Discussion 7: Handshake Synchronizer and SHA-256

ECE 111: Advanced Digital Design Project
Fall 2024

(Courtesy: Vishal Karna)

Announcements

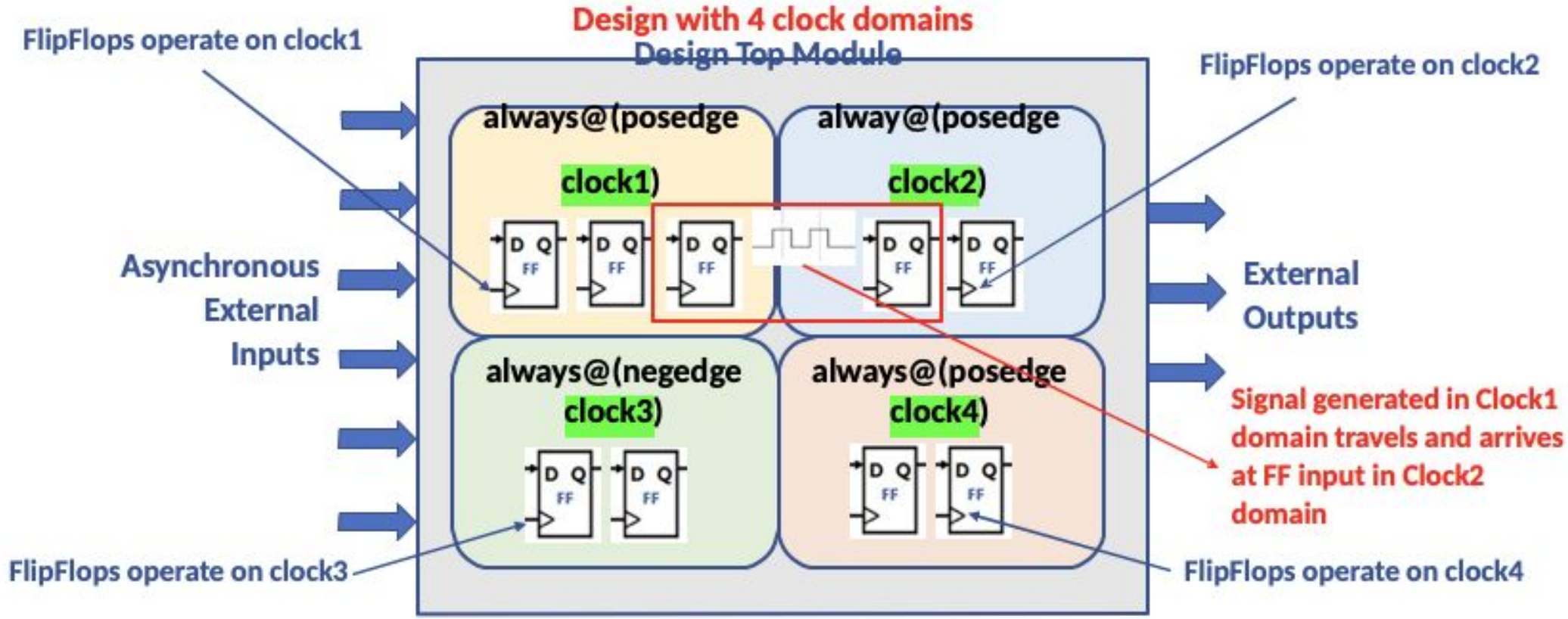
- Final Project Part1 released. It includes:
  - Part-1 : Develop Handshake Synchronizer RTL model
  - Part-2 : Develop SHA256 RTL model

- Due Date: 27 Nov, 2024 (Max 2 late days allowed; 20% grade reduction per late day).

- HW6 due today (13 Nov, 2024). Late due date 15 Nov, 2024.

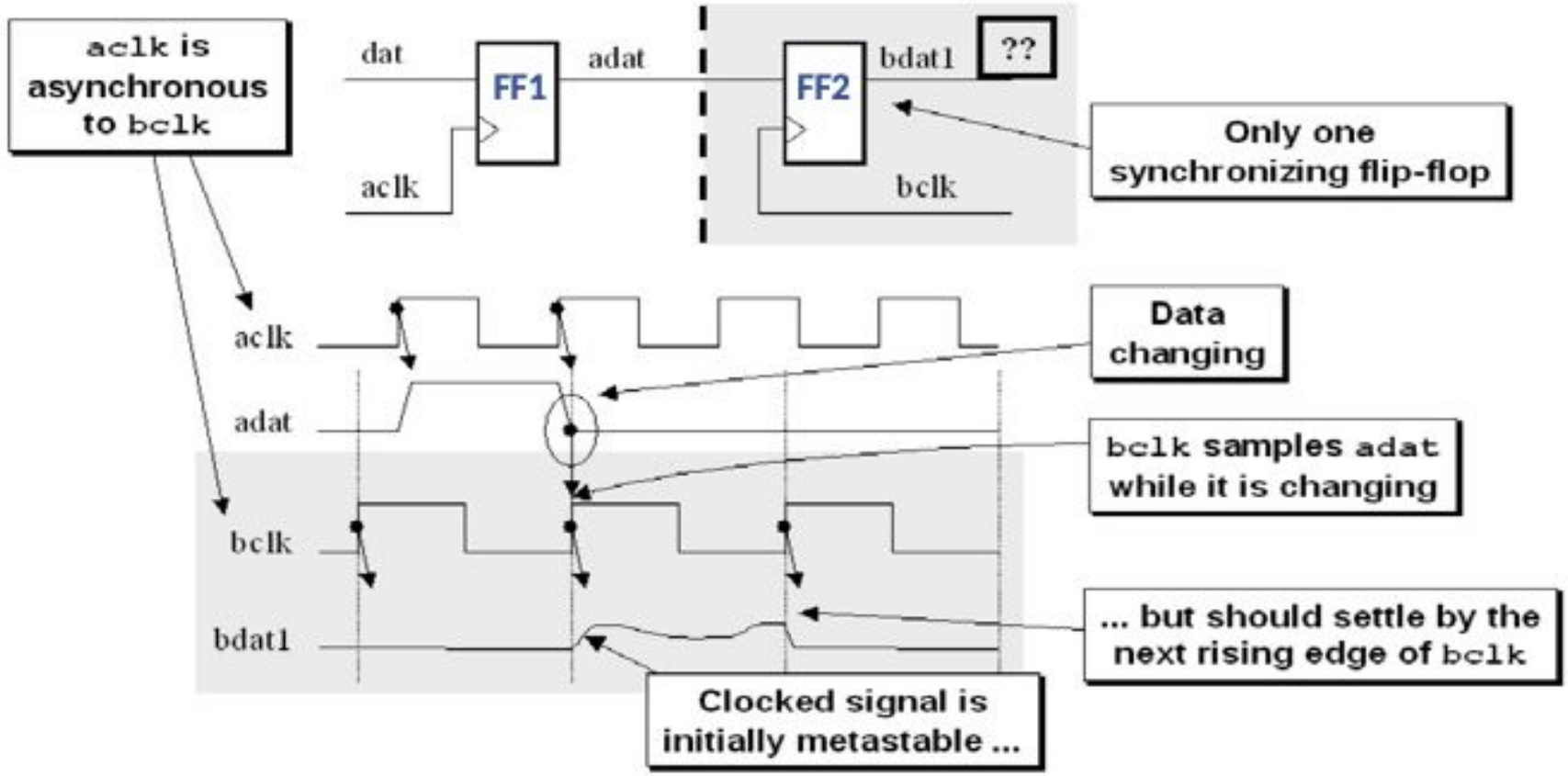
Handshake Synchronizer

Clock Domain Crossing (CDC)



- **What is Clock Domain Crossing (CDC) ?**
  - **Signal travels from one clock domain to another clock domain**
  - **CDC** takes place anytime the inputs to a given flipflop were set based upon some other clock
  - In example above signal generated from FF operating on clock1 travels and arrives at the flipflop inputs which is operating on clock2 → **Signal crossed clock domains**

What is the issue when signal crosses clock domains ?



- Synchronization failure that occurs when a signal (**adat**) generated in one clock domain (**aclk**) is sampled too close to the rising edge of a clock signal (**bclk**) from a second clock domain.
- **adat does not meet setup time requirement for FF2 hence output of FF2 (bdat1) goes metastable**
  - output (**bdat1**) going metastable and not converging to a legal stable state by the time the output must be sampled again.

Source of this slide courtesy of Clock Domain Crossing publication from Clifford E. Cummings Sunburst Design

Synchronization Techniques

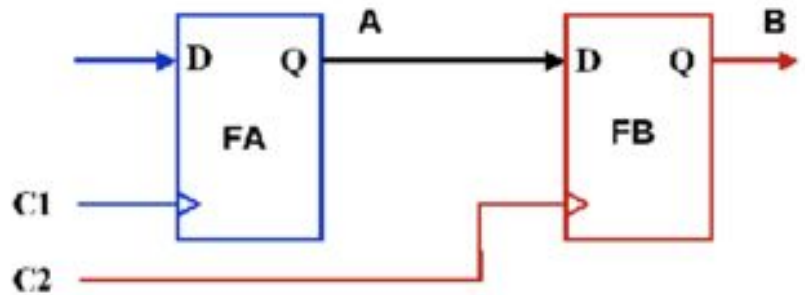
- **Open-loop and Close-loop Synchronization**

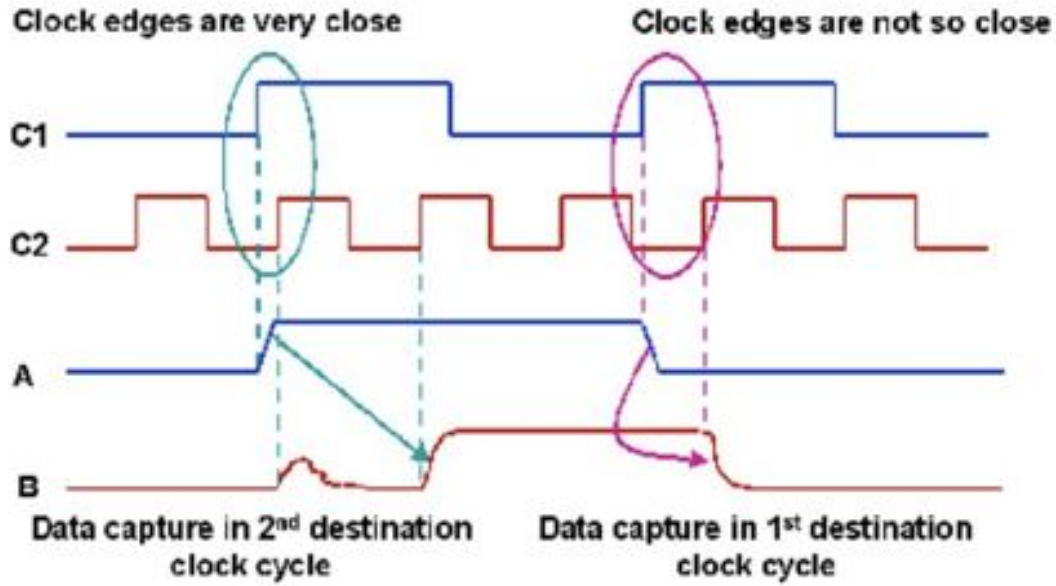
- **For Single-bit control signal synchronization :**
  - 2-FF or 3-FF Synchronizer (open-loop)

- **For Multi-bit data bus synchronization :**
  - Handshake Mechanism (closed loop)
  - Asynchronous FIFO (open loop)

Signal Crossing Slow Clock Domain to Fast Clock Domain

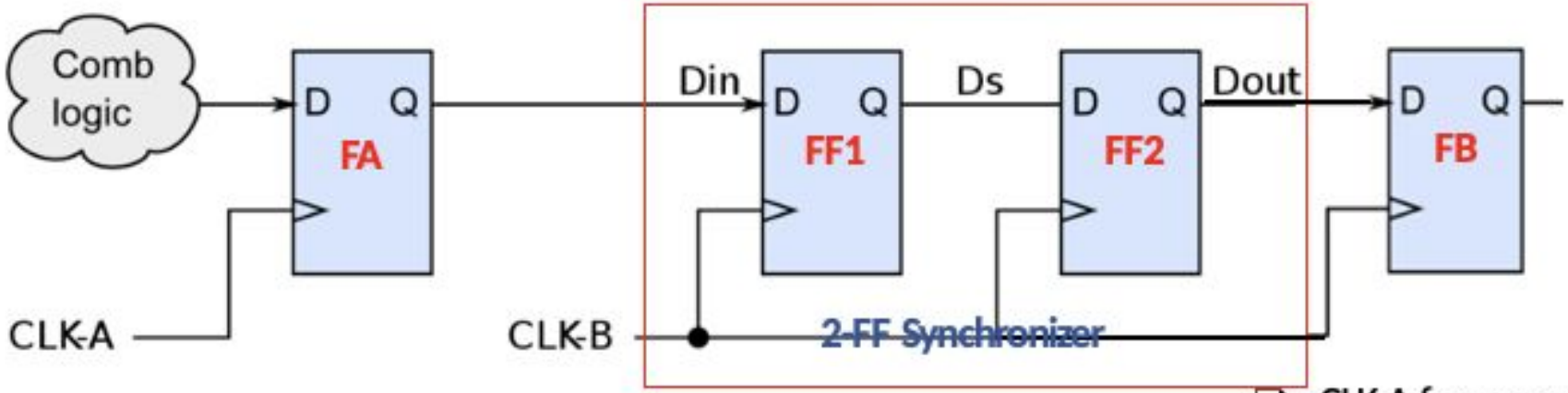
- **Scenario -1 : Flipflop-A clock frequency < Flipflop-B clock frequency**

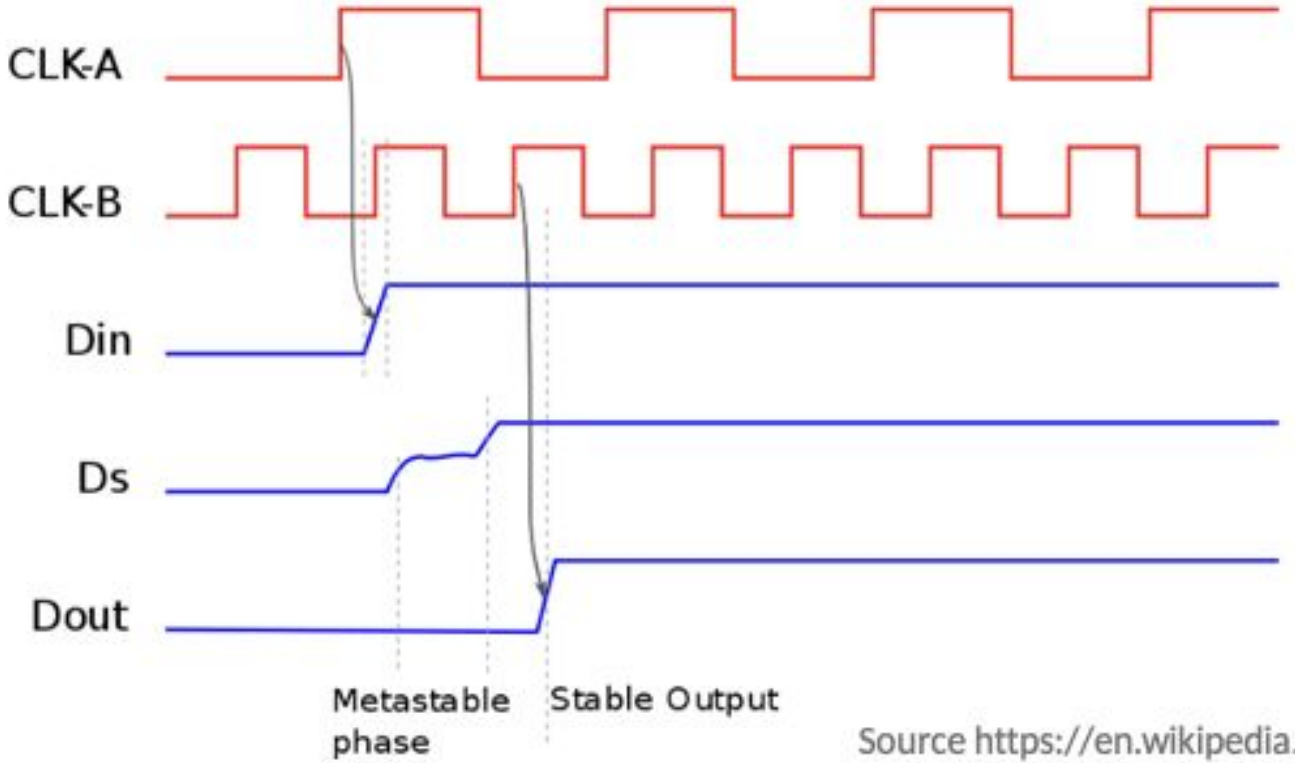




- If the transition on signal A happens very close to the active edge of clock C2, it could lead to setup or hold violation at the destination flop "FB" and "FB" will enter metastable state.
- Output signal B from FA will be unstable and may or may not settle down to some stable value before the next clock edge of C2 arrives
- Unstable signal B fanout cones may read different values, and may cause the design to enter into an unknown functional state, leading to functional issues in the design
- **To address such slow to fast clock domain crossing scenario, two or three flipflop synchronizer can be used !**

2-FlipFlop Synchronizer For Slow to Fast Clock Domain



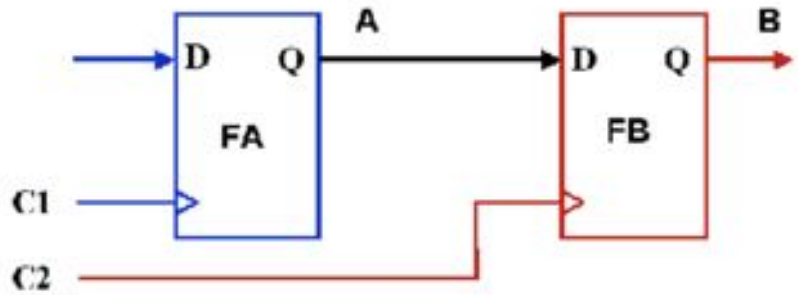


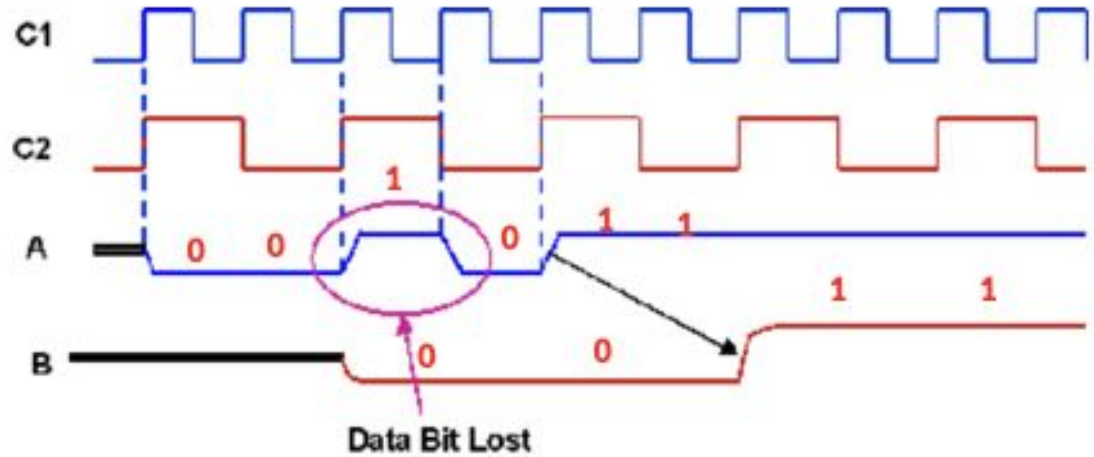
- CLK-A frequency is slower than CLK-B
- Din entering into CLK-B domain goes through 2-FF synchronizer stage (FF1, FF2)
- 2-FF synchronizer will operate on receiving clock which is CLK-B
- Even if Din changes close to CLK-B clock edge violating setup violation of FF1, causing metastable output Ds, output of second stage of synchronizer (FF2) will be stable.
- If Din changes fast and frequently, sometimes 2-FF synchronizer is not enough for metastable output to settle. In such cases 3rd FF can be added (i.e.3-FF synchronizer)

Source https://en.wikipedia.org/wiki/Metastability_(electronics)

Signal Crossing Fast Clock Domain to Slow Clock Domain

- **Scenario-2 : Flipflop-A clock frequency > Flipflop-B clock frequency**

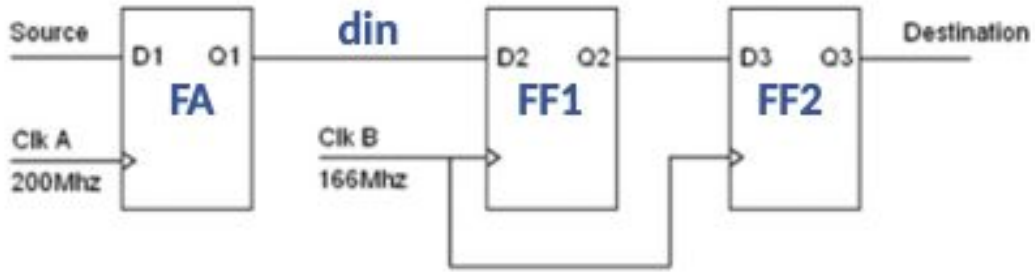


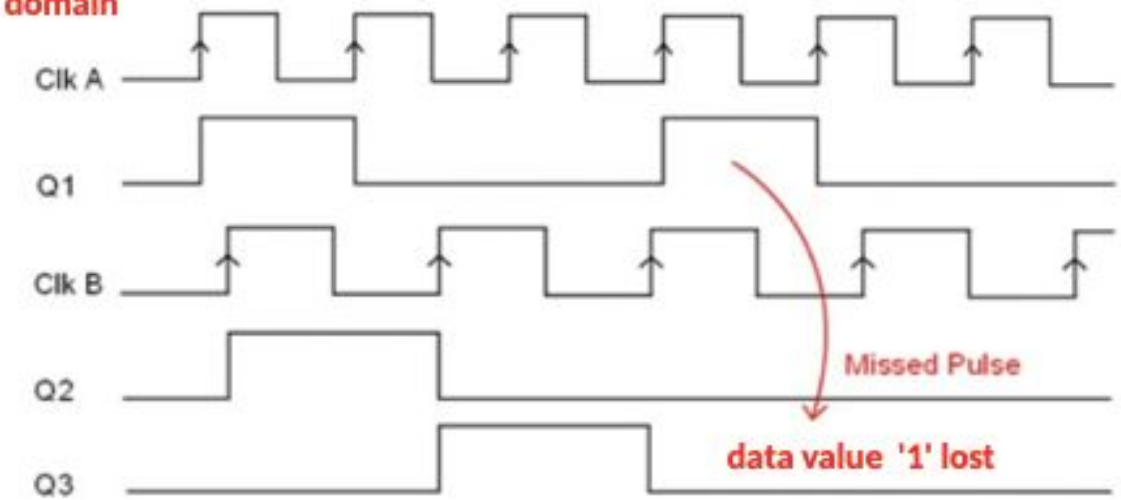


- C1 is two times faster than C2 and there is no phase difference between C1 and C2
- If signal A is changing rapidly, say for example signal A sequence is "**001011**", output from FB in C2 clock domain will be "**0011**".
  - Here the third data value in the input sequence which is **"1"** is lost → **data losss**
  - For some circuits data loss is not acceptable
- **Two or three Flipflop synchronizer technique will avoid data loss, instead either of the technqiues can be used to avoid data loss when data is sent from fast to slow clock domain :**
  - Storage between FA and FB (Asynchronous FIFO)
  - Handshake Synchronization technique between FA and FB

2-FF Synchronizer for Signal crossing Fast Clock to Slow Clock Domain

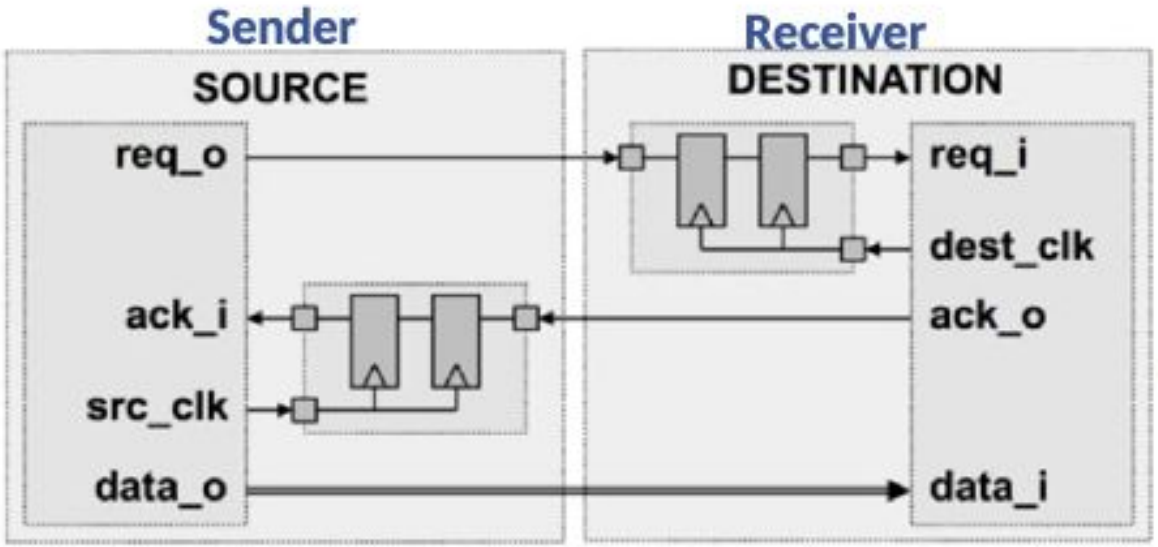
Example of data Loss when data flowing from fast to slow to fast clock domain

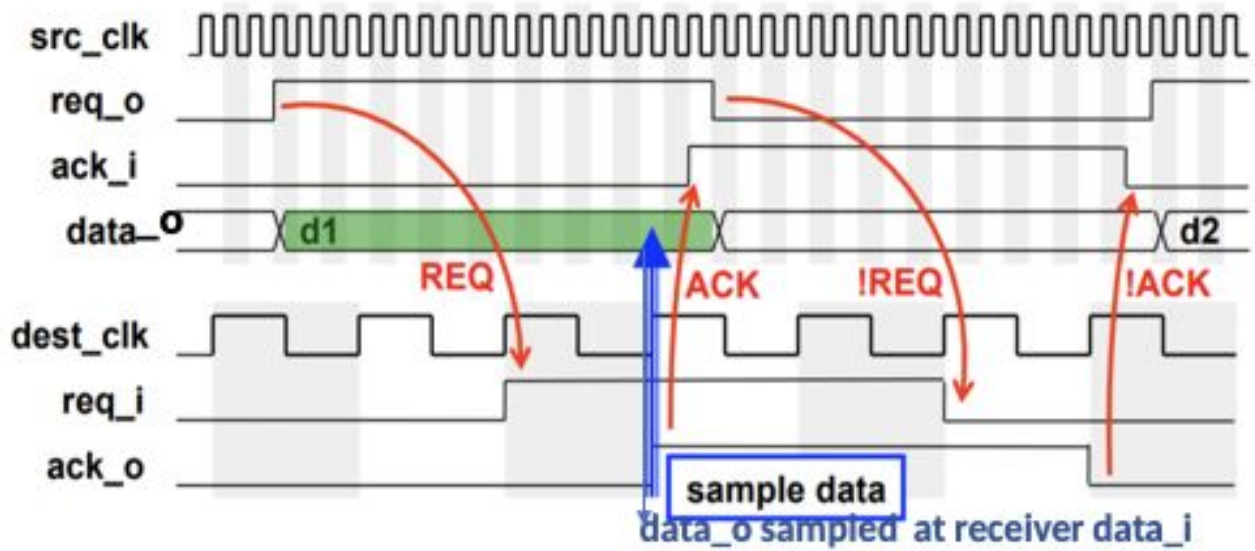




- Consider, signal crossing from source clock domain ClkA to Destination ClkB domain where ClkA is faster than ClkB
- ClkA is 200 Mhz and Clk B is 166 MHz
- Signal is Crossing Fast to Slow clock domain through 2-FF synchronizers (FF1 and FF2)
- Adding 2-FF or 3-FF synchronizer will still result in data loss when signal crossed fast to slow clock domain as seen in the waveform above
- If source signal "din" width entering FF1 can be guaranteed to be 1.5 times of receiving clock ClkB then 2-FF or 3-FF synchronizer still can be used for signals crossing fast to slow clock domain
- If source signal cannot be guaranteed to 1.5 times of receiving clock and if the data loss is not acceptable for subsequent circuit in destination clock domain **then 2-FF or 3-FF open ended synchronization technique should not be deployed !**

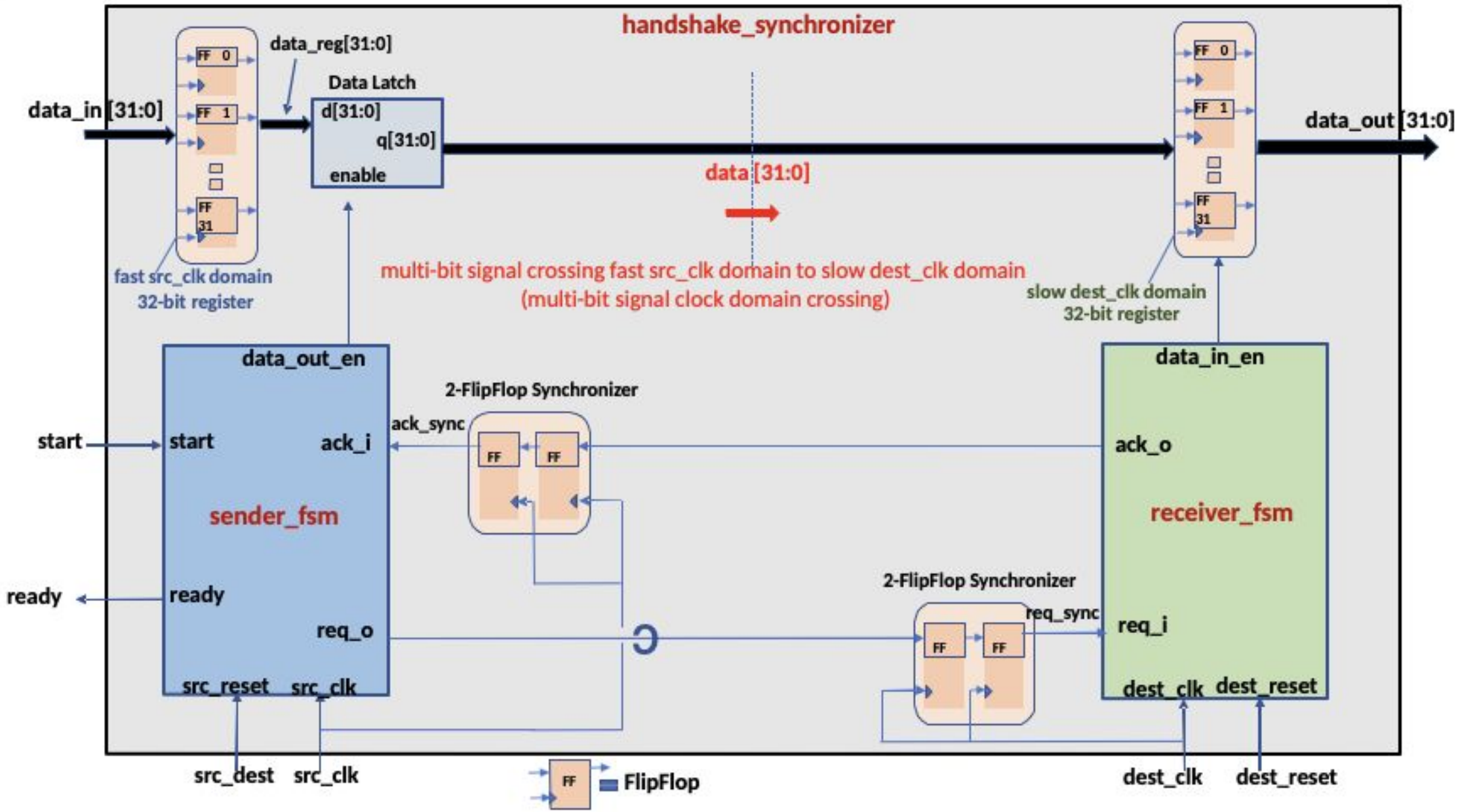
Handshake Synchronizer For Multi-bit Signal Crossing Fast to Slow Clock Domain



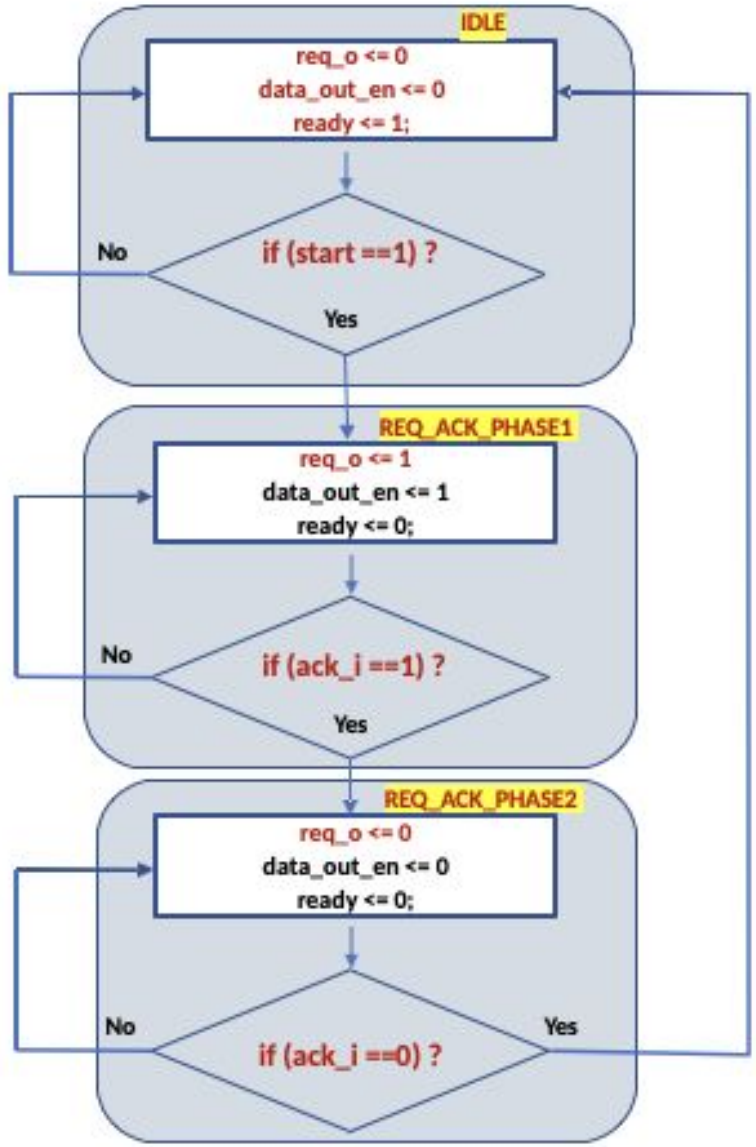


- For many open-ended data-passing applications, a simple **req-ack** handshaking sequence is sufficient
- The sender places data onto a data bus (**data_o**) and then synchronizes a "**req_o**" signal (request) to the receiving clock domain.
- When the "**req_o**" signal is recognized in the destination clock domain, the receiver clocks the data_o into a register (the data should have been stable for at least two/three sampling clock edges in the destination clock domain)
- Receiver then passes an "**ack_o**" signal (acknowledgement) through a synchronizer to the sender.
- When the sender recognizes the synchronized "**ack_o**" signal, the sender can change the value being driven onto the data bus (**data_o**)

Homework : Handshake Synchronizer



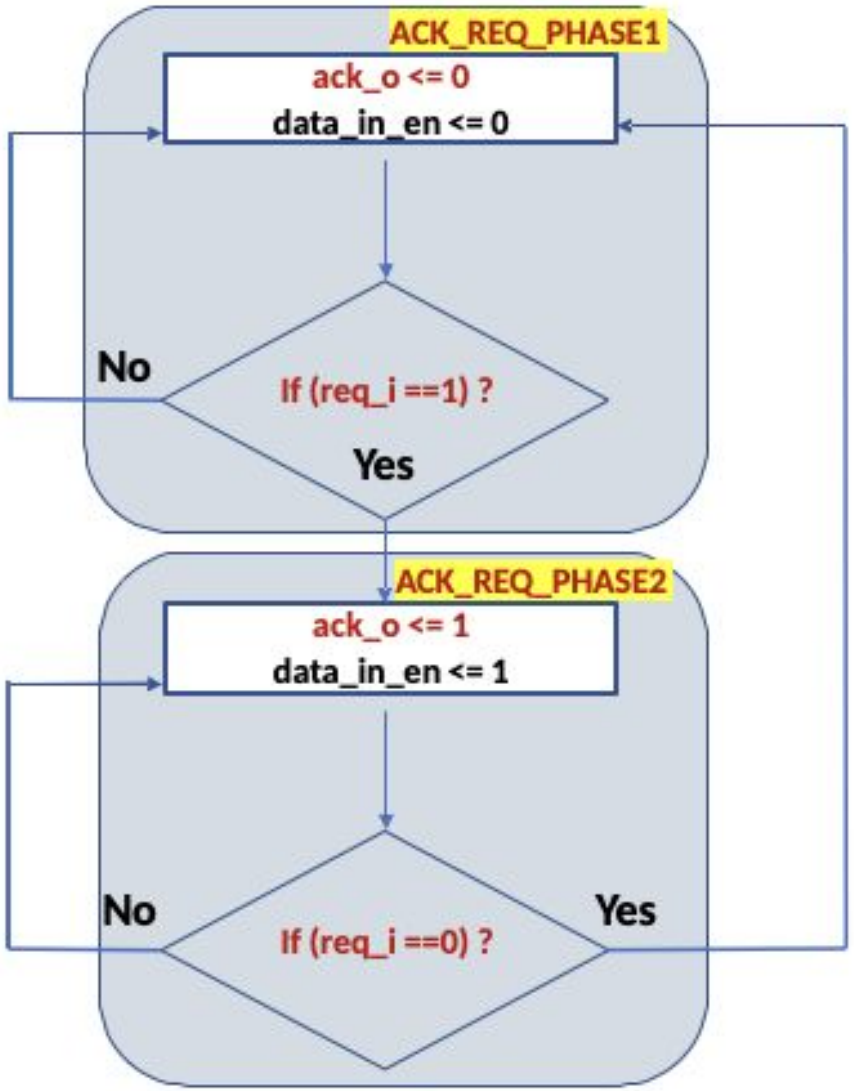
Homework : Handshake Sender FSM



❑ Handshake Sender FSM

- **Purpose :** Sends data from faster src_clk domain to slower dest_clk domain using req ack handshake protocol. Holds data until dest_clk domain receiver_fsm has not acknowledged acceptance of data. Once data is accepted by receiver_fsm, then sender_fsm gets ready to send next data to dest_clk domain
- **IDLE State :**
  - Asserts ready = 1 indicating testbench that handshake sender_fsm is ready to receive next 32-bit data
  - Waits for start == 1 from testbench, which indicates next 32-bit data is available to be sent to dest_clk domain. And then move to REQ_ACK_PHASE1
- **REQ_ACK_PHASE1 :**
  - Asserts req_o amd data_out_en to '1' which indicates request is available for receiver_fsm to accept
  - Then waits for ack_i == 1 from receiver_fsm and then moves to REQ_ACK_PHASE2 once request acceptance has been acknowledged
- **REQ_ACK_PHASE2 :**
  - De-asserts req_o amd data_out_en to '0' since receiver_fsm has captured 32-bit data
  - Then waits for ack_i == 0 from receiver_fsm which is indication that receiver_fsm is now ready to receive next 32 bit data from src_clk domain and then moves to IDLE state

Homework : Handshake Receiver FSM



❑ Handshake Receiver FSM

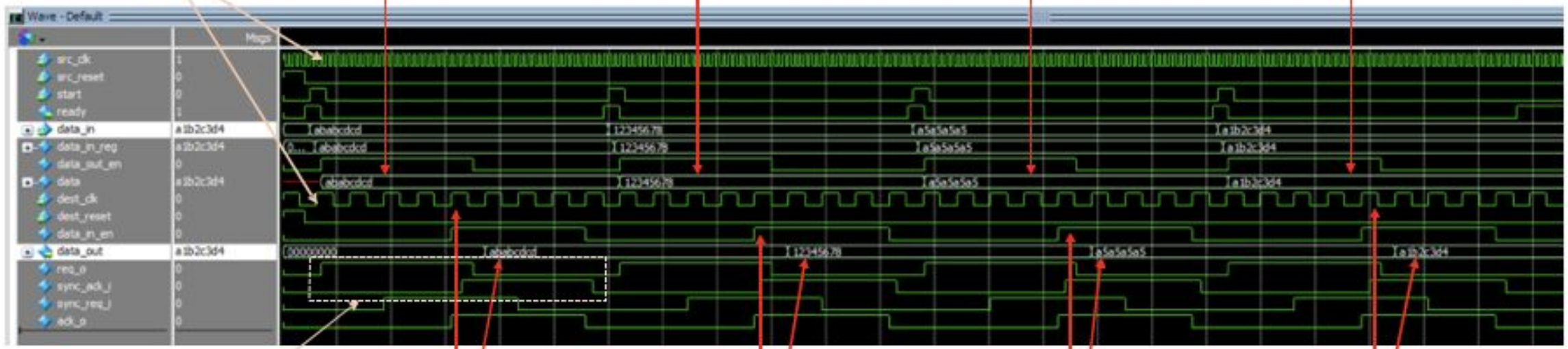
- **Purpose :** Receive data from fast src_clk domain and capture it in slow dest_clk domain register (flipflop) using req ack handshake protocol. Provides acknowledgement to receiver_fsm once data is captured in the destination register (flipflop)

- **ACK_REQ_PHASE1 :**
  - De-asserts ack_o amd data_in_en to '0' and then waits for req_i == 1
  - This indicates to sender_fsm that it is ready to receive next 32 bit data and then moves to ACK_REQ_PHASE2

- **ACK_REQ_PHASE2 :**
  - Asserts ack_o amd data_in_en to '1' which indicates 32-bit data from src_clk domain sender_fsm has been captured in dest_clk domain
  - Then waits for req_i == 0 from handshake sender_fsm and then moves to REQ_ACK_PHASE1 to complete acknowledgement handshake protocol

Homework: Handshake Synchronizer

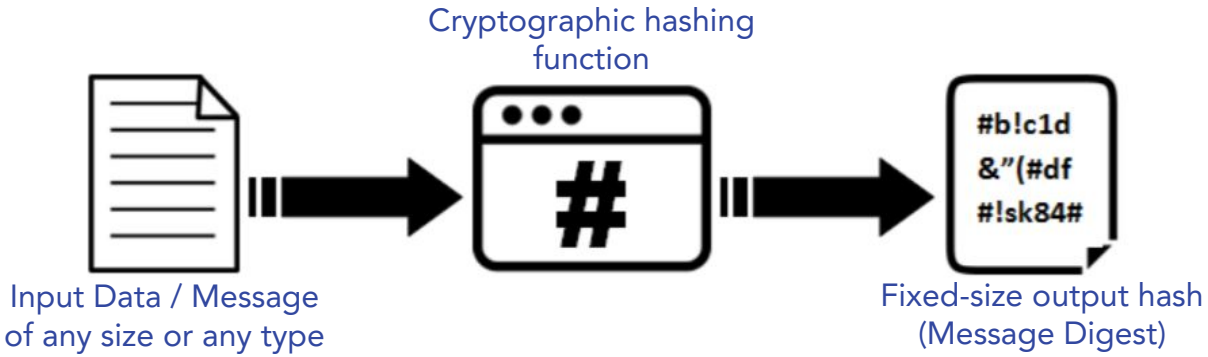
- Reference simulation waveform for handshake_synchronizer module



SHA-256

Cryptographic hashing functions

- Cryptographic hashing functions are functions or algorithms to convert input data of arbitrary size, called message, into a fixed-size output, called hash value or message digest



- Secure cryptographic hash functions have the following properties:
  - Compression
  - Avalanche Effect
  - Determinism
  - Pre-Image Resistant (One Way Function)
  - Collision Resistance
  - Efficient (Quick Computation)

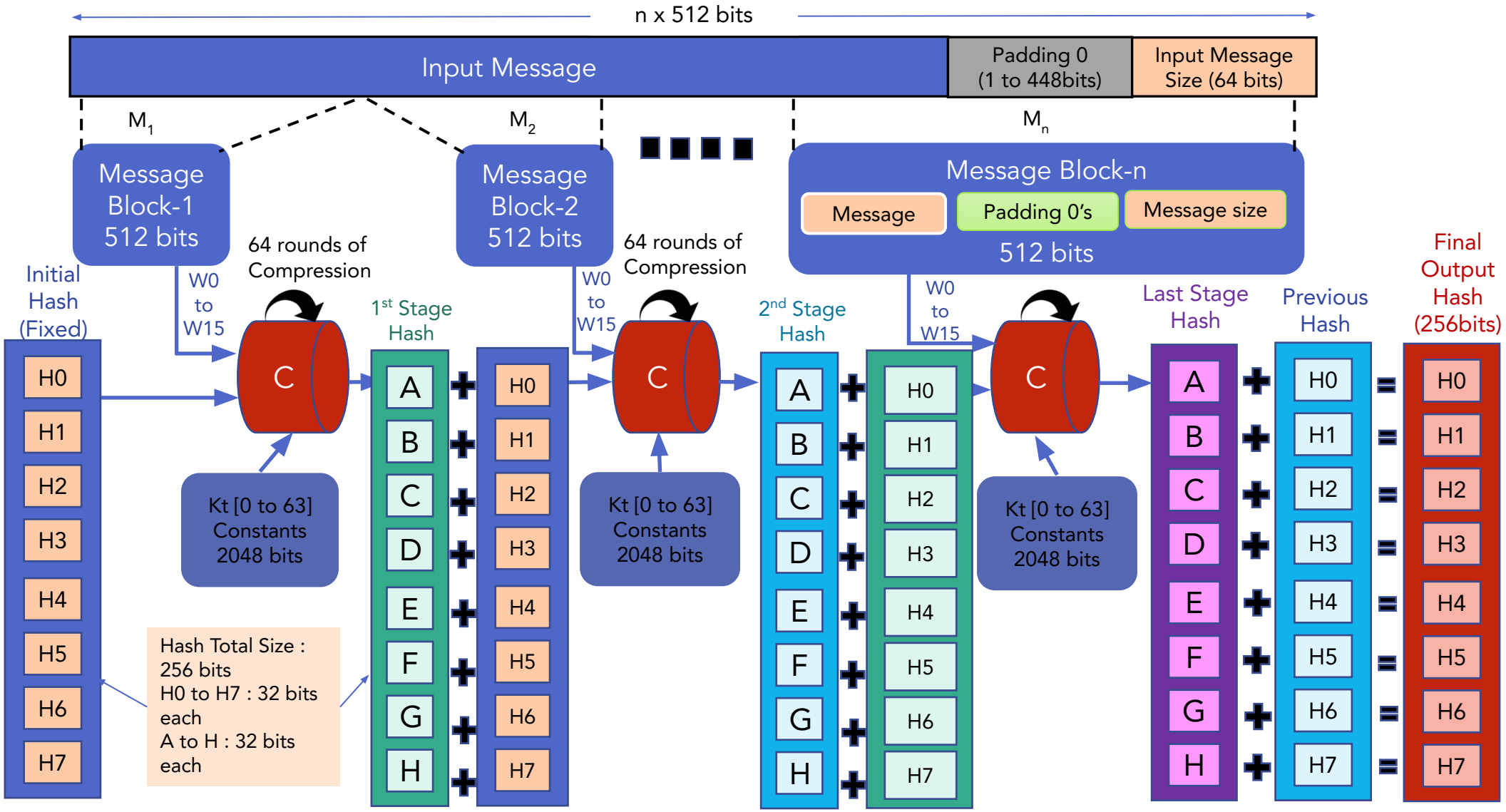
Secure Hash Algorithm (SHA)

- SHA is family of cryptographic hash functions designed by the United States National Security Agency (NSA) and published by the United States National Institute of Standards and Technology (NIST)

- 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
    - consists of six hash functions with digests with 224, 256, 384 or 512 bits: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256.
  - SHA-3 : Input message of up to 2^{128} bits produces 512-bit output hash value

- SHA-256 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

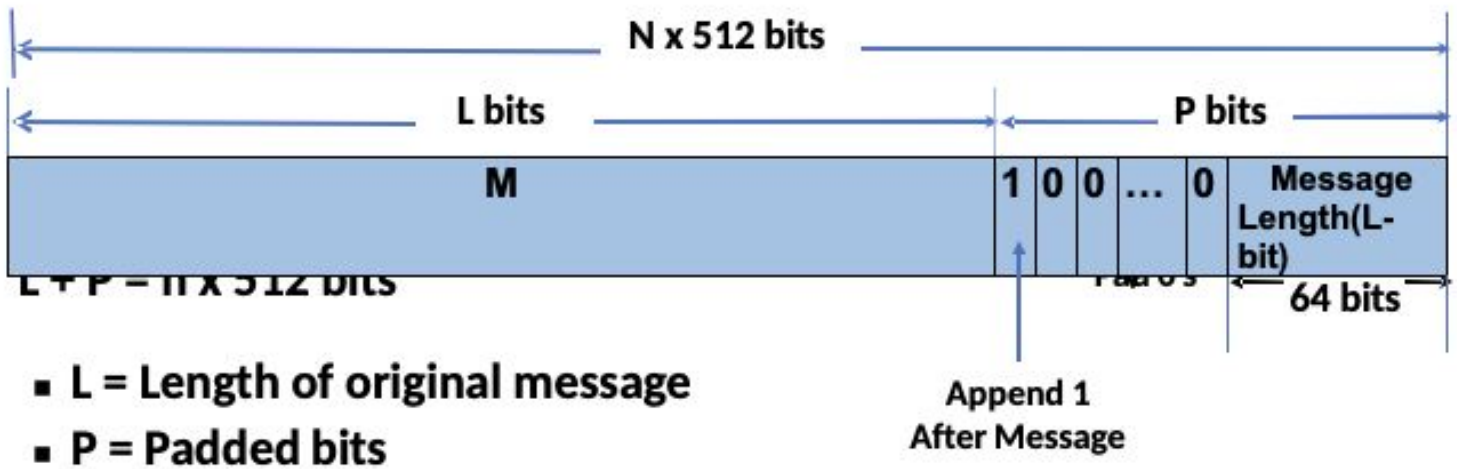
SHA-256 Algorithm



SHA256 Algorithm

- **Step 1:** Append padding bits (1 and 0's)
  - A **L**-bit 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 is 64-bits less than some multiple of 512).

- **Step 2** : Append message length bits in 0 to 63 bit position
  - Since SHA256 supports until 2^64 input message size, 64 bits are required to append message length



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

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

- **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 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
  - K_t constants

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

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)

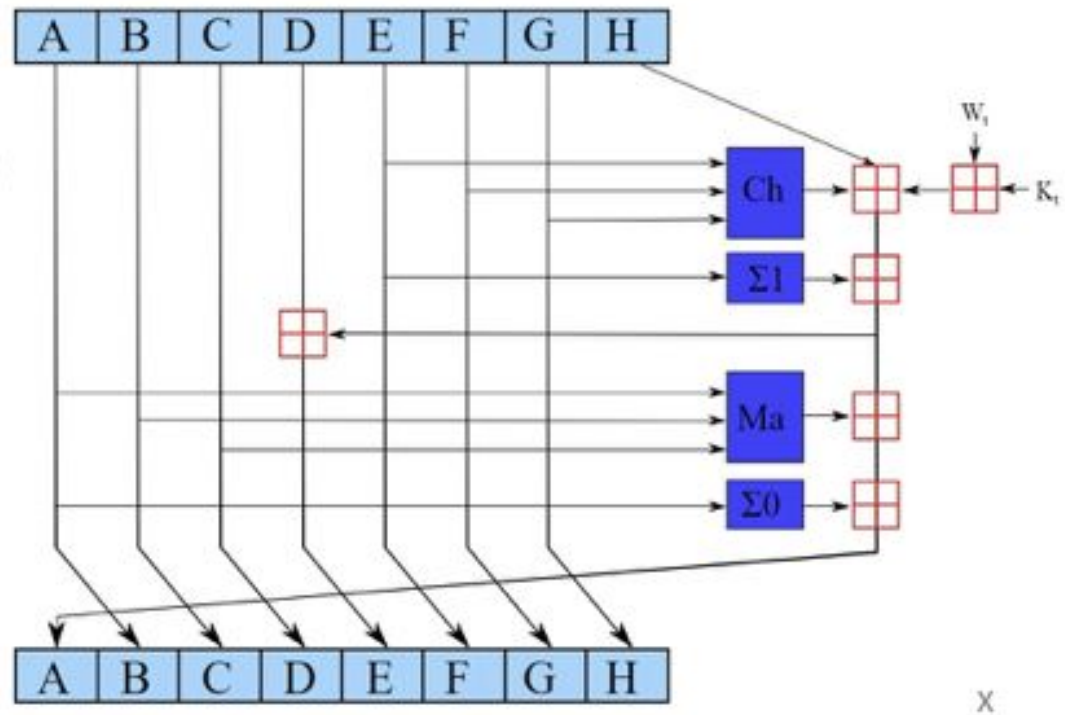
$t_2 = S_0 + maj$

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

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

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

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



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

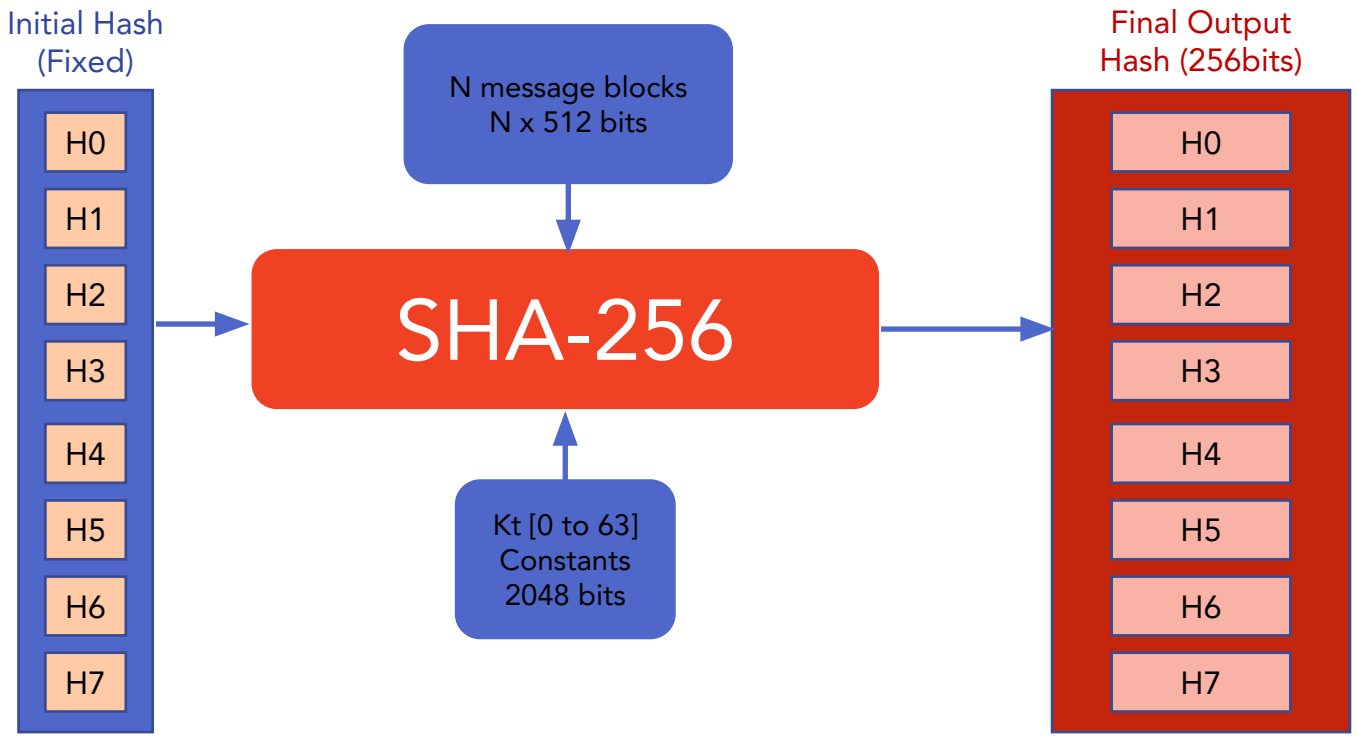
    H5 = H5 + f

    H6 = H6 + g

    H7 = H7 + h

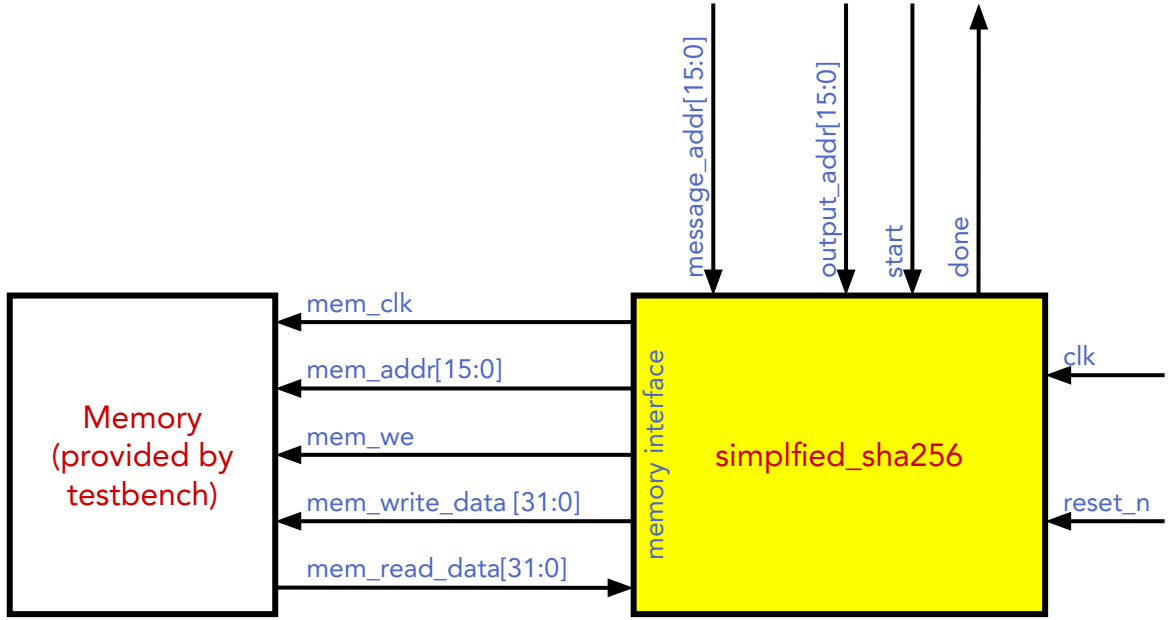
- **Step 5 :** Output
  - When all M_j have been processed, the 256-bit hash of M is available in H0, H1, H2, H3, H4, H5, H6, H7

SHA-256 Algorithm Overview



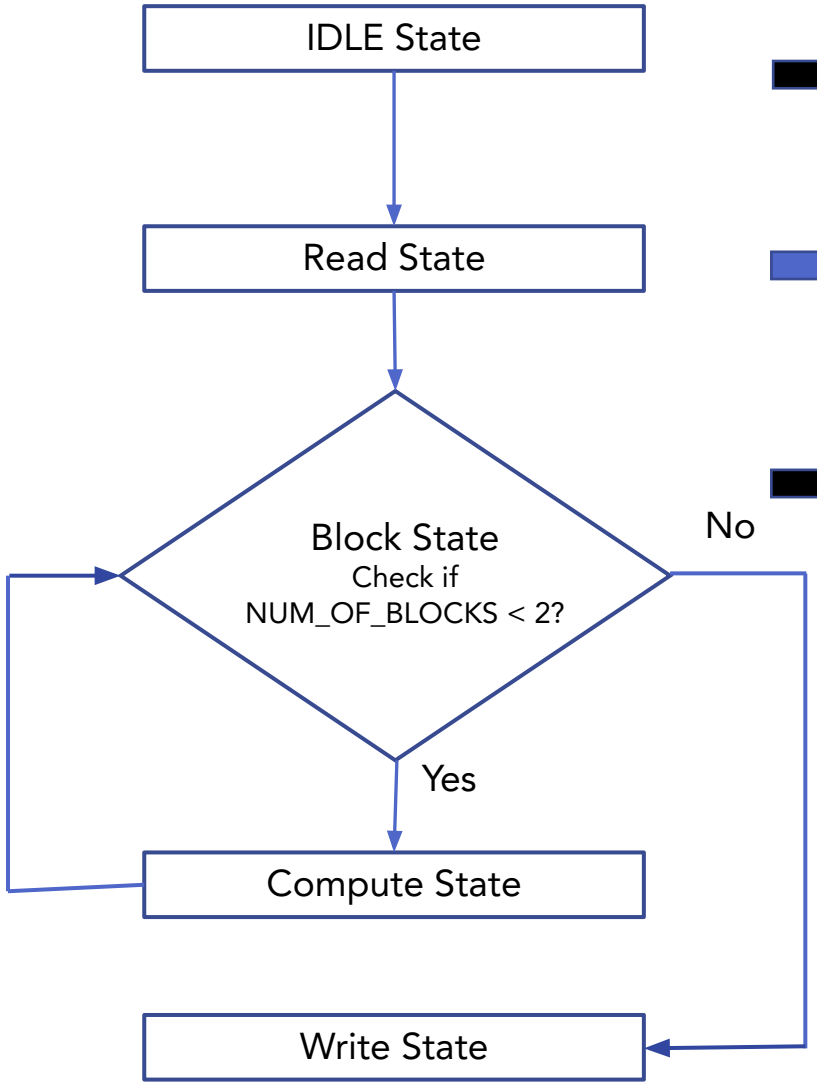
SHA-256: Module Interface

- 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



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

SHA-256: FSM Design



- 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' through 'h' array. Increment number of blocks iteration variable
- Add previous hash values with 'a' through '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