Simplified SHA-256 Report ECE-111

What is SHA-256?

SHA stands for "Secure Hash Algorithm." SHA is a cryptography algorithm that converts input data of any kind and size, into a string of fixed length. The goal of the SHA algorithm is to compute a unique hash value for any input data, and no matter the input size, the output size is fixed. There are many different SHA algorithms (SHA-1, SHA-2, SHA-3, …, SHA-256,...). Each of these algorithms can handle a certain input message size and they produce a specific fixed size output hash. For SHA-256, the algorithm can handle input messages up to 2^64 bits and those are hashed into an output of 256 bits no matter the input.

The hashing function needs to have 6 specific properties in order to be completely cryptographically secure: compression, avalanche effect, determinism, pre-image resistant, collision resistant, and efficient. The compression property says that an output hash should be a fixed length, regardless of the size of the input message. The avalanche effect property says that a minimal change in the input should dramatically change the output hash. The determinism property says that the same input must always generate the same output by different systems. The pre-image resistant property says that a SHA algorithm should be a one-way function, meaning no algorithm can reverse the hashing process to retrieve the original input message (unlike encryption which is a two-way function, i.e. encryption-decryption). The efficient property says that creating the output hash should be a fast process that does not make heavy use of computing power. And finally, the collision resistance property says that a hashing algorithm should be rigorous and it must withstand collisions (hackers can take advantage of this).

There are many different applications of SHA-256, so here are a few examples. Let's say that a software manufacturer wants to ensure that an executable file is received by users without modifications. They could send out a file to the users and publish its hash in the NY Times. It would be incredibly difficult for hackers to create a bad file that will have the same hash as the good file. This way, once users check the hash of their received file, they will know if it has been tampered with. Another example would be having some sort of software that requires usernames and passwords. A backend database would store the hash of a password instead of the password itself in order to protect against attacks and data breaches. Those were a couple different examples of applications of SHA-256, but there are many more that help us secure our data every day.

Reference: Final Project Part-1: SHA256 Slide Deck -Vishal Karna

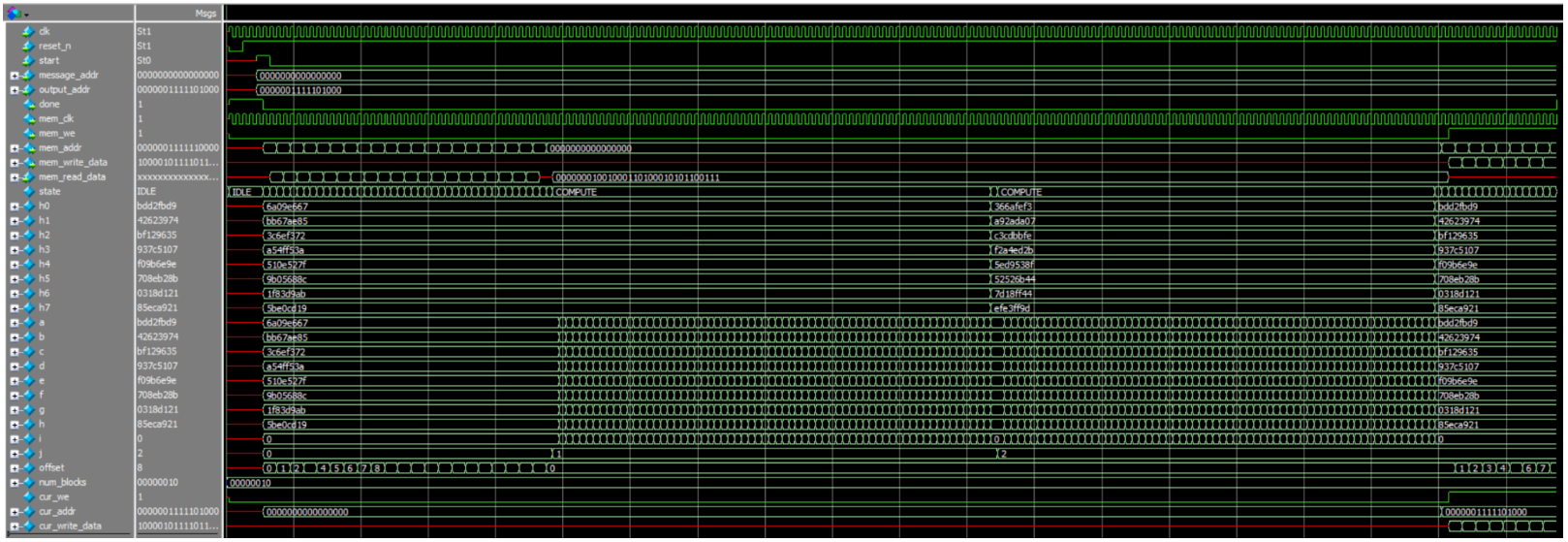
Algorithm

Unoptimized: The basic SHA-256 algorithm implementation comprises a few different steps. We receive an input message of n x 512 bits that we pad with 1 to 448 bits of 0's with 64 bits of the input message size appended to it. We then split those bits up into message blocks of 512 bits. We perform word expansion on each block to get 64 bits, and then we take each message block and do 64 rounds of compression. At the end of the 64 rounds of compression, we add the results of the SHA operation to the previous stage's hash. This process will be repeated for each message block. Once the final message block's hash is computed, the initial hash is added to get the final output hash.

This implementation was done in SystemVerilog using states: IDLE, READ_BUFFER, READ, BLOCK, COMPUTE, WRITE_BUFFER, WRITE. IDLE is the state we start in and it is the state that marks the hashing is complete. READ_BUFFER and READ states are used to ingest the input message. BLOCK is used to perform word expansion and split up the message into blocks. COMPUTE is used to perform the 64 rounds of compression and the final addition to the previous hash. And finally, WRITE and WRITE_BUFFER are used to write the output hash to the output port.

Optimized: We did not need to make significant changes to optimize the SHA-256 module to run in under 200 cycles. In the unoptimized module, it ran in around 320 cycles, and after optimization, it ran around 190 cycles. The first thing that we optimized was using a for loop to block the message instead of using a more global index. This brought the module down a few cycles. The bulk of the optimization came from removing the word expansion to 64 bits, and keeping the word size at 16 bits. Word expansion for 64 bits was very expensive because it means that 64 32-bit registers and 64:1 multiplexors. Removing word expansion, and keeping it to 16 bits significantly reduced the number of cycles, and we were able to get the simulation under 200 cycles. We could have optimized the module even further by pipelining and pre-computing values that were being used in the SHA compression operation itself.

Simulation



Transcript

```
VSIM 17> run
# GetModuleFileName: The specified module could not be found.
#
#
# --------
# 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:         195
#
#
# ****************************
#
# ** Note: $stop    : C:/Users/vijpr/OneDrive/Desktop/ECE 111/Fi
nalProject/simplified_sha256/tb_simplified_sha256.sv(262)
#    Time: 3950 ps  Iteration: 2  Instance: /tb_simplified_sha25
6
```

All hash results are correct. Completed in 195 cycles.

Resource Usage

	Resource	Usage
1	▼ Estimated ALUTs Used	1714
1	-- Combinational ALUTs	1714
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	2136
3		
4	▼ Estimated ALUTs Unavailable	0
1	-- Due to unpartnered combinational logic	0
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	1714
7	▼ Combinational ALUT usage by number of inputs	
1	-- 7 input functions	0
2	-- 6 input functions	176
3	-- 5 input functions	10
4	-- 4 input functions	70
5	-- <=3 input functions	1458
8		
9	▼ Combinational ALUTs by mode	
1	-- normal mode	1218
2	-- extended LUT mode	0
3	-- arithmetic mode	368
4	-- shared arithmetic mode	128
10		
11	Estimated ALUT/register pairs used	2658
12		
13	▼ Total registers	2136
1	-- Dedicated logic registers	2136
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	2137
22	Total fan-out	14486
23	Average fan-out	3.55

1. **ALUTs:** 1714
2. **Registers:** 2136
3. **Area:** 3850 - Calculated in “simplified_sha256/finalsummary.xlsx”
4. Fitter file located in “simplified_sha256/simplified_sha256.fit.rpt”

Timing Report

Revision Name	simplified_sha256
Device Family	Arria II GX
Device Name	EP2AGX45DF29I5

Slow 900mV 100C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name
1	152.86 MHz	152.86 MHz	clk

1. **Fmax:** 152.86 MHz
2. Static timing analysis file located in “simplified_sha256/simplified_sha256.sta.rpt”