

ECE 111 Final Project Report

Sha256

Hamad Alajeel, Mack Markham, & Isabel Senske
A16369878, A16391630, & A15876840

➤ Explanation of What Sha256 Is:

SHA-256 is a Secure Hashing Algorithm that transforms messages of up to 2.3 billion gigabytes into fixed 256 bit digests. It computes a unique 256 bit hash value for any unique input message, no matter the size of the input or the data stored in it. In order to make it more secure, any slight changes in input create large avalanche effects that dramatically change the output hash. However, the same input will always generate the same output, no matter the system it is run on. Because it is a hash, not an encryption, SHA-256 is one-way, meaning there is no way to reverse the process and get the input message from the hash. Because of the large number of possible hashes, it is also collision-resistant, which means it is practically impossible for two inputs to create the same output.

➤ Explanation of Code Implementation:

To begin with, we should specify a few things about the design which form the basis of what the design is built on. The clock used in this design is the clock obtained from the input, so it is in sync with that clock. Secondly, this design is an FSM which is based upon 5 states: IDLE, READ, BLOCK, COMPUTE, and WRITE. These states are enumerated logic variables. To begin processing whatever message is required, the design has to read from memory, that is done in the READ state, and which is why we have declared and used these variables: `cur_we` (is the enable signal to read/write from memory), `cur_addr` (the address from which we want to read/write from memory), `cur_write_data` (data which we want to write to memory), `offset` (used as the offset value from the initial address of memory or the message variable we use). These local variables which change during the execution of the sha256 process continuously drive these outputs (using assign) to the

module to be able to communicate with the memory it is writing/reading from: `mem_addr`, `mem_we`, and `mem_write_data`. The constants used in the sha256 process are defined within the module. The variable `w`, an unpacked array, carries whatever words which are processed during each stage, and we use the variable `message` to input the data we read from memory. The variables used for the hash production are `h0` to `h7`, `a` to `h`, and `optim` (a variable used for optimization). We have also defined functions to aid us in the processing of the sha256, and the most fundamental function would be the `sha256_op` which does the main part of the hash production. Moreover, our design is based on an always_ff@ which triggers at the positive edge of the clock or the negative edge of the reset signal, active low reset.

Now that we have detailed the layout we'll begin to explain the algorithm step by step. Firstly, the process begins in IDLE where we wait for the external input, `start`, to be high, otherwise everything is initialized to 0. When `start` is high, we initialize `h0` to `h7` and `a` to `h` to our initial hashes 32'h6A09E667 to 32'h5be0cd19; we input the external input `message_addr` to `cur_addr`, `offset` to 0, and we input `READ` to `state`, because at the next state we begin reading from memory. Effectively, this means that at the next cycle we command the memory to provide us with data (note it takes one cycle for the memory to provide a 32 bit data block).

Now that we are at `READ`, we know that we have signaled for the memory, in IDLE, to provide the 32 bit data block at memory offset 0, but we still need to wait for one more clock cycle for that to be provided. We also increase the offset by 1, sending another command to memory to give us the data at offset 1 (again this will need one more clock cycle). After this one clock cycle delay, our offset is 1 but we only have the data from the memory at offset 0 ready, so we input `message[offset-1] <= mem_read_data`, and at the next clock cycle the data at offset 1 will be ready. This form of pipelining repeats itself until all the 20 32-bit words from memory are inputted into our message variable. When we have finished reading, we can proceed to the block state.

At the block state, we know, initially, that our message is 640 bits, and we also know that each message block in the abstract model of the sha256 process is 512 each, so we can't fit 640 bits in one block. Therefore, we will have two blocks to accommodate that. One block is filled with the first 16

words, 16×32 , while the other block is filled with the rest of the 4 words in the message variable and padding determined by the sha256 specification (a 1, then 0's, then the message size). The variable j is used as a counter in this state to check for which block will be currently processed. According to the block we are at, different data is inputted to the w array, which holds 16 32-bit values according to what we have discussed earlier. Until all the blocks have been processed, BLOCK will transition to COMPUTE, otherwise it will transition to WRITE where it will begin writing to memory. Moreover, for each block, a variable $optim$ is computed to be used as a pre-computation which helps with the aggressive pipelining we have implemented.

In the COMPUTE stage, two aggressive pipelining methods have been applied: pipelining the word expansion, and pipelining the computation of $h + k$ in the $t1$ term of that sha256 function which is the second stage of the COMPUTE state. Instead of 64 cycles of word expansion, and then 64 cycles of the sha256 process, we can do them in parallel because we know that the first 16 words of the word expansion are equal to the beginning 16 words of the w array. We also know that when expanding the words, each expansion at indexes above 15 are based on the values of w at previous indexes, so we can use values of w at previous indexes to calculate the expansion of indexes that are greater than 15. We use the value $w[0]$ for the first iteration of the sha256 function, then we rotate the values of w backward, so $w[0] \leftarrow w[1]$ etc in the form of a for loop, and at index 15, we input the word expansion for the 16th index, $w[15] \leftarrow w_{new}$. This process is repeated 64 times. Effectively, we reduce the amount of cycles by 64 and we use less area because the w array is only 16 elements. The second form of pipelining is the $optim$ variable. Because we know that the next value of h will be g , and only g , we can precompute $h + k + w$ used for the $t1$ term, the k and w at the index $i + 1$ are used (again, because we are pre-computing this for the next cycle). So, the computation part runs for 64 cycles and outputs new hashes for a to h . After this, a to h and $h0$ to $h7$ are initialized to $h0$ to $h7 + a$ to h . These form the output hashes from this COMPUTE stage. We then transition back to the BLOCK state where we either continue with the next block, or we are done and can begin writing to memory.


At the WRITE state there is one basic thing that is noted. When `cur_we`, the enable signal to write to memory, becomes high we need to wait one cycle to begin writing to memory. A case statement implements that alongside the counter variable `i`. We output each final hash value to each offset in memory, and when we finish writing to memory, we transition back to the IDLE state.

```
assign mem_write_data = cur_write_data;
```

```
assign mem_addr = cur_addr + offset;
```


These two equations determine the data that will be written and the address to be written to. When the state is equal to IDLE, the done signal is high.

➤ Resource Usage Summary:


Analysis & Synthesis Resource Usage Summary		
 <<Filter>>		
	Resource	Usage
1	▼ Estimated ALUTs Used	2132
1	-- Combinational ALUTs	2132
2	-- Memory ALUTs	0
3	-- LUT_REGS	0
2	Dedicated logic registers	1782
3		
4	▼ Estimated ALUTs Unavailable	15
1	-- Due to unpartnered combinational logic	15
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	2132
7	▼ Combinational ALUT usage by number of inputs	
1	-- 7 input functions	15
2	-- 6 input functions	155
3	-- 5 input functions	12
4	-- 4 input functions	9
5	-- <=3 input functions	1941
8		
9	▼ Combinational ALUTs by mode	
1	-- normal mode	1511
2	-- extended LUT mode	15
3	-- arithmetic mode	478
4	-- shared arithmetic mode	128
10		

11	Estimated ALUT/register pairs used	2608
12		
13	▼ Total registers	1782
1	-- Dedicated logic registers	1782
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	1783
22	Total fan-out	14084
23	Average fan-out	3.39

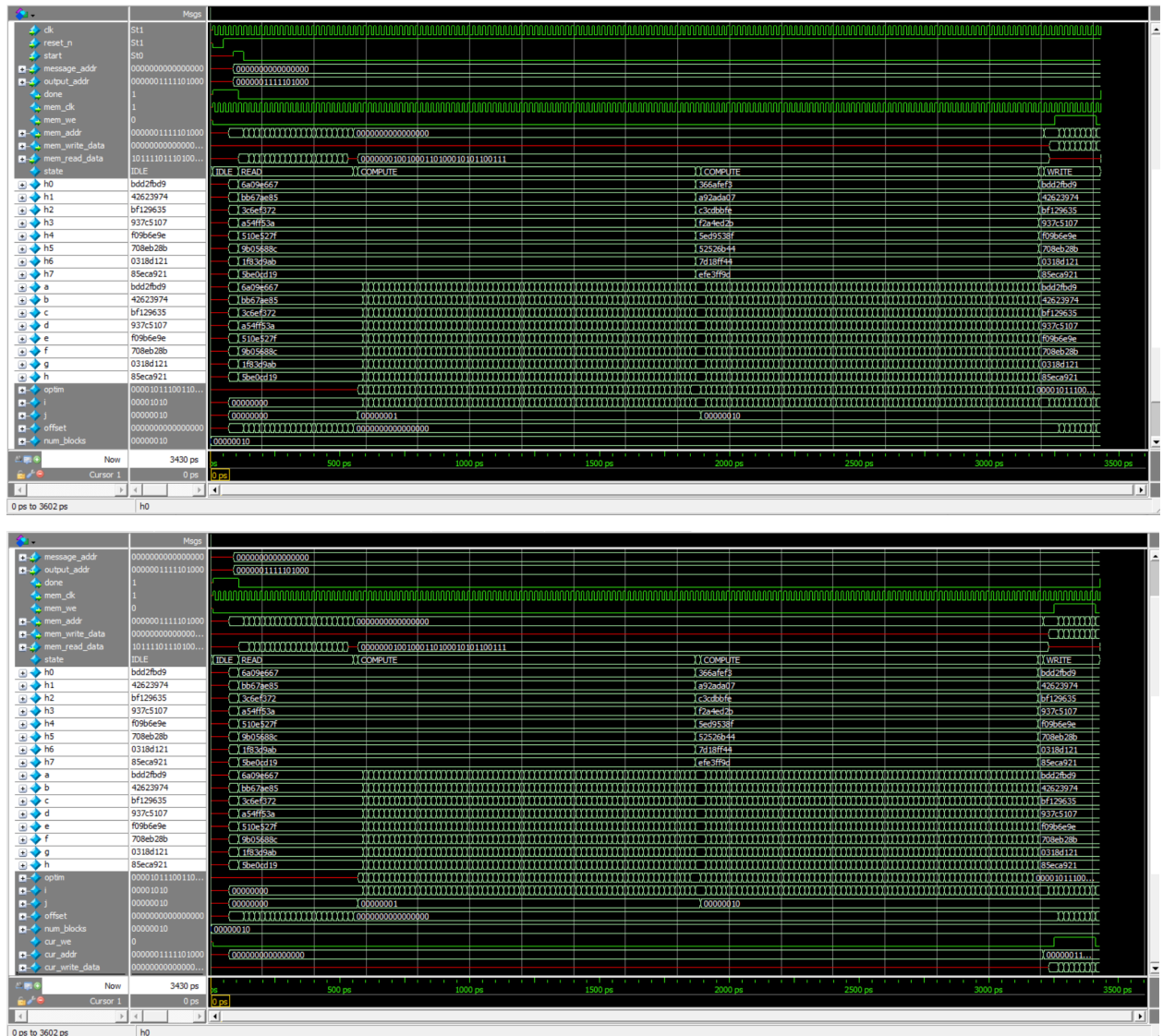
➤ Fitter Report Snapshot:

Fitter Summary	
 <<Filter>>	
Fitter Status	Successful - Fri Jun 09 21:45:23 2023
Quartus Prime Version	20.1.0 Build 711 06/05/2020 SJ Lite Edition
Revision Name	simplified_sha256
Top-level Entity Name	simplified_sha256
Family	Arria II GX
Device	EP2AGX45DF29I5
Timing Models	Final
Logic utilization	8 %
Total registers	1782
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 Analysis Snapshot:

Slow 900mV 100C Model Fmax Summary				
 <<Filter>>				
	Fmax	Restricted Fmax	Clock Name	Note
1	185.49 MHz	185.49 MHz	clk	

➤ Simulation Waveform Snapshot:



- Note: The snapshots of this waveform you are seeing here are all the items in the `simplified_sha256_function` (The instantiation of our design module). These aren't the items from the testbench.

➤ Simulation Transcript Snapshot:

```
VSIM 6> run
# -----
# 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:      169
#
# *****
#
# ** Note: $stop      : C:/Users/hamad/Desktop/ECE111/Project_Files/simplified_sha256/tb_simplified_sha256.sv(262)
#      Time: 3430 ps  Iteration: 3  Instance: /tb_simplified_sha256
# Break in Module tb_simplified_sha256 at C:/Users/hamad/Desktop/ECE111/Project_Files/simplified_sha256/tb_simplified_sha256.sv line 262
```