SHA-256

1. SHA-256 explanation

SHA-256 is one of the most widely used hashing functions in cryptographic systems, which is a member of the SHA algorithm family, including SHA-1, SHA-2, and SHA-3. The main function of SHA-256 is to securely convert any size of input message and data into a fixed 256-bit output digest. This digest plays a role like a digital footprint of the original input data or message, which makes the SHA-256 algorithm have a lot of applications, like Digital Signatures and certificates, Password storage and verification, Software updates and distribution, Data integrity and verification, and so on.

The goal of SHA-256 is to generate a unique hashing value for any size of input data and message.

SHA-256 has several strong properties as mentioned following:

1. Compression

Its output hash is always a 256-bit fixed output, no matter what size of input data and message is, which ensures consistency in wild applications.

2. Avalanche Effect

Even a tiny change in the input data and message will lead to a completely different hash output. This helps to prevent hackers from predicting the output value, ensuring unpredictivity and security.

3. Determinism

The same input message or data must generate the identical output hash value, regardless of how the computation works and on a different system.

4. Pre-Image Resistant (One-Way Function)

It is impossible to reverse the hash value to get the original input data and message, which acts like a one-way function, ensuring security for storing sensitive data and messages.

5. Collision Resistance

It is also computationally impossible to generate the same output hash value from 2 different input data and messages, which could prevent hackers from generating the same value from fake input data.

6. Efficient (Quick Computation)

SHA-256 is fast and efficient, which does not require excessive computational power and resources.

2. Algorithm description

Our SHA-256 algorithm takes a fixed 640-bit message, processes it in 512-bit chunks, and generates a final 256-bit digest. Our design utilizes several helper functions and a finite state machine, as well as optimizations. Details will be described below.

Functions:

determine_num_blocks( ): This function will calculate how many 512-bit chunk is required to process a 640-bit input message, which will generate 2 blocks.

wtnew( ): This function utilizes rotations, shifts, and XOR operations to compute new 32-bit message words during the word expansion phase.

sha256_op( ): This function implements one round of SHA-256 compression following the formula, which computes Σ functions with choices and majority functions, updates a-h variables, and k[t].

rightrotate( ): This is a helper function that performs bit rotation for both message expansion and compression.

FSM states:

IDLE: This state will initialize H0-H7, reset all variables, and wait for the start signal.

WAIT: This state will wait for 1 cycle.

READ: This state will read messages into the message[] buffer.

BLOCK: This state will prepare a 512-bit block. For the first block, 16 words are directly taken from the message. For the second block, get the remaining 4 words and padding to complete the block.

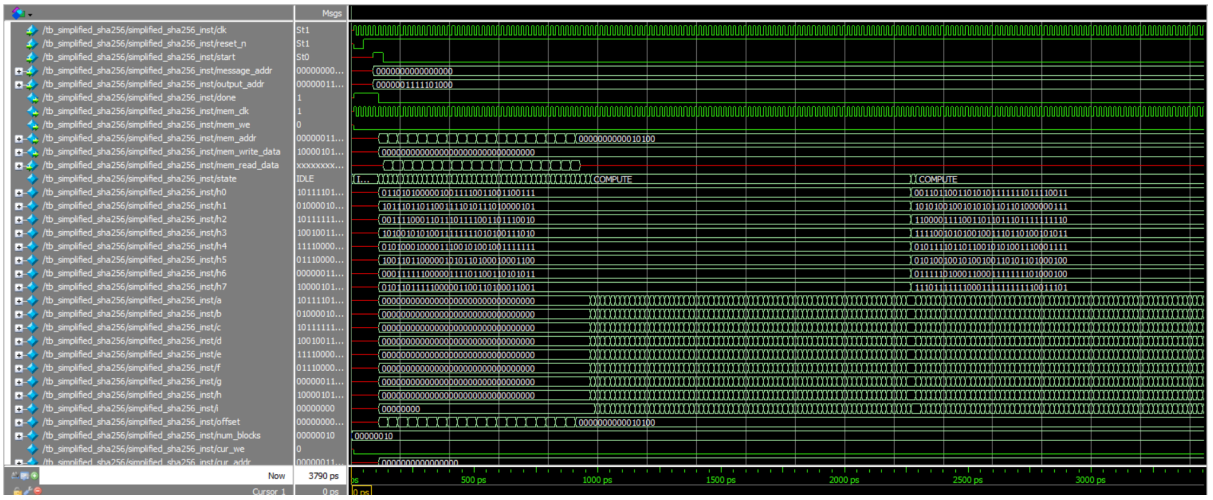
COMPUTE: This state will run 64 rounds of SHA-256 operations, which will also perform word expansion and sha256_op( ), put the result back to H0-H7.

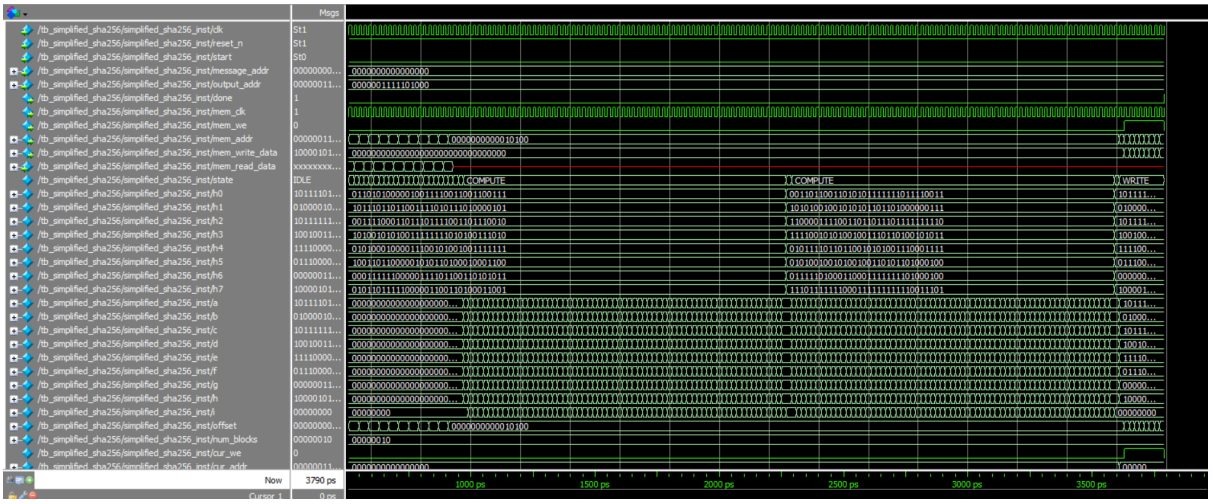
WRITE: This state will write the final digest H0-H7 into memory.

Optimization:

Our code uses w[16] for word expansion instead of w[64], which can save resources. Also, in the COMPUTE state, we applied the optimization technique from the lecture slides, which will reduce the cycle counts.

3. Simulation results





4. Modelsim transcript

```
VSIM 6> 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:         187
#
#
# ***************************
```

As the simulation transcript shown, all of the hash results are correct, and by the optimization technique applied, the number of cycles is reduced to 187, which is significantly improved.

Bitcoin hashing

1. Bitcoin hashing explanation

Bitcoin hashing is the process of securing and validating blocks in the Bitcoin blockchain, which uses the SHA-256 algorithm, to generate a unique digital footprint of each block header. Due to the Avalanche Effect property of the SHA-256 algorithm, small changes in the blockchain will result in a different hash value, which ensures security and uniqueness.

A Bitcoin block header has a fixed size of 19 words of data, as fields of Version, hashPrevBlock, hashMerkleRoot, Time, Bits, Nonce. For Nonce, in our final project, instead of trying the full random nonce space, we only test values from 0 to 15, which means a total of 16 attempts will be made.

The hashing procedure is divided into three phases. Phase 1 will process the first 512-bit block of the block header using SHA-256 with initial constants. Phase 2 will apply SHA-256 to the second block, including nonce and padding. Phase 3 will apply the SHA-256 algorithm again on the output of Phase 2, completing the operation to generate the final digest.

This simplified project shows the fundamental techniques and mechanics of Bitcoin mining, which ensures the security of the blockchain.

2. Algorithm description

The Bitcoin hashing module implements the Bitcoin hashing using the SHA-256 algorithm, which supports parallelization of multiple nonces. A block header of a size of 20 words is read, processes 3 phases, generates a digital digest, and writes back to memory.

Functions:

sha256_op( ): This function implements one round of SHA-256 compression following the formula, which computes Σ functions with choices and majority functions, updates a-h variables, and k[t].

wt_expansion( ): Generates new message words during SHA-256 word expansion and make it pararrel.

rightrotate( ): This is a helper function that performs bit rotation for both message expansion and compression.

FSM states:

IDLE: This state will initialize all variables and reset all local variables, wait for the start state.

WAIT: This state will wait for 1 cycle.

READ: This state will load 20 words into the buffer.

PHASE1_BLOCK / PHASE1_COMPUTE: This state will process the first 512-bit block, generating values for all nonces.

PHASE2_BLOCK / PHASE2_COMPUTE: This state will compute the first SHA-256 output hash value for each nonce in parallel.

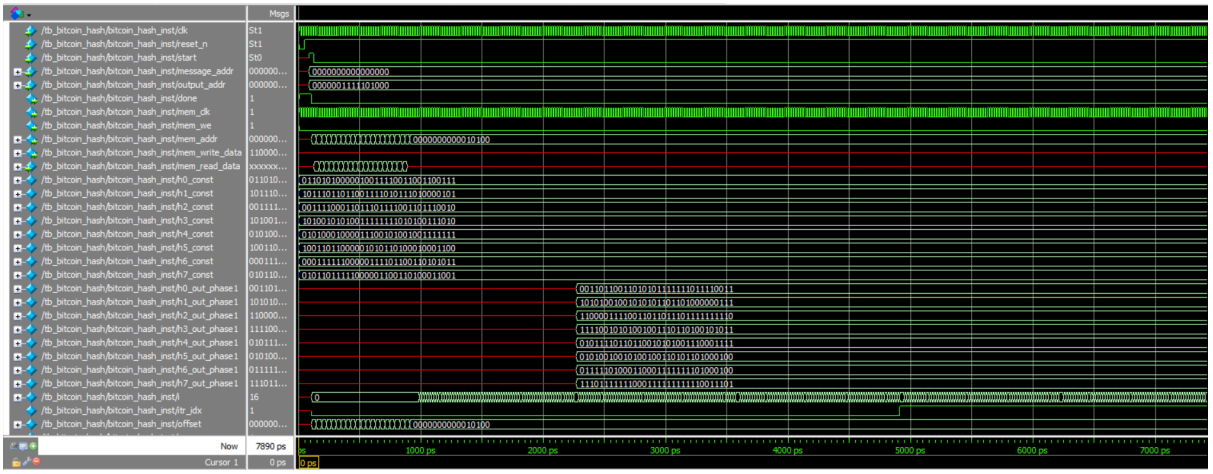
PHASE3_BLOCK / PHASE3_COMPUTE: This state will compute the final digest from the output of the PHASE2_BLOCK state.

WRITE: This state will write the final hash value back to memory.

Optimization:

1. Our code uses w[16] for word expansion instead of w[64], which can save resources.
2. Our code has Parallel Nonce Handling. Multiple nonces are processed simultaneously.
3. Our code also splits 16 nonces into the for loops of 8 iterations twice, to reduce hardware usage.

Simulation results



3. Modelsim transcript

```
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
```

```
# --------------------
# 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:         392
#
```

By applying the 8 nonce parallel optimization, the number of cycles is reduced to 392, which is improved significantly.

4. Resources usage

Analysis & Synthesis Resource Usage Summary

<<Filter>>

	Resource	Usage
1	▼ Estimated ALUTs Used	13218
1	-- Combinational ALUTs	13218
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	9676
3		
4	▼ Estimated ALUTs Unavailable	658
1	-- Due to unpartnered combinational logic	658
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	13218
7	▼ Combinational ALUT usage by number of inputs	
1	-- 7 input functions	64
2	-- 6 input functions	2040
3	-- 5 input functions	1378
4	-- 4 input functions	49
5	-- <=3 input functions	9687
8		
9	▼ Combinational ALUTs by mode	
1	-- normal mode	9250
2	-- extended LUT mode	64
3	-- arithmetic mode	2880
4	-- shared arithmetic mode	1024
10		
11	Estimated ALUT/register pairs used	15803
12		
13	▼ Total registers	9676
1	-- Dedicated logic registers	9676
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	9677
22	Total fan-out	83046
23	Average fan-out	3.59

5. **Timing and Fitter report**

Slow 900mV 100C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	164.1 MHz	164.1 MHz	clk	

```
Fitter Status : Successful - Tue Sep  9 22:29:31 2025
Quartus Prime Version : 22.1std.1 Build 917 02/14/2023 SC Lite Edition
Revision Name : bitcoin_hash
Top-level Entity Name : bitcoin_hash
Family : Arria II GX
Device : EP2AGX45CU17I3
Timing Models : Final
Logic utilization : 46 %
    Combinational ALUTs : 13,223 / 36,100 ( 37 % )
    Memory ALUTs : 0 / 18,050 ( 0 % )
    Dedicated logic registers : 9,676 / 36,100 ( 27 % )
Total registers : 9676
Total pins : 118 / 176 ( 67 % )
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 / 4 ( 0 % )
Total GXB Receiver Channel PMA : 0 / 4 ( 0 % )
Total GXB Transmitter Channel PCS : 0 / 4 ( 0 % )
Total GXB Transmitter Channel PMA : 0 / 4 ( 0 % )
Total PLLs : 0 / 4 ( 0 % )
Total DLLs : 0 / 2 ( 0 % )
```