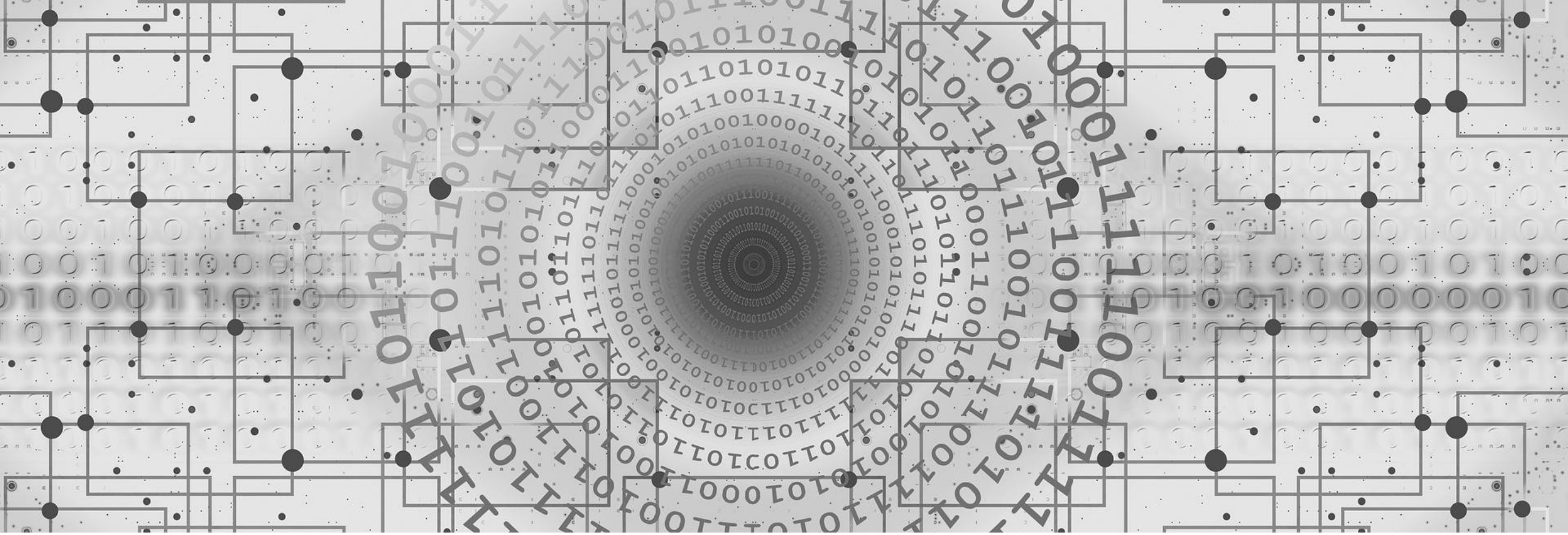
ECE-111 (SP24): Final Project Part I

Designs	Handshake synchronizer and Secure Hash Algorithm (SHA-256)
Deadline	Nov 27, 2024 at 11:59pm
Max. late days	2

Final Project Part1 Guidelines

- **Final Project Part1 Includes:**
  - **SubPart-1 :** Handshake synchronizer requirements including Sender and Receiver FSM Requirements
  - **SubPart-2 :** Develop SHA256 RTL mode

- **Testbench will be provided for both Part-1 and Part-2:**
  - Expected behavior of SHA256 and handshake synchronizer 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



SubPart1 : Handshake Synchronizer

ECE-111 Advanced Digital Design project

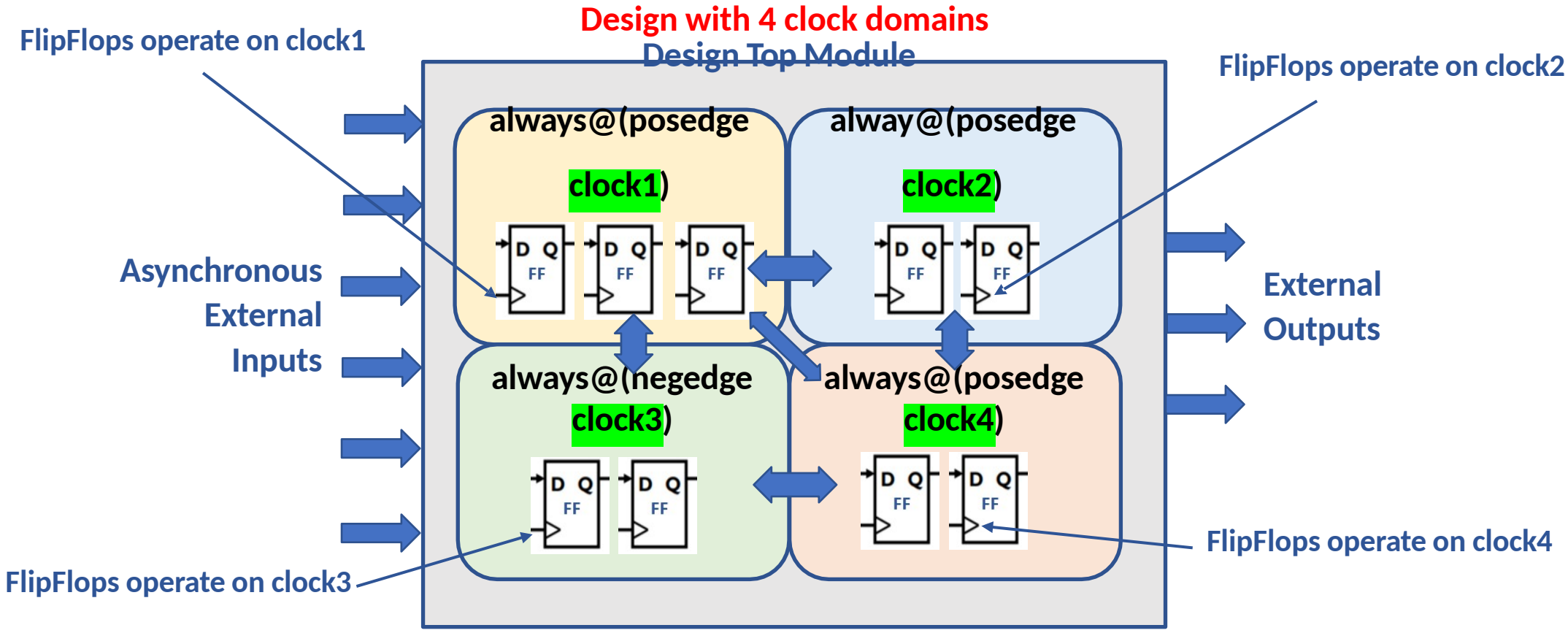
UC San Diego
JACOBS SCHOOL OF ENGINEERING
Electrical and Computer Engineering

Outline

- **First let's learn clock domain crossing (CDC) and data synchronization Concepts**
  - What is clock domain ?
  - What is clock domain crossing (CDC) ?
  - Synchronization issues when data flows from one clock domain to another clock domain
  - Data synchronization techniques
    - 2-FlipFlop Synchronizer
    - Handshake Synchronizer
  - Which synchronization technique to apply when data is sent from :
    - Slow to Fast clock domain
    - Fast to Slow clock domain

- **Homework : Handshake Synchronizer**
  - Handshake synchronizer requirements including Sender and Receiver FSM Requirements
  - 4-phase handshake protocol
  - Block diagram of handshake synchronizer to understand overall data synchronization implementation
  - Simulation waveforms for reference purpose

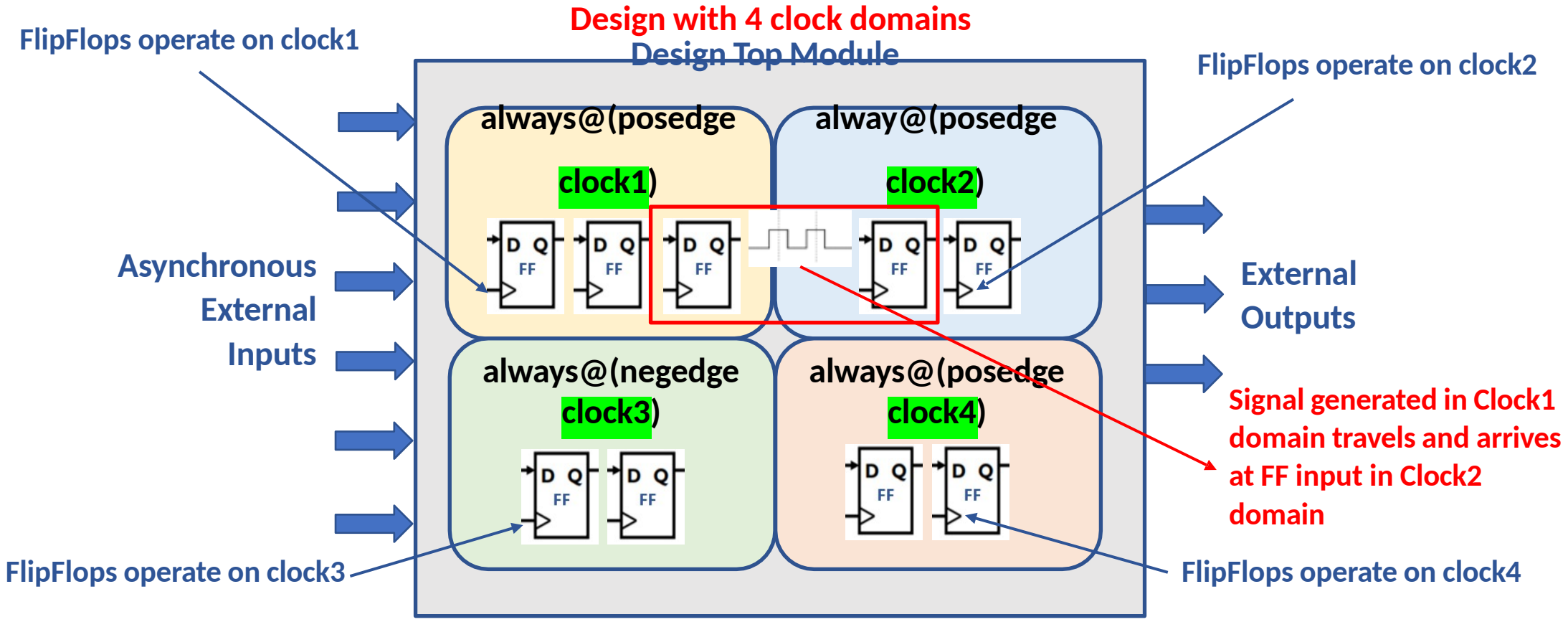
Clock Domain



❑ What is a Clock Domain ?

- **Clock Domain** is that portion of a circuit that is generated and processed by a single clock
- In diagram below, there are 4 clock domains in design top module. These clock domains are :
  - clock1, clock2, clock3 and clock4
  - All 4 clocks can be of different frequency, out of phase and asynchronous to each other
  - Flipflops in each clock domain operate on their respective clock

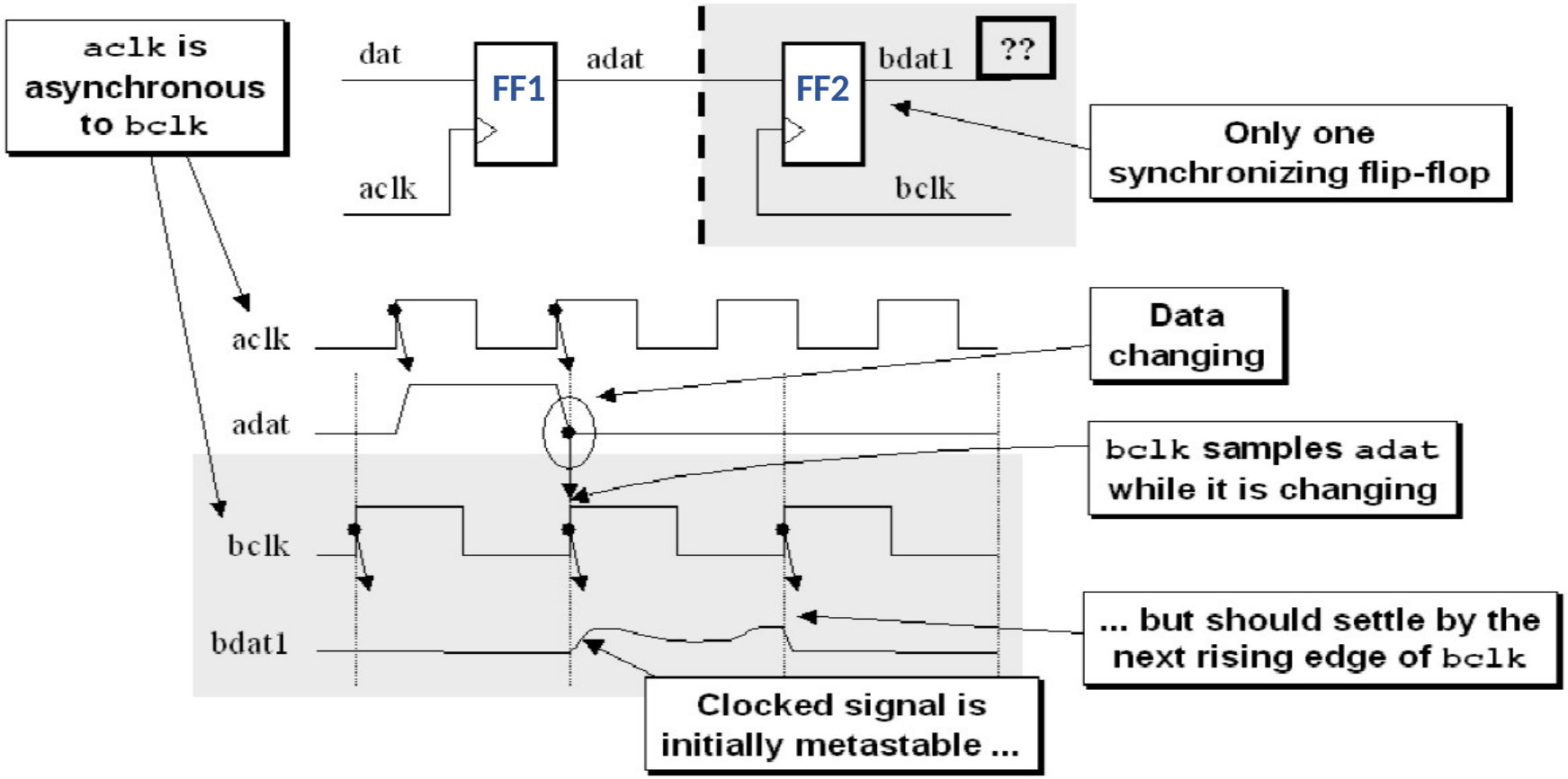
Clock Domain Crossing (CDC)



❑ 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

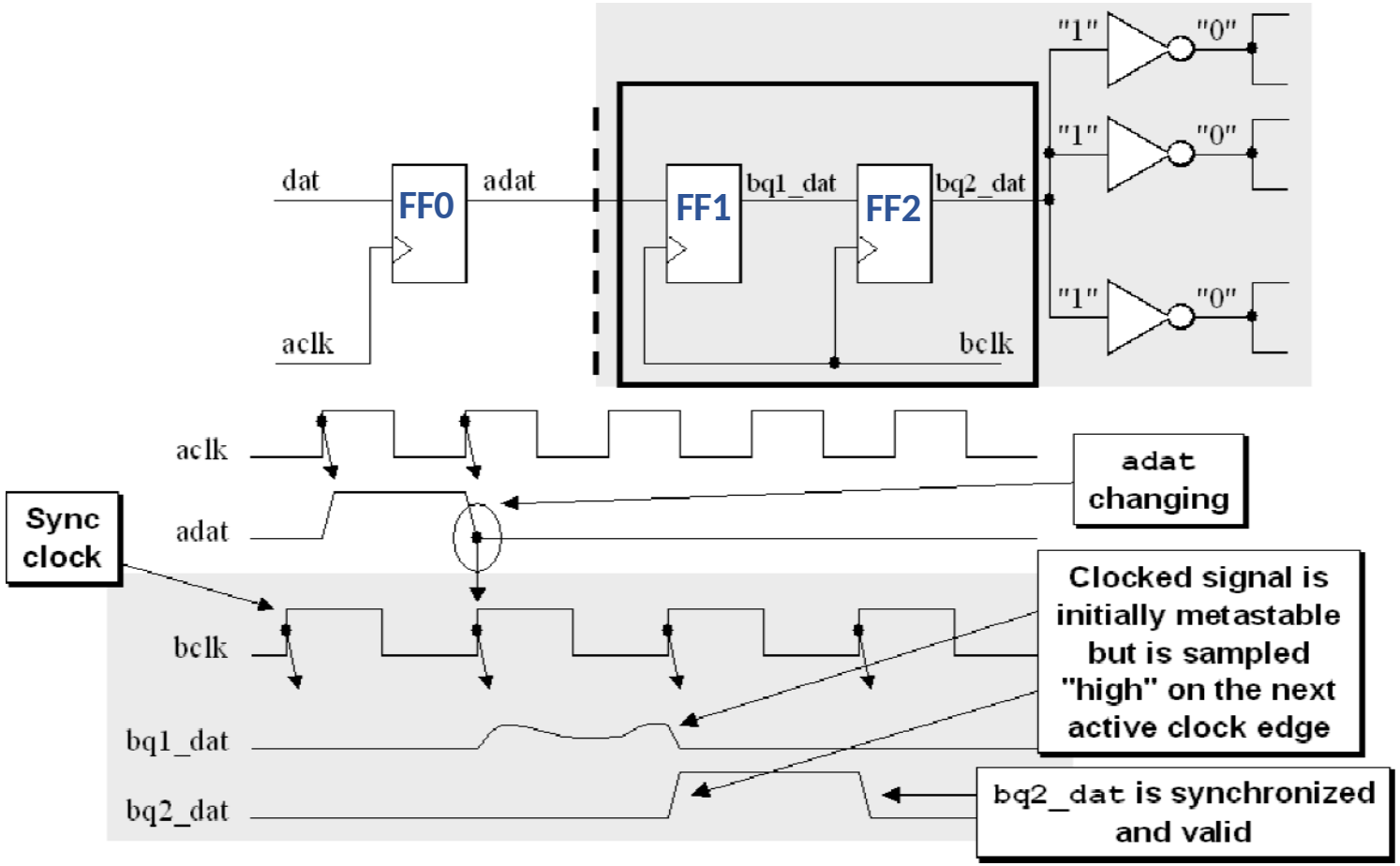
Synchronization Techniques

- **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)

2-FlipFlop Synchronizer to Address Metastability



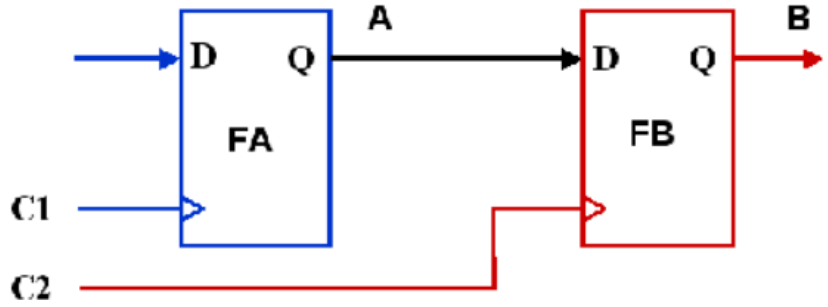
Note : Multi-Flop synchronizers allow sufficient time for the oscillations to settle down and ensure that a stable output is obtained in the destination domain

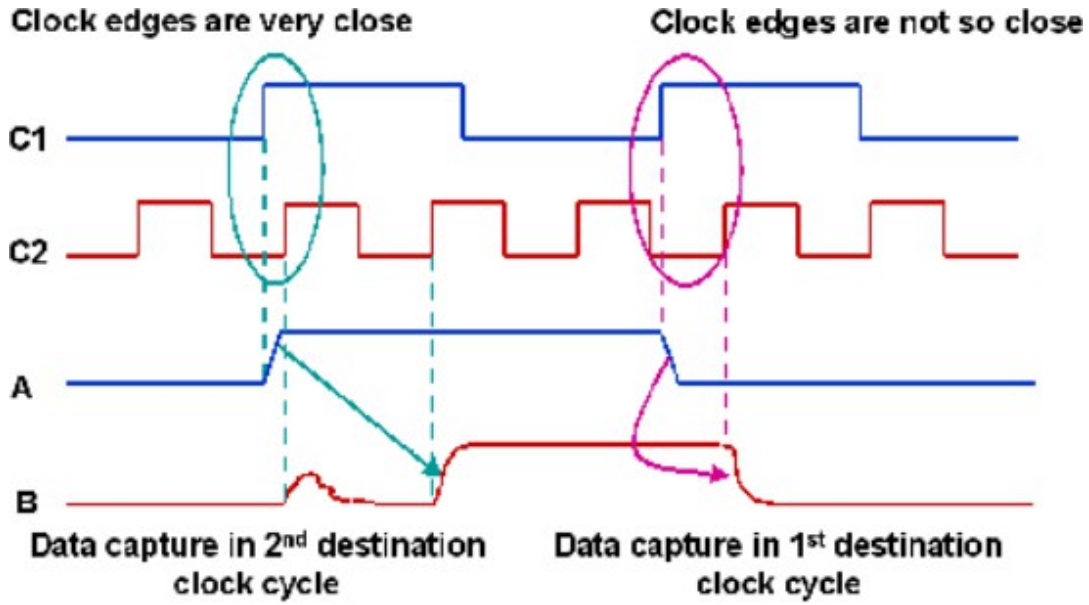
- First flipflop (**FF1**) samples the asynchronous input signal into the new clock domain (**bclk**)
- Waits for full clock cycle to permit any metastability on the synchronizer stage-1 output signal (**bq1_dat**) to decay, then the stage-1 signal is sampled by the same clock(**bclk**) into a second stage flipflop (**FF2**)
- The stage-2 signal (**bq2_dat**) is now a stable and valid signal synchronized and ready for distribution within the new clock domain

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

Signal Crossing Slow Clock Domain to Fast Clock Domain

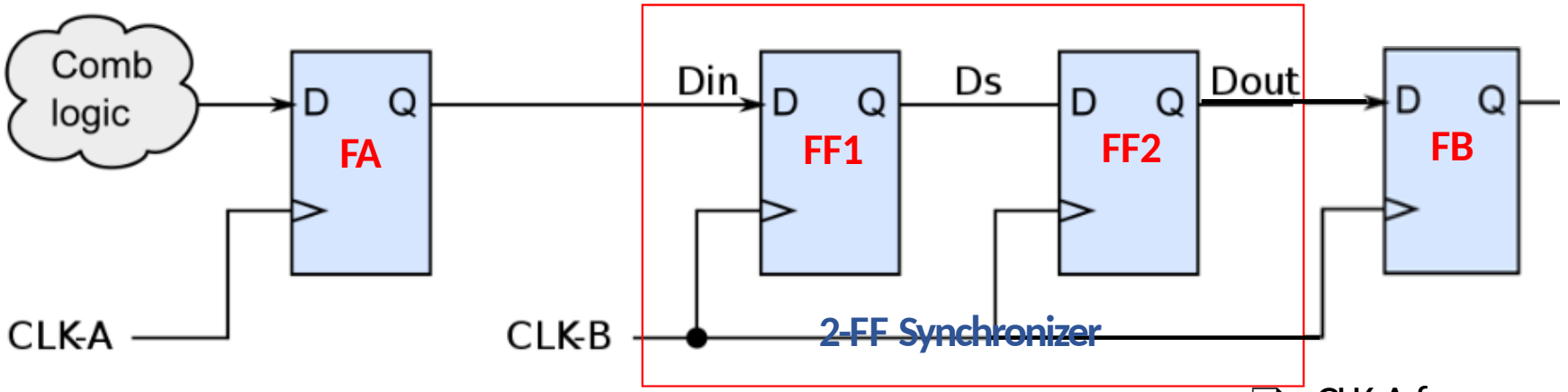
- **Scenario -1 : Flipflop-A clock frequency < Flipflop-B clock frequency**

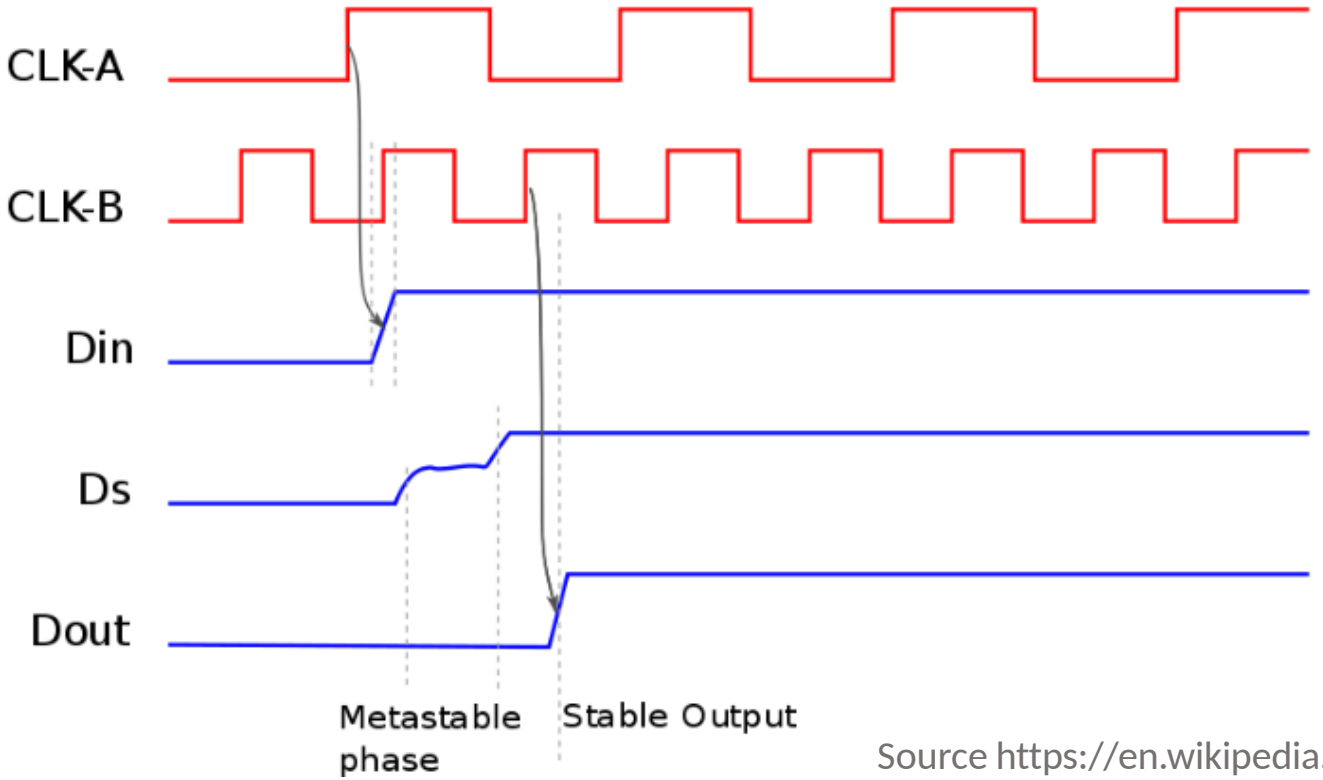




- If the transition on signal A happens very close to the active edge of clock C2, it could lead to setup or hold violation at the destination flop "FB“ and “FB” will enter metastable state.
- Output signal B from FA will be unstable and may or may not settle down to some stable value before the next 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



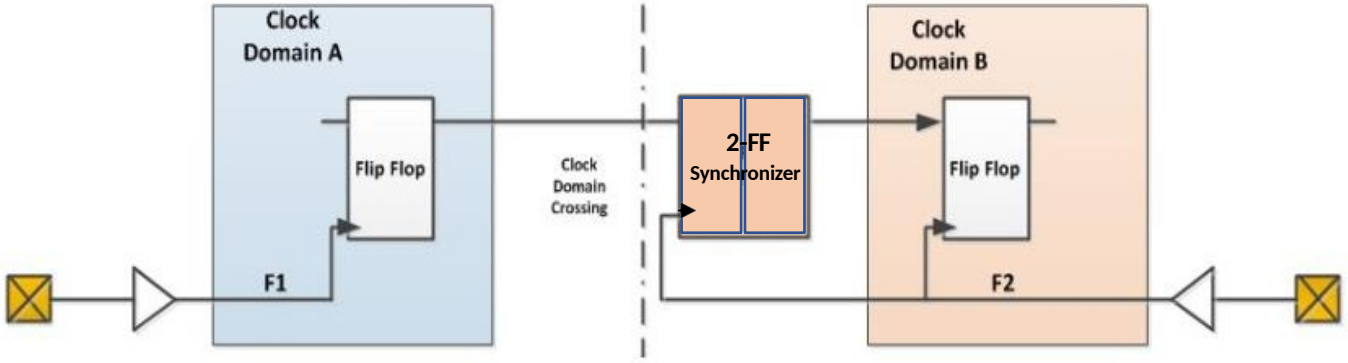


- CLK-A frequency is slower than CLK-B
- Din entering into CLK-B domain goes through 2-FF synchronizer stage (FF1, FF2)
- 2-FF synchronizer will operate on receiving clock which is CLK-B
- Even if Din changes close to CLK-B clock edge violating setup violation of FF1, causing metastable output Ds, output 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)

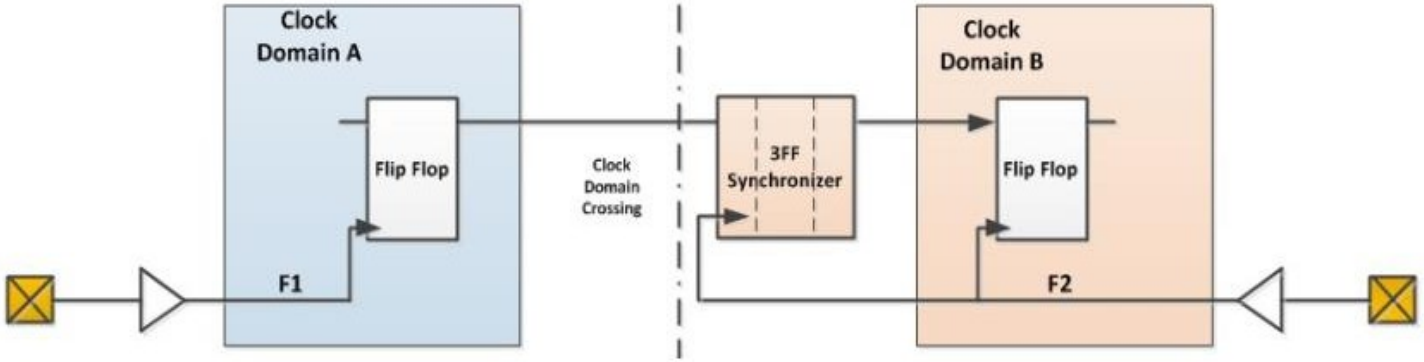
Source https://en.wikipedia.org/wiki/Metastability_(electronics)

2-FlipFlop and 3-FlipFlop Synchronizer

2-FF Synchronizer



3-FF Synchronizer



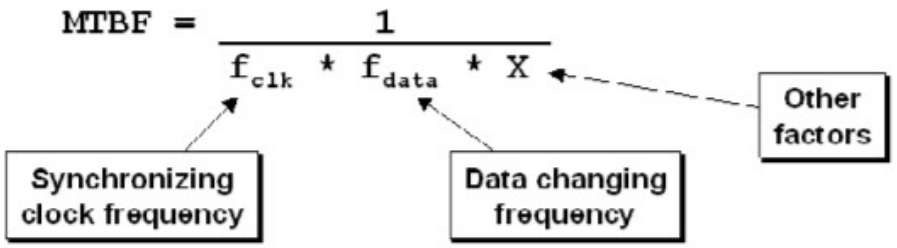
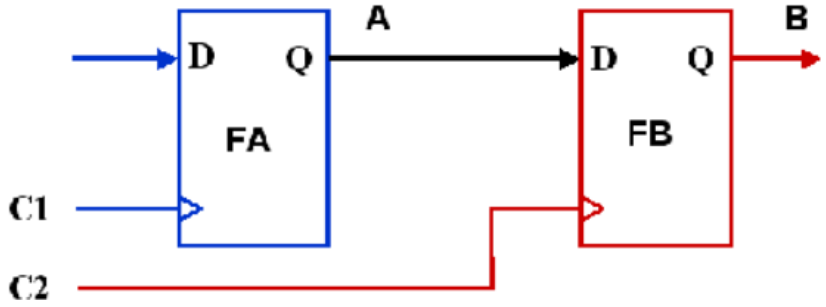


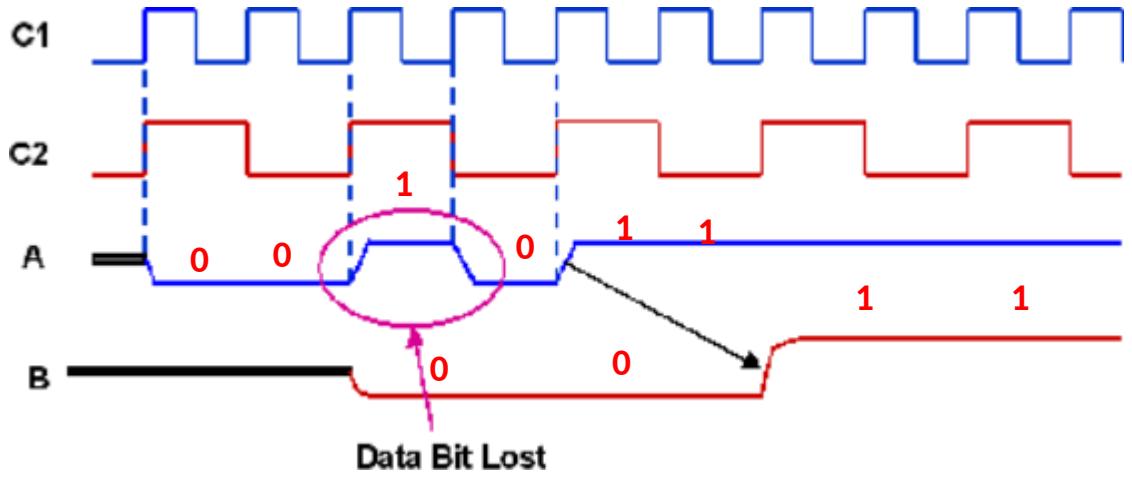
Figure 4 - Primary contributing factors to short MTBF values

- **Synchronizers must be designed to reduce the chances of system failure due to metastability**
- **Synchronizer requirements**
  - Reliable [high **MTBF** : Mean Time Between Failure]
  - Low latency [works as quickly as possible]
  - Low power/area impact
- **For some designs 3-FF synchronizer might be required if source frequency is high and input data is changing fast**
- **Can increase MTBF by adding more series stages**
  - 3-FF synchronizer can have higher MTBF

Signal Crossing Fast Clock Domain to Slow Clock Domain

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

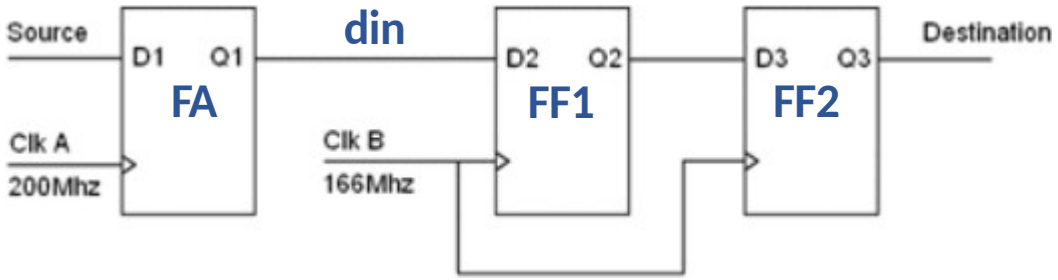


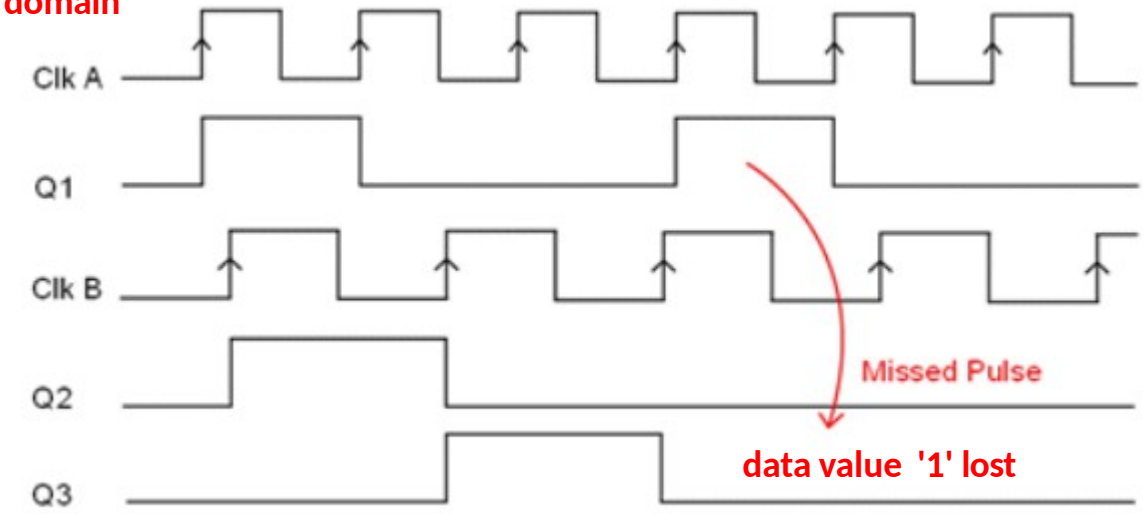


- C1 is two times faster than C2 and there is no phase difference between C1 and C2
- If signal A is changing rapidly, say for example signal A sequence is "**001011**", output from FB in C2 clock domain will be "**0011**".
  - Here the third data value in the input sequence which is **"1"** is lost → **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

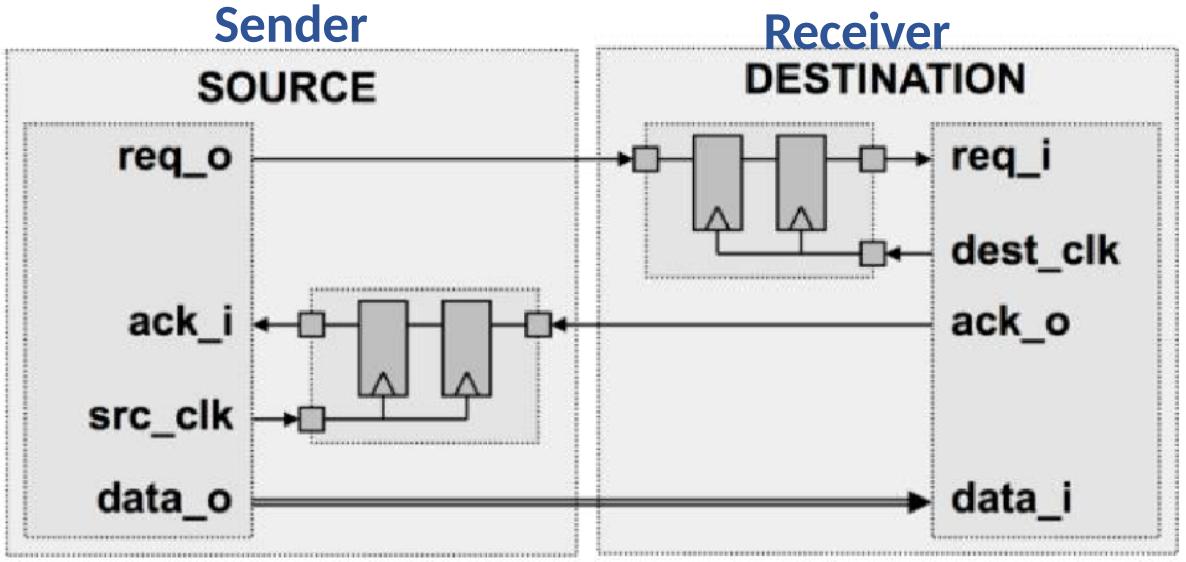
Example of data Loss when data flowing from fast to slow to fast clock domain

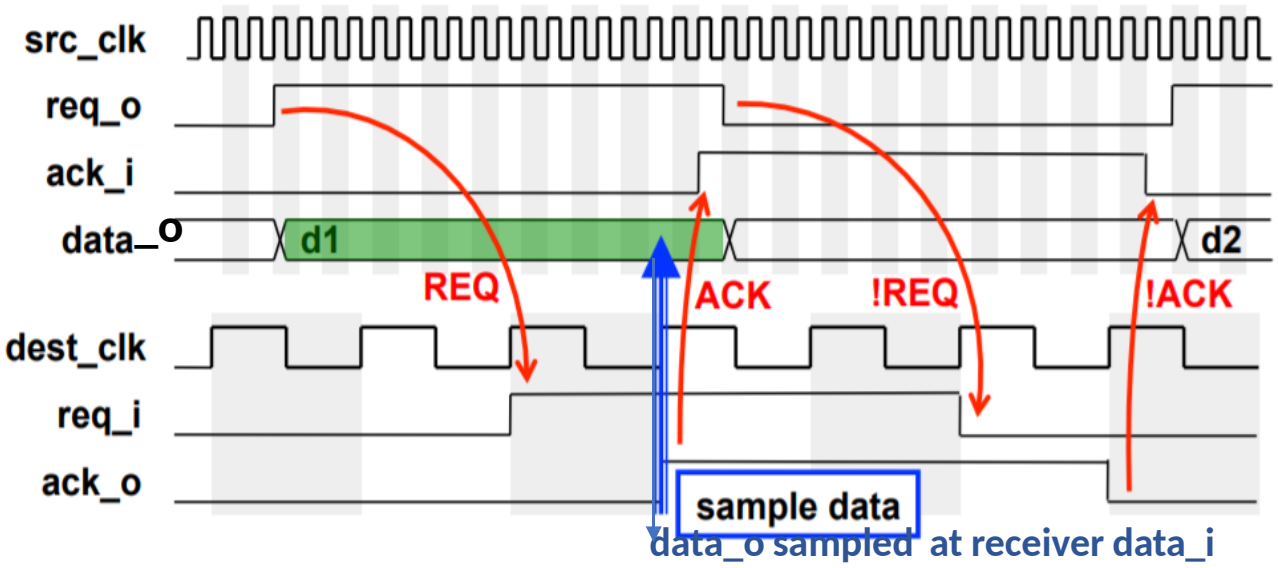




- Consider, signal crossing from source clock domain ClkA to Destination ClkB domain where ClkA is faster than ClkB
- ClkA is 200 Mhz and Clk B is 166 MHz
- Signal is Crossing Fast to Slow clock domain through 2-FF synchronizers (FF1 and FF2)
- Adding 2-FF or 3-FF synchronizer will still result in data 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 !**

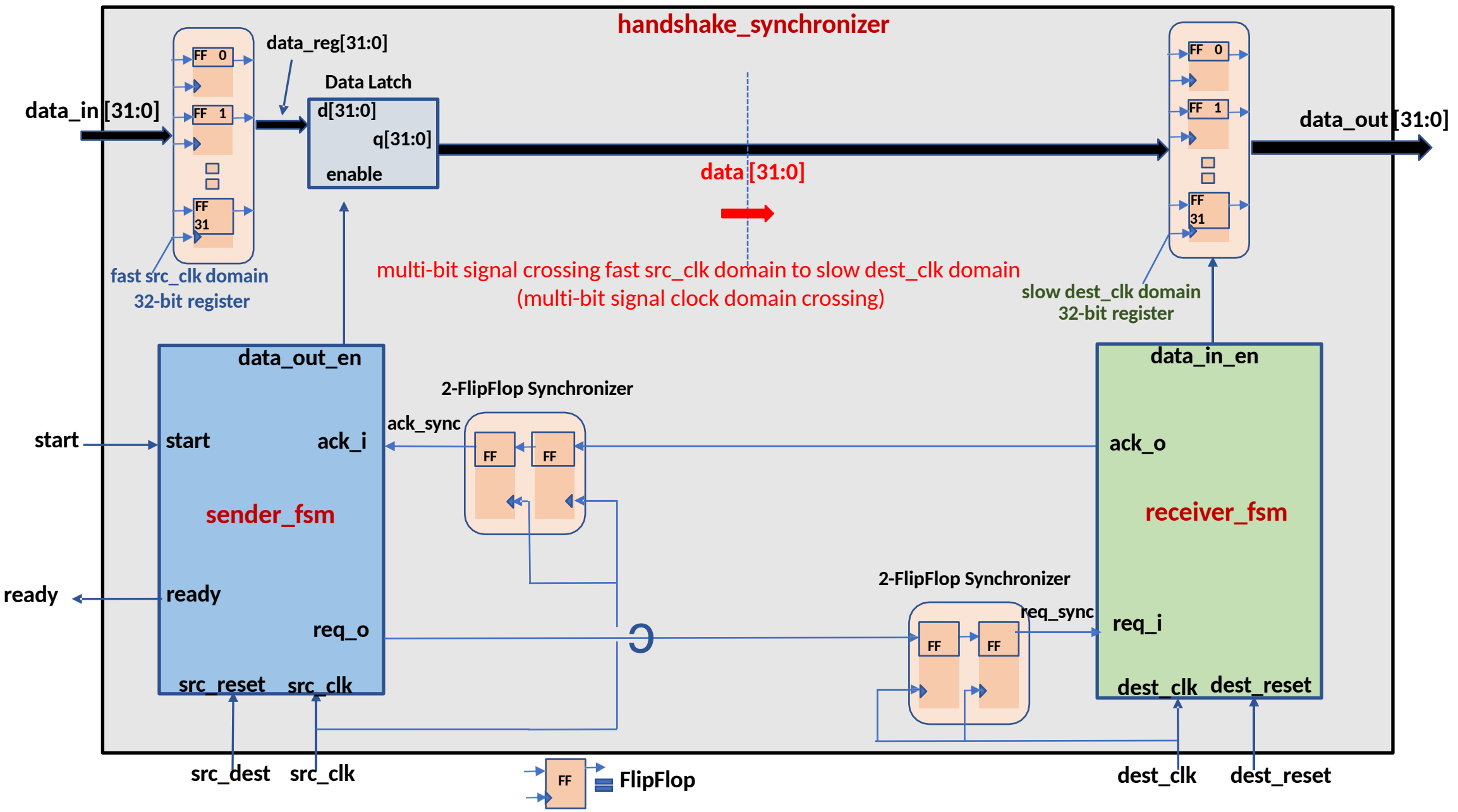
Handshake Synchronizer For Multi-bit Signal Crossing Fast to Slow Clock Domain





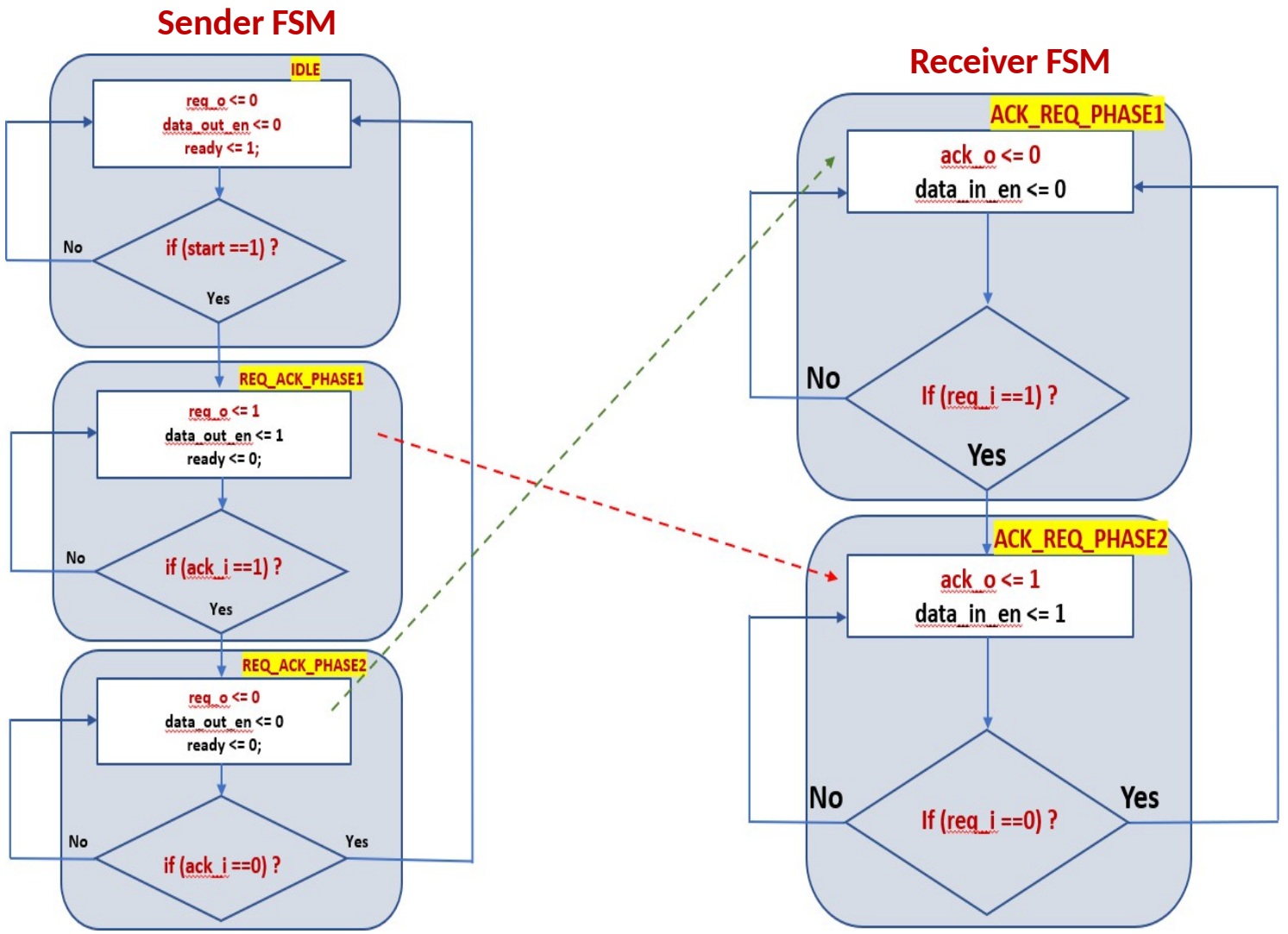
- For many open-ended data-passing applications, a simple **req-ack** handshaking sequence is sufficient
- The sender places data onto a data bus (**data_o**) and then synchronizes a "**req_o**" signal (request) to the receiving clock domain.
- When the "**req_o**" signal is recognized in the destination 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**)

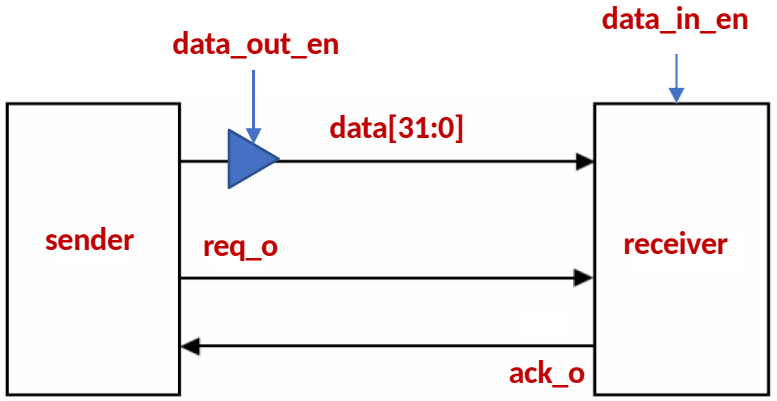
Handshake Synchronizer



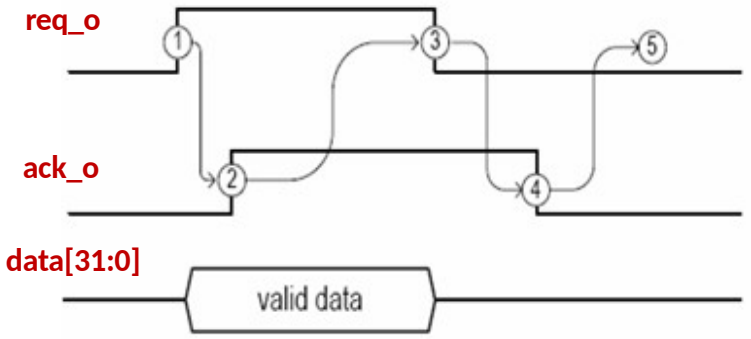
Handshake Synchronizer

- 4-phase req-ack handshake protocol between sender_fsm and receiver_fsm

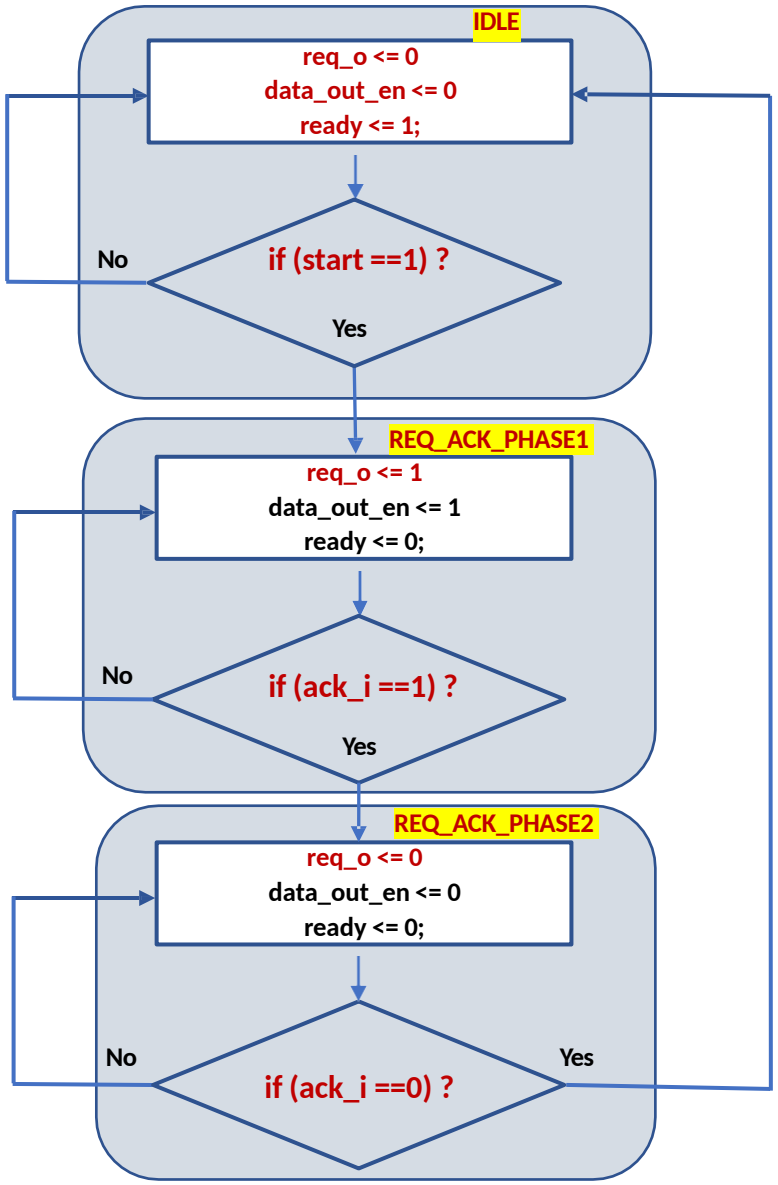




4-phase req-ack handshake protocol



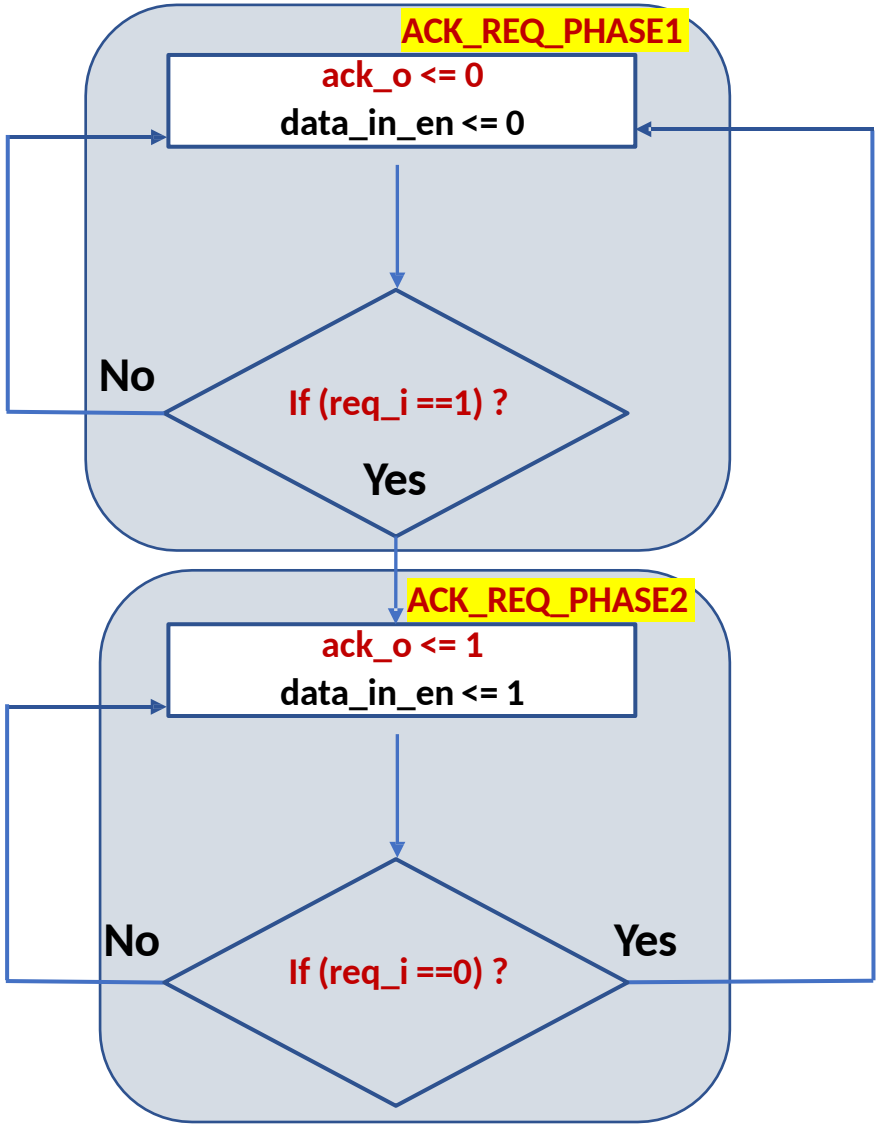
Handshake Sender FSM



❑ 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

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

Handshake Synchronizer

❑ Reference simulation waveform for handshake_synchronizer module

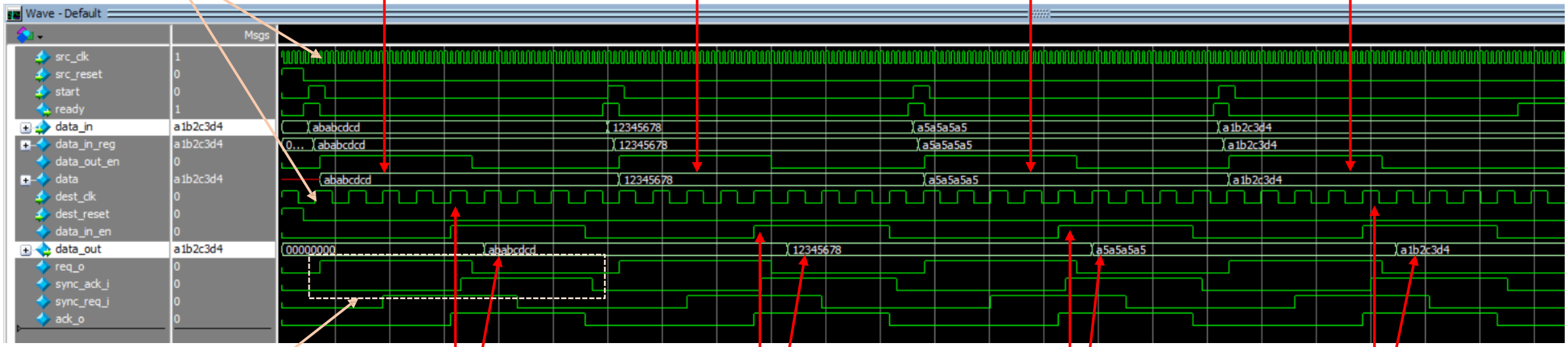
src_clk = 20 ns clock period
dest_clk = 124 ns clock period
dest_clk is ~6 times slower than src_clk

First 32-bit data 32'hababcdcd sent from fast src_clk domain by sender_fsm to receiver with data_out_en = 1

Second 32-bit data 32'h12345678 sent from fast src_clk domain by sender_fsm to receiver with data_out_en = 1

Third 32-bit data 32'ha5a5a5a5 sent from fast src_clk domain by sender_fsm to receiver with data_out_en = 1

Fourth 32-bit data 32'ha1b2c3d4 sent from fast src_clk domain by sender_fsm to receiver with data_out_en = 1



4-phase req-ack handshake between sender_fsm and receiver_fsm

First 32-bit data 32'ha5ababcdcd captured by in slow dest_clk domain by receiver_fsm when data_ie_en = 1 and available on data_out 1 cycle later

Second 32-bit data 32'h12345678 captured by in slow dest_clk domain by receiver_fsm when data_ie_en = 1 and available on data_out 1 cycle later

Third 32-bit data 32'ha5a5a5a5 captured by in slow dest_clk domain by receiver_fsm when data_ie_en = 1 and available on data_out 1 cycle later

Fourth 32-bit data 32'ha5a1b2c3d4 captured by in slow dest_clk domain by receiver_fsm when data_ie_en = 1 and available on data_out 1 cycle later

Handshake Synchronizer

Requirements for handshake synchronizer

- Design handshake synchronizer to send data form fast clock domain to slower clock domain without dropping any data
- Create **sender_fsm** and **receiver_fsm** modules
- Review flipflop_synchronizer code provided in lab folder
- Create **handshake_synchronizer** module and instantiate **sender_fsm, receiver_fsm, flipfliop synchronizer** and make connections
  - Refer to handshake_synchronizer block diagram on next slide when making connections between the modules
  - Add latch after the source data register to hold data until data is captured by the destination register
  - Size of data to be sent and captured = 32 bits
- sender_fsm and receiver_fsm should implement 4-phase req-ack handshake protocol (see slide on this requirement)
- Review resource utilization including total flipflops and combination ALUT's in report
- Simulate **handshake_synchronizer** with testbench provided
  - Testbench generates src_clk with time period of 20ns and dest_clk with time period of 124 ns (i.e. src_clk is around 6 times slower than dest_clk).
  - Note : Higher the time period lower the frequency (f = 1 / T). Since src_clk time period is lower its frequency is higher
- Review simulation waveform and explain results of **handshake_synchronizer**
  - There is no self-checker implemented in testbench so review input data_in and data_out values of handshake_synchronizer in simulation waveform
  - Ensure each data value sent by sender_fsm are successfully captured by receiver_fsm
- Primary port list for **sender_fsm, receiver_fsm, handshake_synchronizer** modules :
  - **handshake_synchronizer** : **input** src_clk, src_reset, dest_clk, dest_reset, start, **input** [31:0] data_in, **output** ready **output**[31:0] data_out
  - **sender_fsm** : **input** src_clk, src_reset, start, ack_i, **output** data_out_en, ready, req_o
  - **receiver_fsm** : **input** dest_clk, dest_reset, req_i, **output** data_in_en, ack_o

Handshake Folder

- **handshake_synchronizer folder includes :**
  - Handshake_synchronizer folder which includes
    - Design template code for handshake_synchronizer, sender_fsm, receiver_fsm, flipflop_synchronizer modules
    - handshake_synchronizer full testbench code
  - For learning purpose, student can change the stimulus in initial block in testbench files

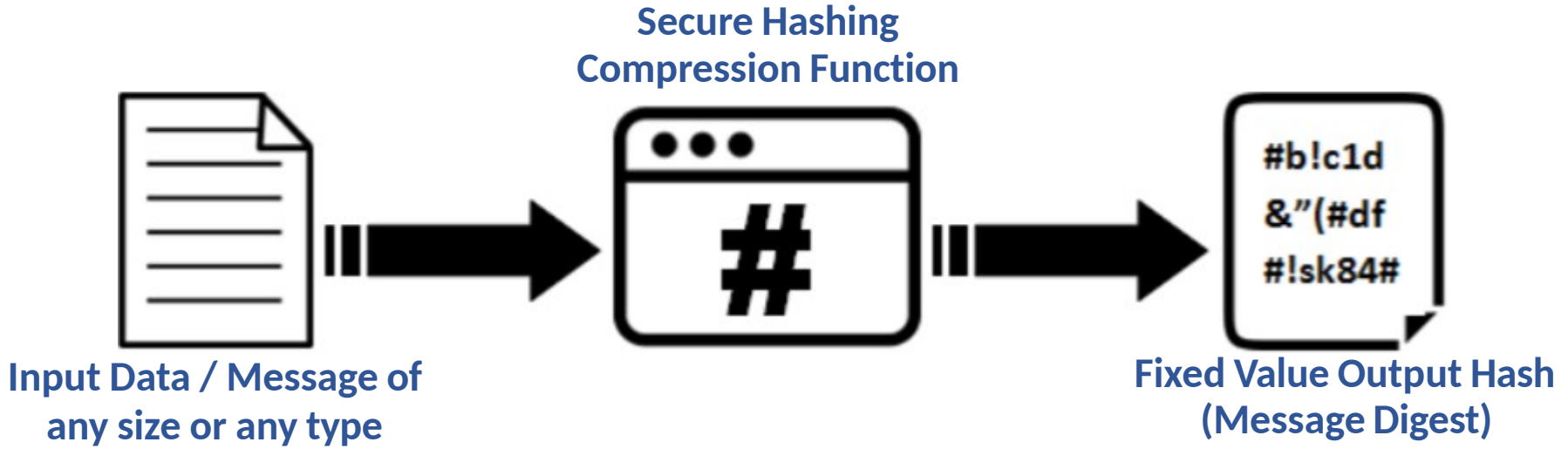
References

- **Clock Domain Crossing (CDC) Design & Verification Techniques Using SystemVerilog Clifford E. Cummings Sunburst Design, Inc**
  - http://www.sunburst-design.com/papers/CummingsSNUG2008Boston_CDC.pdf

- **About Clock Domain Crossing**
  - https://en.wikipedia.org/wiki/Clock_domain_crossing

SubPart2: 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 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

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

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

SHA-256

78e7771b8b46e11ddb34ba48887e1330525215f96d94778980d1186e6f09f6b4

Fixed 256-bit Value Digest (Output Hash Value)

The quick brown fox jumps over the lazy dog

88-bit Hexadecimal Input Message

SHA-256

c03905fcdab297513a620ec81ed46ca44ddb62d41cbbd83eb4a5a3592be26a69

Fixed 256-bit Value Digest (Output Hash Value)

32KB AVATAR.JPG Image File

SHA-256

017fa150d2b079a9fee41169cb09142f2b91badbbbf311797e90982b8dc1bf14

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 !

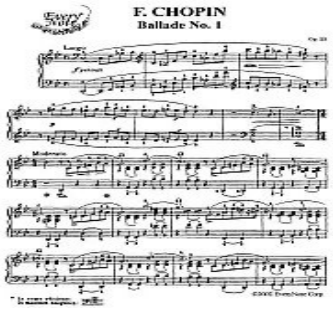
a

2-bit Hexadecimal Value

SHA-256

a7da489976d0047490617adb4f7a1f27f7af8b52a5176fd002ffe471863520ab

Fixed 256-bit Value Digest (Output Hash Value)



5 MB MP3 File

SHA-256

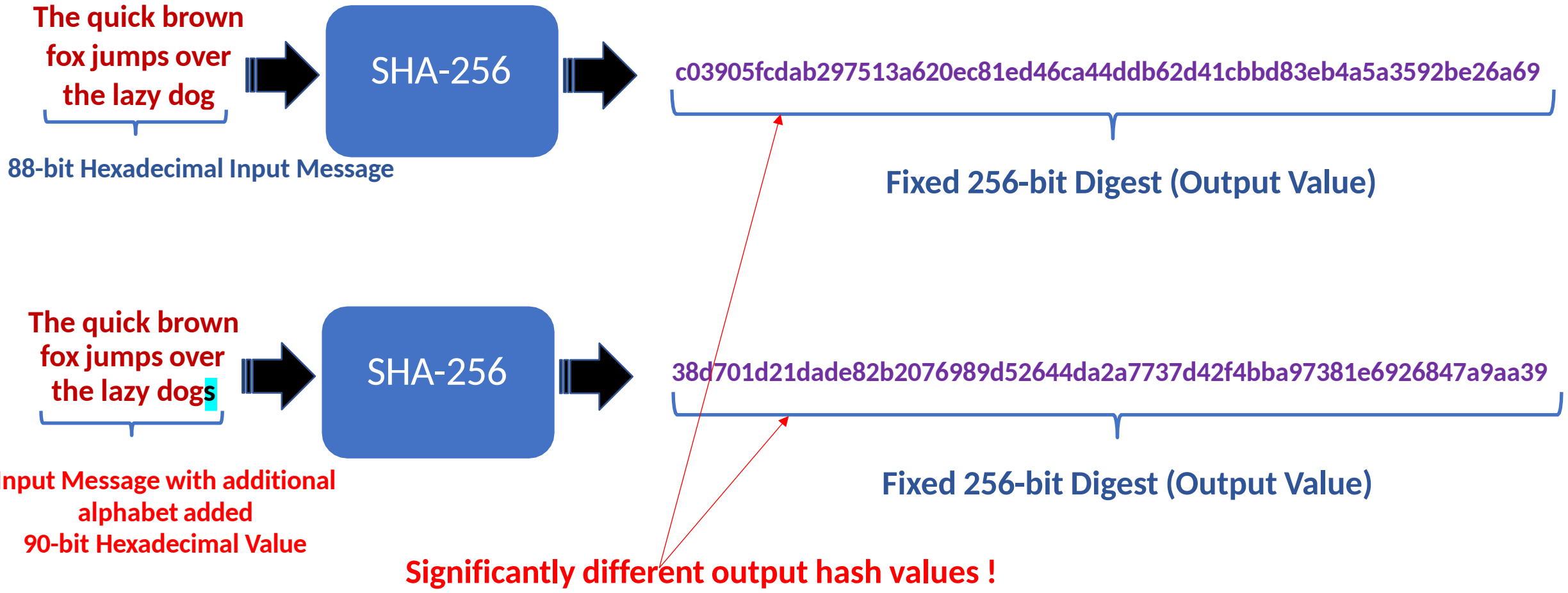
d9904266f642d4659769bba49eb8552e2d53ef09b1faabd8f72154530ba16fc8

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



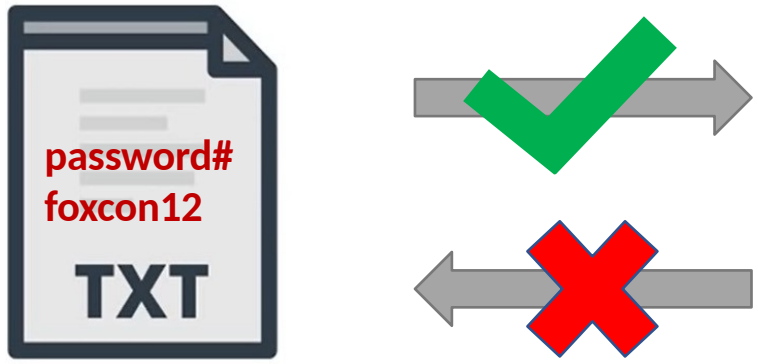
Determinism

- **Same input must always generate the same output by different systems**
  - Any machine in the world which understands hashing algorithm should able 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 !

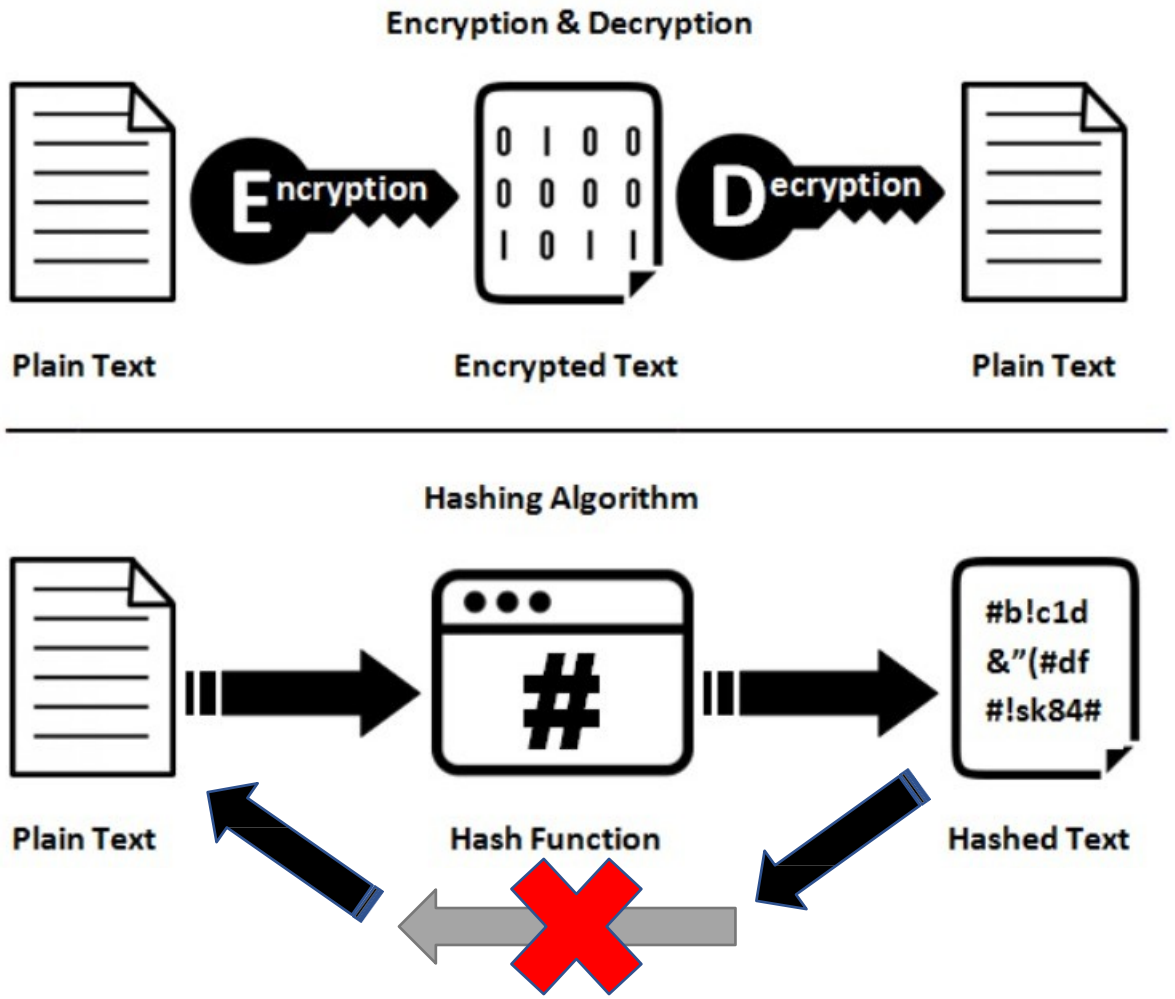


db2c26da2750dea1add7d7677c22d6dc
b6dc4e2674357c82c39bb96d563f0578

- **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

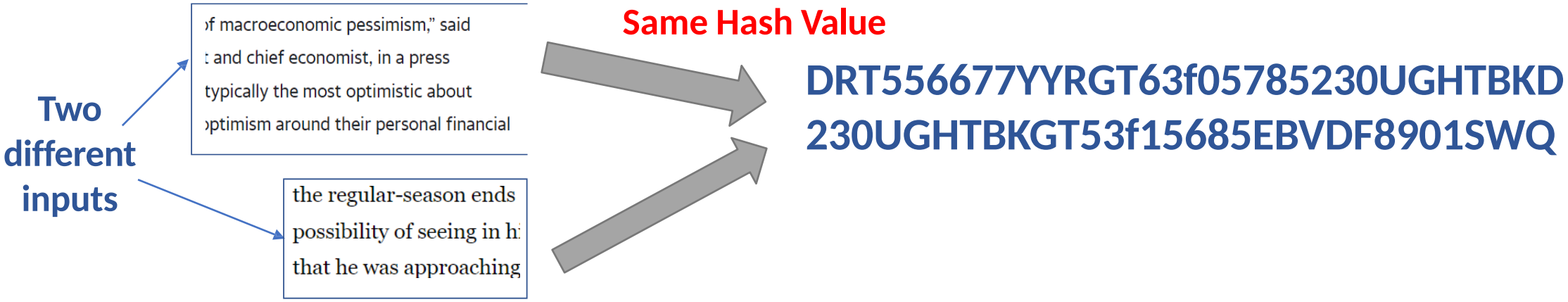
❑ Hashing Function suffers from the same birthday problem

▪ 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**

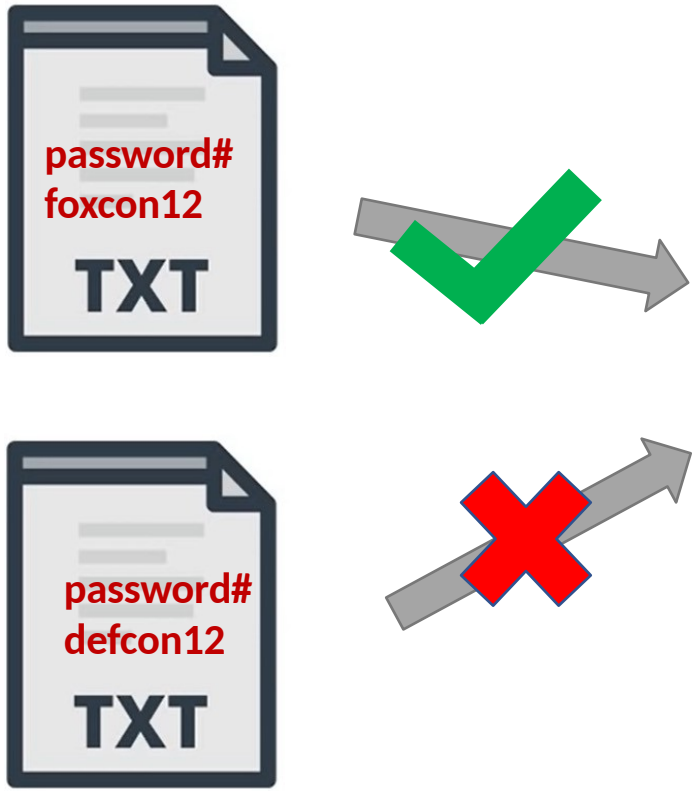
▪ In rare occasions hashing may produce hash collision ! – Similar to Birthday Hash Problem

- 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

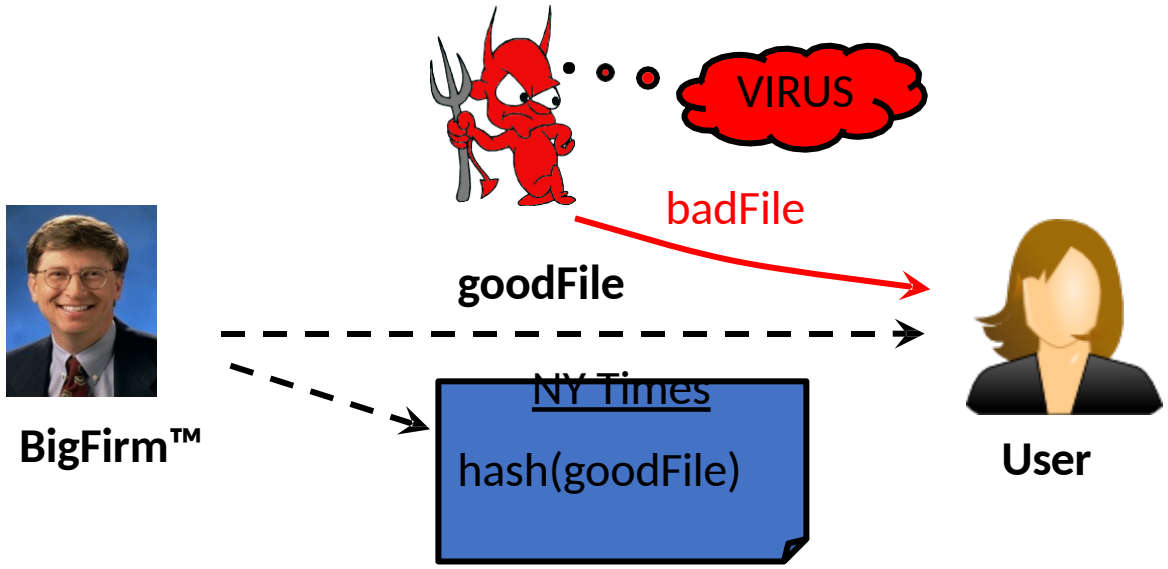
- **Hashing algorithm should be rigorous and it must withstand collision !**
  - Hackers may take advantage of hash collision
  - To avoid hash collision, the output length of the hash value can be large enough so that birthday attack becomes computationally infeasible
    - **Example :** Document hashed 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

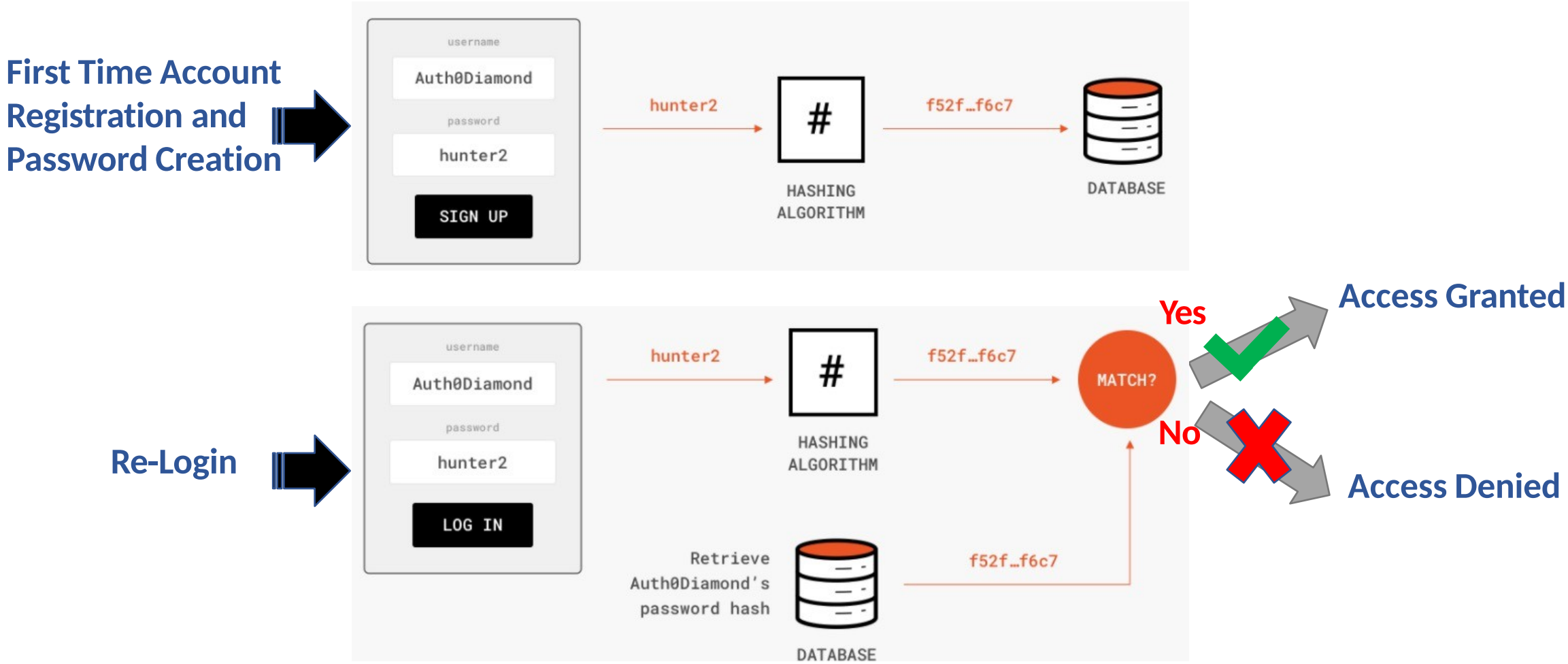
Two different input blocks with same output hash value should be practically impossible even though theoretically possible !

Applications of SHA256 : Verifying File Integrity



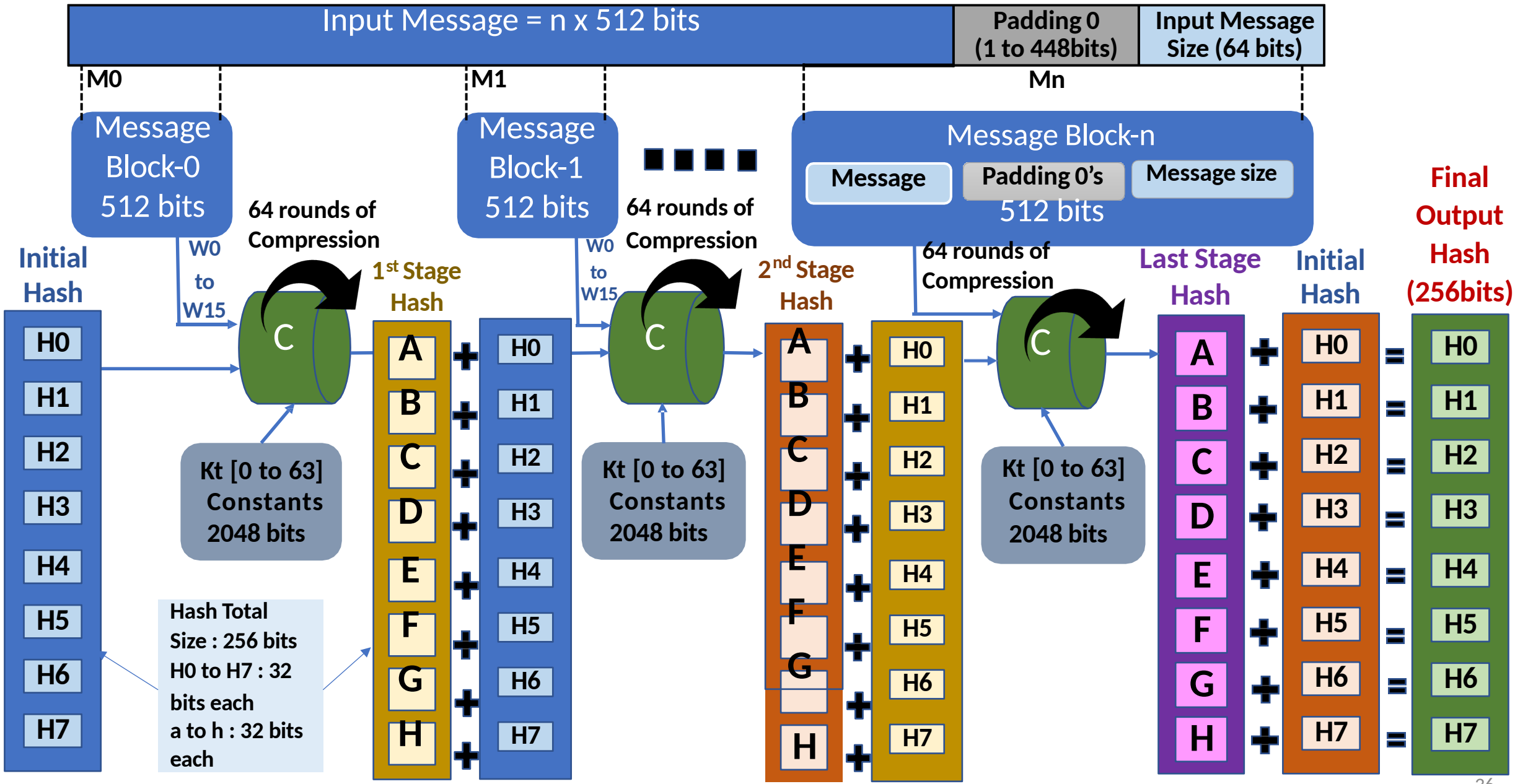
- Software manufacturer wants to ensure that the executable file is received by users without modification
- Sends out the file to users and publishes its hash in NY Times
- The goal is integrity, not secrecy
- **Idea: given goodFile and hash(goodFile), very hard to find badFile such that hash(goodFile)=hash(badFile)**

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

SHA256 Algorithm



SHA256 Algorithm

Compression Function includes two steps :

Work Expansion followed by SHA256 operation

C

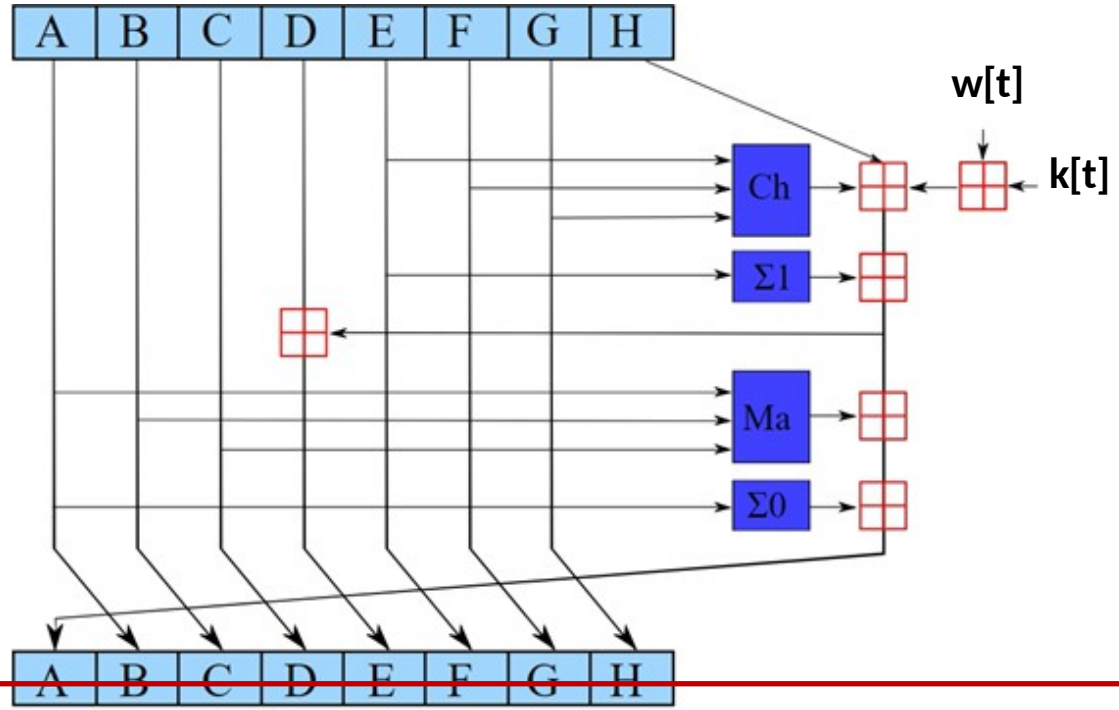
Step 1: Word Expansion

```
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



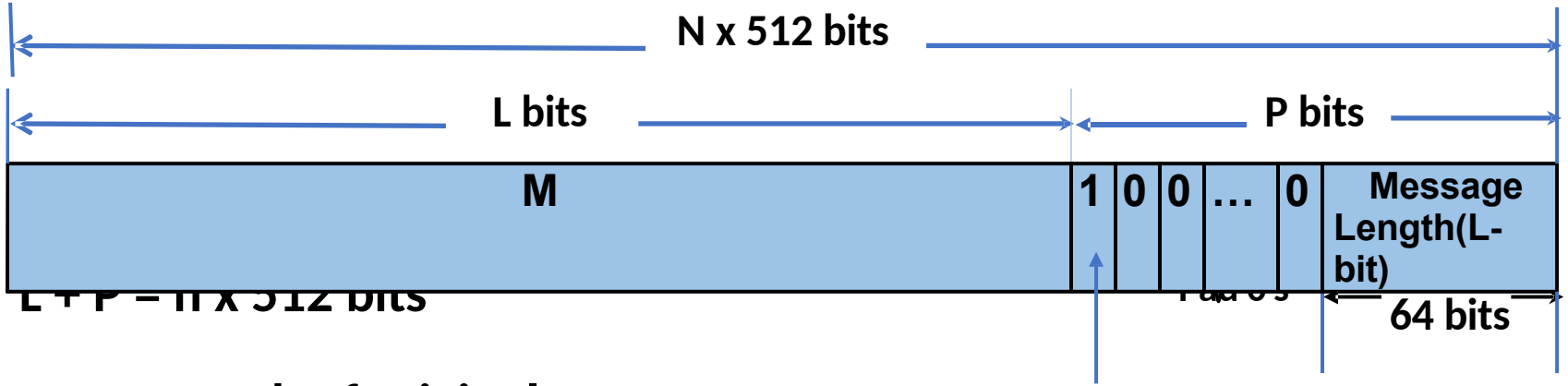
SHA256 Algorithm

- **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**-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).

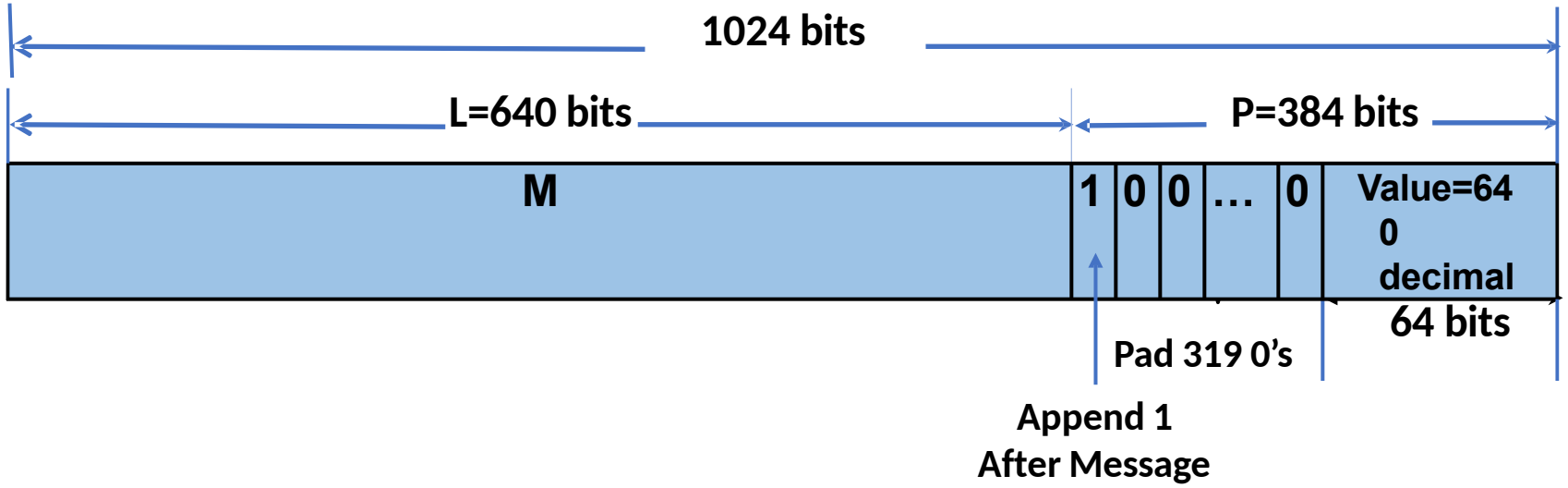
- **Step 2** : Append message length bits in 0 to 63 bit position
  - Since SHA256 supports until 2^64 input message size, 64 bits are required to append message length



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

SHA256 Algorithm

- **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

- **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)

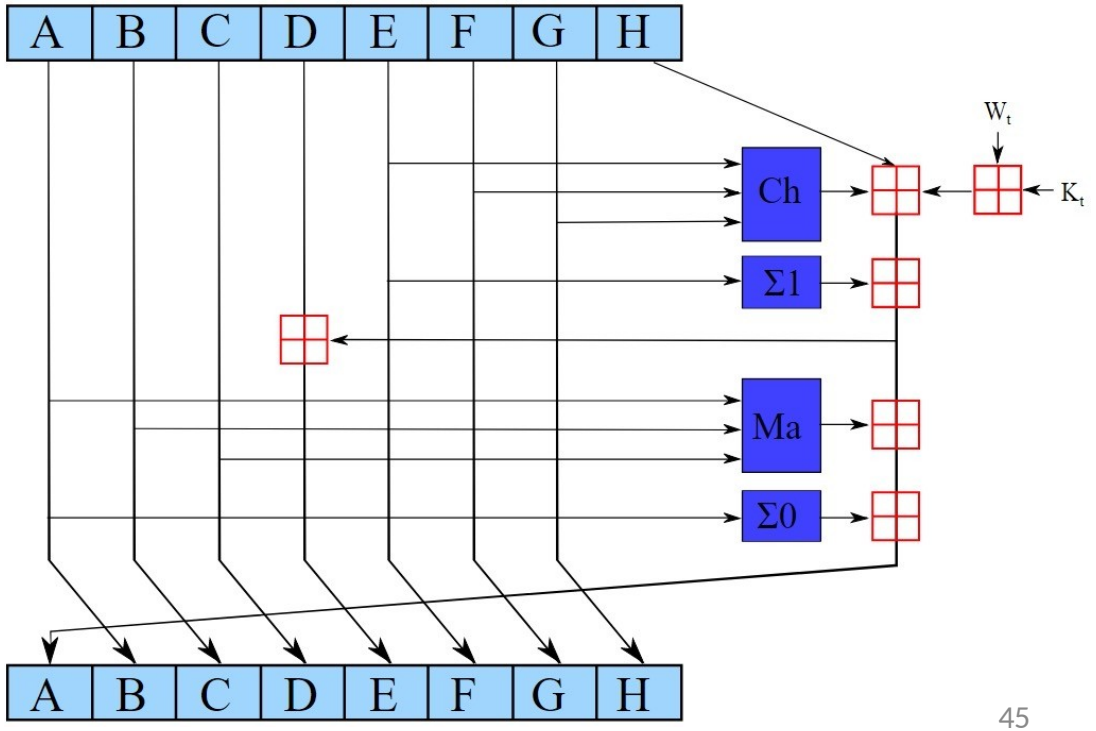
$t_2 = S_0 + maj$

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

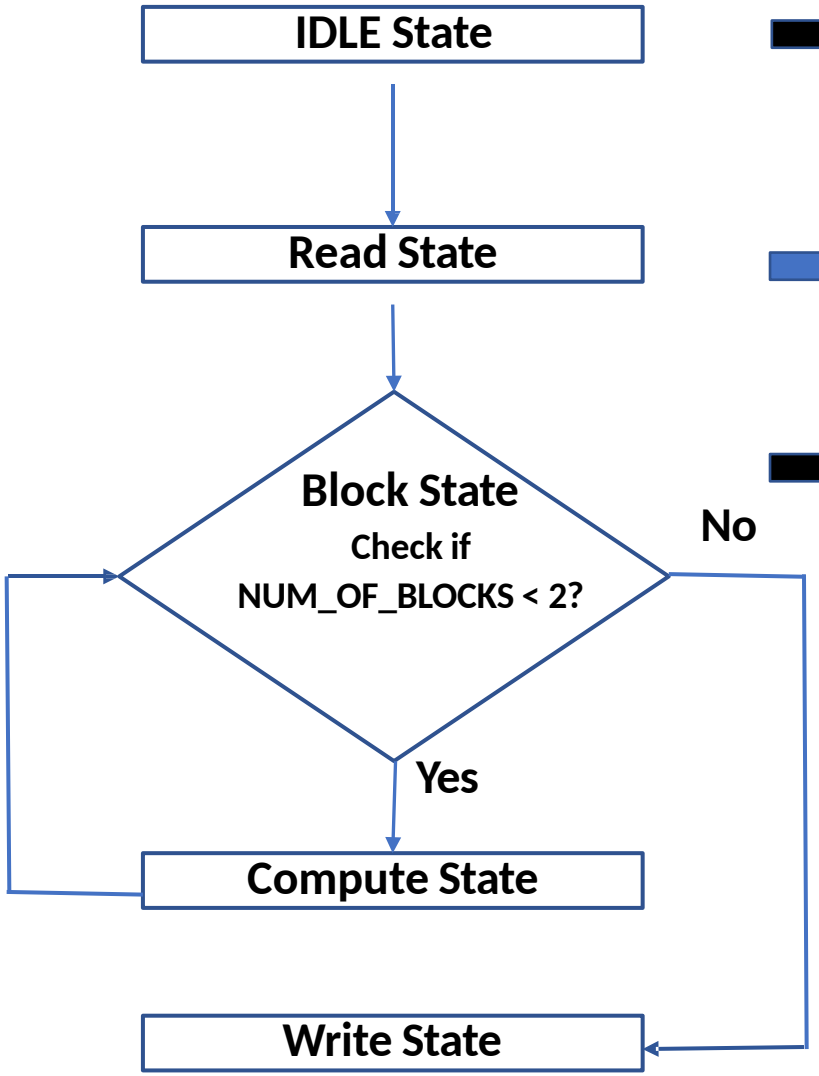
    H5 = H5 + f

    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

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

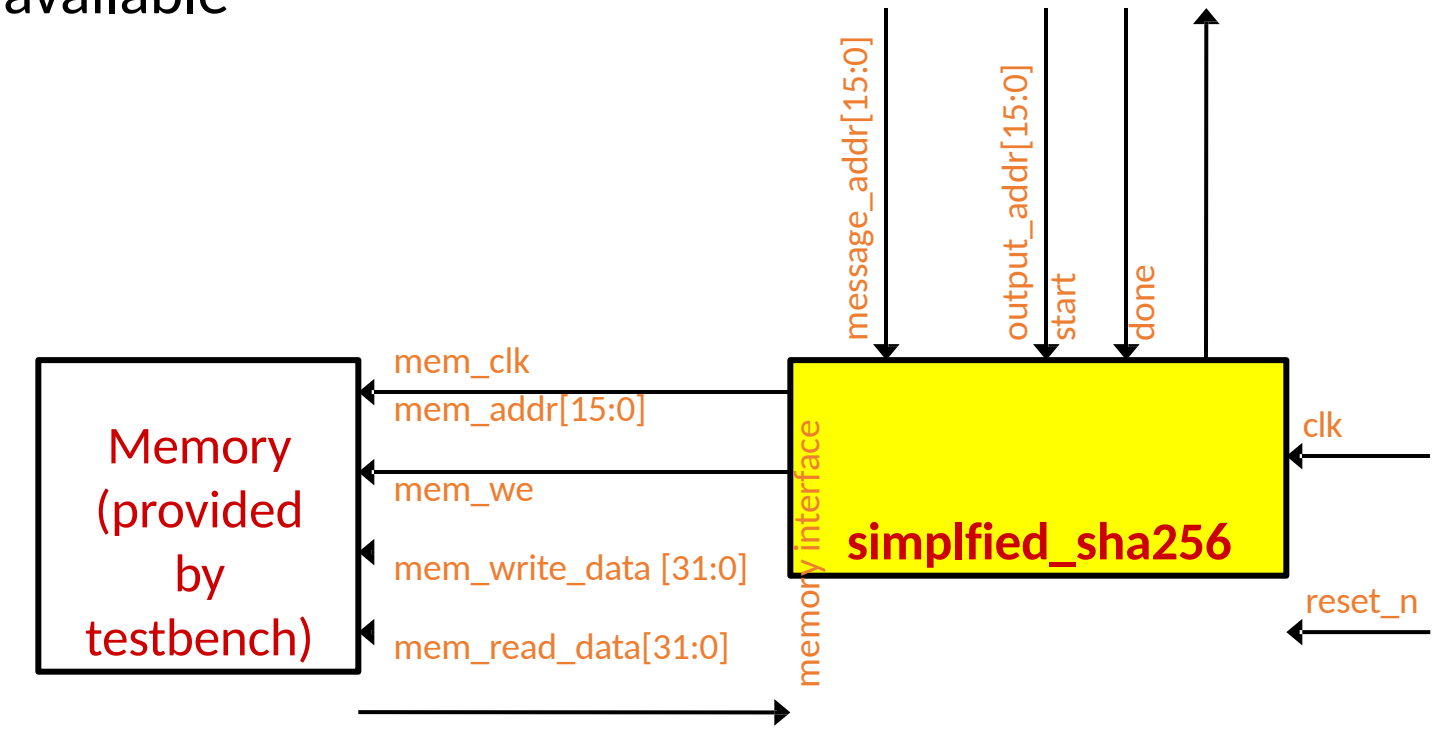
- In Block FSM state, Create two message blocks each of 512 bits
- First message block has first 16 words of input message stored in 'w' array
- Second message block has 4 words of input message, value 1, padding 0 and message size 640
- Assign h0, h1, h2, h3, h4, h5, h6, h7 to a, b, c, d, e, f, g, h 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

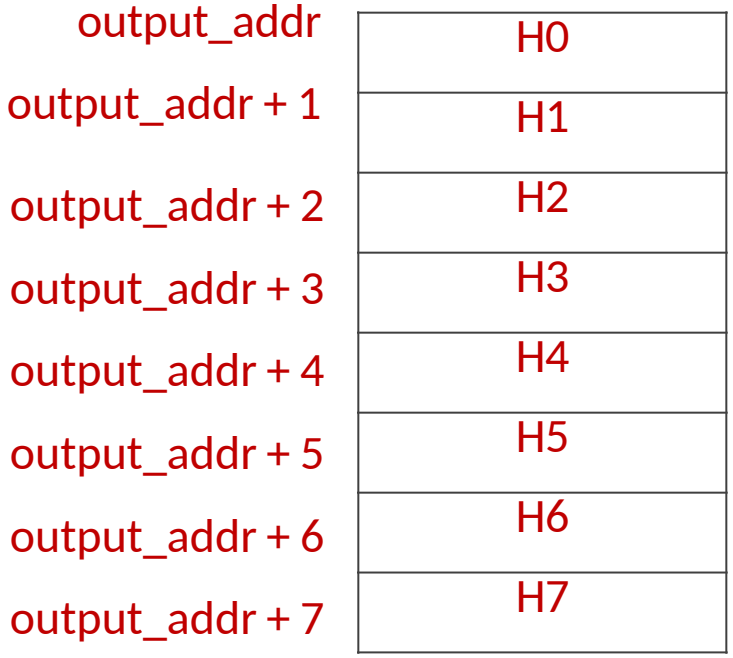
- Wait in idle state for **start**, read message starting at **message_addr** and write final hash **{H0, H1, H2, H3, H4, H5, H6, H7}** in 8 words to memory starting at **output_addr**. **message_addr** and **output_addr** are word addresses.
- Message size is "hardcoded" to 20 words (640 bits).
- Set **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:

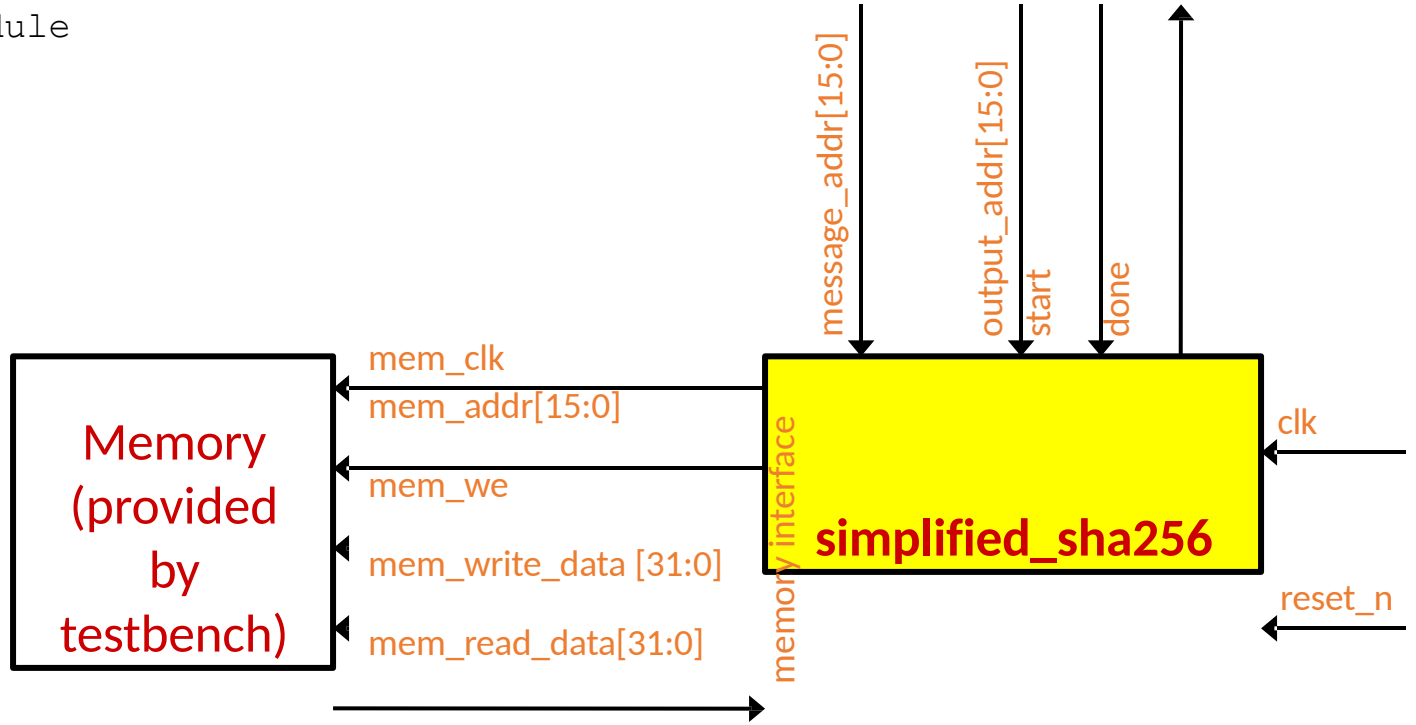
```
mem_addr <=
output_addr;
mem_write_data <= H0;
mem_addr <=                + 1;
output_addr
mem_write_data <= H1;
...

mem_addr <=                + 7;
output_addr
mem_write_data <= 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 Megafunctions or Latches

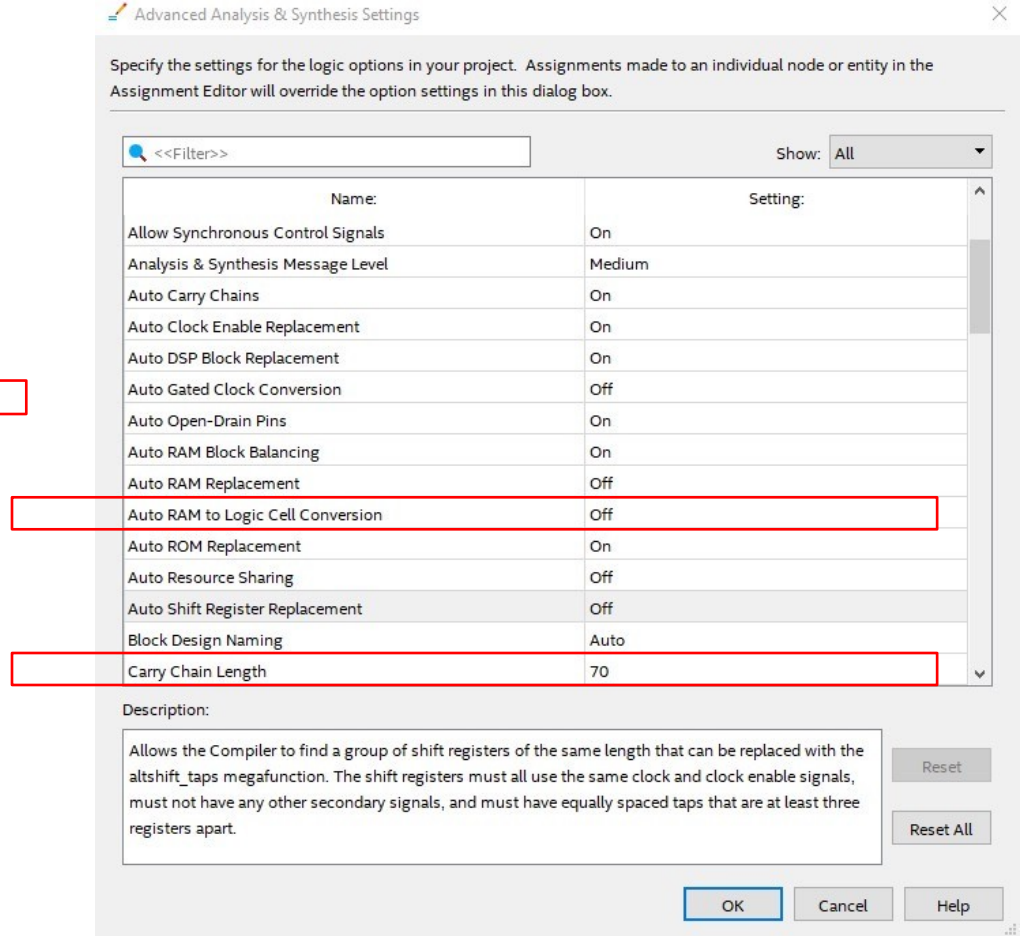
- **In your Quartus compilation message ensure :**
  - **No inferred megafunctions**: Most likely caused by block memories or shift-register replacement. Can turn OFF “Automatic RAM Replacement” and “Automatic Shift Register Replacement” in “Advanced Settings (Synthesis)”. If you still see “inferred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

No Block Memory Bits

- In your bitcoin_hash.fit it **must** say **Total block memory bits is 0** (otherwise will not pass).

```
+-----------------------------------------------------------------------------+
; 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 % )                         ;
```

- If not, go to "Assignments→Settings" in Quartus, go to "Compiler Settings", click "Advanced Settings (Synthesis)"
- Turn OFF "Auto RAM Replacement" and "Auto Shift Register Replacement"



Final Project 1 SubPart2 Submission

- **Put following files into (LastName, FirstName)_(LastName, FirstName)_finalproject1.zip**
  - Both design files and also testbench code for both parts - Handshake (handshake_synchronizer, sender_fsm, receiver_fsm, flipflop_synchronizer modules) and SHA256
  - Modelsim transcript files msim_transcript for both parts
  - For SHA256 provide fitter and sta files (files with extension .fit, .sta)
  - Final report.

- **Final report should including following mentioned :**
  - **Handshake Synchronizer:**
  - Explain briefly what handshake_synchronizer is (may use lecture slide contents)
  - Synthesis resource usage snapshot generated from Quartus for handshake_synchronizer top level module
  - Simulation snapshot and explain simulation result of handshake_synchronizer
  - Explanation of FPGA resource usage in report is not required.
  - **SHA256:**
  - Explain briefly what SHA-256 is (may use lecture slide contents)
  - Describe algorithm for both SHA-256 implemented in your code
  - Simulation waveform snapshot for both SHA-256
  - Provide modelsim transcript window output indicating passing test results generated from self-checker in testbench for SHA-256
  - 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.
  - Provide synthesis resource usage and timing report.
    - Should include ALUTs, Registers, Area, Fmax snapshots
    - Provide fitter report snapshot
    - Provide Timing Fmax report snapshots
  - Make sure to use **Arria II GX EP2AGX45DF29I5** device and use Fmax for **Slow 900mV 100C Mod**

simplified_sha256.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 %)

simplified_sha256.sta

- Copy of the sta (static timing analysis) reports.
- Make sure to use Fmax for **Slow 900mV 100C Model**
- **IMPORTANT**: Make sure “clk” is the ONLY clock.
- You must,
  assign mem_clk = clk;

```
+-----------------------------------------------------+
; Slow 900mV 100C Model Fmax Summary                  ;
+------------+-----------------+------------+------+
; Fmax       ; Restricted Fmax ; Clock Name ; Note ;
+------------+-----------------+------------+------+
; 151.95 MHz ; 151.95 MHz      ; clk        ;      ;
+------------+-----------------+------------+------+
```

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))

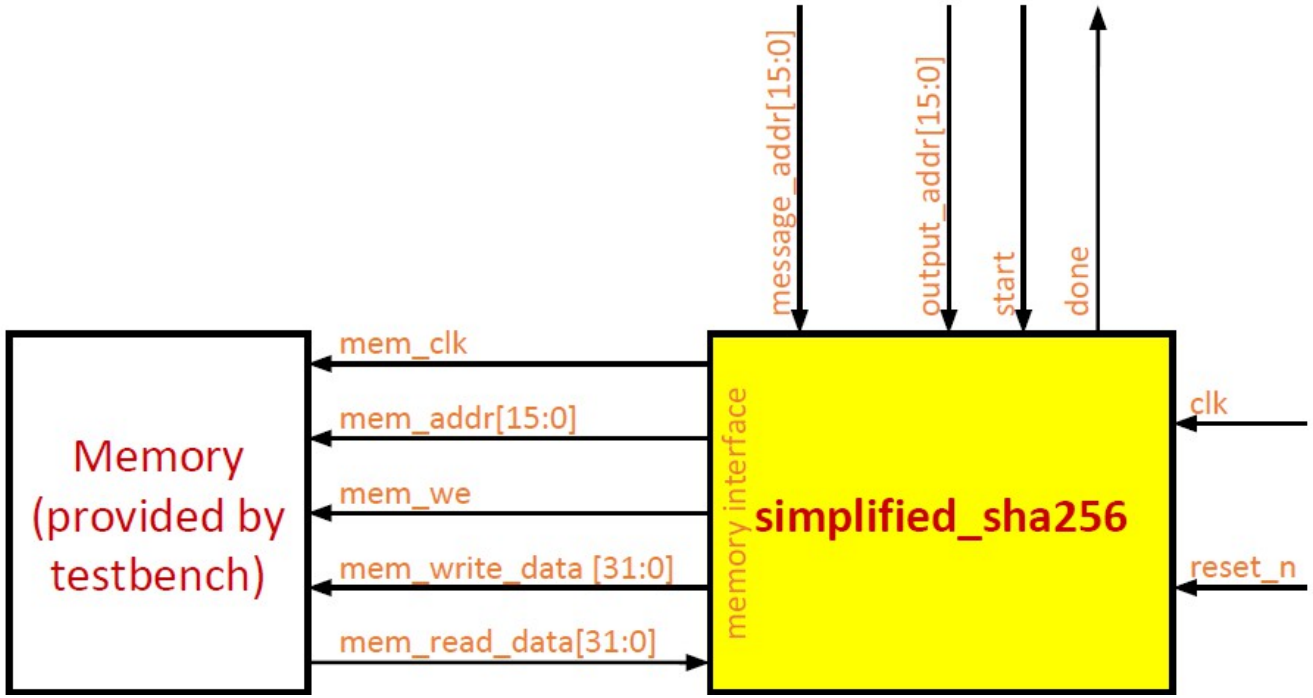
```
// right rotation
function logic [31:0] rightrotate(input logic [31:0] x,
                                  input logic [ 7:0] r);
   rightrotate = (x >> r) | (x << (32-r));
endfunction
```

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

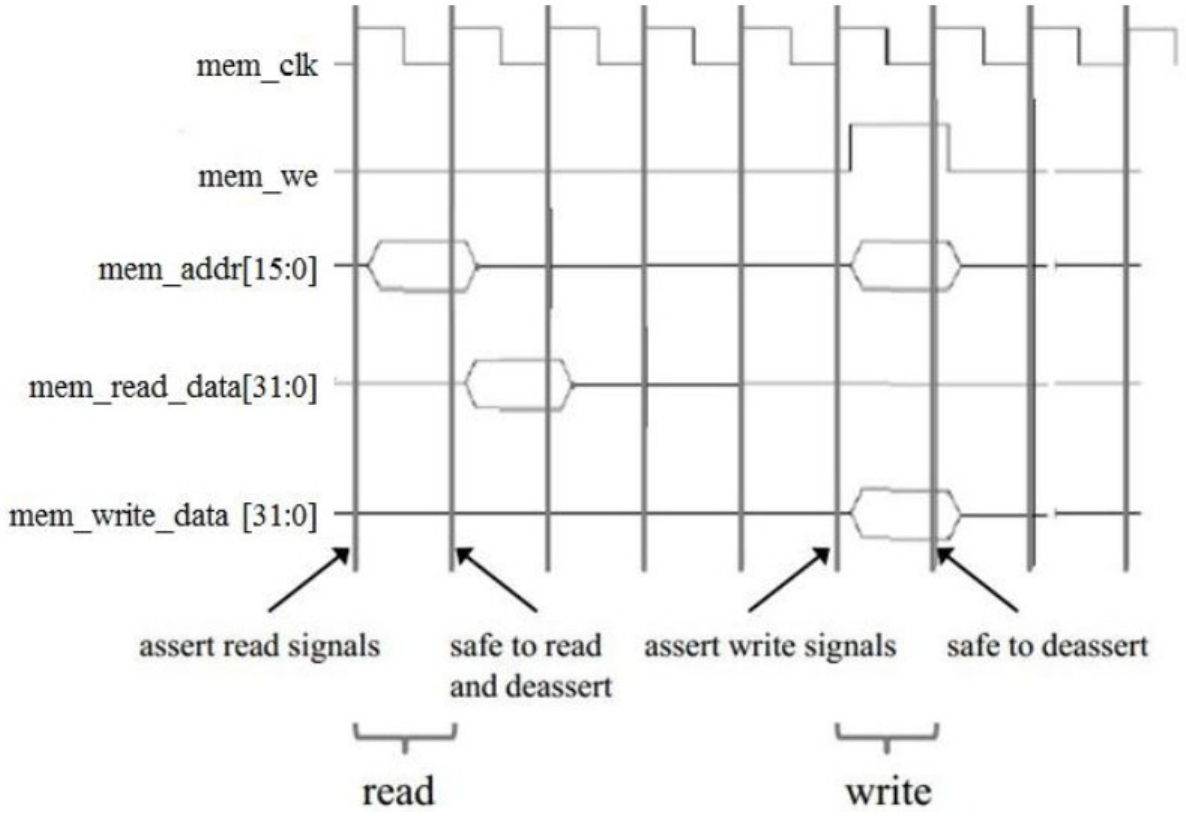
Memory Model

- 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

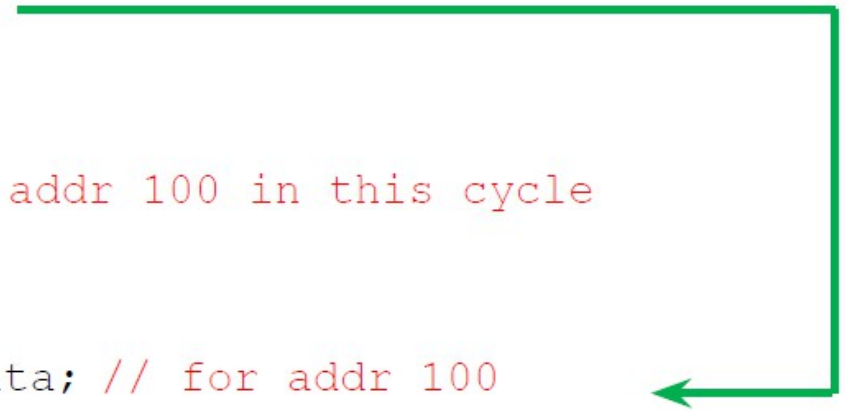
- THIS IS INCORRECT

```
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

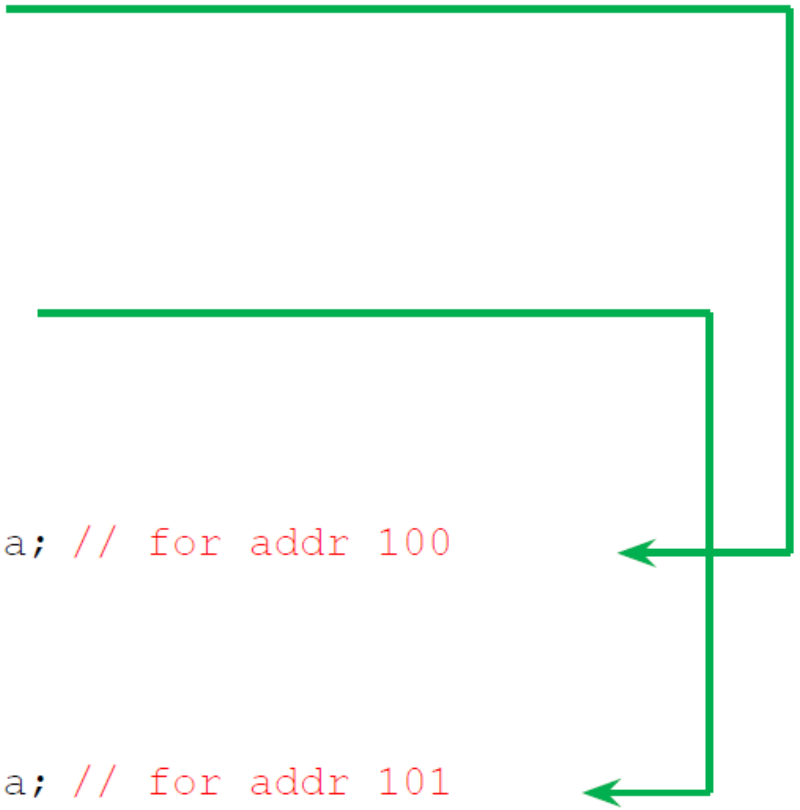
- Have to wait an extra cycle, correct way of reading from memory

```
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

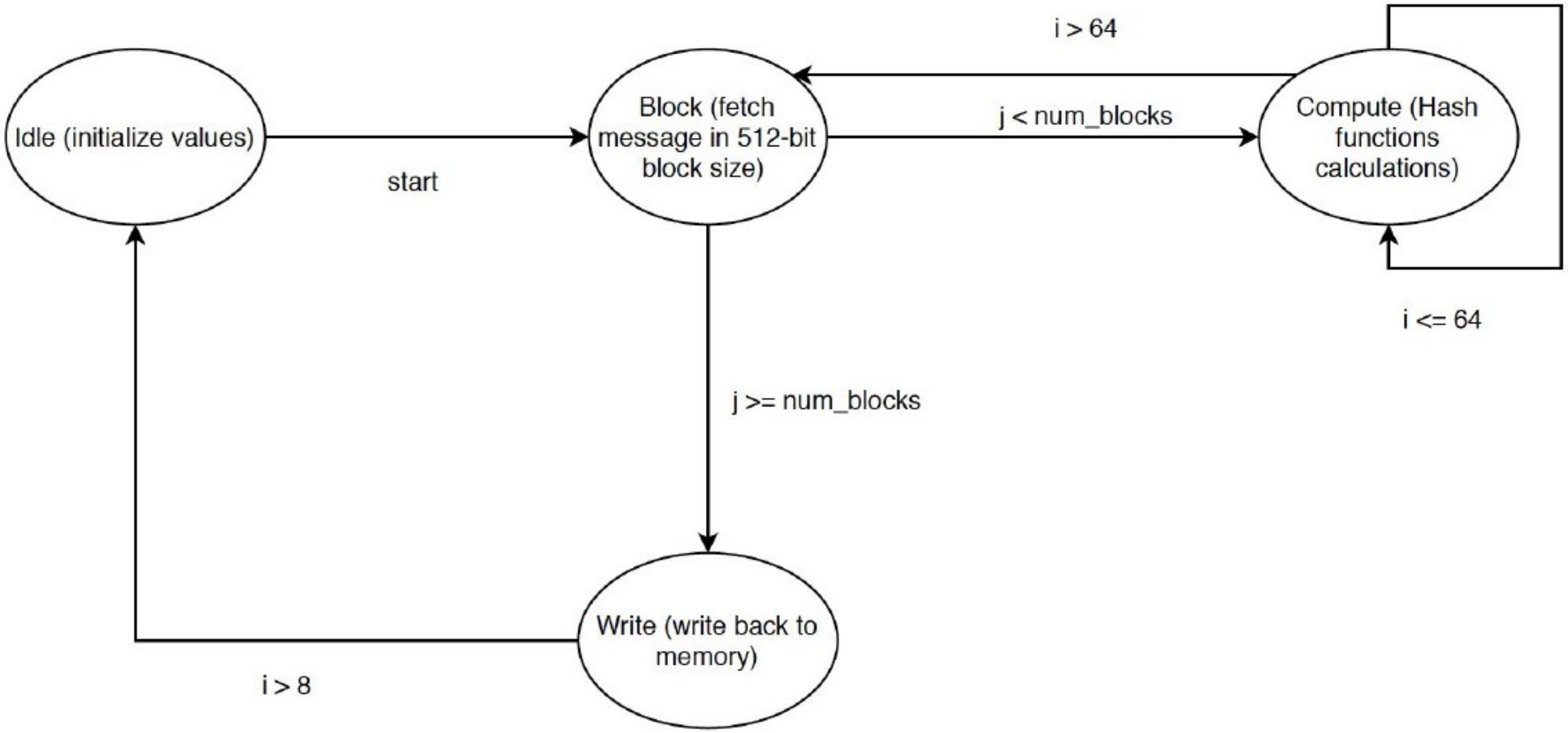
- Notice here that we assign address to mem_addr and data to mem_write_data in the same cycle.

```
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

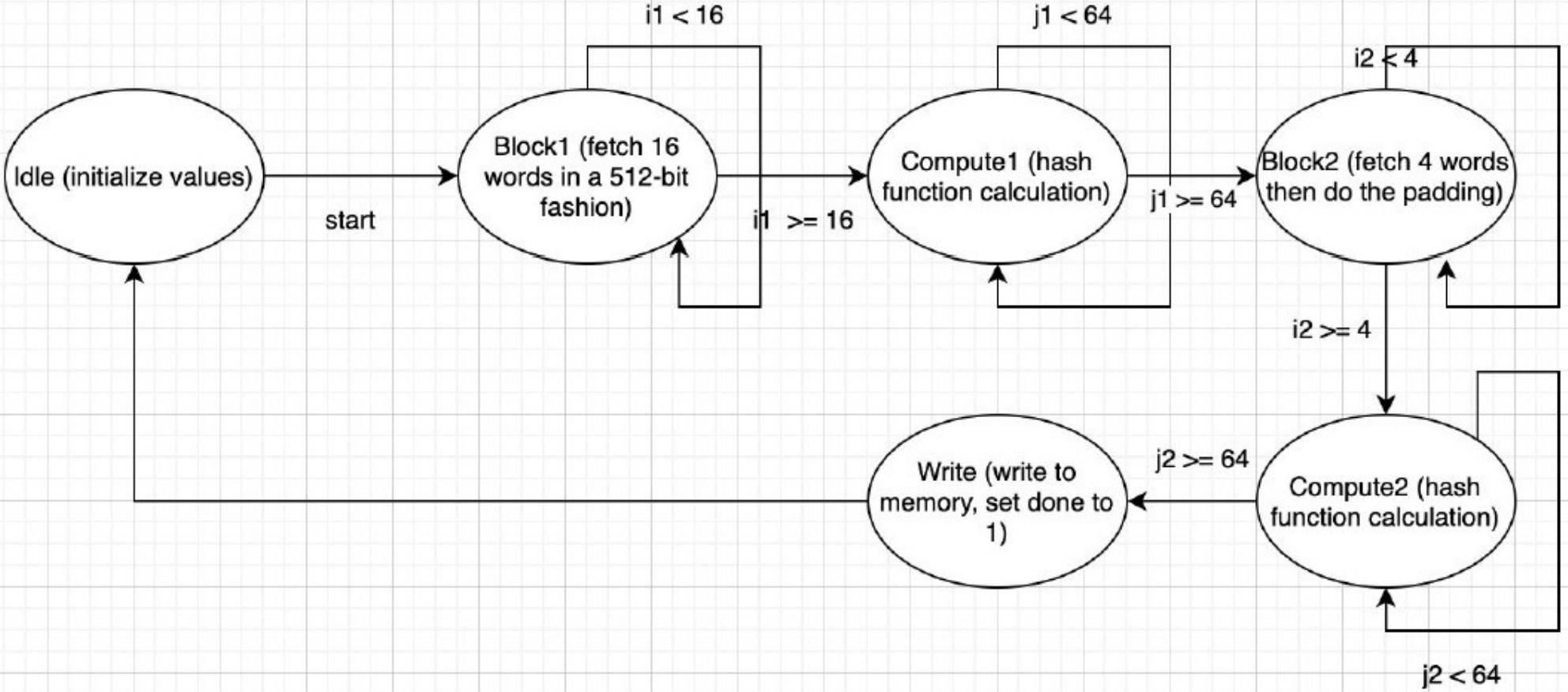


i1 := first message block index

i2 := second message block index

j1 := compute counter variable for first block

j2 := compute counter variable for second block



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