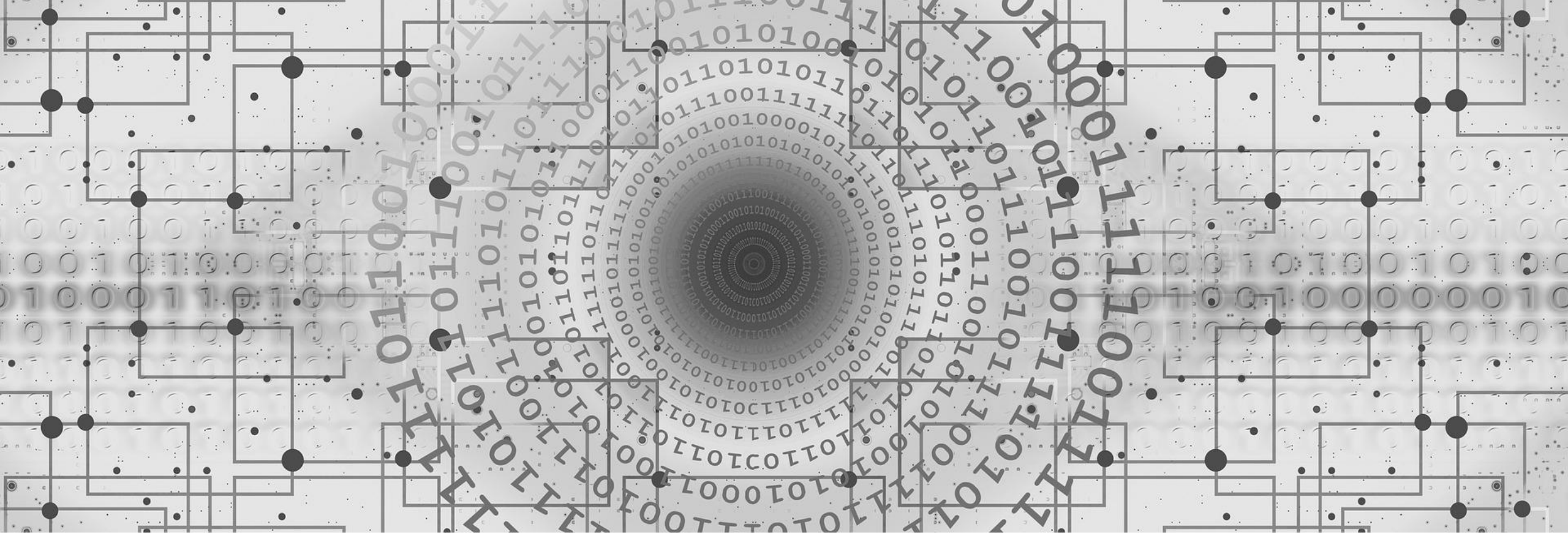


Final Project Part-1 : SHA256

ECE-111 Advanced Digital Design Project

Vishal Karna

UC San Diego
JACOBS SCHOOL OF ENGINEERING
Electrical and Computer Engineering

Summer 2025

Final Project Guidelines

- **Final Project Includes:**
  - **Part-1 :** Develop SHA256 RTL model
  - **Part-2 :** Develop bitcoin hashing RTL model using SHA256 hash function
    - **2a :** Serial Implementation
    - **2b :** Parallel Implementation

- **Testbench will be provided for both Part-1 and Part-2:**
  - Expected behavior of SHA256 and bitcoin model will be implemented in testbench
  - If RTL model does not generate correct hash value, then testbench will generate failure message otherwise it will generate success messages.
  - Students have to ensure RTL models developed work as per the expectations

Final Project Guidelines

❑ Final Project Submission :

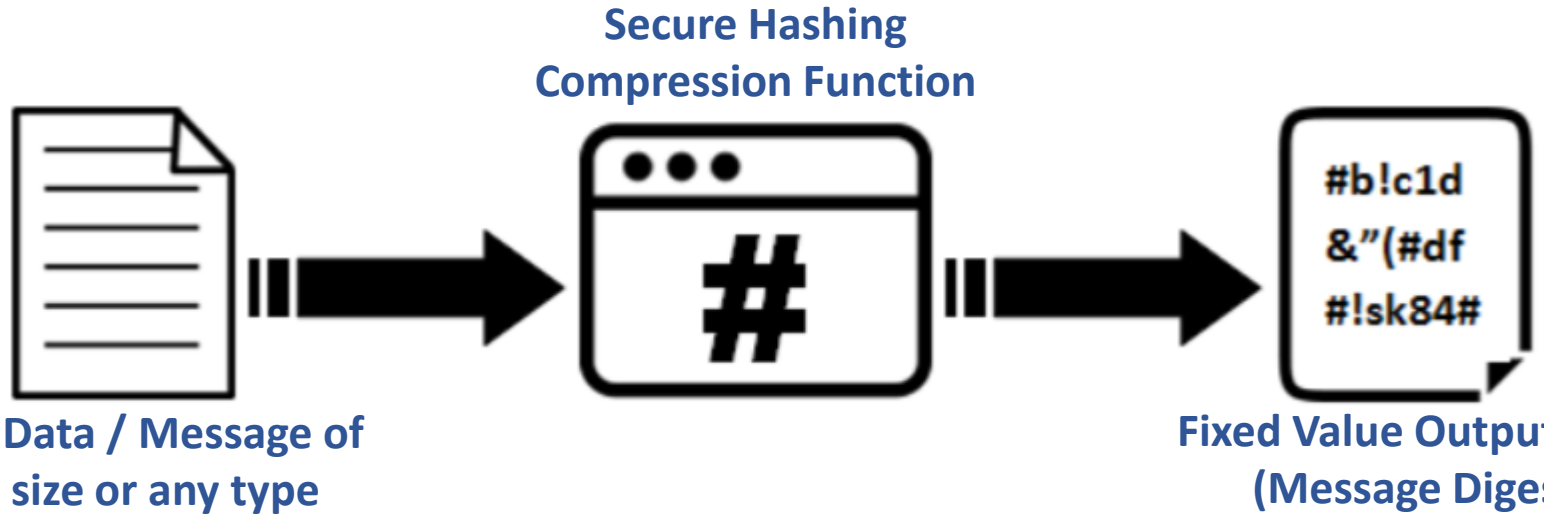
- Due date is **September 7th, 2025 by 11.59 PM**
- Final report should include both **SHA256** and **Bitcoin** model implementation results and details
- **Note :**
  - Specifics details on what should final project submission include will be provided to students
  - Specific files along with project report will be required to be submitted along with report.
    - List of all files will be provided to students
  - Project goals such as speed, area, timing, etc will be provided to students

❑ Project Discussion Sessions :

- Teaching staff (TA) will conduct project discussion sessions and office hours
- Students can reach out to TA and Tutor to help with final project either in their office hours and/or through 1-1 zoom sessions. Students should request for such 1-1 session if required.
- Specific project discussion sessions conducted by instructor will be announced on piazza

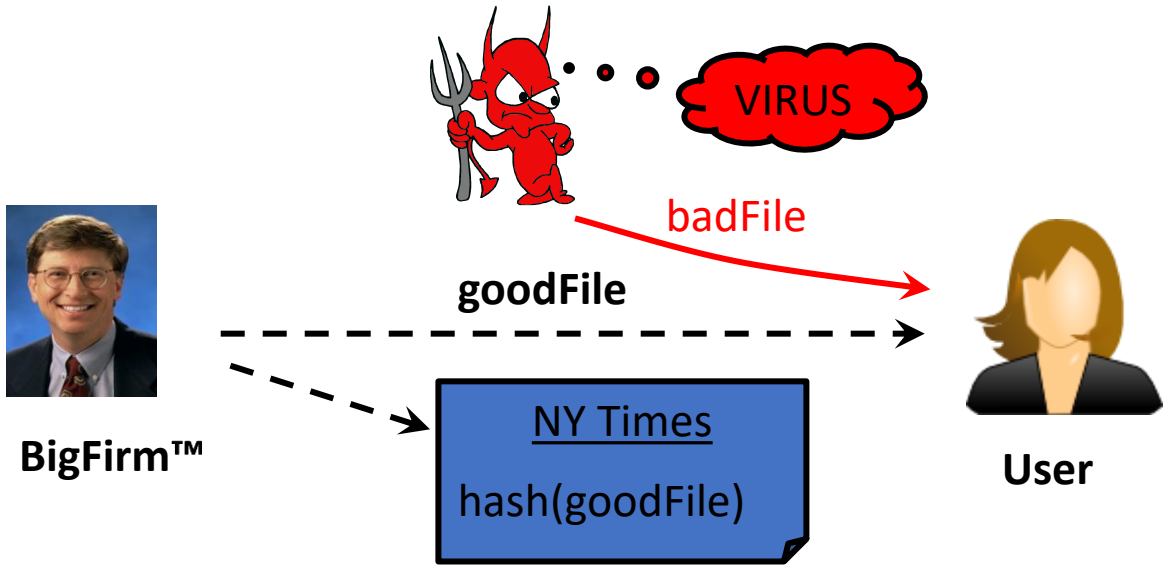
What is Secure Hash Algorithm (SHA256) ?

- **SHA stands for "Secure Hash Algorithm"**
  - It is a cryptographic method of converting input data of any kind and size, into a string of fixed number of characters



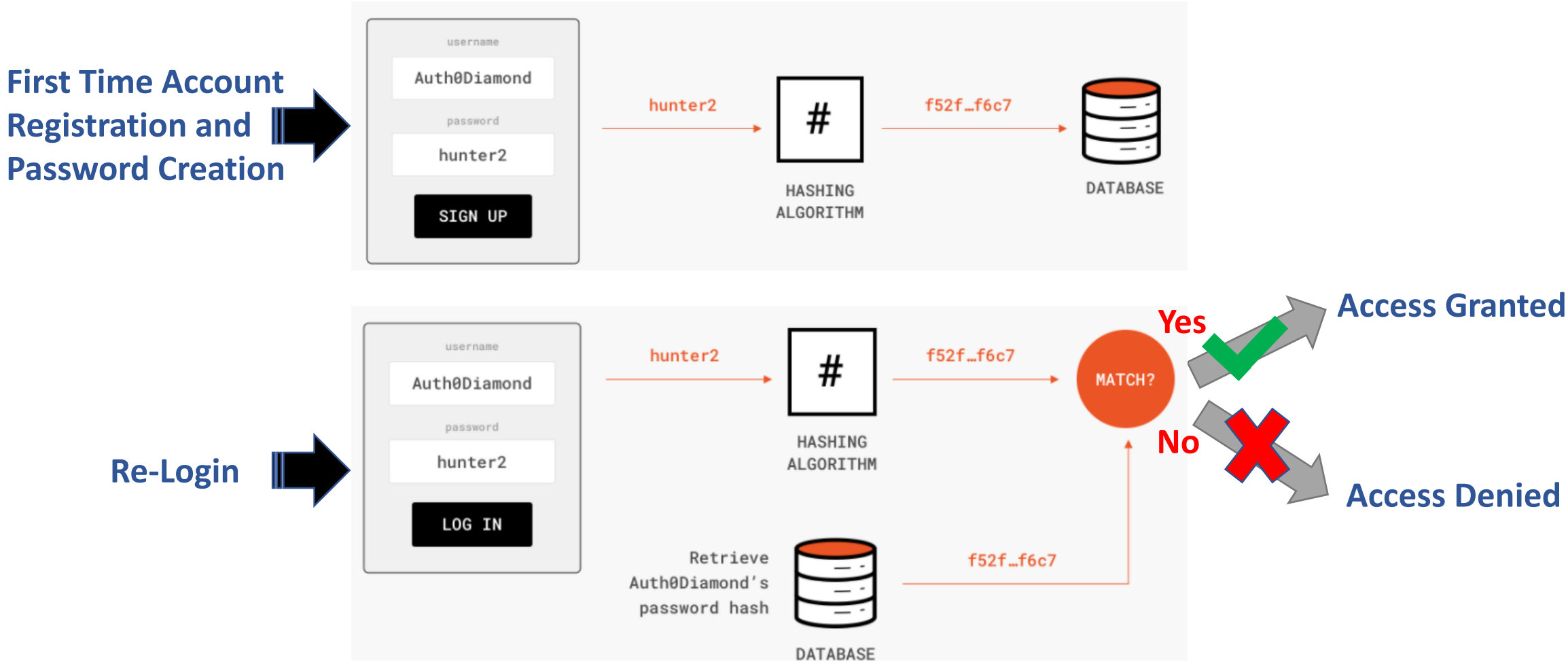
- **Goal is to compute a unique hash value for any input data or message**
- **No matter the size of the input, the output is the fixed size hash value (a.k.a) message digest**
- **There are multiple SHA Algorithms**
  - **SHA-1** : Input message up to <2^64 bits produces **160-bit** output hash value (**a.k.a** message digest)
  - **SHA-2** : Input message up to 2^64 bits produces **256-bit** output hash value
  - **SHA-3** : Input message of 2^128 bits produces **512-bit** output hash value

Applications of SHA256 : Verifying File Integrity



- Software manufacturer wants to ensure that the executable file is received by users without modification
- Sends out the file to users and publishes its hash in NY Times
- The goal is integrity, not secrecy
- **Idea: given goodFile and hash(goodFile), very hard to find badFile such that hash(goodFile)=hash(badFile)**

Applications of SHA256 : Storing and Validating Password



- Instead of storing password directly, password is stored in database as a hash value.
- When user enters the password, first hash is created from the password and then hash value is checked against the originally stored hash value before granting the access

Applications of SHA256

- Digital Signatures and certificates
- Password storage and verification
- Data integrity and verification
- Blockchain technology and cryptocurrencies
  - Verification during Minning process
  - Ensure blockchain is immutable
- Software updates and distribution
- User Authentication and Session management
  - Create Secure login credentials
  - Generate secure tokens for sessions management

and more….

What is Secure Hash Algorithm (SHA256) ?

- In SHA-256 messages up to 2^64 bits (2.3 billion gigabytes) are transformed into 256-bit digest

Orange

SHA-256

78e7771b8b46e11ddb34ba48887e1330525215f96d94778980d1186e6f09f6b4

4f 72 61 6e 67 65
12-bit Hexadecimal Input Message

Fixed 256-bit Value Digest (Output Hash Value)

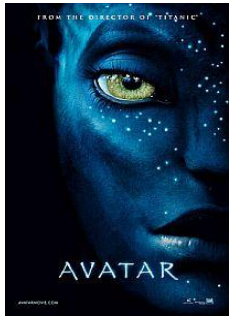
The quick brown fox jumps over the lazy dog

SHA-256

c03905fcdab297513a620ec81ed46ca44ddb62d41cbbd83eb4a5a3592be26a69

88-bit Hexadecimal Input Message

Fixed 256-bit Value Digest (Output Hash Value)



SHA-256

017fa150d2b079a9fee41169cb09142f2b91badbbbf311797e90982b8dc1bf14

32KB AVATAR.JPG Image File

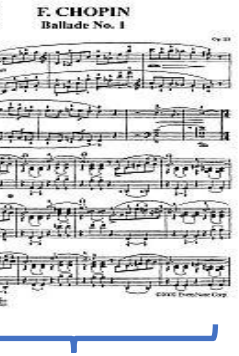
Fixed 256-bit Value Digest (Output Hash Value)

SHA256 Properties

- **Cryptographic hashing function needs to have certain properties in order to be completely secured. These are :**
  - Compression
  - Avalanche Effect
  - Determinism
  - Pre-Image Resistant (One Way Function)
  - Collision Resistance
  - Efficient (Quick Computation)

Compression

- **Output hash should be a fixed number of characters, regardless of the size of the input message !**



a

2-bit Hexadecimal Value

5 MB MP3 File

SHA-256

SHA-256

a7da489976d0047490617adb4f7a1f27f7af8b52a5176fd002ffe471863520ab

Fixed 256-bit Value Digest (Output Hash Value)

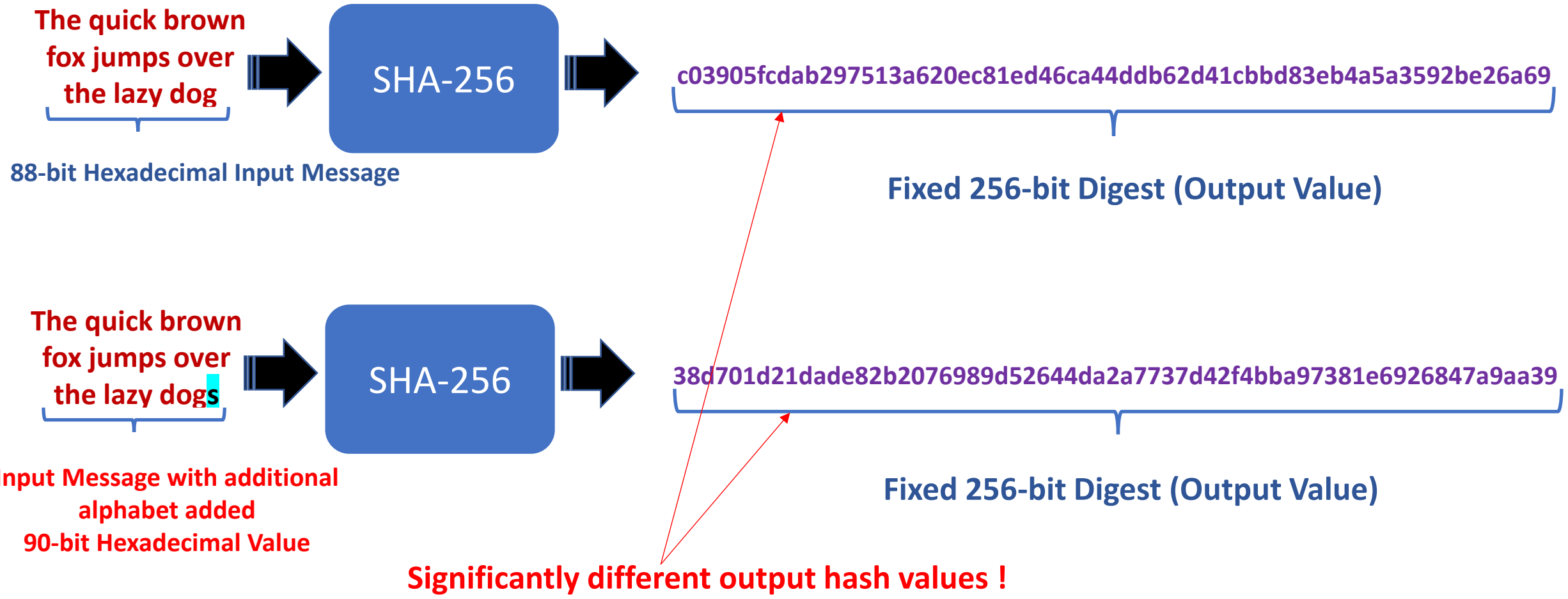
d9904266f642d4659769bba49eb8552e2d53ef09b1faabd8f72154530ba16fc8

Fixed 256-bit Value Digest (Output Hash Value)

For input message of 2-bit value or 5 MB MP3 File, output hash value is fixed size of 256-bit value

Avalanche Effect

- **A minimal change in the input change the output hash value dramatically !**
  - This is helpful to prevent hacker to predict output hash value by trial and error method



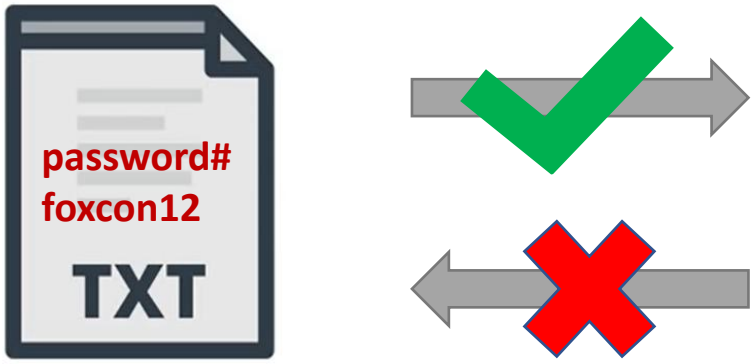
Determinism

- **Same input must always generate the same output by different systems**
  - Any machine in the world which understands hashing algorithm should be able to generate same output hash value for a same input message

Same output hash value generated from same Input message by any machine which runs same secure hashing algorithm !

Pre-Image Resistant (One-Way) And Efficient

- **Secure hashing algorithm should be a One-Way function**
  - No algorithm to reverse the hashing process to retrieve the original input message
    - **Only way is trial and error method, to try each possible input combination to find matching hash value. Not practical !**
  - If input message can be retrieved from output hash value then the whole concept will fail !

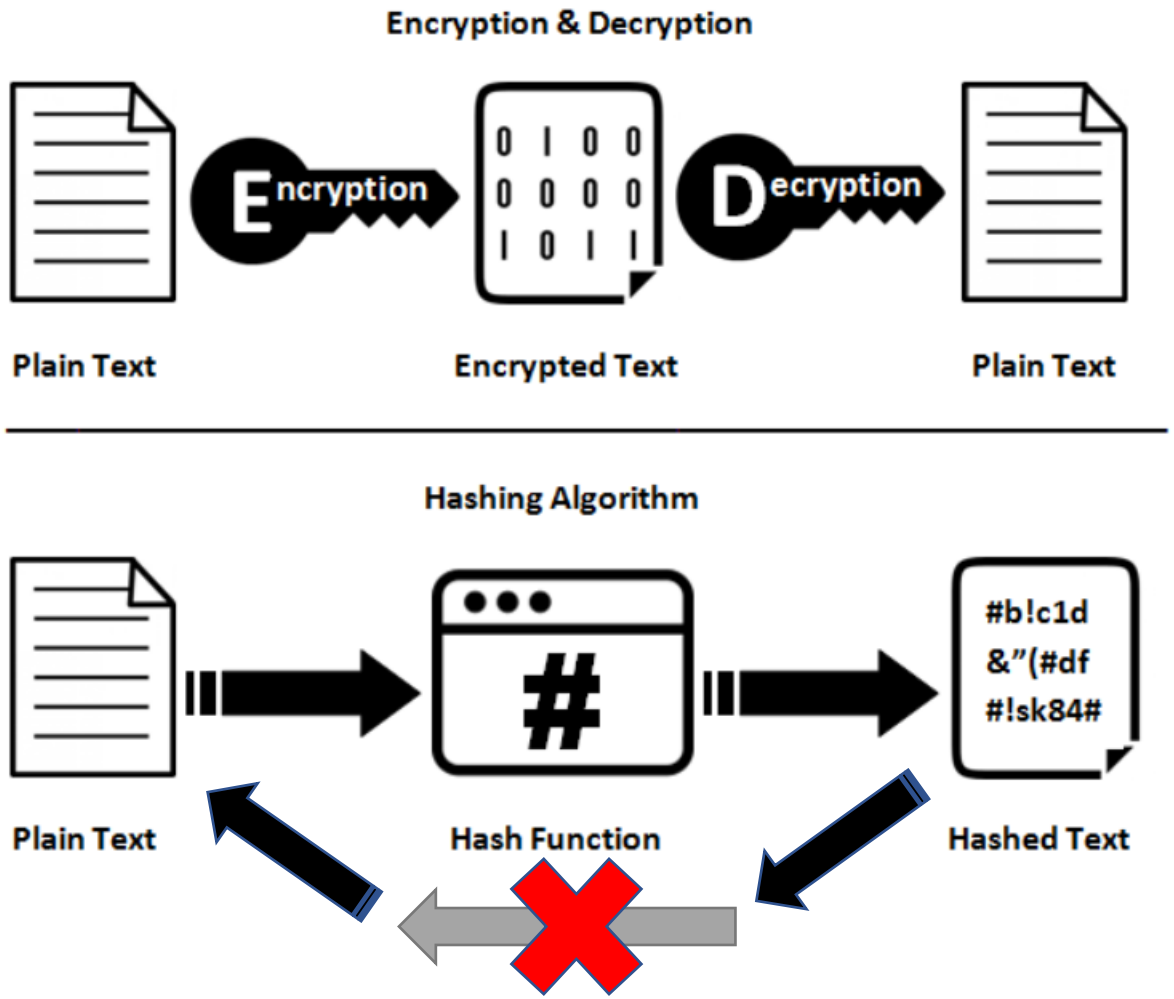


db2c26da2750dea1add7d7677c22d6dc
b6dc4e2674357c82c39bb96d563f0578

- **Efficient : Creating the output hash should be a fast process that doesn't make heavy use of computing power**
  - Should not need supercomputers or high-end machines to generate hash !
  - More feasible for usage !

Hashing vs Encryption

- Encryption is reversible as original message can be retrieved but Hashing has to be irreversible !

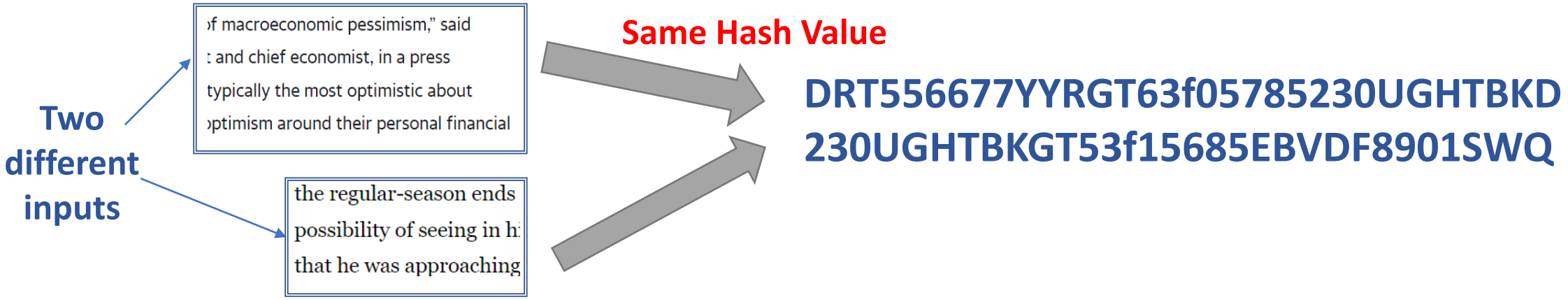


Two-Way Function

One-Way Function

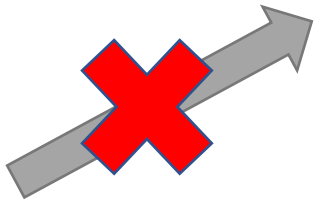
Collision Resistance

- **Hashing Function suffers from the same birthday problem**
  - **What is a birthday problem !**
    - Two people can share same birthday as there are **365 days** in a year and there are **7.7+ billion** human beings on earth as of year 2020
      - **Tyron's** birthday is on June 1 → **152** (day of the year)
      - **Jenny's** birthday is on December 31 → **365** (day of the year)
      - **Sasha's** birthday is on June 1 → **152** (day of the year) **– shares Birthday hash 152 with Tyron**
  - **In rare occasions hashing may produce hash collision ! – Similar to Birthday Hash Problem**
    - Since input can be any large combination values and output is smaller fixed value, so it is theoretically possible to find two input messages having same output hash value



Collision Resistance

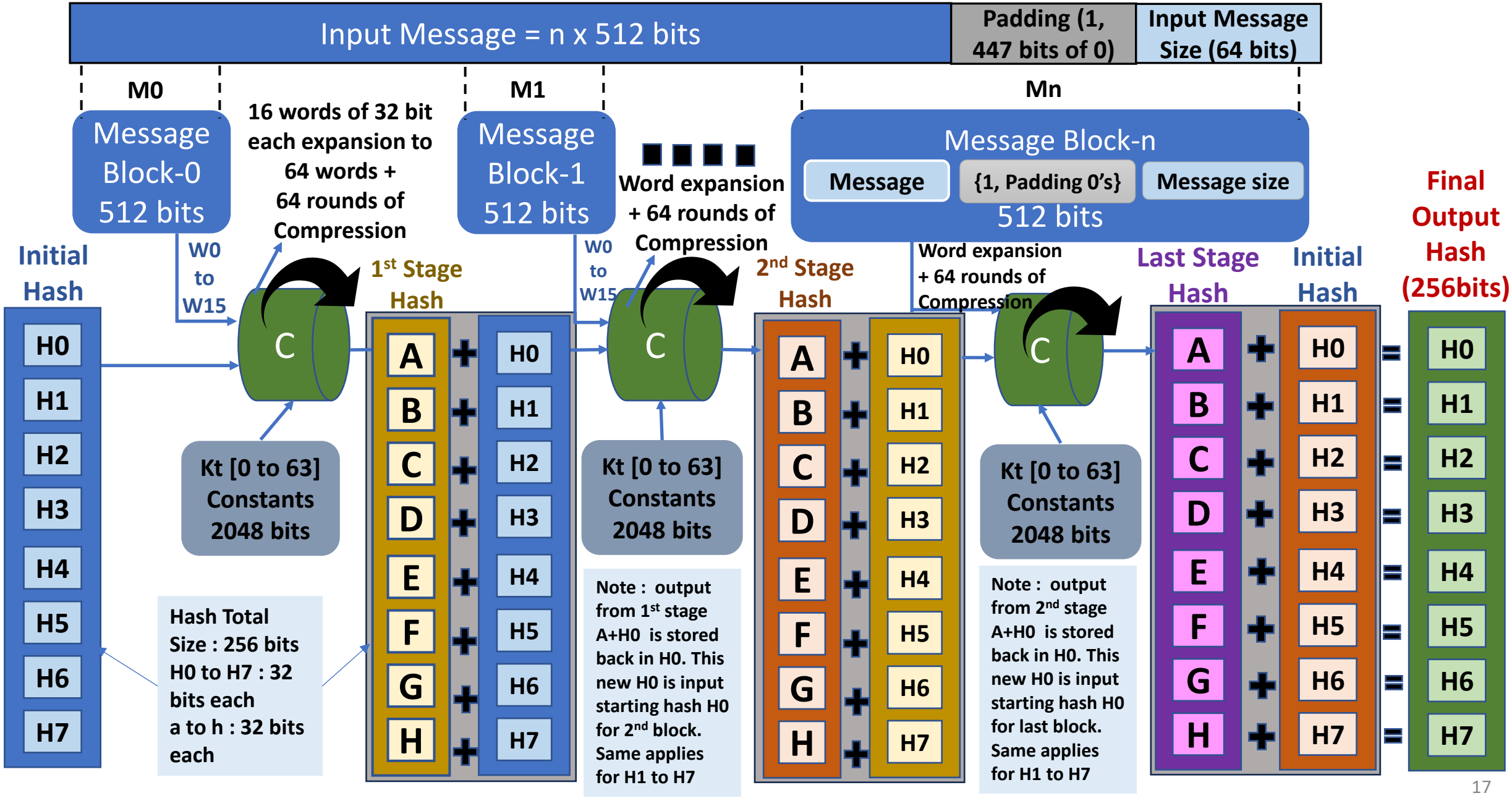
- **Hashing algorithm should be rigorous and it must withstand collision !**
  - Hackers may take advantage of hash collision
  - To avoid hash collision, the output length of the hash value can be large enough so that birthday attack becomes computationally infeasible
    - **Example :** Document hashed with SHA512 is more robust compared to SHA256 as possibility of two inputs to generate same long hash value is almost none !



Db2c26da2750dea1add7d7677c22d6dc
b6dc4e2674357c82c39bb96d563f0578
78FGHWQ23409J5639bb96d77752190
8789VBDWTROPUTGHIKLMNOPTT891

Two different input blocks with same output hash value should be practically impossible even though theoretically possible !

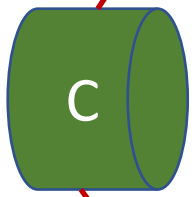
SHA256 Algorithm



SHA256 Algorithm

Compression Function includes two steps :

1. Word Expansion 16 message words expanded to 64 words
2. SHA256 operation

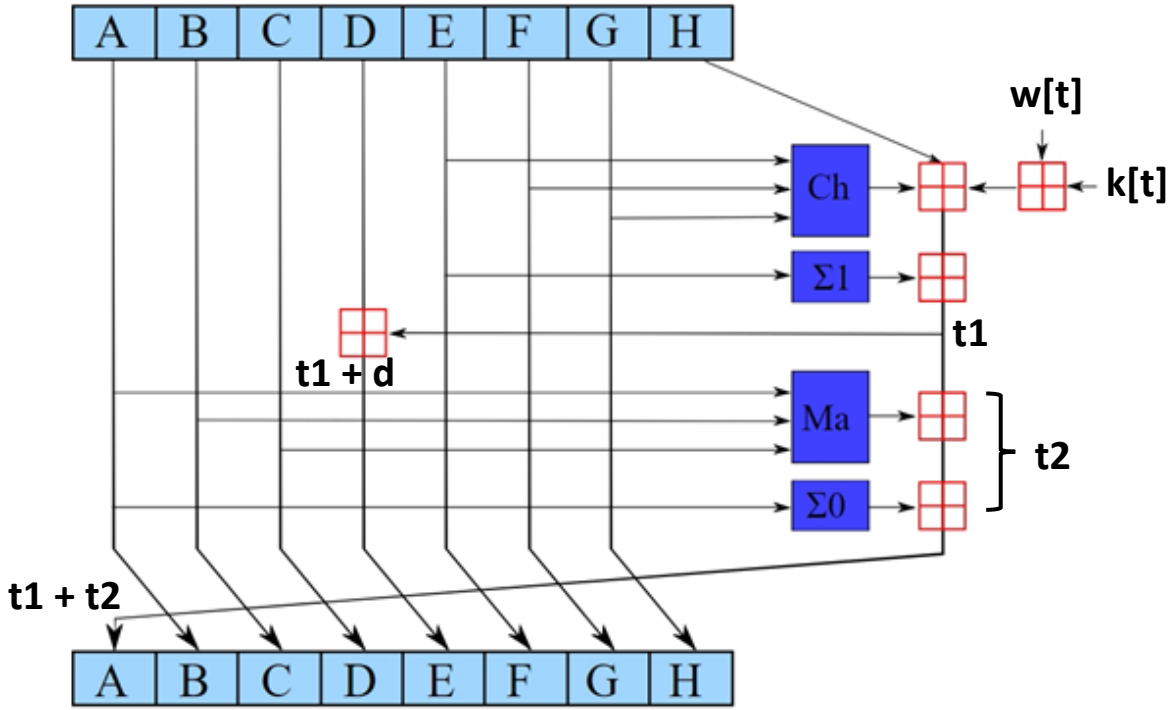


Step 1: Word Expansion

```
for (t = 0; t < 64; t++) begin
    if (t < 16) begin
        w[t] = dpsram_tb[t]; // Get Input Message 512-bit block and store in Wt array
    end else begin
        s0 = rightrotate(w[t-15], 7) ^ rightrotate(w[t-15], 18) ^ (w[t-15] >> 3);
        s1 = rightrotate(w[t-2], 17) ^ rightrotate(w[t-2], 19) ^ (w[t-2] >> 10);
        w[t] = w[t-16] + s0 + w[t-7] + s1;
    end
end
```

Step 2: SHA256 Operation

Performed 64 times t = 0 to 63



SHA256 Algorithm

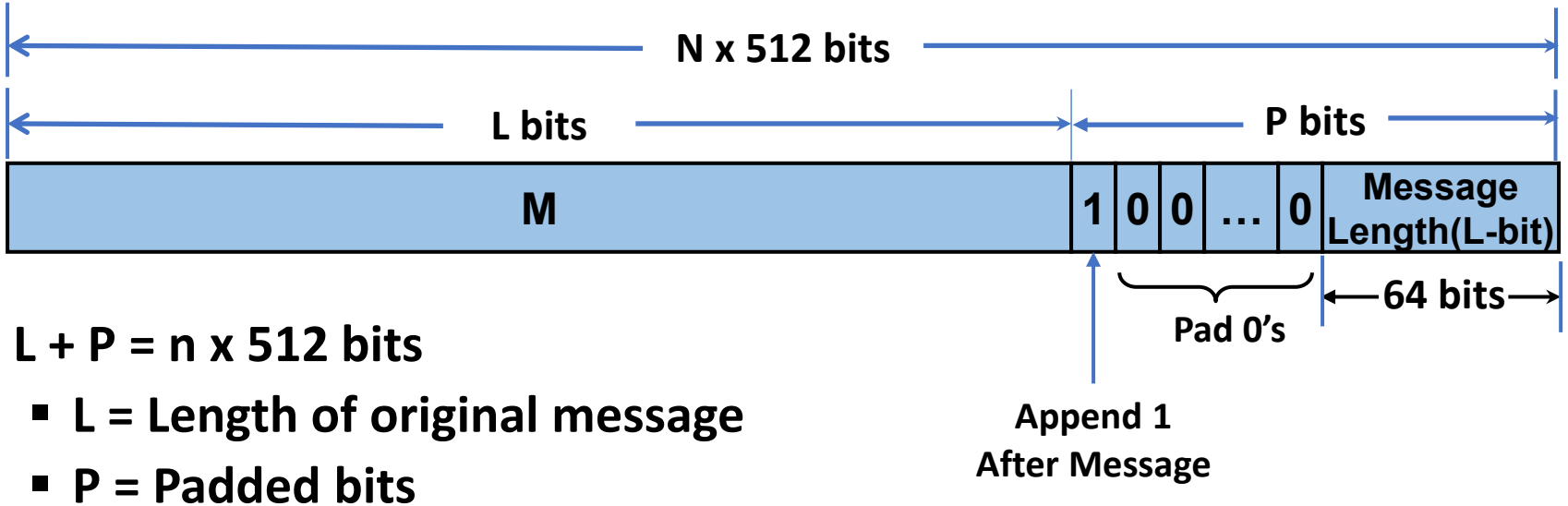
General Assumptions

- Input message must be <=2^{64} bits
- Message is processed in 512-bit blocks sequentially
- Message digest (output hash value) is 256 bits

SHA256 Algorithm

- **Step 1:** Append padding bits (1 and 0's)
  - A **L**-bits of message **M** is padded in the following manner:
    - Add a single "1" to the end of **M**
    - Then pad message with "0's" until the length of message is congruent to 448, modulo 512 (which means pad with 0's until message bits is 64-bits less than 512 bits).

- **Step 2** : Append message length bits (64 bits reserved to store size of the message)
  - Since SHA256 supports until 2^64 input message size, 64 bits are required to append message length

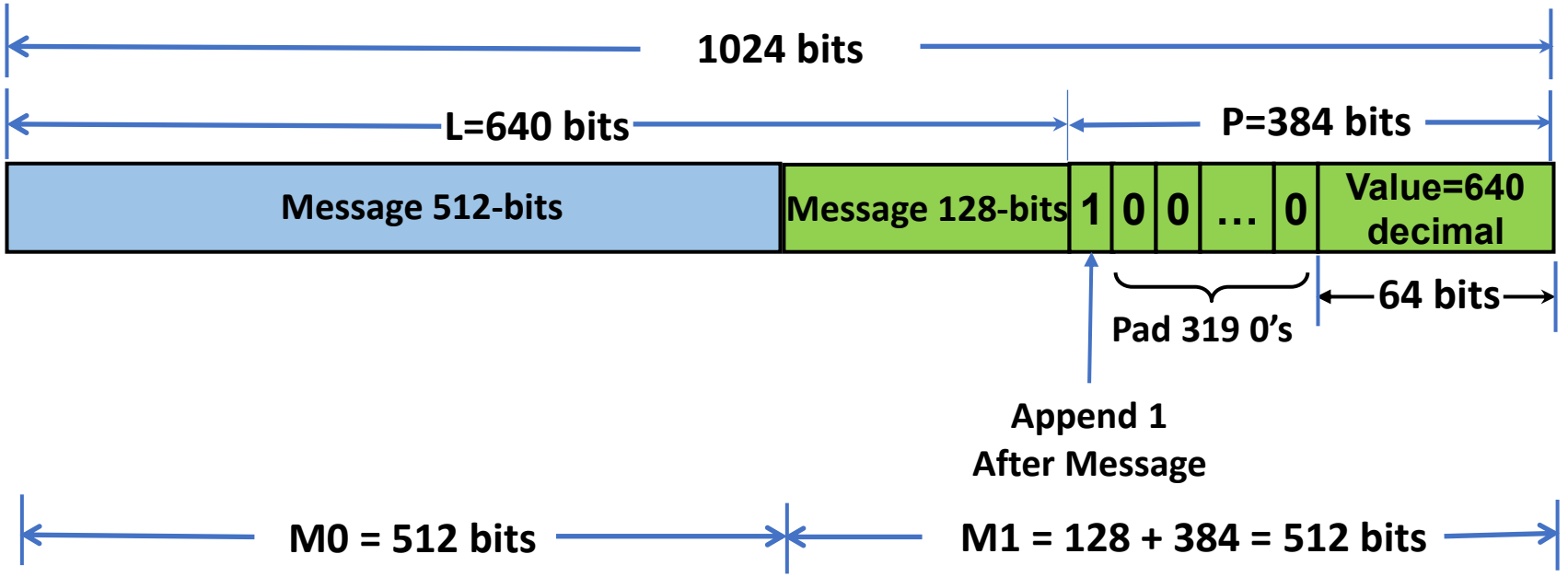


L + P = n x 512 bits

- **L = Length of original message**
- **P = Padded bits**

SHA256 Algorithm

- **Example :** Lets say, original Message is **L** = 640 bits
  - Since message blocks have minimum 512 chunks, to fit original message of 640 bits in 512 bits chunks, it would require 2 message blocks **(n = 2)**
    - **M0** (first block) Size = 512 bits (no padding required)
    - **M1** (second block) Size = 512 bits after padding
      - **512 bits** = 128 bits of original message + 1 bit for appending '1' + 319 bits of 0's + 64 bit message length
    - **Message length**=decimal value 640 stored in 0 to 63 bits



SHA256 Algorithm

- **FAQ : What happens if the total message size is exact multiple of 512 bits :**
  - **Example :** Let's say, original Message is **L** = 1024 bits
  - Since message blocks have minimum 512 chunks, to fit original message of 1024 bits in 512 bits chunks, it would require 2 message blocks **+ 1 additional block for padding and message length**
    - **M0** (first block) Size = 512 bits (no padding required)
    - **M1** (second block) Size = 512 bits (no padding required)
    - **M2** (third block) Size =
      - **512 bits** = 1 bit for appending '1' + 447 bits of 0's + 64-bit message length
    - **Message length**=decimal value "1024" stored in 64-bit of message length bits

Block#	Block Content
Block-1 (M0)	Message[511:0] → First chunk of 512 bits message
Block-2 (M1)	Message[1023:512] → Second chunk of 512 bits message
Block-3 (M3)	1, + 447 0's + 64-bit length field → Padding bits and message length

Note : SHA256 Final Project Design involves 640 bit of total message bits and not 1024. Hence information on this slide is only for information purpose

SHA256 Algorithm

- **Step 3 :** Buffer Initialization
  - Initialize message digest (MD) buffers / output hash to these 8 32-bit words

    H0 = 6a09e667
    H1 = bb67ae85
    H2 = 3c6ef372
    H3 = a54ff53a
    H4 = 510e527f
    H5 = 9b05688c
    H6 = 1f83d9ab
    H7 = 5be0cd19

SHA256 Algorithm

❑ FAQ : How is the initial value of each H0 to H7 computed ?

- **Each initial hash value is a 32-bit word is obtained by:**
  - Taking the square root of the first 8 prime numbers:
    - 2, 3, 5, 7, 11, 13, 17, 19
  - Extracting the fractional part of each square root (i.e., the part after the decimal)
  - Converting that fractional part into binary
  - Taking the first 32 bits of that binary fraction
  - Interpreting those bits as a 32-bit word in hexadecimal

Hash	Prime Number	Square Root of Prime	Fractional Part of Square Root Value	First 32 Bits of Fractional Part Converted into Binary	Convert Binary to Hex Value
H0	2	1.41421356	0.41421356	01101010000010011110011001100111	6a09e667
H1	3	1.7320508	0.7320508	10111011011001111010111010000101	bb67ae85
H2	5	2.23606797	0.23606797	00111100011011011011001101110010	3c6ef372
H3	7	2.64575131	0.64575131	10100101010011111111010100111010	a54ff53a
H4	11	3.31662479	0.31662479	01010001000011100101001001111111	510e527f
H5	13	3.60555127	0.60555127	10011011000001010110100010001100	9b05688c
H6	17	4.12310562	0.12310562	00011111100000111101100110101011	1f83d9ab
H7	19	4.35889894	0.35889894	01011011111000001100110100011001	5be0cd19

Note : Above mentioned is for information only. Hence when designing SHA256 FSM, use directly the H0 to H7 hexadecimal values

SHA256 Algorithm

❑ FAQ : Why initial hash values (H0 to H7) matter in SHA-256

- They seed the compression Function and these values act as the starting point for the iterative compression rounds.
- They Ensure Deterministic Output
  - Given the same input, SHA-256 always produces the same output — thanks to fixed initial values.
- They're Derived from Mathematical Constants
  - Each H0–H7 is the first 32 bits of the fractional part of the square root of the first 8 prime numbers (2–19).
  - This method avoids arbitrary or biased constants, reducing the risk of hidden vulnerabilities.
- They Anchor the Algorithm's Security Properties
  - SHA-256's resistance to collision and pre-image attacks depends on its internal structure.
  - The initial hash values are part of that structure — altering them could compromise security
- They Enable Reproducibility Across Platforms
  - Every SHA-256 implementation (hardware or software) uses the same H0–H7 values.
  - This ensures interoperability and standardization across systems.

SHA256 Algorithm

- **Step 4 :** Processing of the message (algorithm)
  - Divide message M into 512-bit blocks, M_0, M_1, … M_j, …
  - Process each M_j sequentially, one after the other
  - Input:
    - W_t: a 32-bit word from the message
    - K_t : a constant array
    - H0, H1, H2, H3, H4, H5, H6, H7 : current MD (Message Digest)
  - Output:
    - H0, H1, H2, H3, H4, H5, H6, H7 : new MD (Message Digest)

SHA256 Algorithm

- **Step 4 :** Cont'd
  - At the beginning of processing each message block M_j, initialize (A, B, C, D, E, F, G, H) = (H0, H1, H2, H3, H4, H5, H6, H7)
  - Then 64 processing rounds of 512-bit blocks
  - Each step t ($0 \le t \le 63$): Word expansion for W_t
    - If $t < 16$
      - W_t = t^{th} 32-bit word of block M_j
    - If $16 \le t \le 63$
      - $s_0 = (W_{t-15}$ **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)
      - $W_t = W_{t-16} + s_0 + W_{t-7} + s_1$

SHA256 Algorithm

- **Step 4:** Cont'd
  - SHA-256 uses 64 K_t constants, labeled K_0 through K_{63}, one for each round of its compression function

K [0..63] = 0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5, 0xd807aa98,
0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe,
0x9bdc06a7, 0xc19bf174, 0xe49b69c1, 0xefbe4786, 0x0fc19dc6,
0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3,
0xd5a79147, 0x06ca6351, 0x14292967, 0x27b70a85, 0x2e1b2138,
0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e,
0x92722c85, 0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3,
0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070, 0x19a4c116,
0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a,
0x5b9cca4f, 0x682e6ff3, 0x748f82ee, 0x78a5636f, 0x84c87814,
0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2

Note :

- K[0] to K[63] values are derived from the fractional parts of the cube roots of the first 64 prime numbers (2 through 311)
  - ✓ Similar, to how H0 to H7 initial values are computed
- Significance of K_t constants
  - ✓ Introduces nonlinearity : Each round of SHA-256 uses a different K_t value. This variation prevents predictable patterns and strengthens resistance to cryptanalysis.
  - ✓ K_t constants mix with message schedule words (W_t) to produce unique transformations in each round.
  - ✓ They are mathematically derived to avoid suspicion of hidden backdoors.
  - ✓ They ensure deterministic yet secure hashing : Guarantees consistent output while maintaining cryptographic strength.

SHA256 Algorithm

- **Step 4 :** Cont'd
  - Each step t ($0 \le t \le 63$):

Σ0 S_0 = (A **rightrotate** 2) **xor** (A **rightrotate** 13) **xor** (A **rightrotate** 22)

Ma maj = (A **and** B) **xor** (A **and** C) **xor** (B **and** C)

$t_2 = S_0 + maj$

Σ1 S_1 = (E **rightrotate** 6) **xor** (E **rightrotate** 11) **xor** (E **rightrotate** 25)

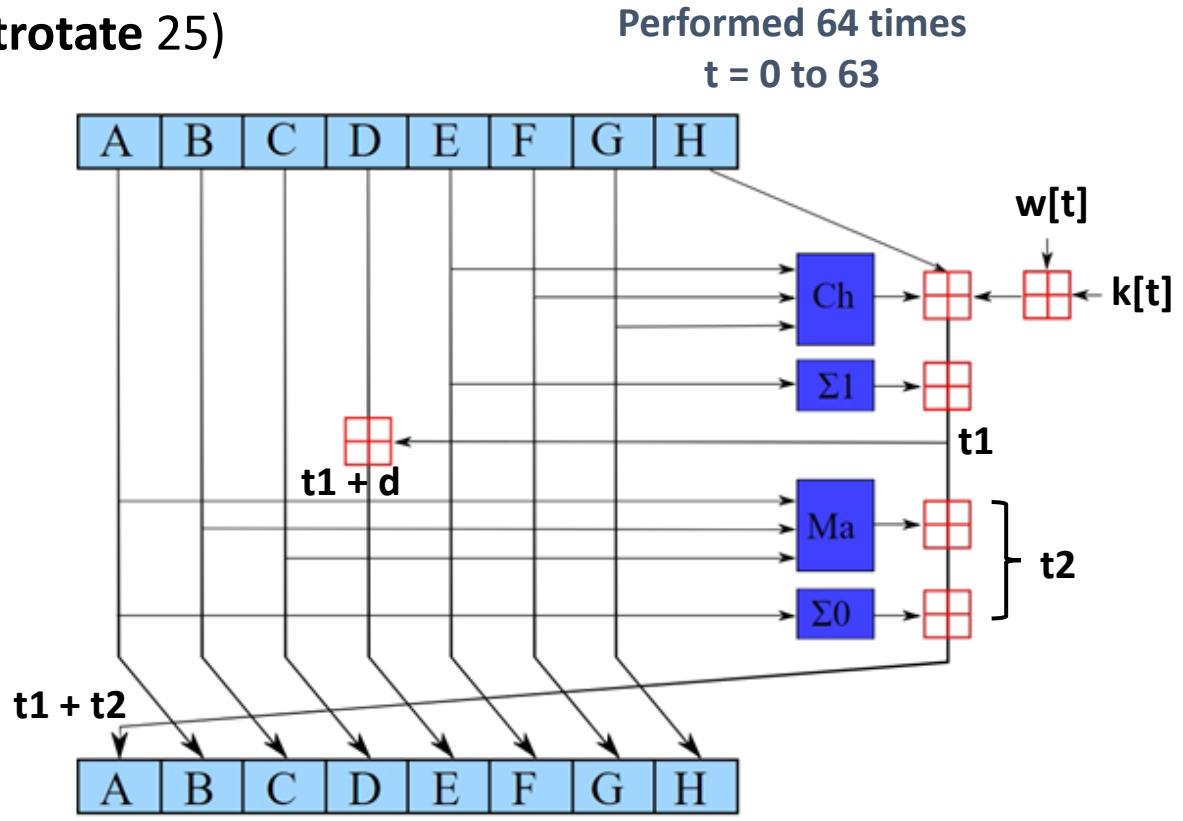
Ch ch = (E **and** F) **xor** ((**not** E) **and** G)

$t_1 = H + S_1 + ch + K[t] + W[t]$

$(A, B, C, D, E, F, G, H) = (t_1 + t_2, A, B, C, D + t_1, E, F, G)$

Note :

- **Ch** stands for "choose function"
  - **Ch**(E, F, G)[i] = E[i] ? F[i] : G[i] → makes a bitwise multiplexer
  - Enhances bit mixing, non-linearity, security
- **Σ_0**(x) and **Σ_1**(x) are Summation Functions
  - They are designed to introduce nonlinearity, bit diffusion, avalanche behavior during each round of hashing
- **Maj** is the majority function
  - Introduces non-linearity, bit mixing, enhances avalanche effects and enhances cryptographic strength
- **Ch** and **Maj non-linear functions** used in each round is complex enough to prevent attackers from predicting or reversing the hash



SHA256 Algorithm

- **Step 4 :** Cont'd
  - Finally, when all 64 steps have been processed, set

    H0 = H0 + a

    H1 = H1 + b

    H2 = H2 + c

    H3 = H3 + d

    H4 = H4 + e

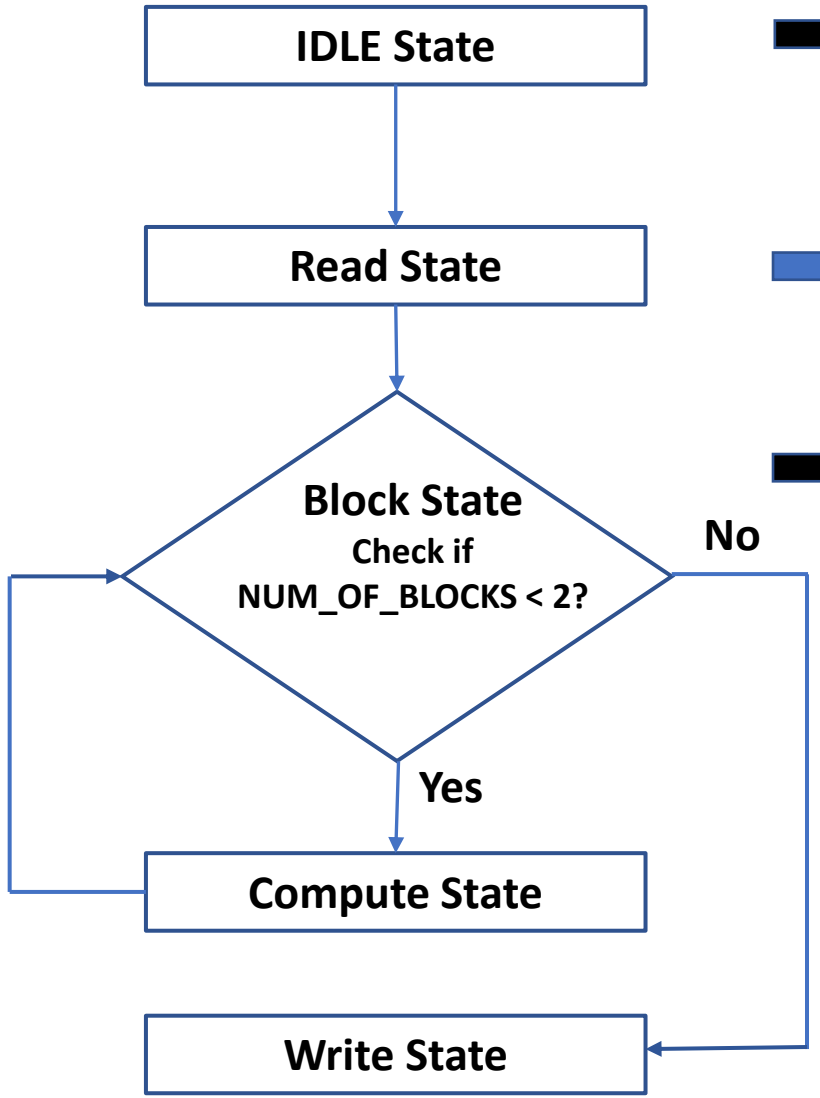
    H5 = H5 + f

    H6 = H6 + g

    H7 = H7 + h

- **Step 5 :** Output
  - When all M_j have been processed, the 256-bit hash of M is available in H0, H1, H2, H3, H4, H5, H6, H7

SHA256 Algorithm Flow Chart for FSM Designing



- If start==1, Initialize h0, h1, h2, h3, h4, h5, h6, h7 to initial hash values
- Initialize a, b, c, d, e, f, g, h
- Initialize write enable to memory, memory address offset, curr_addr to first message location in memory, index variables, etc
- Move to read input message FSM state

- Read 640 bits message from testbench memory in chunks of 32bits words (i.e. read 20 locations from memory by incrementing address offset)
- Move to Block creation FSM state

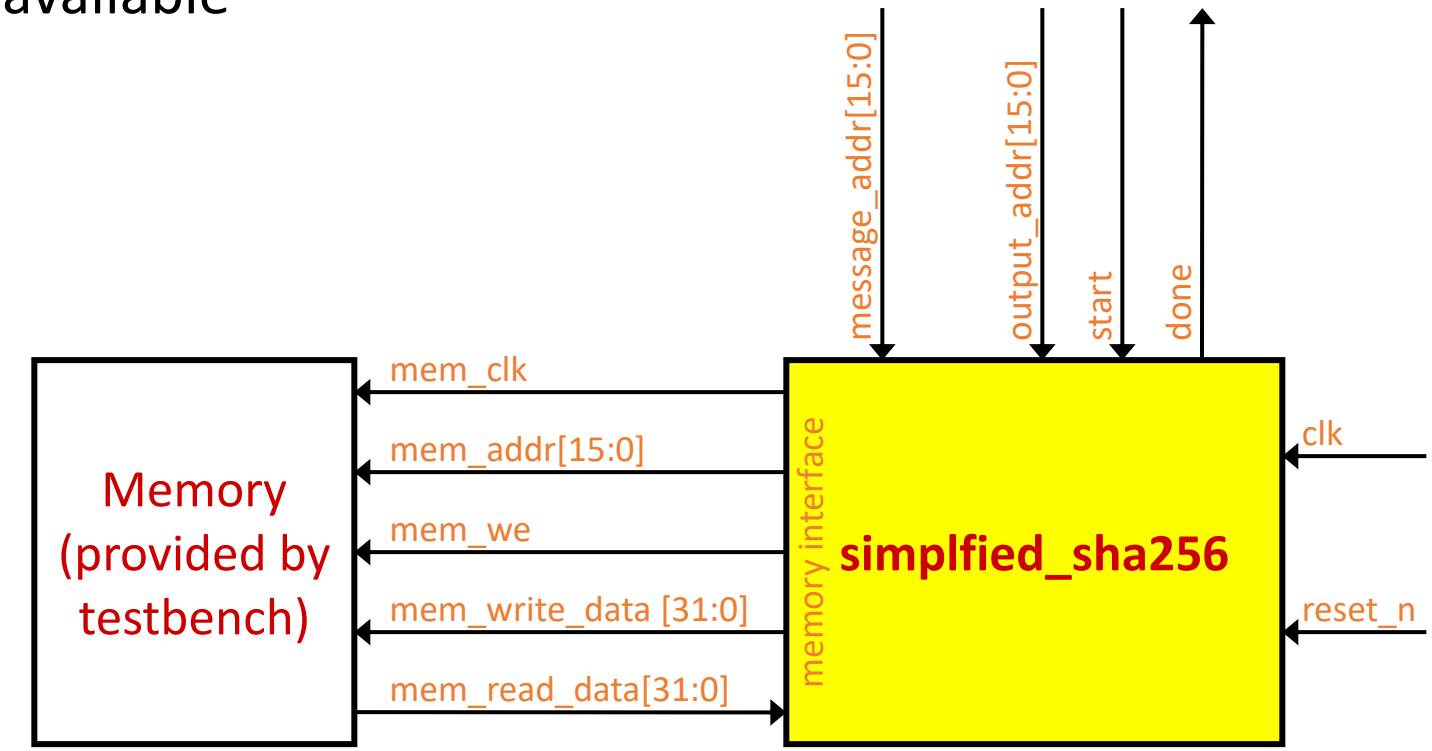
- In Block FSM state, Create two message blocks each of 512 bits
- First message block has first 16 words of input message stored in 'w' array
- Second message block has 4 words of input message, value 1, padding 0 and message size 640
- Assign h0, h1, h2, h3, h4, h5, h6, h7 to a, b, c, d, e, f, g, h respectively
- Check if for both blocks SHA256 operation has been processed and hash is created, if yes then move to WRITE state otherwise move to compute state

- Perform Word expansion of 16 elements of input message block (512 bits) and create total 64 word array each having 32 bits
- Perform 64 rounds of SHA operation to generate hash values in 'a' thru 'h' array. Increment number of blocks iteration variable
- Add previous hash values with 'a' thru 'h' hash values, go back to BLOCK State

- Write 256 bit hash value stored in h0 to h7 hash variables in testbench memory using output_addr as a starting address location
- Set done = 1 and then move to IDLE state

Module Interface

- Wait in idle state for **start**, read message starting at **message_addr** and write final hash **{H0, H1, H2, H3, H4, H5, H6, H7}** in 8 words to memory starting at **output_addr**. **message_addr** and **output_addr** are word addresses.
- Message size is “hardcoded” to 20 words (640 bits).
- Set **done** to 1 when finished.
- Testbench has memory defined named “dpsram[0:16383]” which has all 20 word of input message available



Module Interface

- Write the final hash **{H0, H1, H2, H3, H4, H5, H6, H7}** in 8 words to memory starting at **output_addr** as follows:

```
mem_addr <= output_addr;
mem_write_data <= H0;

mem_addr <= output_addr + 1;
mem_write_data <= H1;

...

mem_addr <= output_addr + 7;
mem_write_data <= H7;
```

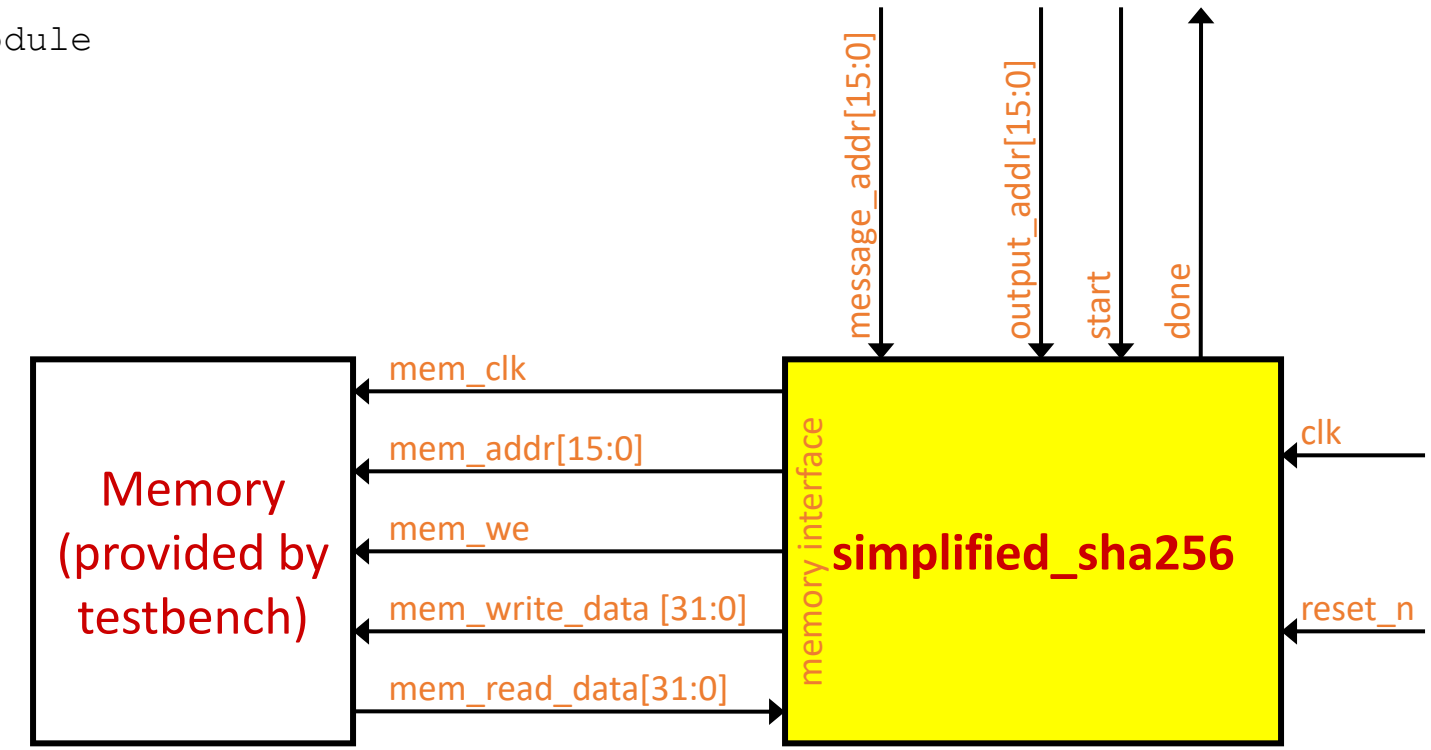
output_addr	H0
output_addr + 1	H1
output_addr + 2	H2
output_addr + 3	H3
output_addr + 4	H4
output_addr + 5	H5
output_addr + 6	H6
output_addr + 7	H7

Module Interface

- Your assignment is to design the yellow box:

```
module simplified_sha256(input logic clk, reset_n, start,
                         input logic [15:0] message_addr, output_addr,
                        output logic done, mem_clk, mem_we,
                        output logic [15:0] mem_addr,
                        output logic [31:0] mem_write_data,
                         input logic [31:0] mem_read_data);

    ...
endmodule
```

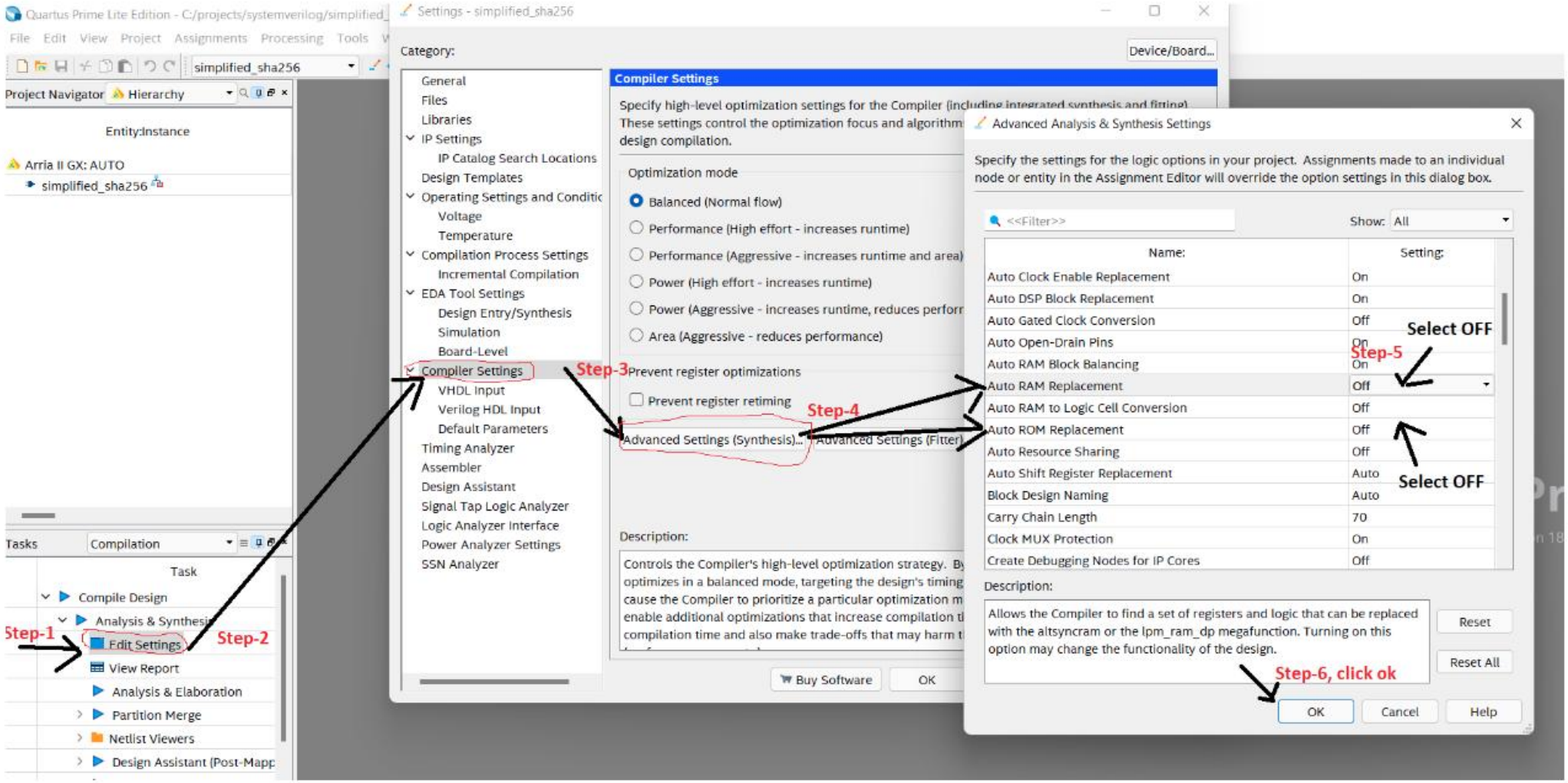


No Inferred Mega functions or Latches

- **In your Quartus compilation message ensure :**
  - **No inferred megafunctions**: Most likely caused by block memories or shift-register replacement. Can turn OFF “Automatic RAM Replacement” and “Automatic Shift Register Replacement” in “Advanced Settings (Synthesis)”. If you still see “inferred mega functions”, contact Professor. Your design will not pass if it has inferred mega functions.
  - **No inferred latches**: Your design will not pass if it has inferred latches.

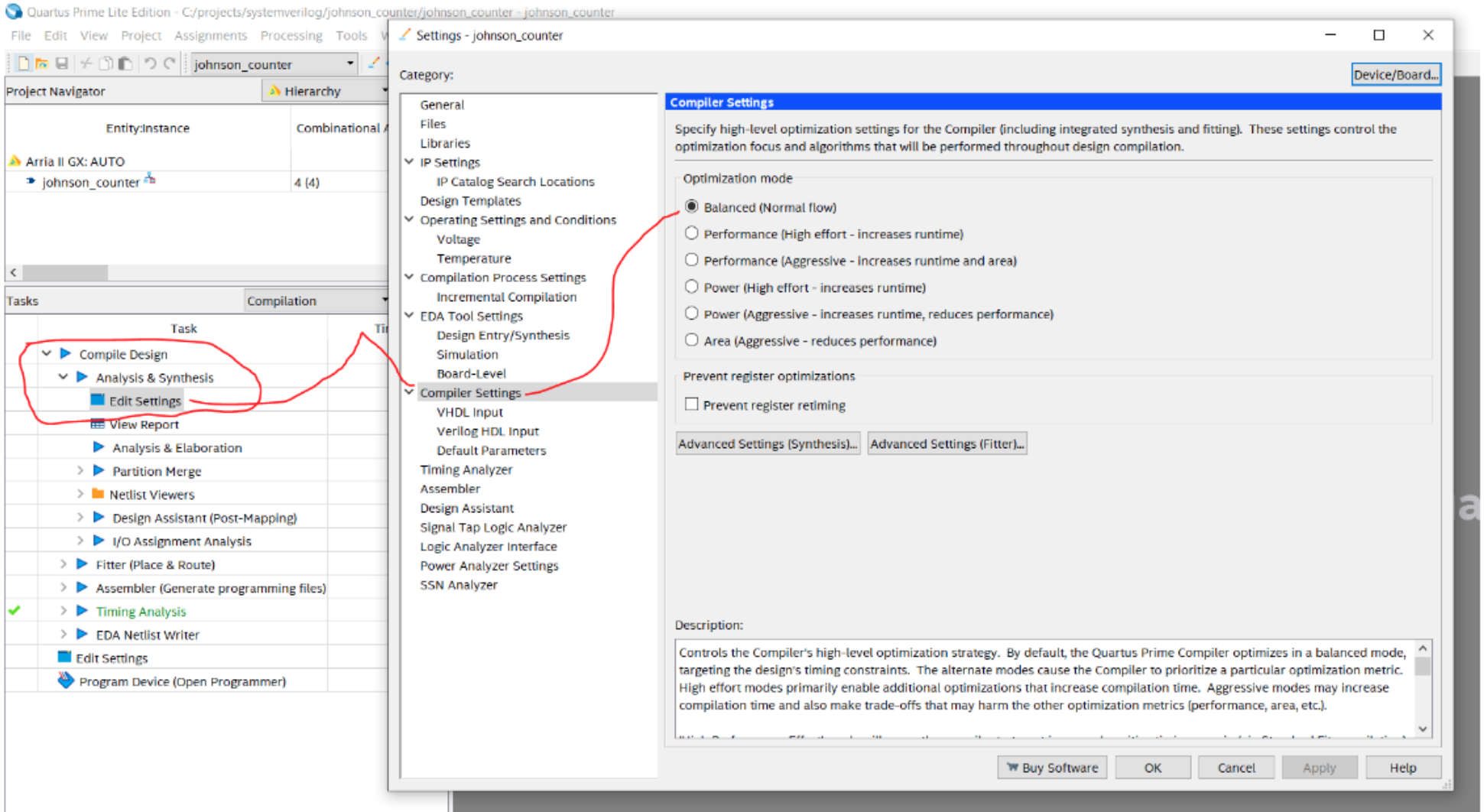
Disable Usage of Embedded Block Memory in FPGA

- Prior to performing synthesis of SHA256 and Bitcoin FSM ensure AUTO RAM and AUTO ROM replacement are set to OFF as shown below :



Setting Quartus Synthesis Compiler to Balanced Mode

- Before running synthesis of SHA256 and Bitcoin in Quartus, ensure synthesis compiler Settings "Optimization Mode" is set as "Balanced". By default, it should be set as Balanced. However, do ensure this. Setting balanced efforts tells Quartus to do synthesis considering area and performance both. See snapshot below



Final Project Submission

- **Put following files into (LastName, FirstName)_(LastName, FirstName)_finalproject.zip folder which should include :**
  - SystemVerilog code design files for both SHA256 and Bitcoin hashing project
  - Modelsim transcript files (i.e. msim_transcript) for both SHA256 and Bitcoin hashing project
  - For both SHA256 and bitcoin hashing provide, fitter and sta files (files with extension .fit, .sta)
  - Report for both SHA256 and Bitcoin hashing project.
    - Merge report in one file or separate either way is fine
    - Report file should be submitted as PDF file
  - Finalsummary.xls file with fmax, number of cycles, aluts, registers detail filled. Template of this file is provided as part of Final_Project.zip folder.
    - **finalsummary.xls should be submitted for both SHA256 and bitcoin hash!**

- **Note :**
  - Each project group to submit only one Final project zip folder on gradescope
  - Ensure final project zip folder and files within it can be opened on windows PC machine

Final Project Submission

- **Final report should be saved in PDF file format and it should include following mentioned :**
  - Explain briefly what SHA-256 is and bitcoin hashing (may use lecture slide contents)
  - Describe algorithm for both SHA-256 and Bitcoin hashing implemented in your code
  - Simulation waveform snapshot for both SHA-256 and Bitcoin hashing
  - Provide modelsim transcript window output indicating passing test results generated from self-checker in testbench for both SHA-256 and Bitcoin hashing
  - Provide synthesis resource usage and timing report for bitcoin_hash only.
    - Should include ALUTs, Registers, Area, Fmax snapshots
    - Provide fitter report snapshot. And provide Timing Fmax report snapshots
    - Make sure to use **Arria II GX EP2AGX45DF29I5** device and use Fmax for **Slow 900mV 100C Mod**

Note : Please make sure not to submit final report, SystemVerilog file and required other files as a .rar file

Fill up finalsummary.xlsx

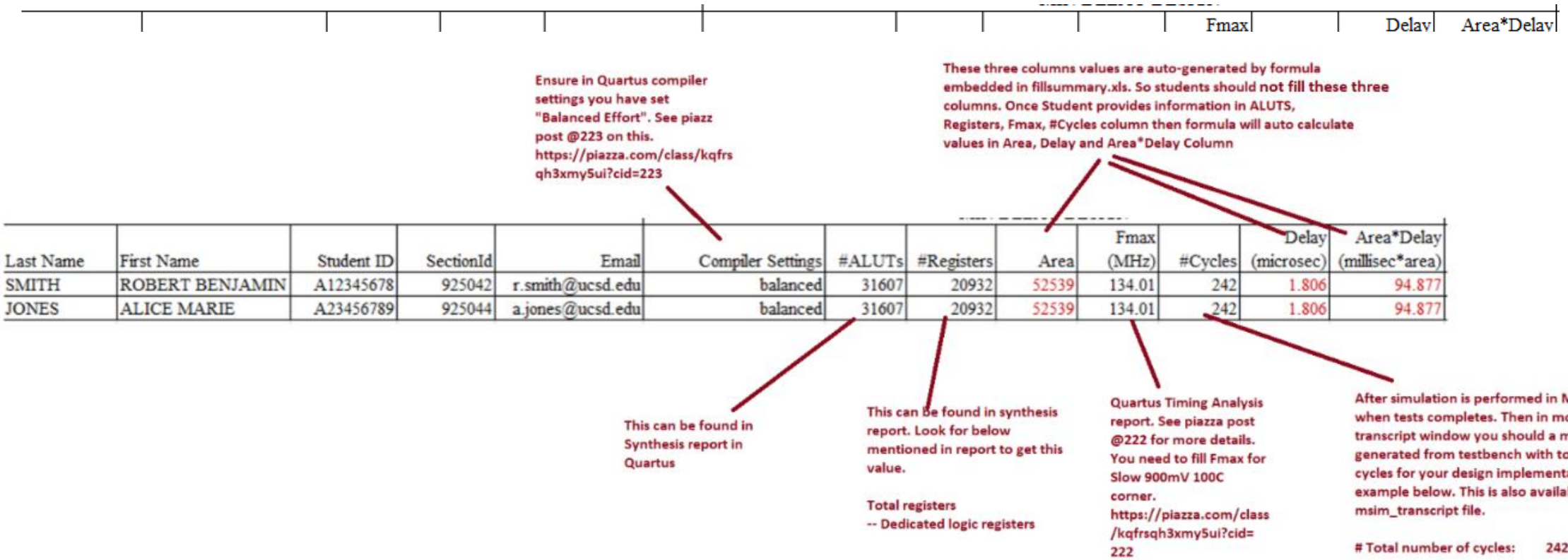
- Fill up finalsummary.xlsx posted on Piazza as part of Final_Project.zip (to be filled for both simplified_sha256 bitcoin_hash project in separate fillsummary.xlsx)

Last Name	First Name	Student ID	SectionId	Email	Compiler Settings	#ALUTs	#Registers	Area	Fmax (MHz)	#Cycles	Delay (microsec)	Area*Delay (millisec*area)
SMITH	ROBERT BENJAMIN	A12345678	925042	r.smith@ucsd.edu	balanced	31607	20932	52539	134.01	242	1.806	94.877
JONES	ALICE MARIE	A23456789	925044	a.jones@ucsd.edu	balanced	31607	20932	52539	134.01	242	1.806	94.877

- If you worked alone, just fill out one row
- Spreadsheet already contains calculation fields: e.g. Area = #ALUTs + #Registers. Please use them.
- Students to fill ALUTs, Registers, Fmax and Cycles column in excel sheet.
- #cycles will be generated for your design from testbench code.
- Make sure to use **Arria II GX EP2AGX45DF29I5** device
- Make sure to use Fmax for **Slow 900mV 100C Model**
- Make sure to use **Total number of cycles**

How to Fill up finalsummary.xlsx

- See annotated snapshot below on how to get ALUT's, Registers, Fmax, Cycles information to provide in finalsummary.xlsx



simplified_sha256.fit & bitcoin_hash.fit Fitter Report

- Copy of the **fitter reports** (not the flow report) with area numbers.
- Make sure to use **Arria II GX EP2AGX45DF29I5** device
- **IMPORTANT**: Make sure **Total block memory bits is 0**.

Fitter Summary	
Fitter Status	Successful - Wed May 09 15:37:04 2018
Quartus Prime Version	17.1.0 Build 590 10/25/2017 SJ Lite Edition
Revision Name	bitcoin_hash
Top-level Entity Name	bitcoin_hash
Family	Arria II GX
Device	EP2AGX45DF29I5
Timing Models	Final
Logic utilization	8 %
Combinational ALUTs	2,009 / 36,100 (6 %)
Memory ALUTs	0 / 18,050 (0 %)
Dedicated logic registers	1,257 / 36,100 (3 %)
Total registers	1257
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 %)

FSM Implementation Hints

Hints

- Since message size is hardcoded to 20 words, then there will be exactly 2 blocks.
- First block:
  - w[0]…w[15] correspond to first 16 words in memory
- Second block:
  - w[0]…w[3] correspond to remaining 4 words in memory
  - w[4] <= 32’80000000 to put in the “1” delimiter
  - w[5]…w[13] <= 32’00000000 for the “0” padding
  - W[14] <= 32’00000000 for the “0” padding (these are upper 32 bits of message length bits)
  - w[15] <= 32’d640, since 20 words = 640 bits (these are lower 32 bits of message length bits)

Hints

- You must use “clk” as the “mem_clk”.

  assign mem_clk = clk

- Using “negative” phase of “clk” for “mem_clk” is not allowed.

Hints : Parameter Arrays

- Declare SHA256 K array like this:

```
// SHA256 K constants
parameter int sha256_k[0:63] = '{
  32'h428a2f98, 32'h71374491, 32'hb5c0fbcf, 32'he9b5dba5, 32'h3956c25b, 32'h59f111f1, 32'h923f82a4, 32'hab1c5ed5,
  32'hd807aa98, 32'h12835b01, 32'h243185be, 32'h550c7dc3, 32'h72be5d74, 32'h80deb1fe, 32'h9bdc06a7, 32'hc19bf174,
  32'he49b69c1, 32'hefbe4786, 32'h0fc19dc6, 32'h240ca1cc, 32'h2de92c6f, 32'h4a7484aa, 32'h5cb0a9dc, 32'h76f988da,
  32'h983e5152, 32'ha831c66d, 32'hb00327c8, 32'hbf597fc7, 32'hc6e00bf3, 32'hd5a79147, 32'h06ca6351, 32'h14292967,
  32'h27b70a85, 32'h2e1b2138, 32'h4d2c6dfc, 32'h53380d13, 32'h650a7354, 32'h766a0abb, 32'h81c2c92e, 32'h92722c85,
  32'ha2bfe8a1, 32'ha81a664b, 32'hc24b8b70, 32'hc76c51a3, 32'hd192e819, 32'hd6990624, 32'hf40e3585, 32'h106aa070,
  32'h19a4c116, 32'h1e376c08, 32'h2748774c, 32'h34b0bcb5, 32'h391c0cb3, 32'h4ed8aa4a, 32'h5b9cca4f, 32'h682e6ff3,
  32'h748f82ee, 32'h78a5636f, 32'h84c87814, 32'h8cc70208, 32'h90befffa, 32'ha4506ceb, 32'hbef9a3f7, 32'hc67178f2
};
```

- Use it like this:

```
tmp <= g + sha256_k[i];
```

Hints : Right Rotation

- **Right rotate by 1**

  {x[30:0], x[31]}

  ((x >> 1) | (x << 31))

- **Right rotate by r**

  ((x >> r) | (x << (32-r))

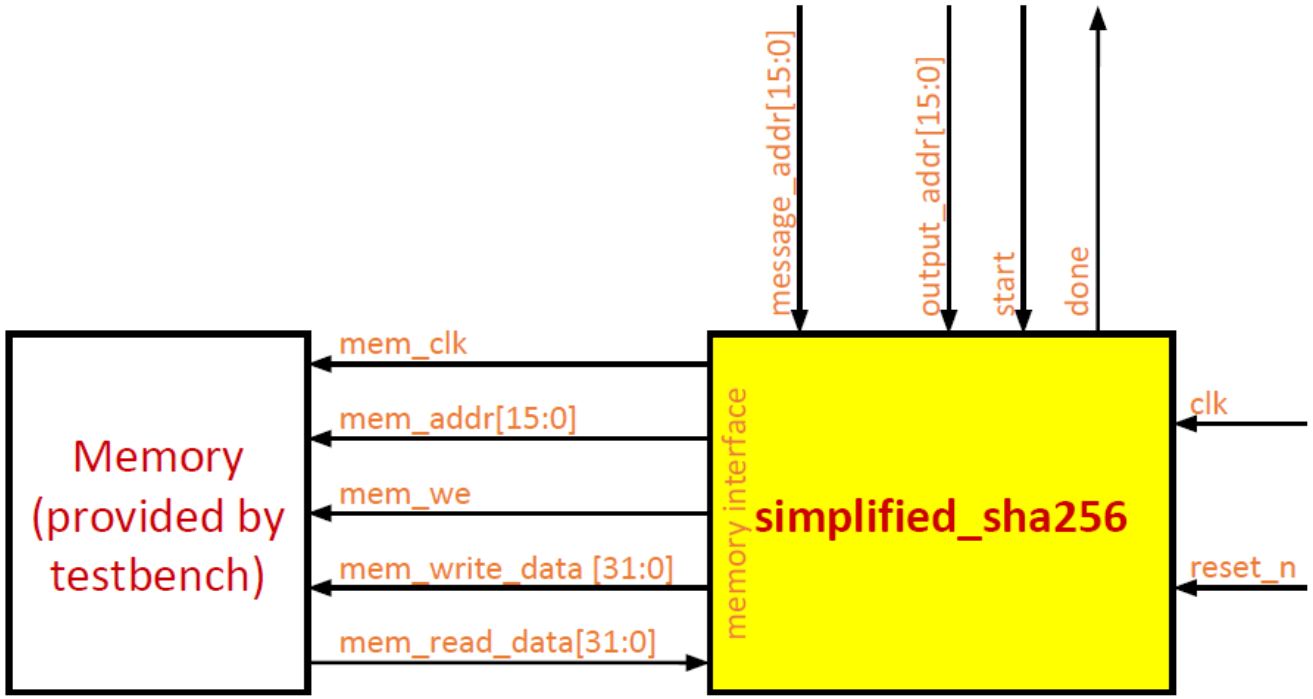
```
// right rotation
function logic [31:0] rightrotate(input logic [31:0] x,
                                  input logic [ 7:0] r);
   rightrotate = (x >> r) | (x << (32-r));
endfunction
```

Possible Results

- A reasonable "median" target:
  - #ALUTs = 1768, #Registers = 1209, Area = 2977
  - Fmax = 107.97 MHz, #Cycles = 147
  - Delay (microsecs) = 1.361, Area*Delay (millesec*area) = 4.053

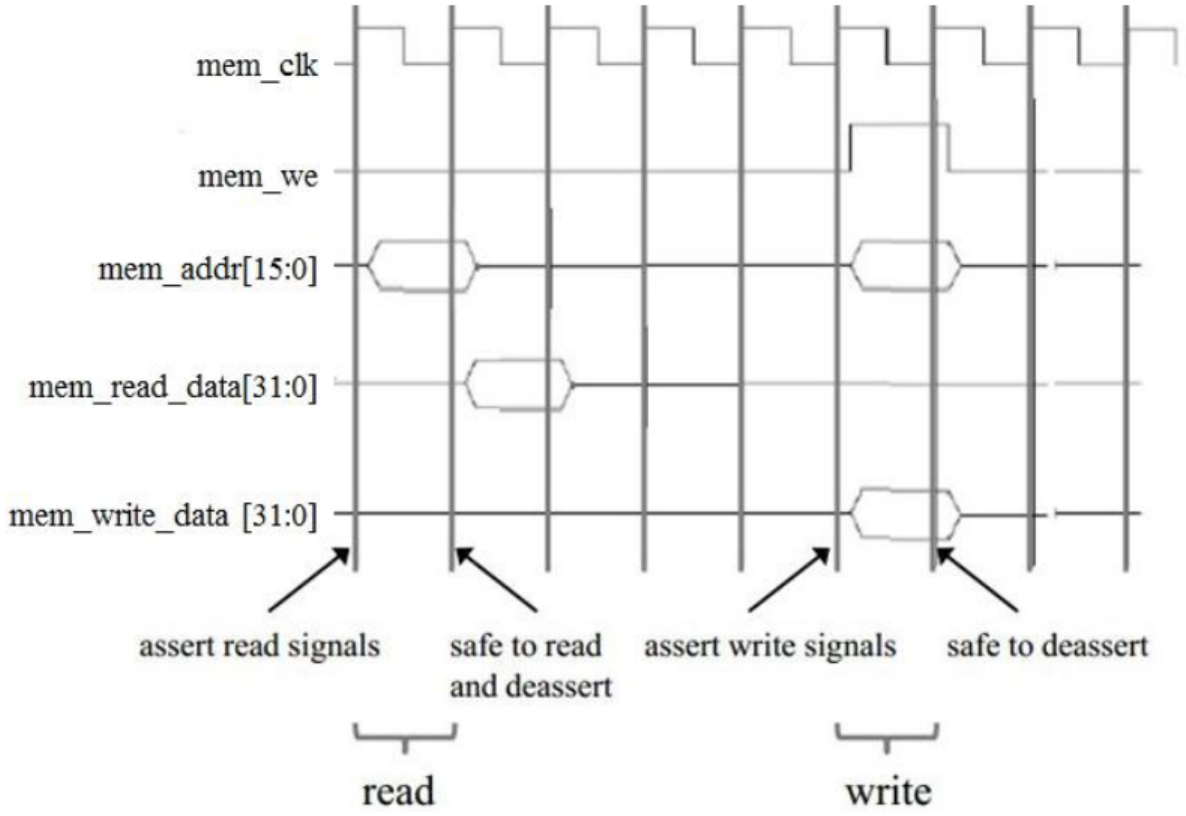
Memory Model

- To **read** from the memory:
  - Set **mem_addr** = address to read from (ex: 0x0000), **mem_we** = 0
  - At next clock cycle, read data from **mem_read_data**
- To **write** to the memory:
  - Set **mem_addr** = address to write to (ex: 0x0004), **mem_we** = 1, **mem_write_data** = data that you wish to write



Memory Model

- You can issue a new **read** or **write** command every cycle, **but** you have to wait for next cycle for data to be available on **mem_read_data** for a **read** command.
- **Be careful** that if you set **mem_addr** and **mem_we** inside **always_ff** block, compiler will produce flip-flops for them, which means external memory will not see the address and write-enable until another cycle later.



Memory Model

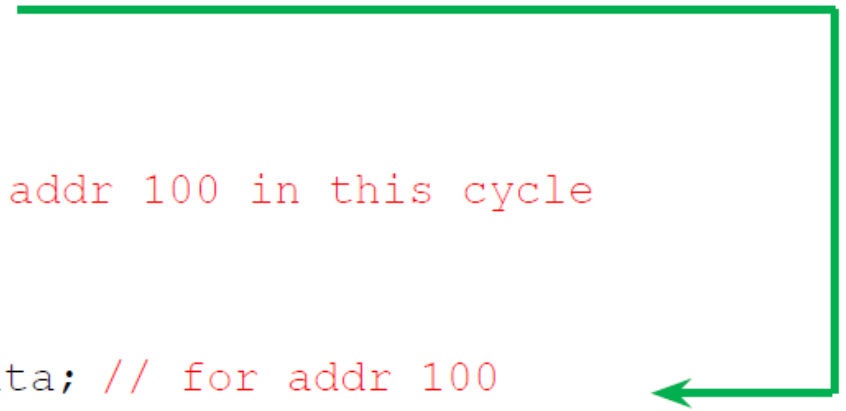
- THIS IS INCORRECT

```
always_ff @(posedge clk, negedge reset_n) begin
    if (!reset_n) begin
      state <= S0;
    end else
      case (state)
        S0: begin
            mem_we <= 0; // mem_we is 0 for memory read
            mem_addr <= 100; // address from where we want to read
            state <= S1;
        end
        S1: begin
            value <= mem_read_data; // data not yet available
            state <= S2;
        end
        ...
```

Memory Model

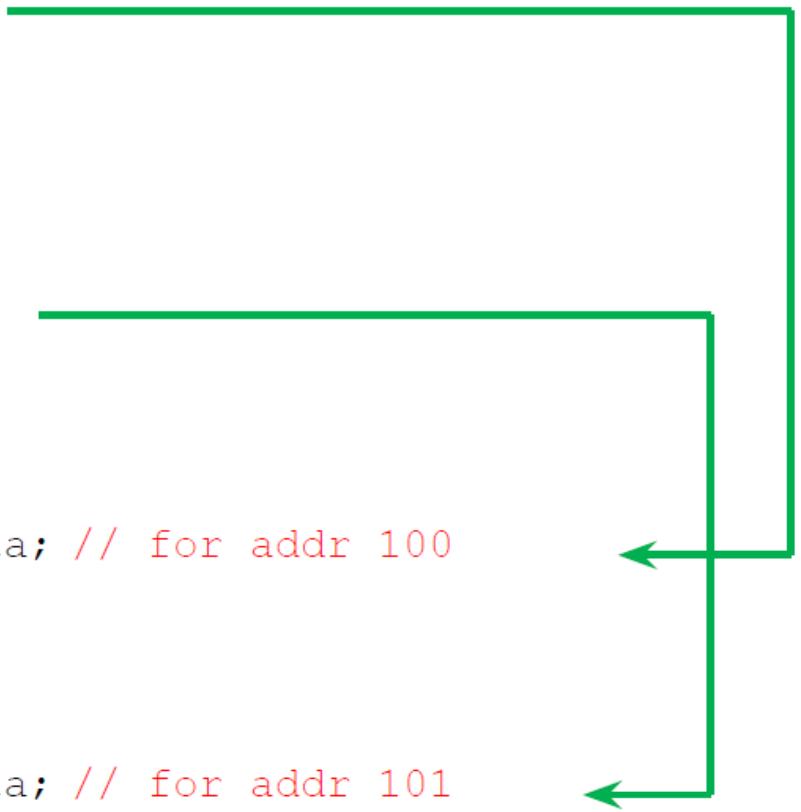
- Have to wait an extra cycle, correct way of reading from memory

```
always_ff @(posedge clk, negedge reset_n) begin
    if (!reset_n) begin
      state <= S0;
    end else
      case (state)
        S0: begin
            mem_we <= 0;
            mem_addr <= 100;
            state <= S1;
        end
        S1: // memory only sees addr 100 in this cycle
            state <= S2;
        S2: begin
            value <= mem_read_data; // for addr 100
        ...
```



Pipelining the Memory Read

```
case (state)
    S0: begin
        mem_we <= 0;
        mem_addr <= 100;
        state <= S1;
    end
    S1: begin
        mem_we <= 0;
        mem_addr <= 101;
        state <= S2;
    end
    S2: begin
        value <= mem_read_data; // for addr 100
        state <= S3;
    end
    S3: begin
        value <= mem_read_data; // for addr 101
        state <= S4;
    ...
```



Memory Write Example

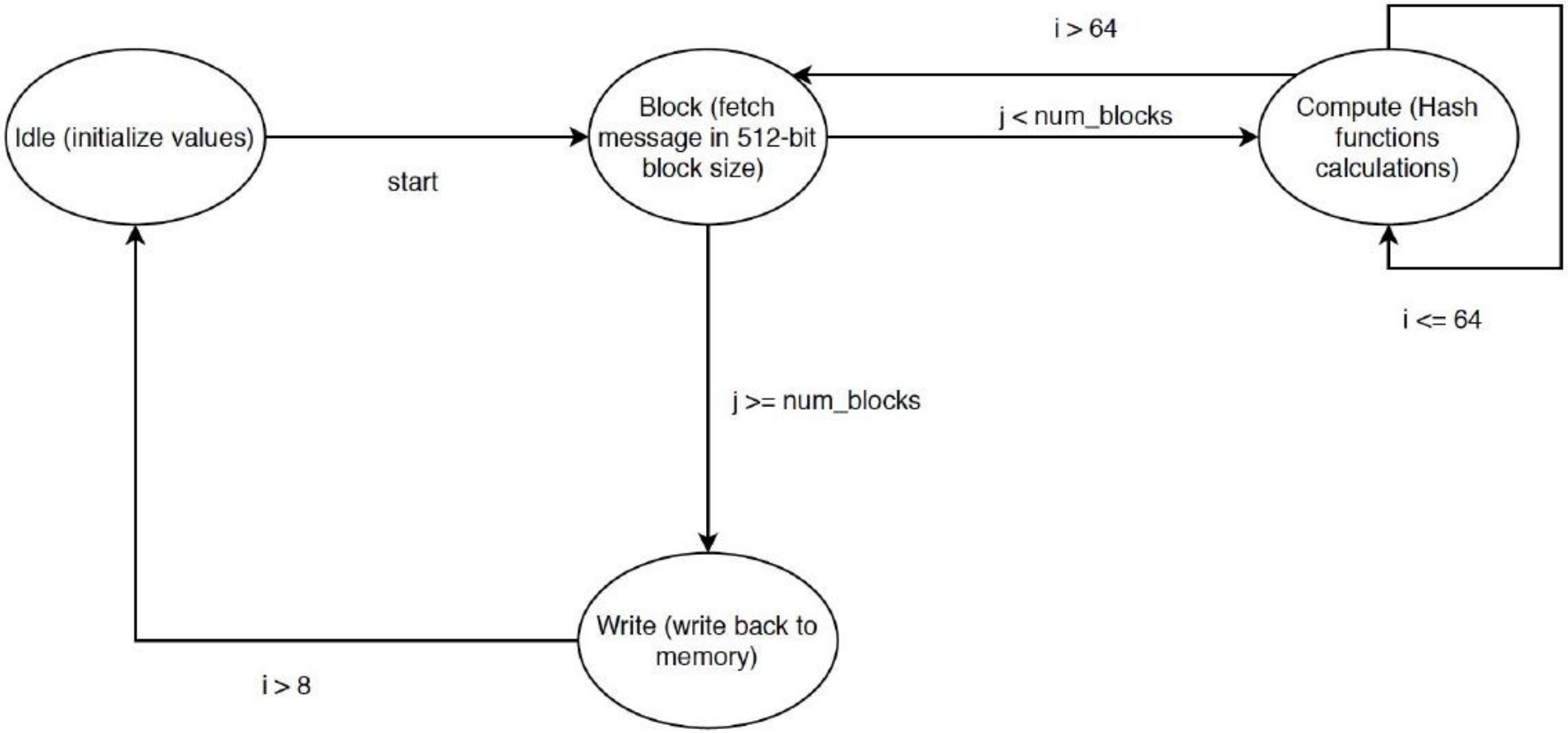
- Notice here that we assign address to mem_addr and data to mem_write_data in the same cycle.

```
always_ff @(posedge clk, negedge reset_n) begin
    if (!reset_n) begin
      state <= S0;
    end else
      case (state)
        S0: begin
            mem_we <= 1; // mem_we is 1 for writing
            mem_addr <= 100; // assigning address where we want to write
                mem_write_data <= 20; //assigning the value which we want to write
            state <= S1;
        end
        S1: begin
            state <= S2;
        end
        ...
```

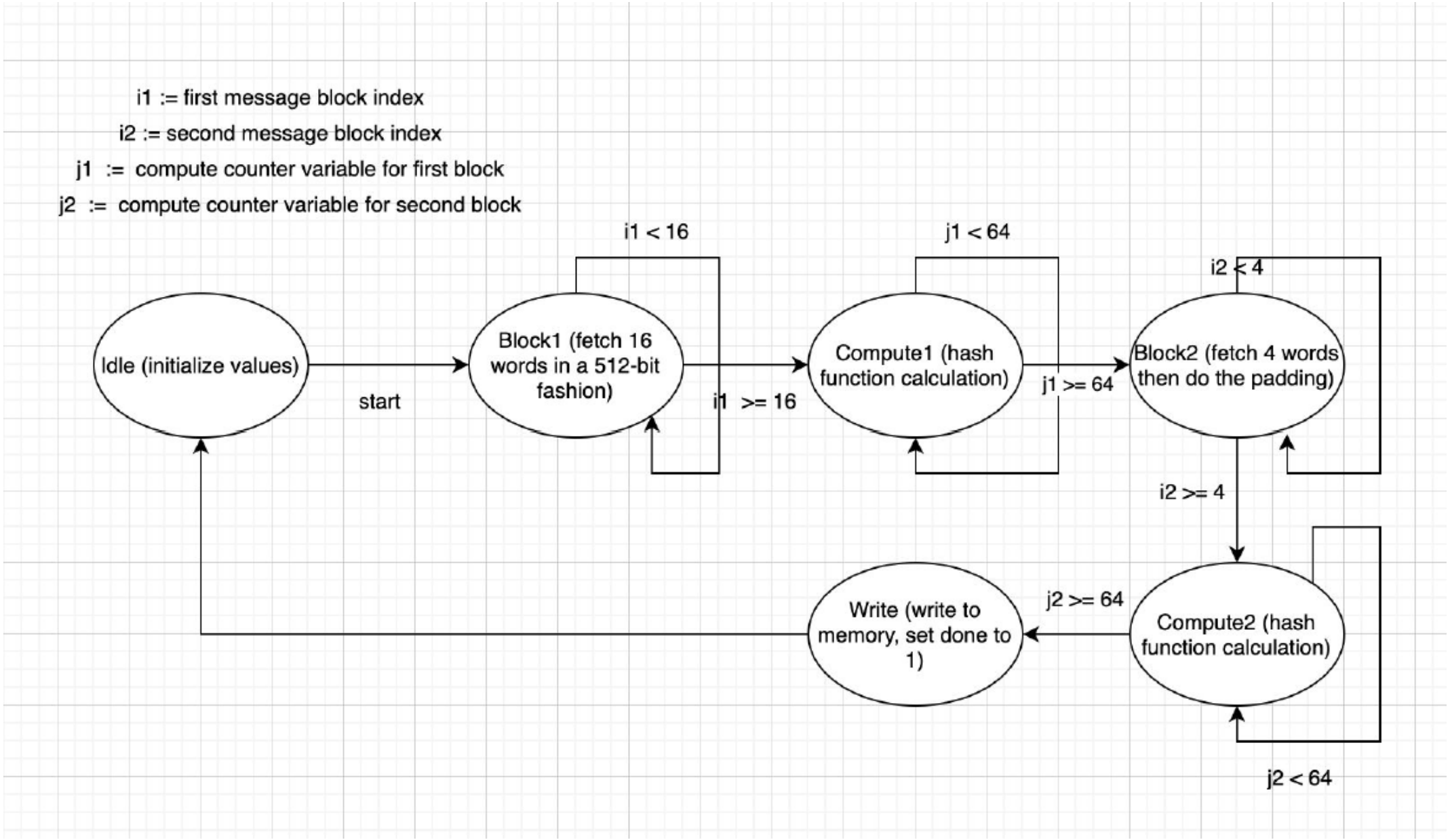
FSM Design Template (Part-1 Scalable Implementation to Part-2)

j := number of block iteration variable

i := number of processing counter variable



FSM Design Template (Part-1 Non-Scalable Implementation to Part-2)



Debug Hints

Debug Hints

- **When System Verilog Code for simplified_sha256 does not generate correct output hash values, then you may follow step by step debugging as suggested in this document**

- **Note :**
  - Perform methodical debug. i.e. Debug each FSM state one by one and do not move to next FSM state until current FSM state under debug is generating correct output values
  - Refer to two helper excel files in project file zip folder to compare SHA operation output in a through h and final hash h0 to h7 values for both message blocks including starting first and second message block creation and word expansion. These files are :
    - simplified_sha256.xls and simplified_sha256_w_values.xls
  - Next slides which has reference waveforms for each FSM state is based on the implementation which has :
    - 1 always_ff block FSM style
    - 5 FSM states, IDLE, READ, BLOCK, COMPUTE, WRITE
    - "message" 2D array stores message words after reading from memory
    - "w" array has message blocks and expanded words.
    - Waveform is for SHA256 implementation using w[64] implementation (unoptimized)

Debug Hints

- **Step-1 : First make sure that in IDLE states below mentioned variables are initialized and with correct values**
  - h0 to h7 local variables should be initialized to initial hash constant
  - Intermediate hash variables a, b, c, d, e, f, g, h each is initialized to 0
  - Message block iteration index variable is initialized to 0
  - Memory offset variable is initialized to 0
  - Any other index variables used in FSM code is initialized to 0 or required value
  - Local signals controlling memory such as cur_we, cur_write_data are all initialized to 0
  - Memory address cur_addr is initialized to message_addr
  - Ensure from IDLE state after initialization next state moves to READ state when start == 1

Debug Hints

- **Step-2 : Ensure at the end of the READ state, memory from testbench is read correctly and stores in message array**
  - Whichever 2D array is declared for storing input message in your FSM code (say it is named as "message"), at the end of the READ FSM state, content in message array has stored all 640-bit message in 32-bit words from message[0] to message[19]
  - Check in waveform that message[0] to message[10] has below mentioned values :

Word Index (Total : 20 words)	Input Message Words (split in 32-bits word chunks)	32-bits x 20 = 640 bits total message
0	01234567	32bit Message Word-0
1	02468ace	32bit Message Word-1
2	048d159c	32bit Message Word-2
3	091a2b38	32bit Message Word-3
4	12345670	32bit Message Word-4
5	2468ace0	32bit Message Word-5
6	48d159c0	32bit Message Word-6
7	91a2b380	32bit Message Word-7
8	23456701	32bit Message Word-8
9	468ace02	32bit Message Word-9
10	8d159c04	32bit Message Word-10
11	1a2b3809	32bit Message Word-11
12	34567012	32bit Message Word-12
13	68ace024	32bit Message Word-13
14	d159c048	32bit Message Word-14
15	a2b38091	32bit Message Word-15
16	45670123	32bit Message Word-16
17	8ace0246	32bit Message Word-17
18	159c048d	32bit Message Word-18
19	00000000	32bit Message Word-19
	Total = 32 x 20 = 640 bits	

Note : values shown above is in hexadecimal format.

Debug Hints

❑ Step-2 : Continued…

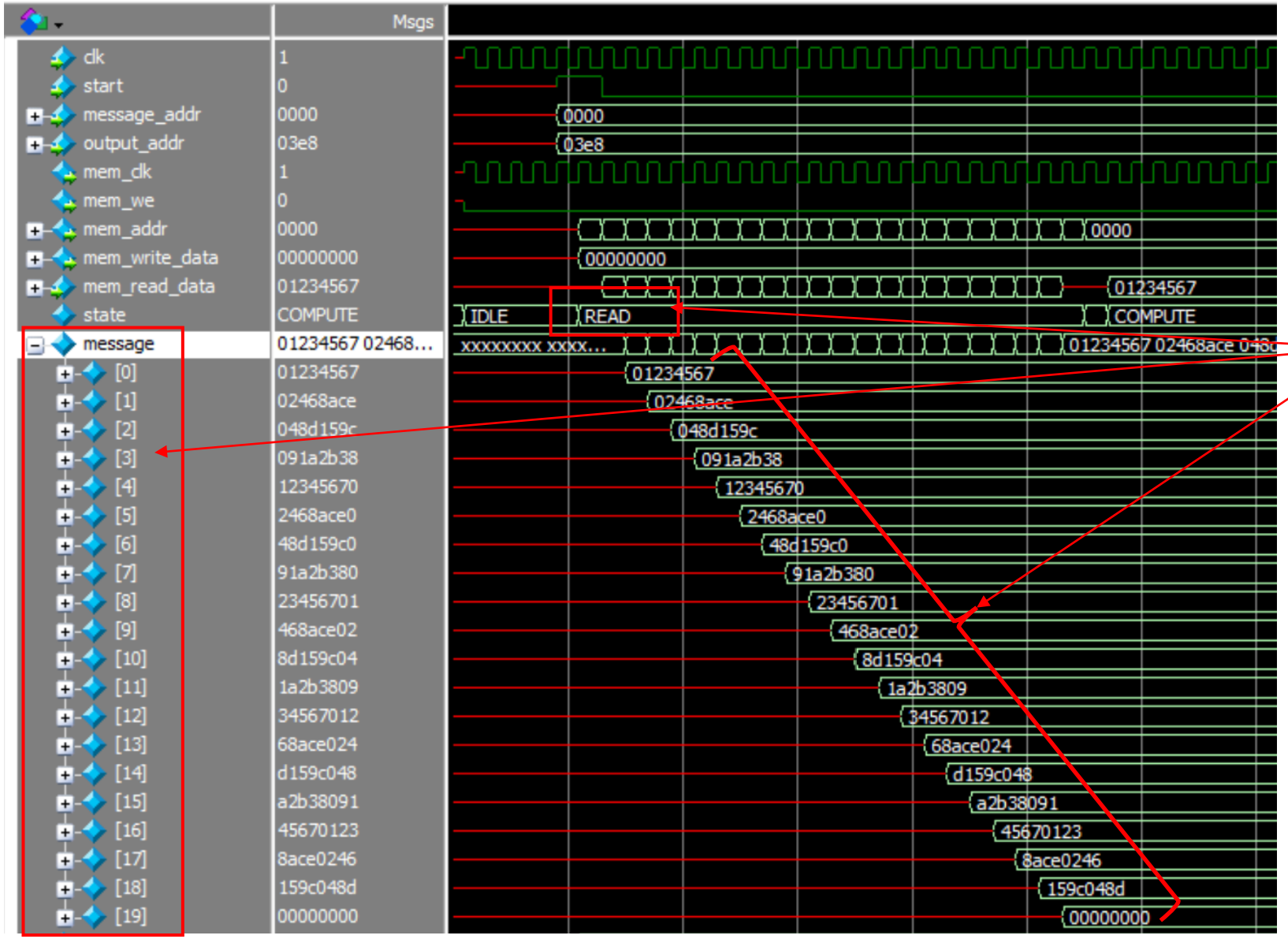
Reference simulation waveform for READ FSM state with memory control signals

In READ FSM State, the memory "dpsram" content is read one by one in each clock cycle.
mem_addr is 1 cycle ahead of mem_read_data. In another words, after generating mem_addr, the data from the memory is available next cycle. See above "mem_addr" is 0000 and mem_read_data "01234567" from memory coming into FSM is after 1 clock cycle.
Note : values shown above is in hexadecimal format.

Debug Hints

- **Step-2 : Continued …**

Reference simulation waveform for READ FSM state with message array



In READ FSM state in 20 clock cycles 640-bit message words are read and stored in 32-bits chunks in message[0] to message[19] array. Values shown in message are with Radix set to hexadecimal in modelsim waveform.

Debug Hints

- **Step-2 : Continued…**
  - If the message array has all "x" values , then ensure memory control signals are driven correctly
  - If the message array has "x" in message[0] and rest of the message[1] to message[19] words have known value but shifted values by 1 position, then it is due to 1 cycle early reading of mem_read_data in your FSM Code.
    - Remember, after memory gets mem_we=1 and mem_address value, the read data from the memory will be available 1 clock cycle later. Hence memory read in READ FSM state should account for sampling mem_read_data 1 cycle later after mem_addr is incremented using offset variable
      - One way is to add a 1 cycle WAIT state before READ FSM state. Essentially, from IDLE state transition to WAIT state and then move to READ FSM state. And in IDLE state, set cur_we=0, cur_addr to message_addr, offset to 0

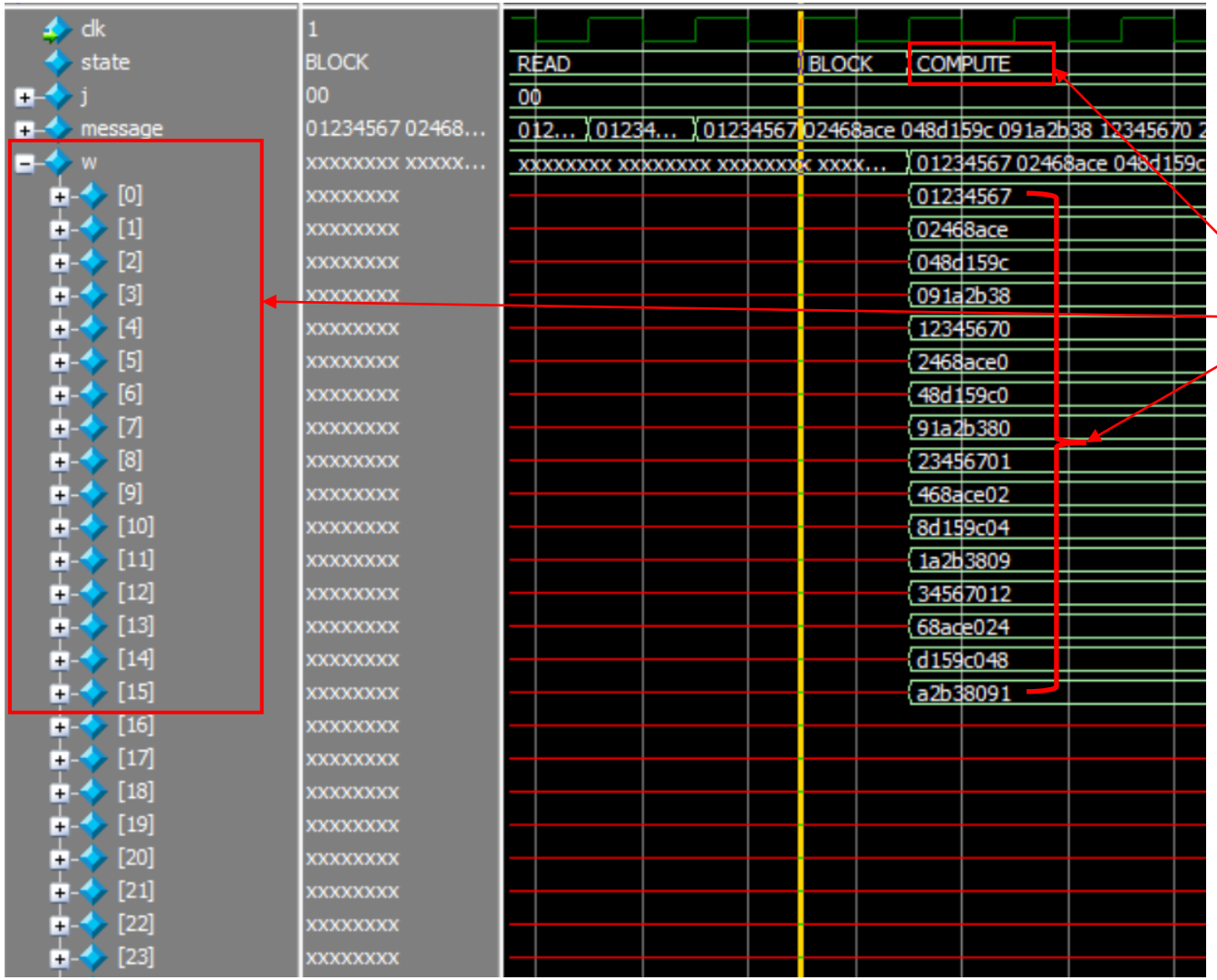
Debug Hints

- **Step-3 : Ensure formation of Message block is correct in BLOCK FSM state**
  - Let's say your code has "w" 2D array.
  - For first message block, ensure in waveform that w[0] to w[15] word array should reflect message[0] to message[15]
  - For second message block, ensure in waveform that :
    - w[0] to w[3] word array should reflect message[16] to message[19]
    - W[4] should have MSB bit '1' and all remainder 31 bits 0 i.e. 32'h80000000
    - W[15] should have message size 32'd640 in decimal or in hex 32'h280
  - Ensure in waveform, intermediate hash variables "a" through "h" is initialized to "h0" to "h7" for each message block before entering COMPUTE FSM state. This should be done in IDLE state.
  - Ensure, if there is an index variable, say "i", used in COMPUTE FSM state for loop as part of word expansion, index "i" is initialized to 0 in READ state or IDLE state.
  - Ensure BLOCK state has jump to COMPUTE state after creation of each first message block and second message block
  - If all message blocks have COMPUTE stage called, then jump to WRITE state.

Debug Hints

❑ Step-4 : Ensure there are no "x" values in w[0] to w[15] array at the end of BLOCK FSM state

Reference simulation waveform for BLOCK FSM state after composing first message block



Simulation snapshot creation of first message block. At the end of the BLOCK FSM State, the first 512 bits out of 640 bits total message bits are stored in 32-bit chunks in w[0] to w[15] after copying it from message[0] to message[15]

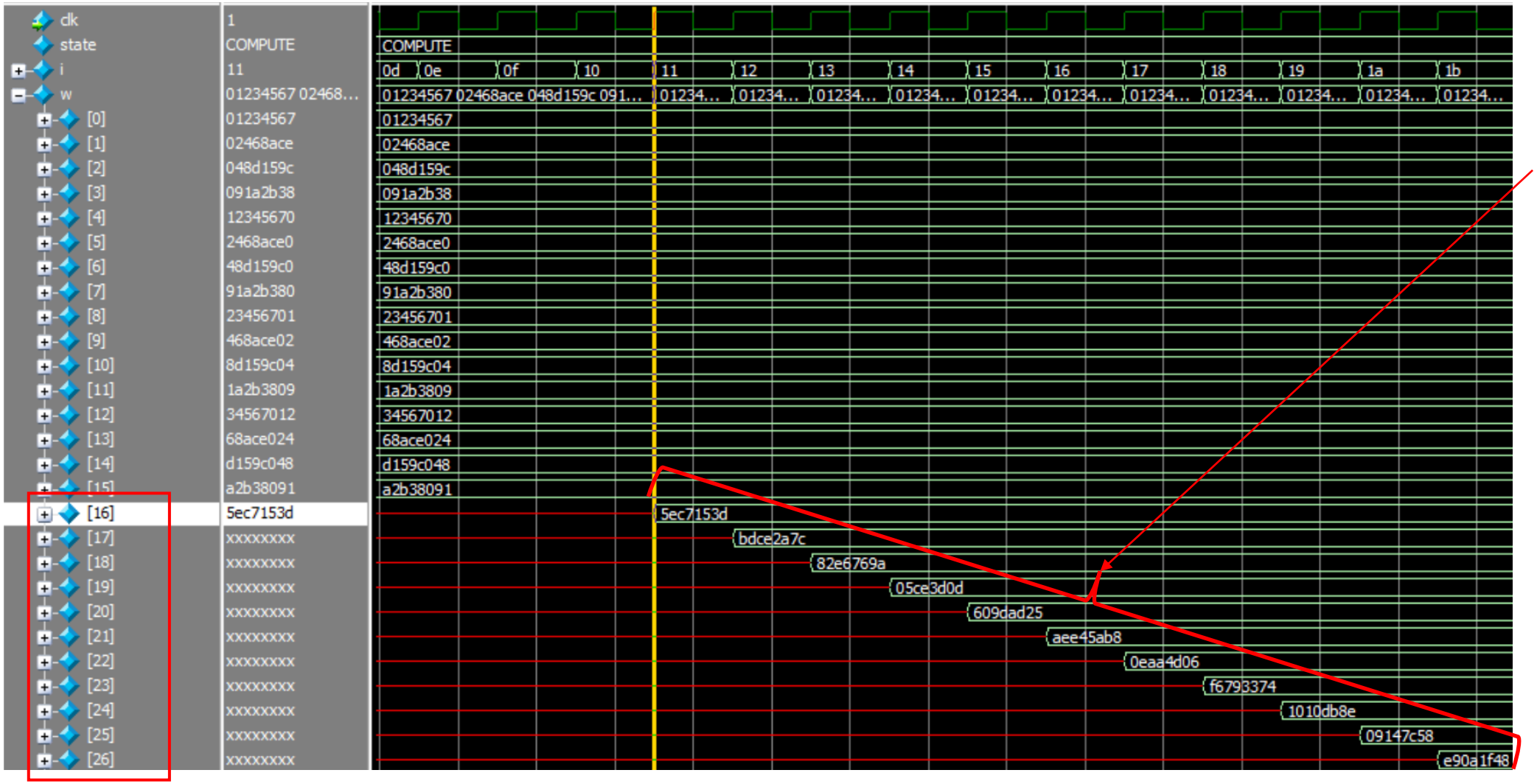
Note :

- "w" array will get message bits at the end of the Block FSM state which is at the clock when FSM enters COMPUTE state
- w[16] to w[63] will have 'x' at the starting the compute state as the word expansion has not yet started

Debug Hints

- **Step-5 : Ensure word expansion of first message block from w[16] to w[63] is performed correctly in COMPUTE FSM sate**

Reference simulation waveform for first message word expansion in COMPUTE state



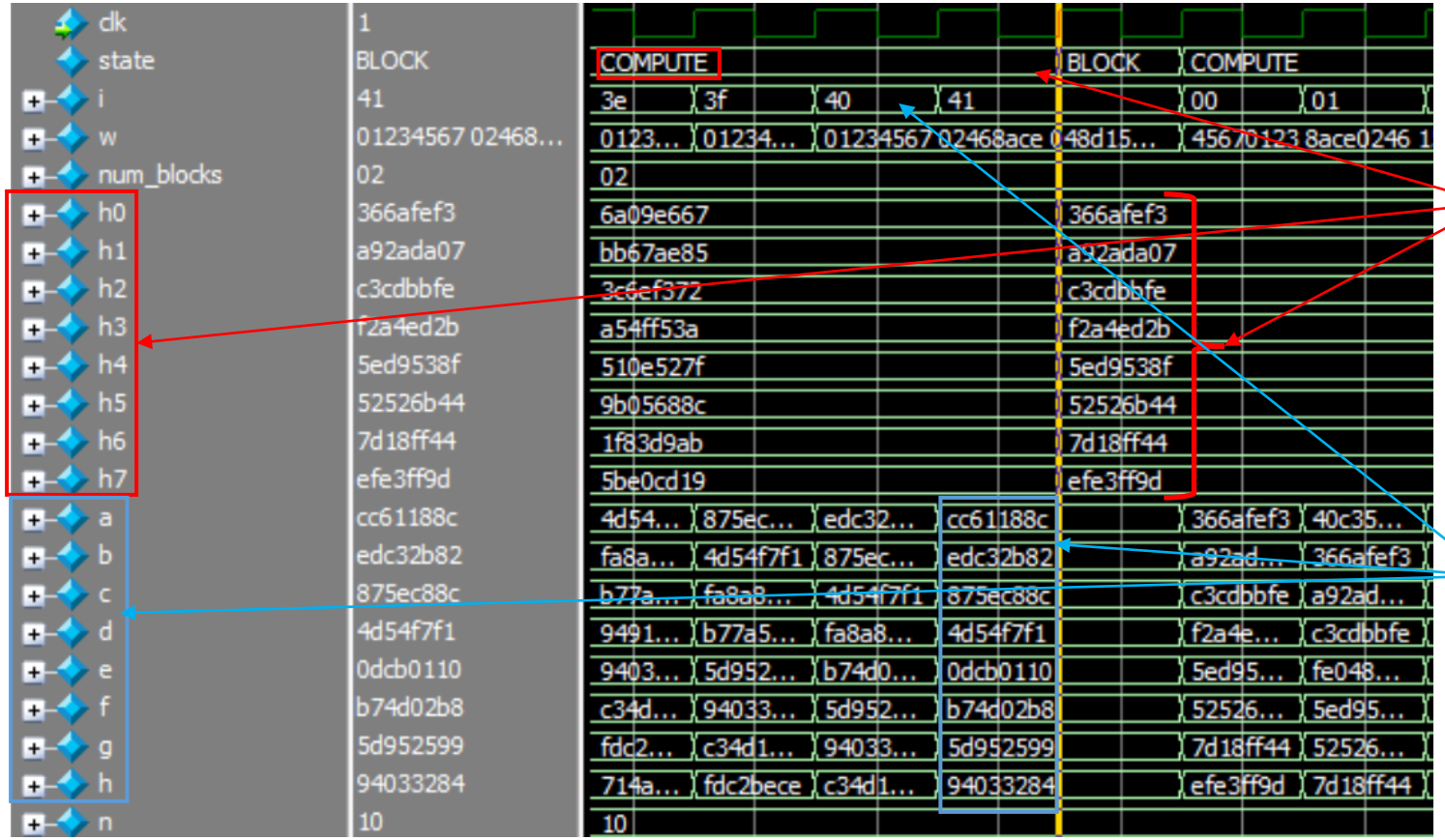
Confirm using simplified_sha256_w_values.xls which has expected w[16] to w[63] for 1st message block is reflected in simulation Waveform

Simulation snapshot is not fully shown below. It only shown below until w[26]

Debug Hints

- **Step-6 : Ensure after SHA256 operation in COMPUTE FSM state, output hash h0 to h7 has correct hash values for first message block. Also check intermediate hash values 'a' through 'h' if these are ending with correct values and there are no 'x' values**

Reference simulation waveform for COMPUTE FSM state after performing sha operation on word expanded first message block w[0] to w[63]



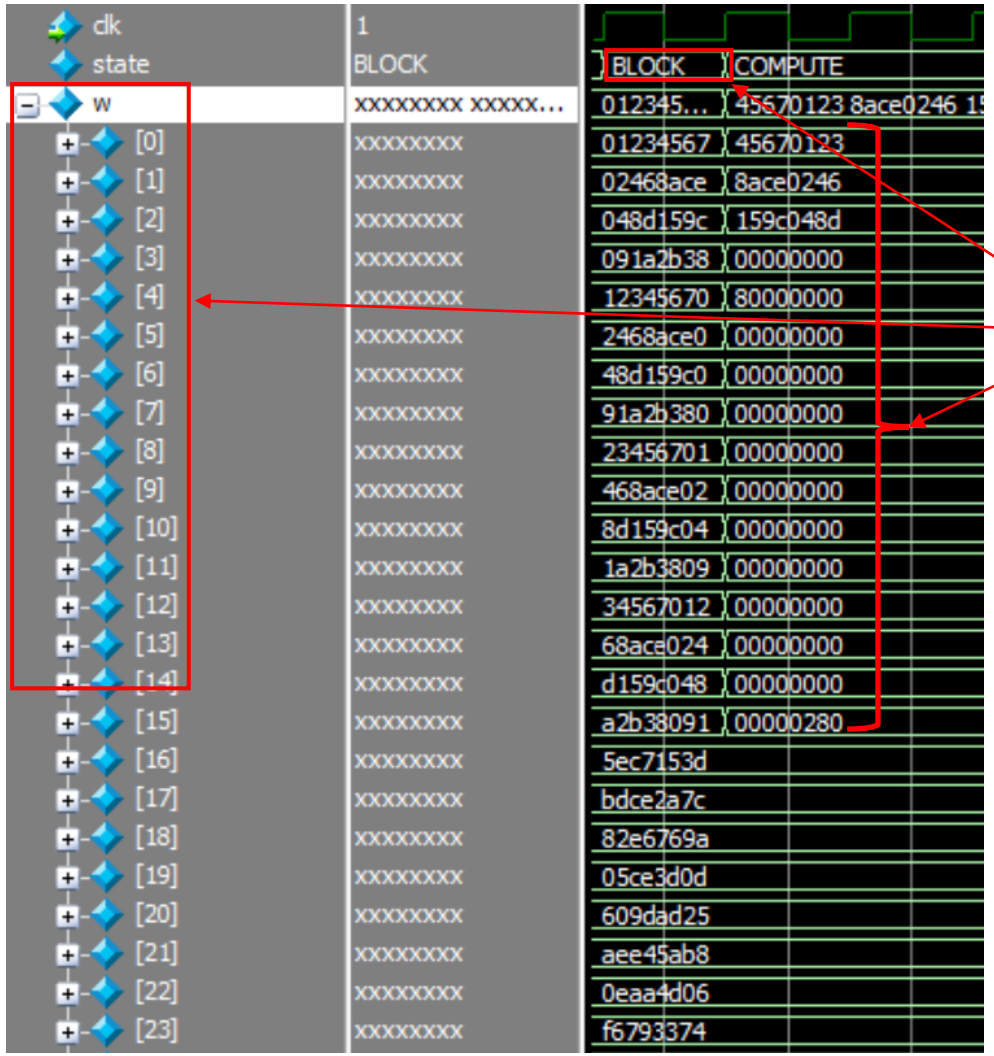
At the end of the SHA256 operation of first expanded message block in COMPUTE FSM state, output hash h0 to h7 should reflect the correct values as shown in the snapshot. It should not be 'x' value either. These values will be reflected at the start of the 2nd message BLOCK FSM state

At the end of 64th iteration (i.e. index 'i' = 40 in hexadecimal) of sha operation of first expanded message block, intermediate hash 'a' to 'h' should reflect the correct values as shown in the snapshot. It should not be 'x' value either. Refer to simplified_sha256.xls in project folder which has value of 'a' to 'h' for all 64 iterations of first message block

Debug Hints

- **Step-7 : Ensure in BLOCK FSM state as part of second message block formation, w[0] to w[15] has remaining input message bits with constant 1, padding 0 bits and message size**

Reference simulation waveform for BLOCK FSM state after composing second message block



Simulation snapshot creation of second message block. At the end of the BLOCK FSM State and start of the COMPUTE FSM state, the remaining 128-bit message out of 640 bits total message bits are stored in 32-bit chunks in w[0] to w[3] after copying it from message[16] to message[19]

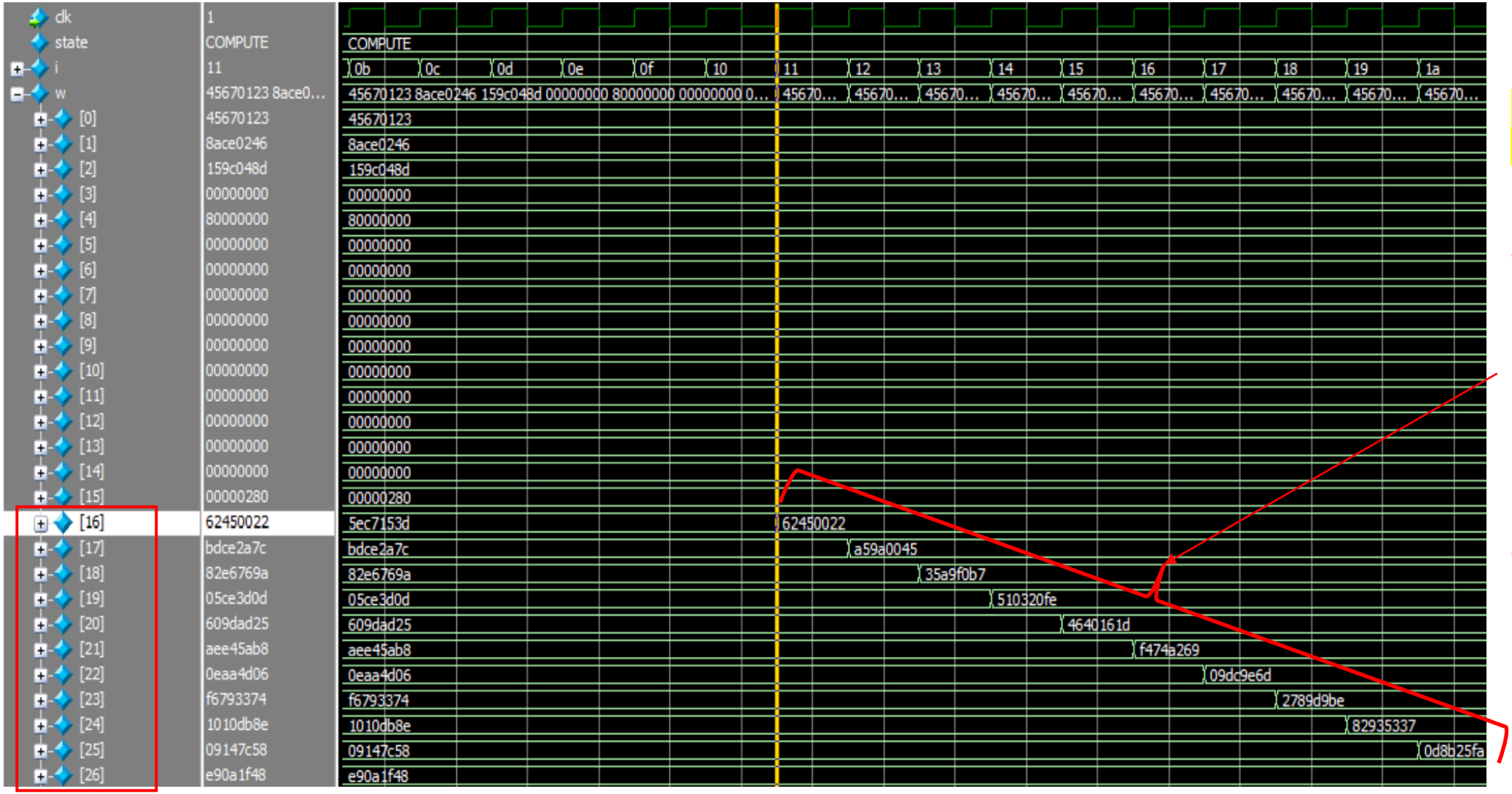
w[4] to w[14] are padding bits

w[15] is the total size of the message which is 280 in hexadecimal and 640 in decinal representation

Debug Hints

- **Step-8 : Ensure in waveform word expansion of second message block from w[16] to w[63] performed correctly in COMPUTE FSM sate**

  Reference simulation waveform for second message word expansion in COMPUTE state



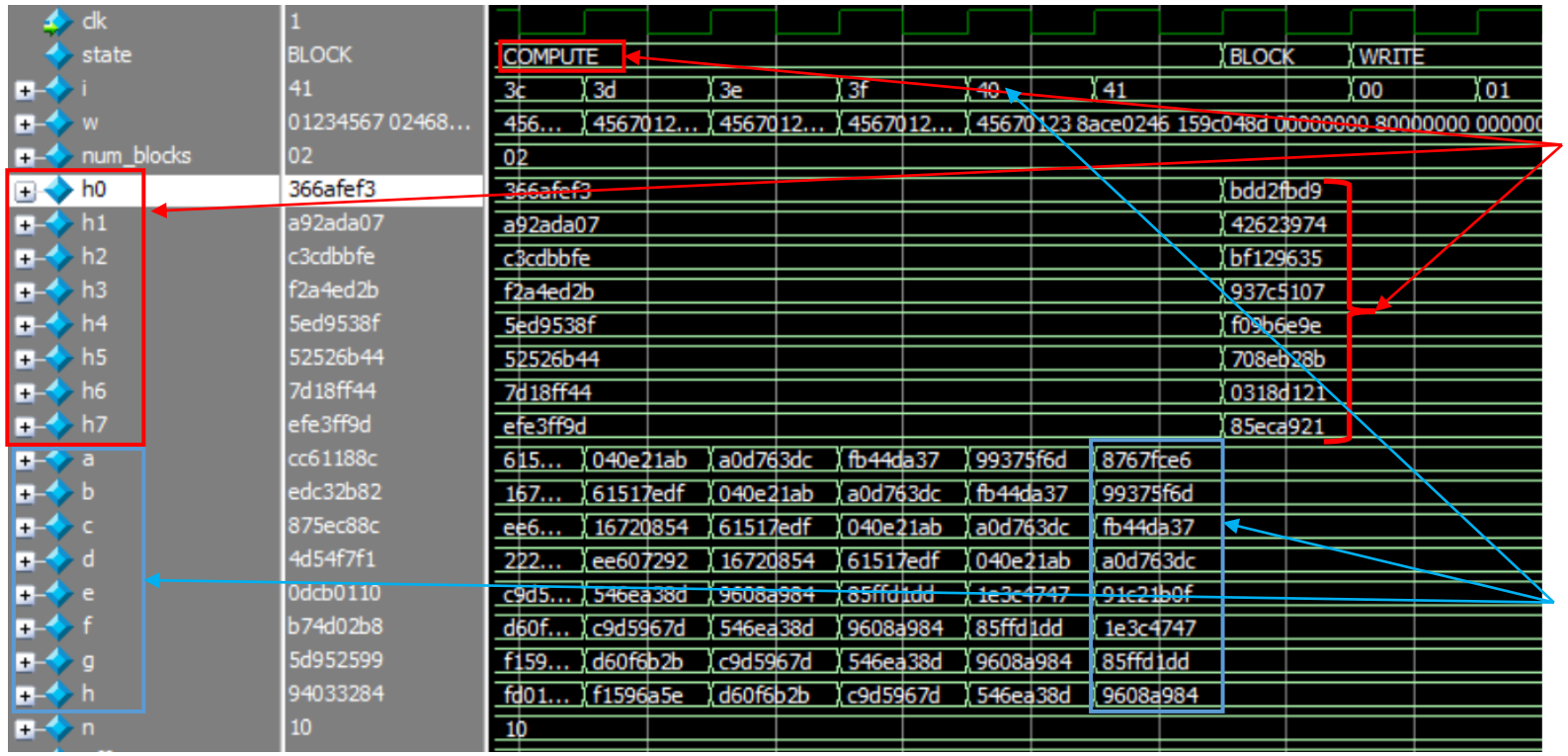
Confirm using simplified_sha256_w_values.xls which has expected w[16] to w[63] for 2nd message block is reflected in simulation Waveform.

Simulation snapshot is not fully shown below. It only shown below until w[25]

Debug Hints

- **Step-9 : Ensure after SHA256 operation in COMPUTE FSM state, output hash h0 to h7 has correct hash values for first message block. Also check intermediate hash values 'a' through 'h' if these are ending with correct values and there are no 'x' values**

Reference simulation waveform for COMPUTE FSM state after performing sha operation on word expanded first message block w[0] to w[63]



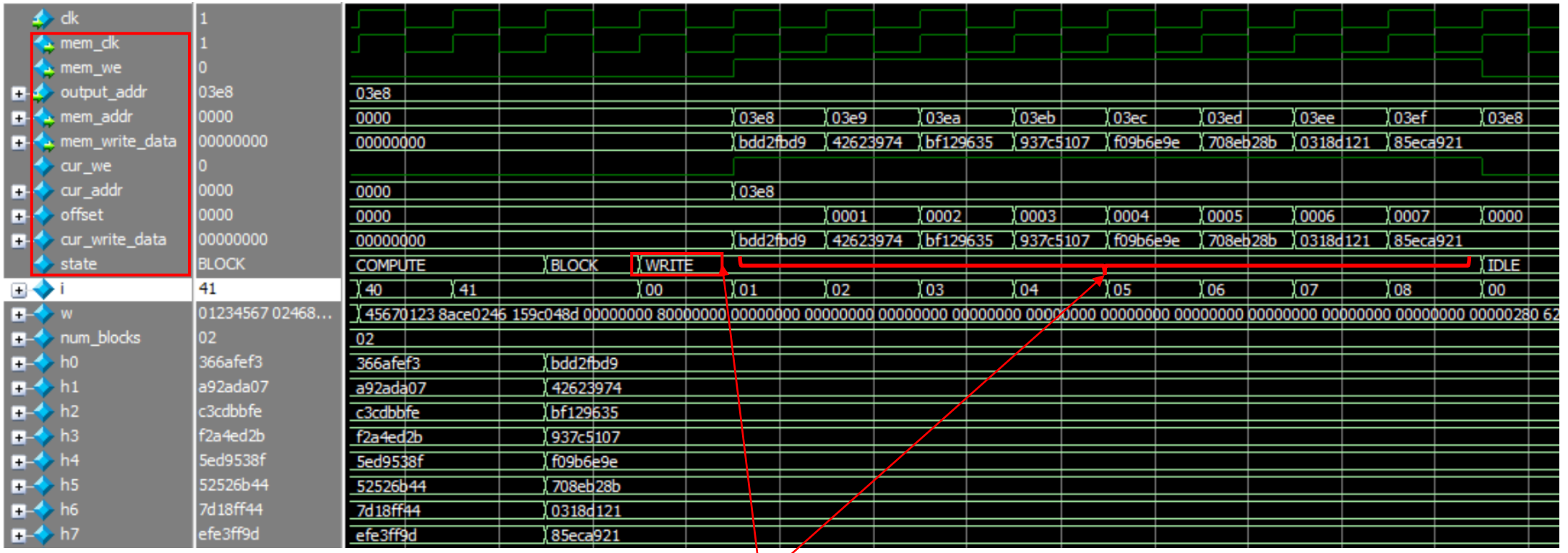
At the end of the SHA256 operation of second expanded message block in COMPUTE FSM state, output hash h0 to h7 should reflect the correct values as shown in the snapshot. It should not be 'x' value either. These values will be reflected after the end of COMPUTE FSM State and before entering WRITE FSM sate

At the end of 64th iteration (i.e. index 'i' = 40 in hexadecimal) of SHA256 operation of second expanded message block, intermediate hash 'a' to 'h' should reflect the correct values as shown in the snapshot. It should not be 'x' value either. Refer to simplified_sha256.xls in project folder which has value of 'a' to 'h' for all 64 iterations of first message block

Debug Hints

- **Step-10 : Ensure in WRITE FSM state control signals for memory write and final hash values h0 to h7 are written correctly to the memory inside testbench**

Reference simulation waveform for WRITE FSM state



During WRITE FSM state, output_addr is incremented by '1' using cur_addr each clock cycle and h0 to h7 is written one by one each cycle to the memory inside testbench

References

- **SHA256 Algorithm References :**
  - https://en.wikipedia.org/wiki/SHA-2
  - https://medium.com/bugbountywriteup/breaking-down-sha-256-algorithm-2ce61d86f7a3

- **Hashing Function Application (Password Protection) :**
  - https://www.youtube.com/watch?v=cczlpiiu42M&t=3s