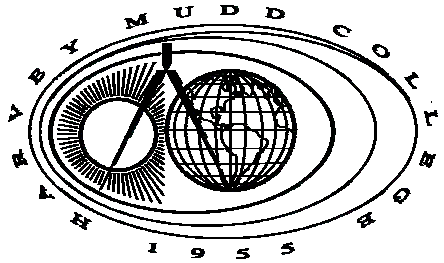
Timing & Synchronization

January 31, 2006



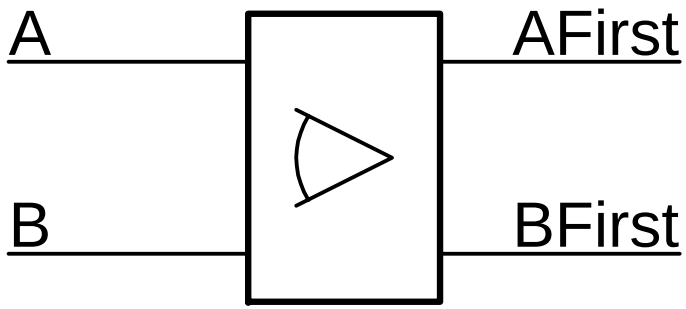
Sarah Harris
Engineering Department
Harvey Mudd College
sarah_harris@hmc.edu

A Quick Overview

- Synchronization
  - determining an event order
  - used for
    - moving a signal into a clock domain
    - asynchronous arbitration
- Synchronization Failure
  - as the time between two signals decreases it becomes more difficult to tell which came first
  - synchronizer may hang in a metastable state, unable to decide
  - different parts of the circuit may interpret result differently
- Failure Probability
  - is proportional to fraction of *vulnerable* time
  - exponentially decreases with waiting period
  - exponentially increases with flip-flop regeneration time constant
  - failure rate is proportional to event rate
- Synchronization Hierarchy
- Mesochronous Synchronizers
  - delay-line synchronizer
  - two-register synchronizer
  - FIFO synchronizer
- Plesiochronous Synchronizers
  - phase slip and flow control
- Periodic Synchronizers
  - clock prediction - looking into the future

What is Synchronization?

- A *synchronizer* determines the order of events on two signals
- Which event came first?
  - Does it matter? Some times synchronization is unnecessary
- Often one signal is a clock
  - did the data go high before or after the clock went high?
- Why is this problem hard?



Uses of Synchronization

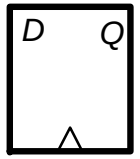
- Sampling asynchronous inputs with a clock
  - e.g., particle counter or pushbutton
- Crossing clock domains
  - sampling a synchronous signal with a *different* clock
  - this is an easier problem if both clocks are periodic
- Arbitration of asynchronous signals
  - e.g., request line for shared resource
  - game-show pushbutton

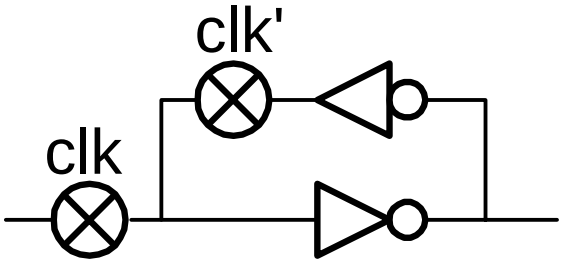
Synchronization Failure

- Which came first, event on A or event on B?
- The closer the race, the harder it is to call
- When the events are very close, the synchronizer may enter a *metastable* state
- The synchronizer may take an arbitrary amount of time to *exit* this state
- Synchronizer output may be interpreted inconsistently in the meantime

Static Flip-Flop Dynamics

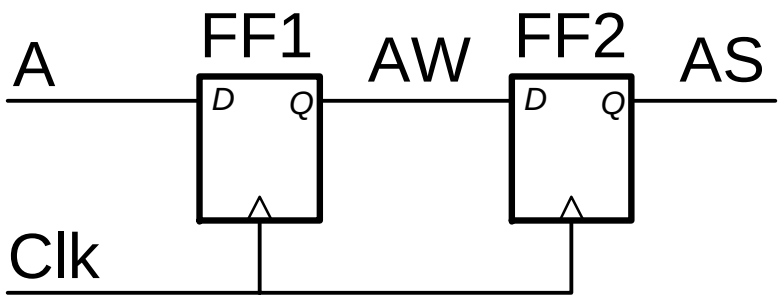
- Initial voltage difference depends on Δt
- Voltage difference increases exponentially after clock rises

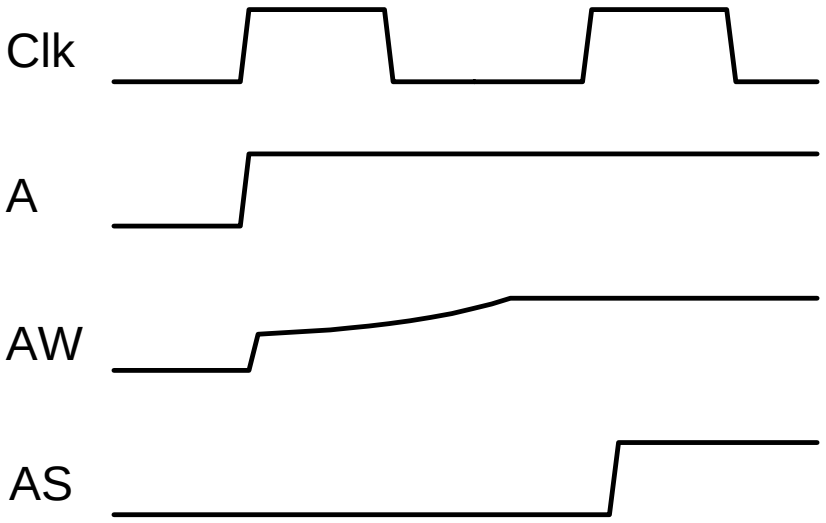




A Brute-Force (Waiting) Synchronizer

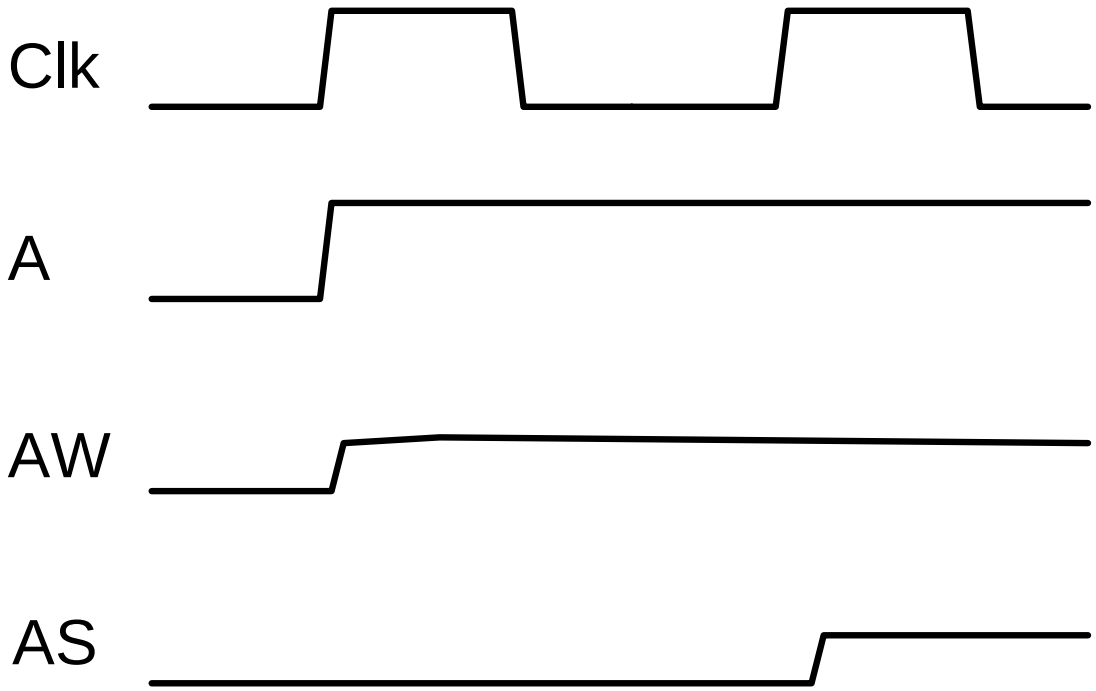
- To sample an asynchronous signal with a clock
- Sample signal with FF1
  - may go into a metastable state
- Wait for possible metastable stages to decay
  - time t_w
- Sample output of FF1





Synchronization Failure

- What happens if FF1 is still in a metastable state when FF2 is clocked?
- What is the probability that this will happen?

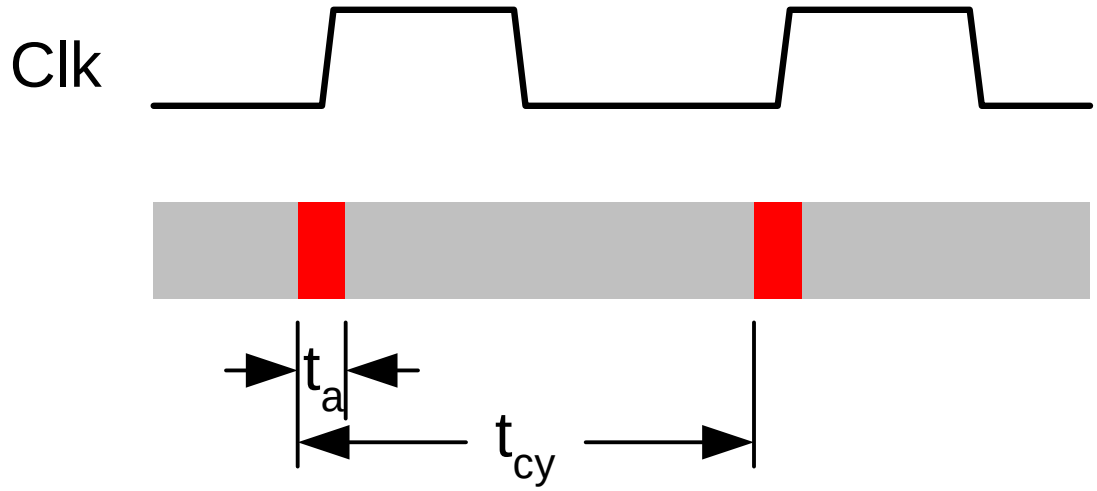


Calculating Synchronization Failure (The Big Picture)

P(failure) = P(enter metastable state) x P(still in state after t_w)

Probability of Entering a Metastable State

- FF1 may enter the metastable state if the input signal transitions during the *aperture* time of the flip flop
- Probability of a given transition being in the aperture time is the fraction of time that *is* aperture time



$$P_E = \frac{t_a}{t_{cy}} = f_{cy} t_a$$

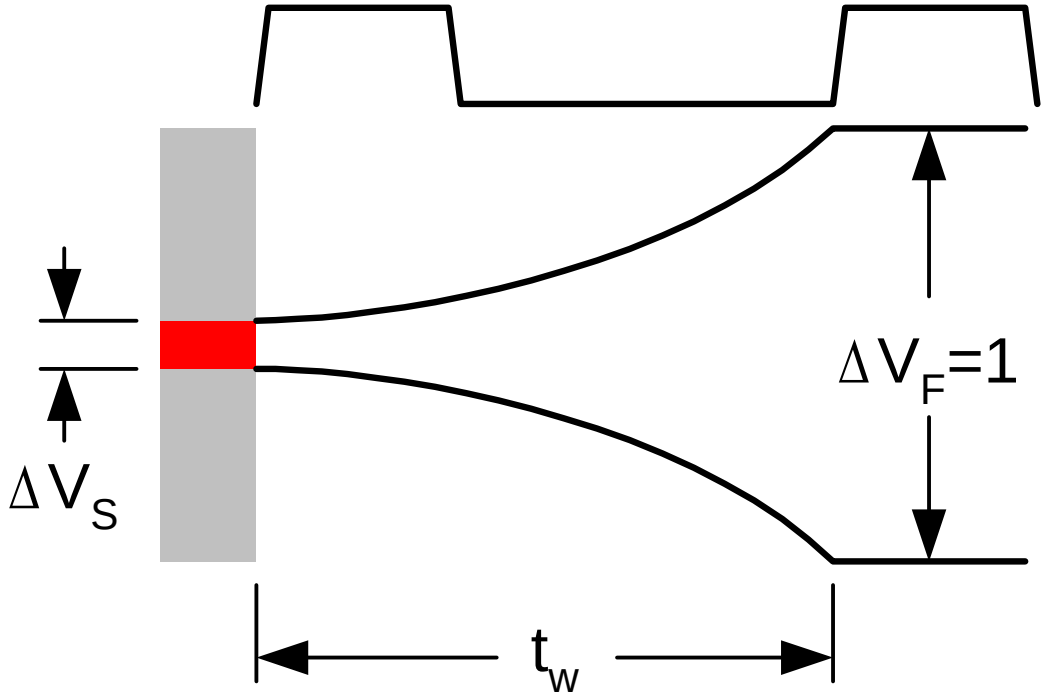
Probability of Staying in the Metastable State

- Still in metastable state if initial voltage difference was too small to be exponentially amplified during wait time

$$\Delta V_S = \Delta V_F \exp\left(\frac{-t_w}{\tau_S}\right)$$

$$P_S = \exp\left(\frac{-t_w}{\tau_S}\right)$$

- Probability of starting with this voltage is proportion of total voltage range that is 'too small'



Example Failure Rate Calculation

- Suppose a 500MHz clock samples a 10MHz asynchronous signal
- Flip-flops have aperture and regeneration time of 100ps
- What is the probability of synchronization failure?
- What is the failure frequency?

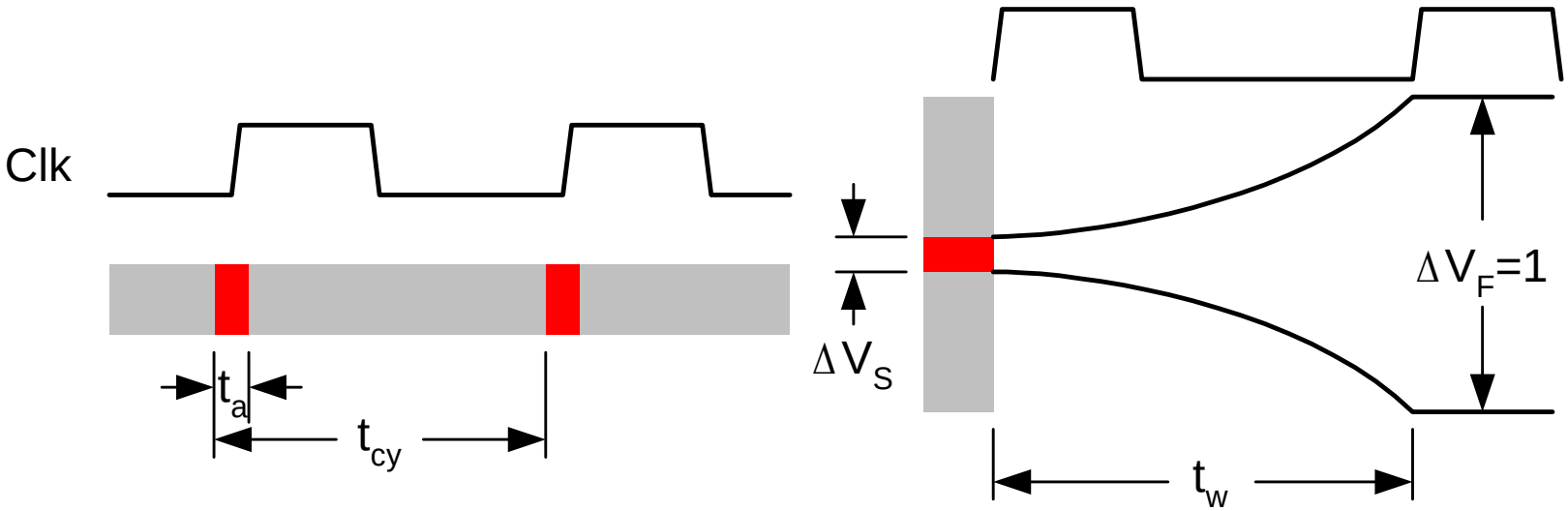
t_a	1.0E-10
f_cy	5.0E+08
τ_s	1.0E-10
t_w	2.0E-09
P_E	5.0E-02
P_S	2.1E-10
P_F	1.0E-11
f_e	1.0E+07
f_F	1.0E-04

Failure Probability and Error Rate

- Each event can potentially fail.
- Failure rate = event rate x failure probability

$$P_F = P_E P_S = t_a f_{cy} \exp\left(\frac{-t_w}{\tau_S}\right)$$

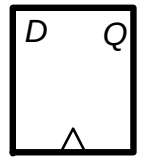
$$f_F = f_e P_F = t_a f_e f_{cy} \exp\left(\frac{-t_w}{\tau_S}\right)$$

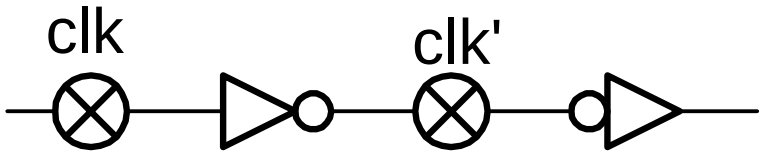


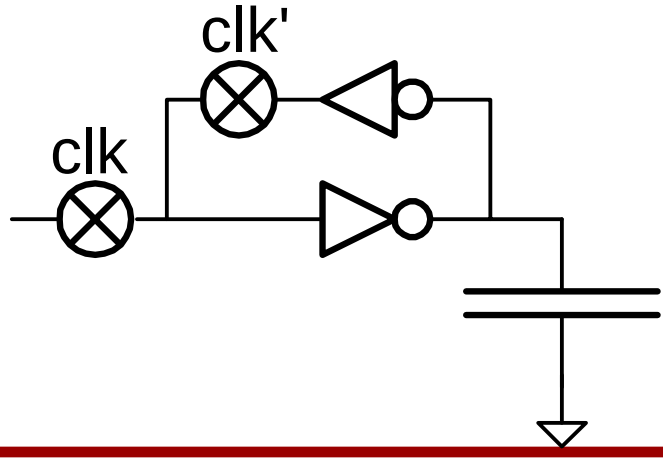
Common Pitfalls

- Its easy to get a synchronizer design wrong
- The two most common pitfalls are:
  - using a non-restoring (or slowly restoring) flip-flop
    - τ_S needs to be small
  - not isolating the flip-flop feedback loop

$$P_F = t_a f_{cy} \exp\left(\frac{-t_w}{\tau_S}\right)$$







Synchronization Hierarchy

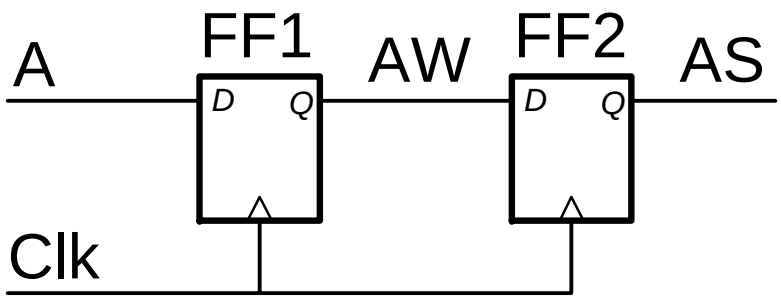
- The difficulty of synchronization depends on the relationship between events on the signal and events on the clock
- Synchronous
  - signal events always happen outside of the clock's keep-out region
    - same clock
- Mesochronous
  - signal events happen with a fixed but unknown phase relative to the clock
    - same frequency clock
- Plesiochronous
  - phase of signal events changes slowly with time
    - slightly different frequency clock
- Periodic
  - signal events are periodic
    - includes meso- and pleisochronous
    - signal is synchronized to some periodic clock
- Asynchronous
  - signal events may occur at any time

Synchronization Hierarchy Summary

Type	Frequency	Phase
Synchronous	Same	Same
Mesochronous	Same	Constant
Plesiochronous	Small Difference	Slowly Varying
Periodic	Different	Periodic Variation
Asynchronous	N/A	Arbitrary

The Brute-Force Synchronizer

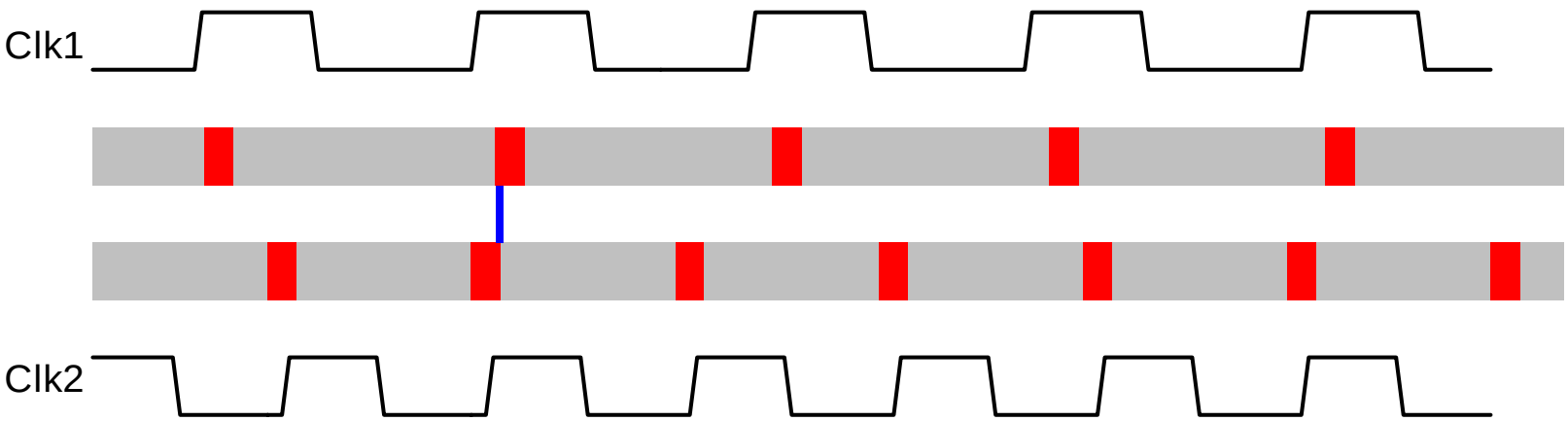
- How do we compare synchronizers?
  - synchronizer delay
  - failure rate
- For the brute-force synchronizer
  - $t_d=t_w+2(t_s+t_{dCQ})$
  - $f_f=t_a f_e f_{cy} \exp(-t_w/\tau_s)$
- Can we do better?



Periodic Synchronizers
The Big Picture

- If an input signal is synchronized to *some* periodic clock, we can predict when its events are allowed to happen arbitrarily far into the future
- Thus, we can determine well in advance if the signal is *safe* to sample on a given clock cycle
  - if it is, we just sample it
  - if it isn't, we delay the signal (or the clock) to make it safe
- This allows us to move the waiting time, t_W, out of the critical path.
  - we can make it very long without adding latency

Periodic Synchronizers
The Illustration

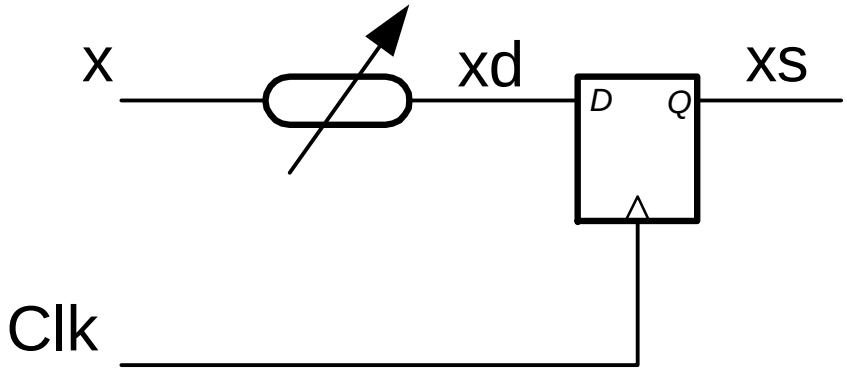


Mesochronous Synchronization

- The phase difference between the signal and the clock is **constant**
  - typical of systems where we distribute a master clock with no deskew
- Thus, we only need to synchronize once for all time!
- During reset check the phase
  - if its OK, sample the signal directly for ever
  - if its not, sample the signal after delay for ever
  - this phase check is the only asynchronous event we ever sample - and we can afford to wait a long time

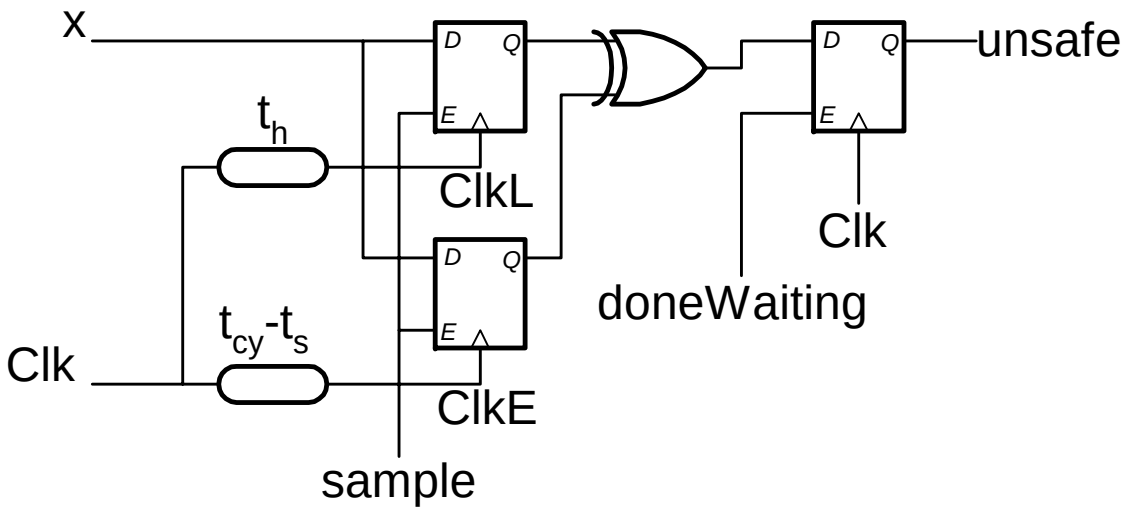
Delay-Line Synchronizer

- For mesochronous and plesiochronous signals
- Delay signal as needed to keep transitions out of the *keep-out* region of the synchronizer clock
- How do we set the delay line?



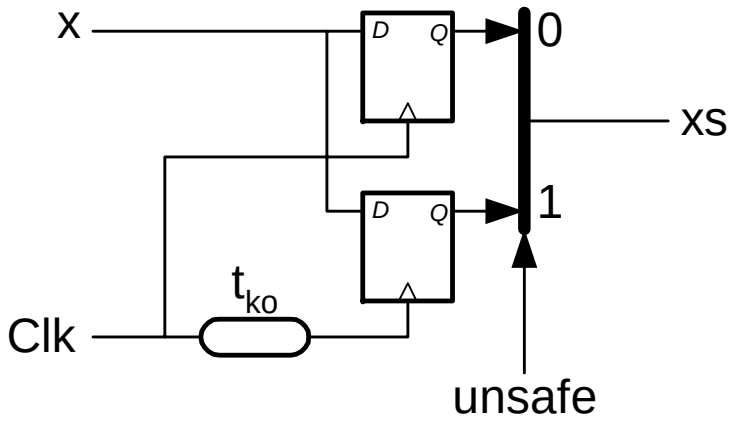
Detecting an Unsafe Signal

- To see if a signal is unsafe, see if it changes in the forbidden region
  - sample just before and after the forbidden region and see if result is different
- These samples may hang the flip-flop in a metastable state
  - need to wait for this state to decay
  - if mesochronous we can wait a very long time since we only have to do this once



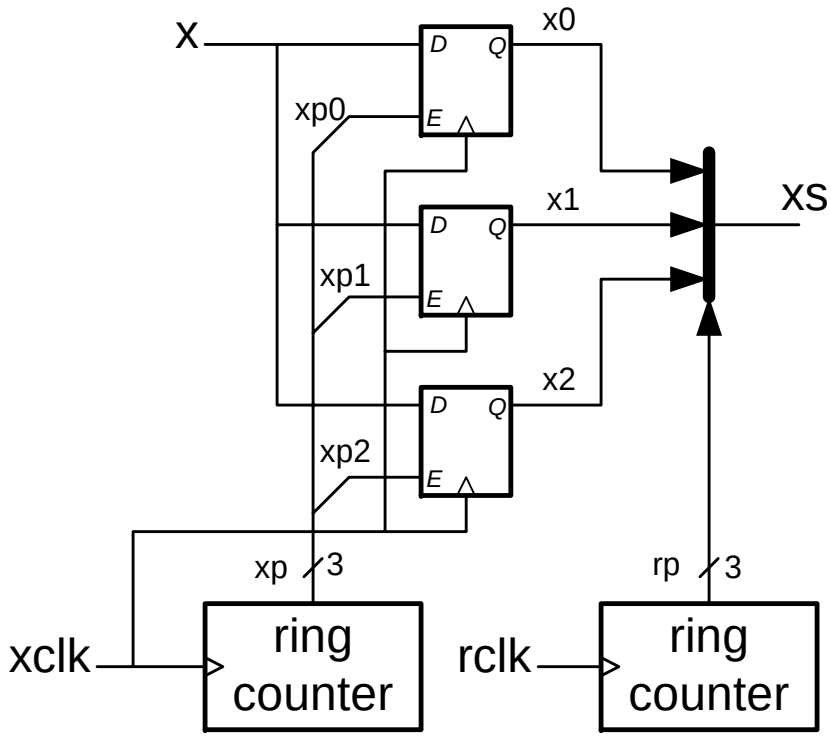
Two-Register Synchronizer

- The delay-line synchronizer has two problems
  1. Its expensive, we need a delay line for each input
  2. We can't use it with clocked receivers
- Both problems are solved by the *two-register synchronizer*
- We delay the clock rather than the data
  - sample the data with normal and delayed clock
  - pick the 'safe' output
- Can we just mux the clock?



FIFO Synchronizer

- A first-in-first-out (FIFO) buffer can be used to move the synchronization out of the data path
- Clock the data into the FIFO in one clock domain (*xclk)*
- Mux the data out of the FIFO in a second clock domain (*clk)*

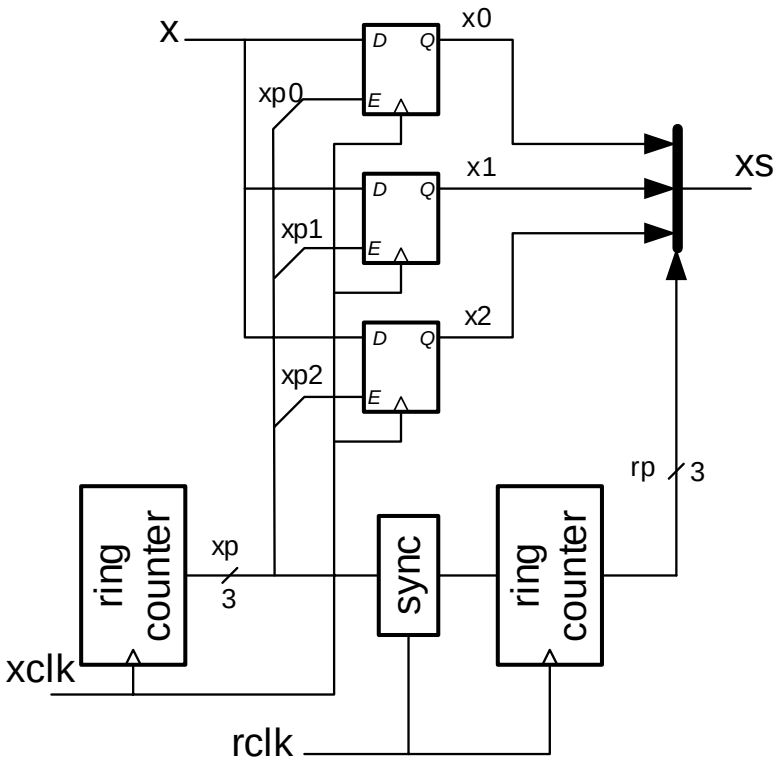


Plesiochronous Timing

- With plesiochronous timing, one clock is running slightly faster than the other
  - e.g., system with independent crystal oscillators with same nominal frequency (±200ppm)
- The same basic synchronizer types apply
  - delay line
  - two-register
  - FIFO
- But...
- we need to resynchronize periodically
  - e.g., once every 1,000 clocks
- we need flow control
  - have to match data rate of tx and rx even if clock rate is different
  - eventually the phase *wraps* and we either get 2 or 0 data elements during a particular clock
    - unless we make sure we are not sending data when the phase wraps

A Plesiochronous FIFO Synchronizer

- Insert data with transmit clock (xclk)
- Remove data with receive clock (rclk)
- Periodically update the receive pointer (rp) by synchronizing the transmit pointer (xp) to the receive clock
  - how do we know when to do this?
  - what do we do if rp increments by 2 or 0 when we update it?



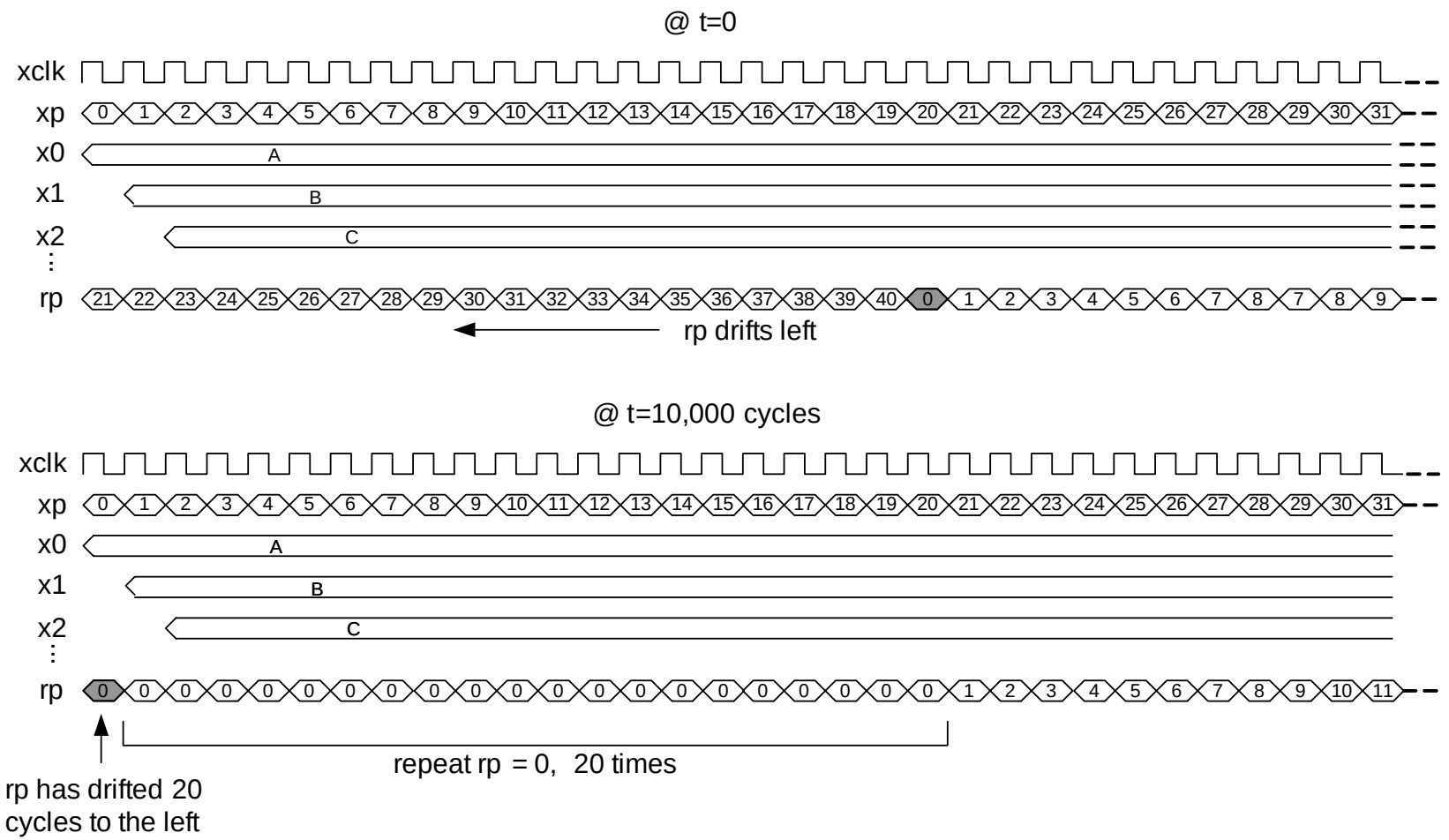
Open-Loop and Closed-Loop Flow Control

- We need to keep a fast transmitter from overrunning a receiver
- (or a slow transmitter from underrunning the receiver)
- Open-loop approach
  - insert *lots* of *nulls* into the data stream at the transmitter
  - enough so that rate of non-nulls is less than the rate of the slowest possible receiver
  - when the receiver underruns it inserts another null
- Closed-loop approach
  - receiver applies *back pressure* when it is about to be overrun
  - still has to insert nulls when it is underrun

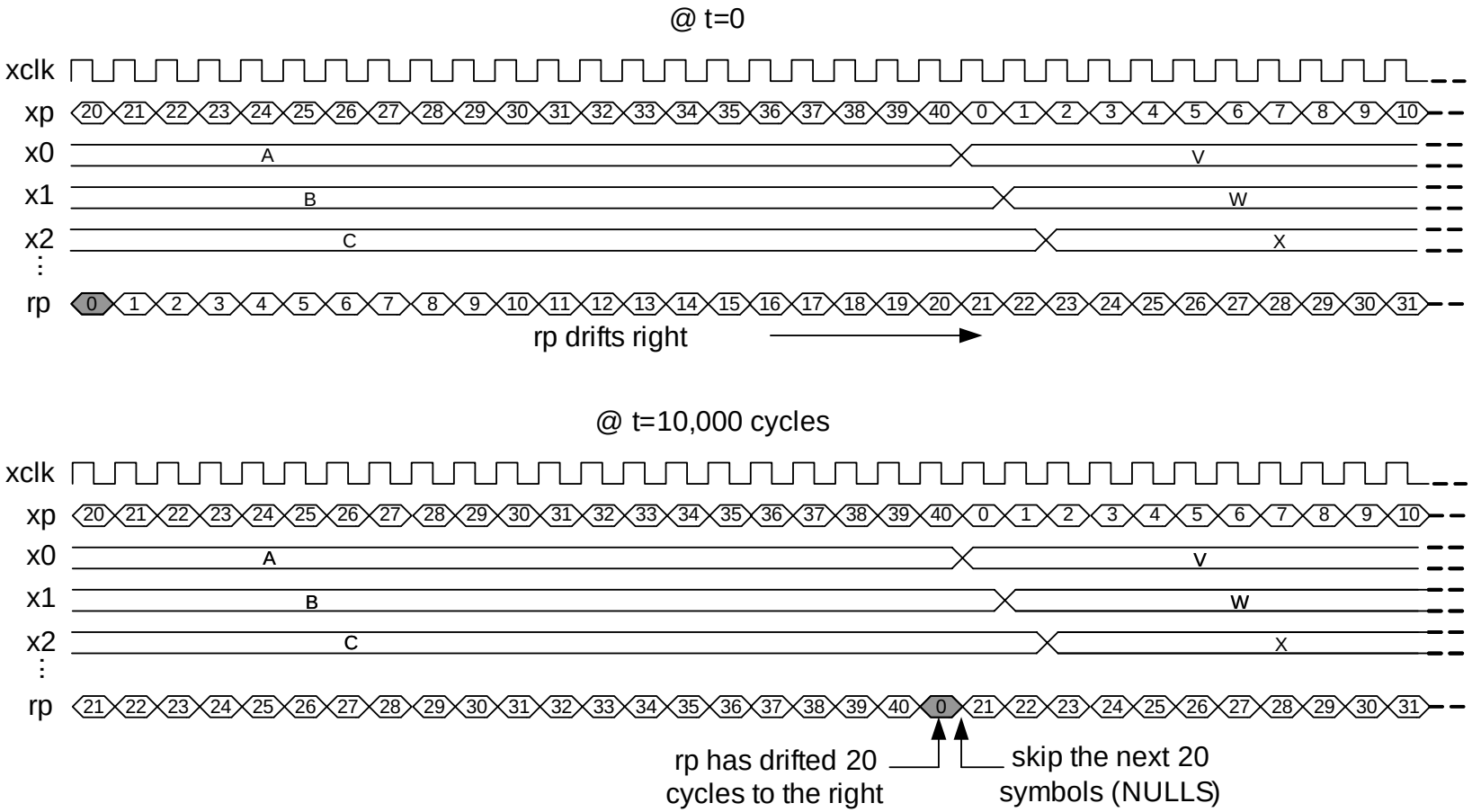
Example Synchronizer Design

- Designing an interface where clocks nominally match but might differ by as much as 1000ppm.
- Channel sends data in blocks of 10^4 symbols separated by fields of at least 20 NULL symbols

Example Synchronizer Design: rclk fast, xclk slow



Example Synchronizer Design: rclk slow, xclk fast

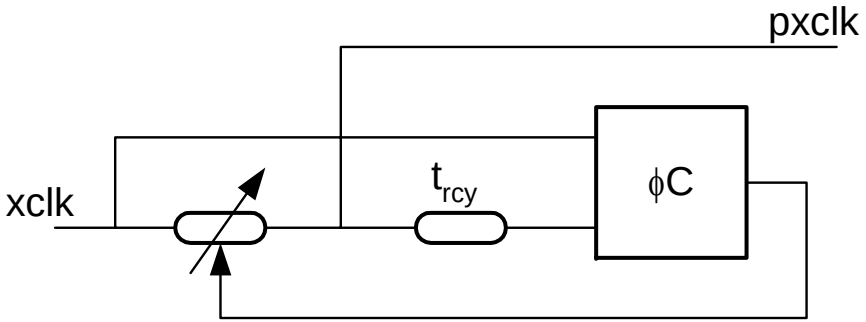


Periodic Timing

- Transmit and receive clocks are periodic but at unrelated frequencies
  - e.g., modules in a system operate off of separate oscillators with independent frequencies
  - case where one is rationally derived from the other is an interesting special case
- In this situation, a single synchronization won't last forever (like mesochronous) or even for a long time (like plesiochronous)
- However, we can still look into the future and predict clock conflicts far enough ahead to reduce synchronizer delay

Clock-Predictor Circuit

- Suppose we want to know the value of xclk, one rclk cycle (t_{rcy}) in the future
- This is just a phase shift of t_{xcy}-t_{rcy}
- It is easy to generate this phase shift using a simple timing loop
- Note that we could just as easily predict xclk several rclk cycles in the future
- So how do we build a synchronizer using this?



Asynchronous Timing

- Sometimes we need to sample a signal that is truly asynchronous
- We can still move the synchronization out of the datapath by using an asynchronous FIFO synchronizer
- However this still incurs a high latency on the full and empty signals as we have to wait for a brute force synchronizer to make its decision
- We can still avoid delay in this case if we don't *really* need to synchronize
  - often synchronization is just an expensive convenience