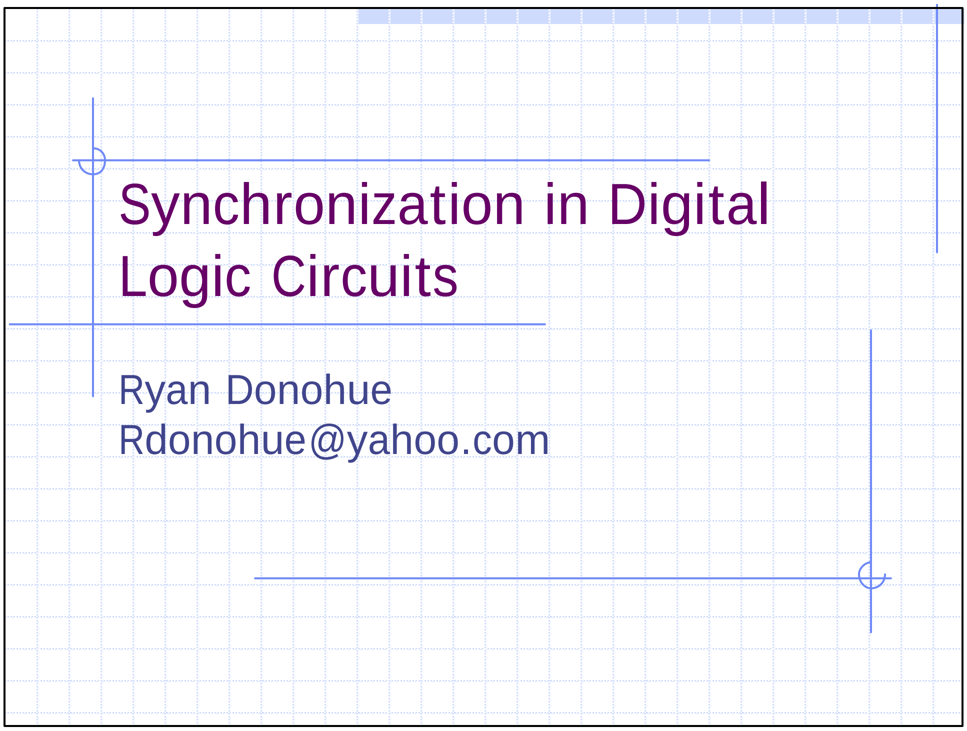
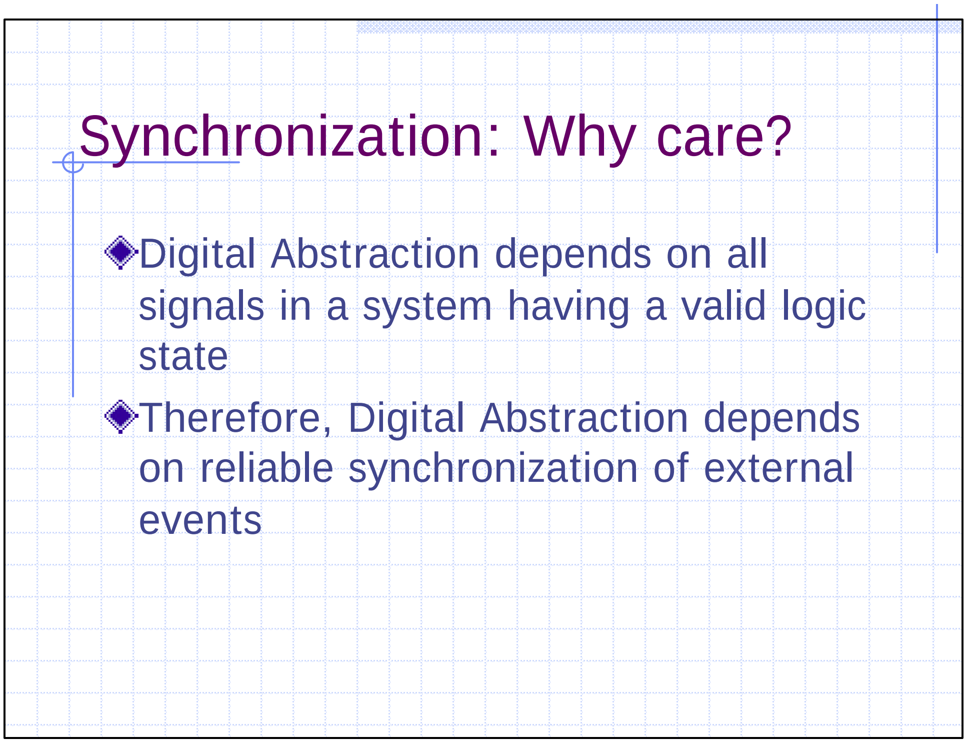


Synchronization in Digital Logic Circuits

Ryan Donohue
Rdonohue@yahoo.com

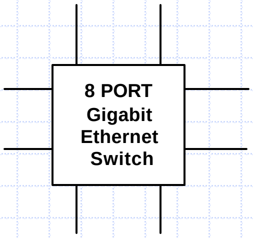


Synchronization: Why care?

- Digital Abstraction depends on all signals in a system having a valid logic state
- Therefore, Digital Abstraction depends on reliable synchronization of external events

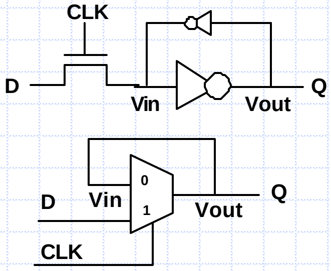
The Real World

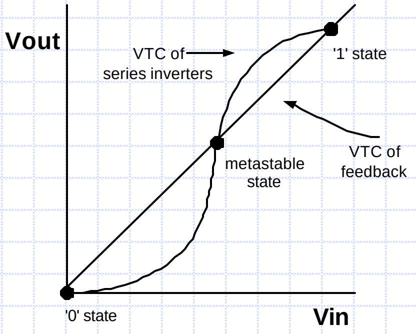
- Real World does not respect the Digital Abstraction!
  - Inputs from the Real World are usually asynchronous to your system clock
  - Inputs that come from other synchronous systems are based on a different system clock, which is typically asynchronous to your system clock

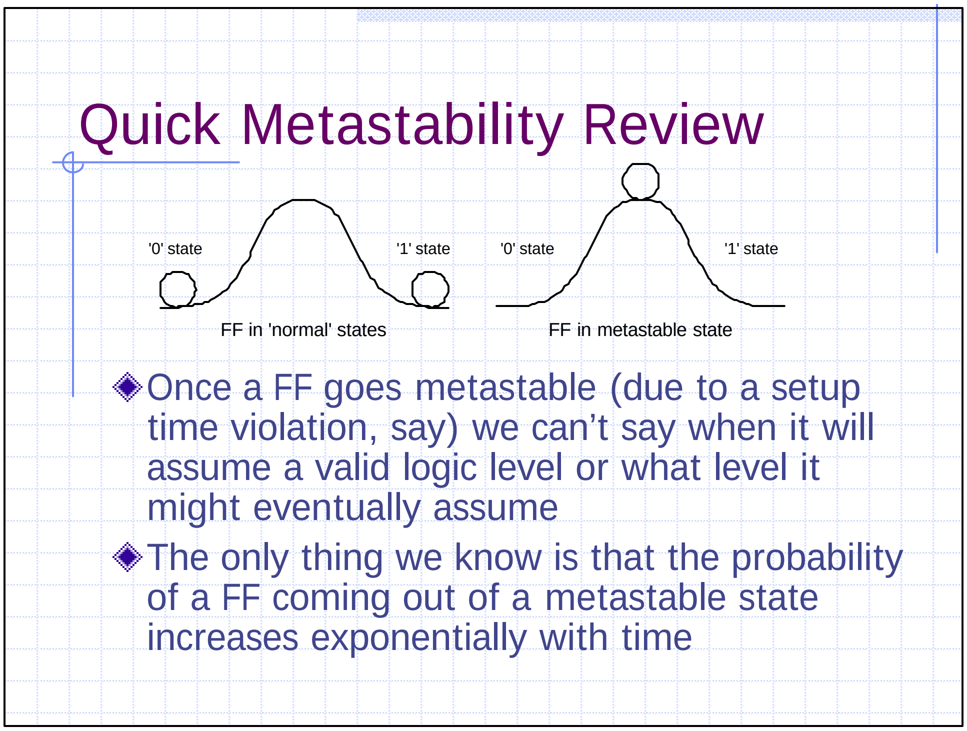


Metastability

- When asynchronous events enter your synchronous system, they can cause bistables to go into metastable states
- Every real life bistable (such as a D-latch) has a metastable state



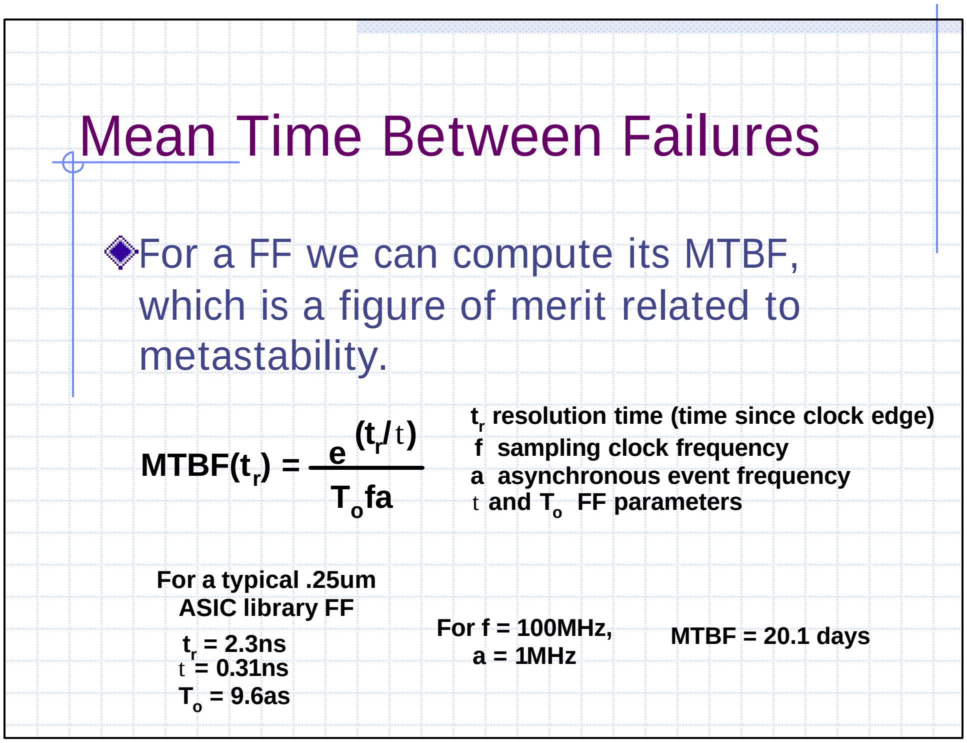




Quick Metastability Review

'0' state
'1' state
FF in 'normal' states
'0' state
'1' state
FF in metastable state

- Once a FF goes metastable (due to a setup time violation, say) we can't say when it will assume a valid logic level or what level it might eventually assume
- The only thing we know is that the probability of a FF coming out of a metastable state increases exponentially with time



Mean Time Between Failures

- For a FF we can compute its MTBF, which is a figure of merit related to metastability.

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

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

$t_r = 2.3\text{ns}$
$\tau = 0.31\text{ns}$
$T_o = 9.6\text{as}$

For f = 100MHz,
a = 1MHz

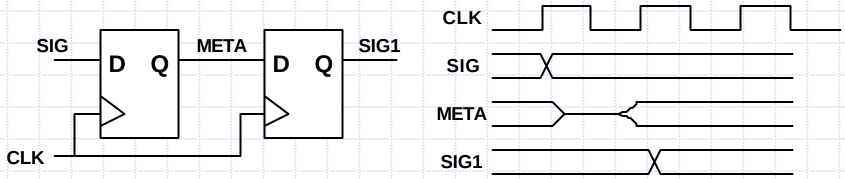
MTBF = 20.1 days

Synchronizer Requirements

- Synchronizers must be designed to reduce the chances system failure due to metastability
- Synchronizer requirements
  - Reliable [high MTBF]
  - Low latency [works as quickly as possible]
  - Low power/area impact

Single signal Synchronizer

- Traditional synchronizer
  - SIG is asynchronous, and META might go metastable from time to time
  - However, as long as META resolves before the next clock period SIG1 should have valid logic levels
  - Place FFs close together to allow maximum time for META to reslove



Single Synchronizer analysis

- MTBF of this system is roughly:

$$\text{MTBF}(t_r) = \frac{e^{(t_r/t)}}{T_o f a} \text{ x } \frac{e^{(t_r/t)}}{T_o f}$$

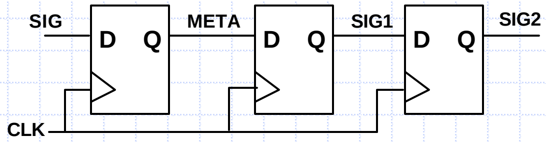
MTBF = $9.57 \text{x} 10^{10}$ years
Age of Earth = $5 \text{x} 10^{9}$ years

For a typical .25um
ASIC library FF

t_r = 2.3ns
t = 0.31ns
T_o = 9.6as

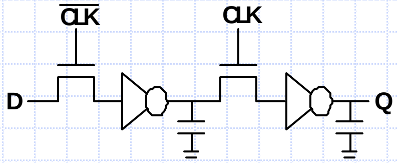
For f = 100MHz,
a = 1MHz

- Can increase MTBF by adding more series stages

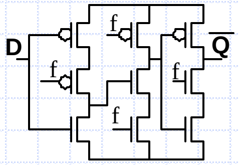


Flip Flop design is important?

- Dynamic FFs not suitable for synchronizers since they have no regeneration



CMOS Dynamic FF

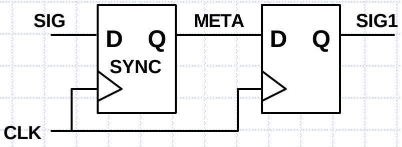


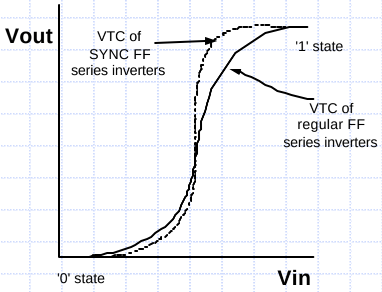
TSFF (Svenson)

- Special ‘SYNC’ FFs should be used for the primary synchronizer if available

SYNC Flip Flop

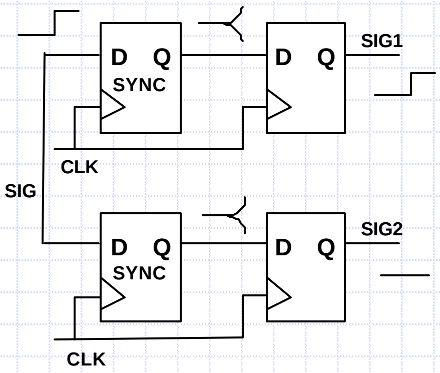
- SYNC Flip Flops are available in some ASIC libraries
  - Better MTBF characteristics due to high gain in the feedback path
  - Very large (5x regular FF) and very high power





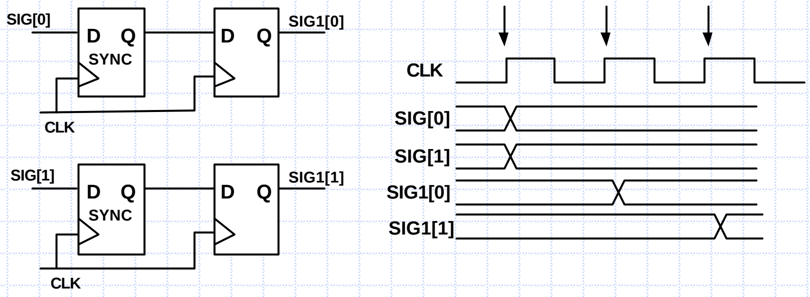
Synchronization Pitfall

- Never synchronize the same signal in multiple places! Inconsistency will result!



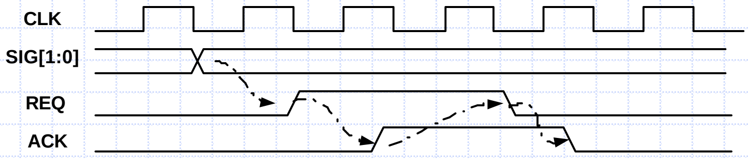
Bus Synchronization

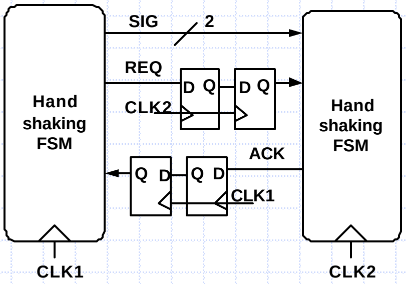
- Obvious approach is to use single signal synchronizers on each bit
- WRONG!



Handshaking is the Answer

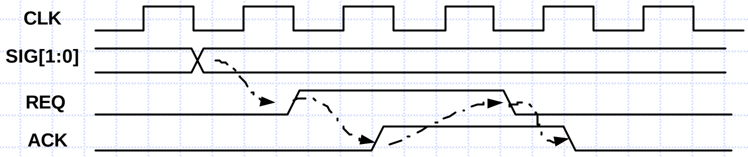
- Need a single point of synchronization for the entire bus





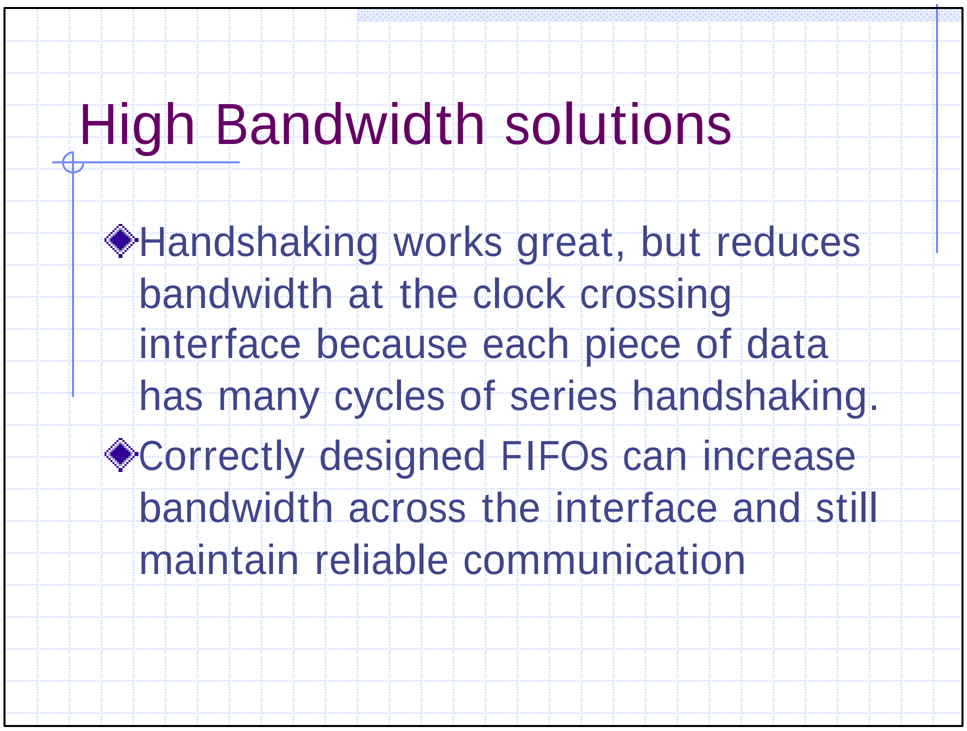
Handshaking Rules

- Sender outputs data and THEN asserts REQ
- Receiver latches data and THEN asserts ACK
- Sender deasserts REQ, will not reassert it until ACK deasserts
- Receiver sees REQ deasserted, deasserts ACK when ready to continue



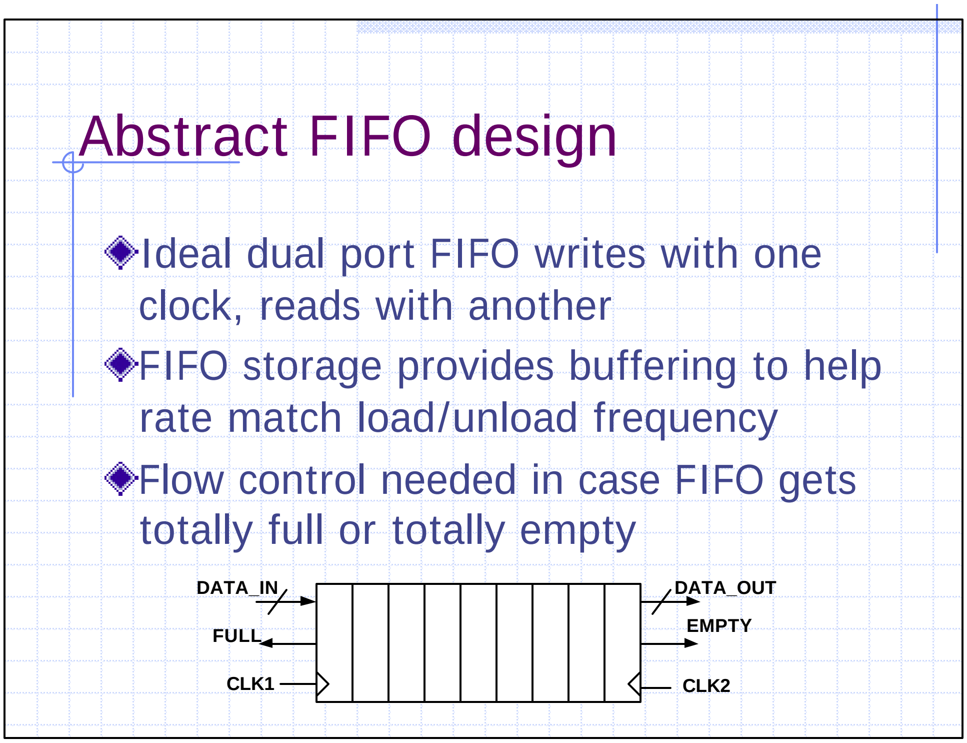
Alternate Handshaking Scheme

- Previous example is known as 4-phase handshaking
- 2-phase (or edge based) handshaking is also suitable
  - Sender outputs data and THEN changes state of REQ, will not change state of REQ again until after ACK changes state.
  - Receiver latches data. Once receiver is ready for more it changes state of ACK.
- 2-phase requires one bit of state be kept on each side of transaction. Used when FFs are inexpensive and reliable reset is available.



High Bandwidth solutions

- Handshaking works great, but reduces bandwidth at the clock crossing interface because each piece of data has many cycles of series handshaking.
- Correctly designed FIFOs can increase bandwidth across the interface and still maintain reliable communication



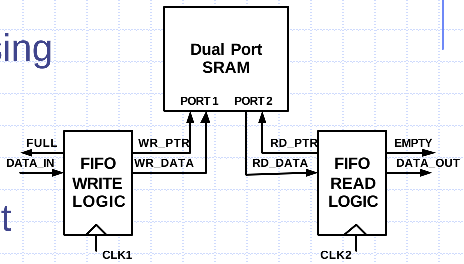
Abstract FIFO design

- Ideal dual port FIFO writes with one clock, reads with another
- FIFO storage provides buffering to help rate match load/unload frequency
- Flow control needed in case FIFO gets totally full or totally empty

DATA_IN
FULL
CLK1
DATA_OUT
EMPTY
CLK2

FIFO in detail

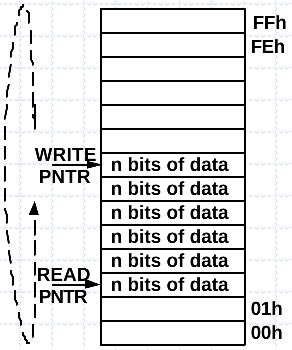
- FIFO of any significant size is implemented using an on-chip SRAM
- SRAM must be dual-ported for our design [have two independent ports]
- We will use a write pointer to determine the write address, and a read pointer to determine the read address



FIFO pointer control

- FIFO is managed as a circular buffer using pointers.
- First write will occur at address 00h. Next write will occur at 01h.

  After writing at FFh, next write will wrap to 00h.
- Reads work the same way. First read will occur at address 00h.



FIFO pointers and flow control

- Generation of FULL and EMPTY signals.
  - FIFO is FULL when write pointer catches read pointer

```
always @(posedge clk1)
FULL <= (WR_PNTR == RD_PNTR) && ((OLD_WR_PNTR + 1 == RD_PNTR) || FULL)
```

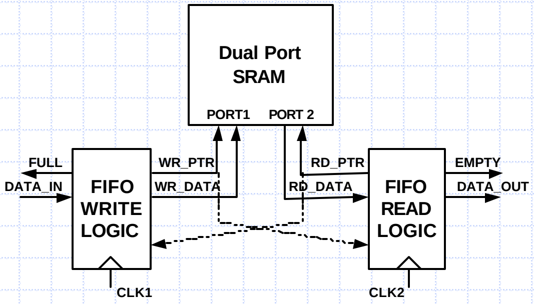
  - FIFO is empty when read pointer catches write pointer

```
always @(posedge clk2)
EMPTY <= (WR_PNTR == RD_PNTR) && ((OLD_RD_PNTR + 1 == WR_PNTR) || EMPTY)
```

- Write pointer and read pointer must never pass each other.
  - Write passing read overwrites unread data
  - Read passing write re-reads invalid data

FIFO in detail

- We have a problem!

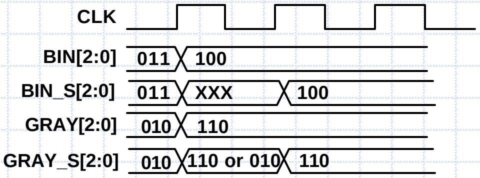


To generate FULL/EMPTY conditions the write logic needs to see the read pointer and the read logic needs to see the write pointer!

Pointer Synchronization

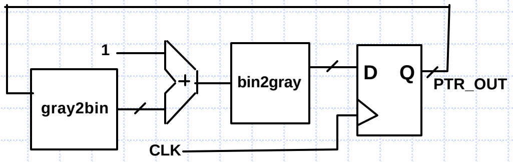
- Our pointers change in a very specific way (when they change, they increment by 1)
  - Applying a traditional two stage FF synchronizer on each bit of a binary pointer could cause a wildly invalid pointer value to be produced
  - Gray coding the pointer value means at most one bit will change per cycle – we can only be 'off by one'

Binary	Gray
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

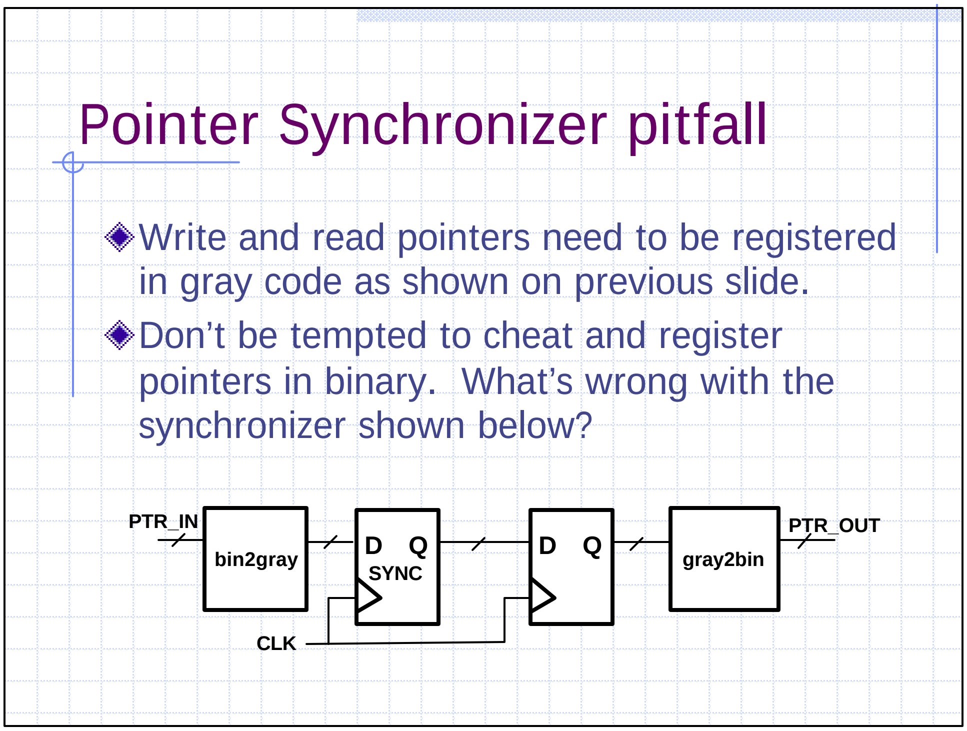


Pointer Synchronizer

- Pointer is stored in gray code.
- A standard single bit synchronizer is used on each bit of PTR_OUT. At most one bit changes per cycle!
- We can still do binary math to increment the pointer.

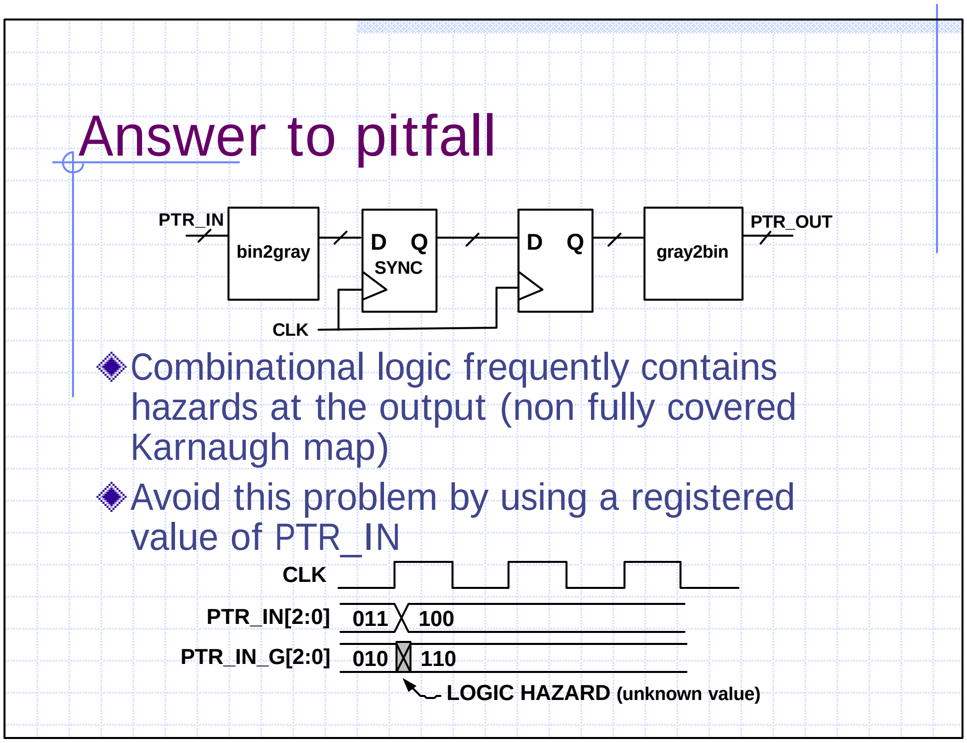


```
module bin2gray (bin,gray);

parameter SIZE = 4;

input [SIZE-1:0] bin;
output [SIZE-1:0] gray;
reg [SIZE-1:0] gray;

always @(bin)
      gray = (bin >> 1) ^ bin;

endmodule
module gray2bin (gray,bin);

parameter SIZE = 4;

input [SIZE-1:0] gray;
output [SIZE-1:0] bin;
reg [SIZE-1:0] bin;

integer i;

always @(gray)
for (i=0; i<SIZE; i=i+1)
      bin[i] = ^(gray >> i);

endmodule
```



Pointer Synchronizer pitfall

- Write and read pointers need to be registered in gray code as shown on previous slide.
- Don’t be tempted to cheat and register pointers in binary. What’s wrong with the synchronizer shown below?



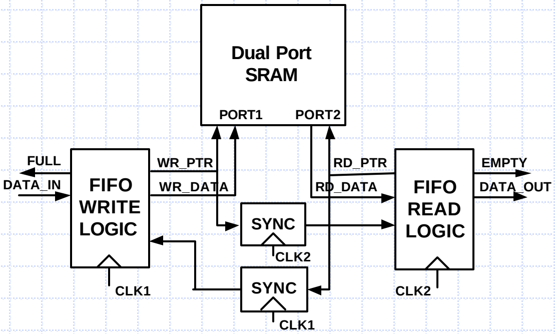
Answer to pitfall

- Combinational logic frequently contains hazards at the output (non fully covered Karnaugh map)
- Avoid this problem by using a registered value of PTR_IN

Pointer math pitfall

- When our pointer synchronizer goes metastable our new pointer value may not be updated until one cycle later.
- We need to be conservative when generating FULL and EMPTY signals to reflect this.
  - Typically FULL = 1 when WRITE catches READ. We need FULL = 1 when WRITE catches READ-1.
  - Typically EMPTY = 1 when READ catches WRITE. We need EMPTY = 1 when READ catches WRITE-1.

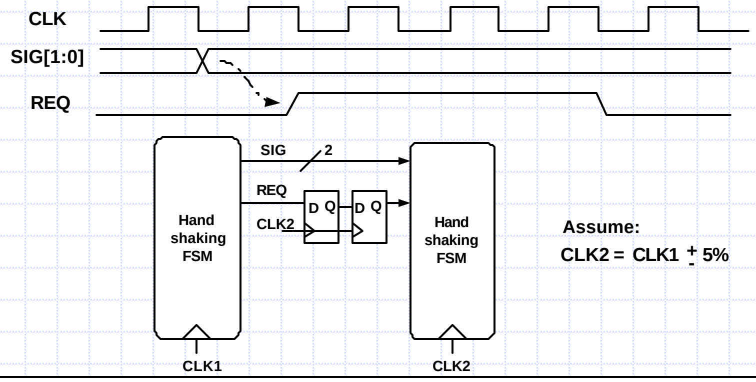
Final Synchronizer FIFO Design



Works for any phase/frequency relationship between CLK1 and CLK2

Mesosynchronous Designs

- When two systems of bounded frequency need communicate, open loop synchronization circuits can be used (no ACK)

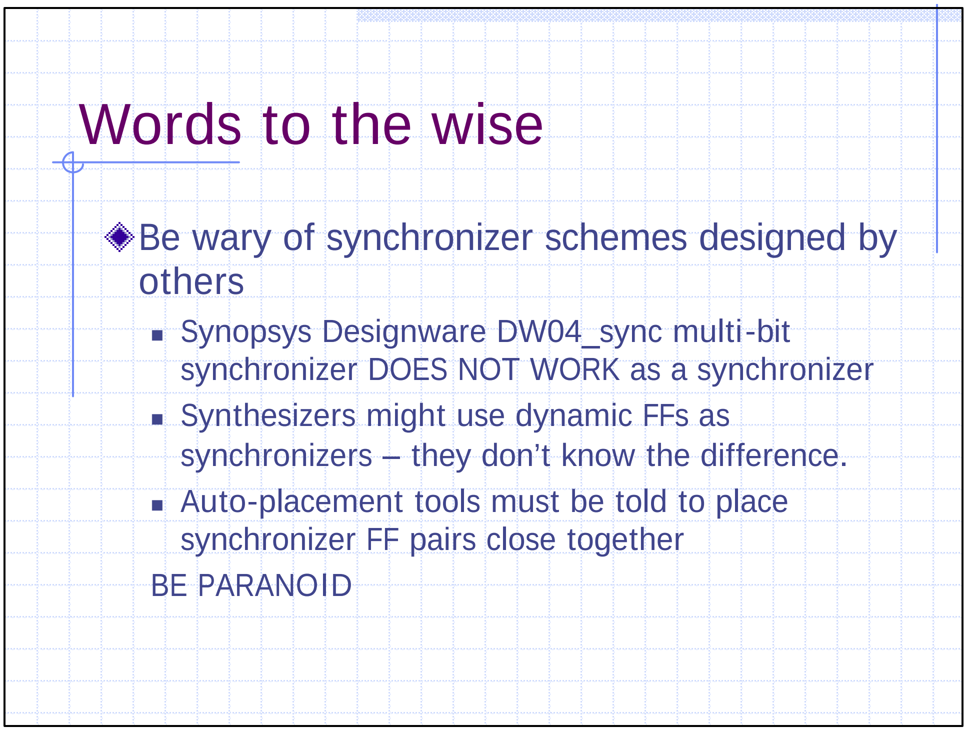


Assume:

CLK2 = CLK1 ± 5%

Mesosynchronous Tradeoffs

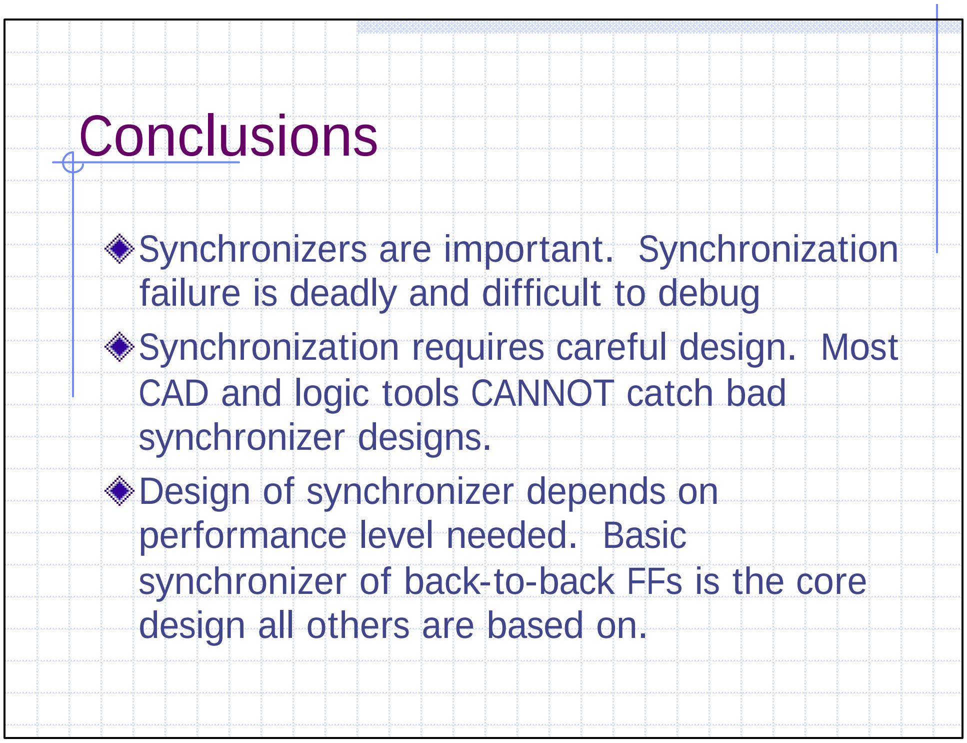
- Benefits to mesosynchronous designs
  - Less synchronization circuitry
  - Synchronizer might have lower latency vs. full 4-phase handshaking
- Costs of mesosynchronous designs
  - Synchronizer only works at certain frequency ratios (may hamper bringup/debug)
  - Intolerant of spec mistakes (maybe that unload frequency was supposed to be +- 50%!)



Words to the wise

- Be wary of synchronizer schemes designed by others
  - Synopsys Designware DW04_sync multi-bit synchronizer DOES NOT WORK as a synchronizer
  - Synthesizers might use dynamic FFs as synchronizers – they don't know the difference.
  - Auto-placement tools must be told to place synchronizer FF pairs close together

  BE PARANOID



Conclusions

- Synchronizers are important. Synchronization failure is deadly and difficult to debug
- Synchronization requires careful design. Most CAD and logic tools CANNOT catch bad synchronizer designs.
- Design of synchronizer depends on performance level needed. Basic synchronizer of back-to-back FFs is the core design all others are based on.