

Design of Digital Circuits and Systems

Clock Domain Crossing

Instructor: Justin Hsia

Teaching Assistants:

Colton Harris

Deepti Anoop

Gayathri Vadhyan

Jared Yoder

Lancelot Wathieu

Matthew Hung

Relevant Course Information

- ❖ Homework 5 due Friday (5/10)
- ❖ Lab 5 due next week (5/17)
 - Hardest/longest lab
 - You will need to use the VGA interface on LabsLand

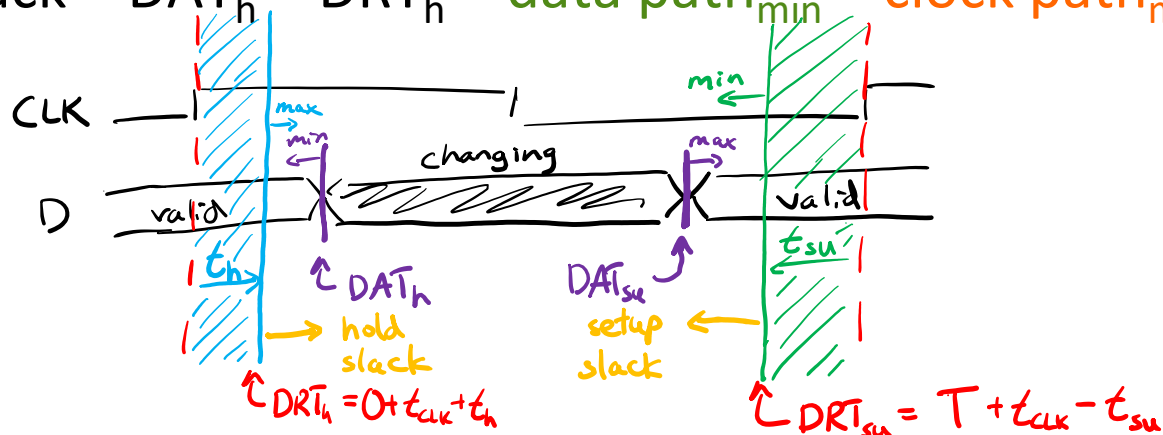
Mid-Quarter Survey Notes

- ❖ Pace is a bit fast and workload is heavy
 - More lecture examples, better & bigger handwriting
- ❖ Many complaints about LabsLand, understandably 😞
- ❖ More guidance on autograder expectations (*e.g.*, test cases, required formatting)
- ❖ Some complaints about the vagueness of some specs
 - Wishing for more artifacts to check if students are on the right track
- ❖ TAs are doing well in OHs and the queue works well
 - Wish there were more Ohs
- ❖ Website is generally working well
 - Sync assignment due dates to Canvas?
 - Assignments page for due dates (not just on Schedule)
 - Added Tools link to Gitlab

Review: Timing Slack

- ❖ *Slack* is how much wiggle room we have before we run into a potential timing violation (setup or hold)

- Can think of as worst-case analysis
- Setup slack = $DRT_{su} - DAT_{su} = \text{clock path}_{\min} - \text{data path}_{\max}$
- Hold slack = $DAT_h - DRT_h = \text{data path}_{\min} - \text{clock path}_{\max}$



- We looked specifically at the slack between a single source and destination register, but in reality, we'd want to know the minimum slack between *any two registers* in our system

Review: Timing Slack

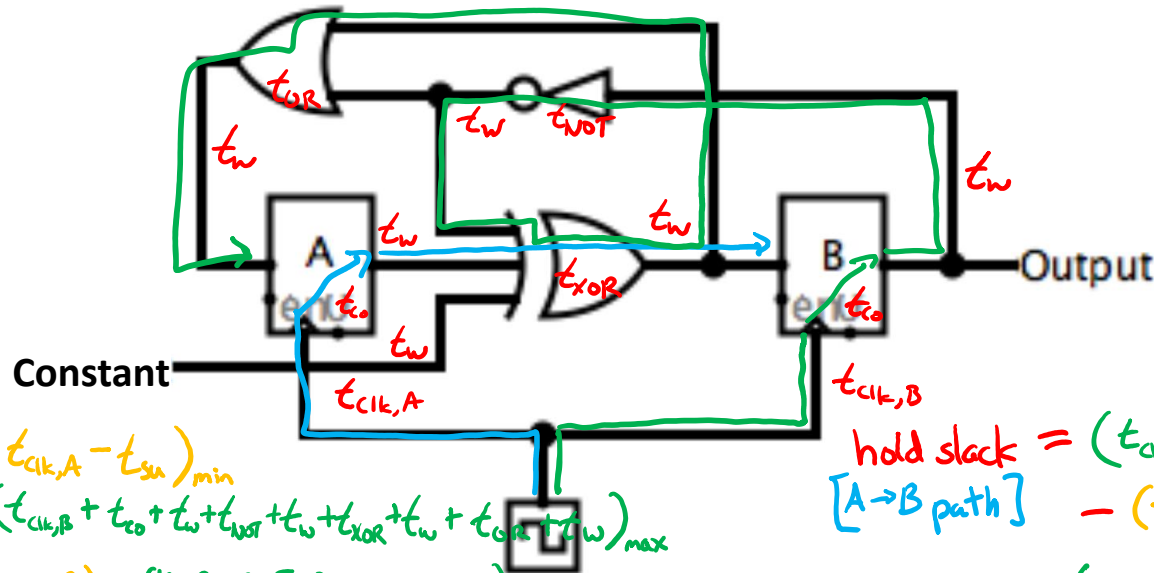
$$\text{setup slack} = \text{clock path}_{\min} - \text{data path}_{\max}$$

$$\text{hold slack} = \text{data path}_{\min} - \text{clock path}_{\max}$$

for destination register

need to analyze paths from: $A \rightarrow A, A \rightarrow B, B \rightarrow A, B \rightarrow B$

- ❖ Solve for the min setup and hold slacks if $T = 40$ ns
 - Registers: $t_h = 1$ ns, $t_{su} = 3$ ns, $t_{co} \in [6,9]$ ns
 - Delays: $t_{\text{wire}} \in [1,2]$ ns, $t_{\text{clk},A} \in [1,2]$ ns, $t_{\text{clk},B} \in [2,4]$ ns
 $t_{\text{NOT}} \in [3,5]$ ns, $t_{\text{OR}} \in [6,8]$ ns, $t_{\text{XOR}} \in [9,12]$ ns



$$\text{setup slack [B} \rightarrow \text{A path]} = (T + t_{\text{clk},A} - t_{su})_{\min} - (t_{\text{clk},B} + t_{co} + t_w + t_{\text{NOT}} + t_w + t_{\text{XOR}} + t_w + t_{co} + t_w)_{\max}$$

$$= (40 + 1 - 3) - (4 + 9 + 2 + 5 + 2 + 12 + 2 + 8 + 2)$$

$$= 38 \text{ ns} - 46 \text{ ns} = \boxed{-8 \text{ ns (setup violation!)}$$

$$\text{hold slack [A} \rightarrow \text{B path]} = (t_{\text{clk},A} + t_{co} + t_w + t_{\text{XOR}} + t_w)_{\min} - (t_{\text{clk},B} + t_h)_{\max}$$

$$= (1 + 6 + 1 + 9 + 1) - (4 + 1)$$

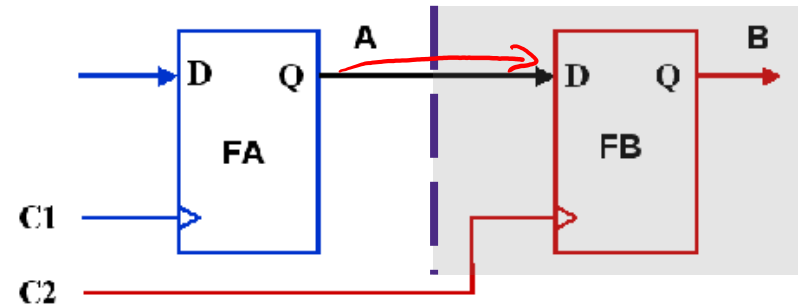
$$= 18 \text{ ns} - 5 \text{ ns} = \boxed{13 \text{ ns}}$$

Clock Domains

- ❖ A **clock domain** is all of the sequential logic that runs on the same clock/clock frequency
 - May have multiple clock domains in one device or different clock domains when communicating across devices
 - These can arise purposefully (*e.g.*, using multiple clocks) or inadvertently (*e.g.*, really bad clock skew and jitter)

Clock Domain Crossing (CDC)

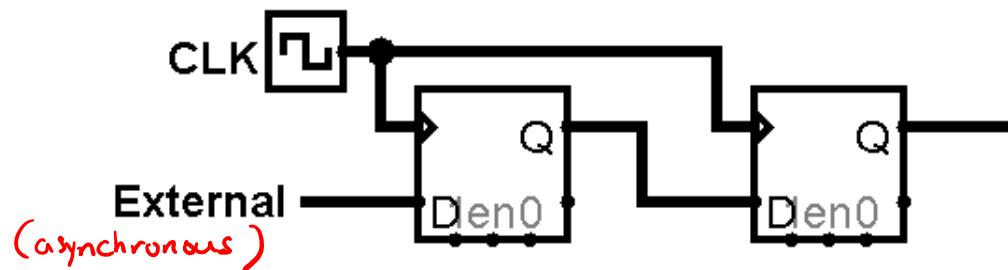
- ❖ Sending data from one clock domain to another is called a **clock domain crossing**
- ★ Can cause timing issues: metastability, data loss, and data incoherence
 - The **sending clock domain** and **receiving clock domain** are separated by a **CDC boundary**



- ❖ The phase and frequency relationships between clock domains may be *known* (**synchronous CDC**) or *unknown* (**asynchronous CDC**)

Synchronization

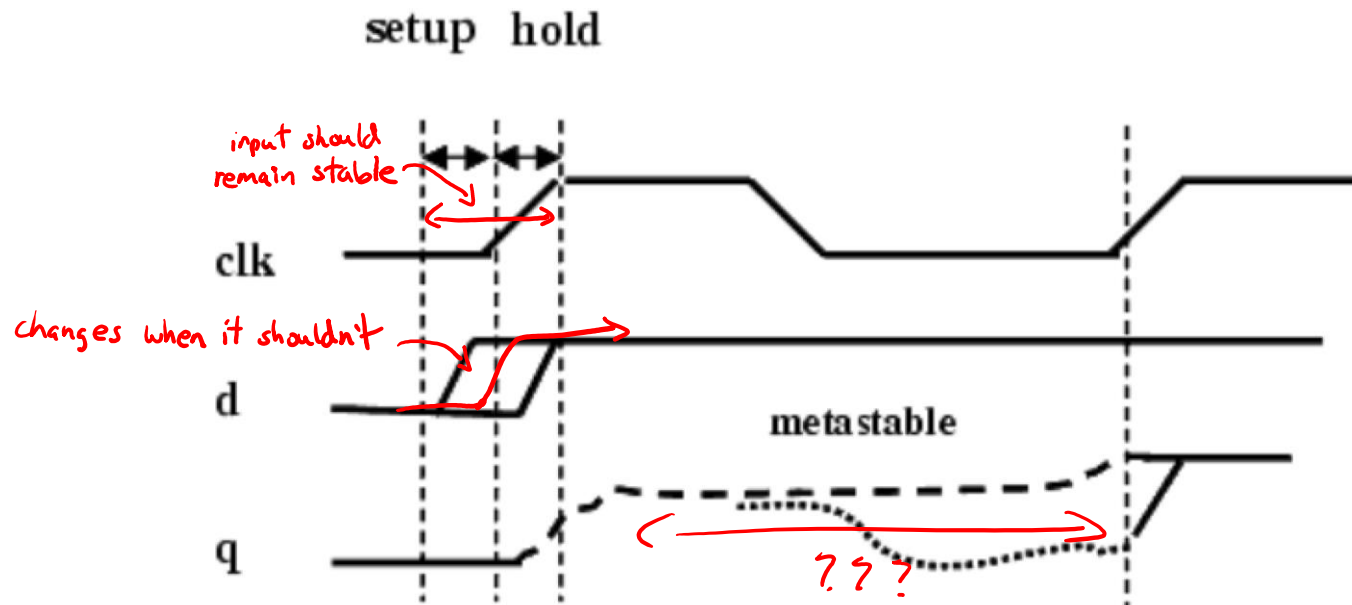
- ❖ **Synchronization:** coordination of events for proper transfer of data
 - *e.g.*, the two-flip-flop circuit recommended in 271



- We will discuss a number of other synchronizers today

Metastability

- ❖ **Metastability** is the ability of a digital system to persist for an unbounded time in an unstable equilibrium or metastable state
 - Caused by timing constraint violations
 - https://en.wikipedia.org/wiki/Metastability_in_electronics



Metastability

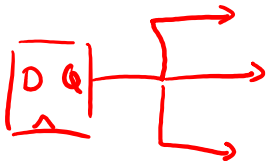
❖ Why is metastability bad?

- Circuit may be unable to settle into a stable '0' or '1' logic level within the time required for proper circuit operation
 - Unstable signals can also cause current spikes
- Unpredictable behavior or random value
 - Metastable signal is passed to combinational logic



CL reacts immediately to changes in signal

- Metastable signal passed to multiple destinations



value might be read differently at different destinations

- Downstream timing issues

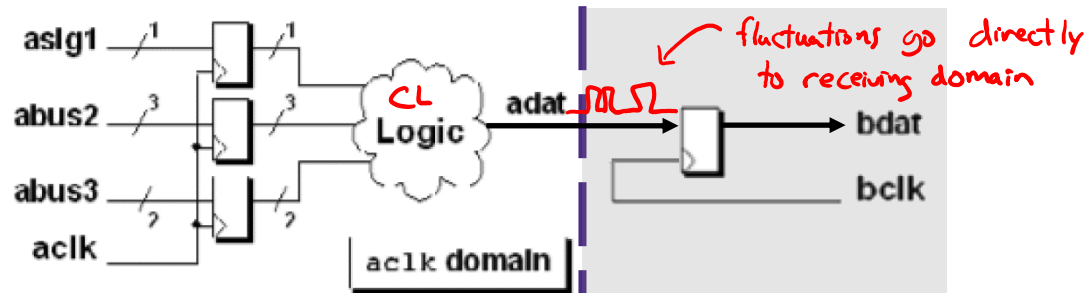


could violate timing constraints downstream

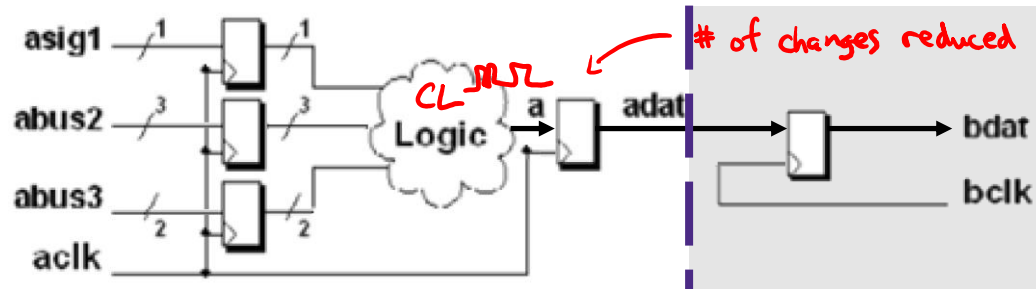
Metastability and CDC

- ❖ Metastability is inevitable in a multi-clock design, but want to prevent it as much as possible
 - Caused by different frequencies, clock skew, clock jitter
- ❖ Signals in the **sending clock domain** should be *synchronized* before being passed to a CDC boundary

■ Bad:

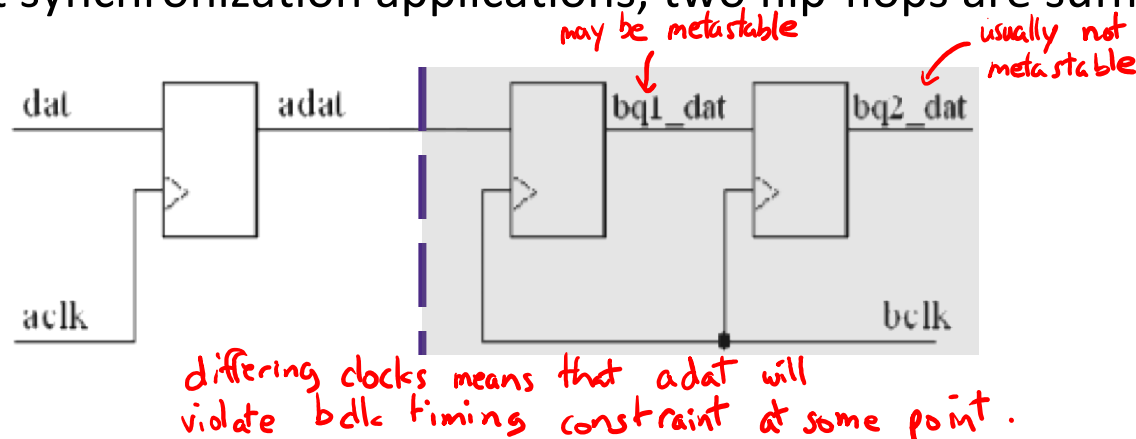


■ Good:



Metastability and CDC

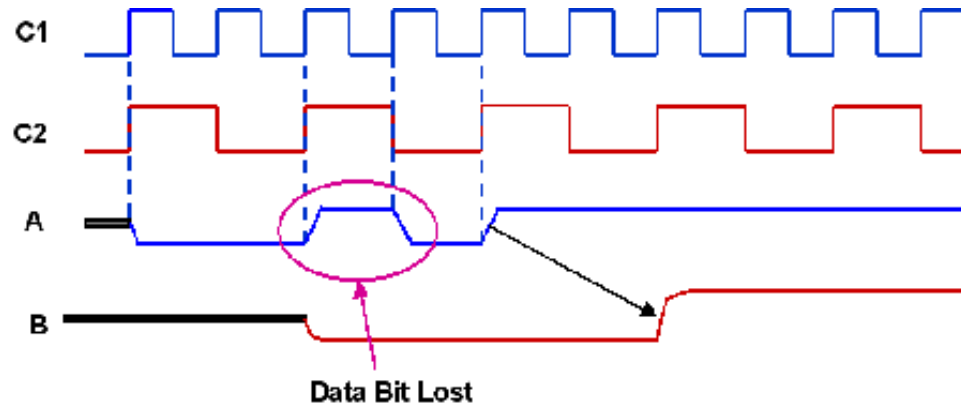
- ❖ Metastability is inevitable in a multi-clock design, but want to prevent it as much as possible
 - Caused by different frequencies, clock skew, clock jitter
- ❖ Signals in the **receiving clock domain** should be passed through a *synchronizer* before rest of system
 - Add flip-flops to give metastable signals time to settle so rest of system receives clean/valid signals
 - For most synchronization applications, two flip-flops are sufficient



Technology Break

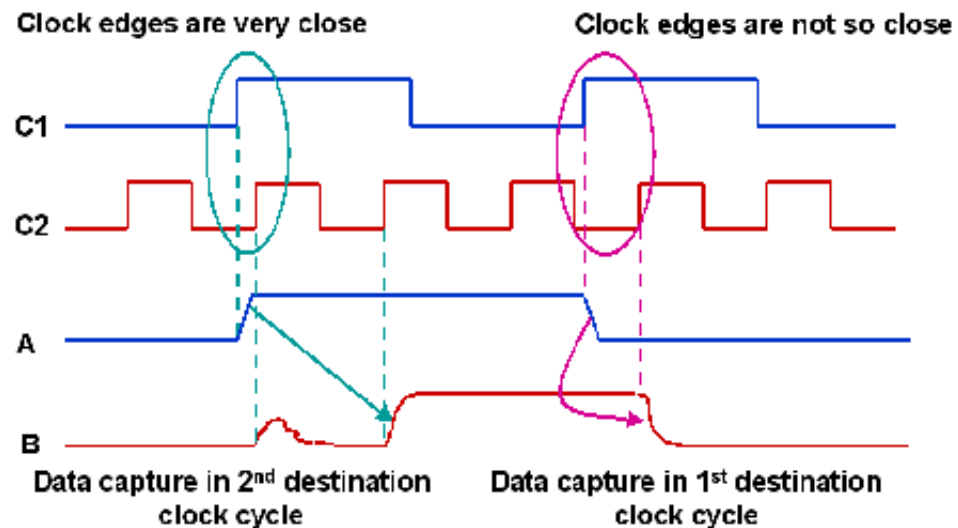
Data Loss

- ❖ **Data loss** refers to information lost due to failures
- ❖ Data loss during CDC:
 - Typically means *incorrect reads* or *missed input changes*



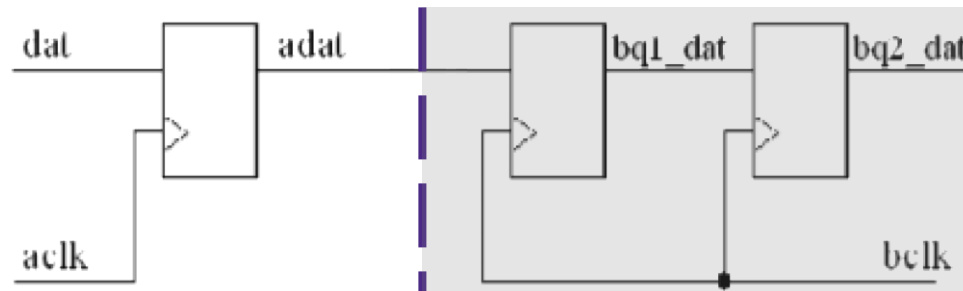
Data Loss

- ❖ **Data loss** refers to information lost due to failures
- ❖ Data loss during CDC:
 - Typically means *incorrect reads* or *missed input changes*
 - Change in **sending domain** data may not be properly captured on first clock edge in **receiving domain** due to metastability:



Dealing with Data Loss

- ❖ **Sending domain** data should be held constant long enough to be captured in the **receiving domain** (at least 2 receiving clock cycles)
 - No longer a cycle-by-cycle correspondence between **sending** and **receiving** domain data
- ❖ Need to account for differences in clock speeds
 - Slower → Faster (*synchronous*): holding input constant for one clock cycle is probably enough since $T_{\text{sending}} > T_{\text{receiving}}$
 - Can do longer to be more sure

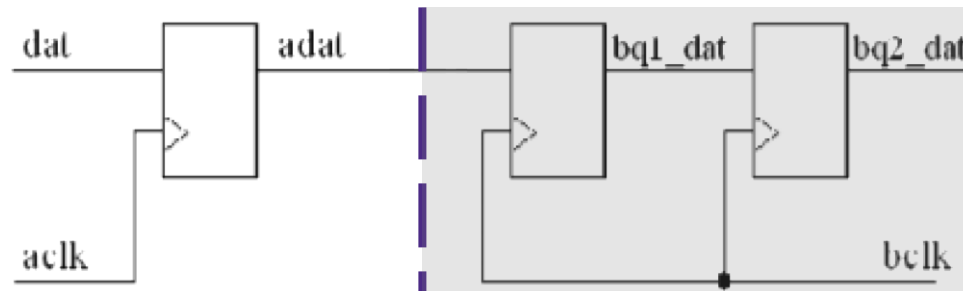


Dealing with Data Loss

- ❖ **Sending domain** data should be held constant long enough to be captured in the **receiving domain**
 - No longer a cycle-by-cycle correspondence between **sending** and **receiving** domain data
- ❖ Need to account for differences in clock speeds
 - Faster → Slower (*synchronous*): “open loop” solution is to holding input constant for longer by “stretching” them

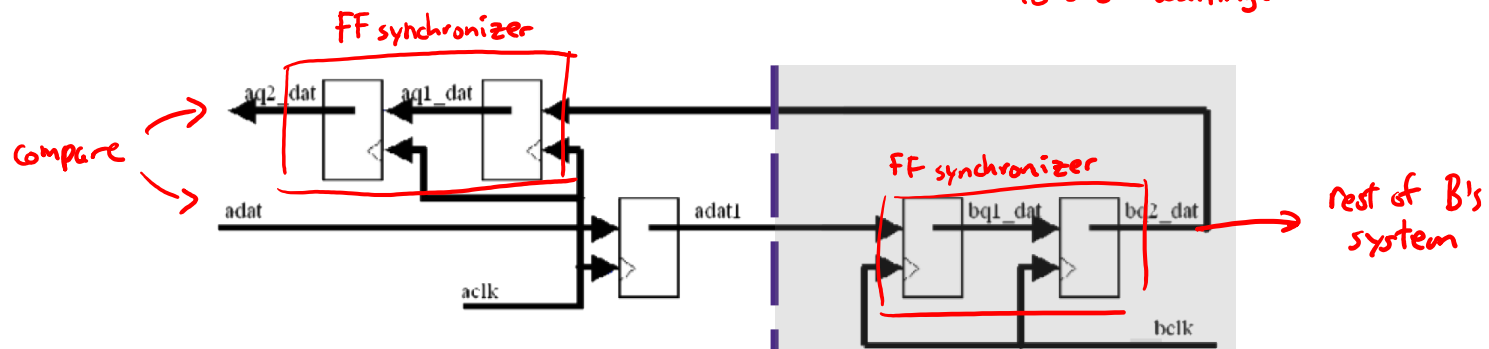
(no feedback)

every change in *adat* needs to be held for a certain # of clock cycles



Dealing with Data Loss

- ❖ **Sending domain** data should be held constant long enough to be captured in the **receiving domain**
 - No longer a cycle-by-cycle correspondence between **sending** and **receiving** domain data
- ❖ Need to account for differences in clock speeds
 - Faster → Slower (*synchronous*): “closed loop” solutions can check for proper receipt of signal before changing signal

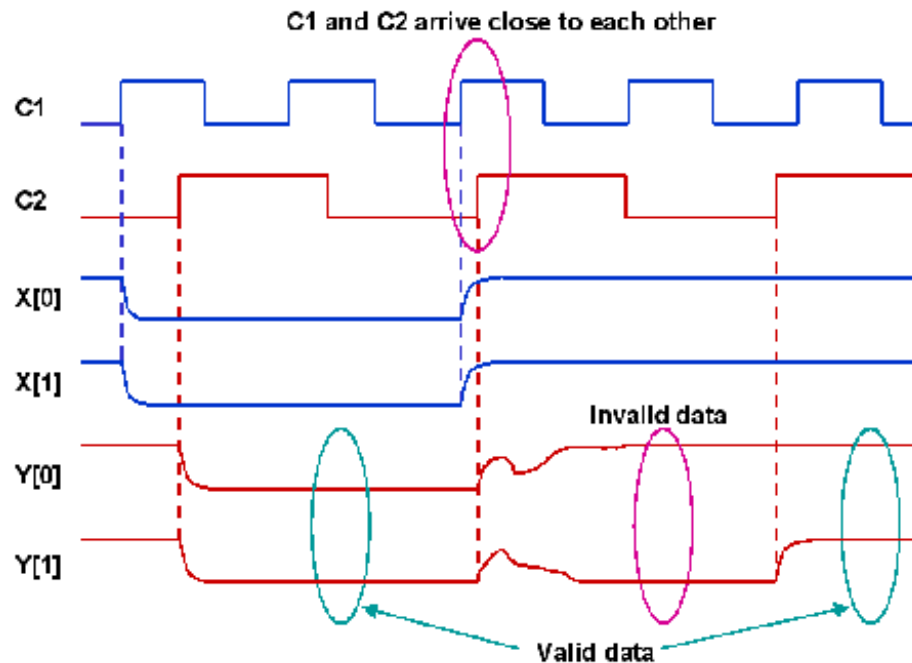


Dealing with Data Loss

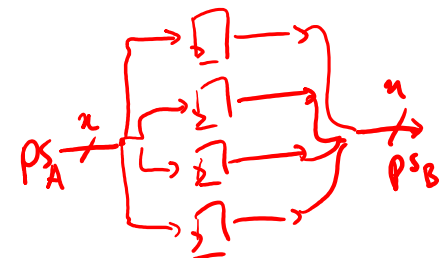
- ❖ **Sending domain** data should be held constant long enough to be captured in the **receiving domain**
 - No longer a cycle-by-cycle correspondence between **sending** and **receiving** domain data
- ❖ Need to account for differences in clock speeds
 - For *asynchronous* CDC, will need to employ a more sophisticated synchronization technique
 - e.g., handshake or FIFO buffer *(discussed later today)*

Data Incoherence

- ❖ **Data incoherence** is the invalid combination of values caused by differing outcomes when passing multiple signals (*e.g.*, a vector) through a CDC simultaneously
 - Depending on what the data represents, this invalid state may lead to functional errors



passing a bus through a register is really passing each bit through a FF:



each FF could experience metastability differently and at different times!

Dealing with Data Incoherence

- ❖ Specifically trying to avoid *invalid states*
 - The issue of incorrect reads is metastability and data loss
- ❖ When possible, restrict data changes to 1 bit at a time
 - Even on failed transition, will remain in a valid state
 - e.g., a Gray code counter

Decimal	0	1	2	3	4	5	6	7
Binary	000	001	010	011	100	101	110	111
Gray	000 → 001 → 011 → 010 → 110 → 111 → 101 → 100							

only 1 bit changes at a time

- ❖ Ensure that system can recover from invalid states
 - May result in less optimized logic (fewer don't cares)
 - e.g., force transitions from invalid to valid states in FSM (make sure not a sink state)

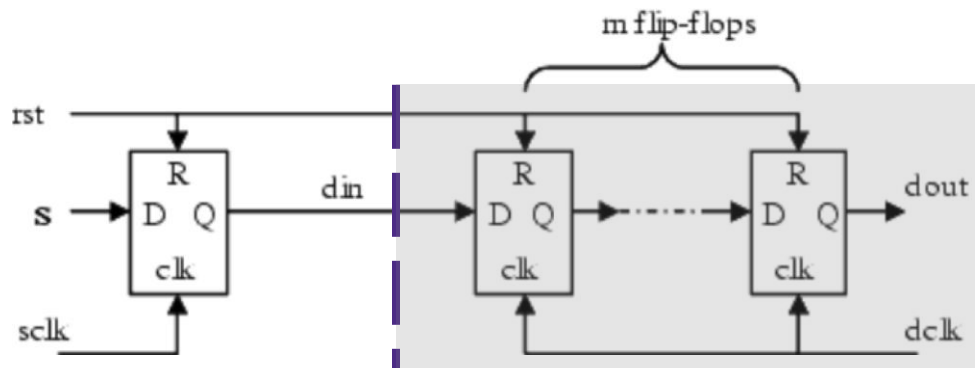
Synchronization of CDC Data Signals

- ❖ The previously mentioned techniques of adding synchronizing flip-flops and using Gray codes are not generally sufficient for data buses through a CDC
- ❖ Three common methods for synchronizing data between clock domains are:
 - MUX-based synchronizers
 - Handshake signals
 - FIFO buffers

Flip-Flop Synchronizer

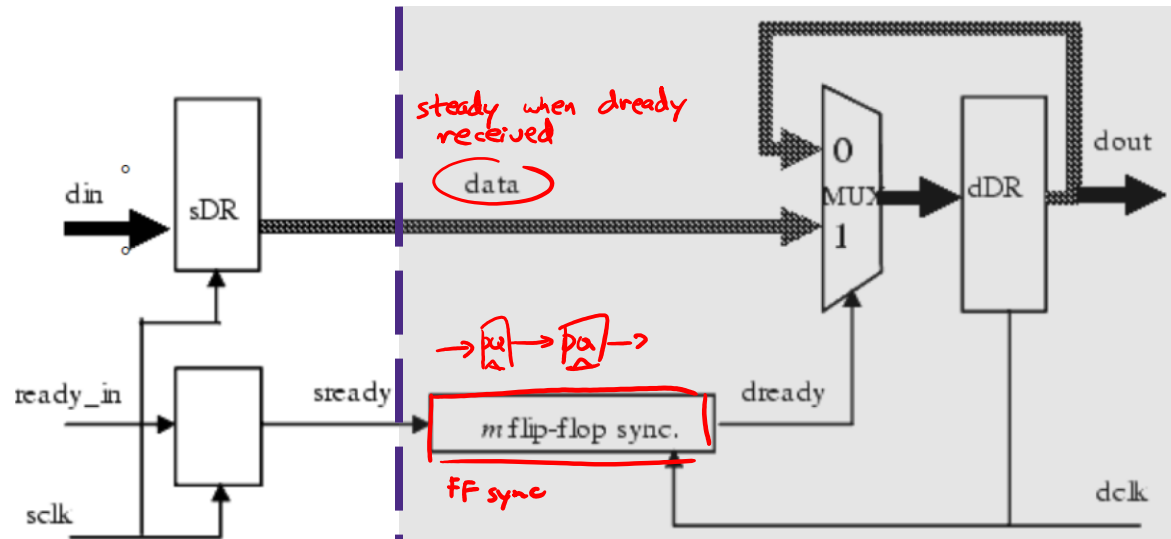
(usually 2)

- ❖ Add m flip-flops in the receiving domain
 - Generally want to hold input data stable for $m+1$ receiving domain clock edges to ensure settling of data



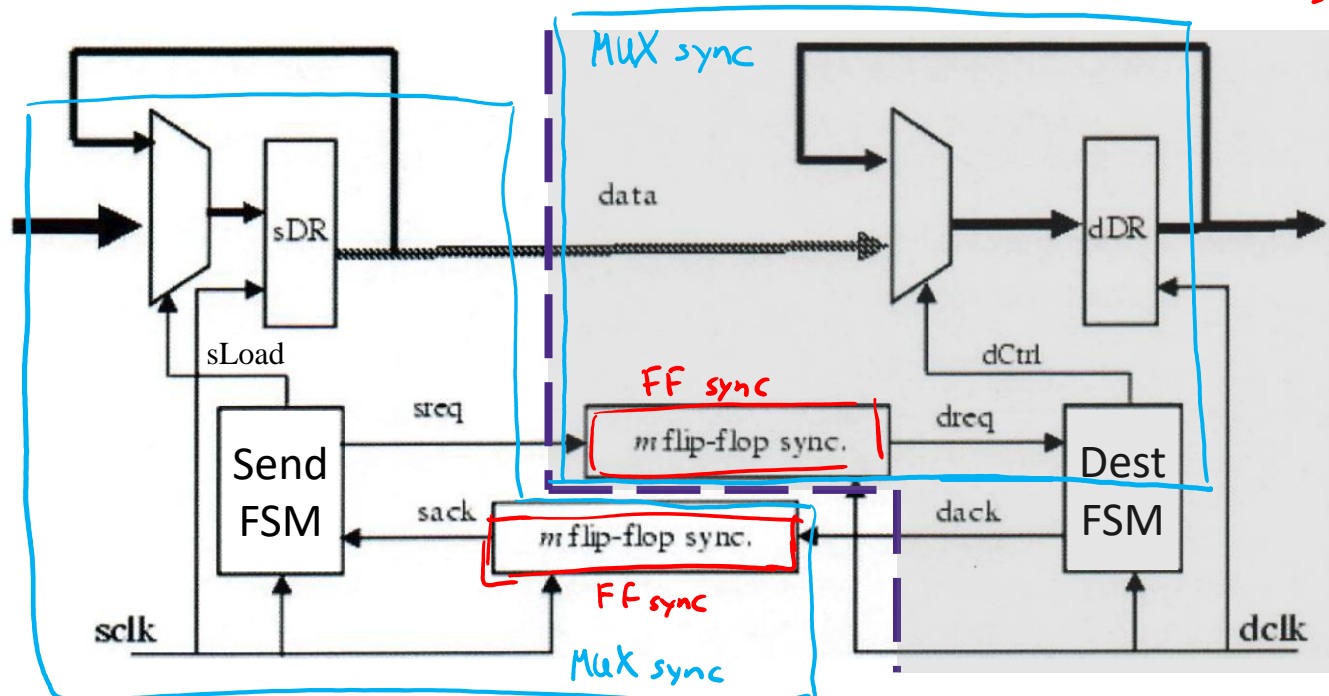
Recirculation MUX Synchronizer

- ❖ Control signals typically can sufficiently be handled by flip-flop synchronizers, so can use these to choose when to sample the data in the receiving domain



Handshake Synchronizer

- ❖ A *request-and-acknowledge* scheme to guarantee the sampling of correct data
 - The relationship of the two clocks doesn't matter!
 - Best if data doesn't change very frequently *because handshake takes so long*



Handshake Implementation

❖ Sender:

- Input data should be held stable (\overline{sLoad}) until acknowledgement received
- Request ($sreq$) signal should be stable for $m+1$ cycles in the receiving clock domain to ensure transmission
- A new request should not be asserted until the acknowledgement from the previous data value has been de-asserted

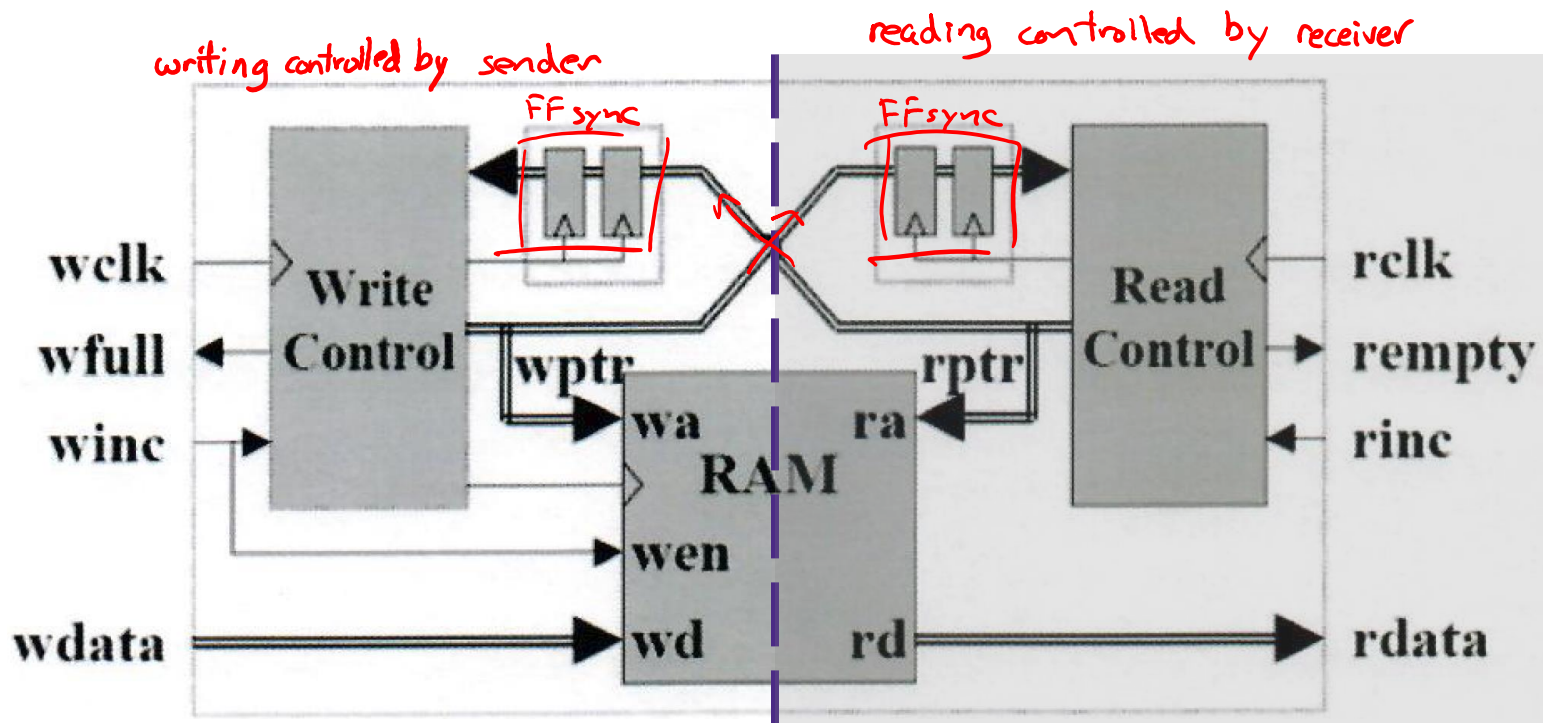
Handshake Implementation

❖ Receiver


- Input data should not be sampled ($dCtr1$) until request received
- Acknowledgement ($dack$) signal should be stable for $m+1$ cycles in the sending clock domain to ensure transmission
- A new acknowledgement should not be asserted until the next request is received

FIFO Synchronizer

- ❖ A dual-clock asynchronous FIFO can be used when the high latency of a handshake synchronizer cannot be tolerated



FIFO Synchronizer

-  The golden rules of FIFO buffers still apply:
- The sender should never write when the FIFO is full
 - The receiver should never reads when the FIFO is empty

❖ Implementation details:

- Have to deal with FF synchronizer delays, but not every time
- Pointer (read and write) must be gray-coded at their source
 - Comparison of gray-coded positions is more complex
- Determining full/empty signals on time is tricky
- May need to slow data generation rate (or introduce breaks/pauses) if not being read fast enough

Review Questions

- There are reasonable arguments for both → incorrect read counts as "lost" data
 → 1-bit state can be incorrect/incoherent
- ❖ A 1-bit signal becomes metastable in the receiving domain and is read incorrectly. This is an example of:

Data loss Data incoherence Both Neither

missed sending domain transitions (arrow from Data loss to Data incoherence)
requires multiple bits (arrow from Data incoherence to Both)

- ❖ What is the trade-off between using an open-loop and closed-loop CDC solution?

speed vs. certainty / feedback.

open loop is quicker, but you are trusting that the signal is read properly

closed loop gives you the certainty of receipt via feedback but takes longer

- ❖ Which synchronizer is slowest (and why)?

Flip-flop Recirculation MUX Handshake FIFO

uses both FF synch & recirculation MUX with feedback