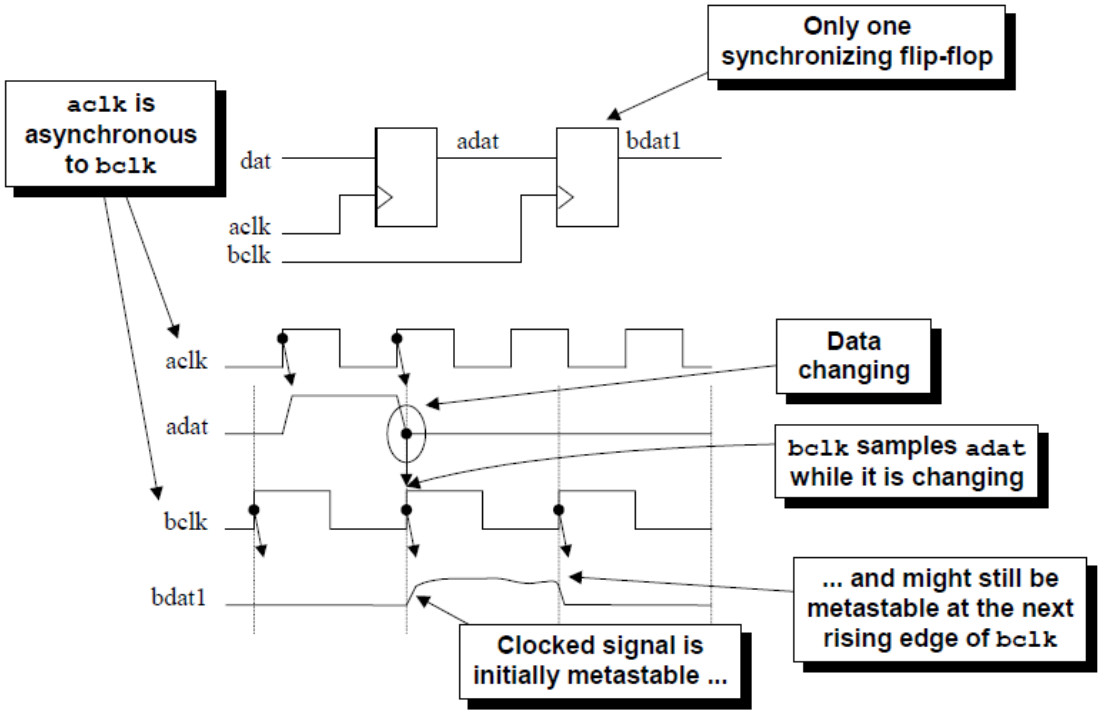
Clock Domain Crossing & Asynchronous FIFO Basics

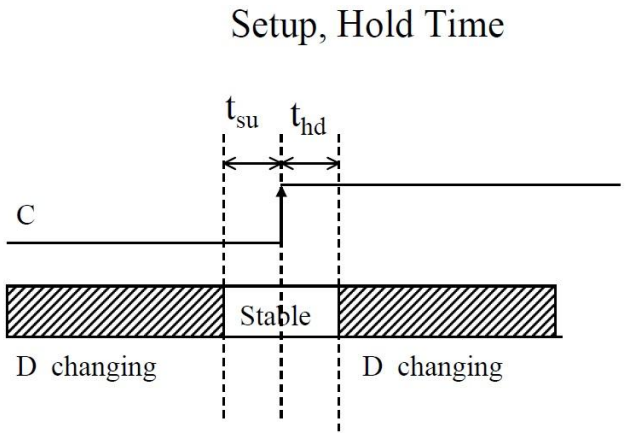
Siba K Panda

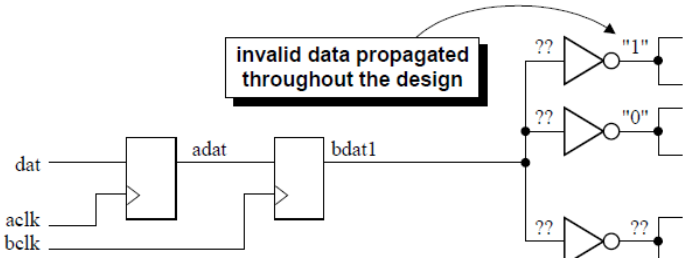
Introduction

- Why we need Asynchronous FIFO?
  - To safely pass data (multi-bit) from one clock domain to another clock domain.
  - Basically it's a Synchronization mechanism.
- What will happen if we don't use any kind of Synchronization Techniques
  - **Metastability**
    - Output of the receiving flip-flop goes to metastable state.
    - Which causes propagation of invalid data throughout the design.
    - Synchronization Failure !!

First – We will try to pass single bit signal !!



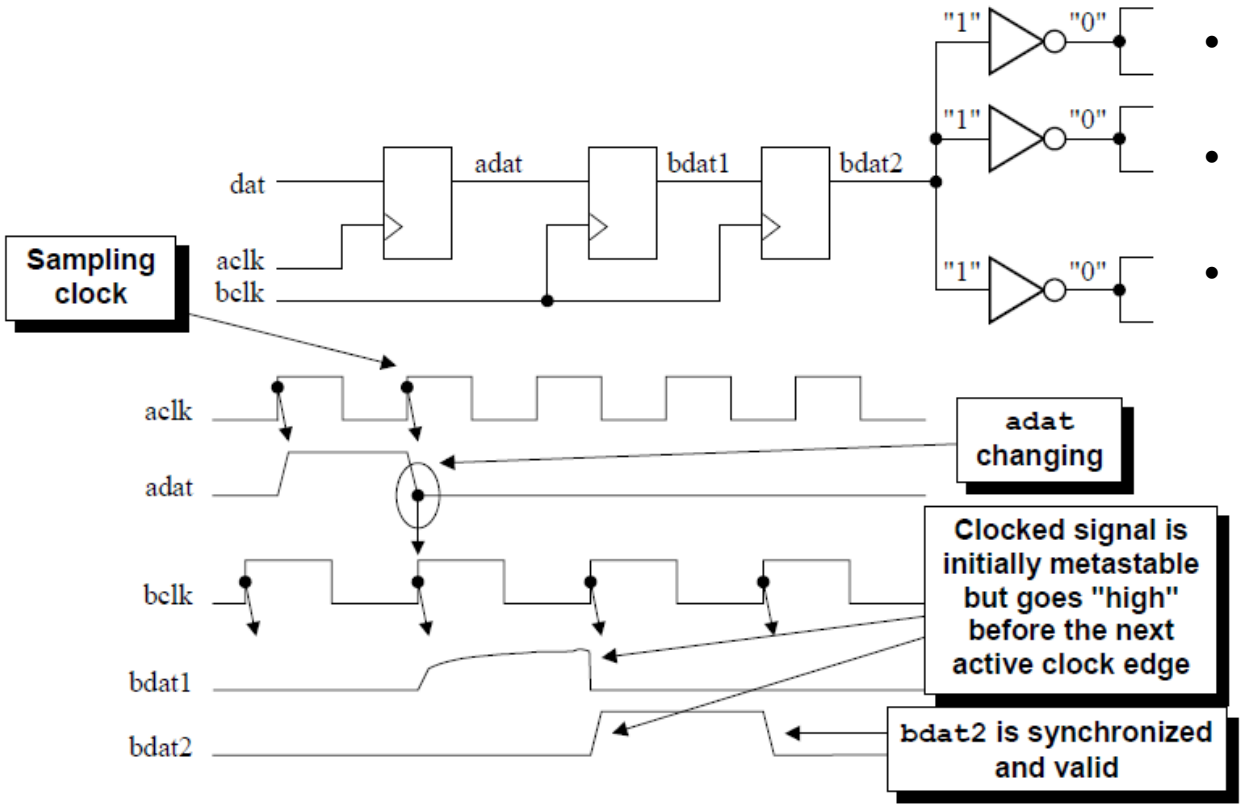




- Synchronization failure => occurs when a signal generated in one clock domain is sampled too close to the rising edge of a clock signal from another clock domain.

Solution - Synchronizers

- Two flip-flop synchronizer



- 1st FF will go to metastable state.
- 2nd FF will stabilize the data
- **But, it depends on how fast the data is changing**
  - **If data is changing to fast we might have to consider adding 1 more FF.**

MTBF (Mean Time Before Failure)

- Figure of merit for a FF related to metasabilty.
- MTBF – Basic Idea !!
  - Inversely proportional
    - Frequency of receiving clock domain
    - The rate of change of data

$$MTBF = \frac{1}{f\ receiving\ clock\ *\ f\ data}$$

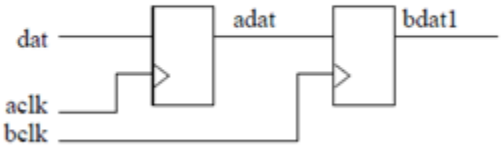
$$MTBF(t_r) = \frac{e^{(t_r/\tau)}}{T_o fa}$$

t_r resolution time (time since clock edge)
f sampling clock frequency
a asynchronous event frequency
τ and T_o FF parameters

For a typical .25um ASIC library FF
t_r = 2.3ns
τ = 0.31ns
T_o = 9.6as

For f = 100MHz, a = 1MHz

MTBF = 20.1 days

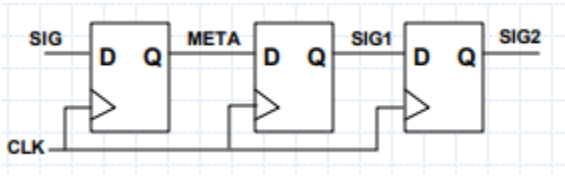


$$MTBF(t_r) = \frac{e^{(t_r/\tau)}}{T_o fa} \times \frac{e^{(t_r/\tau)}}{T_o f}$$

For a typical .25um ASIC library FF
t_r = 2.3ns
τ = 0.31ns
T_o = 9.6as

For f = 100MHz, a = 1MHz

MTBF = 9.57×10^{10} years
Age of Earth = 5×10^{9} years



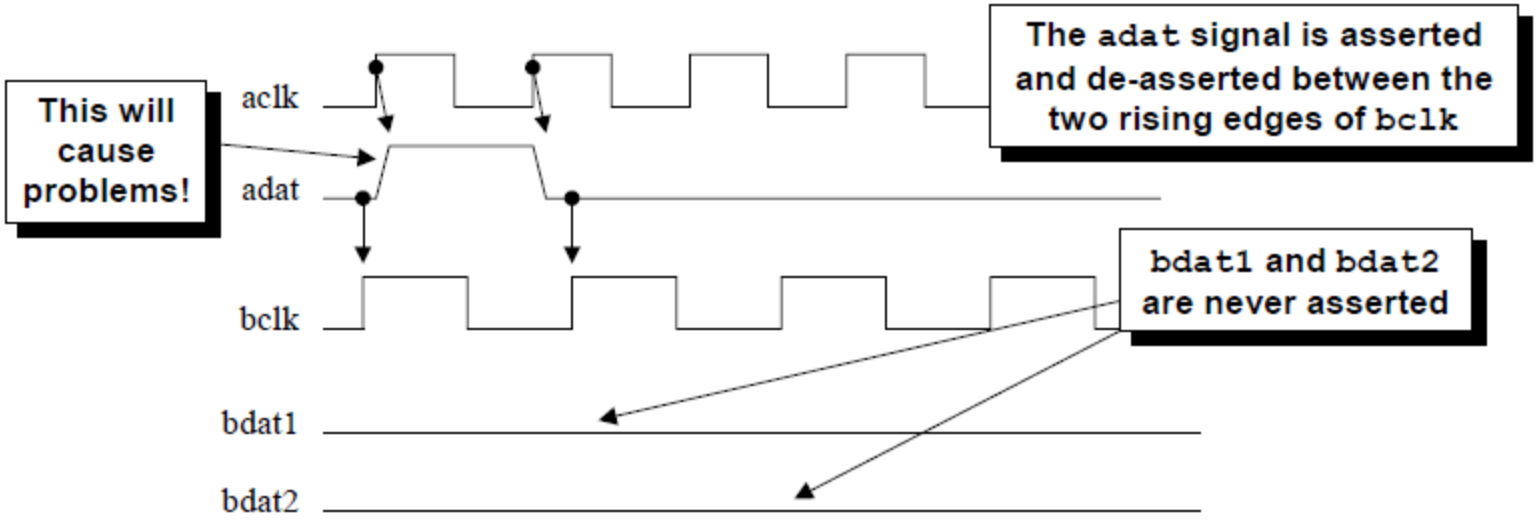
Problem Scenario

- Passing a Signal (1 bit) – Using 2 FF Synchronizers

From	To	Issues !!
Slower Clock	**Faster Clock**	**Not a problem** The faster clock signal will sample the slower signal one or more times
Faster Clock	**Slower Clock**	**There are some exceptions** • Solution 1 – Signal must be wider than the cycle time of the slower clock • Solution 2 – Feedback

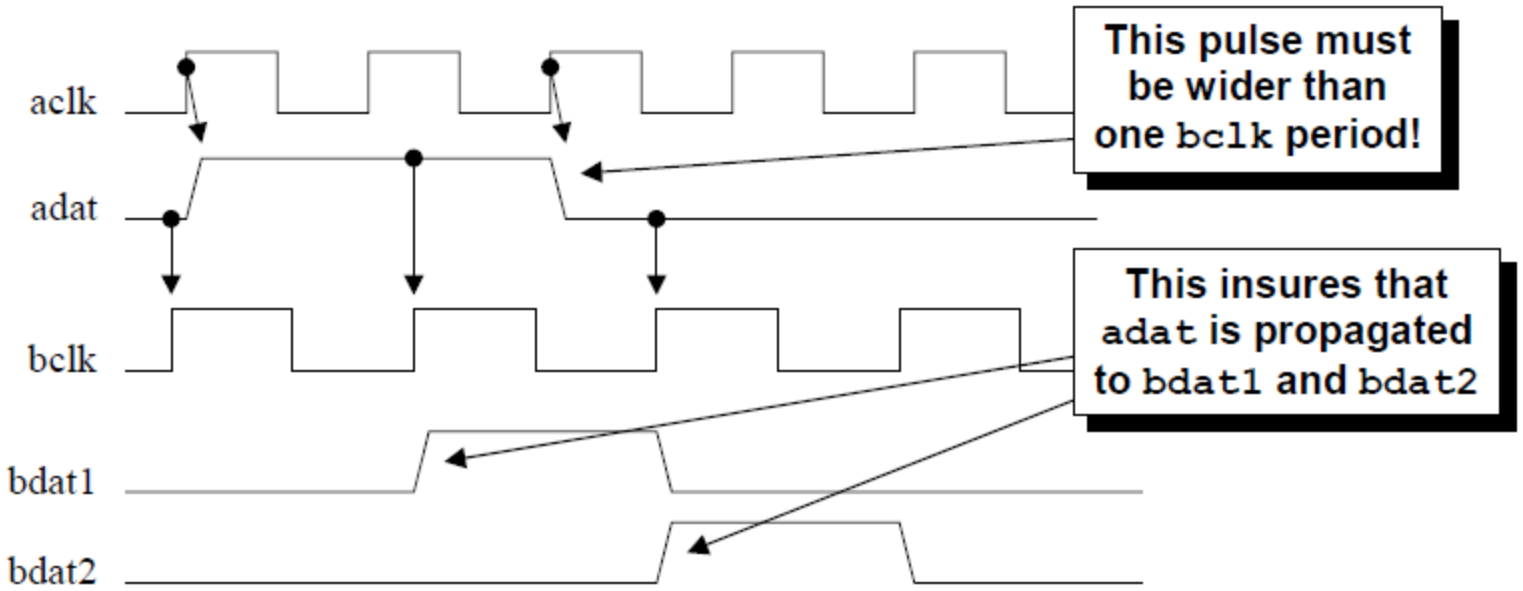
Passing a Signal from Faster clock to Slower Clock

- What is the problem!!
  - Signal will not be captured into the slower clock domain



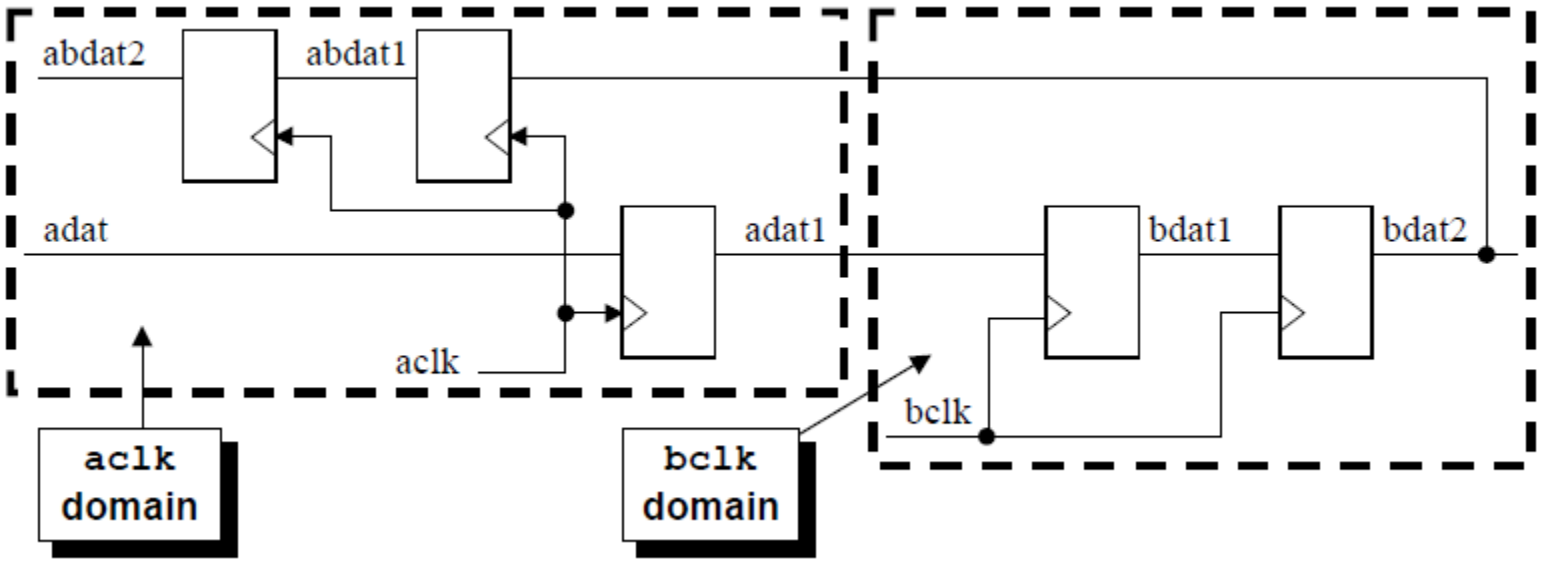
Solution 1 – Open loop Solution

- To assert the signal for a period of time that exceeds the time period of the sampling clock (1.5 x Time period of the receiving clock).
  - The assumption is that the control signal will be sampled at least once by the receiver clock.
  - Frequency of the receiving clock and pulse width of the signal must be know.



Solution 2 – Closed loop Solution (Feedback loop)

- To assert a signal, synchronize it into the new clock domain and then pass the synchronized signal back through another synchronizer into the sending clock domain as an acknowledge signal.
- Then, put some logic to check the data is properly captured.
  - Safe Technique.
  - But, there is considerable delay associated with synchronizing the signal in both directions before releasing it.



Passing Multiple Control Signals

- The importance of the sequencing of the control signals.
  - If the order or alignment of the control signals is significant, care must be taken to correctly pass the signals into the new clock domain.

- Problem 1 - Two simultaneously required control signals.
- Problem 2 - Two phase-shifted sequencing control signals.
- Problem 3 - Two encoded control signals.

Data-Path Synchronization

- **Passing data from one clock domain to another.**
  - Using synchronizers to handle the passing of data is generally unacceptable.
  - Why ??
    - High probability for Multi-bit data changes to be incorrectly sampled using synchronizers.
- **Solution**
  - **Solution 1 :** Use handshake signals to pass data between clock domains.
  - **Solution 2 :** Asynchronous FIFO - store data using one clock domain and to retrieve data using another clock domain.

Handshaking Data Between Clock Domains

- The sender places data onto a data bus and then synchronizes a "data_valid" signal to the receiving clock domain.
- When the "data_valid" signal is recognized the receiver clocks the data into a register
- Receiver passes an "acknowledge" signal through a synchronizer to the sender.
- When the sender recognizes the synchronized "acknowledge" signal, the sender can change the value being driven onto the data bus.
- Third control signal, "ready” - to indicate that the receiver is indeed "ready" to receive data.
- **Disadvantage - The latency required to pass and recognize all of the handshaking signals for each data word**

Passing Data By FIFO Between Clock Domains

- Most popular methods of passing data between clock domains.
- **Asynchronous FIFO – Basic Idea !!**
  - Data are written to a FIFO buffer from one clock domain.
  - Data are read from the same FIFO buffer from another clock domain.
- **Asynchronous FIFO Design – Deals With !!**
  - Generating FIFO Pointers
  - Determining Full and Empty Status

Synchronous FIFO

- FIFO where writes to, and reads from the FIFO buffer are conducted in the same clock domain.
- **Implementation**
  - **Using a single counter**

    Counting the number of writes to, and reads from the FIFO buffer.
    - **Increment** - on FIFO write but no read
    - **Decrement** - on FIFO read but no write
    - **Hold** - no writes and reads
    - **Empty** - FIFO counter = 0
    - **Full** - FIFO counter = depth of the FIFO

Synchronous FIFO

- **Using Write Pointer and Read Pointer**
  - Write Pointer
    - Points to the next word to be written
    - on reset - set to zero
      - Which also happens to be the next FIFO word location to be written.
    - on FIFO- write operation
      - The memory location that is pointed to by the write pointer is written.
      - Then, the write pointer is incremented to point to the next location to be written

Synchronous FIFO

- **Using Write Pointer and Read Pointer**
  - Read Pointer
    - Points to the current FIFO word to be read
    - on reset - set to zero
      - FIFO is empty
      - The read pointer is pointing to invalid data
    - If the write pointer increments .
      - The empty flag is cleared
      - The read pointer that is still addressing the contents of the first FIFO memory word
      - Immediately , drives the that first valid word onto the FIFO data output port

Synchronous FIFO

- **Using Write Pointer and Read Pointer**

  Basically two counters running at same clock

  - **Increment (Write Pointer) -** on FIFO write but no read
  - **Increment (Read Pointer) -** on FIFO read but no write
  - **Hold (Both) -** no writes and reads
  - **Empty -** Write Pointer **-** Read Pointer == 0
  - **Full -** Same ??
    - Technique used to distinguish between full and empty
    - **To add an extra bit to each pointer**

Technique used to distinguish between full and empty

- **Add an extra bit to each pointer.**
- When the **Write pointer** increments past the final FIFO address –
  - The write pointer will increment the unused MSB .
  - i.e. **FIFO has wrapped and toggled the pointer MSB**
- Same as for **Read pointer**
- If the MSBs of the two pointers are different –
  - It means that the write pointer has wrapped one more time that the read pointer
  - **Full Condition**.
- If the MSBs of the two pointers are the same
  - It means that both pointers have wrapped the same number of times
  - **Empty Condition**

EMPTY

```
if (waddr == raddr);
```

FULL

```
if ({~waddr[4],waddr[3:0]} == raddr);
```

Asynchronous FIFO Pointers

- The increment-decrement FIFO fill counter cannot be used.
- Two different and asynchronous clocks (wclk & rclk) would be required to control write pointer and read pointer.
- To determine full and empty status, the write and read pointers will have to be compared.

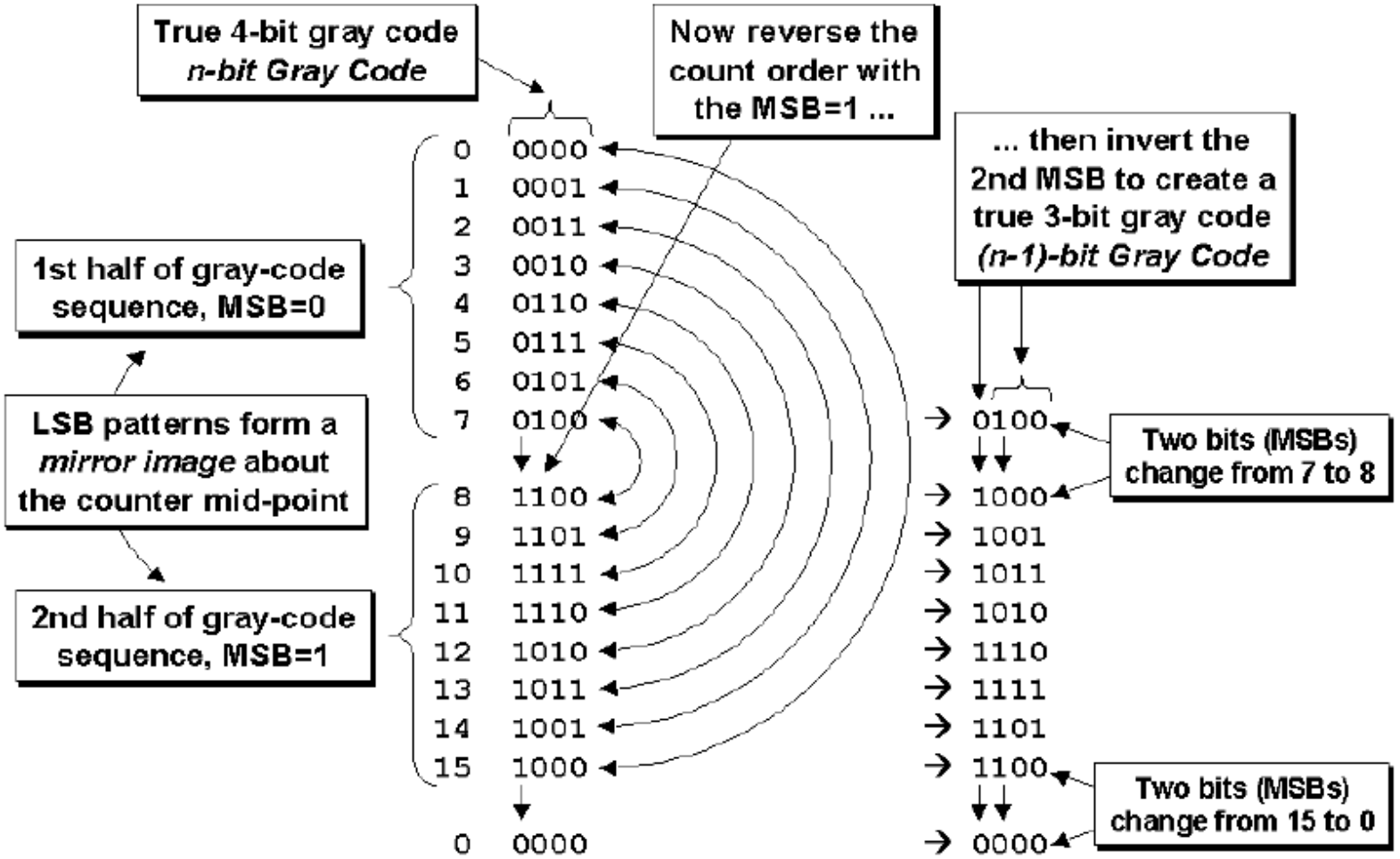
- **As ,Write pointer and Read pointer**
  - Uses n bit.
  - Then,(n-1) is the number of address bits required to access the entire FIFO memory buffer.
  - Therefore ,The FIFO design uses n-bit pointers for a FIFO with 2 power (n-1) write-able locations (depth).

Problem with Binary FIFO pointer

- **Trying to synchronize a binary count value from one clock domain to another is problematic.**
  - Because, every bit of an n-bit counter can change simultaneously
  - 7 -> 8 in binary numbers is 0111->1000, all bits changed.
  - The synchronized and sampled write & read pointer might not reflect the current value of the actual write & read pointer.
- **Solution - Gray code counters**
  - Gray codes only allow one bit to change for each clock transition,
  - eliminating the problem associated with trying to synchronize multiple changing signals on the same clock edge.

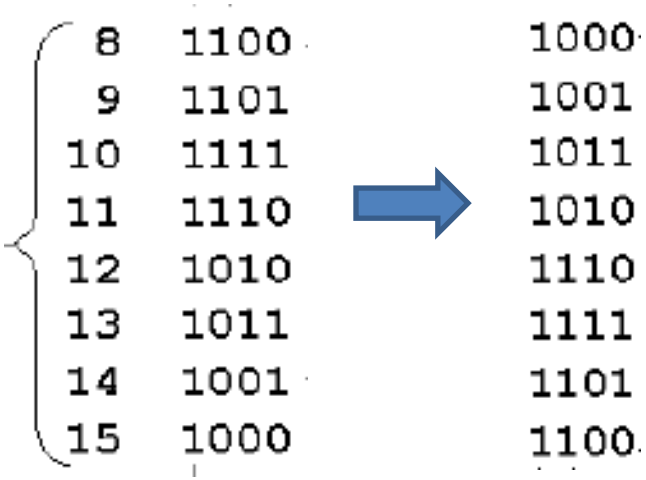
Gray code counter

- Gray code – Unit distance code !!
- The problem of converting an n-bit Gray code to an (n-1)-bit Gray code.



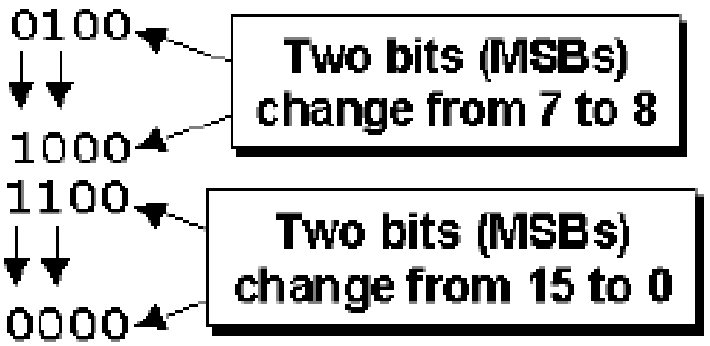
Problem of converting an n-bit Gray code to an (n-1)-bit Gray code.

- **Observe :**
  - The second half of the 4-bit Gray code is a mirror image of the first half with the MSB inverted.
  - To convert a 4-bit to a 3-bit Gray code - we want the LSBs of the second half to repeat the 4-bit LSBs of the first half.



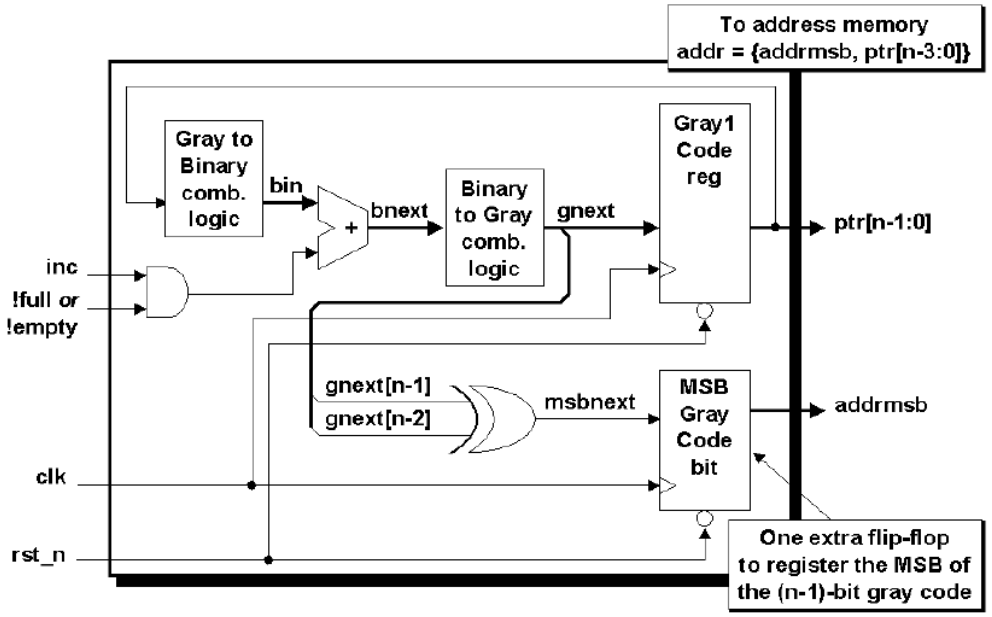
But, This creates another problem!!

- Two bits are changing !!
- A true Gray code only changes one bit between counts.



Solution - Dual n-bit Gray code counter Style 1

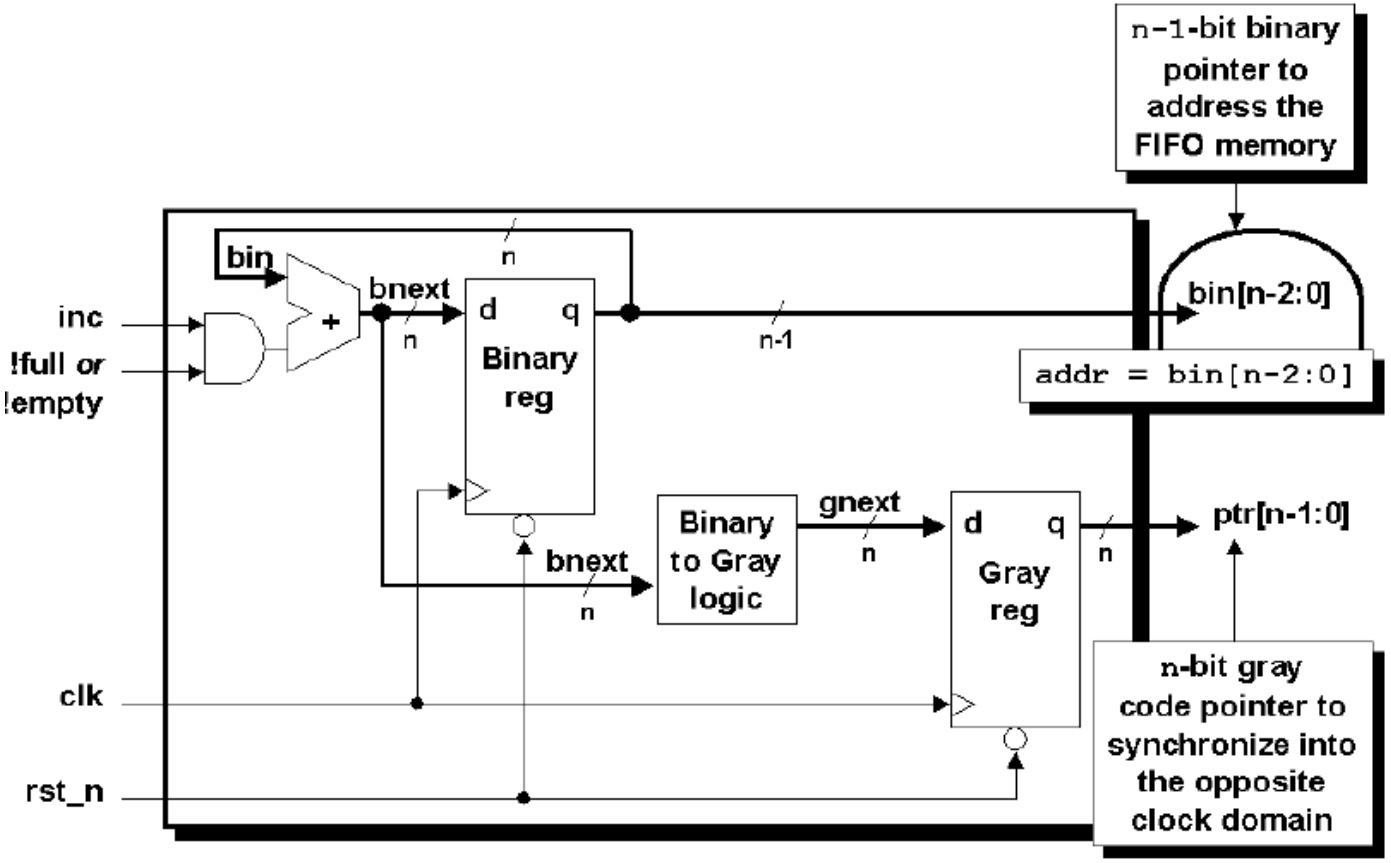
- Create both an n-bit Gray code count and an (n-1)-bit Gray code count.
  - n-bit Gray code sequence - Write / Read pointer.
  - (n-1)-bit Gray code sequence – to address memory.
- Dual n-bit Gray code counter – Style 1



Solution - Dual n-bit Gray code counter Style 1

- Dual n-bit Gray code counter – Style 1
  - Generates both an n-bit Gray code sequence and an (n-1)-bit Gray code sequence.
  - The (n-1)-bit Gray code is simply generated by doing an exclusive-or operation on the two MSBs of the n-bit Gray code to generate the MSB for the (n-1)-bit Gray code.
  - This is combined with the (n-2) LSBs of the n-bit Gray code counter to form the (n-1)-bit Gray code counter
- Drawback - (n-1)-bit Gray code sequence (to address memory) to binary values.

Solution - Dual n-bit Gray code counter Style 2



Solution - Dual n-bit Gray code counter Style 2

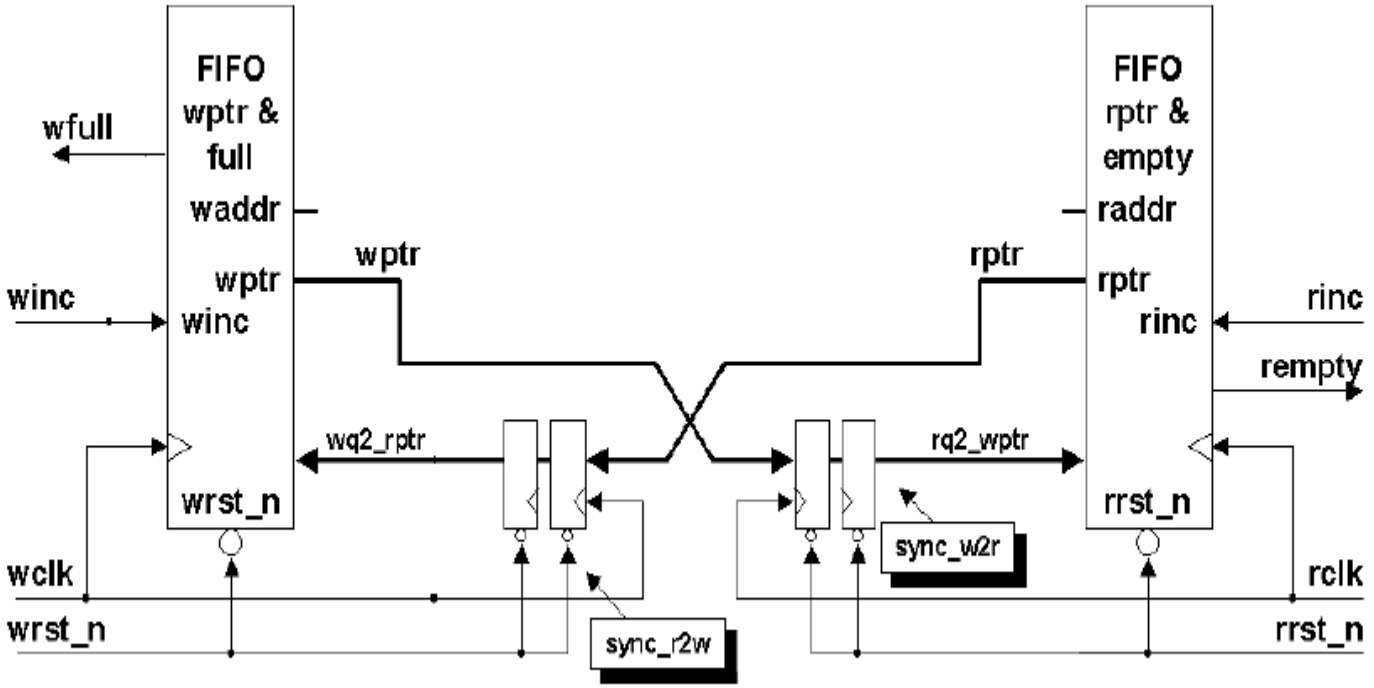
- Employs two sets of registers to eliminate the need to translate Gray pointer values to binary values.
- The second set of registers (the binary registers) can also be used to address the FIFO memory directly without the need to translate memory addresses into Gray codes.

Handling full & empty conditions

- **Empty flag** will be generated in the **read-clock domain**
  - To insure that the **empty flag is detected immediately** when the **FIFO buffer is empty**.
  - i.e. the instant that the **read pointer catches up to the write pointer** (including the pointer MSBs).
- **Full flag** will be generated in the **write-clock domain**
  - To insure that the **full flag is detected immediately** when **the FIFO buffer is full.**
  - i.e. the instant that the **write pointer catches up to the read pointer** (except for different pointer MSBs).

Synchronized pointer comparison

- Write pointer and read pointer must be synchronized into opposite clock domain.



Generating empty

- The FIFO is empty when the read pointer and the *synchronized* write pointer are equal.
  - If the extra bits of both pointers (the MSBs of the pointers) are equal, the pointers have wrapped the same number of times and if the rest of the read pointer equals the synchronized write pointer, the FIFO is empty.
- The Gray code write pointer must be synchronized into the read-clock domain through a pair of synchronizer registers.
- In order to efficiently register the **rempty** output, the synchronized write pointer is actually compared against the **rgraynext** (the next Gray code that will be registered into the **rptr**).
- Example

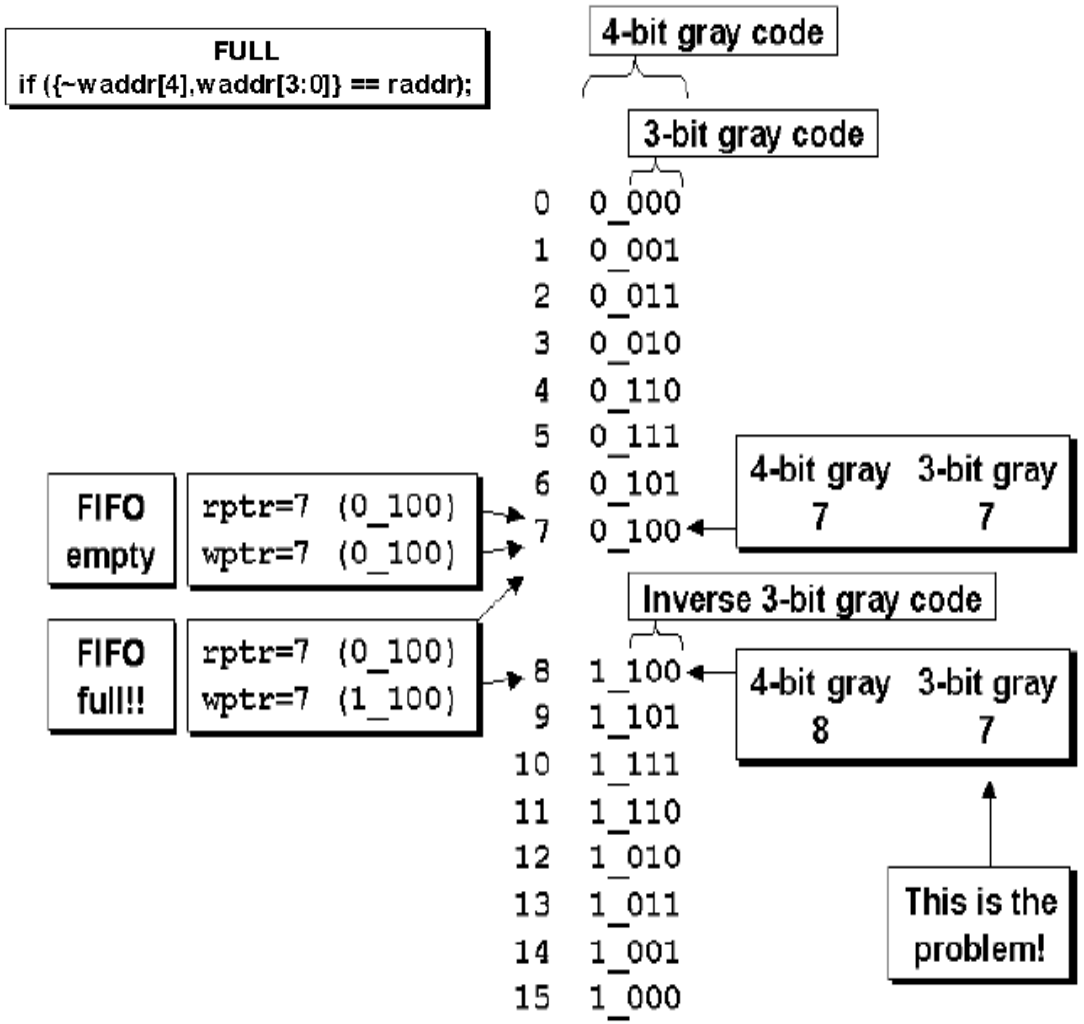
```
assign rempty_val = (rgraynext == rq2_wptr);
always @(posedge rclk or negedge rrst_n)
  if (!rrst_n) rempty <= 1'b1;
 else rempty <= rempty_val;
```

Generating full

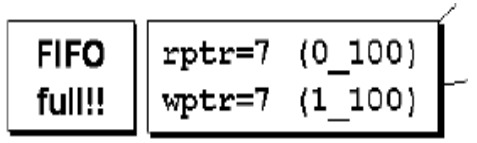
- **Full flag** is generated in the **write-clock domain.**
  - By, running a comparison between the write and read pointers.
  - The read pointer be synchronized into the write clock domain before doing pointer comparison.

- **Full comparison has issues !!**
  - Directly, using Gray code counters with an extra bit to do the comparison is not valid to determine the full condition.
  - The problem is that a Gray code is a symmetric code except for the MSBs

Problems associated with extracting a 3-bit Gray code from a 4-bit Gray code



- Example – 8 bit deep FIFO
- FIFO is allowed to fill seven bit locations(0-6)
- Then, FIFO is emptied.
- Now, both pointers = gray 7
- FIFO empty
- But, the on the next write operation.
- Write pointer will increment
- Making -



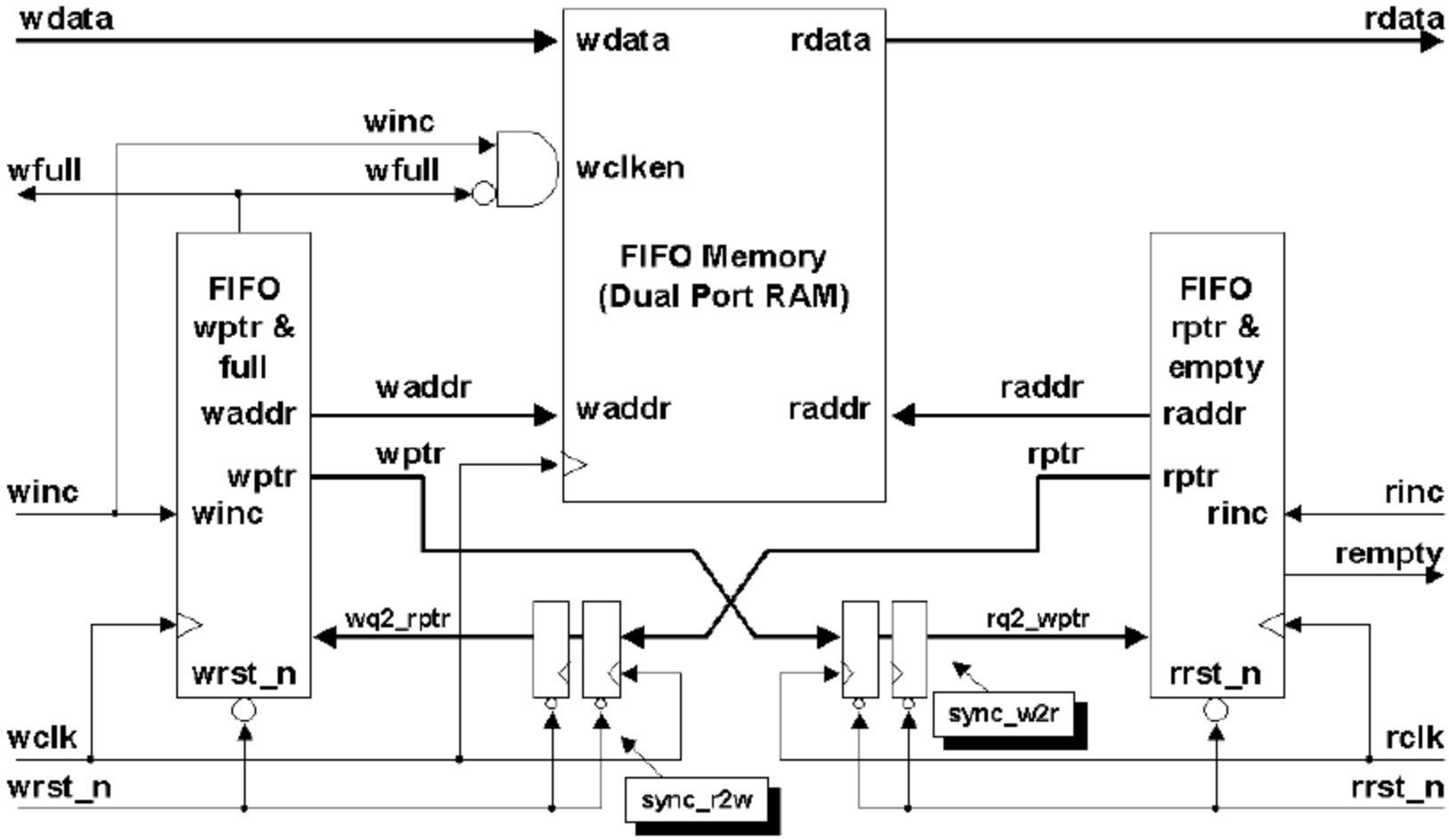
- Which is Wrong !!
- As, FIFO is not actually full
- And 3 LSB's didn't change
- Which overwrite the same location twice

Generating full – Correct Method

- Three conditions that are all necessary for the FIFO to be full
  1. The **write pointer** and the synchronized **read pointer** MSB's must be not equal
  2. The **write pointer** and the synchronized **read pointer** 2nd MSB's must not equal
  3. All other **write pointer** and synchronized **read pointer** bits must be equal.
- Similarly, In order to efficiently register the **wfull** output, the synchronized read pointer is actually compared against the **wgnext**
- Example

```
assign wfull_val = ((wgnext[ADDRSIZE] !=wq2_rptr[ADDRSIZE] ) &&
(wgnext[ADDRSIZE-1] !=wq2_rptr[ADDRSIZE-1]) &&
(wgnext[ADDRSIZE-2:0]==wq2_rptr[ADDRSIZE-2:0]));
```

Asynchronous FIFO Block Diagram



FIFO memory buffer

- Dual-port, synchronous memory
- Synchronous Write
- Asynchronous Read

Read pointer & empty generation logic

- The read pointer is a dual n-bit Gray code counter.
- The n-bit pointer (**rptr**) is passed to the write clock domain through the **sync_r2w** module
- The (n-1)-bit pointer (**raddr**) is used to address the FIFO buffer.
- The FIFO empty output is registered and is asserted on the next rising **rclk** edge when the next **rptr** value equals the synchronized **wptr** value.
- This module is entirely synchronous to the **rclk**

Write pointer & full generation logic

- The write pointer is a dual n-bit Gray code counter.
- The n-bit pointer (**wptr**) is passed to the read clock domain through the **sync_w2r** module.
- The (n-1)-bit pointer (**waddr**) is used to address the FIFO buffer.
- The FIFO full output is registered and is asserted on the next rising **wclk** edge when the next modified **wgnext** value equals the synchronized and modified **wrptr2** value (except 1st MSBs and 2nd MSBs).
- This module is entirely synchronous to the **wclk**

Almost full and almost empty –Different Ways!!

- When the difference between the pointers is smaller than the programmed difference, the corresponding almost full or almost empty bit is asserted.
- Can also be implemented using fixed difference.
- When the MSBs of the FIFO pointers are close.
- The almost full condition could be described as the condition when (**write pointer**+4) catches up to the synchronized **read pointer.**

Issues with FIFO design

- Full and Empty removal is pessimistic(not accurate).
  - “full” and “empty” are both asserted exactly on time but removed late.
- Setting either the full flag or empty flag might not be quite accurate if both pointers are incrementing simultaneously.

Reference

- Clifford E. Cummings, “Simulation and Synthesis Techniques for Asynchronous FIFO Design,” *SNUG 2002 (Synopsys Users Group Conference, San Jose, CA, 2002) User Papers*, March 2002.
- Clifford E. Cummings, “Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs,” *SNUG 2001 (Synopsys Users Group Conference, San Jose, CA, 2001) User Papers*, March 2001.