ECE260B Winter 22

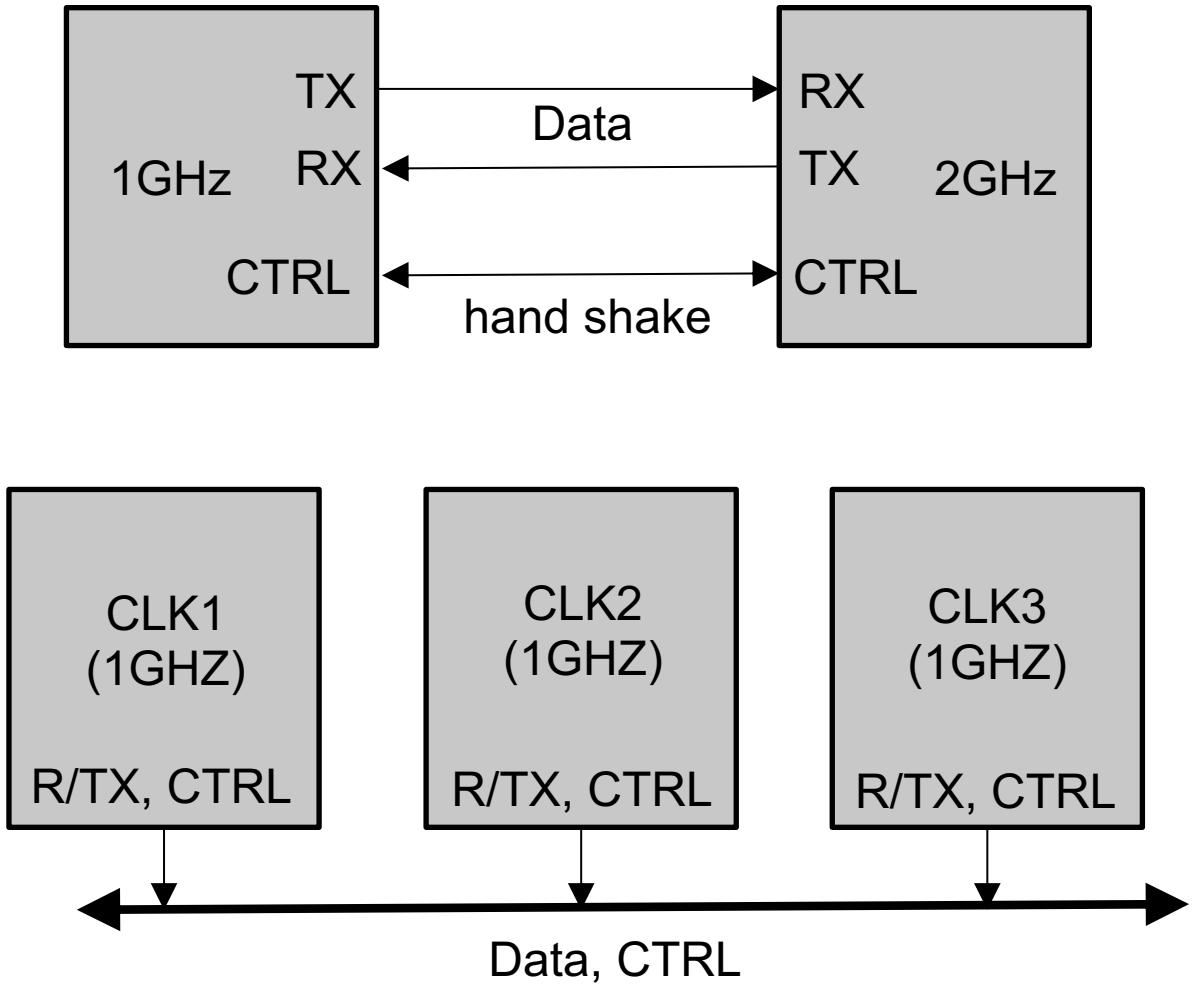
Clock Domain Crossing

Prof. Mingu Kang

Announcement

- **Lecture recording:** in Media Gallery
- **Upcoming HW this week:** deadline March 1st 1pm (Tuesday)
- **Midterm Announcement:** during Thursday lecture (17th)
- **TA will generate a project team randomly this week** if you haven't done so

Clock Domain Crossing (Asynchronous Interface)



Examples of asynchronous interfaces

- **Asynchronous interface**
  - Interface between two or more modules operating with independent clocks
  - Async interface helps independent operation of modules (e.g., freq. modulation)
  - Even in a single core, many modules operate on its own clock domain
  - Also, helps timing closure during P.D
  - Though the freq. is identical, if the clock is independent, it is async interface
  - Despite the source is common, the clock tree is different, it is safe to treat as async.

Synchronous Interface Solutions

- **Synchronizer**

  - minimize the meta stability at async. interface

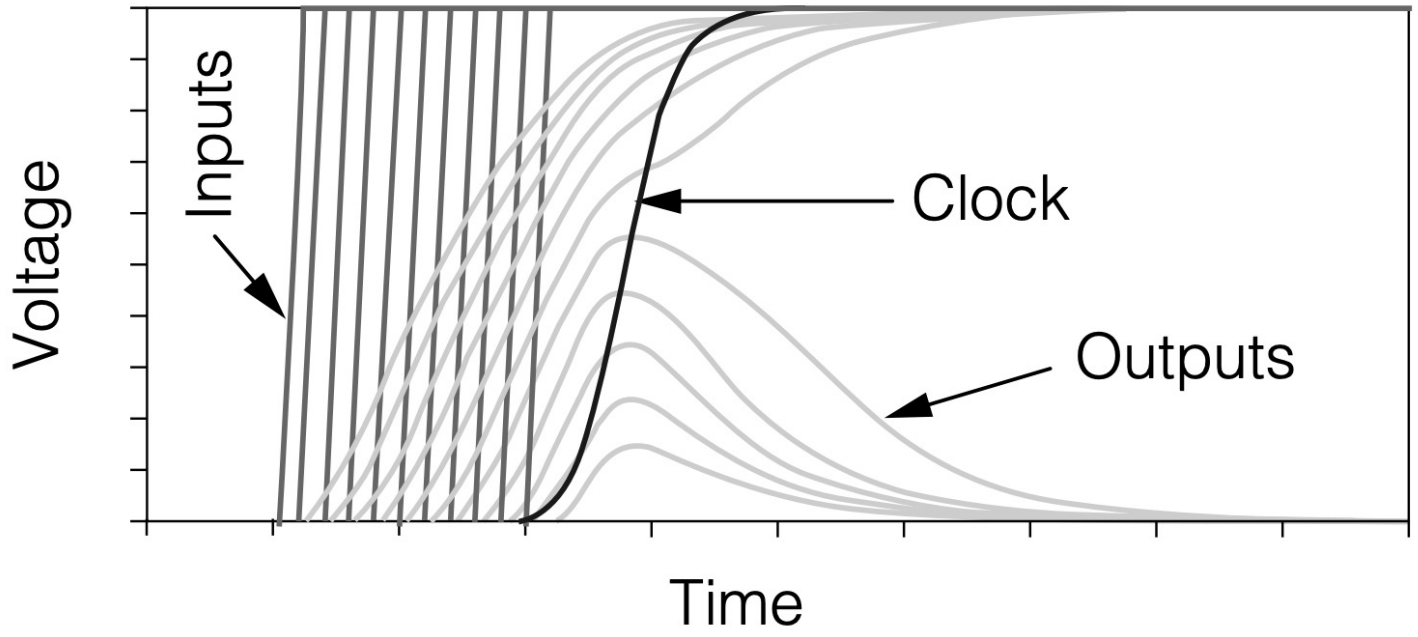
  - often used when a single bit is delivered

- **Hand shaking protocol**

  - used to deliver multiple bits through async. interface

- **FIFO**
  - multi-bit support + fast

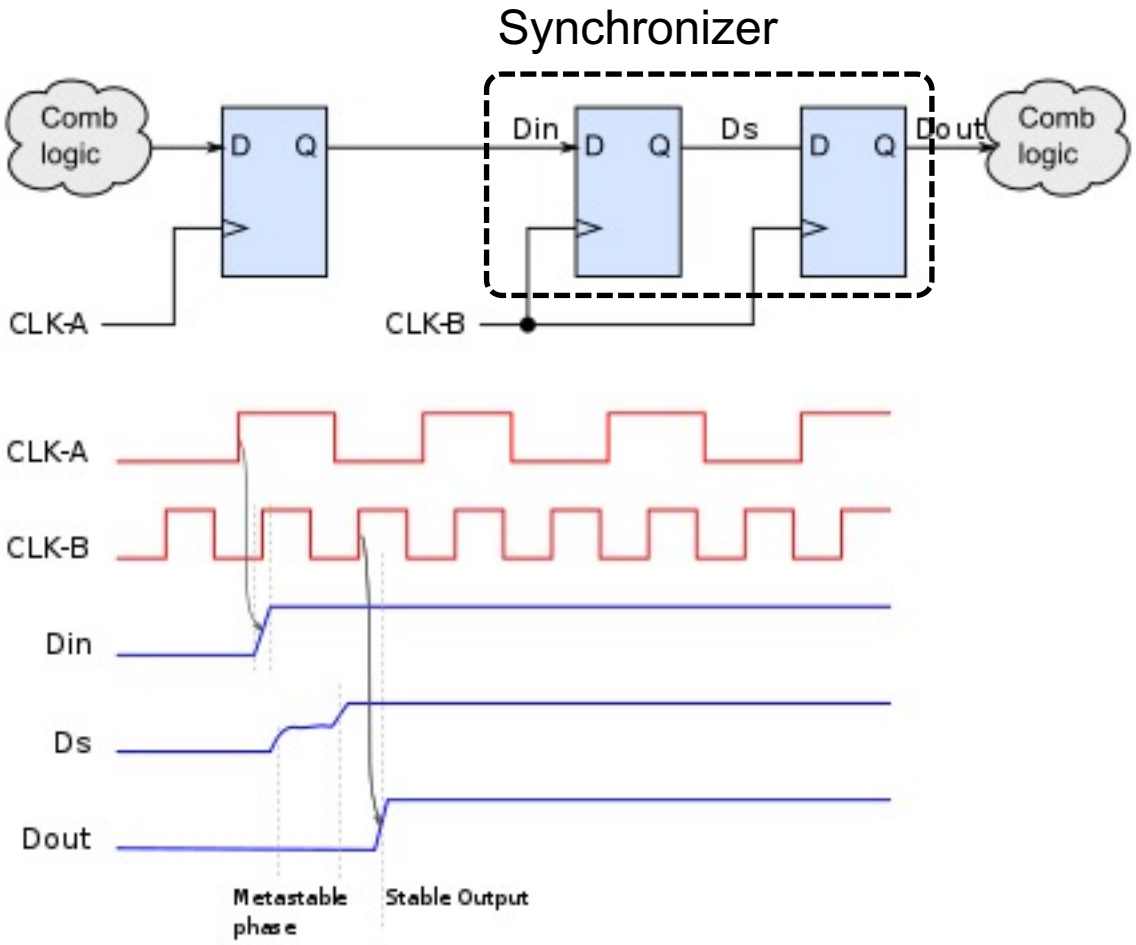
Meta-Stability at Async Interface



- The timing of input w.r.t. the receiving clock is completely random at async interface
- No guarantee that set-up time requirement is met
- Thus, meta-stability can happen

https://webee.technion.ac.il/people/ran/papers/Metastability-and-Synchronizers.IEEEDToct2011.pdf

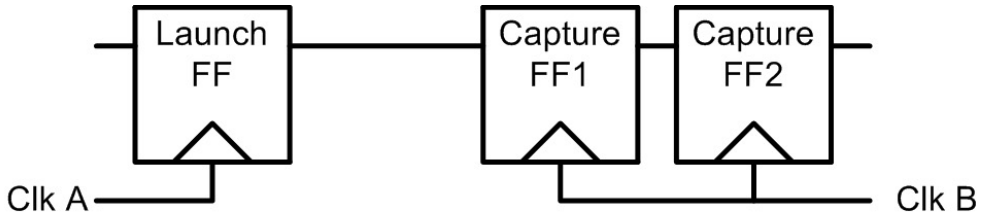
Synchronizer

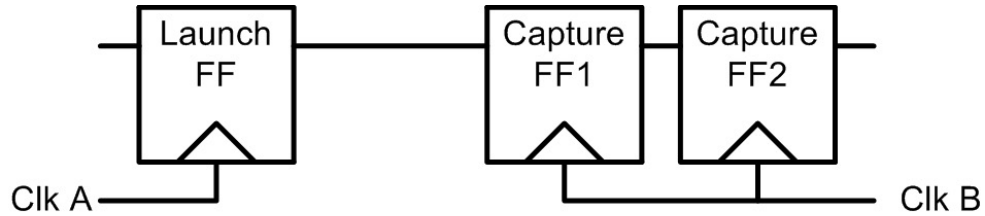


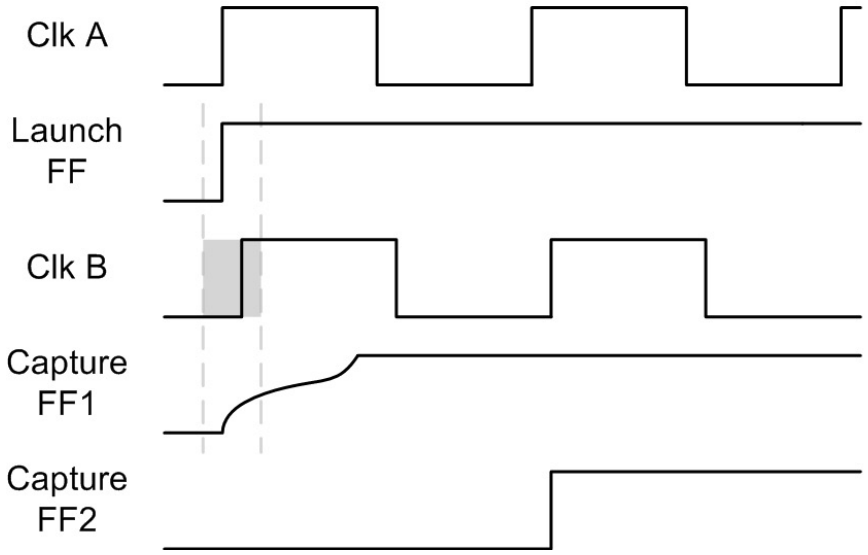
- Synchronizer: closely located 2 or 3 flip flop (FF)s
- 2nd FF's output are either 0 or 1 with high probability
- Combinational logic is protected from meta stable data
- Three FFs provides better stability
- Special STD cells are often provided by PDK

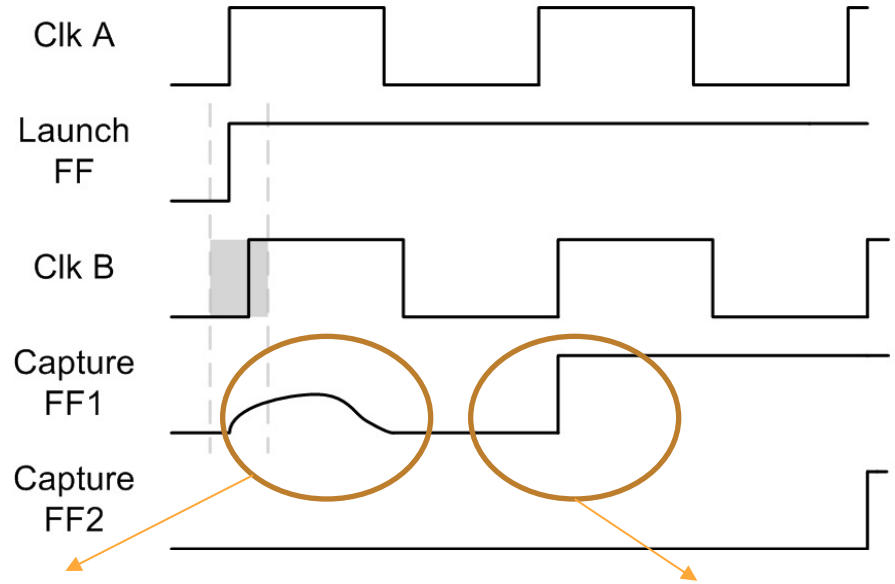
https://www.wikiwand.com/en/Metastability_(electronics)

Timing Uncertainty in Synchronizer









Fail to flip data due to meta-stability

But, flipped in the next cycle

- For this nature, cycle-accurate timing prediction is difficult at the async interface

Synchronizer Implementation

```
module sync (in, clk, out);
  input in, clk;
  ouput reg out;

  reg int_q;
  always @ (posedge clk) begin
     int_q  <= in;
     out    <= int_q;
  end

endmodule
```

Behavioral code for synchronizer

```
module top( … )

wire data_clk1;
wire data_clk2;

sync sync_instance (
        .in(data_clk1),
        .clk(clk2),
        .out(data_clk2)
);
…
```

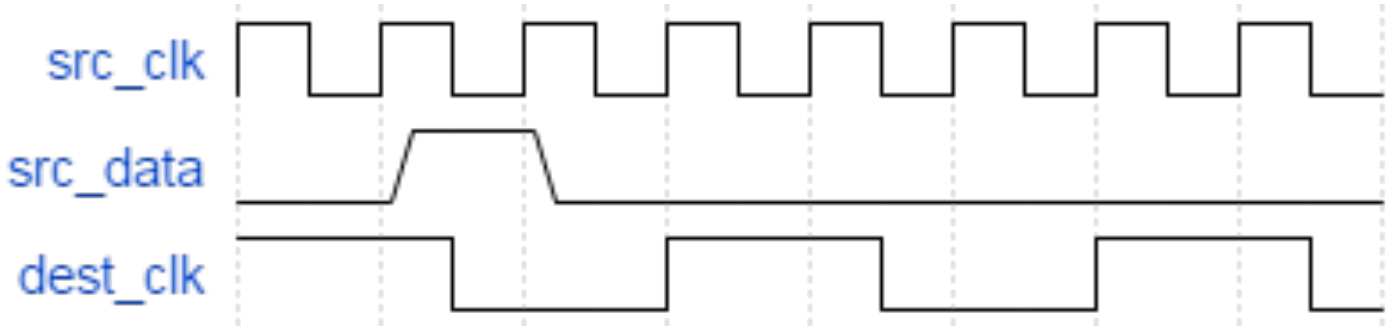
Top module during behavioral sim

```
module top( … )

wire data_clk1;
wire data_clk2;

SDFFQ1MA10TR sync_inst(
        .in(data_clk1),
        .clk(clk1),
        .out(data_clk2)
);
…
```

Top module for tape-out

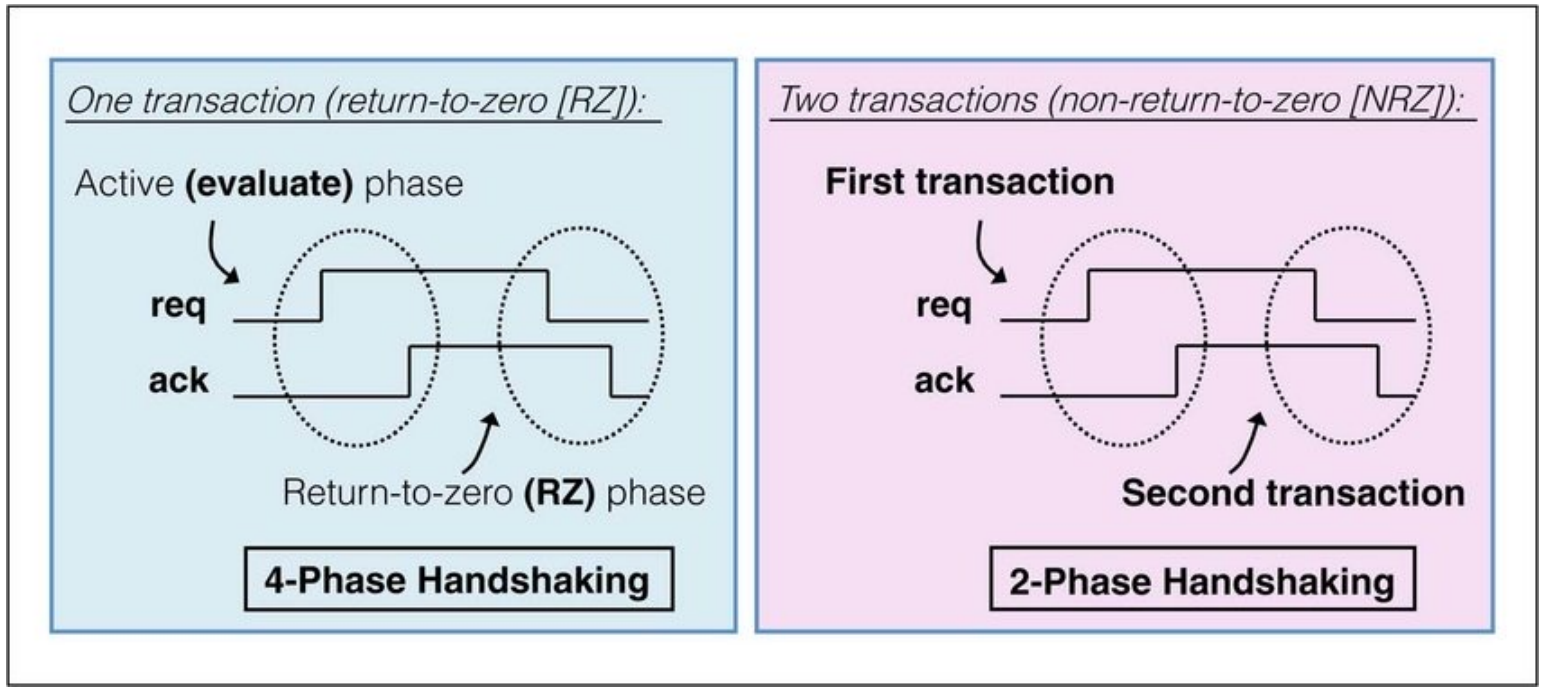
- SDFFQ1MA10TR: special STD cell to maximize the MTTF (mean-time-to-failure) by meta-stability

Synchronizer Failure

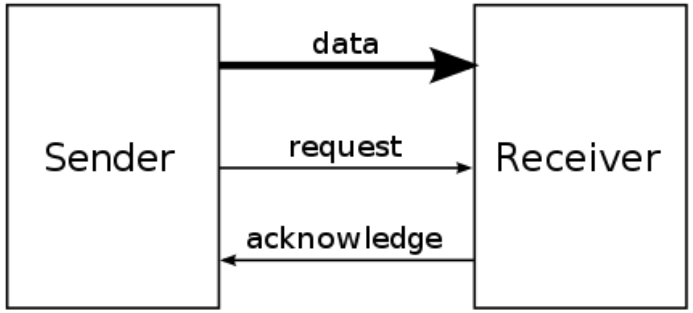


- Synchronizer is a good solution to deliver a single bit data from slow to fast domain crossing
- Not a good solution from fast to slow clock as pulse can be missing as above
- Also, not suitable to deliver multi-bit due to many FFs due to the **timing uncertainty between the data bits**

Hand Shaking Protocol

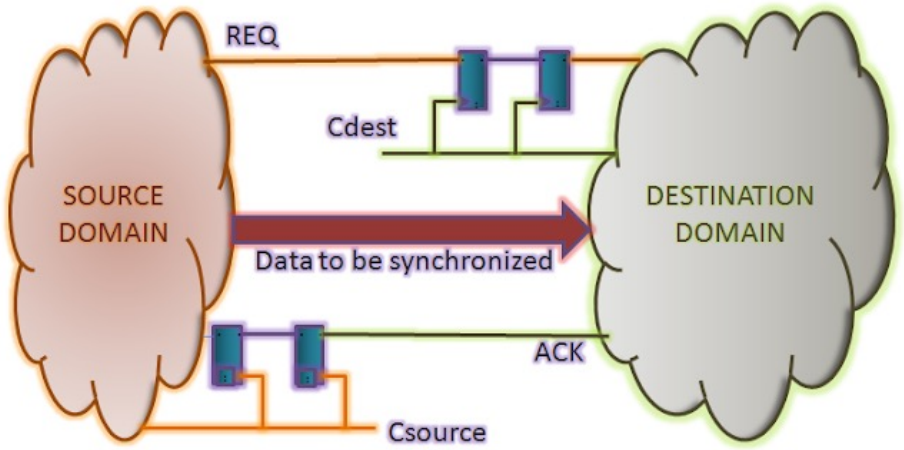


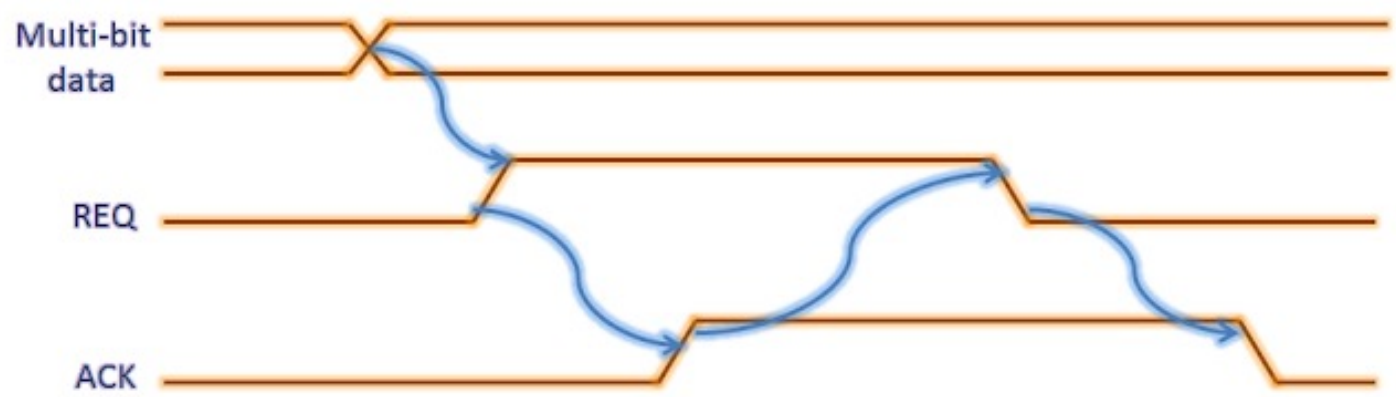
- Hand-shaking is a good solution for multi-bit data to address the issue of synchronizer
- But, speed is limited → 2-phase version achieves better speed



https://www.researchgate.net/figure/Asynchronous-handshake-protocols_fig2_245530456

Hand Shaking Implementation Details





- Data should be prepared before REQ signal
- REQ signal at the receiver is synchronized with receiver clk → no setup/hold time issue
- If REQ is high at the receiver, it means the data is already valid
  → thus, capture the data & enable ACK signal
- ACK is synchronized at transmitter CLK → no setup/hold time issue
- Once ACK is high at transmitter, it means the data has been captured by receiver
  → thus, REQ is disabled and another data packet can be issued

https://vlsiuniverse.blogspot.com/2013/09/synchronization-schemes.html

FIFO (First-in First-out)

- **FIFO at the async. interface**
  - Synchronizer is fast but for a single bit
  - Hand shaking is for multi-bit but slow
  - FIFO is most widely used solution as it is fast and multi-bit
  - FIFO is also used within the same clk domain when in / out rates are different

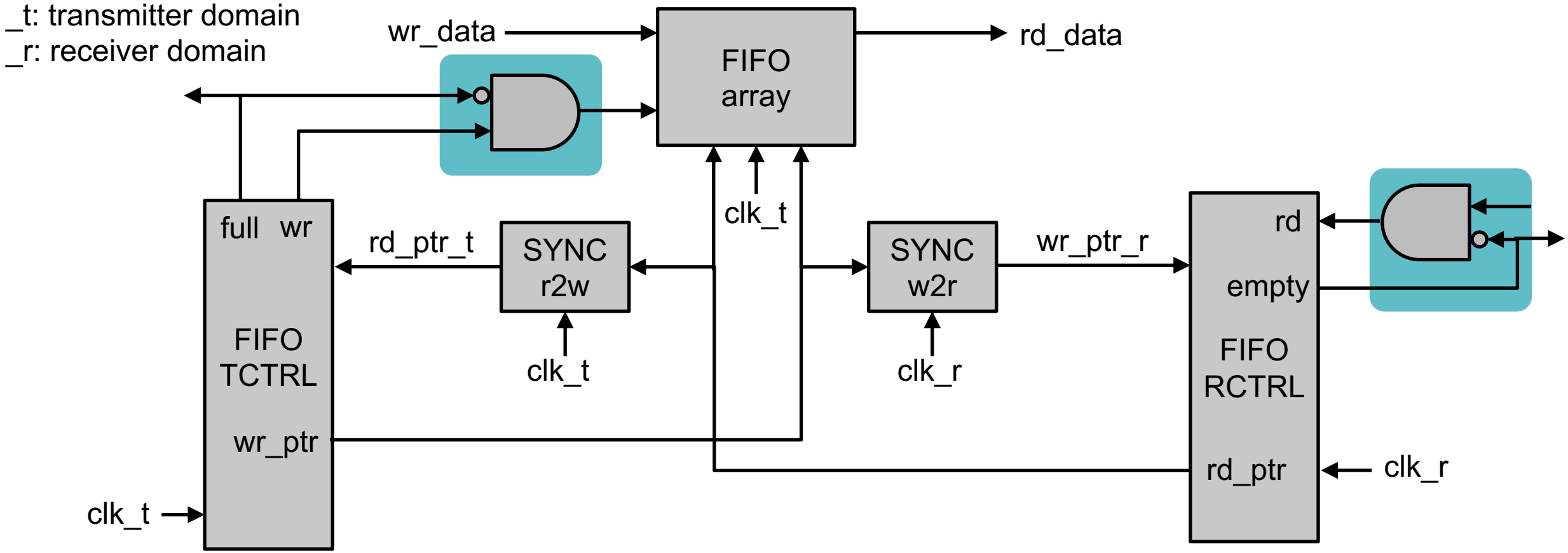
Synchronous (Single-CLK) FIFO Implementation

- **FIFO with same clock domain**
  - Assume transmitter & receiver are at same clock domain
  - But writing and reading are done at random timing
  - When the FIFO depth is 2^D, equip D+1 bits of rd_ptr[D:0] and wr_ptr[D:0]
  - Whenever new value is written, wr_ptr <= wr_ptr +1
  - Whenever new value is, read rd_ptr <= rd_ptr + 1
  - Note data does not move in the array, but only pointer increments
  - wr_ptr and rd_ptr wrap around even after it reaches 2^D without resetting to zero
  - Current output is given as a combinational logic, i.e., assign out = array[rd_ptr]
  - if wr_ptr == rd_ptr, there is no contents, i.e., empty <= 1, and receiver cannot read any more
  - if "abs(wr_ptr - rd_ptr) == 2^D", the fifo is full, i.e., full <= 1, and transmitter cannot send any more
  - Above condition is equivalent to (wr_ptr[D-1:0] == rd_ptr[D-1:0]) && (wr_ptr[D] != rd_ptr[D])

Verilog Example

- **FIFO with same clock domain**
  - Assume each data is 4 bit, and FIFO depth is 64
  - Send data “data2.txt”, which included 64 sequence through the FIFO
  - Check the received data are same as the sent data with similar method in HW1
  - Check rp_ptr (read pointer) and wr_ptr and the contents and output

Asynchronous (Dual-CLK) FIFO Implementation



- wr_ptr / rd_ptr are exchanged through synchronizer to produce full and empty signals
- wr_ptr is not incremented when full = 1 while rd_ptr is not incremented while empty = 1
- full and empty signal should be given to transmitter and receiver circuitry to inform wr / rd are not available

https://www.verilogpro.com/asynchronous-fifo-design/

Issues in FIFO at Clock Domain Crossing

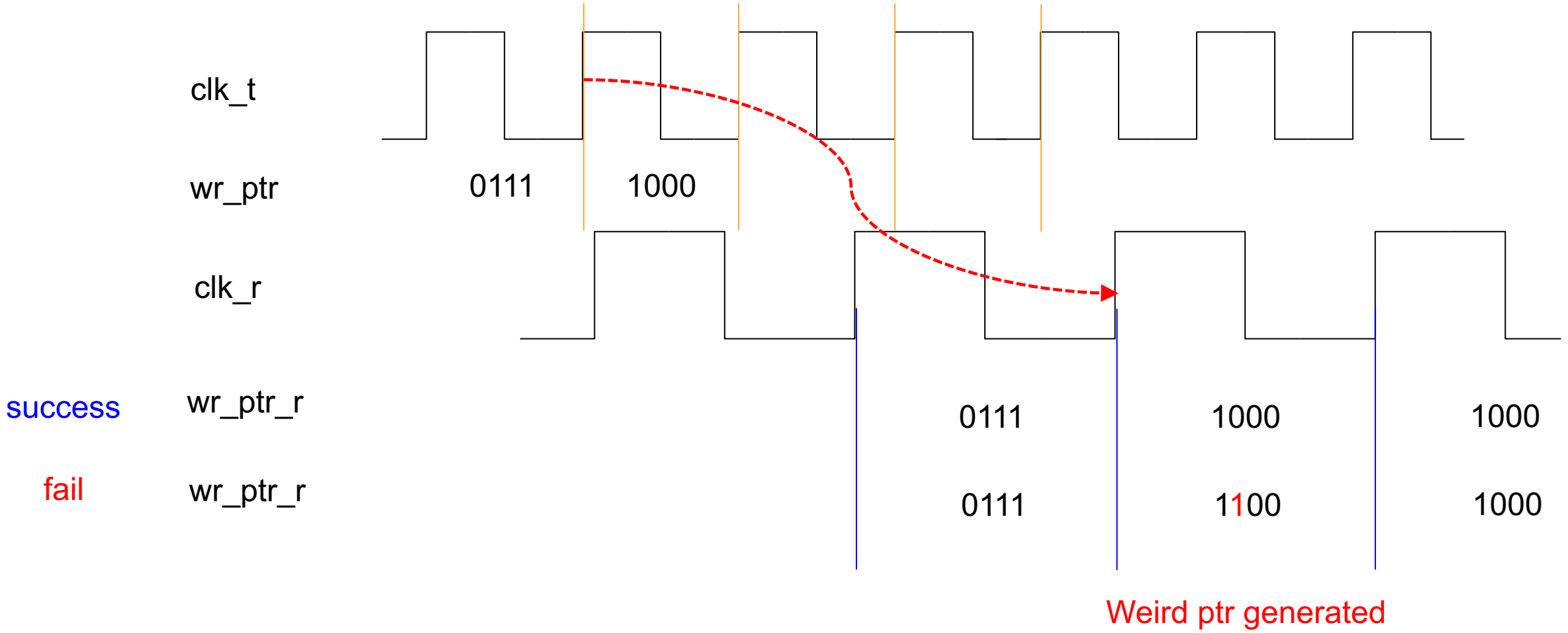
- Note multi-bit rd_ptr / wr_prt cannot cross the async. interface

  → Gray code is required

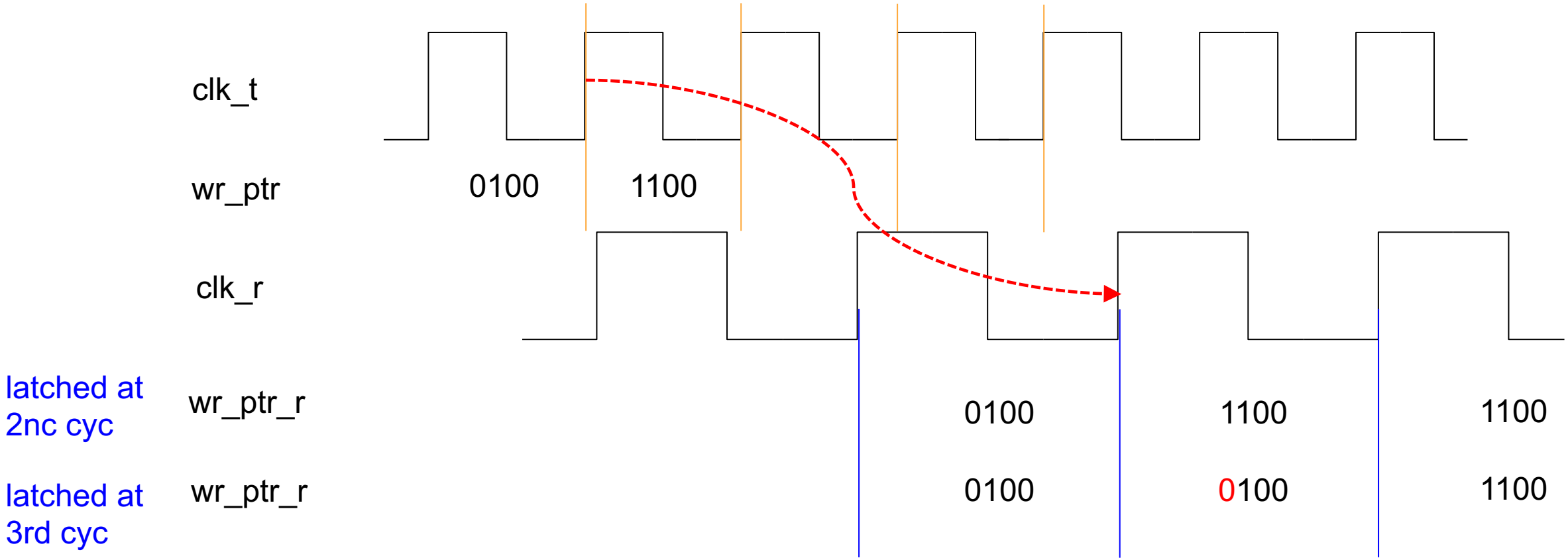
- Relying on full / empty might not be the best practice considering system response time

  → It is safe to equip “almost full” and “almost empty” signals when *th* number before full or empty happens

Binary Code


clk_t
wr_ptr
0111
1000
clk_r
success
wr_ptr_r
0111
1000
1000
fail
wr_ptr_r
0111
1100
1000
Weird ptr generated

Gray Code



Weird ptr does not happens

Gray Code

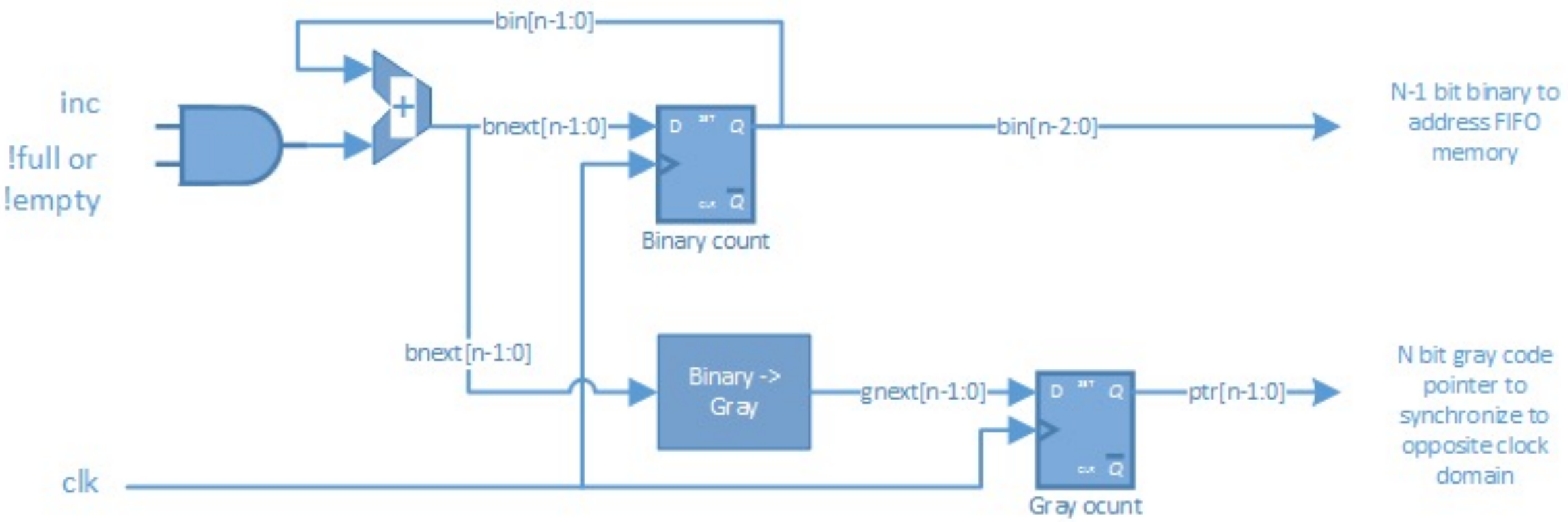
Decimal Count	Binary Count	Gray Code Count
0	4'b0000	4'b0000
1	4'b0001	4'b0001
2	4'b0010	4'b0011
3	4'b0011	4'b0010
4	4'b0100	4'b0110
5	4'b0101	4'b0111
6	4'b0110	4'b0101
7	4'b0111	4'b0100
8	4'b1000	4'b1100
9	4'b1001	4'b1101
10	4'b1010	4'b1111
11	4'b1011	4'b1110
12	4'b1100	4'b1010
13	4'b1101	4'b1011
14	4'b1110	4'b1001
15	4'b1111	4'b1000

```
assign gray[0] = binary[1] ^ binary[0];
assign gray[1] = binary[2] ^ binary[1];
assign gray[2] = binary[3] ^ binary[2];
assign gray[3] = binary[3];
```

```
assign binary[0] = gray[3] ^ gray[2] ^ gray[1] ^ gray[0];
assign binary[1] = gray[3] ^ gray[2] ^ gray[1];
assign binary[2] = gray[3] ^ gray[2];
assign binary[3] = gray[3];
```

- Gray code toggles only one bit at a time
- Safely transfer the data through the synchronizer
- Gray code exists only for 2^N total counts (code is N-bit)
- Thus, FIFO depth should be 2^N

Gray Counter



- Binary-> Gray converter is embedded in the counter

https://www.verilogpro.com/asynchronous-fifo-design/

FIFO Depth Calculation

fifo_depth = B – B * $freq_{rd}$ / $freq_{wr}$

here, B: data packet size, $freq_{wr}$: data writing rate, $freq_{rd}$: data reading rate

Proof

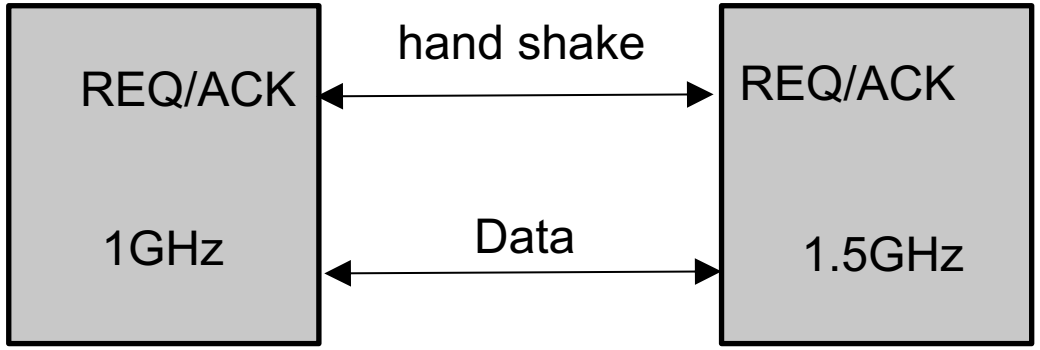
- Time to fill the B data with $freq_{wr}$ = B / $freq_{wr}$
- Number of data read meanwhile = $freq_{rd}$ * (B / $freq_{wr}$)
- Thus, the data not read should be buffered
- Thus, B – B * $freq_{rd}$ / $freq_{wr}$

Example

- $freq_{wr}$: 30MHz
- $freq_{rd}$: 20MHz
- Writing Burst Size =10

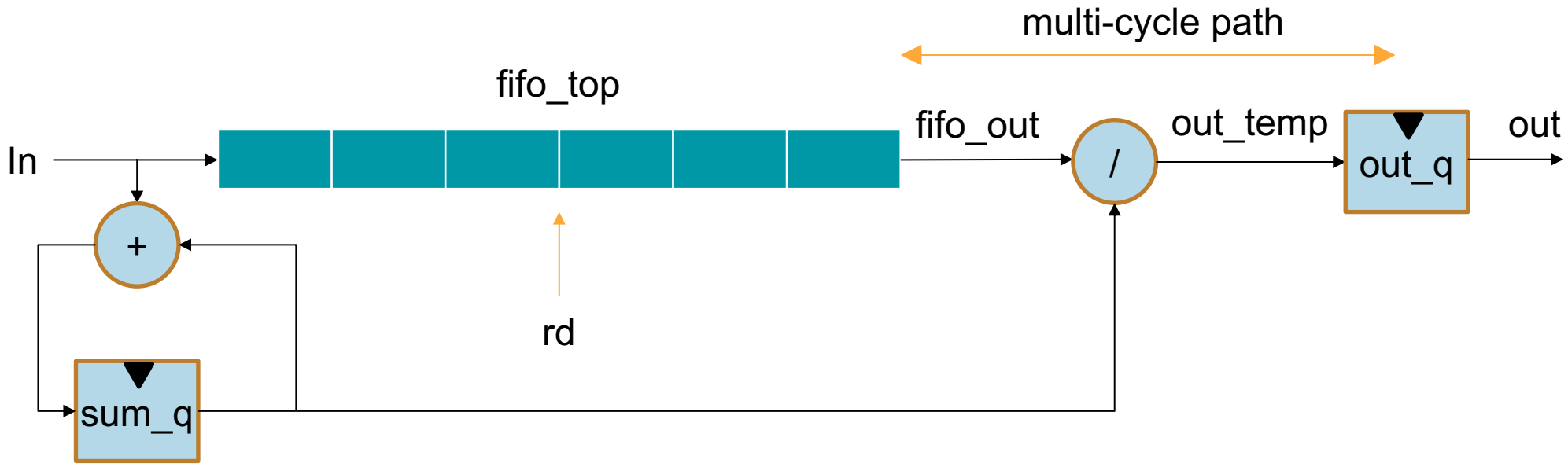
- Then, buffer size = 10 – 10 (20/30)
  = 3.33
  Thus, 4

HW1: Bi-direction Async. Interface with Hand-shaking



- **Asynchronous interface verilog design**
  - Build bi-directional async. interface with verilog, i.e., both sides can be either receiver / transmitter
  - Note new request can start when there is no on-going transaction
  - e.g., you need to build a simple state machine to issue req and ack signals accordingly
  - You also need to equip the synchronizer for the req / ack for both directions.
  - Frequency is 1 GHz & 1.5 GHz
  - Your data is “data.txt” from HW
  - Build test bench to store initial “data.txt” in a big register inside test bench (This is not a part of your DUT, but just for convenience.)
  - Then, send from left to right all the contents in “data.txt” file. Then, send all the contents back to from right to left
  - Compare your received data at the left with the initial data in a big register in the testbench.

HW2: Normalization (FIFO + Multi-cycle Path)



- **Normalization**
  - Assume your inputs are unsigned numbers
  - While writing you get the sum of the elements (b_data.txt)
  - (fifo_out / sum_q)*256 is given to out_q register
  - Design your normalizer by modifying the given Verilog code
  - Synthesize with 1GHz + 0.2ns IO delay + Typ corner
  - Apply multi-cycle path until you meet the 0 WNS from the synthesized result

Hint) remove all the inherited parameters
e.g., #(.bw(bw))