Ashlee Young A17050291
Andrew Onozuka A16760043
27 Nov. 2024

ECE 111 Project Part 1

Handshake Synchronizer

1. What is a Handshake Synchronizer?

The handshake synchronizer in our project is designed to ensure safe and reliable transfer of 32-bit data from a fast clock domain (src_clk) to a slower clock domain (dest_clk). It addresses the challenges of crossing asynchronous clock domains, such as metastability and data corruption, through the implementation of a 4-phase request-acknowledge (REQ-ACK) handshake protocol. The synchronizer prevents data loss or corruption by ensuring that both the sender and receiver agree on the status of data transfer at every step of the communication.

This synchronizer is composed of several key modules, including the sender_fsm, receiver_fsm, flipflop_synchronizer, and the handshake_synchronizer module itself. The sender_fsm operates in the faster source clock domain, where it raises a req_o signal to indicate that new data is ready to be transferred. This signal is synchronized across the clock domains using the flipflop_synchronizer, which mitigates metastability by stabilizing the signal in the destination clock domain. The receiver_fsm, operating in the slower destination clock domain, captures the data and sends back an acknowledgment signal (ack_o) to the sender. The acknowledgment is similarly synchronized back to the source clock domain, completing the handshake. A latch is placed after the sender's data register to hold the data steady until it is fully captured by the receiver, ensuring data stability throughout the process.

The clock periods for the source and destination clocks are 20 ns and 124 ns, respectively, meaning the source clock operates approximately six times faster than the destination clock. This significant difference in clock speeds requires careful synchronization, which is achieved using the REQ-ACK protocol. The protocol ensures that the sender only updates or transmits data when the receiver is ready, and the receiver only processes data that has been fully transmitted. This approach avoids any timing mismatches or race conditions during the transfer.

The testbench provided simulates the operation of the handshake synchronizer, generating the src_clk and dest_clk signals to evaluate its performance. Since the testbench does not include a self-checking mechanism, the simulation results are reviewed manually in part 3 to ensure that each data value sent by the sender FSM is correctly captured by the receiver FSM.

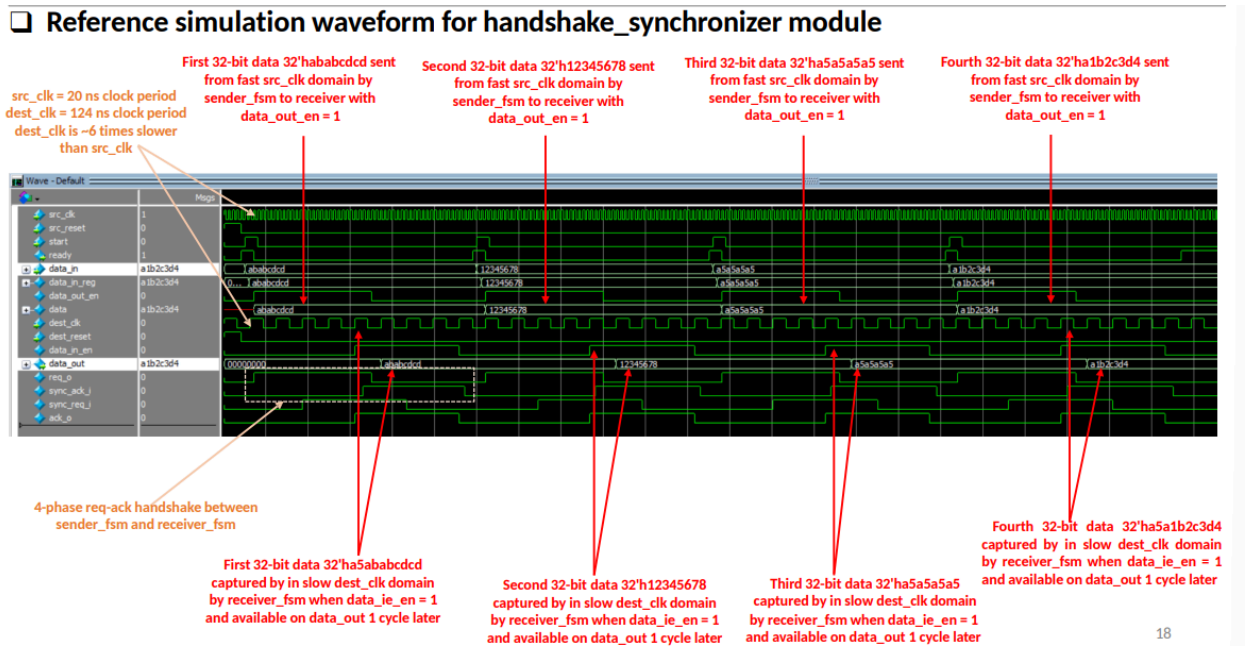
Our handshake synchronizer achieves reliable communication between asynchronous domains by combining FSM logic, signal synchronization, and stable data latching. It implements the 4-phase REQ-ACK protocol, prevents data loss, and ensures smooth data transfer between the two clock domains. The modular design of the synchronizer also is good because it makes it more adaptable to varying data sizes and clock frequencies, to provide a solution for clock domain crossing challenges.

2. **Synthesis Resource Usage Snapshot** (explanation not required)

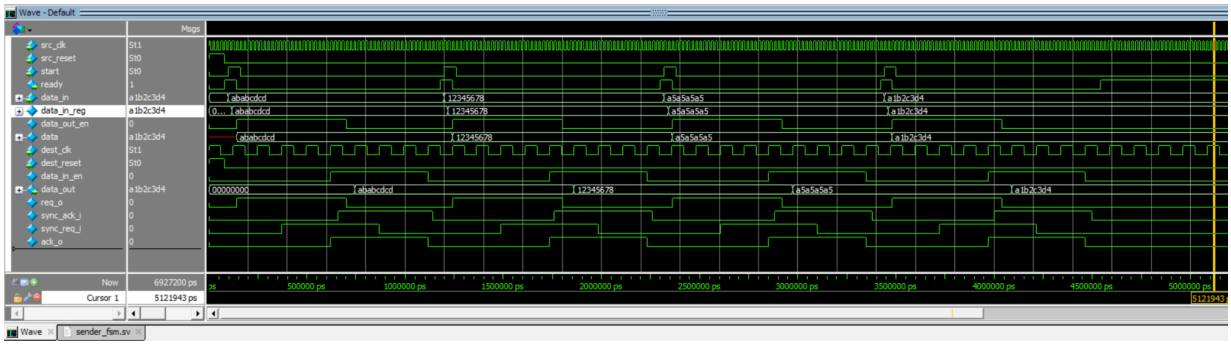
ucsd > ece111 > ProjectPart1 > handshake_synchronizer > handshake_synchronizer-Resource Usage Summary.rpt

```
+-----------------------------------------------------------------------+
; Analysis & Synthesis Resource Usage Summary                          ;
+-----------------------------------------------+-----------------------+
; Resource                                      ; Usage                 ;
+-----------------------------------------------+-----------------------+
; Estimated ALUTs Used                          ; 37                    ;
;     -- Combinational ALUTs                    ; 37                    ;
;     -- Memory ALUTs                           ; 0                     ;
;     -- LUT_REGs                               ; 0                     ;
; Dedicated logic registers                     ; 75                    ;
;                                               ;                       ;
; Estimated ALUTs Unavailable                   ; 0                     ;
;     -- Due to unpartnered combinational logic ; 0                     ;
;     -- Due to Memory ALUTs                    ; 0                     ;
;                                               ;                       ;
; Total combinational functions                 ; 37                    ;
; Combinational ALUT usage by number of inputs  ;                       ;
;     -- 7 input functions                      ; 0                     ;
;     -- 6 input functions                      ; 0                     ;
;     -- 5 input functions                      ; 2                     ;
;     -- 4 input functions                      ; 0                     ;
;     -- <=3 input functions                    ; 35                    ;
;                                               ;                       ;
; Combinational ALUTs by mode                   ;                       ;
;     -- normal mode                           ; 37                    ;
;     -- extended LUT mode                      ; 0                     ;
;     -- arithmetic mode                        ; 0                     ;
;     -- shared arithmetic mode                 ; 0                     ;
;                                               ;                       ;
; Estimated ALUT/register pairs used            ; 76                    ;
;                                               ;                       ;
; Total registers                               ; 75                    ;
;     -- Dedicated logic registers              ; 75                    ;
;     -- I/O registers                          ; 0                     ;
;     -- LUT_REGs                               ; 0                     ;
;                                               ;                       ;
;                                               ;                       ;
; I/O pins                                      ; 70                    ;
;                                               ;                       ;
; DSP block 18-bit elements                     ; 0                     ;
;                                               ;                       ;
; Maximum fan-out node                          ; src_reset~input       ;
; Maximum fan-out                               ; 42                    ;
; Total fan-out                                 ; 504                   ;
; Average fan-out                               ; 2.00                  ;
+-----------------------------------------------+-----------------------+
```

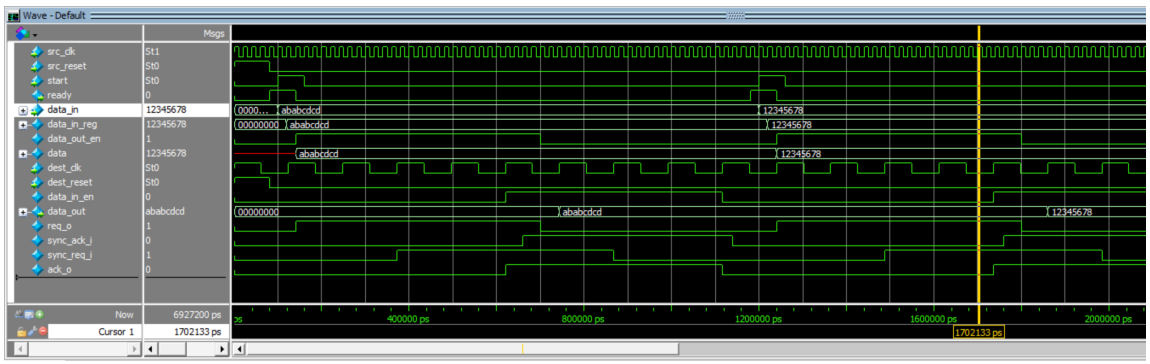
3. Simulation Snapshot & Explanation



above is the reference waveform for handshake. we will be comparing it below to prove correctness.



As we can see, we have the same src_clk = 20 ns clock period and dest_clk = 124 ns clock periods. dest_clk is around 6 times slower than src_clk. We also see the 4-phase req-ack handshake between sender_fsm and receiver_fsm. We also see the first, second, third and fourth 32-bit data be captured in the slow destination clock domain when data_ie_en = 1 and being available on the data_out 1 cycle later, after being sent from the fast source clock domain with data_out_en = 1. Below is a zoomed in snapshot that shows only two cycles in detail as opposed to all four zoomed out above.



SHA-256

1. What is SHA-256?

SHA stands for Secure Hash Algorithm, and there are multiple versions of SHA, each designed for specific cryptographic requirements. The one implemented in this project is SHA-256, a widely-used cryptographic hashing algorithm. SHA-256 can process input messages up to 2^{64} bits (approximately 2.3 billion gigabytes) and produces a fixed 256-bit output, known as a message digest. Regardless of the size of the input, the output hash is always 256 bits, making it a deterministic function.

To qualify as a secure cryptographic hashing function, SHA-256 adheres to several key principles:

a. Compression: It maps an arbitrary-length input to a fixed-length output.
b. Avalanche Effect: A small change in the input causes significant, unpredictable changes in the output hash.
c. Determinism: The same input always produces the same output hash.
d. Pre-image Resistance: It is computationally infeasible to reverse-engineer the input from its hash.
e. Collision Resistance: Finding two distinct inputs that produce the same hash is computationally infeasible.
f. Efficiency: The algorithm can compute the hash quickly, even for large inputs.

These properties make SHA-256 a cornerstone of modern cryptography, ensuring data integrity and authenticity in applications such as digital signatures, password storage, and blockchain technologies. The algorithm is also integral to secure communication protocols like TLS and SSL. More details on the principles and structure of SHA-256 can be found in the project writeup slides (pages 24–33).

2. Algorithm for SHA-256 Implemented in the Code

The SHA-256 implementation in our project is designed to compute a secure 256-bit hash value for a given input message, following the cryptographic principles of the algorithm. It processes input data in 512-bit blocks, using a state-based approach controlled by a finite state machine (FSM) to ensure that each step of the hashing process is executed sequentially and accurately. The implementation divides the algorithm into distinct states: IDLE, WAIT, READ, BLOCK, COMPUTE, and WRITE, with each state handling a specific stage of the process.

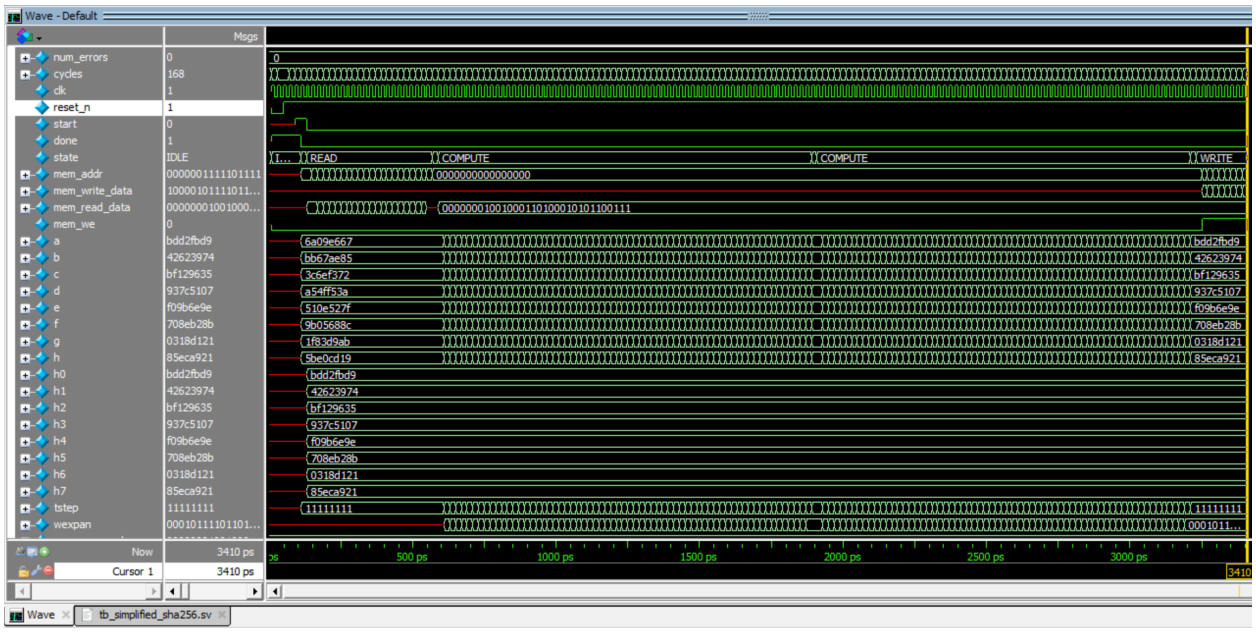
In the IDLE state, the algorithm initializes key variables, including the eight initial hash values (h0 through h7) defined by the SHA-256 standard and the working registers (a through h). Once the initialization is complete, the FSM transitions to the READ state, where the input message is fetched from memory into a message array. The message is then divided into 512-bit blocks in the BLOCK state. For the first block, the first 16 words of the message are directly copied, while the second block includes padding and the 64-bit representation of the message length, as required by the SHA-256 standard.

The COMPUTE state performs the core of the SHA-256 algorithm, executing 64 rounds of hashing for each block. Each round updates the working registers using the SHA-256 compression function, which includes operations like word expansion, bitwise rotations, XOR, and logical functions such as Ch and Maj. After completing 64 rounds, the hash values are updated by adding the working registers to the

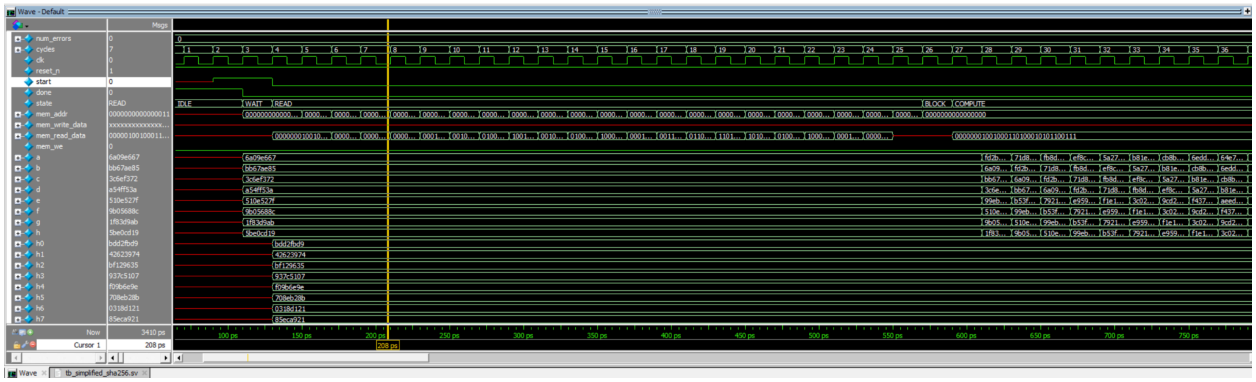
current hash values. If more blocks remain, the FSM returns to the BLOCK state; otherwise, it transitions to the WRITE state.

In the WRITE state, the final 256-bit hash is written back to memory, with each of the eight 32-bit hash values stored sequentially. The FSM then returns to the IDLE state, where it waits for the next input message. The implementation ensures efficient and secure hashing by adhering to the SHA-256 standard and employing a modular FSM-based design. This approach facilitates the computation of deterministic and collision-resistant hash values, making it suitable for cryptographic and data integrity applications.

3. Simulation Waveform Snapshot

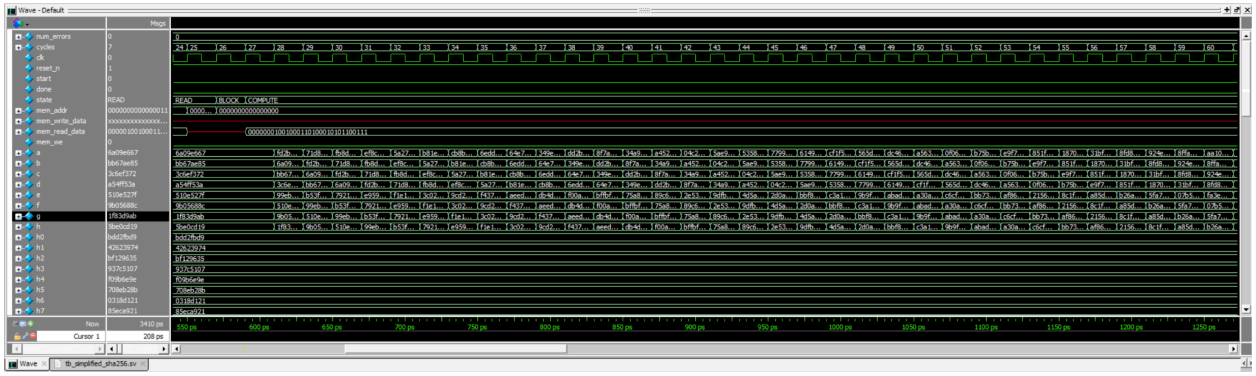


The waveform simulation of the SHA-256 testbench reflects the step-by-step operation of the hashing algorithm as implemented in the simplified_sha256 module. Each signal tracks the progression of the finite state machine (FSM), memory interactions, and computation of intermediate and final hash values. Analyzing the waveforms allows us to verify the correctness of the design and confirm that the expected results are achieved.

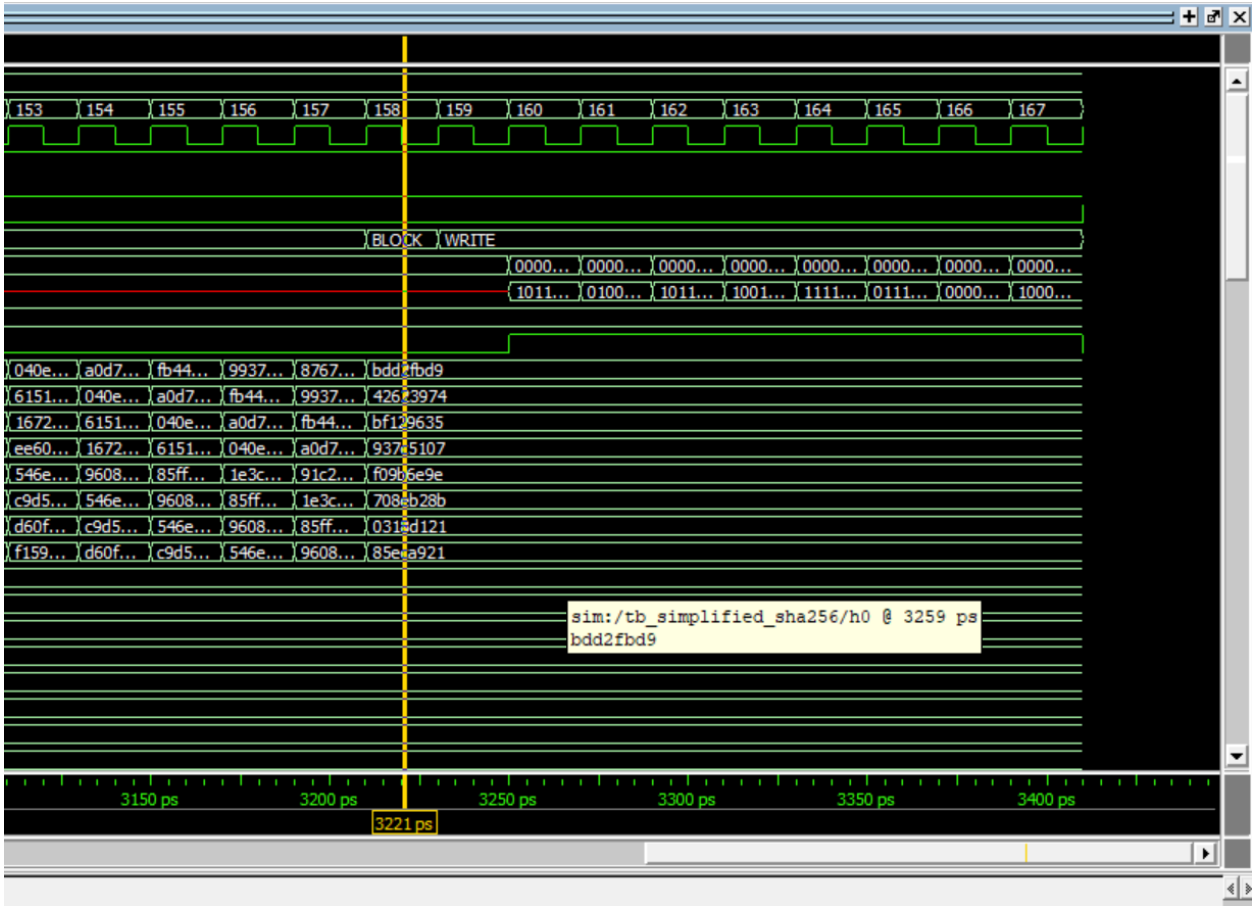


At the start of the simulation, the reset_n signal is asserted low to initialize the module, setting all registers and internal variables to their default states. The FSM transitions to the IDLE state, waiting for the start signal to be asserted. When start is asserted, the state signal transitions to READ, and the input

message is read from memory into the message array. This is reflected in the mem_addr and mem_read_data waveforms, which show the memory addresses being accessed and the corresponding data being read. The offset variable increments with each memory read, ensuring the entire message is fetched.



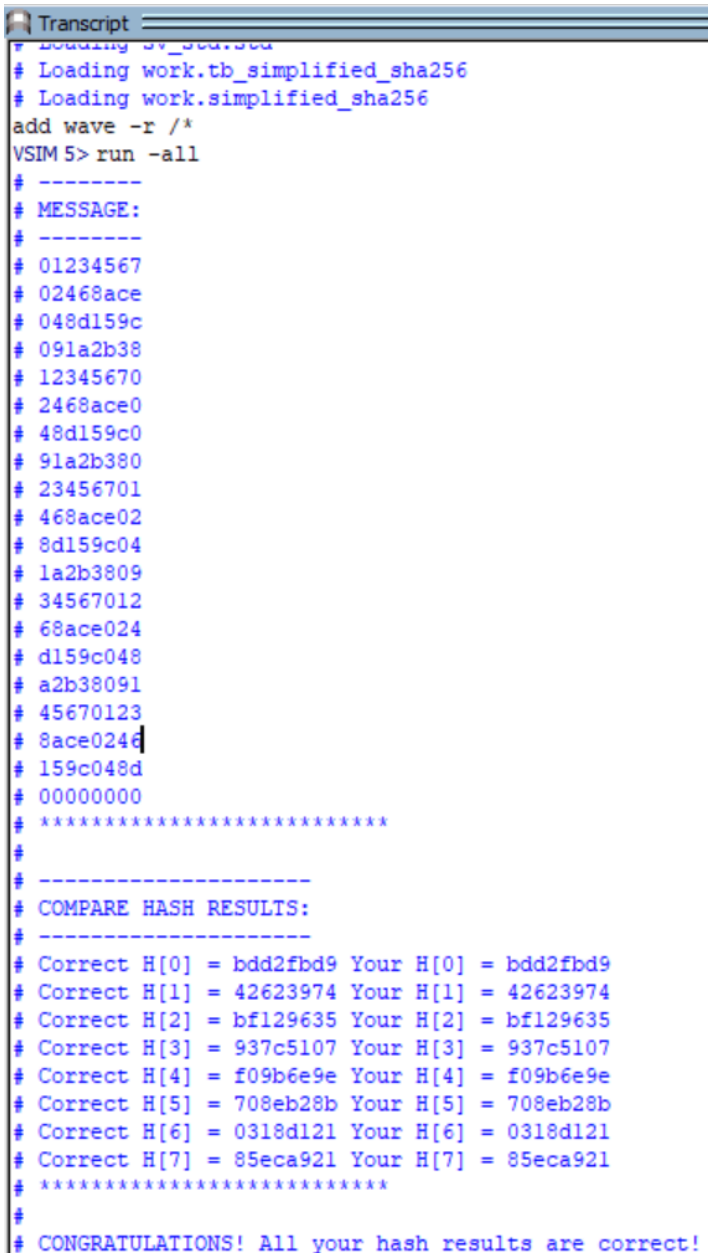
Once the input message is fully loaded, the FSM moves to the BLOCK state, where the message is divided into 512-bit blocks for processing. The state waveform transitions to COMPUTE as the hash computation begins. During this state, the waveforms for a to h display the intermediate values of the working registers, which are updated across 64 rounds of processing. The w array, responsible for word expansion, shows dynamic updates as new words are generated and processed. These operations correspond to the core SHA-256 compression function, which involves bitwise operations and additions.



After completing the computation for a block, the FSM transitions to the WRITE state. The mem_write_data and mem_we waveforms indicate the hash values (h0 to h7) being written back to memory at the specified output_addr. The done signal is asserted when all hash computations are complete, signaling the end of the operation. At this point, the cycles waveform displays the total number of clock cycles taken, providing a performance metric for the design.

The simulation results confirm the correctness of the design, as seen in the final hash values written to memory. The testbench compares these values against the expected outputs, and the following correct results are observed:

4. **ModelSim Transcript Window**



```
# ---------------------
# 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:        168
#
#
# ***************************
#
# ** Note: $stop    : C:/Users/Ryo Andrew Onozuka/Documents/GitHub/notes/ucsd/ece111/ProjectPart1/simplified_sha256/tb_simplified_sha256.sv(262)
#    Time: 3410 ps  Iteration: 2  Instance: /tb_simplified_sha256
# Break in Module tb_simplified_sha256 at C:/Users/Ryo Andrew Onozuka/Documents/GitHub/notes/ucsd/ece111/ProjectPart1/simplified_sha256/tb_simplified_sha256.sv line 262
```

5. **Finalsummary.xls file with fmax, number of cycles, aluts, registers detail filled** attached in .zip file.
6. **Synthesis Resource Usage Snapshot**

	Resource	Usage
1	▼ Estimated ALUTs Used	1810
1	-- Combinational ALUTs	1810
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	1759
3		
4	▼ Estimated ALUTs Unavailable	0
1	-- Due to unpartnered combinational logic	0
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	1810
7	▼ Combinational ALUT usage by number of inputs	
1	-- 7 input functions	0
2	-- 6 input functions	266
3	-- 5 input functions	127
4	-- 4 input functions	6
5	-- <=3 input functions	1411
8		
9	▼ Combinational ALUTs by mode	

9	▼ Combinational ALUTs by mode	
1	-- normal mode	1289
2	-- extended LUT mode	0
3	-- arithmetic mode	393
4	-- shared arithmetic mode	128
10		
11	Estimated ALUT/register pairs used	2217
12		
13	▼ Total registers	1759
1	-- Dedicated logic registers	1759
2	-- I/O registers	0
3	-- LUT_REGs	0
14		
15		
16	I/O pins	118
17		
18	DSP block 18-bit elements	0

20	Maximum fan-out node	clk~input
21	Maximum fan-out	1760
22	Total fan-out	12685
23	Average fan-out	3.33

7. **fitter report snapshot - simplified_sha256.fit.rpt**

ucsd > ece111 > ProjectPart1 > simplified_sha256 > output_files > simplified_sha256.fit.rpt

```
+-----------------------------------------------------------------------------+
; Fitter Summary                                                              ;
+-----------------------------------+-----------------------------------------+
; Fitter Status                     ; Successful - Wed Nov 27 07:55:47 2024     ;
; Quartus Prime Version             ; 22.1std.0 Build 915 10/25/2022 SC Lite Edition ;
; Revision Name                     ; simplified_sha256                       ;
; Top-level Entity Name             ; simplified_sha256                       ;
; Family                            ; Arria II GX                             ;
; Device                            ; EP2AGX45CU17I3                          ;
; Timing Models                     ; Final                                   ;
; Logic utilization                 ; 8 %                                     ;
;     Combinational ALUTs           ; 1,810 / 36,100 ( 5 % )                  ;
;     Memory ALUTs                  ; 0 / 18,050 ( 0 % )                      ;
;     Dedicated logic registers     ; 1,759 / 36,100 ( 5 % )                  ;
; Total registers                   ; 1759                                    ;
; 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 % )                           ;
+-----------------------------------+-----------------------------------------+
```

8. **timing Fmax report snapshots - simplified_sha256.sta.rpt**

```
+-----------------------------------------------------+
; Slow 900mV 100C Model Fmax Summary                  ;
+------------+-----------------+------------+------+
; Fmax       ; Restricted Fmax ; Clock Name ; Note ;
+------------+-----------------+------------+------+
; 172.41 MHz ; 172.41 MHz      ; clk        ;      ;
+------------+-----------------+------------+------+
```