

6.205

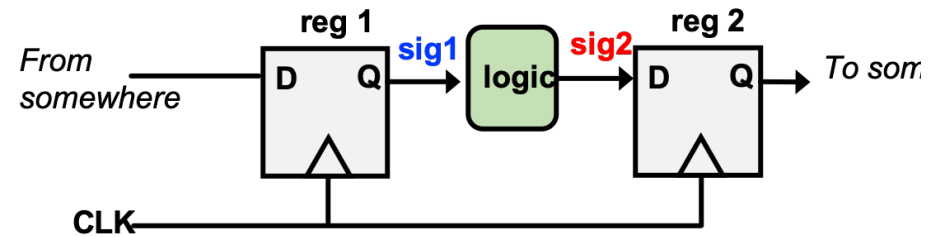
Sequential Logic III:

Sequential Logic Timing, Vivado, Metastability,
FSMs

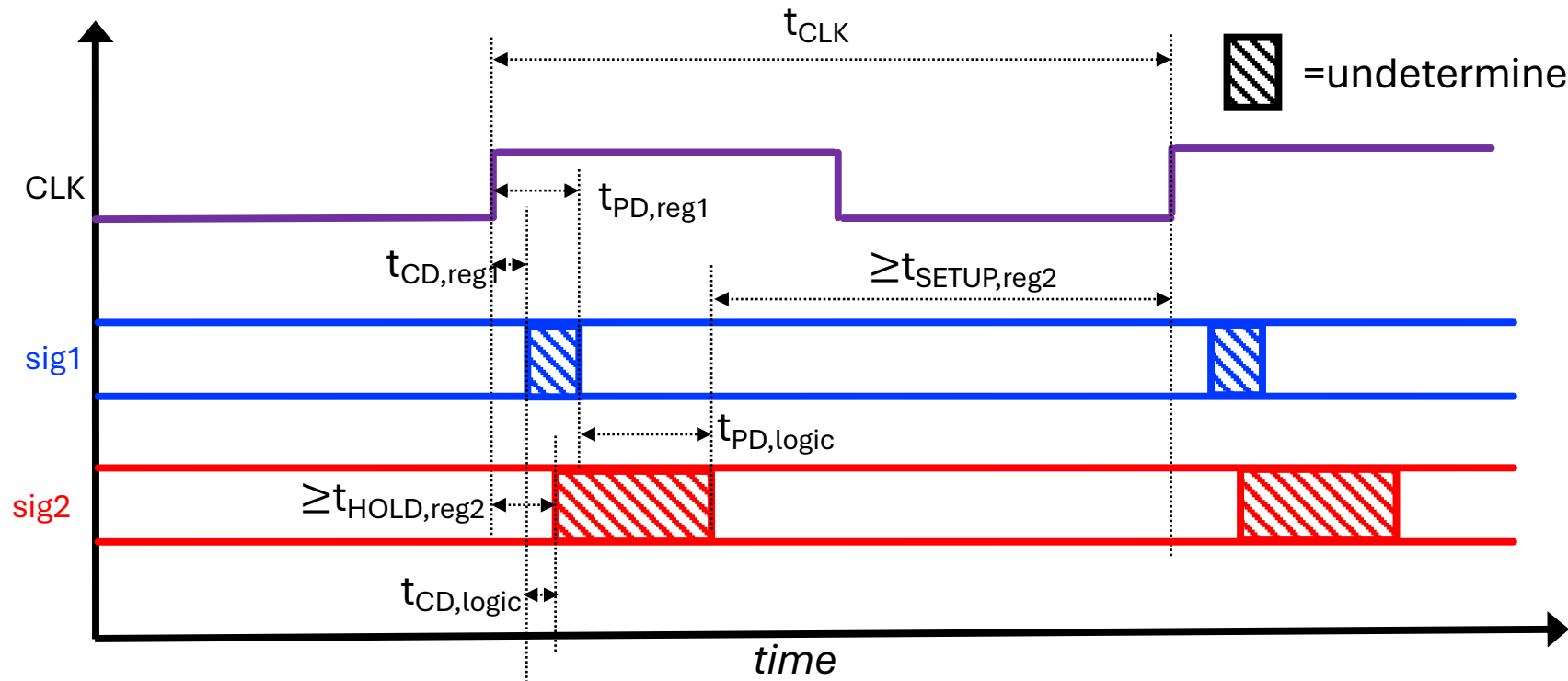
Administrivia

- Week 2 due last night.
- Week 3 out after class today
- Keysight Tech Talk/Recruiting
Tonight....Thursday, September 24 from 7:30
PM-9:00 PM in 4-231

Register-to-Register Timing



— = determined state
 = undetermined state



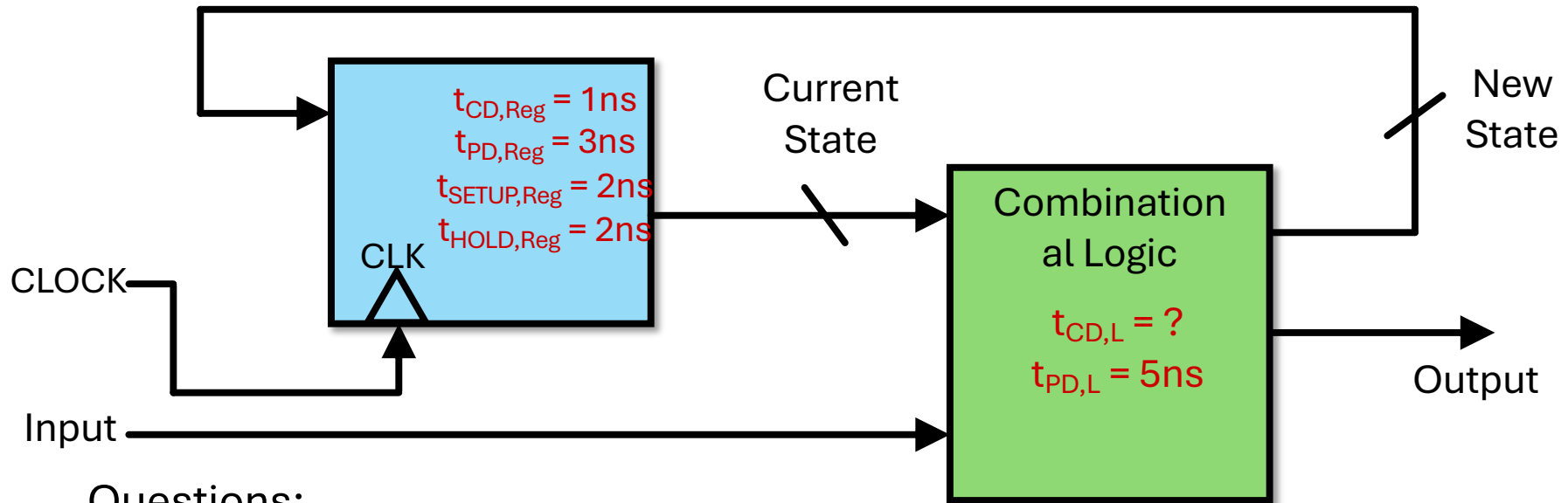
**Two Requirements/
Conclusions:**

$$t_{PD,reg1} + t_{PD,logic} + t_{SETUP,reg2} \leq t_{CLK}$$

$$t_{CD,reg1} + t_{CD,logic} \geq t_{HOLD,reg2}$$

Sequential Circuit Timing

Assume input is also coming from a clocked system

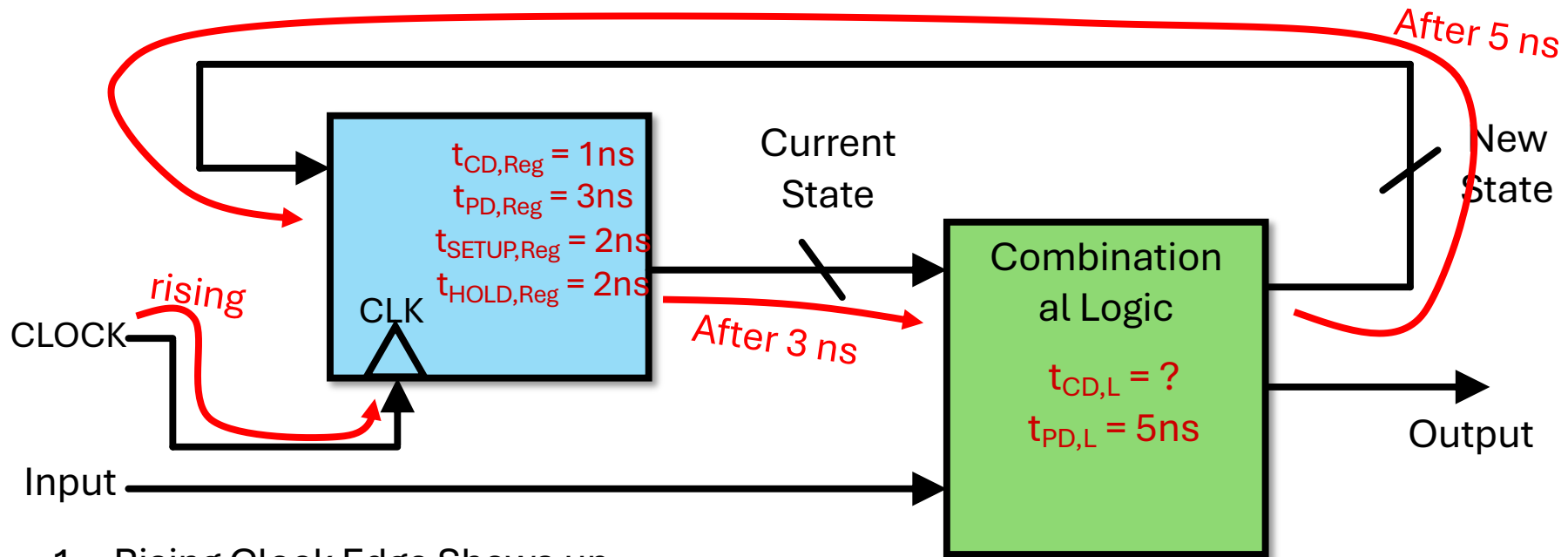


Questions:

- Minimum clock period?
- Constraints on $t_{CD,L}$?
- Setup, Hold times for entire system?
(Aka for the system input?)

Minimum Clock Period?

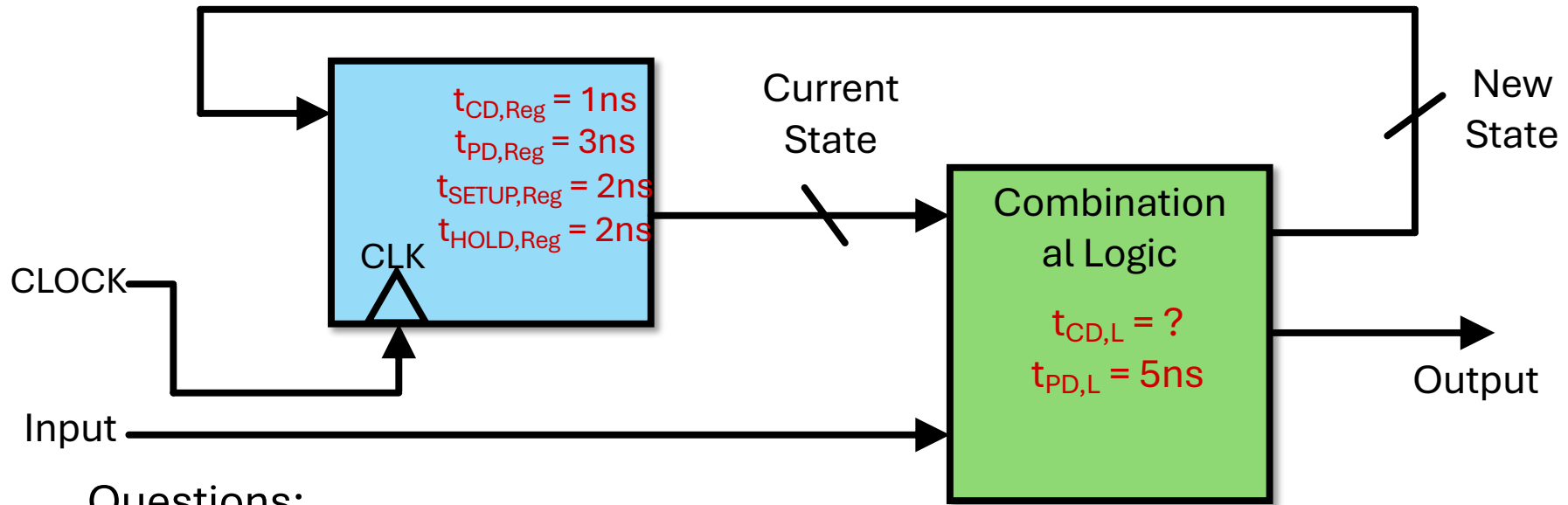
Assume input is also coming from a clocked system



1. Rising Clock Edge Shows up...
2. It is $t_{PD,Reg} = 3ns$ until flop has finalized changing
3. It is *then* an additional $t_{PD,L} = 5ns$ until logic has finalized changing and starts sending data back to flop
4. That change must be done at least $t_{SETUP,Reg} = 2ns$ before the next rising clock edge

Sequential Circuit Timing

Assume input is also coming from a clocked system

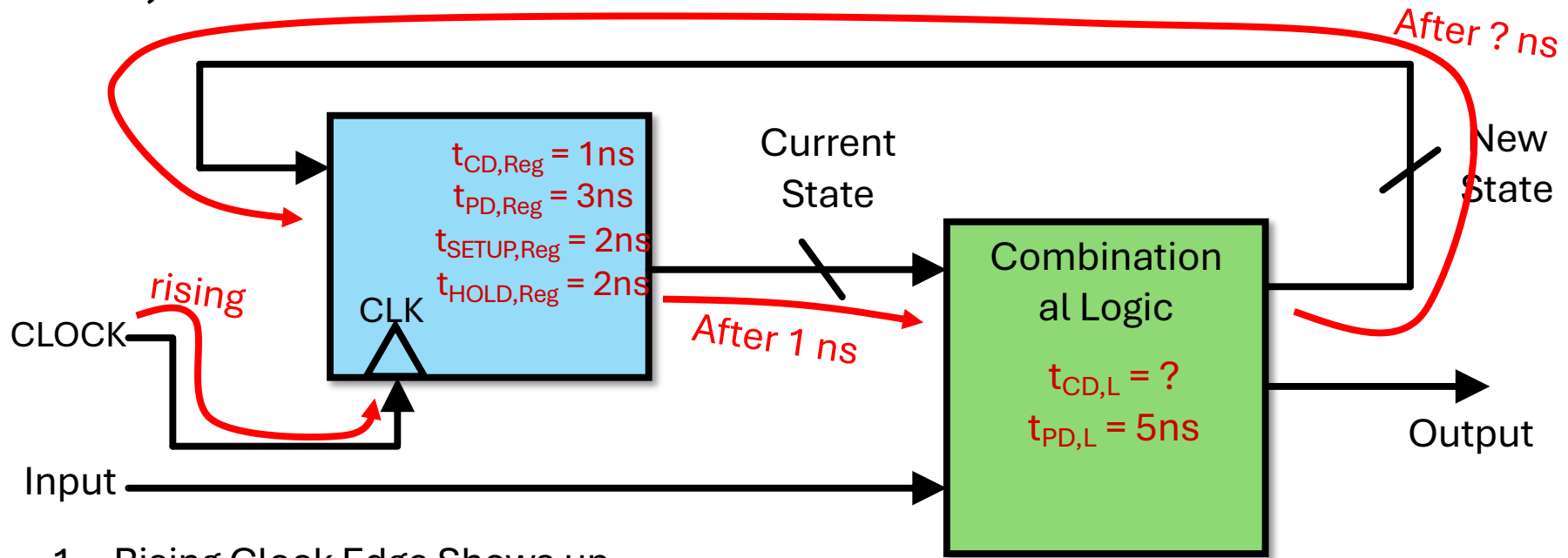


Questions:

- Minimum clock period?
- • Constraints on $t_{CD,L}$?
- Setup, Hold times for entire system?
(Aka for the system input?)

$t_{CD,L}$ Constraints?

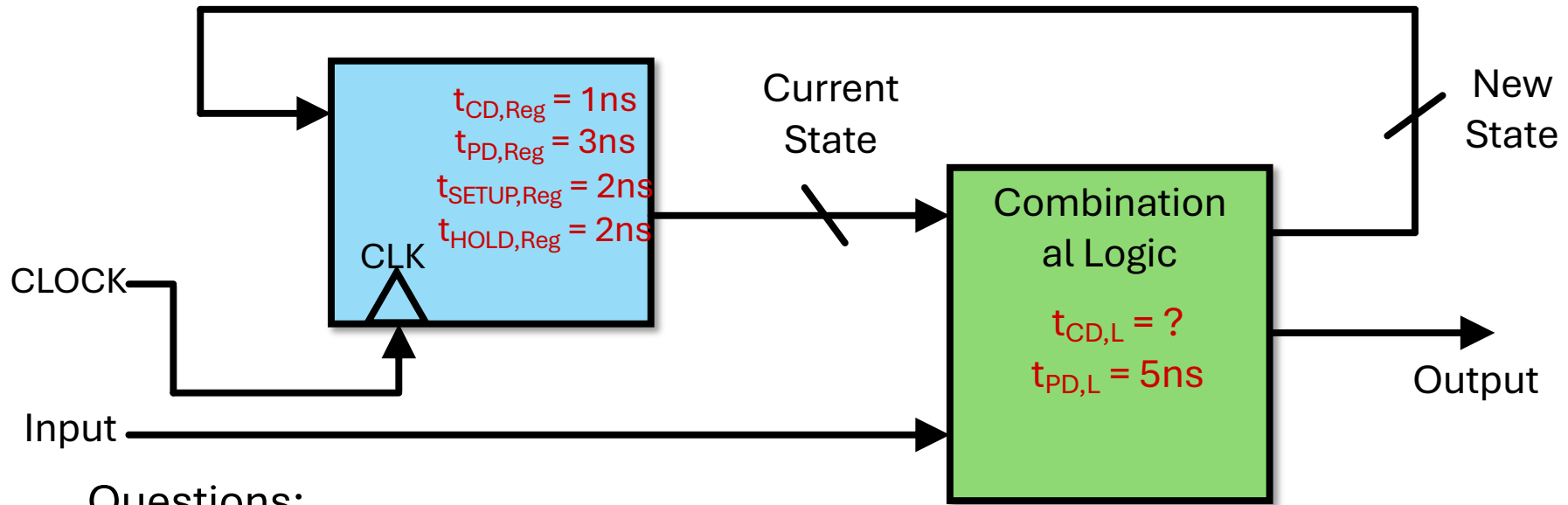
Assume input is also coming from a clocked system



1. Rising Clock Edge Shows up...
2. It is $t_{CD,Reg} = 1ns$ until flop output starts changing
3. It is then additional $t_{CD,L} = ?$ Until feedback wire starts changing
4. That change cannot be happening until at least $t_{HOLD,Reg} = 2ns$ has passed from clock edge

Sequential Circuit Timing

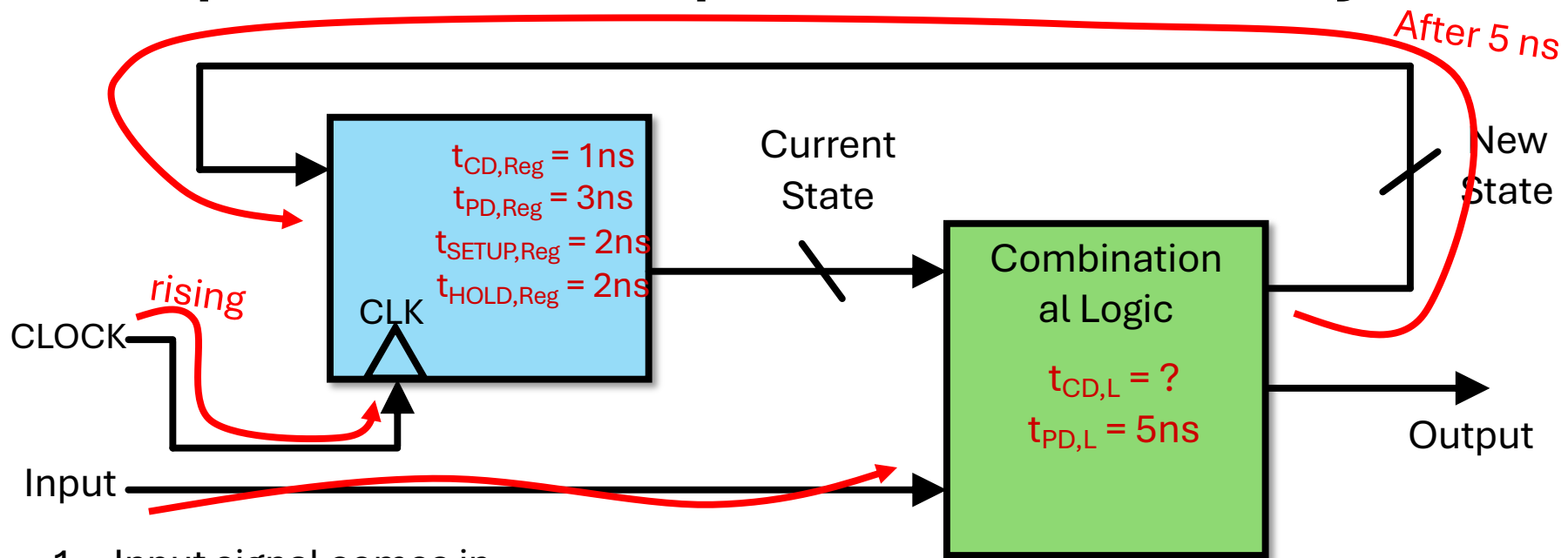
Assume input is also coming from a clocked system



Questions:

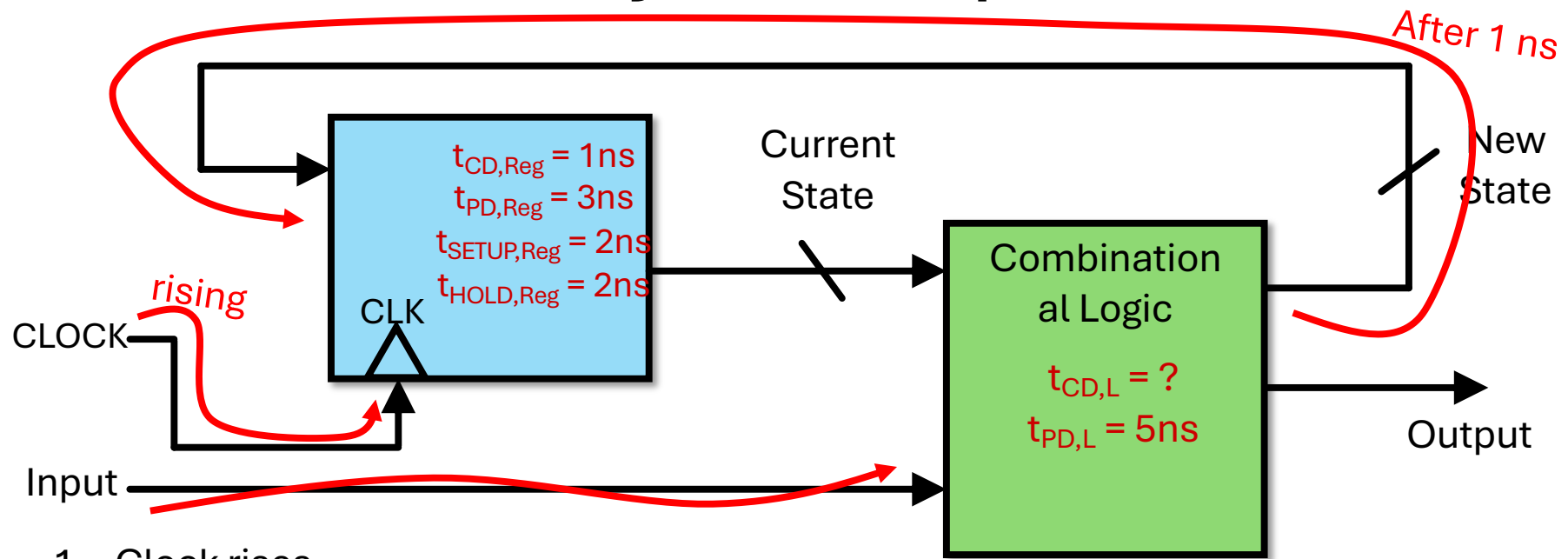
- Minimum clock period?
- Constraints on $t_{CD,L}$?
- • Setup, Hold times for entire system?
(Aka for the system input?)

Setup Time for Inputs of Whole System?



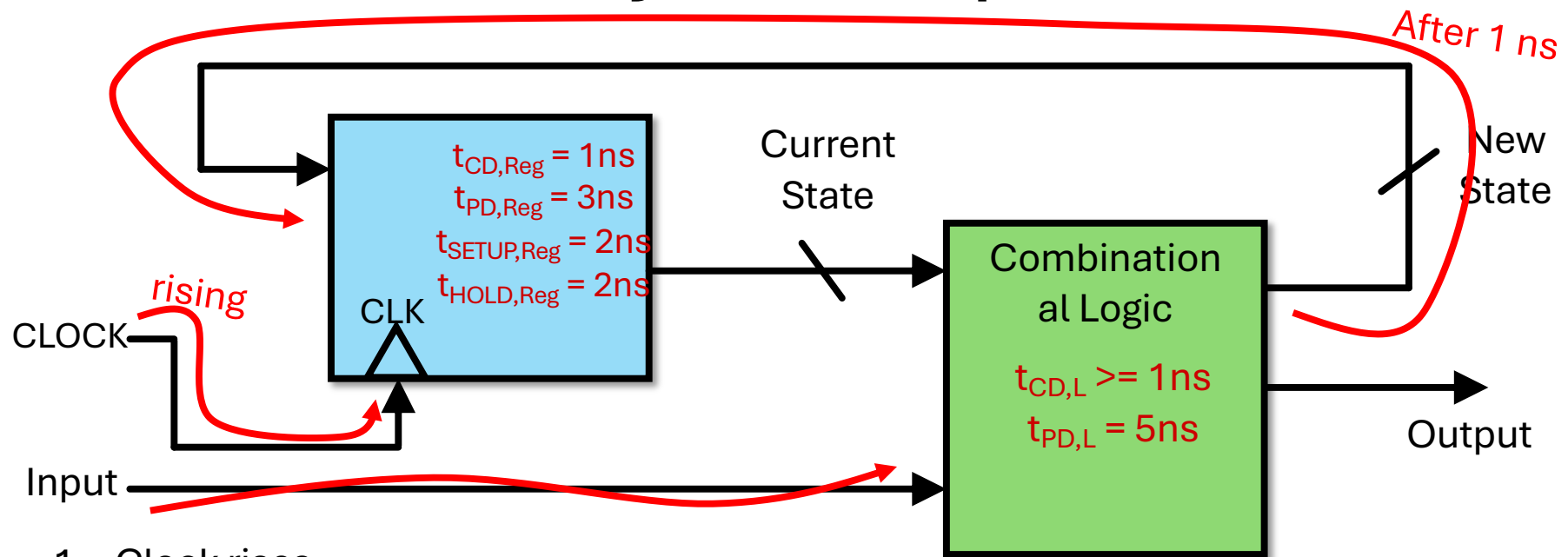
1. Input signal comes in
2. It is $t_{PD,L} = 5ns$ until comb logic has processed it and it is fed back...
3. That change *must be completed* at least $t_{SETUP,Reg} = 2ns$ before the next rising clock edge

Hold Time for System Input?



1. Clock rises
2. Input signal comes in
3. It is additional $t_{CD,L} = ?$ until comb logic starts to feed back any change...
4. That change must be done at least $t_{HOLD,Reg} = 2ns$ after the prior rising clock edge

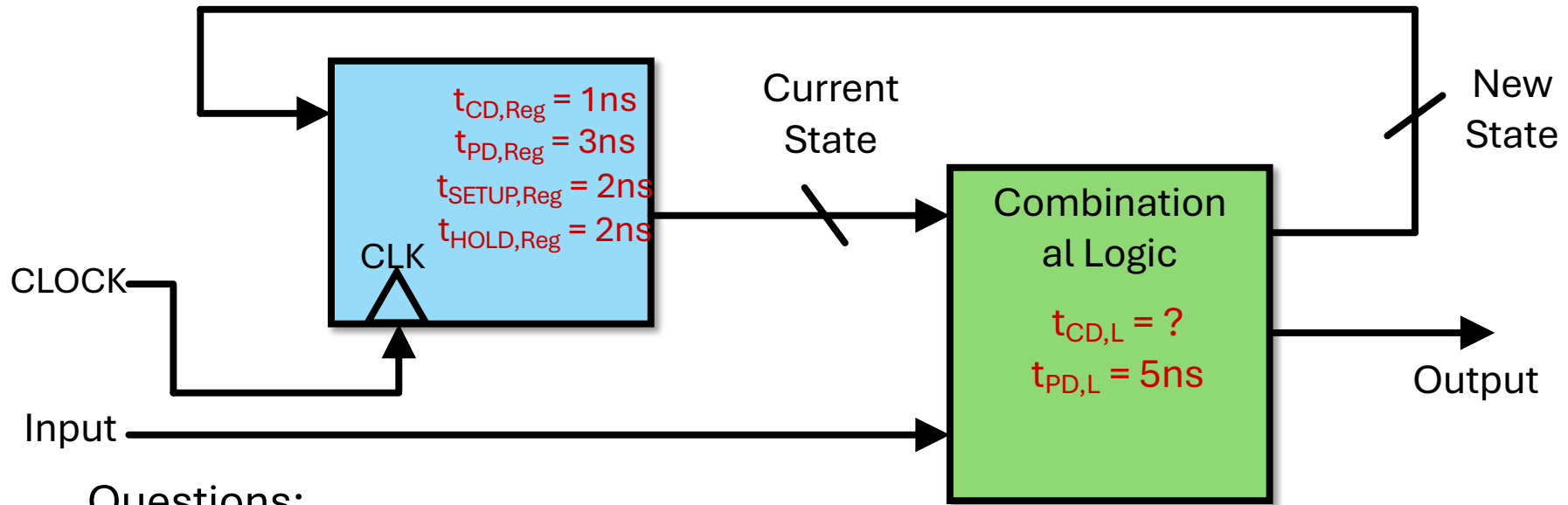
Hold Time for System Input?



1. Clock rises
2. Input signal comes in
3. It is additional $t_{CD,L} = 1ns$ (worst case of $\geq 1ns$) until comb logic starts ...
4. That change must be done at least $t_{HOLD,Reg} = 2ns$ after the prior rising clock edge

Sequential Circuit Timing

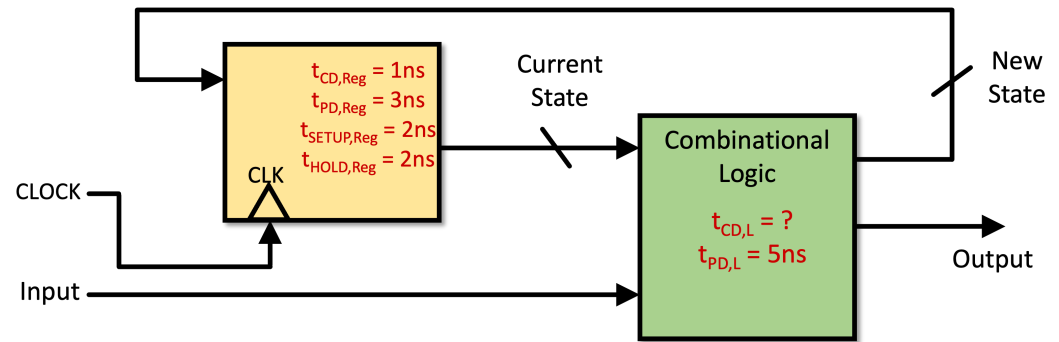
Assume input is also coming from a clocked system



Questions:

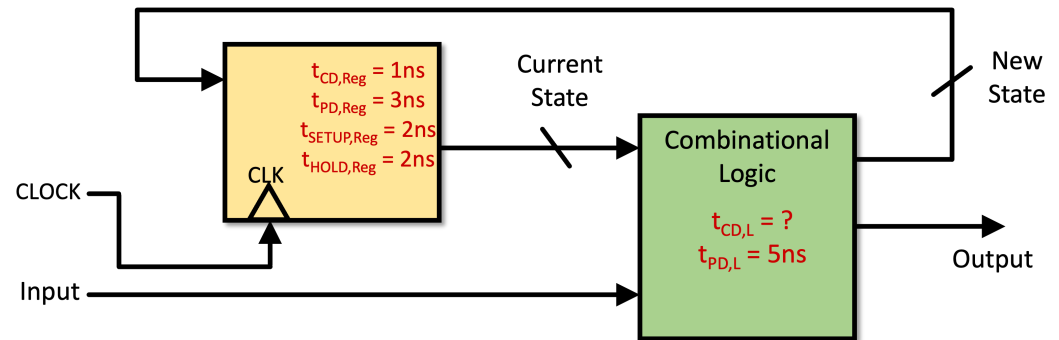
- Minimum clock period?
- Constraints on $t_{CD,L}$?
- Setup, Hold times for Inputs?

Conclusions I



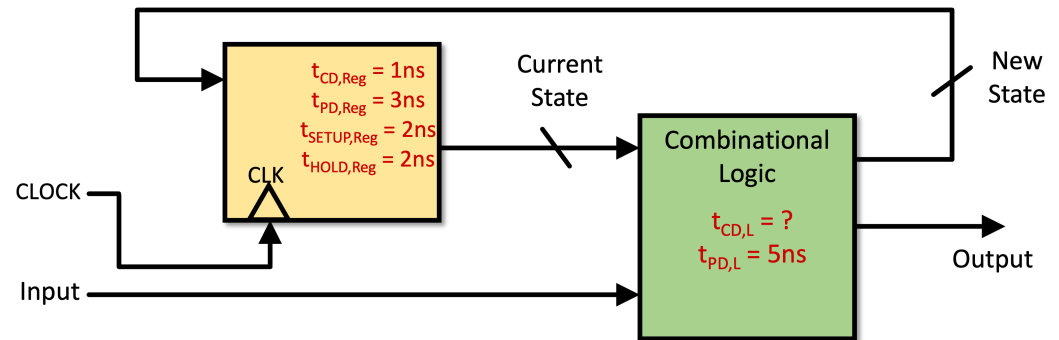
- Propagation delay, setup time, and clock all constrain each other!
- Contamination delay and hold time constrain one another

Conclusions II



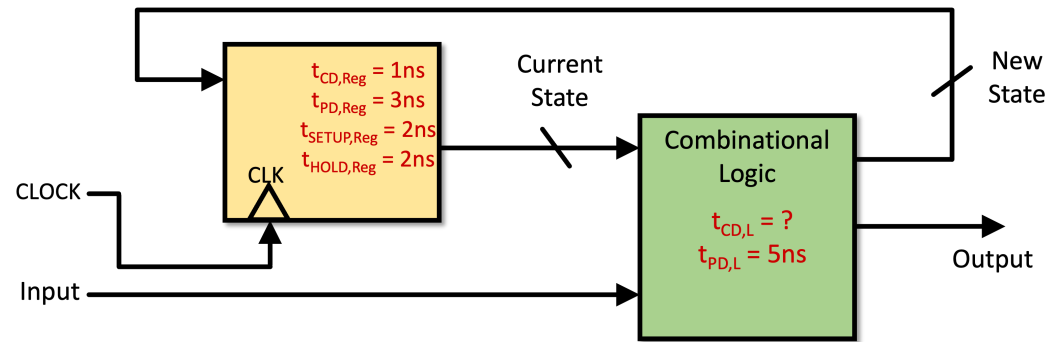
- The thicker the propagation delay of a piece of combinational logic, the higher your clock period needs to be to adhere to setup constraints

Conclusions III



- When driving a registered system with a signal, having combinational logic between it and the input to the register worsens the setup constraint on the system!
 - *Meaning signal at the input needs to be “ready” sooner than it would otherwise be relative to the clock edge compared to if you were driving the register directly.*

Conclusions IV



- When driving a registered system with a signal, having combinational logic between it and the input to the register may lessen the hold constraint
 - *Meaning a signal may be allowed to start to change sooner after the edge of the clock because it may be able to rely on some of the contamination delay of the combinational logic inside*

Why *Doesn't* Hold Time Mess with Clock Period?

$$t_{CD,reg1} + t_{CD,logic} \geq t_{HOLD,reg2}$$

- The hold constraint is simply related to how soon after a given clock edge that inputs can't change to another register.
- It does not have anything to say about the upcoming clock edge
- Since it doesn't relate two clock edges, it has nothing to say about clock edge periodicity or “clock period”

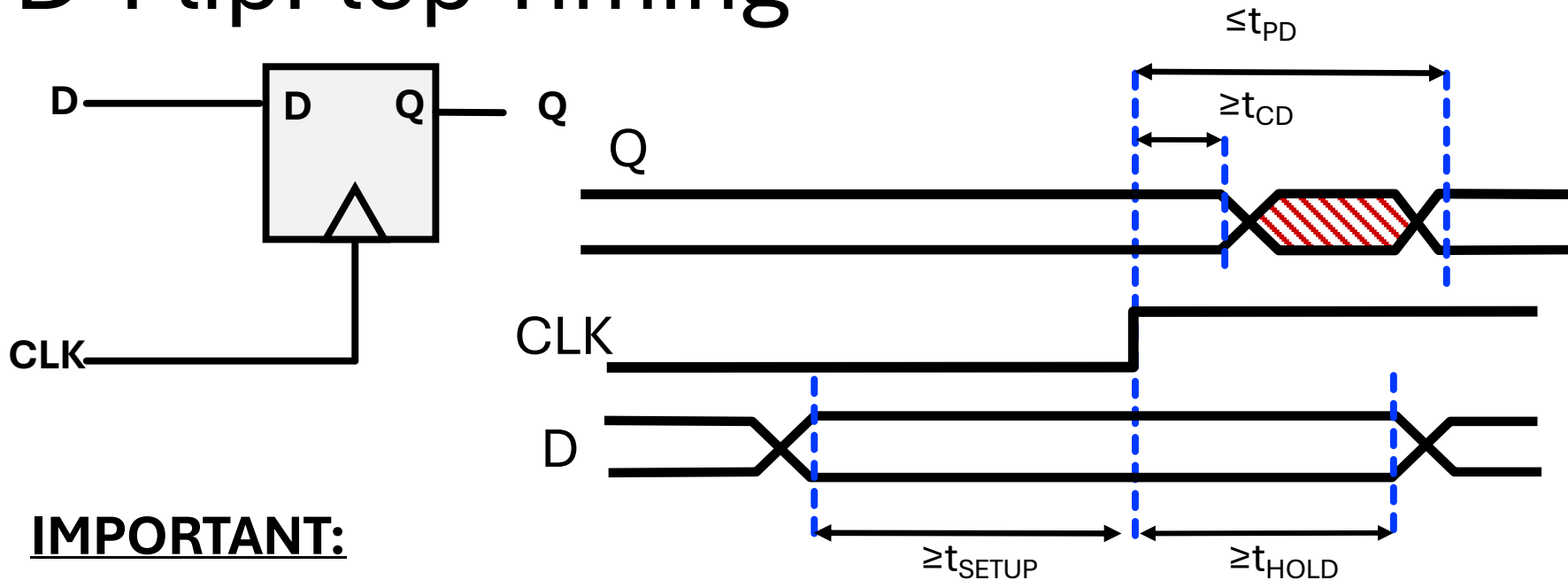
Why *Does* Setup Time Mess with Clock Period?

$$t_{PD,reg1} + t_{PD,logic} + t_{SETUP,reg2} \leq t_{CLK}$$

- When a clock edge rises, the output of a reg will start calculating and generating a new output ($t_{PD,reg1}$).
- Once that is stable, intermediate logic further slows getting a final result ($t_{PD,logic}$).
- That result must be present $t_{SETUP,reg2}$ ahead of the *next* clock edge.
- This relates two clock edges, so constrains clock edge periodicity...or clock period

D-FlipFlop Timing

 =undetermined state



IMPORTANT:

t_{PDr} : maximum propagation delay, @posedge CLK $D \rightarrow Q$
Maximum time it takes for Q to change after rising edge of CLK

t_{CDr} : minimum contamination delay, @posedge CLK $D \rightarrow Q$
Minimum time it takes for Q to start to change after rising edge of CLK

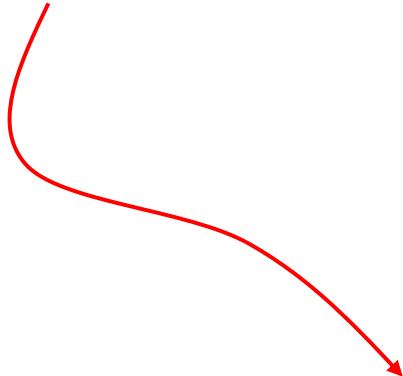
t_{SETUP} : setup time
 How long D input must be stable **before** the rising edge of CLK

t_{HOLD} : hold time
 How long D input must be stable **after** the rising edge of CLK

*How it
behaves*

*What it
needs from
us/the rest
of the circuit*

Cool/Not Cool Bug

- What's wrong with this
- 

```
module simple_counter( input wire clk,  
                      input wire rst,  
                      input wire evt,  
                      output logic[15:0] count  
                      );  
    always_ff @(posedge clk) begin  
        if (rst) begin  
            count <= 16'b0;  
        end else if (evt) begin  
            count <= count + 1;  
        end  
    end  
endmodule
```

```
simple_counter msc(  
    .clk(clk),  
    .rst(counter_reset),  
    .evt(clk),  
    .count(count));
```

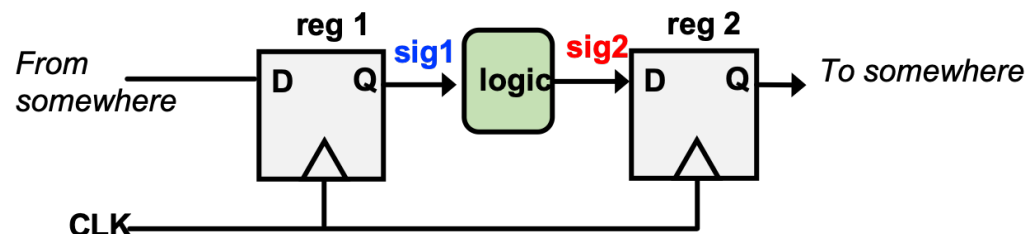
***evt can't *not* violate setup
and hold***

Clocks Stay with Clocks!

- And Signals stay with signals. None of this stuff:

```
always_ff @(posedge some_button_thing)begin
    x <= 5+x;
    //other code
end
```

This will not play nice with other circuits that are clocked off an actual clock! All those equations we've been talking about will not work...



Goal: Use as few clock domains as possible

Suppose we have a clk at frequency f and we want periodic signals at $f/2$, $f/4$, $f/8$, etc.:????

```
logic clk2,clk4,clk8,clk16;

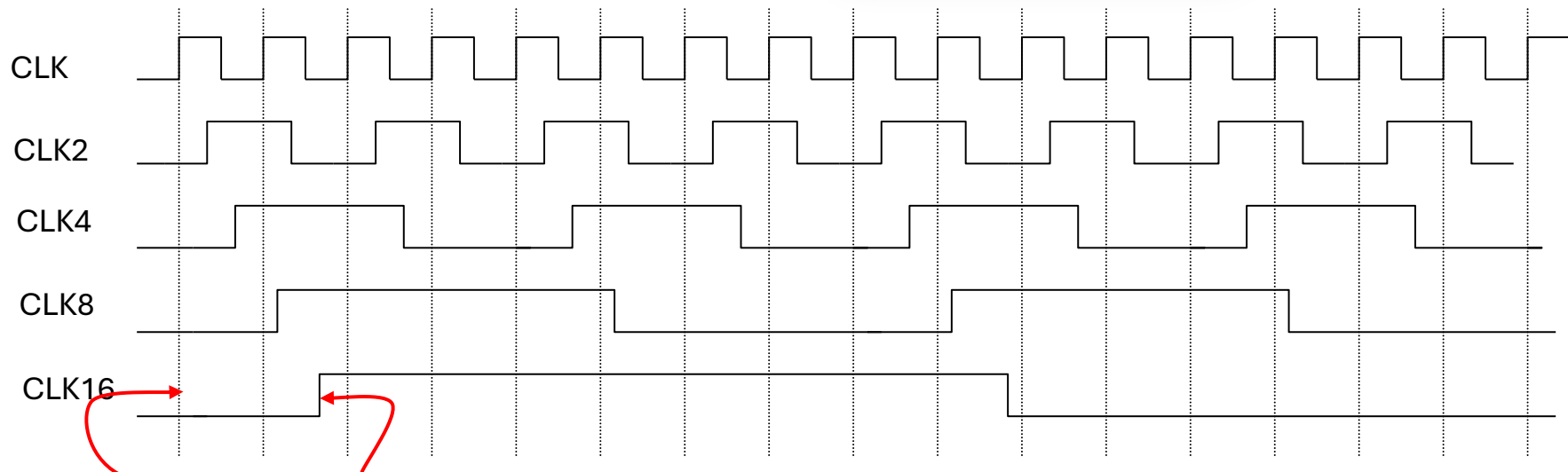
always_ff @(posedge clk)begin
|   clk2 <= ~clk2;
end
always_ff @(posedge clk2)begin
|   clk4 <= ~clk4;
end
always_ff @(posedge clk4)begin
|   clk8 <= ~clk16;
end
always_ff @(posedge clk8)begin
|   clk16 <= ~clk16;
end
```

Goal: use as few clock domains as possible

Suppose we have a reference clk at frequency f and we want signals at $f/2$, $f/4$, $f/8$, etc.:????

```
logic clk2,clk4,clk8,clk16;  
  
always_ff @(posedge clk)begin  
    clk2 <= ~clk2;  
end  
always_ff @(posedge clk2)begin  
    clk4 <= ~clk4;  
end  
always_ff @(posedge clk4)begin  
    clk8 <= ~clk8;  
end  
always_ff @(posedge clk8)begin  
    clk16 <= ~clk16;  
end
```

No! don't do it that way, you'll never make a dime.



Very hard to have synchronous communication between clk and clk16 domains... Can lead to lots of timing violations!

Goal: Use as few clock domains as possible

Suppose we have a reference clk at frequency f and we want signals at $f/2$, $f/4$, $f/8$, etc.:????

```
logic [3:0] count;
always_ff @(posedge clk) begin
    count <= count + 1;    // counts 0..15
end
logic enb2, enb4, enb8, enb16;
assign enb2 = (count[0] == 1'b1);
assign enb4 = (count[1:0] == 2'b11);
assign enb8 = (count[2:0] == 3'b111);
assign enb16 = (count[3:0] == 4'b1111);

always_ff @(posedge clk) begin
    if (enb2) begin
        // get here every 2nd cycle
    end
end
```

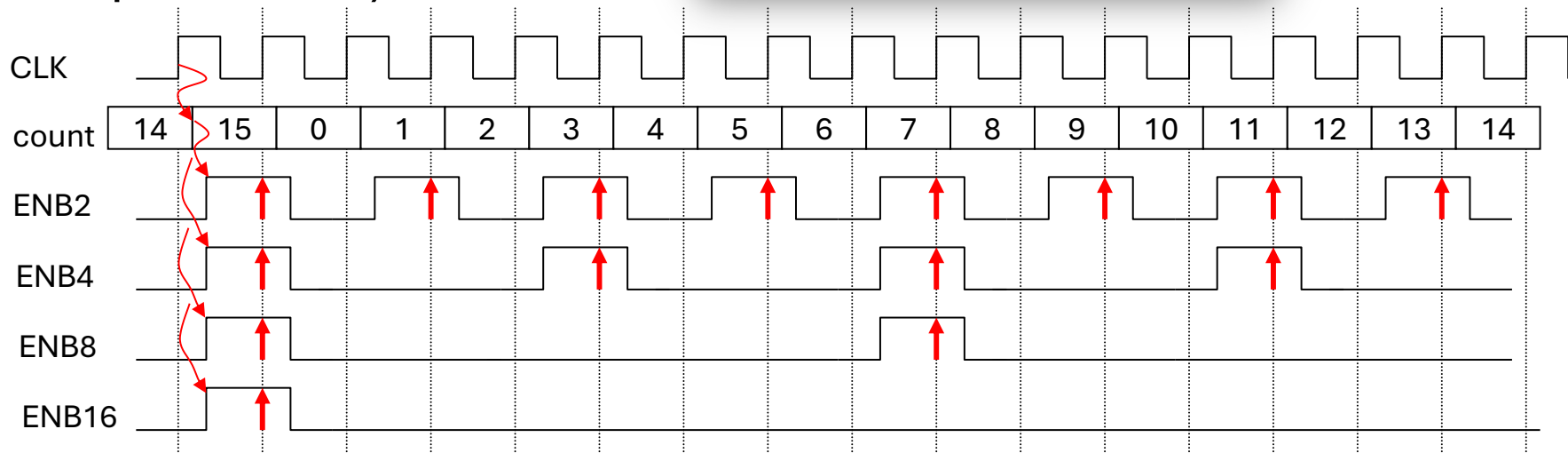
Solution: One clock, Many enables

- Use one (high speed) clock, but create enable signals to select a subset of the edges to use for a particular piece of sequential logic (much easier on timing requirements)

```
logic [3:0] count;
always_ff @(posedge clk) begin
    count <= count + 1; // counts 0..15
end
logic enb2, enb4, enb8, enb16;
assign enb2 = (count[0] == 1'b1);
assign enb4 = (count[1:0] == 2'b11);
assign enb8 = (count[2:0] == 3'b111);
assign enb16 = (count[3:0] == 4'b1111);

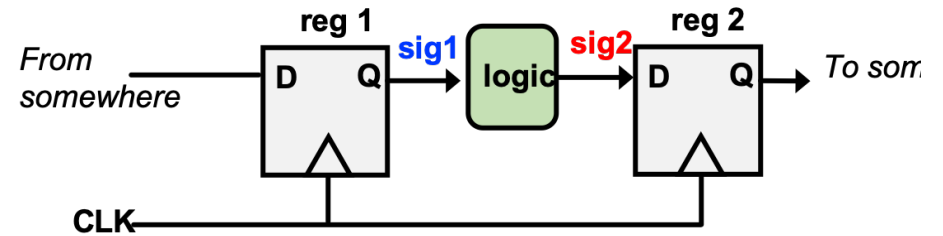
always_ff @(posedge clk) begin
    if (enb2) begin
        // get here every 2nd cycle
    end
end
```

Yes! A good idea that will lead to good outcomes and happiness

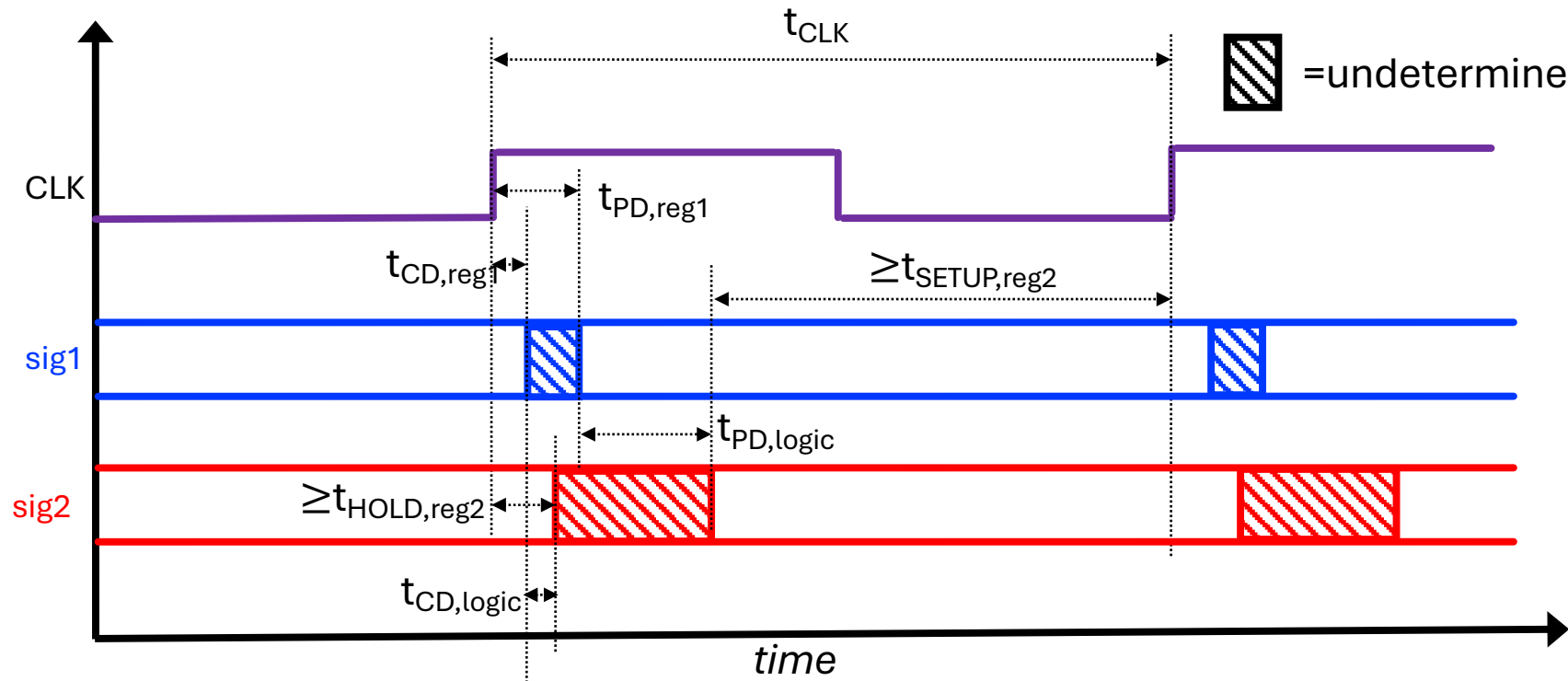


= clock edge selected by enable signal

Register-to-Register Timing



— = determined state
 = undetermined state



**Two Requirements/
Conclusions:**

$$t_{PD,reg1} + t_{PD,logic} + t_{SETUP,reg2} \leq t_{CLK}$$

$$t_{CD,reg1} + t_{CD,logic} \geq t_{HOLD,reg2}$$

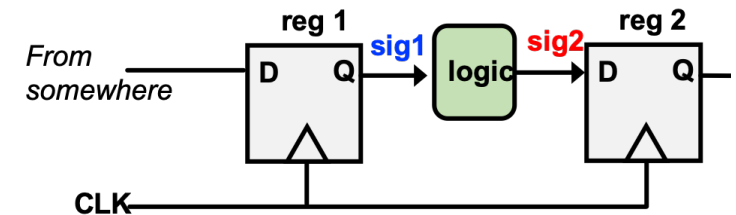
Now...how do you actually design within that?

- Inequalities are given to us from on high
- But we are beset on all sides by temptation to violate them
 - “Nobody is looking surely I can add one more LUT.”

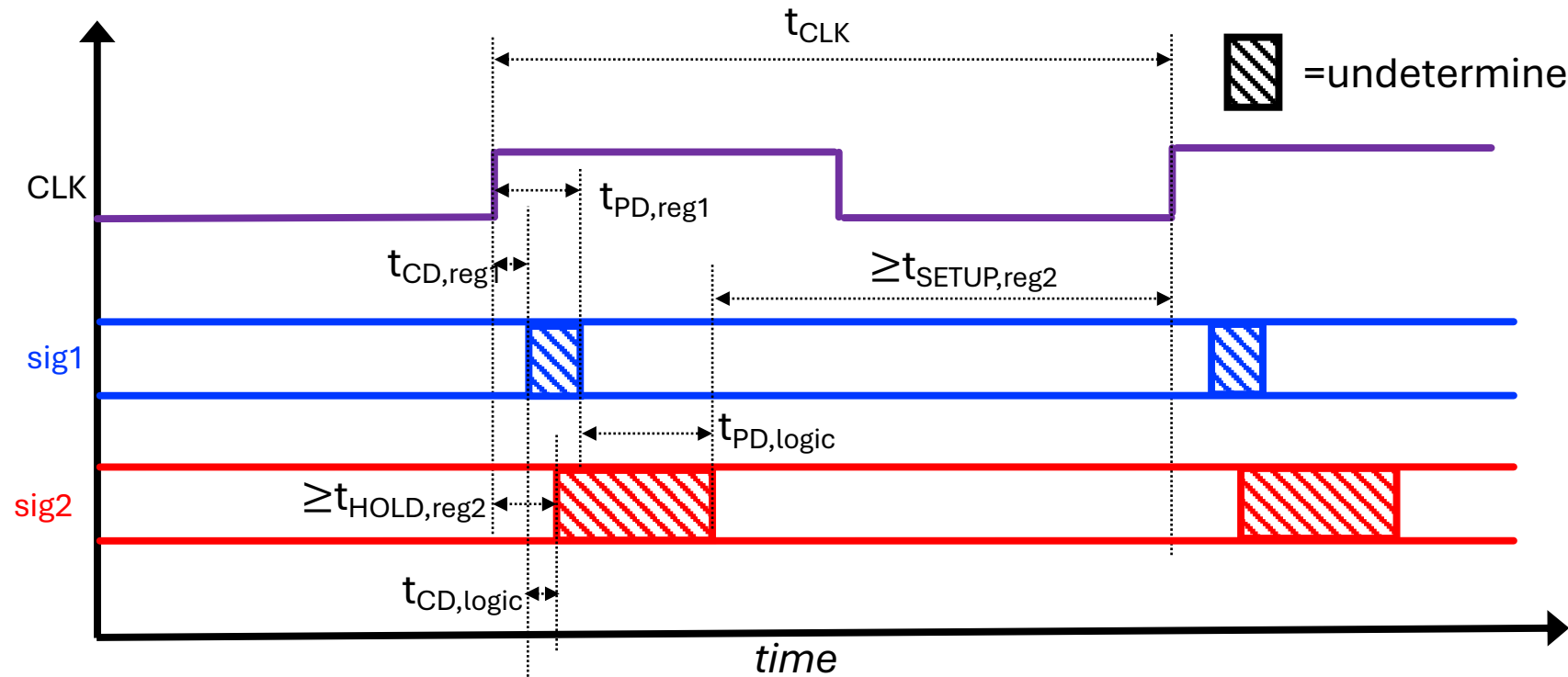


Timing Diagram

This thing again. Unbelievable, it is almost like this diagram is important or something



— =determined state
 =undetermined state



**Two Requirements/
Conclusions:**

$$t_{PD,reg1} + t_{PD,logic} + t_{SETUP,reg2} \leq t_{CLK}$$

$$t_{CD,reg1} + t_{CD,logic} \geq t_{HOLD,reg2}$$

What is Slack?

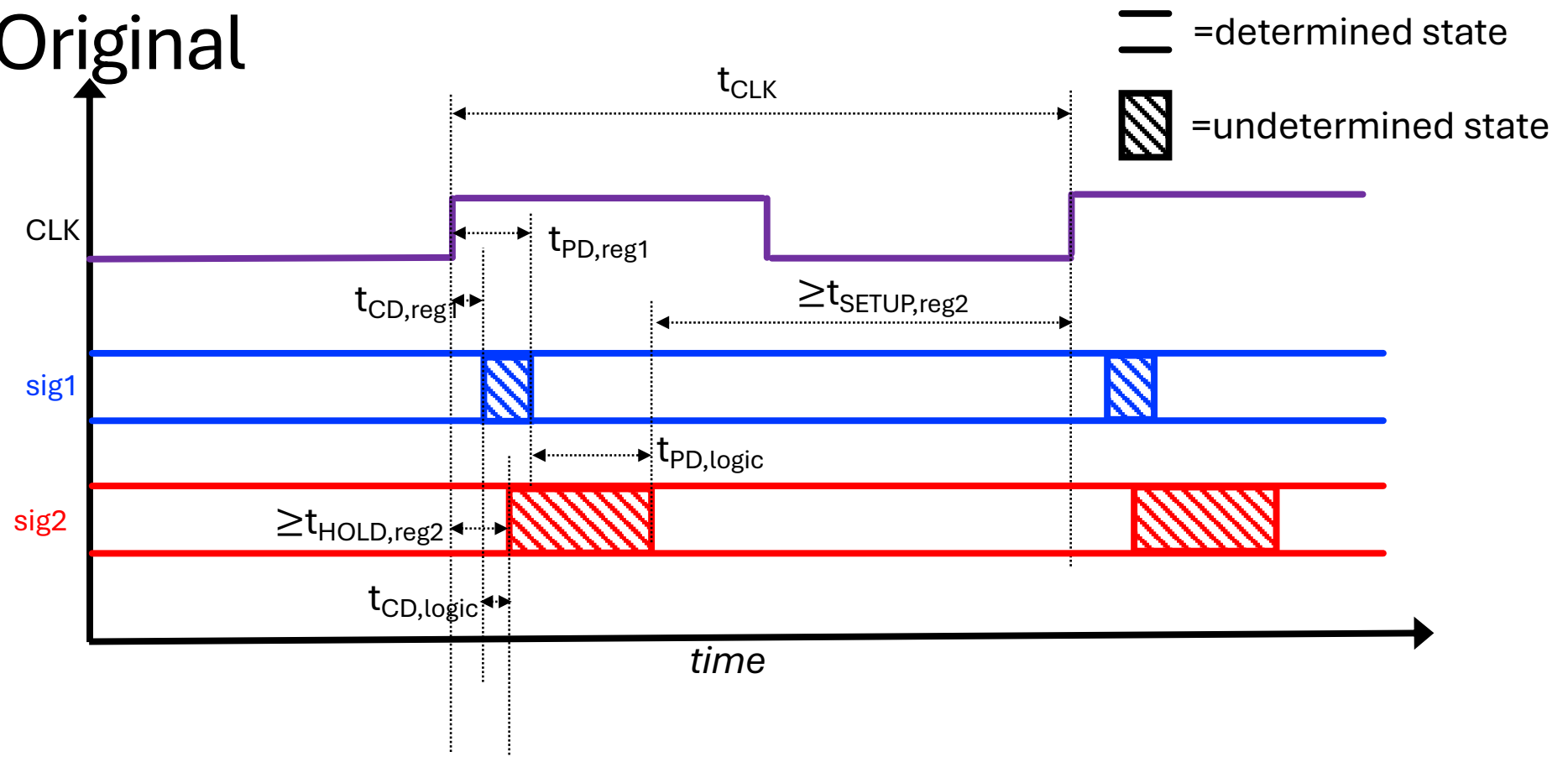
- It is what Jack cuts for you upon request
- Slack: measure of how safe your timing is
- The two big timing constraints we worry about are related to setup and hold
- Therefore there are two Slack values:
 - Setup slack: $t_{\text{required}} - t_{\text{actual}}$
 - Hold slack: $t_{\text{actual}} - t_{\text{required}}$

These are defined such that:

*Positive is **GOOD** 😊😊😊😊*

*Negative is **BAD** 😞😞😞😞*

Original

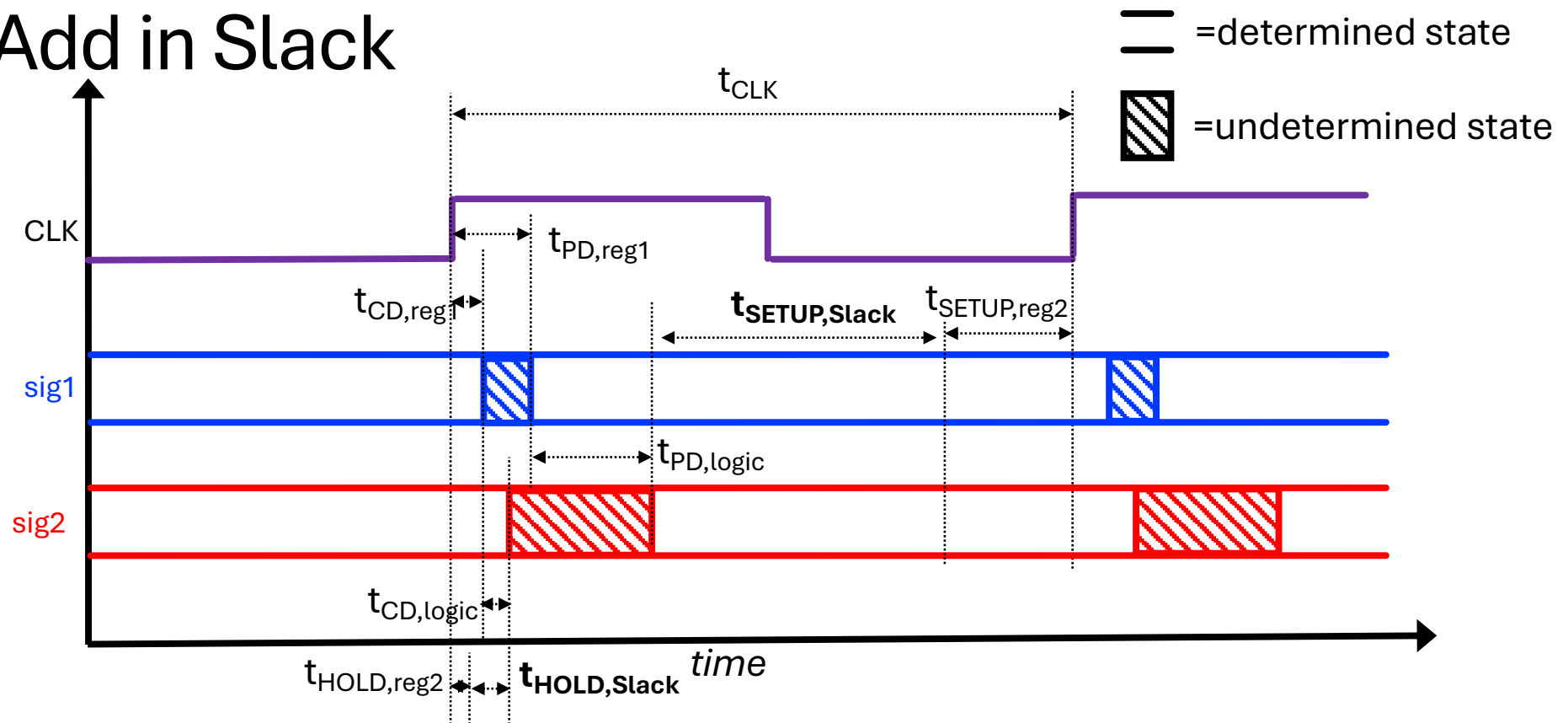


**Two Requirements/
Conclusions:**

$$t_{PD,reg1} + t_{PD,logic} + t_{SETUP,reg2} \leq t_{CLK}$$

$$t_{CD,reg1} + t_{CD,logic} \geq t_{HOLD,reg2}$$

Add in Slack



$$t_{PD,reg1} + t_{PD,logic} + t_{SETUP,reg2} + t_{SETUP,Slack} = t_{CLK}$$

$$t_{CD,reg1} + t_{CD,logic} = t_{HOLD,reg2} + t_{HOLD,Slack}$$

Equations* $t_{SETUP,Slack} = t_{CLK} - (t_{PD,reg1} + t_{PD,logic} + t_{SETUP,reg2})$

$$t_{HOLD,Slack} = t_{CD,reg1} + t_{CD,logic} - t_{HOLD,reg2}$$

Conclusion

- Positive Slack is ***GOOD***
- Negative Slack is ***BAD***
- **Slack becomes the number that Vivado searches over and optimizes for.**

How to See Your Slack?

- Open up vivado.log and search for the *first instance of WNS* (“**worst negative slack**”)

Initial estimate of timing based off of just the physical primitives and minimal guess at wiring and layout

Phase 1 Physical Synthesis Initialization

```
INFO: [Physopt 32-721] Multithreading enabled for phys_opt_design using a maximum of 4 CPUs  
INFO: [Physopt 32-619] Estimated Timing Summary | WNS=3.962 | TNS=0.000 |  
Phase 1 Physical Synthesis Initialization | Checksum: 140305231
```

Noice. This is positive. This means the “worst” slack in the entire design is 3.926 ns...clock period is 10 ns....we got plenty of room

A summation of the negative slack of all offending paths. 0 if all good.

Scroll further down

Phase 2 Router Initialization

Phase 2.1 Fix Topology Constraints

Phase 2.1 Fix Topology Constraints | Checksum: 1f7e5faf6

Time (s): cpu = 00:00:07 ; elapsed = 00:00:06 . Memory (MB): peak = 3036.672 ; gain = 0.000 ; free physical = 11693 ; free virtual = 36164

Phase 2.2 Pre Route Cleanup

Phase 2.2 Pre Route Cleanup | Checksum: 1f7e5faf6

Time (s): cpu = 00:00:07 ; elapsed = 00:00:06 . Memory (MB): peak = 3036.672 ; gain = 0.000 ; free physical = 11693 ; free virtual = 36164
Number of Nodes with overlaps = 0

Phase 2.3 Update Timing

Phase 2.3 Update Timing | Checksum: 229dba4e6

Time (s): cpu = 00:00:07 ; elapsed = 00:00:06 . Memory (MB): peak = 3036.672 ; gain = 0.000 ; free physical = 11693 ; free virtual = 36164
INFO: [Route 35-416] Intermediate Timing Summary | WNS=3.968 | TNS=0.000 | WHS=-0.118 | THS=-1.636 |

- Further down you'll see the timing at the start of routing and placing.
 - WNS refers to “worst negative setup slack”
 - WHS refers to “worst negative hold slack”
 - TNS and THS are the sums of all negative respective paths

When starting routing...

Intermediate Timing Summary WNS=3.968 TNS=0.000 WHS=-0.118 THS=-1.636

- We can see WHS is negative and THS is non-zero.
- Not surprising, more realistic view of situation is now at hand. WNS is fine, but WHS is negative...so Vivado will start playing games with:
 - Placement, Routing, Clocking...
- In order to add in some contamination delay

Keep monitoring further down the logs

Phase 7 Post Hold Fix

Phase 7.1 Hold Fix Iter

INFO: [Route 35-416] Intermediate Timing Summary | WNS=4.071 | TNS=0.000 | WHS=0.166 | THS=0.000 |

Phase 7.1 Hold Fix Iter | Checksum: 2789c1f2d

- After some work, it'll start getting everything positive (and 0 for the TNS, THS arguments)
- Then it'll be happy and end the build. The final will be:

Phase 12 Post Router Timing

INFO: [Route 35-20] Post Routing Timing Summary | WNS=4.104 | TNS=0.000 | WHS=0.166 | THS=0.000 |

Final Numbers are What matters

- Even if your WNS and WHS start out negative, it does not mean, the place-and-route/implementation steps will not solve the problem...in fact they almost always do have to come in and make things better
- The synthesis-stage estimates are often on the conservative side too...
- Most designs push the limits so that's where the majority of your build will spend its time.

Understand the Goal

- There are already safety margins built into the setup and hold and all the other numbers.
- The goal is to get positive on WNS, WHS, TNS, THS...that's it
- There is no reward for having a more positive negative slack than somebody else.* If you're positive, you're good.

Positive is Positive

Phase 25 Post Router Timing

INFO: [Route 35-20] Post Routing Timing Summary | WNS=0.076 | TNS=0.000 | WHS=0.061 | THS=0.000 |

- Here's a different build for something else. 76 and 61 picoseconds over the line.
- This one took many more iterations (phase 25 vs phase 12 earlier), so our friend Vivado had to put in some work, but it did it.
- Just like when one of your TAs PNR'ed 7.016 their senior spring despite my protestations...just get it over the pass line and Timtim the Beaver will still hand you that diploma

And Vivado is not a wizard

- It can only fix your messes so much.

Let's Look at Some Code:

- Very Simple top_level:
- Use sw[15:0] and buttons to seed two values into 16 bit registers:

```
`timescale 1ns / 1ps
`default_nettype none

module top_level(
    input wire clk_100mhz, //clock @ 100 mhz
    input wire [15:0] sw, //switches
    input wire [3:0] btn, //all four momentary button switches
    output logic [15:0] led //just here for the funs
);

    logic [3:0] old_btn;
    logic [15:0] quotient;
    logic [15:0] dividend;
    logic [15:0] divisor;
    assign led = quotient;

    always_ff @(posedge clk_100mhz)begin
        for (int i=0; i<4; i=i+1)begin
```

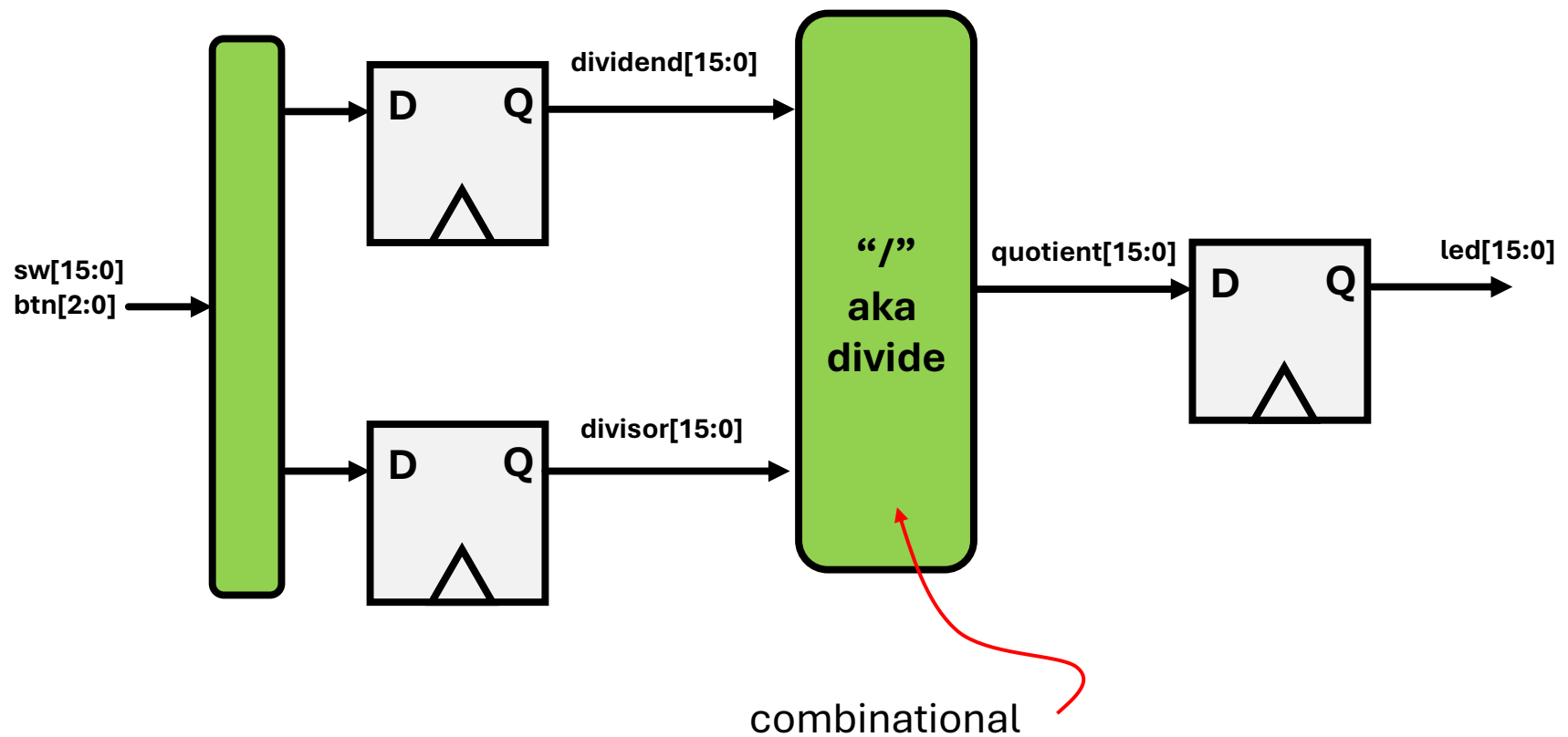
quotient <= dividend/divisor; //divide

- When btn[0] is pushed:
 - DIVIDE the 16 bit numbers

```
        if (btn[0] & ~old_btn[0])begin
            quotient <= dividend/divisor; //divide
        end
        if (btn[1] & ~old_btn[1])begin
            dividend <= sw; //divide //load dividend
        end
        if (btn[2] & ~old_btn[2])begin
            divisor <= sw; //divide //load dividend
        end
    end
endmodule

`default_nettype wire
```

What does this build???

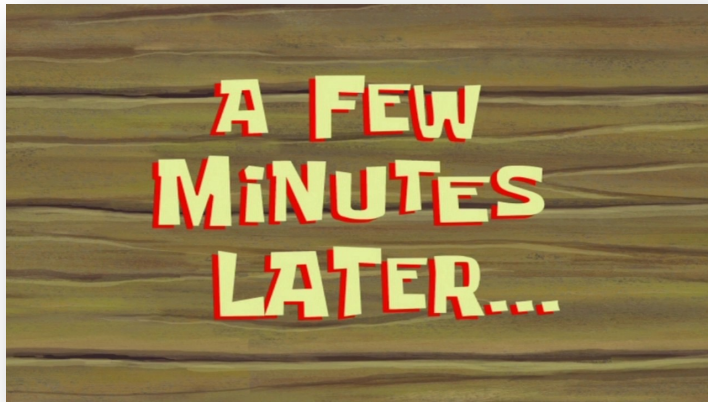


Let's Build it.

- Terminal Output:

```
jodalyst@Josephs-MBP lec06 % ./remote/r.py build.py build.tcl hdl/* xdc/* obj  
  
...  
...  
...  
  
Writing bitstream obj/final.bit...  
INFO: [Vivado 12-1842] Bitgen Completed Successfully.  
INFO: [Project 1-1876] WebTalk data collection is mandatory when using a ULT device.  
To see the specific WebTalk data collected for your design, open the  
usage_statistics_webtalk.html or usage_statistics_webtalk.xml file in the  
implementation directory.  
INFO: [Common 17-83] Releasing license: Implementation  
7 Infos, 0 Warnings, 0 Critical Warnings and 0 Errors encountered.  
write_bitstream completed successfully  
write_bitstream: Time (s): cpu = 00:00:04 ; elapsed = 00:00:14 . Memory (MB): peak =  
2729.707 ; gain = 206.934 ; free physical = 2837 ; free virtual = 8407
```

"Hmmm Looks good."



“Jeeze when I deploy this in a high-throughput system where I have a new pair of numbers to divide every 10ns, the division results are trash. Oh well it must be a failure of the FPGA or literally anything other than things that I have done in my life like playing some stick figure game and then posting it on Instagram under the assumption that a former TA for the class won’t see it and then send it to the instructor. Gonna ask for the checkoff without looking at the log.”

-Quote overheard in “verbatim” lab.

You look through the output from the build...

Starting at line 1322:

Time (s): cpu = 00:00:12 ; elapsed = 00:00:09 . Memory (MB): peak = 3031.699 ; gain = 0.000 ; free physical = 15405 ; free virtual = 36429

Phase 22 Post Process Routing
Phase 22 Post Process Routing | Checksum: 1e34d7eca

Time (s): cpu = 00:00:12 ; elapsed = 00:00:09 . Memory (MB): peak = 3031.699 ; gain = 0.000 ; free physical = 15405 ; free virtual = 36429

Phase 23 Post Router Timing
INFO: [Route 35-20] Post Routing Timing Summary | WNS=-21.127| TNS=-127.926| WHS=0.180 | THS=0.000 |

Phase 23 Post Router Timing | Checksum: 17b76201e

Time (s): cpu = 00:00:12 ; elapsed = 00:00:09 . Memory (MB): peak = 3031.699 ; gain = 0.000 ; free physical = 15405 ; free virtual = 36429

CRITICAL WARNING: [Route 35-39] The design did not meet timing requirements. Please run report_timing_summary for detailed reports.

Resolution: Verify that the timing was met or had small violations at all previous steps (synthesis, placement, power_opt, and phys_opt). Run report_timing_summary and analyze individual timing paths.

INFO: [Route 35-253] TNS is the sum of the worst slack violation on every endpoint in the design. Review the paths with the biggest WNS violations in the timing reports and modify your constraints or your design to improve both WNS and TNS.

Total Elapsed time in route_design: 9.5 secs

Look at

post_route_timing.rpt

Timing Report

Slack (VIOLATED) : -21.127ns (required time - arrival time)
Source: divisor_reg[0]/C
(rising edge-triggered cell FDRE clocked by gclk {rise@0.000ns fall@4.000ns period=10.000ns})
Destination: quotient_reg[0]/D
(rising edge-triggered cell FDRE clocked by gclk {rise@0.000ns fall@4.000ns period=10.000ns})
Path Group: gclk
Path Type: Setup (Max at Slow Process Corner)
Requirement: 10.000ns (gclk rise@10.000ns - gclk rise@0.000ns)
Data Path Delay: 31.190ns (logic 21.759ns (69.762%) route 9.431ns (30.238%))
Logic Levels: 89 (CARRY4=79 LUT2=1 LUT3=9)
Clock Path Skew: 0.053ns (DCD - SCD + CPR)
Destination Clock Delay (DCD): 4.934ns = (14.934 - 10.000)
Source Clock Delay (SCD): 5.067ns
Clock Pessimism Removal (CPR): 0.186ns
Clock Uncertainty: 0.035ns ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
Total System Jitter (TSJ): 0.071ns
Total Input Jitter (TIJ): 0.000ns
Discrete Jitter (DJ): 0.000ns
Phase Error (PE): 0.000ns

Location	Delay type	Incr(ns)	Path(ns)	Netlist Resource(s)
N15	(clock gclk rise edge)	0.000	0.000 r	clk_100mhz (IN)

Conclusion...

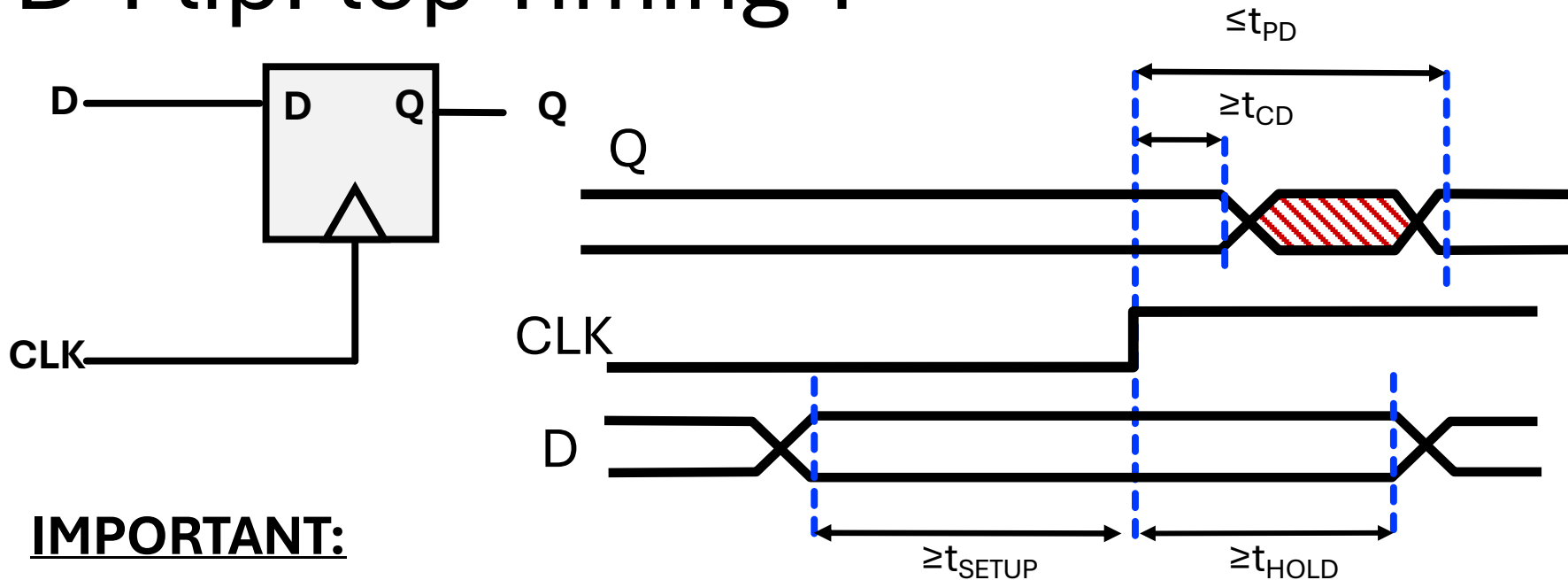
- Fix the design
- Future classes

Metastability and Dealing With It

...Or what are the problems with working with Sequential Logic?....

D-FlipFlop Timing 1

 =undetermined state



IMPORTANT:

t_{PDr} : maximum propagation delay, @posedge CLK $D \rightarrow Q$
Maximum time it takes for Q to change after rising edge of CLK

t_{CDr} : minimum contamination delay, @posedge CLK $D \rightarrow Q$
Minimum time it takes for Q to start to change after rising edge of CLK

t_{SETUP} : setup time
 How long D input must be stable **before** the rising edge of CLK

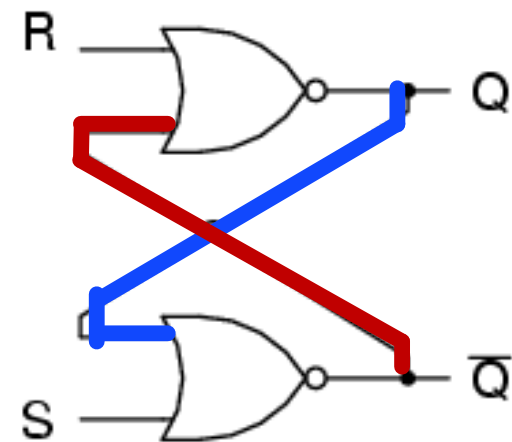
t_{HOLD} : hold time
 How long D input must be stable **after** the rising edge of CLK

*How it
behaves*

*What it
needs from
us/the rest
of the circuit*

Setup and Hold rules...

- Avoid there being a chance that the internal bistable memory storage elements (latches) could possibly move through a truth table transition that could result in undefined/unreliable behavior



S	R	Q	$\bar{Q}$
0	0	latch	latch
0	1	0	1
1	0	1	0
1	1	0	0

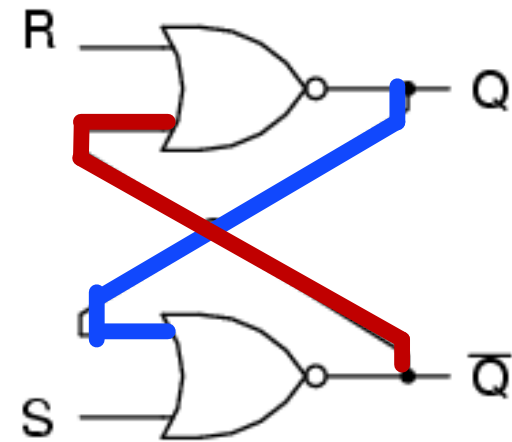
bistable

Not bistable

DO NOT DO

Setup and Hold rules...

- During that undefined behavior, the high gain and feedback could result in:
 - Oscillations
 - Non-digital outputs (analog...yuck!)
 - Loss of state



S	R	Q	$\bar{Q}$
0	0	latch	latch
0	1	0	1
1	0	1	0
1	1	0	0

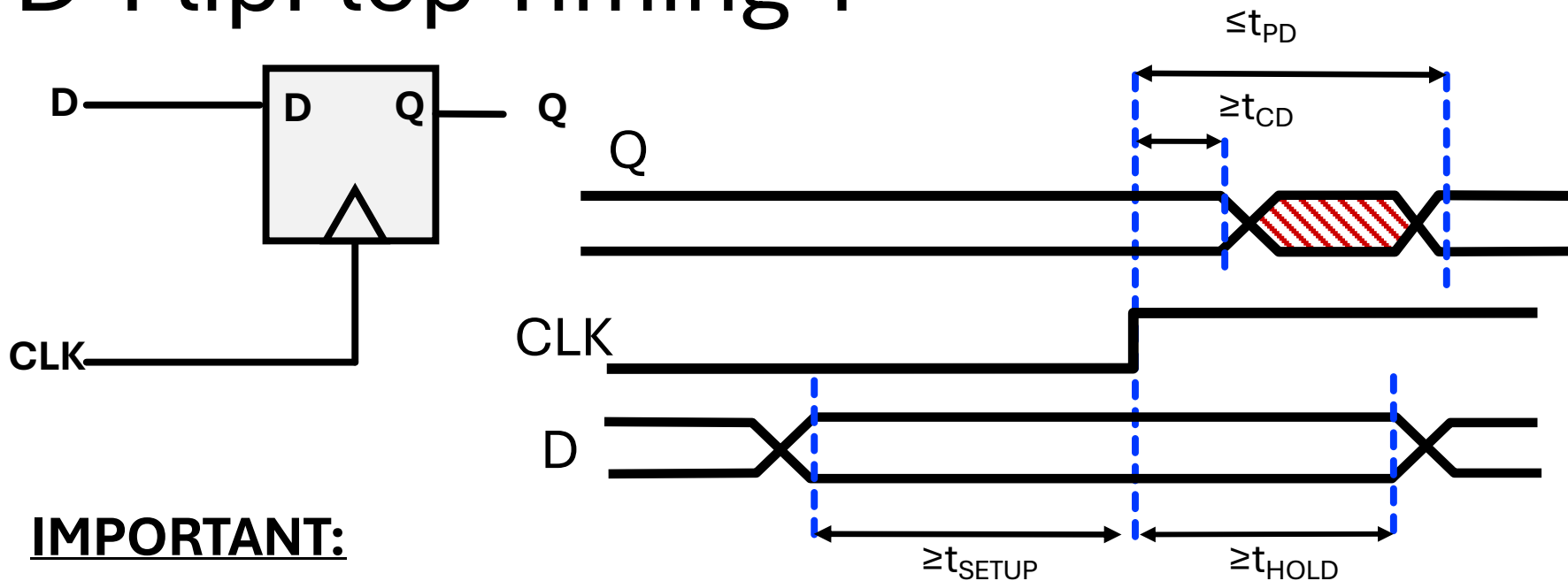
bistable

Not bistable

DO NOT DO

D-FlipFlop Timing 1

 =undetermined state



IMPORTANT:

t_{PDr} : maximum propagation delay, @posedge CLK $D \rightarrow Q$
Maximum time it takes for Q to change after rising edge of CLK

t_{CDr} : minimum contamination delay, @posedge CLK $D \rightarrow Q$
Minimum time it takes for Q to start to change after rising edge of CLK

t_{SETUP} : setup time
 How long D input must be stable **before** the rising edge of CLK

t_{HOLD} : hold time
 How long D input must be stable **after** the rising edge of CLK

**How it
behaves**

**What it
needs from
us/the rest
of the circuit**

But like...you know...what if ?...



- ...We did violate the rules? Rules are meant to be broken teehee.
- This is the big glowing cigarette lighter in digital design. The Man says “DO NOT VIOLATE TIMING”...I’m gonna violate timing.
- Some modern cars still have these weird looking powers sockets in them ->



But like...you know...what if ?...

- Years ago, a defining moment of many children's was sitting bored in a car unsupervised, pushing in the banal looking cigarette lighter, then removing it and seeing a bright curious looking glowing orange spiral thing, and then proceeding to touch it with your finger.



So like...what if you *did* violate setup and hold? What would happen?

- Change data within setup and hold boundary...
- The output could very well go metastable for a certain amount of time before resolving

Figure 1. Metastability Timing Parameters

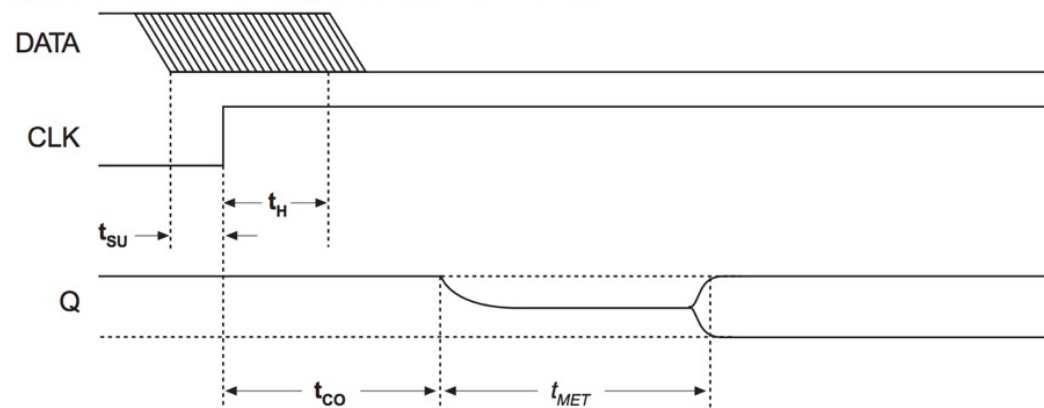
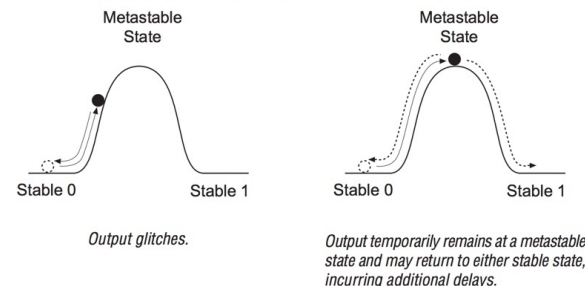
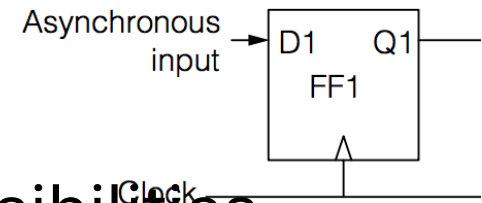


Figure 2. Effects of Violating t_{SU} & t_H Requirements

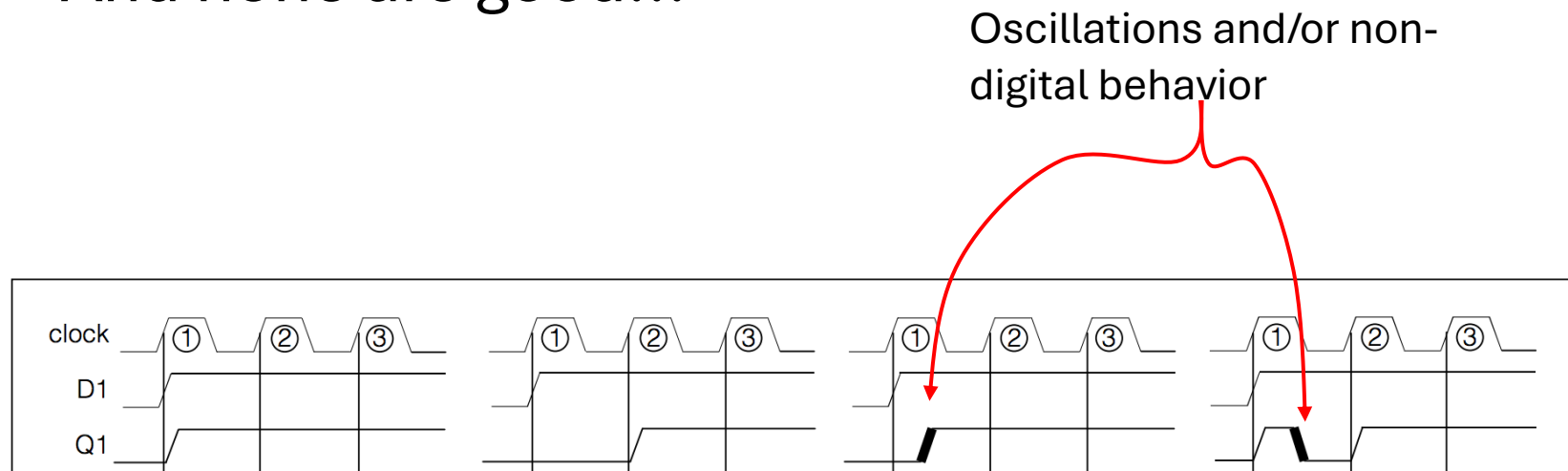


Metastability in Altera (☹) Devices Altera Application Note 42 (1999)

And what would that actually look like?

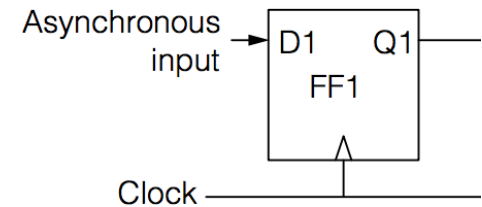


- There's a chance that any of the possibilities below would happen...
- And none are good...



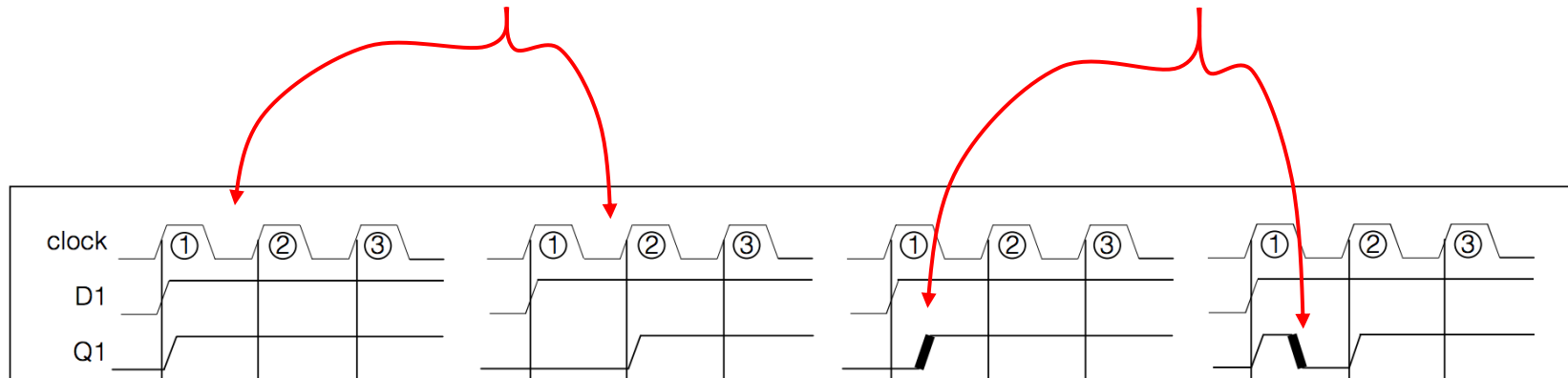
*“Metastability and Synchronizers: A Tutorial”
Ran Ginosar, Technion Israel Institute of Technology*

And what are the odds?



Much much more likely...
(still not good at all)

Much much less likely... Though
potentially catastrophic
depending on how long the
metastability persists...



*“Metastability and Synchronizers: A Tutorial”
Ran Ginosar, Technion Israel Institute of Technology*

So like...what if you *did* violate setup and hold? What would happen?

- Chances are pretty good that nothing catastrophic (metastability) would happen.
- But from a functional perspective, bits would go missing, be skipped, get unsynced, etc... This will cause massive errors in data.
- The odds that you would *actually* nail the flip flop just right that it could go metastable and matter will range from 1-in-a-million to well up to 1-in-the billions or even higher
- But we do need to worry about it.

Wait what...?

- “Joe, my brother in Christ, you’re telling me we’re worried about a problem that...
- Won’t happen if we follow a set of rules?...
- And even if we will violate said rules, will only happen 1 in 1 million chance
- And even then only 1 in a 1 billion chance that it will actually show up in a way that matters?..
- The chance that the earth will suffer a mass extinction event from asteroids in our life times is higher than that...
- Or...I mean....Kalshi has the current probability of the US defaulting on its debt before 2027 as 3%....what are we even worried about?”

But...it isn't just that simple

- We don't run flip flops once and then are done with them. We cycle them 100 million times per second...
- And then we'll have thousands or millions of flip flops in a design all getting exposed independently
- So in practice routinely violating setup and hold times could result in statistically probable situations of many actual metastable events per second

Mean Time Between Failure (MTBF)

- These probability numbers come not necessarily from a ton of theory, but often from in lab testing.
- They'll set up flip flops running and hit them asynchronously (without regard to setup and hold) and then track how often they see metastability showing up.
- Then generate statistics about how they fail including MTBF
- Hard to get exact numbers on modern devices, but a few FPGAs gens ago, you can find data reporting a MTBF of ~one minute on test situations of 50 million attempts per second. (1 in 3 billion?)

So conclusion:

- You do want to violate setup and hold.
- There will be a clear functional bug where bits get dropped/mispipelined (especially when working in systems where many bits need to get delivered at once)
 - Often times when timing *does fail* it'll be on paths in the circuit related to upper bits in math calculations, for example...
 - Situations which have the hardest to achieve portion of the math operation.
- But even “rare” events can show up pretty reliably

Single Clock Synchronous Discipline

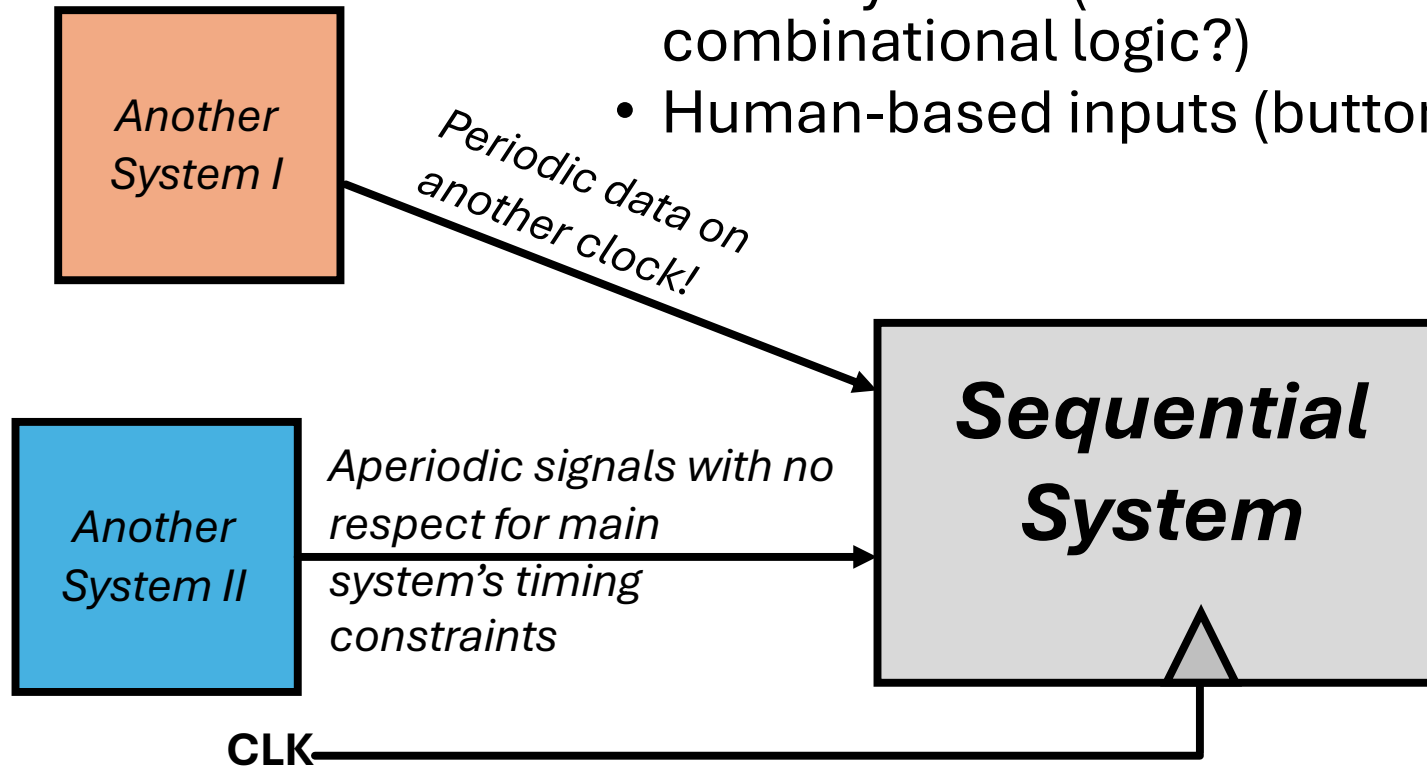
- The timing requirements are already complicated enough with one clock. Avoid multiple clocks where possible! DO NOT clock flip flops on non-clock lines.
- Single Clock signal shared among all clocked devices (one clock domain)
- Only care about the value of combinational circuits just before rising edge of clock
- Clock period greater than every propagation delay delay+setup time
- Change saved state after noise-inducing logic changes have stopped!

But Metastability can still be a huge threat

- Not so much when everything is clocked together
- But when it isn't.
- And you really can't avoid this problem.

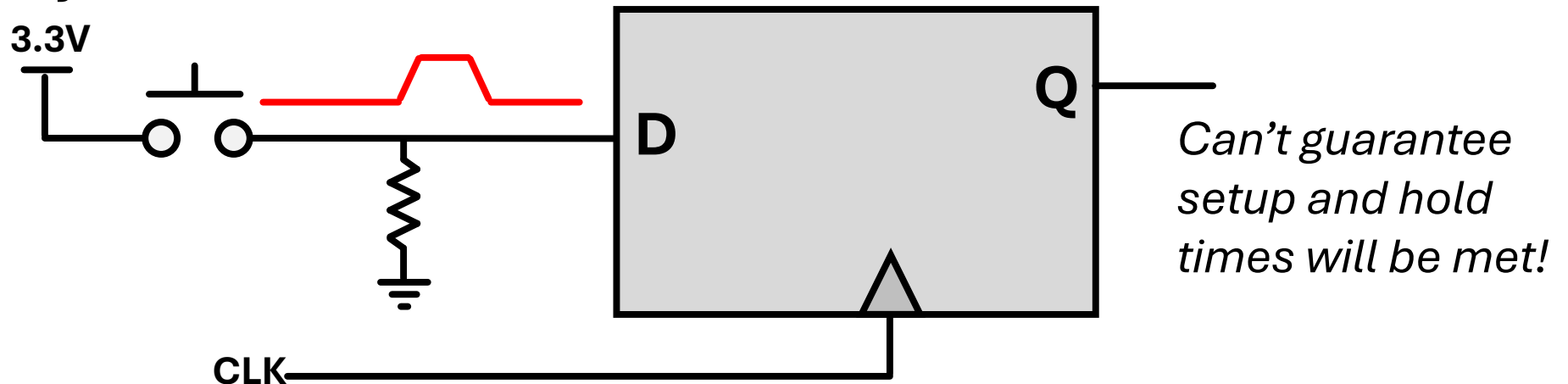
What if...?

- ...we need to interface with outside equipment:
 - Other systems (on different clocks or from combinational logic?)
 - Human-based inputs (buttons)

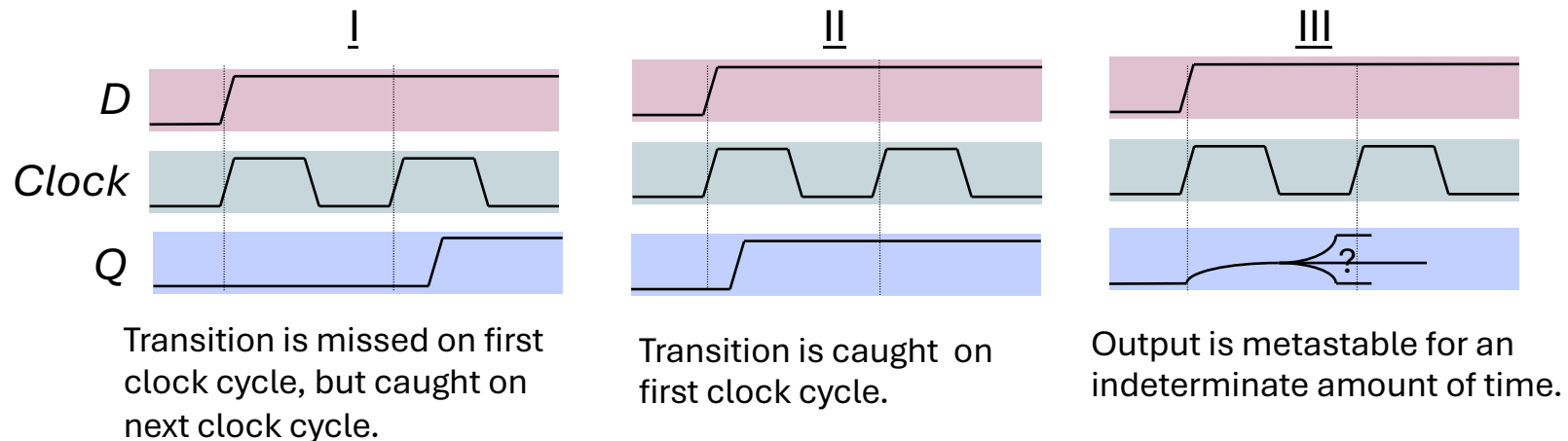


Can't guarantee setup and hold times will be met!

Example: Asynchronous Inputs in Sequential Systems



When an asynchronous signal causes a setup/hold violation...



Q: Which cases are problematic?

We can't actually protect against this.

- We could try to mandate that people and all other entities are phase locked to our 100 MHz clock, but we can't even agree on the best type of pizza, so that's not going to happen
- We need to just accept that asynchronous things will encounter our logic.
- We need to protect against that.

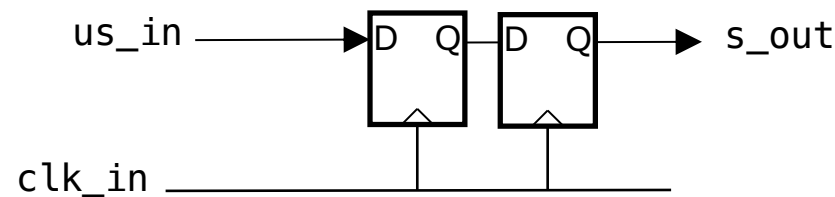
Huh?

- In Week 3:
- You'll be using these occasionally... or an equivalent...
- They build this
- Obviously that's a typo
- Right???????

```
module synchronizer #(parameter SYNC_DEPTH = 2)
    ( input wire clk,
      input wire rst,
      input wire us_in, //unsync_in
      output logic s_out); //sync_out

    logic (* ASYNC_REG = "TRUE" *) [SYNC_DEPTH-1:0] sync;

    always_ff @(posedge clk)begin
        if (rst)begin
            sync <= {(SYNC_DEPTH){us_in}};
        end else begin
            sync[SYNC_DEPTH-1] <= us_in;
            for (int i=1; i<SYNC_DEPTH; i= i+1)begin
                sync[i-1] <= sync[i];
            end
        end
    end
    assign s_out = sync[0];
endmodule
```



Handling Metastability

- Can't globally prevent metastability, but can isolate it!
- Stringing several registers together can isolate any freakouts!

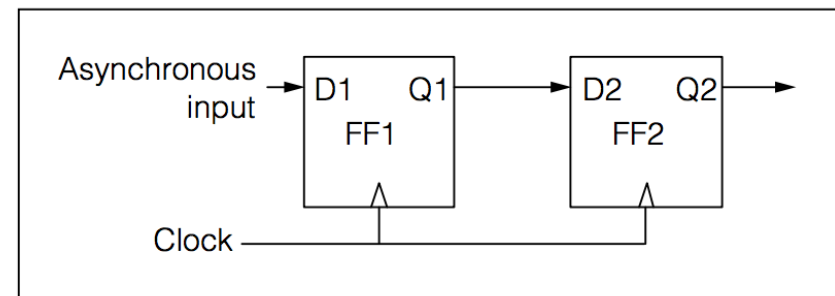
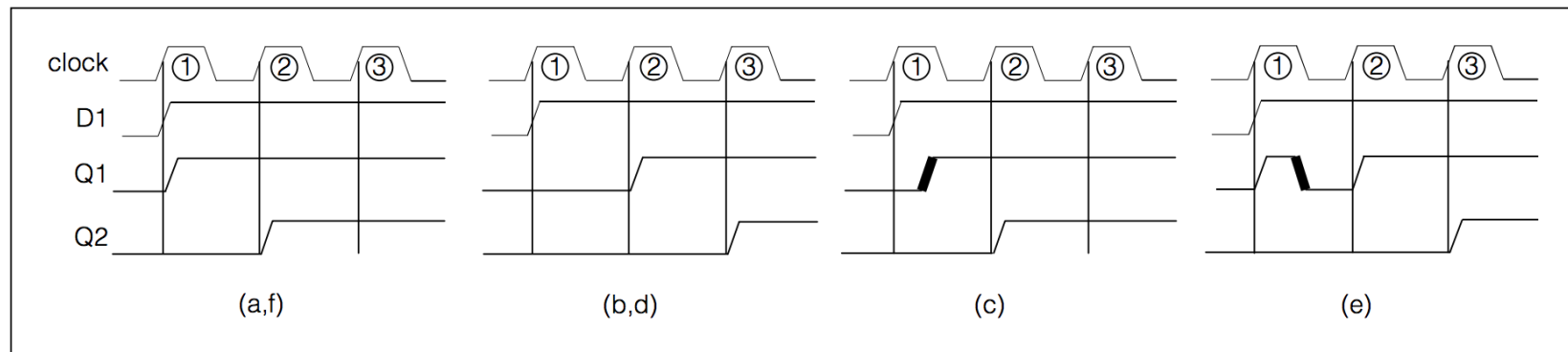


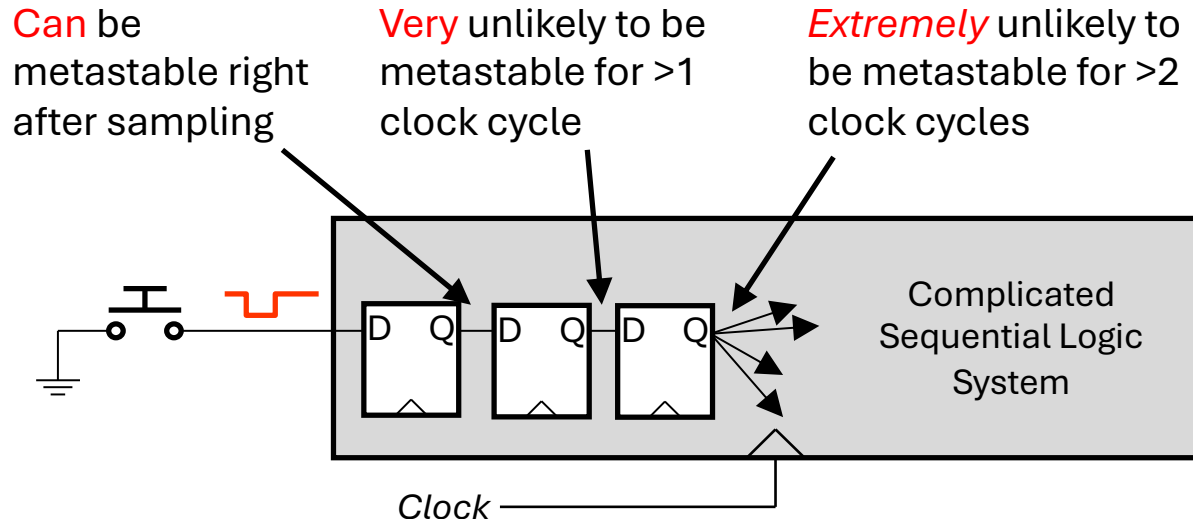
Figure 8. Two-flip-flop synchronization circuit.



“Metastability and Synchronizers: A Tutorial”
Ran Ginosar, Technion Israel Institute of Technology

Handling Metastability

- Completely preventing metastability turns out to be an impossible problem
- High gain of digital devices makes it likely that metastable conditions will resolve themselves quickly
- Solution to metastability: allow time for signals to stabilize

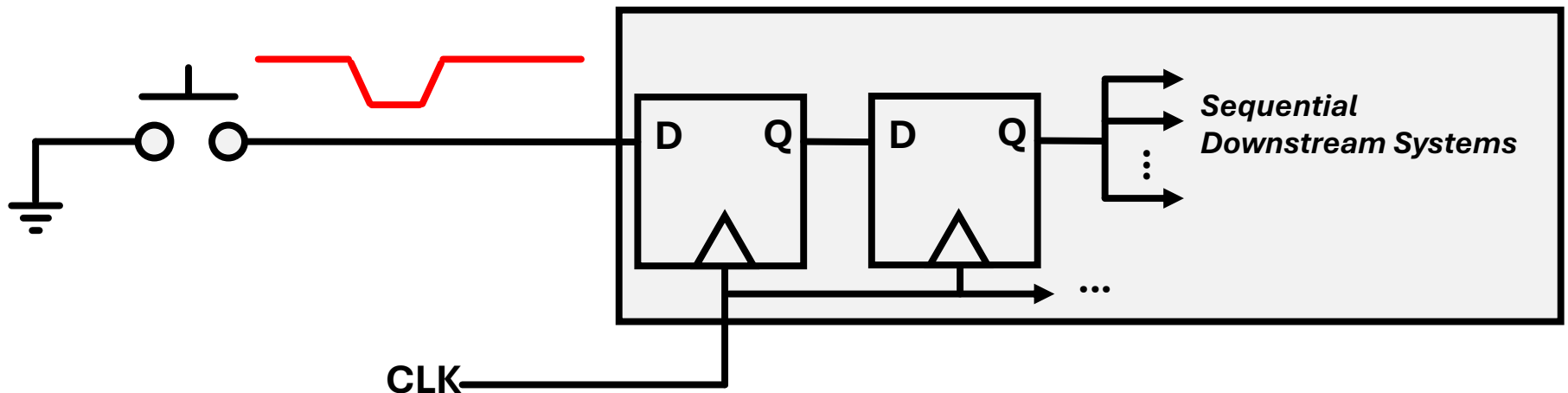
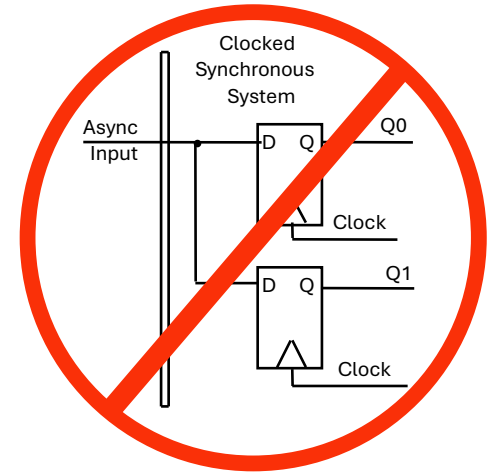


How many registers are necessary in 6.205?

- Depends on many design parameters (clock speed, device speeds, ...)
- In 6.205, a **pair of synchronization** registers is sufficient
- And for simple designs...with low t_{pd} you may not even need anything practically, but it is never a good idea to ignore it

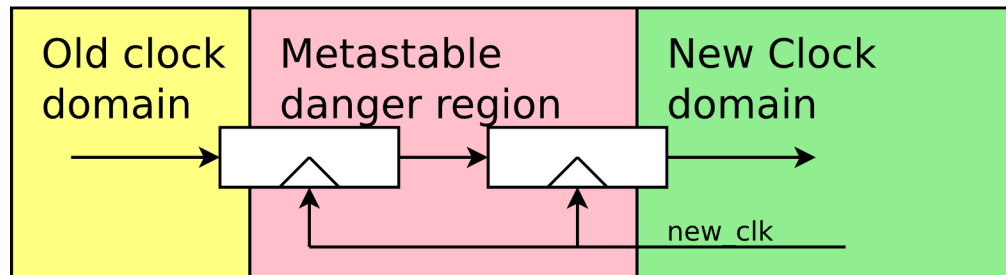
Handling Metastability

- Don't break off an asynchronous input until it has gone through some registers
- Basically: Ensure that external signals feed **exactly one** flip-flop chain before branching



Related: Clock Domain Crossing

- For example:
 - Data gets sent in at 25 MHz from one device (running on its own clock)
 - Your system runs at 50 MHz



```
1 //xfer_pipe can be >2 bits wide (2 is usually fine.)
2 always_ff @(posedge new_clock)
3   { new_val, xfer_pipe } <= { xfer_pipe, i_val };
```

- This only works when original clock domain frequency is less than or equal to new clock domain frequency

<https://zipcpu.com/blog/2017/10/20/cdc.html>

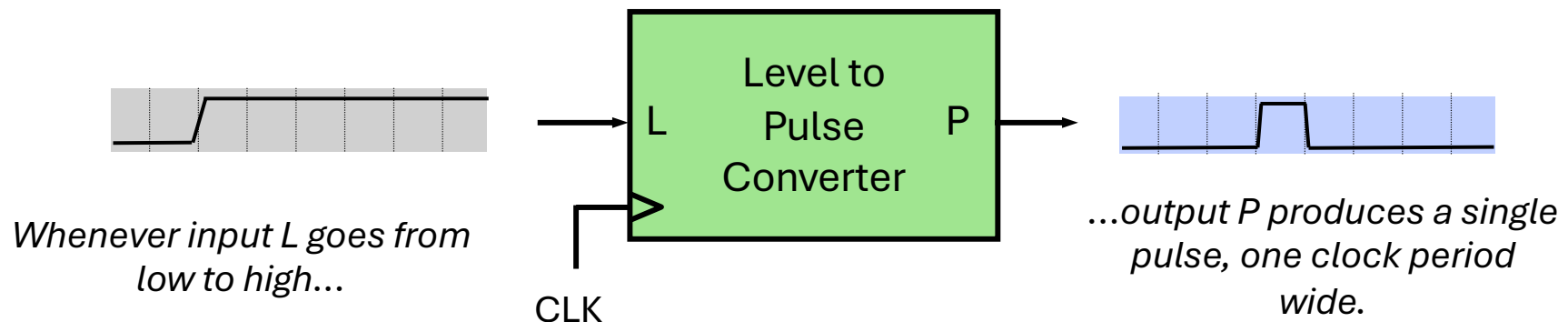
Finite State Machines

Finite State Machines

- You've gone over those in 6.191 so we're not going to spend too much time on these.
- Many of you have already been approaching designs like that, which is good.
- FSMs are an extremely productive way of structuring your digital designs.
- It allows you to break down your design into smaller sub-decisions.

Design Example: Level-to-Pulse

- A **level-to-pulse converter** produces a single-cycle pulse each time its input goes high.
- It's a **synchronous** rising-edge detector.
- Sample uses:
 - Buttons and switches pressed by humans for arbitrary periods of time
 - Single-cycle enable signals for modules



Level-to-Pulse

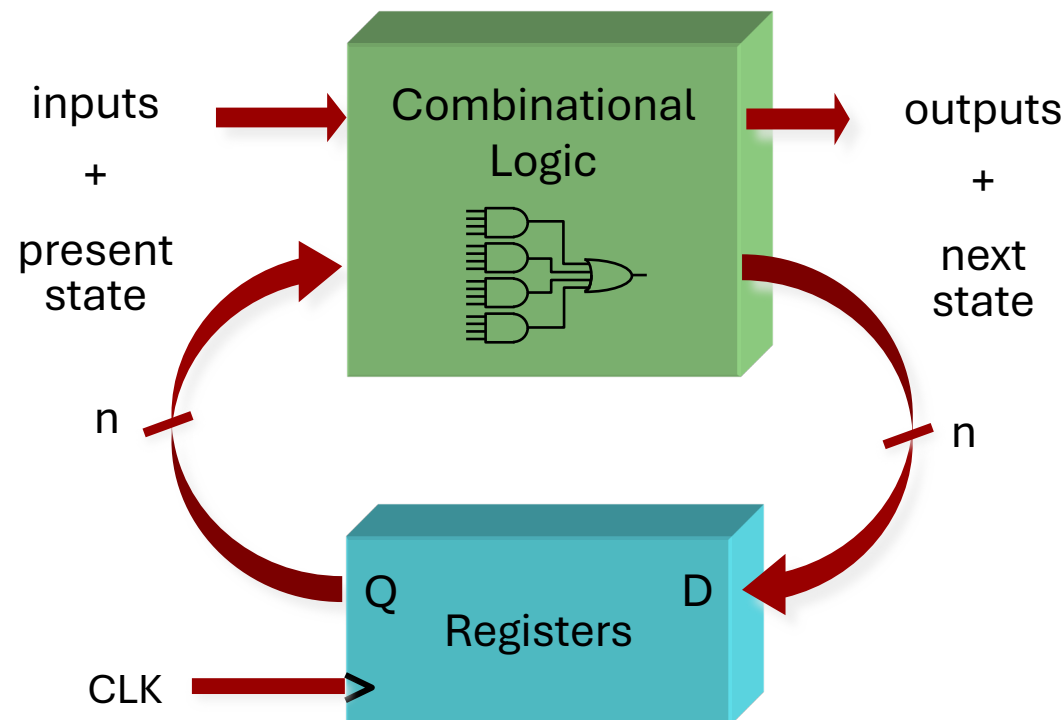
- One simple solution (~from Lab 02)
- One bit positive discrete time positive

```
module simple_soln( input wire clk,  
                   input wire l_in,  
                   output logic p_out);  
    logic old_l; //remember previous value!  
    always_ff @(posedge clk) begin  
        old_l <= l_in; //remember it!  
    end  
    assign p_out = l_in & ~ old_l; //high and prev low  
endmodule
```

- Let's try to formalize this a bit more

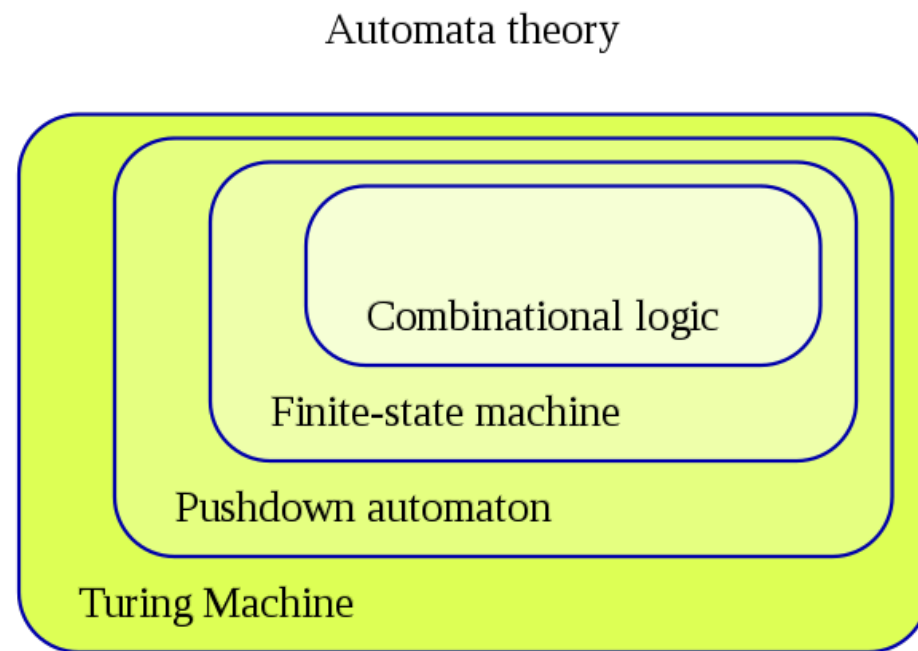
Finite State Machines

- Finite State Machines (FSMs) are a useful abstraction for **sequential circuits** with centralized “**states**” of operation
- At each clock edge, combinational logic computes **outputs** and **next state** as a function of **inputs** and **present state**



Even When you Don't Mean to...

- FSMs are a fundamental layer in the foundations of computation so you end up building them either way, so it is good to embrace it.



https://en.wikipedia.org/wiki/Automata_theory

Level-to-Pulse

- **State:** how/what stores past information?
- **Output Logic:** How does state and input influence output
- **State Transition Logic:** Logic dictating next state
- **State Transition:** Actual updating of state

State

```
module simple_soln( input wire clk,
                    input wire l_in,
                    output logic p_out);
    logic old_l; //remember previous value!
    always_ff @(posedge clk) begin
        old_l <= l_in; //remember it!
    end
    assign p_out = l_in & ~old_l; //high and prev low
endmodule
```

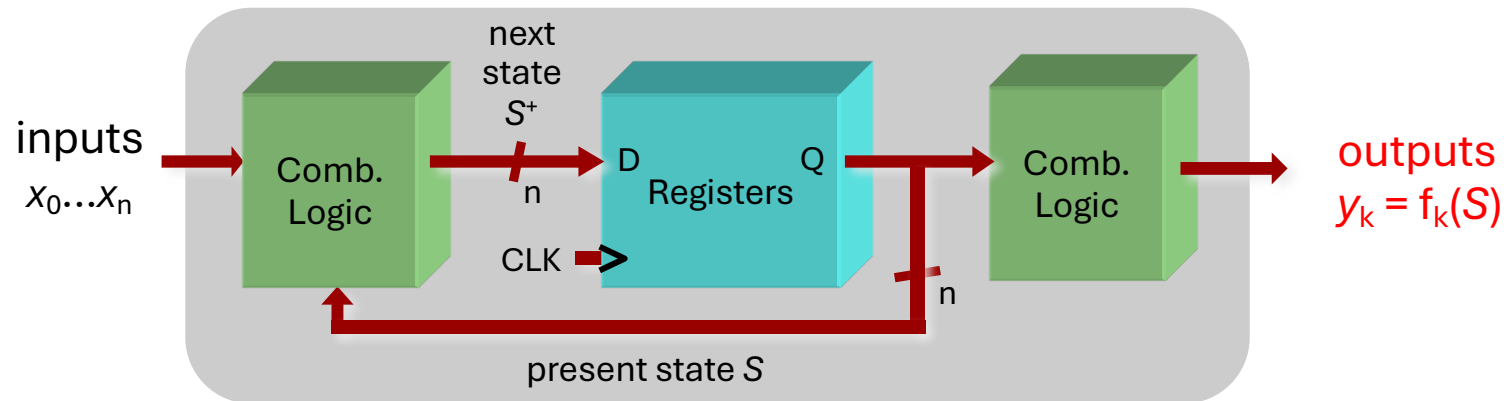
State Transition
and
State Transition Logic

Output Logic

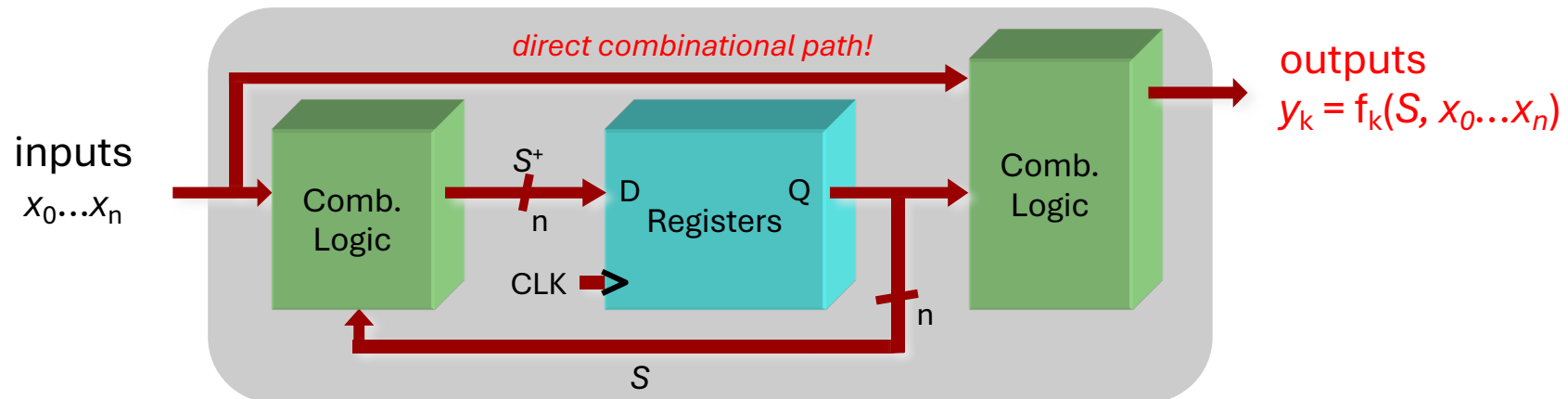
Let's Formalize it: Two Types of FSMs

Moore and **Mealy** FSMs : different output generation

- **Moore FSM:**



- **Mealy FSM:**



Moore



- Edward F. Moore
 - 1925-2003
 - Virginia Tech
 - Worked with Claude Shannon
-
- Not same Moore as Moore's Law...that was Gordon Moore from Intel

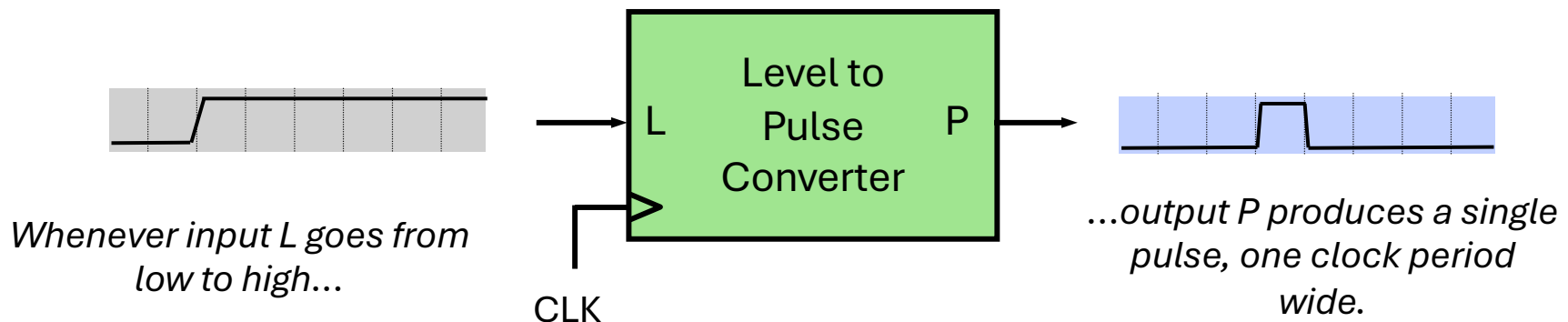
Mealy



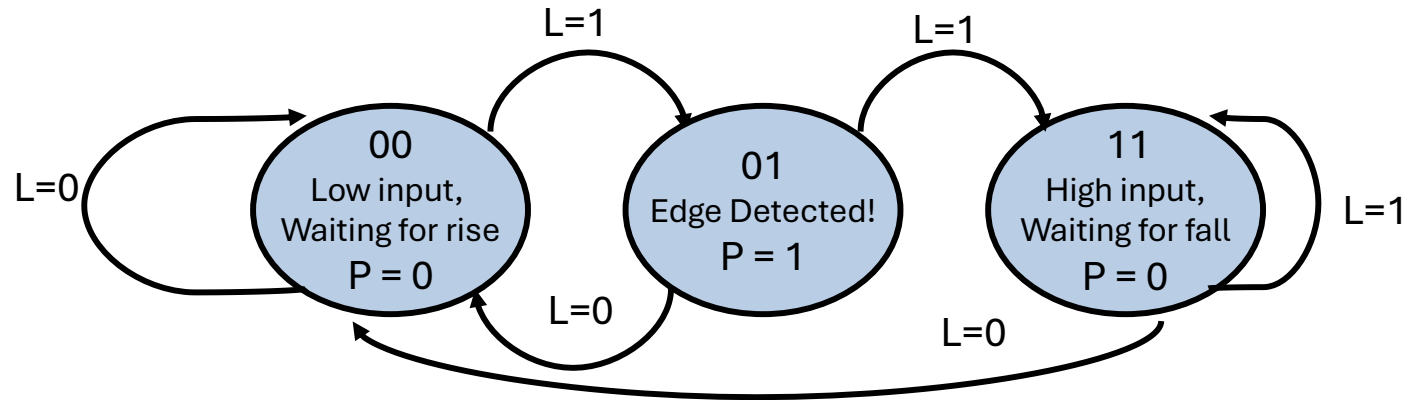
- George H. Mealy
- 1927-2010
- Harvard, Bell Labs

Design Example: Level-to-Pulse

- A **level-to-pulse converter** produces a single-cycle pulse each time its input goes high.
- It's a synchronous rising-edge detector.
- Sample uses:
 - Buttons and switches pressed by humans for arbitrary periods of time
 - Single-cycle enable signals for modules



Valid State Transition Diagrams



- Arcs leaving a state are **mutually exclusive**, i.e., for any combination of input values there's at most one applicable arc
- Arcs leaving a state are **collectively exhaustive**, i.e., for any combination of input values there's at least one applicable arc**
- So for each state: for any combination of input values there's exactly one applicable arc (**no ambiguity**)
- Often a starting state is specified
- Each state specifies values for all outputs (in the case of Moore)

Moore Level-to-Pulse Converter (SystemVerilog)

- An example of a **very explicit** Moore FSM implementation of the level-to-pulse converter:

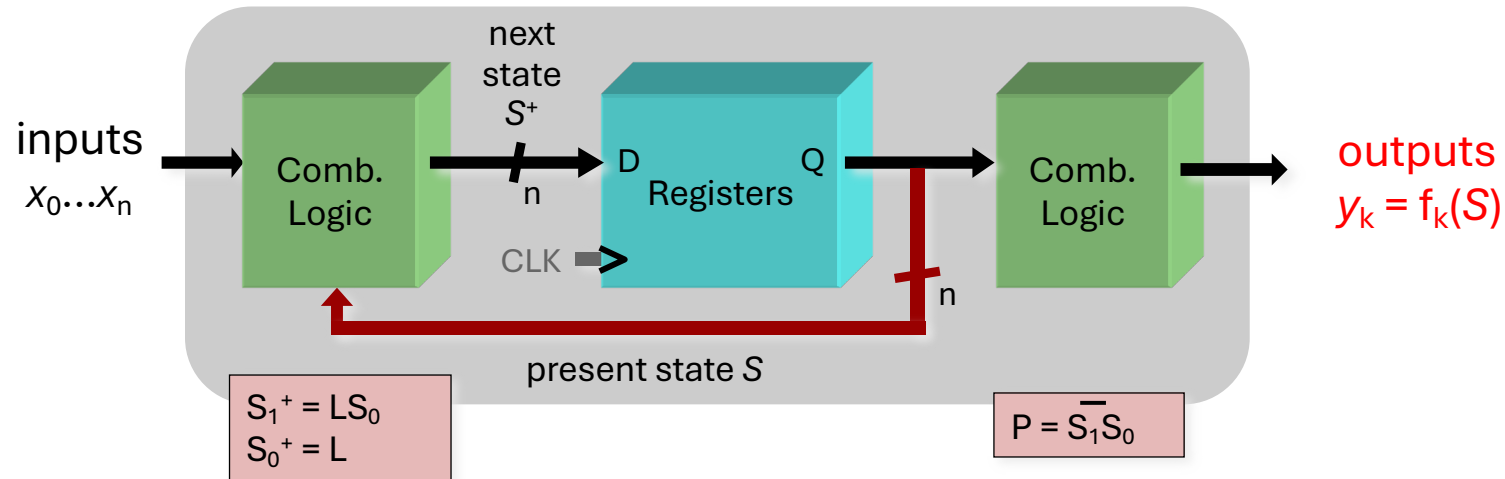
```
module moore_fsm( input wire clk,
                  input wire l_in,
                  output logic p_out);
    localparam LOW_WAITING = 2'b0; //define your states as...
    localparam EDGE_DETECTED = 2'b01; //parameters for easy...
    localparam HIGH_WAITING = 2'b10; //reading!
    logic [1:0] state; //contain state!
    logic [1:0] next_state; //hold next state!
    //Output Logic:
    always_comb begin
        case(state)
            LOW_WAITING: p_out = 1'b0; //output based only on...
            EDGE_DETECTED: p_out = 1'b1; //current state! This is...
            HIGH_WAITING: p_out = 1'b0; //a characteristic of Moore FSM
            default: p_out = 1'b0;
        endcase
    end
    //State Transition Logic (Combinational):
    always_comb begin
        case(state) //Also consider explicit if/elses
            LOW_WAITING: next_state = l_in?EDGE_DETECTED:LOW_WAITING;
            EDGE_DETECTED: next_state = l_in?HIGH_WAITING:LOW_WAITING;
            HIGH_WAITING: next_state = l_in?HIGH_WAITING:LOW_WAITING;
            default: next_state = LOW_WAITING;
        endcase
    end
    //State Transition
    always_ff @(posedge clk) begin
        //consider adding a reset here as well!
        state <= next_state; //state becomes calculated next_state
    end
endmodule
```

Moore Level-to-Pulse Converter (SystemVerilog)

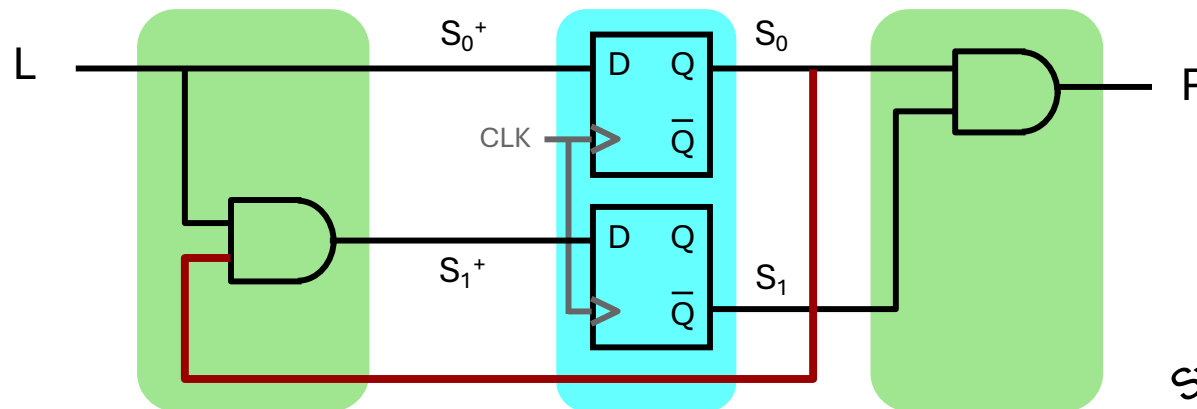
- Merging **State Transition Logic** and **State Transition** into one block
- Some people like this more

```
module moore_fsm( input wire clk,
                  input wire l_in,
                  output logic p_out);
    localparam LOW_WAITING = 2'b0;
    localparam EDGE_DETECTED = 2'b01;
    localparam HIGH_WAITING = 2'b10;
    logic [1:0] state;
    //Output Logic:
    always_comb begin
        case(state)
            LOW_WAITING: p_out = 1'b0;
            EDGE_DETECTED: p_out = 1'b1;
            HIGH_WAITING: p_out = 1'b0;
            default: p_out = 1'b0; //default
        endcase
    end
    //State Transition and Logic:
    always_ff @(posedge clk) begin
        //consider adding a reset here as well!
        case(state)
            LOW_WAITING: state <= l_in?EDGE_DETECTED:LOW_WAITING;
            EDGE_DETECTED: state <= l_in?HIGH_WAITING:LOW_WAITING;
            HIGH_WAITING: state <= l_in?HIGH_WAITING:LOW_WAITING;
            default: state <= LOW_WAITING;
        endcase
    end
endmodule
```

Moore Level-to-Pulse Converter



Moore FSM circuit implementation of level-to-pulse converter:

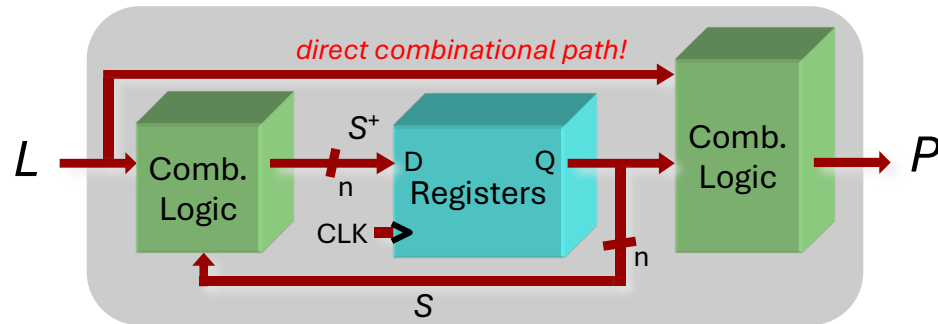


Synthesizable with
two LUT2s and two
flip flops

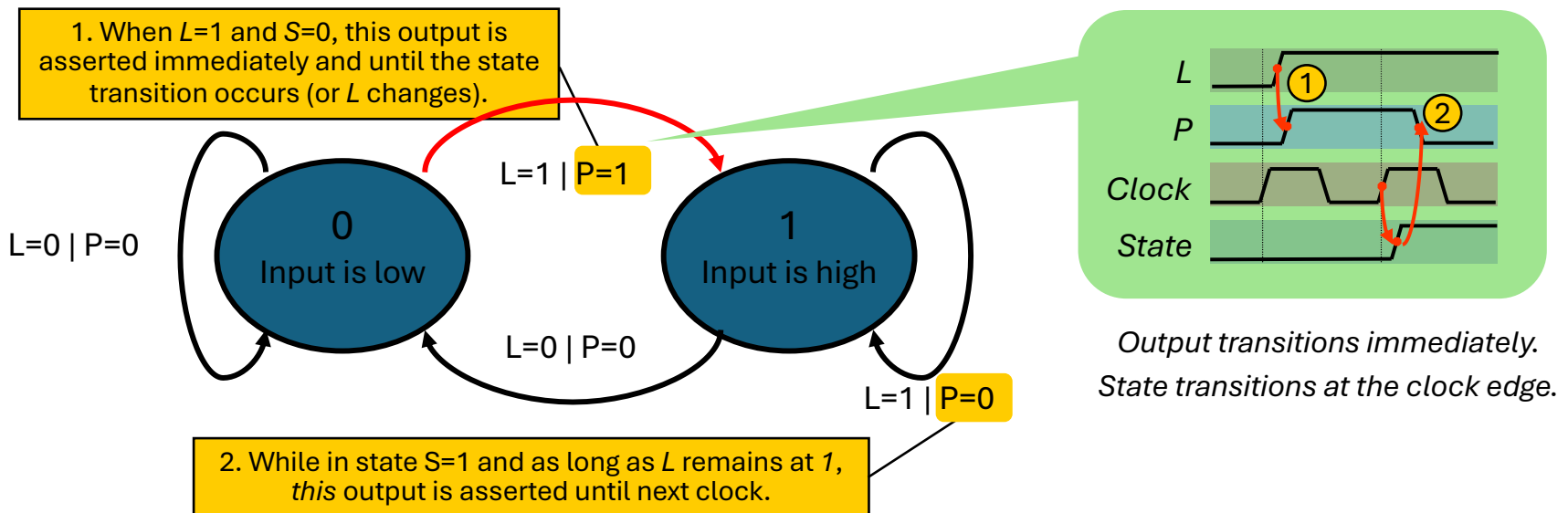
Moore is One Way

- There is another...

Design of a Mealy Level-to-Pulse



- Since outputs are determined by state *and* inputs, Mealy FSMs may need fewer states than Moore FSM implementations



Mealy Level-to-Pulse Converter (SystemVerilog)

- An example of a **very explicit** Mealy FSM implementation of the level-to-pulse converter:

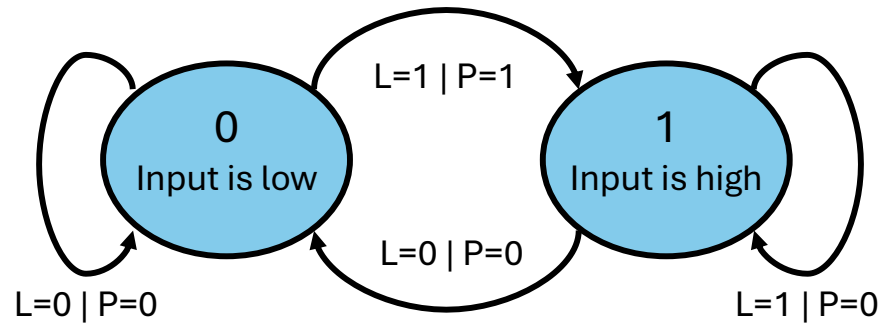
```
module mealy_fsm( input wire clk,
                  input wire l_in,
                  output logic p_out);
    localparam LOW_WAITING = 1'b0; //define states but notice...
    localparam HIGH_WAITING = 1'b1; //fewer needed...Mealy usually...
    //though not always, is like that
    logic state; //state (smaller than before...only two states to rep)
    logic next_state;
    //Output Logic:
    always_comb begin
        case(state) //outputs are based on state AND inputs!
            LOW_WAITING: p_out = l_in?1'b1:1'b0;
            HIGH_WAITING: p_out = 1'b0;
            default: p_out = 1'b0; //default
        endcase
    end
    //State Transition Logic:
    always_comb begin
        case(state)
            LOW_WAITING: next_state = l_in?HIGH_WAITING:LOW_WAITING;
            HIGH_WAITING: next_state = l_in?HIGH_WAITING:LOW_WAITING;
            default: next_state = LOW_WAITING;
        endcase
    end
    //State Transition
    always_ff @(posedge clk) begin
        //consider adding a reset here as well (same goes for any...
        //clocked logic block)
        state <= next_state;
    end
endmodule
```

Mealy Level-to-Pulse Converter (SystemVerilog)

- Merging **State Transition Logic** and **State Transition** into one block

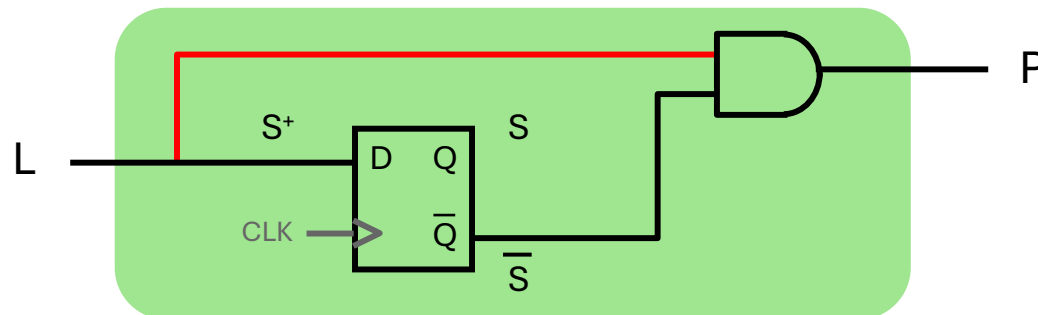
```
module mealy_fsm( input wire clk,
                  input wire l_in,
                  output logic p_out);
    localparam LOW_WAITING = 1'b0;
    localparam HIGH_WAITING = 1'b1;
    logic state;
    //Output Logic:
    always_comb begin
        case(state)
            LOW_WAITING: p_out = l_in?1'b1:1'b0;
            HIGH_WAITING: p_out = 1'b0;
            default: p_out = 1'b0; //default
        endcase
    end
    //State Transition and Transition Logic!
    always_ff @(posedge clk) begin
        //consider adding a reset here as well!
        case(state)
            LOW_WAITING: state <= l_in?HIGH_WAITING:LOW_WAITING;
            HIGH_WAITING: state <= l_in?HIGH_WAITING:LOW_WAITING;
            default: state <= LOW_WAITING;
        endcase
    end
endmodule
```

Mealy Level-to-Pulse Converter



*Synthesizable with
one LUT2 and one
flip flops*

Mealy FSM circuit implementation of level-to-pulse converter:

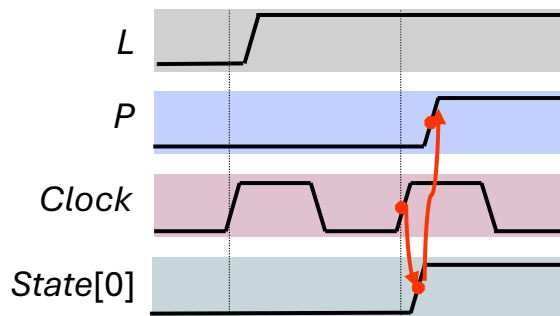


- FSM's state simply remembers the previous value of L
- Circuit benefits from the Mealy FSM's implicit single-cycle assertion of outputs during state transitions

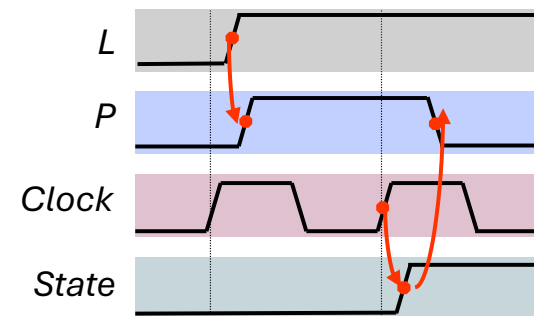
Moore/Mealy Trade-Offs

- How are they different?
 - Moore: $\text{outputs} = f(\text{state})$ only
 - Mealy $\text{outputs} = f(\text{state and input})$
 - Mealy outputs generally occur one cycle earlier than a Moore:

Moore: delayed assertion of P



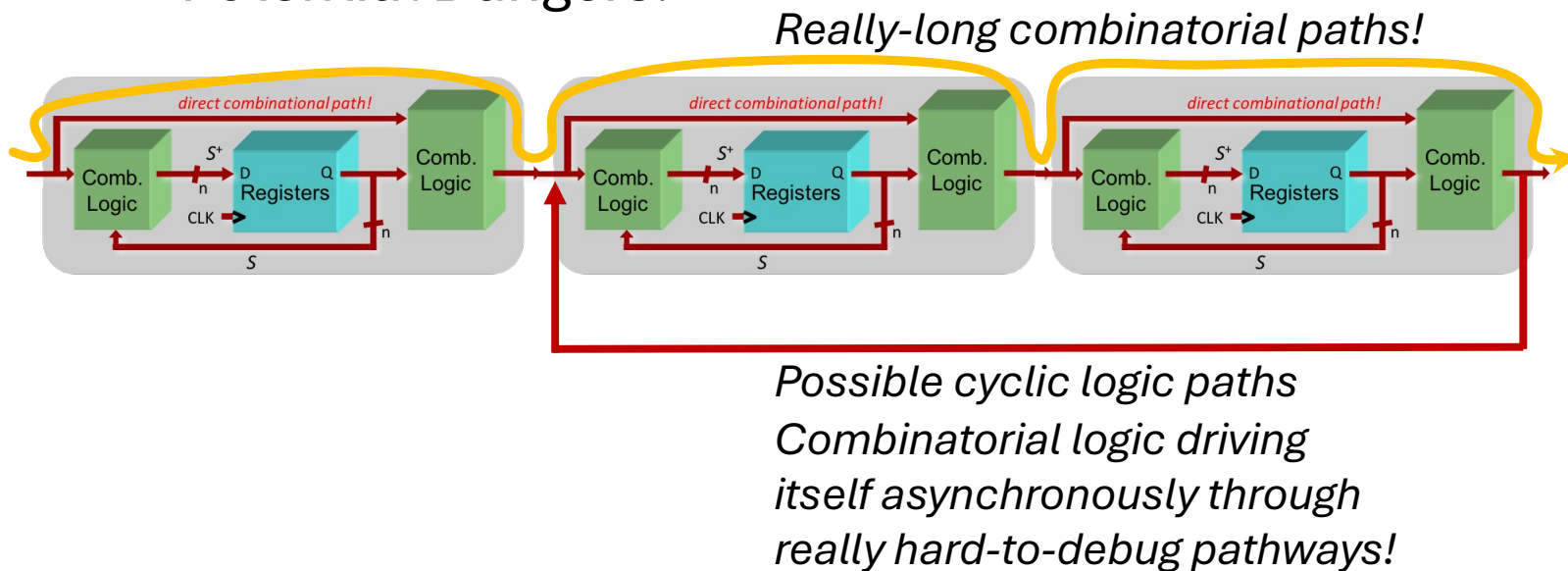
Mealy: immediate assertion of P



- Compared to a Moore FSM, a Mealy FSM **might...**
 - Be more difficult to conceptualize and design (both at circuit level and in HDL)
 - Have fewer states
 - Be expressed using fewer lines of Verilog

Moore/Mealy Trade-Offs

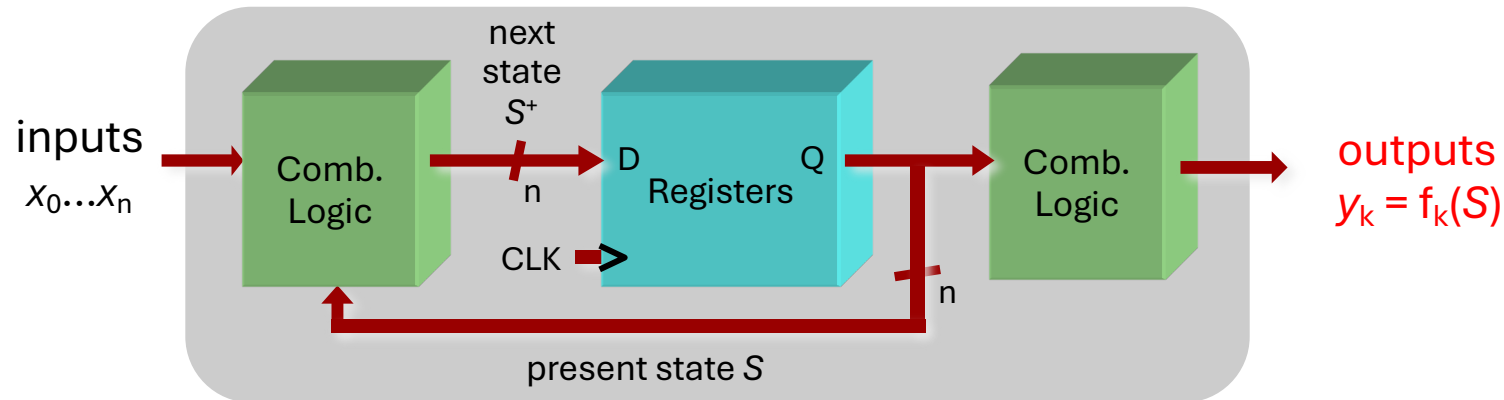
- Moore:
 - Usually more states
 - Each state has a particular output
- Mealy:
 - Fewer states, outputs are specified on edges of diagram
 - Potential Dangers:



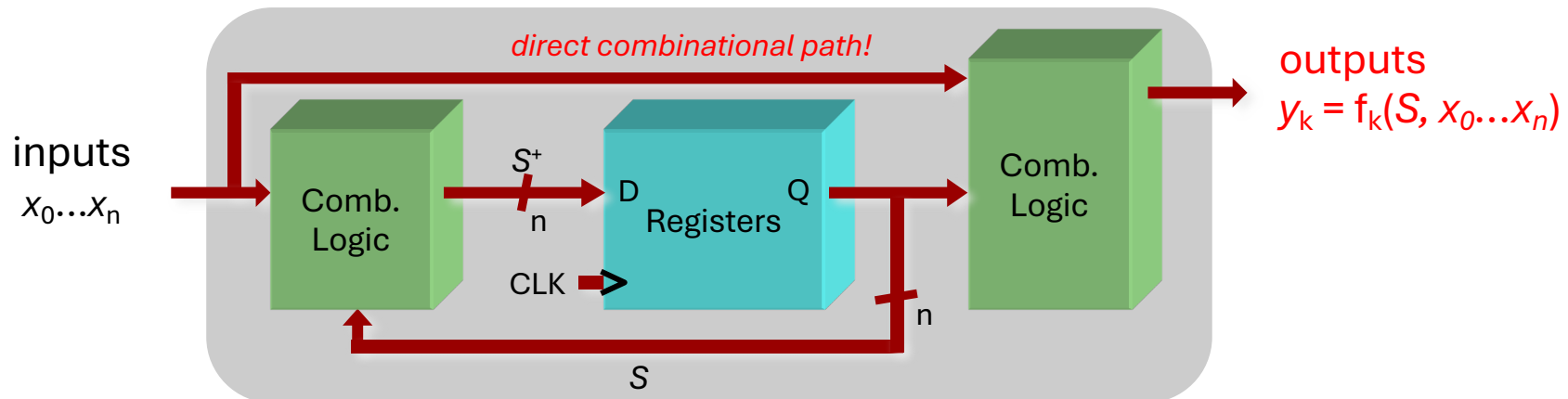
So Review: Two Types of FSMs

Moore and **Mealy** FSMs : different output generation

- **Moore FSM:**



- **Mealy FSM:**



Enums

- Another good practice to prevent against accidental doubling of state variables

```
module mealy_fsm( input wire clk,
                  input wire l_in,
                  output logic p_out);
    localparam LOW_WAITING = 1'b1; //define states but notice...
    localparam HIGH_WAITING = 1'b1; //fewer needed...Mealy usually...
    //though not always, is like that
    logic state; //state (smaller than before...only two states to rep)
    logic next_state;
    //Output Logic:
    always_comb begin
        case(state) //outputs are based on state AND inputs!
            LOW_WAITING: p_out = l_in?1'b1:1'b0;
            HIGH_WAITING: p_out = 1'b0;
            default: p_out = 1'b0; //default
        endcase
    end
end
```

shooooooooottt

Enums

- Enums allow you to define variables that only take on specific values *and* which will guarantee those values will be unique

```
module mealy_fsm( input wire clk,
                  input wire l_in,
                  output logic p_out);
    typedef enum logic {LOW_WAITING, HIGH_WAITING} state_t; //define states but notice...
    //fewer needed...Mealy usually...though not always, is like that
    state_t state; //state (smaller than before...only two states to rep)
    state_t next_state;
    //Output Logic:
    always_comb begin
        case(state) //outputs are based on state AND inputs!
            LOW_WAITING: p_out = l_in?1'b1:1'b0;
            HIGH_WAITING: p_out = 1'b0;
            default: p_out = 1'b0; //default
        endcase
    end
end
```

Encoding with Enums?

- Generally in SystemVerilog, an enum, unless specified will be 0, 1, 2, 3, etc...

```
enum {IDLE,GOT_5c,GOT_10c,GOT_15c,GOT_20c,  
      GOT_25c,GOT_35c,GOT_40c,GOT_45c,  
      GOT_50c,RETURN_20c,RETURN_15c,  
      RETURN_10c,RETURN_5c } state, next;
```

- When synthesizing, Vivado may decide on a different encoding, however if it is advantageous to the design and/or allows logical optimization

```
typedef enum {IDLE,GOT_5c,GOT_10c,GOT_15c,GOT_20c,  
             GOT_25c,GOT_35c,GOT_40c,GOT_45c,  
             GOT_50c,RETURN_20c,RETURN_15c,  
             RETURN_10c,RETURN_5c } state_enc_T;  
  
state_enc_T state, next;
```

Division (an example of an algorithm that takes an unknown amount of time)

```
def divider (dividend, divisor):  
    count = 0  
    if divisor==0:  
        return -1  
    while dividend>=divisor:  
        dividend -= divisor  
        count += 1  
    return (count, dividend)
```

Super efficient divider \s

A Divider

- This is a Verilog FSM example of the algorithm on the previous page which will run an unknown number of times given a set of inputs
- This is how the functionality of a while loop could be developed in your modules
- Will not handle negative, or 0 or other things...

```
3
4 module divider #(parameter WIDTH = 32)
5 (
6     input wire clk,
7     input wire rst,
8     input wire[WIDTH-1:0] dividend,
9     input wire[WIDTH-1:0] divisor,
10    input wire data_in_valid,
11    output logic[WIDTH-1:0] quotient,
12    output logic[WIDTH-1:0] remainder,
13    output logic data_out_valid,
14    output logic error,
15    output logic busy
16);
17 logic [WIDTH-1:0] quotient_g;
18 logic [WIDTH-1:0] dividend_h;
19 logic [WIDTH-1:0] divisor_h;
20 enum (RESTING, DIVIDING) state;
21 always_ff @(posedge clk) begin
22     if (rst) begin
23         quotient_g <= 0;
24         dividend_h <= 0;
25         divisor_h <= 0;
26         remainder <= 0;
27         quotient <= 0;
28         busy <= 1'b0;
29         error <= 1'b0;
30         state <= RESTING;
31         data_out_valid <= 1'b0;
32     end else begin
33         case (state)
34             RESTING: begin
35                 if (data_in_valid) begin
36                     state <= DIVIDING;
37                     quotient_g <= 0;
38                     dividend_h <= dividend;
39                     divisor_h <= divisor;
40                     busy <= 1'b1;
41                     error <= 1'b0;
42                 end
43                 data_out_valid <= 1'b0;
44             end
45             DIVIDING: begin
46                 if (dividend_h <= 0) begin
47                     state <= RESTING; //similar to return statement
48                     remainder <= dividend_h;
49                     quotient <= quotient_g;
50                     busy <= 1'b0; //tell outside world i'm done
51                     error <= 1'b0;
52                     data_out_valid <= 1'b1; //good stuff!
53                 end else if (divisor_h == 0) begin
54                     state <= RESTING;
55                     remainder <= 0;
56                     quotient <= 0;
57                     busy <= 1'b0; //tell outside world i'm done
58                     error <= 1'b1; //ERROR
59                     data_out_valid <= 1'b1; //valid ERROR
60                 end else if (dividend_h < divisor_h) begin
61                     state <= RESTING;
62                     remainder <= dividend_h;
63                     quotient <= quotient_g;
64                     busy <= 1'b0;
65                     error <= 1'b0;
66                     data_out_valid <= 1'b1; //good stuff!
67                 end else begin
68                     //state staying in.
69                     state <= DIVIDING;
70                     quotient_g <= quotient_g + 1'b1;
71                     dividend_h <= dividend_h - divisor_h;
72                 end
73             end
74         endcase
75     end
76 end
77 endmodule
78 `default_nettype wire
79
```