-bit message (i.e., 32 hexadecimal digits).

$$H_0^1 || H_1^1 || H_2^1 || H_3^1 || H_4^1 || H_5^1 || H_6^1 || H_7^1$$

2.4 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 [7].

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 5. 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 [8]. 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 6 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 6, 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.

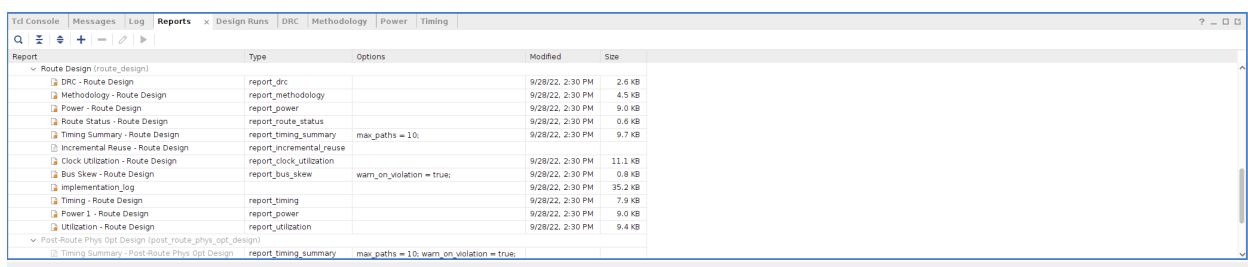


Figure 5: Reports Window within Xilinx Vivado

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 6: Java output for encryption from DES.java

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:

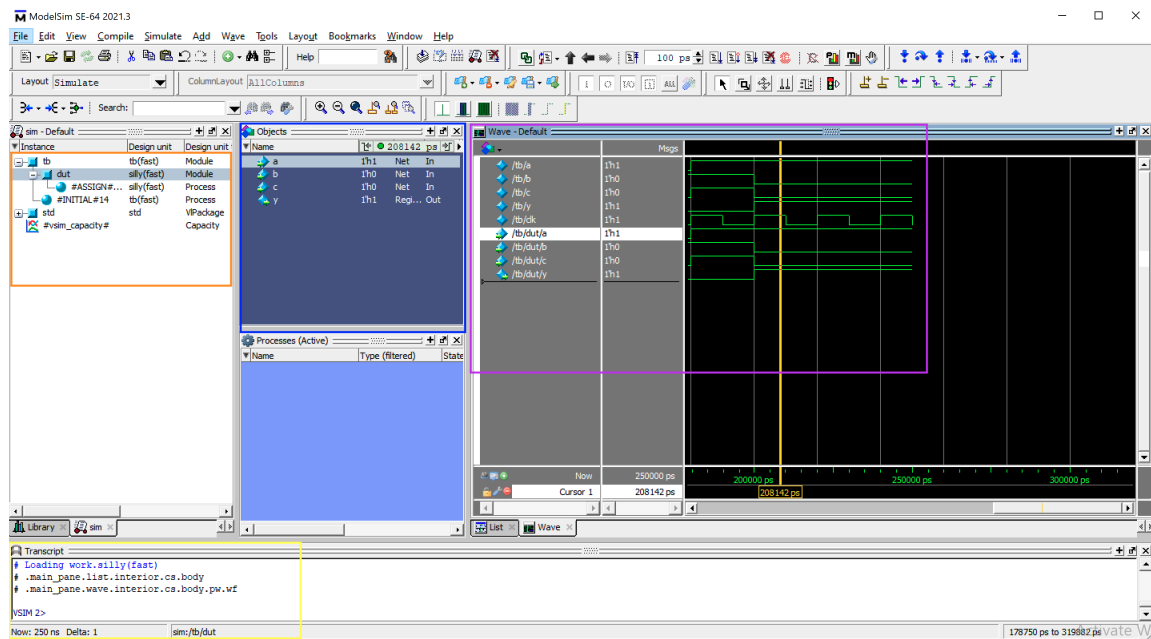
1. Design the DES combinational block for both encryption and decryption in SystemVerilog and simulate with ModelSim.
2. Use the Java verification tool to help you with verifying the correct operation within ModelSim. There is also a decent online DES calculator available at <https://emvlab.org/descalc/> 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 analyze 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



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.

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

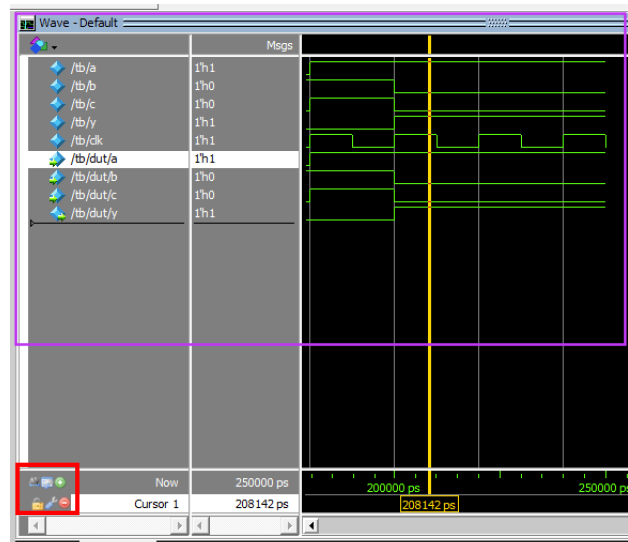


Figure 8: ModelSim toggle leafs. In the red box, the left-most box is the “Now” row.

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 8
- 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 9 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.
- A multibit bus can be search for a specific value using the “Search” buttons in the toolbar. The blue left and right arrows to the right of the “Search” button will search backwards (left) or forwards (right) in time, as shown in Figure 9 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.

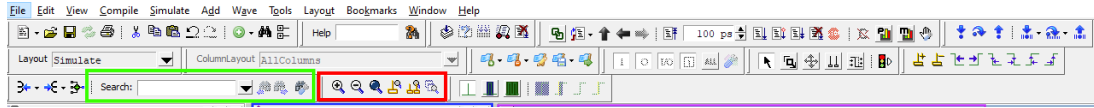


Figure 9: Search inside the green box and zoom controls in the red box.

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] National Institute of Standards, Technology (NIST), and Quynh Dang, “Secure hash standard,” 2015-08-04 00:08:00 2015.
- [5] C. E. Shannon, “A mathematical theory of cryptography,” Classified report, Bell Laboratories, Murray Hill, NJ, USA, Sept. 1945.
- [6] C. E. Shannon, “Communication theory of secrecy systems,” *The Bell System Technical Journal*, vol. 28, no. 4, pp. 656–715, 1949.
- [7] 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.
- [8] Christof Paar and Jan Pelzl, *Understanding Cryptography: A Textbook for Students and Practitioners*, Springer Publishing Company, Incorporated, 1st edition, 2009.