ECE 111 Final Project Report

SHA 256 Introduction

SHA 256 (Secure Hash Algorithm 256) is a cryptographic method in which input data up to 2^64 bits is converted into a fixed value output hash of 256 bits. This secure hashing algorithm is a one-way function, so the original input message can never be retrieved from the output hash value. This is due to the avalanche effect of the algorithm-- a small change in input significantly changes the fixed 256 bit digest. It is also practically impossible to find two different inputs that produce the same output hash. For these reasons, SHA 256 is commonly used to verify file integrity and authenticity. It is also used as the signature that links the Bitcoin blockchain, which will be discussed in the later section.

Algorithm

Preprocessing phase

a. Take message and convert it to binary and put them all into one string
b. Add 1 to the end of the message to separate the original message and the zero padding.
c. We pad the code with zeros to get the block to the correct size of 512 bits. If the message is bigger than 512 bits, then more blocks will be added. The block is then divided into 16 words of 32 bits each.
   The function below computes the # of blocks we need based on the number of words "size" is assigned.

```
function logic [15:0] determine_num_blocks(input logic [31:0] size);
    logic [31:0] test, blocks, temp;

        test=(size*32)/512;
        temp=(size*32)%512;

        if(temp)
            begin
                blocks=test+1; //more than 1 block is required
            end
        else
            begin
                blocks=test;
            end
    determine_num_blocks=blocks;
```

   i. `endfunction`
   ii. The variable "test" takes the size of the message and multiplies it by 32 because each word has 32 bits. We divide this number by 512 to determine the number of blocks we need. The variable "temp" is also used to determine the number of blocks in the case that we need more than one block. If the modulo of 512 is any number other than zero, then "block=test+1", otherwise "test" will be assigned as the number of blocks.

2. Hash computation
   a. We have to compute an additional 48 words to total up the number of words to 64 (16+48=64).
   b. The purpose of the following message schedule logic is to expand the 16 word input message block to the compression into an 64 word array w[t]. So for

0<=t<=15 w[t]=M[t], the 16 words. The remaining 48 words between 16<=t<=63 are calculated using the equation below. However, using w[64] is too expensive because we need 64:1 multiplexer. We optimize our code and solve this problem by implementing w[16].

- If $16 \leq t \leq 63$
  - $s_0 = (W_{t-15}$ **rightrotate** $7)$ **xor** $(W_{t-15}$ **rightrotate** $18)$ **xor** $(W_{t-15}$ **rightshift** $3)$
  - $s_1 = (W_{t-2}$ **rightrotate** $17)$ **xor** $(W_{t-2}$ **rightrotate** $19)$ **xor** $(W_{t-2}$ **rightshift** $10)$
  - $W_t = W_{t-16} + s_0 + W_{t-7} + s_1$

i.

c. We do this by using w[t-15], w[t-2], w[t-16], and w[t-7]. In total, we only need 16 bits of the word, so we can say that t-15 is i = max - 15 = 1, where max = 16. Therefore, the words we use are w[1], w[14], w[0], and w[9]. For every iteration we can take the word/value of the next iteration, w[n+1], and assign it to w[n] so we don't have to compute the previous 16 words again. Then we just load wtnew into w[15]. The logic for wtnew is as follows: We first start by finding the 15th word back at w[1]., then make two copies. We right rotate the first copy 7 places, the right rotate the second copy by 8 places. Lastly we shift the original word and shift it by 3; these three operations are joined by XOR operation. Then, we take the word at w[14], make two copies and do right rotate 17, right rotate 19 and shift by 10. Lastly, The new w[t], "Wtnew", is derived by using the word from 16 places back w[0], the word from 7 places back w[9] as well as the other two other hash operations. Resulting in code below.

i.

```
function logic [31:0] Wtnew; //step 4 sha 256 pdf pg 21
    logic [31:0] S0, S1;
        begin
            ///given on sha optimization pg 20
            S0= rightrotate(w[1], 7) ^ rightrotate(w[1], 18) ^ (w[1] >> 3);
            S1= rightrotate(w[14], 17) ^ rightrotate(w[14], 19) ^ (w[14] >> 10);
            Wtnew = w[0] + S0 + w[9] + S1;
        end
endfunction
```

3. More hashing

a. The logic behind the function for one hash round is below.

```
//Hash logic core
// SHA256 hash round
function logic [255:0] sha256_op(input logic [31:0] a, b, c, d, e, f, g, h, w,
                                 input logic [7:0] t);
    logic [31:0] S1, S0, ch, maj, t1, t2; // internal signals
begin
    // all is given on sha 256 pdf pg 23
    S0 =rightrotate(a,2)^rightrotate(a,13)^rightrotate(a,22);
    ch = (e&f)^((~e)&g);
    S1 = rightrotate(e, 6) ^ rightrotate(e, 11) ^ rightrotate(e, 25);
    t1 = h+S1+ch+k[t]+w;
    maj = (a&b)^(a&c)^(b&c);
    t2 = S0+maj;
    sha256_op = {t1 + t2, a, b, c, d + t1, e, f, g};
end
endfunction
```

```
// final right rotate expression is = (x >> r) | (x << (32-r));
// Right rotation function
function logic [31:0] rightrotate(input logic [31:0] x,
                                  input logic [ 7:0] r);

    begin rightrotate=(x>>r)|(x<<(32-r)); end

endfunction
```

This is the logic for the right rotate function we used for wtnew.

4. Generating request to memory
    a. Below are the usage of local variables.

```
assign mem_clk = clk;
assign mem_addr = cur_addr + offset;
assign mem_we = cur_we;
assign mem_write_data = cur_write_data;
```

        i.
        ii. cur_addr<=message_addr get starting address of message and will later become cur_addr<=output_addr when the final values are hashed .
        iii. cur_we<=1'b0 is set by default in the IDLE state and and will be 1 in the state WRITE where we write the final hash value to memory.
        iv. In WRITE statement, it is used to store hash values and write memory to output location one by one.
        v. offset is used to initialize pointer to access memory location. Offset<=32'b0 is set in the IDLE state and will increment each time one location in memory is read. Offset is used in the READ for offset<=mem_read_data, and for WRITE state is used as cur_write_data<=tmp[offset+1];

5. Sha 256 FSM
    a. IDLE: In this state we initialize the hash values h0 to h7 and a to h. The values below will be hashed by the right rotating iterations mentioned in part 2 and 3.

       Here we also initialize offset<=32'h0, cur_we<=1'b0 that will be 1'b1 in the READ state, i and j for for loops, and the next state PROCESS_1 where it will go to the state of READ. PROCESS_1 is used as a buffer state before READ.

    b. READ: Here if the offset is less than NUM_OF_WORDS=20, then w[offset]<=mem_read_data because we need to access memory location to access more data, which is why offset is incremented each time. This will load in all the words from memory.

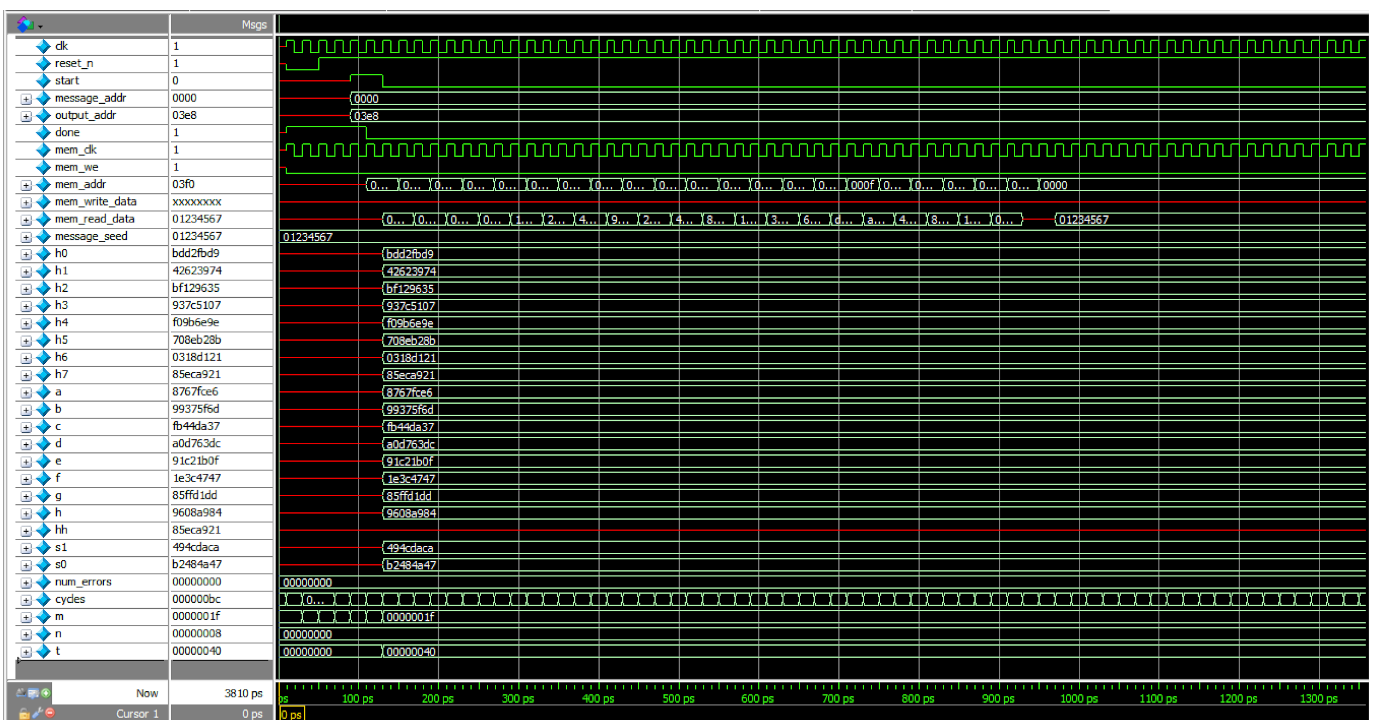
       Otherwise we pad the message with zero like mentioned in part 1, the padding takes place in state READ. Details on this are given in the testbench, but w[31] is where we pad the 64 bits of zero to reach 512 bits, while w[20] is the size of the original input. Offset is set to zero, and the next state is BLOCK.

c. BLOCK: This state is where we fetch w[t] in the 512 bit block and for each of the 512 bit block, we initiate hash value computation.
   If i is less than the number of blocks determined in part 1, we use proceed with the equation w[t]<=w[t+(i*16)] because there are 16 words per block. "i" is incremented each time until the number of blocks are reached, then the next state is COMPUTE. Otherwise, we move onto PROCESS_2 which is buffer stage where the hash value from the COMPUTE is temporarily hold until we get to the WRITE state where the values will be written into memory

d. COMPUTE: This state is where we hash the values for the second block. Here we call the sha256_op hash logic core, and this state will persist for 64 iterations until the final hash values of the second block h0 to h7 is reached taking the initialize hash values and add them to the first block hash value(i.e h0<=a+h0).

e. The last state is WRITE which serves to write the hash values that are temporarily stored in PROCESS_2 to cur_write_data, which is memory. This state also restarts the whole FSM process by initializing it back to the IDLE state.

SHA-256 Simulation waveform and Transcript



```
# Top level modules:
#       simplified_sha256
# End time: 16:15:13 on Mar 13,2021, Elapsed time: 0:00:01
# Errors: 0, Warnings: 0
vlog -reportprogress 300 -work work {C:/Users/user/Desktop/ECE 111 HW/Final_Project (1)/Final_Project/simplified_sha256/tb_simplified_sha256.sv}
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 16:15:13 on Mar 13,2021
# vlog -reportprogress 300 -work work C:/Users/user/Desktop/ECE 111 HW/Final_Project (1)/Final_Project/simplified_sha256/tb_simplified_sha256.sv
# -- Compiling module tb_simplified_sha256
#
# Top level modules:
#       tb_simplified_sha256
# End time: 16:15:13 on Mar 13,2021, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
ModelSim> vsim work.tb_simplified_sha256
# vsim work.tb_simplified_sha256
# Start time: 16:15:16 on Mar 13,2021
# Loading sv_std.std
# Loading work.tb_simplified_sha256
# Loading work.simplified_sha256
VSIM 5> run -all
# --------
# MESSAGE:
# --------
# 01234567
# 02468ace
# 048d159c
# 091a2b38
# 12345670
# 2468ace0
# 48d159c0
# 91a2b380
# 23456701
# 468ace02
# 8d159c04
# 1a2b3809
# 34567012
# 68ace024
# d159c048
# a2b38091
# 45670123
# 8ace0246
# 159c048d
# 00000000
# ***************************
#
```

```
# --------------------
# COMPARE HASH RESULTS:
# --------------------
# Correct H[0] = bdd2fbd9 Your H[0] = bdd2fbd9
# Correct H[1] = 42623974 Your H[1] = 42623974
# Correct H[2] = bf129635 Your H[2] = bf129635
# Correct H[3] = 937c5107 Your H[3] = 937c5107
# Correct H[4] = f09b6e9e Your H[4] = f09b6e9e
# Correct H[5] = 708eb28b Your H[5] = 708eb28b
# Correct H[6] = 0318d121 Your H[6] = 0318d121
# Correct H[7] = 85eca921 Your H[7] = 85eca921
# ***************************
#
# CONGRATULATIONS! All your hash results are correct!
#
# Total number of cycles:         188
#
#
# ***************************
#
# ** Note: $stop    : C:/Users/user/Desktop/ECE 111 HW/Final_Project (1)/Final_Project/simplified_sha256/tb_simplified_sha256.sv(262)
#    Time: 3810 ps  Iteration: 2  Instance: /tb_simplified_sha256
# Break in Module tb_simplified_sha256 at C:/Users/user/Desktop/ECE 111 HW/Final_Project (1)/Final_Project/simplified_sha256/tb_simplified_sha256.sv line 262

VSIM 6>
```

Bitcoin_hash Introduction
Blockchains store digital information about financial transactions like the date, time, sender, receiver, etc, and the chaining of these blocks are dependent on cryptographic hashing algorithms. The bitcoin blockchain, in particular, relies on SHA 256 to generate the signatures that link the blocks together. However, a signature doesn't always qualify for a block to be accepted into the block chain. In order to find the correct signature, a random string of data in the block called the nonce needs to be changed repeatedly. The heavy computational burden it takes to generate the correct signature for every block on the blockchain makes it immutable and very resilient to corruption. In our algorithm we implement 16 iterations of the nonce, and the resulting hash value is considered as the target goal.

Algorithm Description

SHA_256 INCLUSION: The hash logic core and the message schedule logic from the SHA_256 are included to perform the actually hashing of the bitcoin information. In depth analysis of this is in the simplified_sha256 section.

IDLE, BUFFER, ASSIGN_1: The purpose of these processes is to initialize eight h0 registers to the given 32-bit constants. These values are then assigned to A through H for the main hashing algorithm, while the first 16th words from data is assigned to w[15]. We also initialize some other control signals to take and set things into memory and to keep track of the hashing process, such as setting mem_we to 0, counter and t to 0, and incrementing mem_addr.

PRE_COMPUTE 1: Here we calculate and update tmp_value, the variable that contains h, k and w. The purpose of the for loop is to assign the values from memory to w[0] to w[15], the first 16 words. We also increment mem_addr and t each time to keep track of what iteration we are on.

HASH_1: Keep in mind that each hashing block must persist for 64 iterations to fully assign the old values to new values. Thus, if t<65 we continue to assign values to the first 16 words. If the iteration is less than 15, we fetch the data from memory and assign those values to the 16 words. Otherwise, we assign the Wtnew hashed values to the 16 words and call the sha256_op function to update A through H. If all the processes from above go through, we assign the eight h0 registers and the new A-H to the new 8 registers h1 to move on to the second block.

ASSIGN_2, PRE_COMPUTE2: These two processes serve similar purposes as the ASSIGN_1 and PRE_COMPUTE1; we assigned the 8 h1 registers from the first block to A through H, compute tmp_value, organize the first 16 words and assign their values by fetching data from memory all to prepare for the hashing of the second block

HASH_2: We first address the Wt that corresponds to the last 3 words in the memory, the nonce, and padding, which are all within the 64 iterations of hashing. We read the last words of the message and then update the remaining words with the nonce, padding, and message length. Then, hashing is performed using the A-H values from the result of the first hashing round. We add the new A-H values with the hold hash outputs in h1 and assign the result to the h2 registers. We also reset the h0 registers to the default values given to prepare for the processing of the 2nd SHA 256 hash function.

ASSIGN_3, PRE_COMPUTE3: These two processes serve similar purposes as the ASSIGN_2 and PRE_COMPUTE2; h0 values are passed to A-H and w is assigned the results from the 2nd phase with the padding which is stored in h2.

HASH_3: The hashing here is similar to the hashing in HASH_2, but here we process the second sha hash function. We also address the other three words from the first hashm the nonces and padding, all of which take place within the 64 iterations. After processing is finished, we prepare for the WRITE state, like setting mem_we to 1 and changing mem_addr to make sure we are writing to the output address. At the end, we write the result of the first hash round (and all subsequent iterations), h[0] + A, to memory.

CHECKING AND WRITE: If we did not store all 16 final hash values into memory yet, then we increment i, the index of the nonce, to move to the next nonce value and start CHECKING, which assigns h1, the previous hash results of the first 16 words, to h0 so that we do not have to rehash those values again. We then move to ASSIGN_2 and go through the process again for the new nonce value. This is how 16 iterations are performed. If all 16 final hashes are achieved and written into memory, then done is set to 1 and we go back to IDLE.

BITCOIN_HASH Simulation waveform and transcript

```
# Top level modules:
#       tb_bitcoin_hash
# End time: 15:38:27 on Mar 20,2021, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
ModelSim> vsim work.tb_bitcoin_hash
# vsim work.tb_bitcoin_hash
# Start time: 15:38:31 on Mar 20,2021
# Loading sv_std.std
# Loading work.tb_bitcoin_hash
# Loading work.bitcoin_hash
add wave -position insertpoint sim:/tb_bitcoin_hash/*
VSIM 6> run -all
# ---------------
# 19 WORD HEADER:
# ---------------
# 01234567
# 02468ace
# 048d159c
# 091a2b38
# 12345670
# 2468ace0
# 48d159c0
# 91a2b380
# 23456701
# 468ace02
# 8d159c04
# 1a2b3809
# 34567012
# 68ace024
# d159c048
# a2b38091
# 45670123
# 8ace0246
# 159c048d
# ***************************
```

```
#
# ---------------------
# COMPARE HASH RESULTS:
# ---------------------
# Correct H0[ 0] = 7106973a Your H0[ 0] = 7106973a
# Correct H0[ 1] = 6e66eea7 Your H0[ 1] = 6e66eea7
# Correct H0[ 2] = fbef64dc Your H0[ 2] = fbef64dc
# Correct H0[ 3] = 0888a18c Your H0[ 3] = 0888a18c
# Correct H0[ 4] = 9642d5aa Your H0[ 4] = 9642d5aa
# Correct H0[ 5] = 2ab6af8b Your H0[ 5] = 2ab6af8b
# Correct H0[ 6] = 24259d8c Your H0[ 6] = 24259d8c
# Correct H0[ 7] = ffb9bcd9 Your H0[ 7] = ffb9bcd9
# Correct H0[ 8] = 642138c9 Your H0[ 8] = 642138c9
# Correct H0[ 9] = 054cafc7 Your H0[ 9] = 054cafc7
# Correct H0[10] = 78251a17 Your H0[10] = 78251a17
# Correct H0[11] = af8c8f22 Your H0[11] = af8c8f22
# Correct H0[12] = d7a79ef8 Your H0[12] = d7a79ef8
# Correct H0[13] = c7d10c84 Your H0[13] = c7d10c84
# Correct H0[14] = 9537acfd Your H0[14] = 9537acfd
# Correct H0[15] = c1e4c72b Your H0[15] = c1e4c72b
# **************************
#
# CONGRATULATIONS! All your hash results are correct!
#
# Total number of cycles:        2246
#
#
# **************************
#
# ** Note: $stop    : C:/Users/user/Desktop/ECE 111 HW/Final_Project (1)/Final_Project/bitcoin_hash/tb_bitcoin_hash.sv(334)
#    Time: 44970 ps  Iteration: 2  Instance: /tb_bitcoin_hash
# Break in Module tb_bitcoin_hash at C:/Users/user/Desktop/ECE 111 HW/Final_Project (1)/Final_Project/bitcoin_hash/tb_bitcoin_hash.sv line 334
```

Synthesis resource usage report for bitcoin_hash

	Resource	Usage
1	Estimated ALUTs Used	1519
1	-- Combinational ALUTs	1519
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	1646
3		
4	Estimated ALUTs Unavailable	16
1	-- Due to unpartnered combinational logic	16
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	1519
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	16
2	-- 6 input functions	157
3	-- 5 input functions	32
4	-- 4 input functions	45
5	-- <=3 input functions	1269
8		
9	Combinational ALUTs by mode	
1	-- normal mode	675
2	-- extended LUT mode	16
3	-- arithmetic mode	732
4	-- shared arithmetic mode	96
10		
11	Estimated ALUT/register pairs used	2067
12		
13	Total registers	1646
1	-- Dedicated logic registers	1646
2	-- I/O registers	0
3	-- LUT_REGs	0
14		
15		
16	I/O pins	118
17		
18	DSP block 18-bit elements	0
19		
20	Maximum fan-out node	clk~input
21	Maximum fan-out	1647
22	Total fan-out	11309
23	Average fan-out	3.33

Estimated ALUTs used: 1519
Total registers: 1646

Fitter report for bitcoin_hash

Fitter Summary

<<Filter>>

Fitter Status	Successful - Sat Mar 20 15:37:06 2021
Quartus Prime Version	20.1.0 Build 711 06/05/2020 SJ Lite Edition
Revision Name	bitcoin_hash
Top-level Entity Name	bitcoin_hash
Family	Arria II GX
Device	EP2AGX45DF29I5
Timing Models	Final
Logic utilization	6 %
Combinational ALUTs	1,655 / 36,100 (5 %)
Memory ALUTs	0 / 18,050 (0 %)
Dedicated logic registers	1,646 / 36,100 (5 %)
Total registers	1646
Total pins	118 / 404 (29 %)
Total virtual pins	0
Total block memory bits	0 / 2,939,904 (0 %)
DSP block 18-bit elements	0 / 232 (0 %)
Total GXB Receiver Channel PCS	0 / 8 (0 %)
Total GXB Receiver Channel PMA	0 / 8 (0 %)
Total GXB Transmitter Channel PCS	0 / 8 (0 %)
Total GXB Transmitter Channel PMA	0 / 8 (0 %)
Total PLLs	0 / 4 (0 %)
Total DLLs	0 / 2 (0 %)

Timing report

Slow 900mV 100C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	183.18 MHz	183.18 MHz	clk	