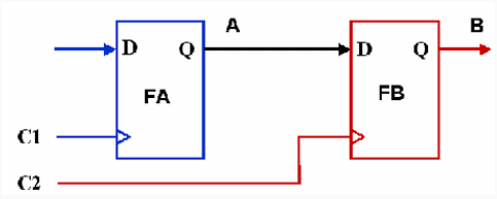
CROSS DOMAIN CROSSING

Clock Domain Crossing refers to the process of transferring data or signals between two different clock domains in a digital design. These domains may operate at different frequencies or have no defined phase relationship.
A clock domain crossing occurs whenever data is transferred from a flop driven by one clock to a flop driven by another clock



CLOCK DOMAIN ISSUES:

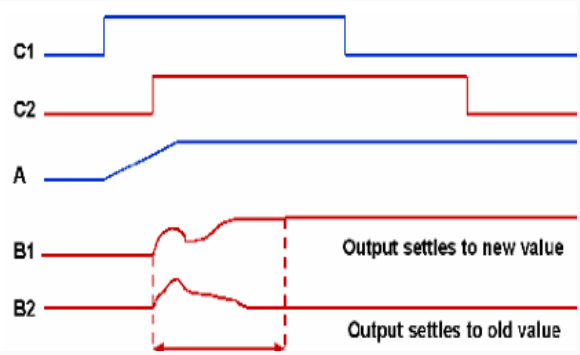
➢ Metastability

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". As a result, the output signal B may oscillate for an indefinite amount of time. Thus the output is unstable and may or may not settle down to some stable value before the next clock edge of C2 arrives. This phenomenon is known as metastability and the flop "FB" is said to have entered a metastable state.

METASTABILITY CONSEQUENCES:

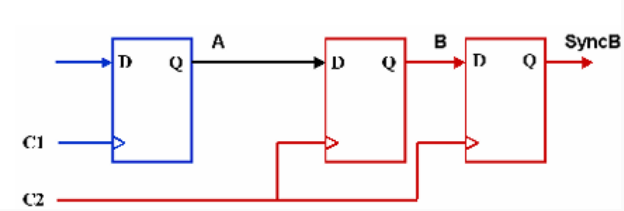
- High current flow .
- Different values on different fan-out cones.
- Output is known settles to any value.

SOLUTIONS:
Metastability problems can be avoided by adding special structures known as synchronizers in the destination domain. The synchronizers allow sufficient time for the oscillations to settle down and ensure that a stable output is obtained in the destination domain.



TWO- STAGE SYNCHRONIZER

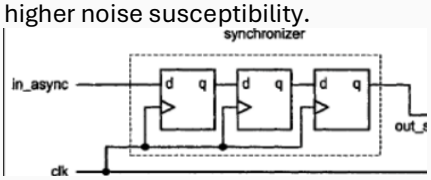
Used for **Single-Bit Signal Synchronization**: Such as reset signals, enable signals, or control flags.



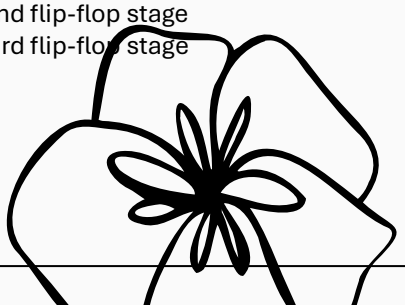

```
// 2-Stage Synchronizer
module two_stage_synchronizer (
   input  wire clk,    // Destination clock domain
  input  wire async_in, // Asynchronous input signal
  output reg  sync_out // Synchronized output signal
);
   reg sync_stage1; // First stage flip-flop
   always @(posedge clk) begin
      sync_stage1 <= async_in;
      sync_out   <= sync_stage1;
   end
endmodule
```

THREE STAGE SYNCHRONIZER

A 3-stage synchronizer is used for critical clock domain crossings where a higher level of metastability protection is required.
Each additional stage gives the metastable state more time to settle.
Particularly useful in high-frequency designs or environments with higher noise susceptibility.

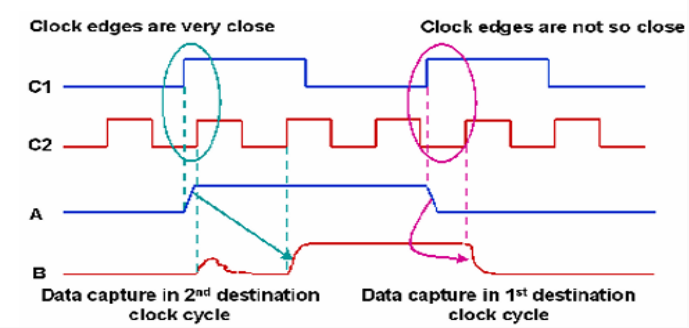



```
module synchronizer_3stage (
   input wire clk_dest,    // Destination clock
   input wire async_signal,  // Asynchronous input
   output reg sync_signal   // Synchronized output
);
  reg stage1, stage2;
always @(posedge clk_dest) begin
  stage1 <= async_signal; // First flip-flop stage
  stage2 <= stage1;      // Second flip-flop stage
  sync_signal <= stage2;  // Third flip-flop stage
   end
endmodule
```



➢ Data loss

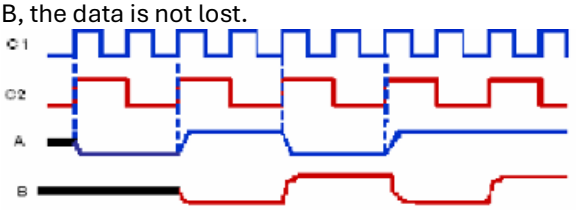
Whenever a new source data is generated, it may not be captured by the destination domain in the very first cycle of the destination clock because of metastability. As long as each transition on the source signal is captured in the destination domain, data is not lost.



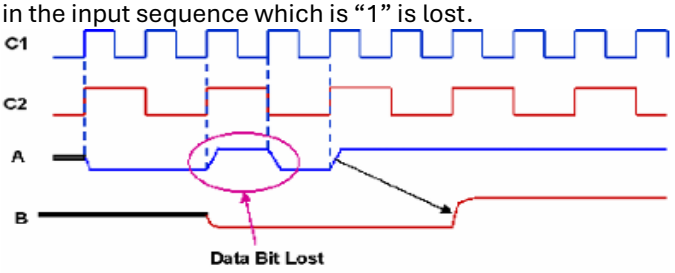
If the active clock edges of C1 and C2 arrive close together, the first clock edge of C2, which comes after the transition on source data A, is not able to capture it. The data finally gets captured by the second edge of clock C2.
However, if there is sufficient time between the transition on data A and the active edge of clock C2, the data is captured in the destination domain in the first cycle of C2

Ex: Assume that the source clock C1 is twice as fast as the destination clock C2 and there is no phase difference between the two clocks.

Assume that the input data sequence "A" generated on the positive edge of clock C1 is "00110011". The data B captured on the positive edge of clock C2 will be "0101". Here, since all the transitions on signal A are captured by B, the data is not lost.



If the input sequence is "00101111", then the output in the destination domain will be "0011". Here the third data value in the input sequence which is "1" is lost.



Solutions:
In order to prevent data loss, the data should be held constant in the source domain long enough to be properly captured in the destination domain. In other words, after every transition on source data, at least one destination clock edge should arrive where there is no setup or hold violation so that the source data is captured properly in the destination domain.
Can be done by using finite state machine (FSM) can be used to generate source data at a rate, such that it is stable for at least 1 complete cycle of the destination clock. This can be generally useful for synchronous clocks when their frequencies are known. For asynchronous clock domain crossings, techniques like handshake and asynchronous FIFO are more suitable.

```
module handshake_synchronizer_2stage (
  input wire clk_src,       // Source clock
  input wire clk_dest,      // Destination clock
  input wire req_src,   // Request signal from source
  input wire [7:0] data_in,   // Data from source
  output reg ack_dest,   // Acknowledge signal in destination
  output reg [7:0] data_out    // Data in destination domain
);
  reg req_sync1, req_sync2;   // Synchronization of req_src
  reg ack_sync1, ack_src;     // Synchronization of ack_dest
 always @(posedge clk_dest) begin
    req_sync1 <= req_src;
    req_sync2 <= req_sync1;end
always @(posedge clk_dest) begin
   if (req_sync2 && !ack_dest) begin
      data_out <= data_in;  // Capture data from source
      ack_dest <= 1'b1;     // Assert acknowledge signal
    end else if (!req_sync2) begin
      ack_dest <= 1'b0;
    end
  end
always @(posedge clk_src) begin
ack_sync1 <= ack_dest;
 ack_src <= ack_sync1;
 end endmodule
```

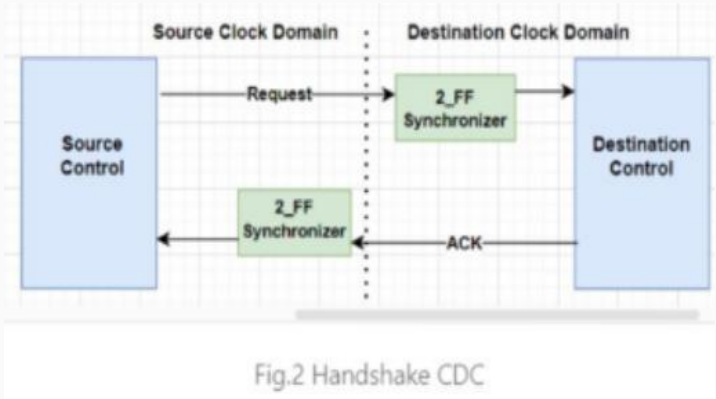
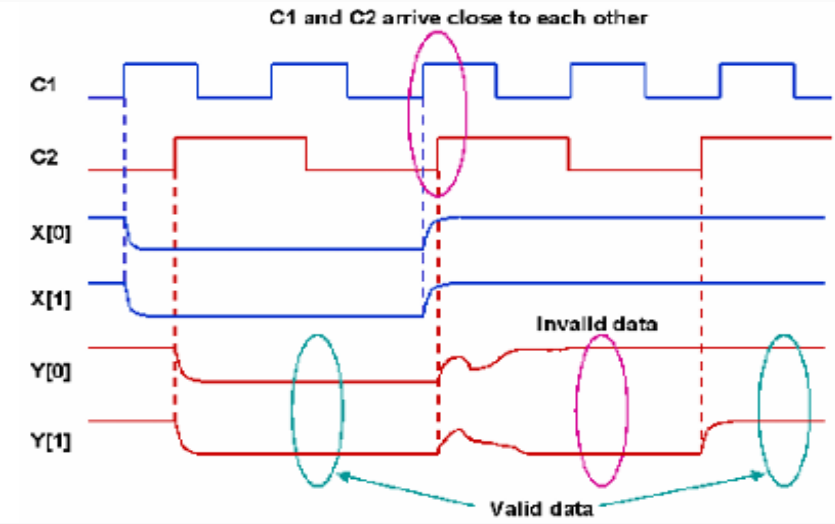


Fig.2 Handshake CDC

➢ Data Incoherency

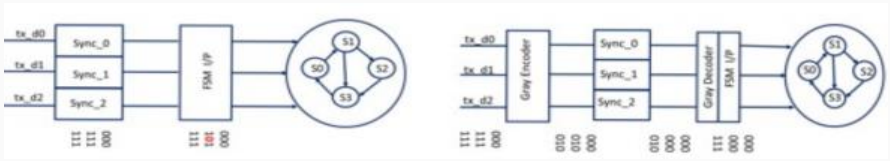
Consider a case where multiple signals are being transferred from one clock domain to another and each signal is synchronized separately using a multi-flop synchronizer. If all the signals are changing simultaneously and the source and destination clock edges arrive close together, some of the signals may get captured in the destination domain in the first clock cycle while some others may be captured in the second clock cycle by virtue of metastability. This may result in an invalid combination of values on the signals at the destination side. Data coherency is said to have been lost in such a case.



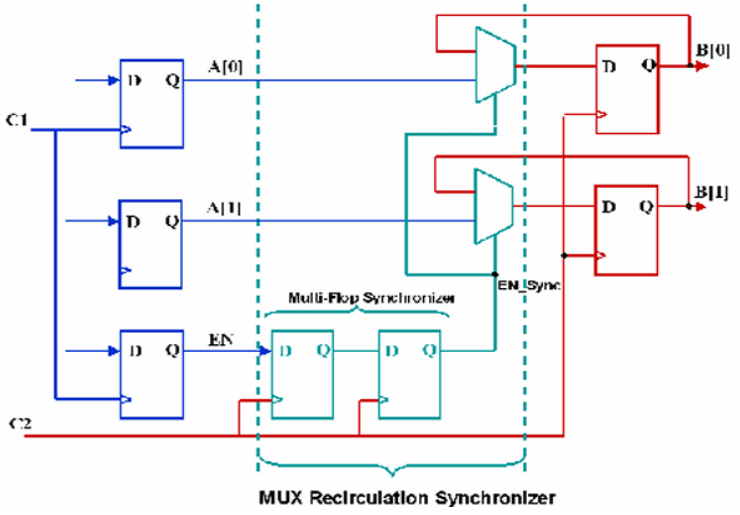
For example:
Assume that “00” and “11” are two valid values for a signal X[0:1] generated by clock C1.
Initially there is a transition from 1->0 on both the bits of X. Both the transitions get captured by clock C2 in the first cycle itself.
Hence the signal Y[0:1] becomes “00”
Next, there is a transition from 0->1 on both the bits of signal X. Here the rising edge of clock C2 comes close to the transition on signal X. While the transition on X[0] is captured in the first clock cycle, the transition on X[1] gets captured in second clock cycle of C2. This results in an intermediate value of “10” on Y[0:1] which is an invalid state. Data coherency is lost in this case.

SOLUTIONS:
problem results because all the bits are not changing to a new state in the same cycle of destination clock. If all the bits either retain their original value or change to the new value in the same cycle, then the design either remains in the original state or goes to a correct new state.
Now, if the circuit is designed in such a way that while changing the design from one state to another, only one bit change is required, then either that bit would change to a new value or would retain the original value. Since all the other bits have the same value in both the states, the complete bus will either change to the new value or retain the original value in this case.
This in turn implies that if the bus is Gray-encoded, the problem would get resolved and an invalid state would never be obtained. For Data bus use MUX Recirculation or FIFO

GREY ENCODED

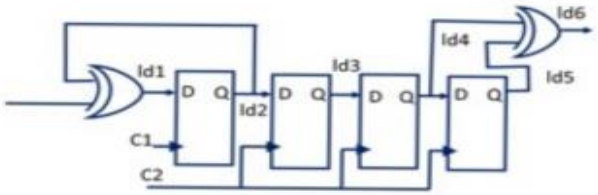


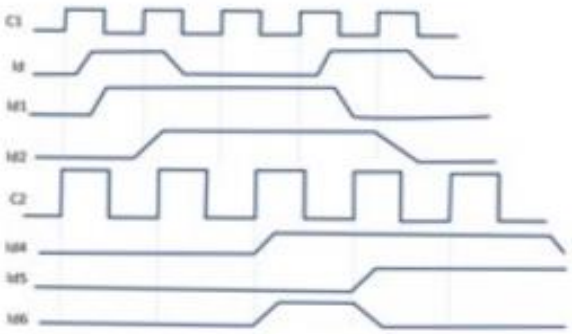
```
module mux_recirculation_synchronizer (
   input wire clk_src,              // Source clock domain
   input wire clk_dest,             // Destination clock domain
   input wire [1:0] data_in,    // 2-bit input data from source
   input wire en,                 // Enable signal from source
 output reg [1:0] data_out   // Output synchronization
);
   reg en_sync1, en_sync2;  // Multi-flop sync. for EN
   wire en_sync;                 // Synchronized enable signal
   assign en_sync = en_sync2 // Final synchronized enable
   reg [1:0] data_ff;      // Flip-flop to hold synchronized data
always @(posedge clk_dest) begin
      en_sync1 <= en;      // First stage of EN synchronization
      en_sync2 <= en_sync1;  // Second stage of EN sync.
   end
always @(posedge clk_dest) begin
      if (en_sync)         // When enable signal is synchronized
 data_ff <= data_in;      // Capture the data from the source
      else
    data_ff <= data_ff;   // Retain previous data
data_out <= data_ff;    // Update the synchronized output
   end
endmodule
```



TOGGLE SYNCHRONIZER:

It avoids metastability issues by leveraging the toggling of a single bit. The toggle flips its value in the source domain and then propagates across clock domains using multi-flop synchronization.






```
module toggle_synchronizer (
  input wire clk_src,       // Source clock
  input wire clk_dest,      // Destination clock
  input wire event_src,     // Event signal in source domain
  output wire event_dest    // Synchronized event in destination domain
);
reg toggle_src;          // Toggle bit in source domain
  reg sync1, sync2;        // Synchronizer flip-flops in destination domain
  reg toggle_dest;         // Toggle bit in destination domain
// Step 1: Generate the toggle signal in the source domain
  always @(posedge clk_src) begin
    if (event_src)
      toggle_src <= ~toggle_src;  // Flip the toggle bit on event
  end
 always @(posedge clk_dest) begin
    sync1 <= toggle_src;         // First stage of synchronization
    sync2 <= sync1;              // Second stage of synchronization
  end
  // Step 3: Detect the toggle change in the destination domain
  always @(posedge clk_dest) begin
    toggle_dest <= sync2;        // Capture the synchronized toggle bit
  end
  // Generate the event in the destination domain when toggle changes
  assign event_dest = toggle_dest ^ sync2; // XOR to detect edge
endmodule
```