6.205

Sequential Logic II:

Sequential Logic Timing

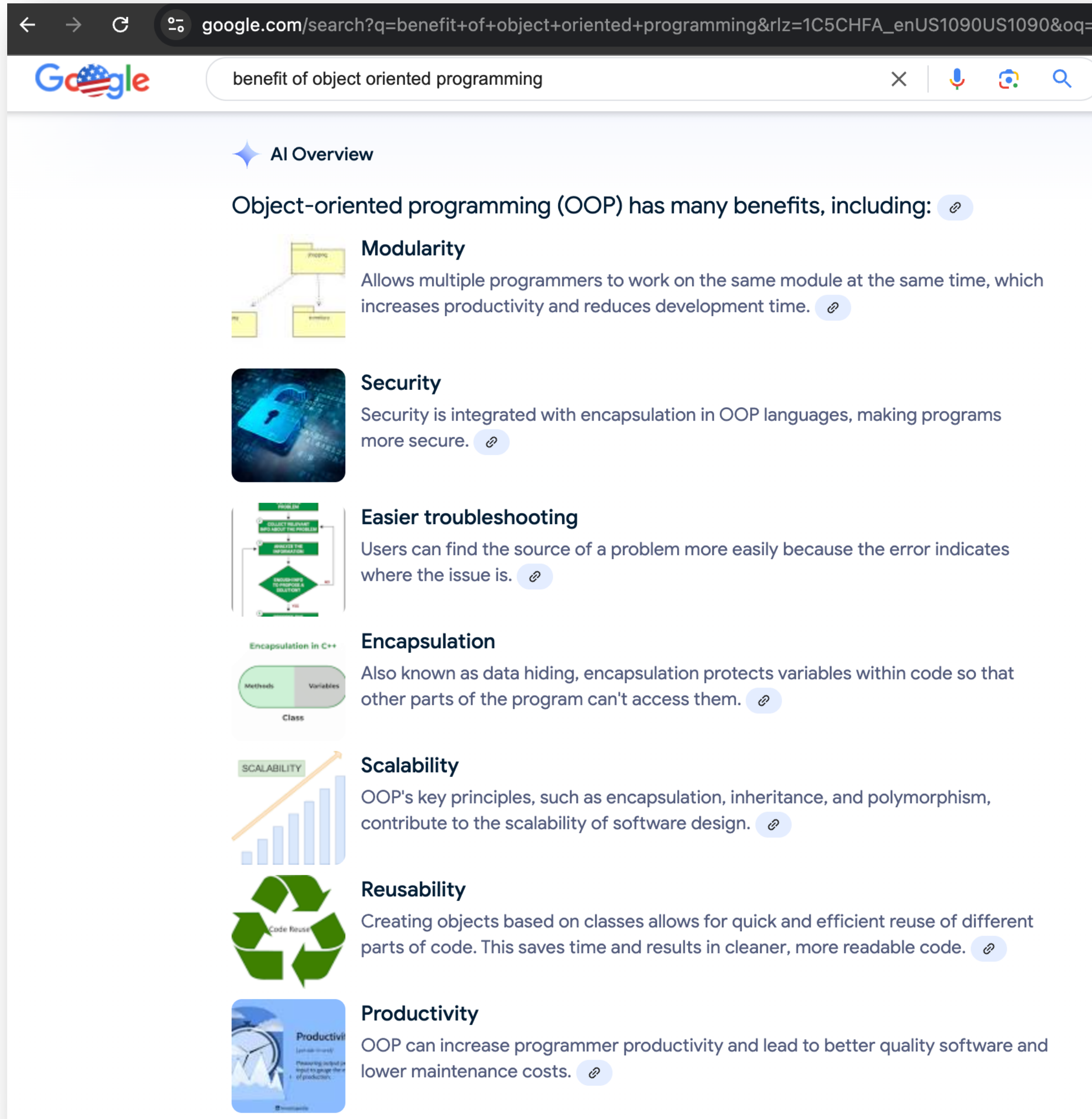
Intro to Finite State Machines

Administrative

- Week 02’s content is due tomorrow
- Week 03’s comes out on Thursday

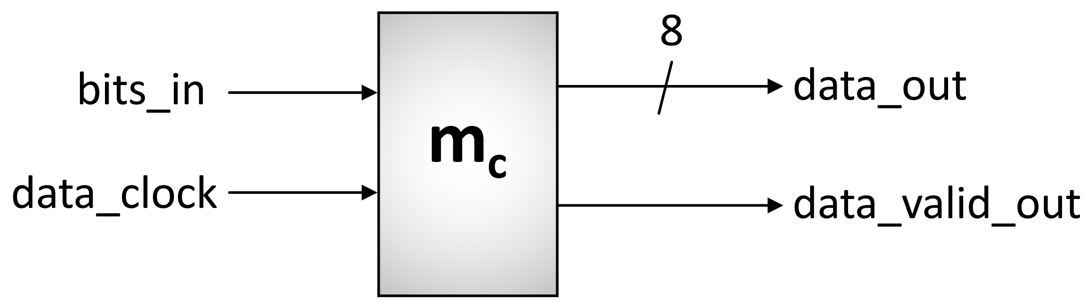
Think Like a Hardware Engineer

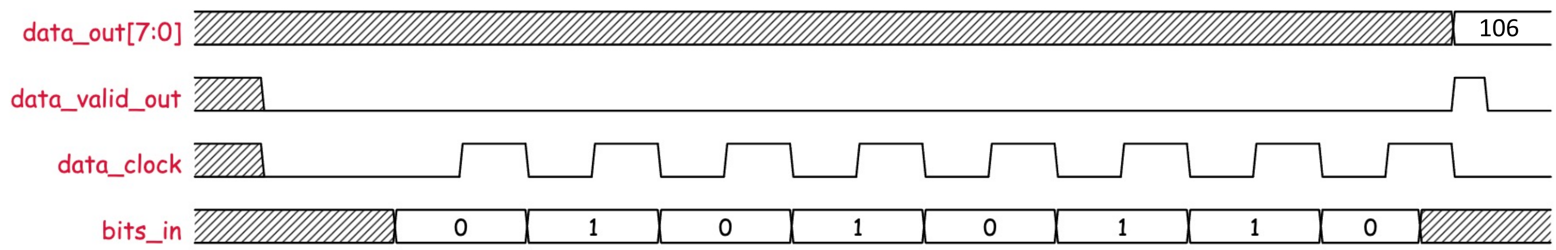
- Many programming constructs/patterns are done to help you, the person, rather than reflect the underlying hardware design
- Part of becoming a good hardware-focused engineer is learning how to give the machine what it wants.
- For example…object oriented programming. Who is that really for?


google.com/search?q=benefit+of+object+oriented+programming&rlz=1C5CHFA_enUS1090US1090&oq=b
benefit of object oriented programming
AI Overview
Object-oriented programming (OOP) has many benefits, including:
Modularity
Allows multiple programmers to work on the same module at the same time, which increases productivity and reduces development time.
Security
Security is integrated with encapsulation in OOP languages, making programs more secure.
Easier troubleshooting
Users can find the source of a problem more easily because the error indicates where the issue is.
Encapsulation
Also known as data hiding, encapsulation protects variables within code so that other parts of the program can't access them.
Encapsulation in C++
Methods
Variables
Class
Scalability
OOP's key principles, such as encapsulation, inheritance, and polymorphism, contribute to the scalability of software design.
SCALABILITY
Reusability
Creating objects based on classes allows for quick and efficient reuse of different parts of code. This saves time and results in cleaner, more readable code.
Code Reuse
Productivity
OOP can increase programmer productivity and lead to better quality software and lower maintenance costs.

Consider this task: Data Deserializer

- Single bits come in **lsb-first*** one after the other on a clock.
- Module assembles them into 8-bit bytes and sends them out





***in SPI in lab, data is transferred msb first**

The solution must obviously involve an Indexable array

- What are you assuming?
- What functionality/capability can be stripped away?
- Are you doing what you're doing…
  - for you? (*not right answer...you don't matter*)
  - or the FPGA? (*right answer...it is what matters*)

So ugh…

```
logic [7:0] buffer;
logic [2:0] buffer_ind;
always_ff @(posedge clk)begin
  buffer[buffer_ind] <= bit_in;
  buffer_ind <= buffer_in +1;
end
```

- Oh but I gotta take care of wrap-around

```
logic [7:0] buffer;
logic [2:0] buffer_ind;
always_ff @(posedge clk)begin
  buffer[buffer_ind] <= bit_in;
  if (buffer_ind > BUFFER_LIMIT)begin
    buffer_ind <= 0;
  end else begin
    buffer_ind <= buffer_in +1;
  end
end
```

- Oh but backwards

```
logic [7:0] buffer;
logic [2:0] buffer_ind;
always_ff @(posedge clk)begin
  buffer[7-buffer_ind] <= bit_in;
  if (buffer_ind > BUFFER_LIMIT)begin
    buffer_ind <= 0;
  end else begin
    buffer_ind <= buffer_in +1;
  end
end
```

6 hours later…

- Your code is full of stuff like this

```
buffer_ind <= 2+ BUFFER_LIMIT - buffer_ind - 1 + new_old_buffer_ind_a;
```

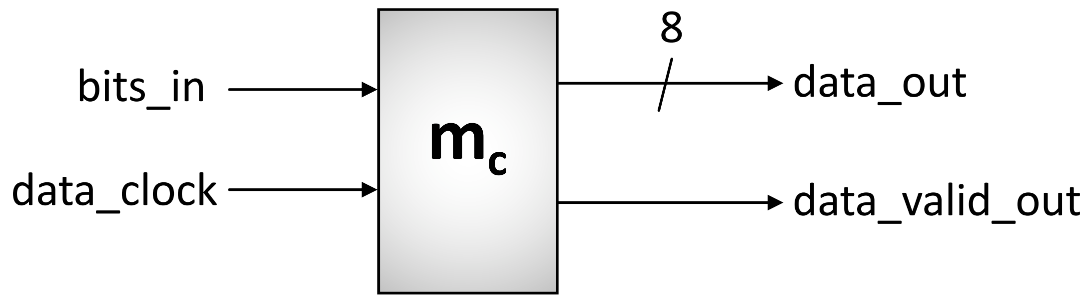
- You’ve gotten this gross functionality “working” but it may not be beneficial for the problem at hand.

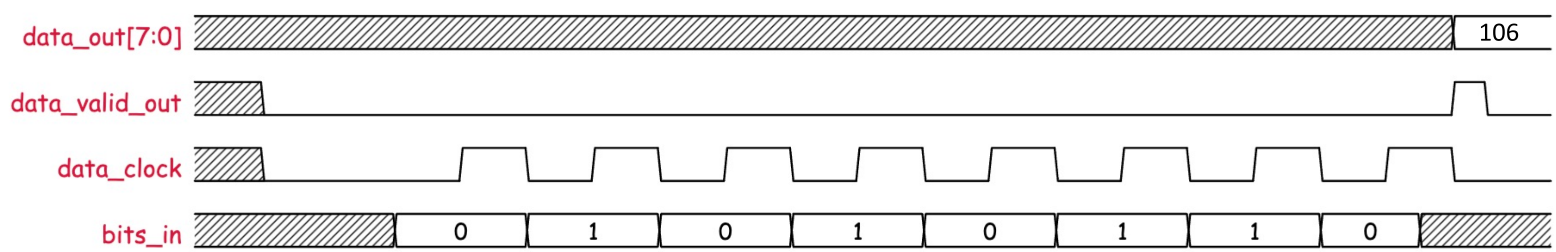
What do indexable arrays give us?

- “Random” Access to an array (*in other words, at any point in time I can access/modify any element of the array*)
- As we’ll see, large-scale Randomly accessible memory is a burden. You do not want to use it unless you need to!
- Do we need that here?

Reconsider this task: Data Deserializer

- The bits are coming in in order of lsb-first
- They're only ever going to go in one place, and we'll use them in order they come in….we don't need array access.





Make a queue or fifo-like structure

- Just push the data in as it comes in

```
logic [7:0] buffer;
always_ff @(posedge clk)begin
  buffer <= {buffer[6:0], bit_in};
end
```

- Oh is that backwards? No biggie…

```
logic [7:0] buffer;
always_ff @(posedge clk)begin
  buffer <= {bit_in, buffer[7:1]};
end
```

“SHIFT BUFFER”….a simple FIFO for bits!

Another Thing: / and %

- We’ve done nothing in this class so far that needs these two operators.
- In the land of digital design, / and % should be avoided at all costs until they are absolutely needed.
- They are *extremely expensive* operations to perform
  - We’ll see how expensive they are in future lectures and labs.
- No need to use these for a cycle counter or for anything so far

The Cycle Counter from Lab 01

- Build a thing that starts at zero and counts up to a number, then goes back to 0.
- Every clock cycle you are asking this thing to perform 32 bit integer division and find the remainder….that is a monumental task to just count a number and wrap it around

```
module counter( input wire clk,
                input wire [31:0] period,
                output logic [31:0] count
               );
  always_ff @(posedge clk)begin
    count <= (count+1) % period ;
  end
endmodule
```

Simpler, Cheaper

- A 32 bit add, a 32 bit compare, an if/else

```
module counter( input wire clk,
                input wire [31:0] period,
                output logic [31:0] count
               );
  always_ff @(posedge clk)begin
    if (count+1 >= period)begin
      count <= 0;
    end else begin
      count <= count + 1;
    end
  end
endmodule
```

Now in some cases…

- The tools may be able to optimize an atrocious line like this one for you, but that can depend on things it knows…and it doesn't know all the stuff you know.

```
module counter( input wire clk,
                input wire [31:0] period,
                output logic [31:0] count
               );
  always_ff @(posedge clk)begin
    count <= (count+1) % period ;
  end
endmodule
```

The tool doesn't really know that count will never be greater than period…it will likely synthesize a device that can do % for all possible count start values*

And at the very least…

- The closer your Verilog matches what should get built, the less you’re asking of the tool.

- Tools will always let you down so you want to help them out as much as possible.

Simulation vs. Reality

- / and % may work in simulation, but likely not in real life.
- Be aware of that.
- You can compute pi to 1000 digits "instantaneously in simulation"…that does not mean it can be done in real life

Asynchronous vs. synchronous reset

- There's very little to no reason to have an asynchronous reset in our class, especially right now

Don't need this. Don't' do:

```
module thing( input wire clk,
              input wire rst,
             );
  always_ff @(posedge clk || posedge rst)begin
    if (rst)begin
      //reset stuff
    end else begin
      //do normal stuff
    end
  end
endmodule
```

Asynchronous vs. <u>synchronous reset</u>

- Just have the flip flop sensitivity list be the positive edge of the clock

```
module thing( input wire clk,
              input wire rst,
             );
  always_ff @(posedge clk)begin
    if (rst)begin
      //reset stuff
    end else begin
      //do normal stuff
    end
  end
endmodule
```

Only thing clocking a Flip Flop should be our *high-speed clock*

- Do not have numerous sequential numerous blocks all being clocked by different signals

HORRIBLE, BAD, DO NOT DO:

```
always_ff @(posedge a)begin
  //stuff
end
always_ff @(posedge b)begin
  //other stuff
end
always_ff @(posedge c)begin
  //other other stuff
end
```

Can make simulations mismatch reality
Can make designs not meet timing and fail
Will be cludge code that will hurt you

INSTEAD DO:

```
always_ff @(posedge clk)begin
  if (a)begin
    //stuff
  end else if (b) begin
    //other stuff
  end else if (c) begin
    //other other stuff
  end
end
```

Reliable Design Practice
Simulations more likely to match reality
Timing easier to meet

Or if you really need things to happen on the “edge” of a non-clock signal…

- Remember old signal values and compare

```
always_ff @(posedge clk)begin
  old_a <= a;
  old_b <= b;
  old_c <= c;
  if (a && !old_a)begin //on the rising edge of a
    //stuff
  end else if (b && !old_b)begin //on the rising edge of b
    //other stuff
  end else if (c && !old_c) begin //on the rising edge of c
    //other other stuff
  end
end
```

Clocks are Special

- Clock signals get special treatment inside the FPGA
- Get to priority routing, go down special “clock lines” to minimize skew (future class)
- Making lots of signals “clocks” can cause congestion and the entire design to fail

Sequential Logic

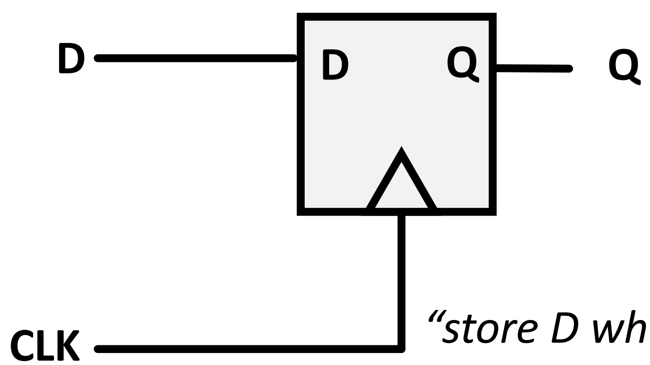
Registers, Latches, and Flip-Flops

- The terminology is a mess for historical reasons and just people in general, including myself. Here's one interpretation:
- A "register" is something that holds a value. Flip-flops and Latches *are* registers
- Further confusing the situation, people, including myself, often use "register" or "reg" to just refer to flip-flops

Use a lot!

D Flip-Flop

Edge-Triggered Sample-and-Hold Device

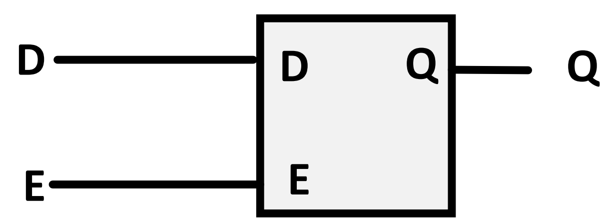


"store D when clk rises"

Won't use as much, if ever

D Latch

Level-Triggered Sample-and-Hold Device



"store D when E is high"

D Flip-Flop Registers Give Us A Few Critical Capabilities

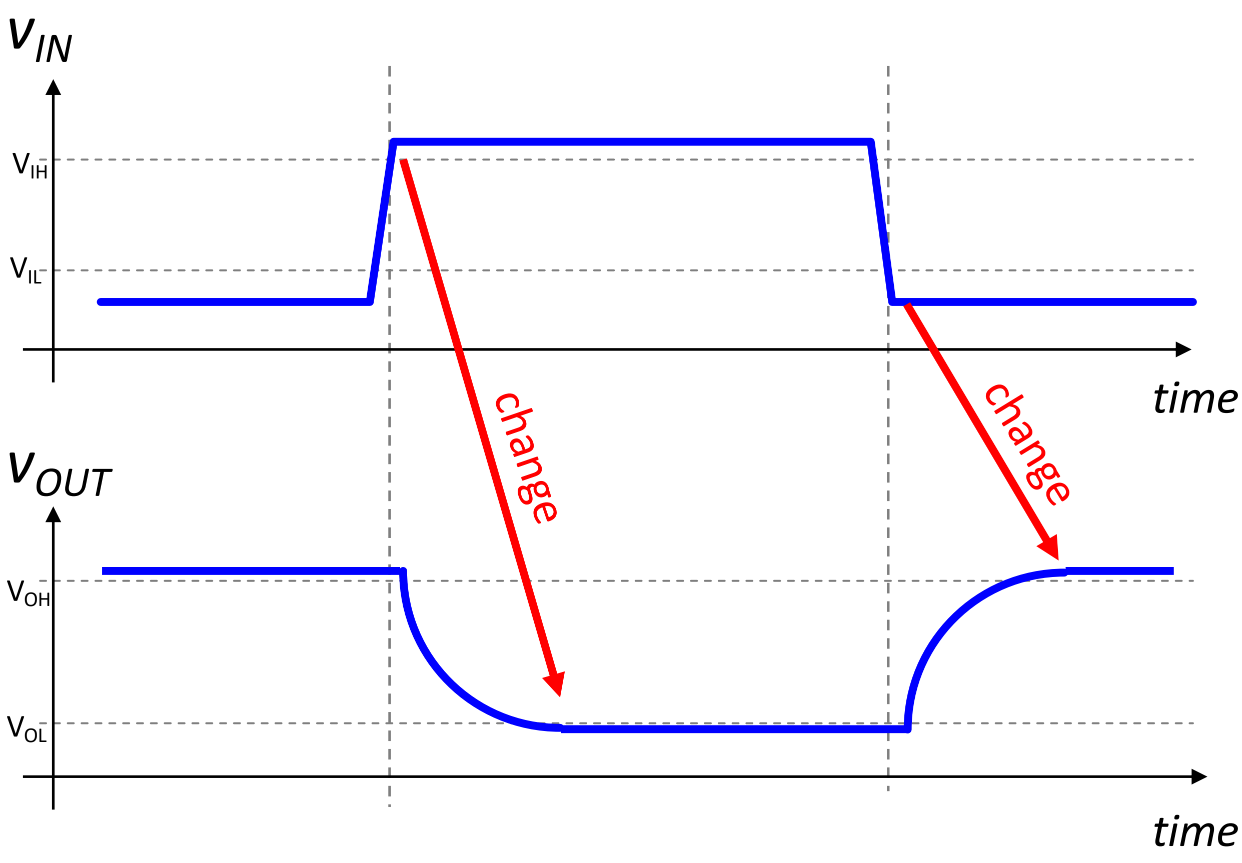
- We can store values for later use (simple memory)
- We can sample values at precise times
  - *A rising edge is as close to a delta-function like event as we can get*
- We can design in stages:
  - Allow us to non-destructively limit signal propagation which prevents:
    - Combinational loops (last week)
    - Glitches (today)

All Electronics are Non-Ideal

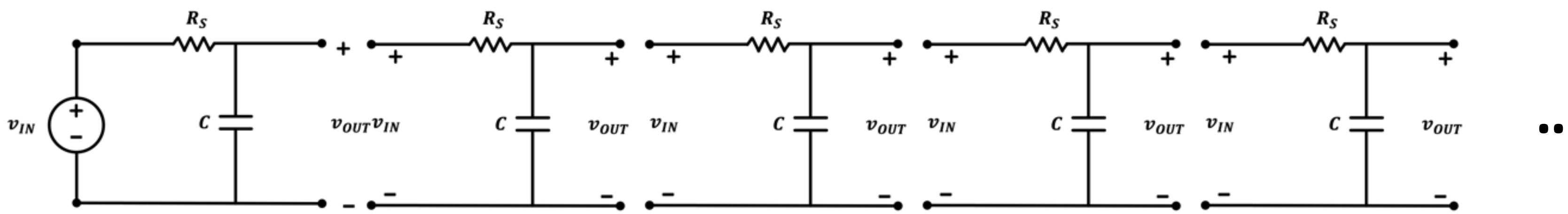
- Inherent to the logic is the need to charge and/or discharge parasitic capacitances and inductances through non-0 value resistances
- As 8.02, or 6.200/6.002 will have shown, this has an inherent time constant involved with it
- …meaning a finite time at which it will respond given a change
- Obviously we don't want this, but I didn't want MIT to start using Okta for login stuff. What are you going to do? So it goes

When one digital circuit drives another digital circuit

- Inputs change outputs…but takes time



The more complex/more layers/the more delay you'll get



- Each individual "stage" needs to charge up/down before it can influence the next stage.
- Very complicated/deep logic will take time

It'll take time to transition

- Response of a function will take time (and energy)
- So if we move around on a truth table it can't be instantaneous

If you're ***here*** *on the truth table*

Truth Table

c	b	a	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

How much time?

And transition to ***here***

Truth Table

c	b	a	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

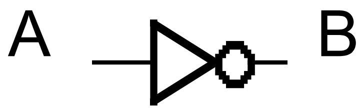
Digital Delays

- For a given digital device, we need to quantify the delay

- Utilize two different numbers:

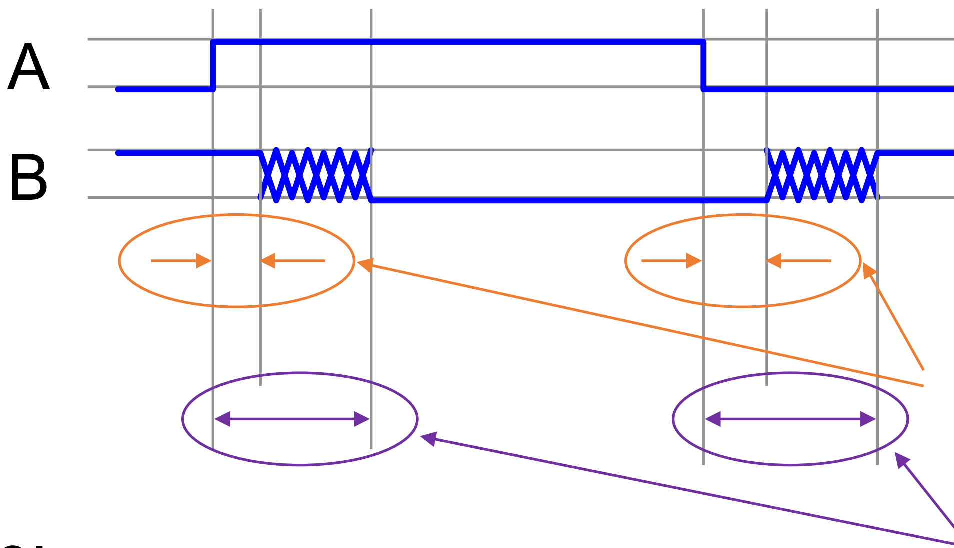
- For a given change at the inputs to a digital system:
  - **Contamination Delay (t_{cd}):** How long before the system will start to respond at its output?
  - **Propagation Delay (t_{pd}):** How long until we can be sure the system has updated to new value (stabilized)?

The Combinational Contract



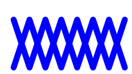
A	B
0	1
1	0

t_{PD} propagation delay
t_{CD} contamination delay



Must be ____**> t_{CD}**____

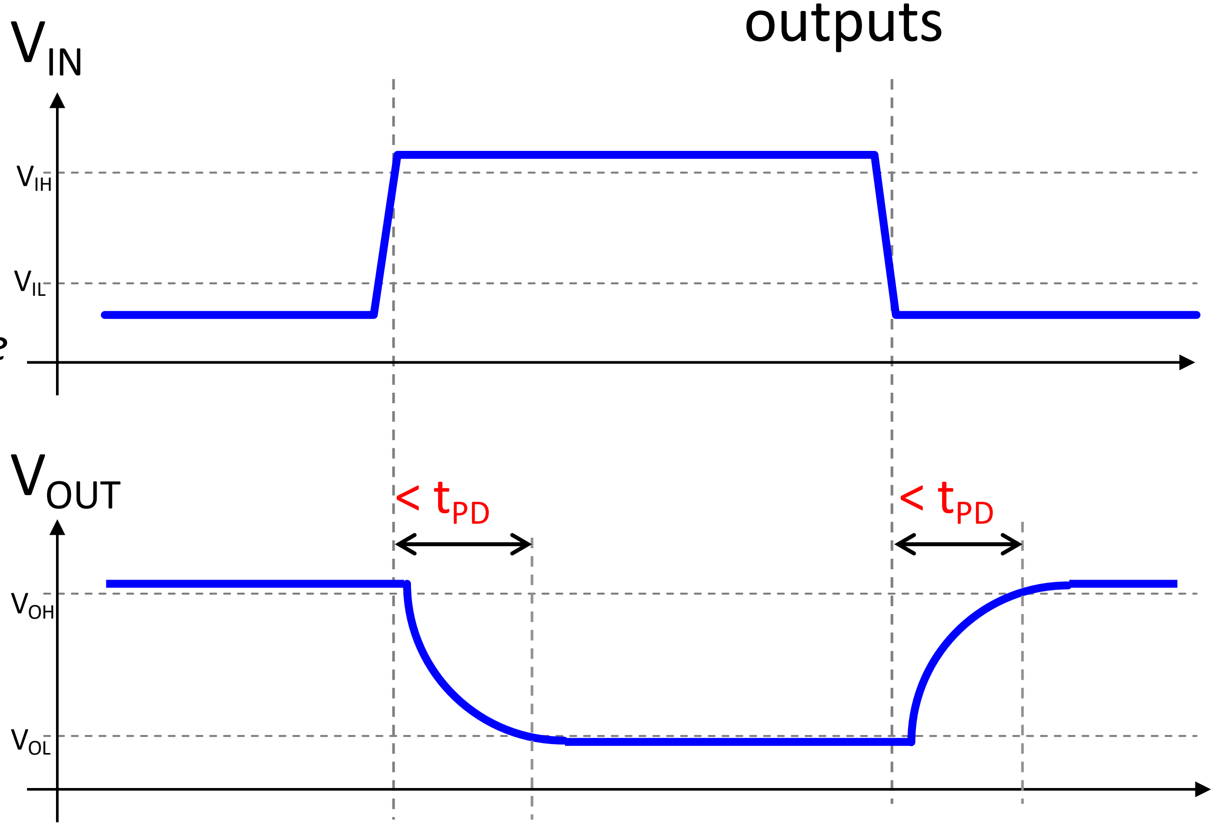
Must be ____**< t_{PD}**____

Note:
1. No Promises during 
2. Default (conservative) spec: $t_{CD} = 0$

Worst Case: Propagation Delay

Propagation delay (t_{PD}): An upper bound on the delay from valid inputs to valid outputs

NOT Gate Demo:

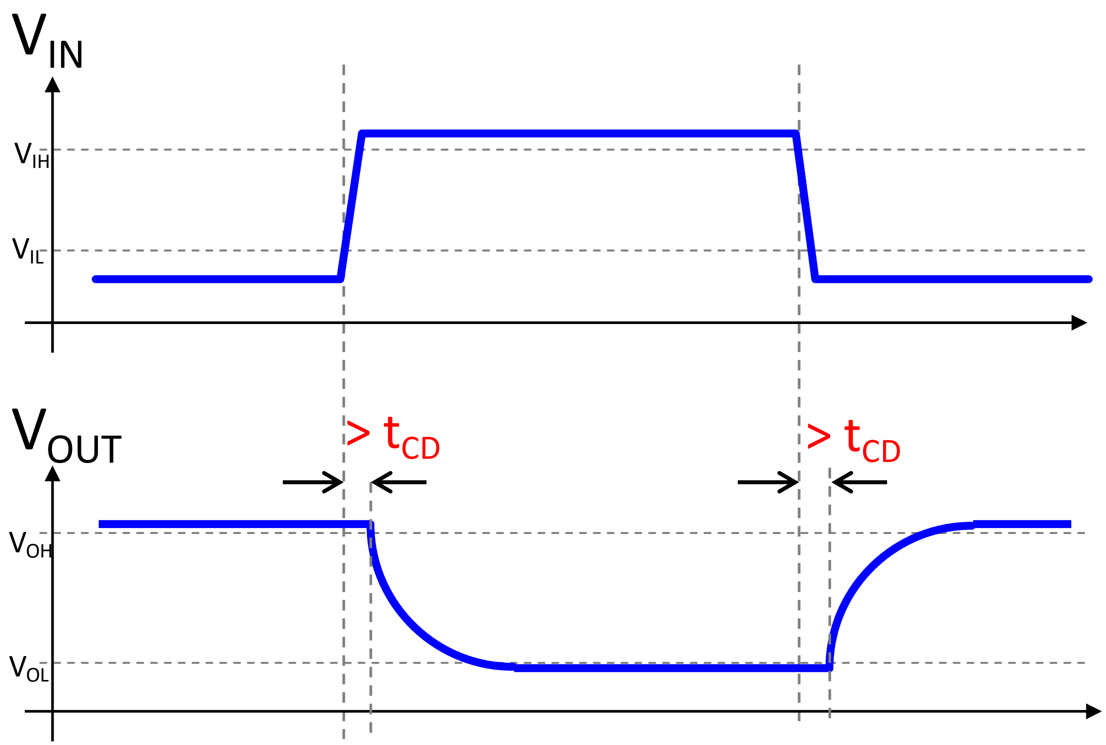


Design goal:
minimize propagation delay

Best Case: Contamination Delay

Contamination delay(t_{CD}): A lower bound on the delay from invalid inputs to invalid outputs

NOT Gate Demo:

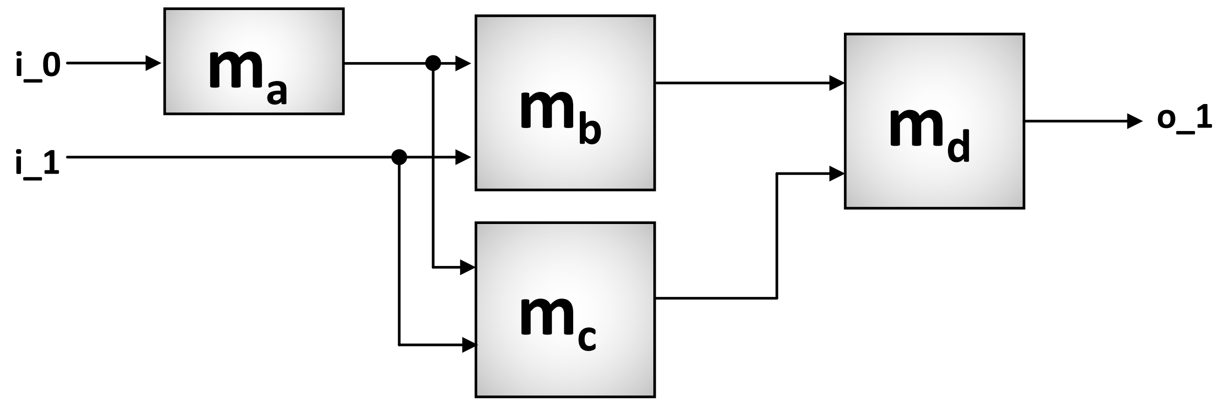


Do we really need t_{CD}?

Sometimes yes, sometimes no… it'll be important when we design circuits with flops (coming next!)

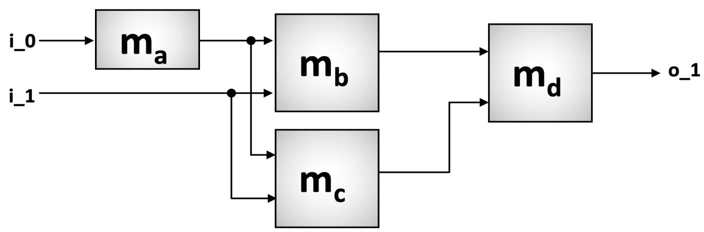
If t_{CD} is not specified, safe to assume it's 0.

Review: Example System

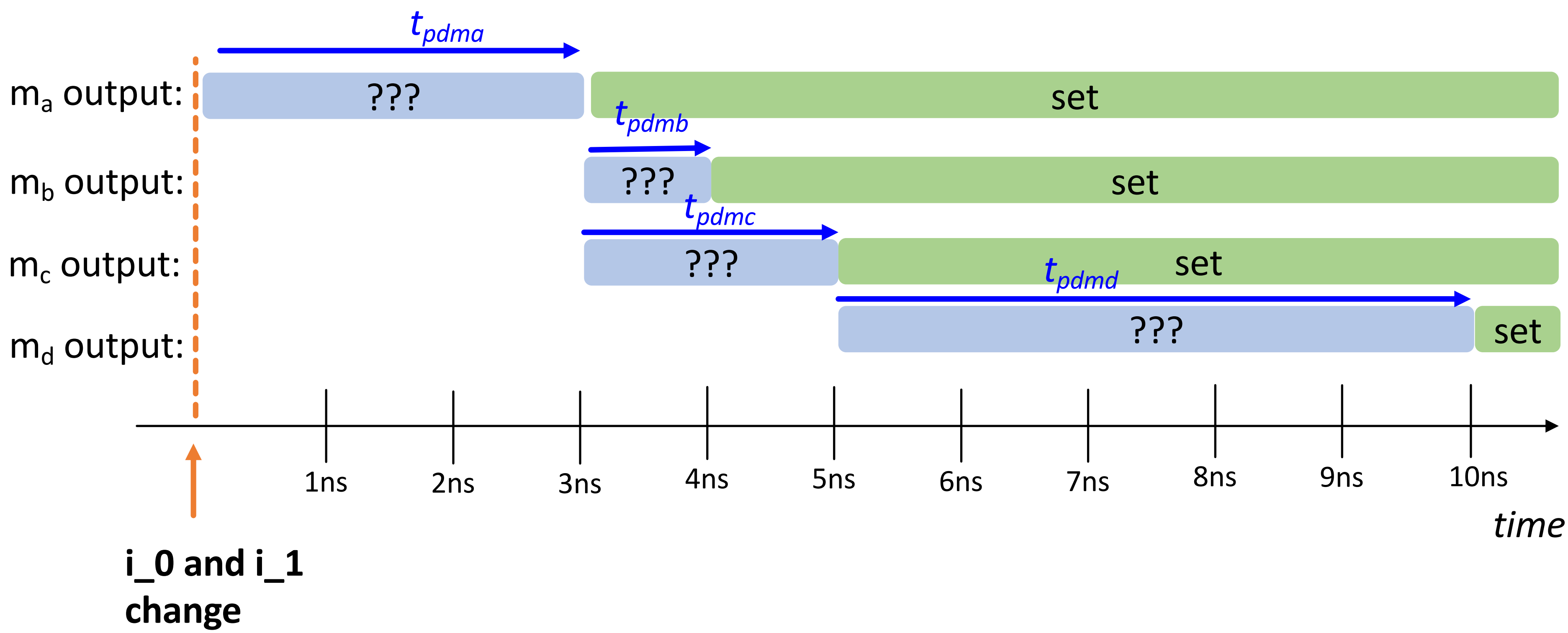


- Let's assume:
  - m_a has t_{pd} = 3ns
  - m_b has t_{pd} = 1ns
  - m_c has t_{pd} = 2ns
  - m_d has t_{pd} = 5ns
  - All four modules have t_{cd} = 0ns
- What is:
  - i_0 to o_1 t_{pd}?
  - i_1 to o_1 t_{pd}?
  - t_{cd} of system?
  - Critical Path of the system and t_{pd}?

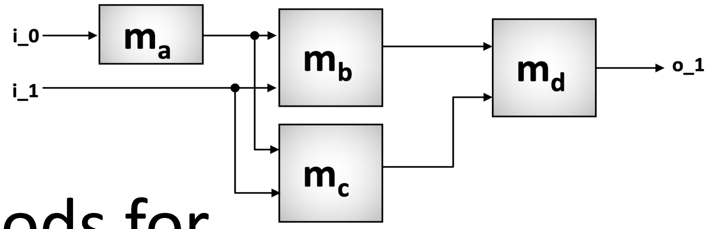
Timing Diagram



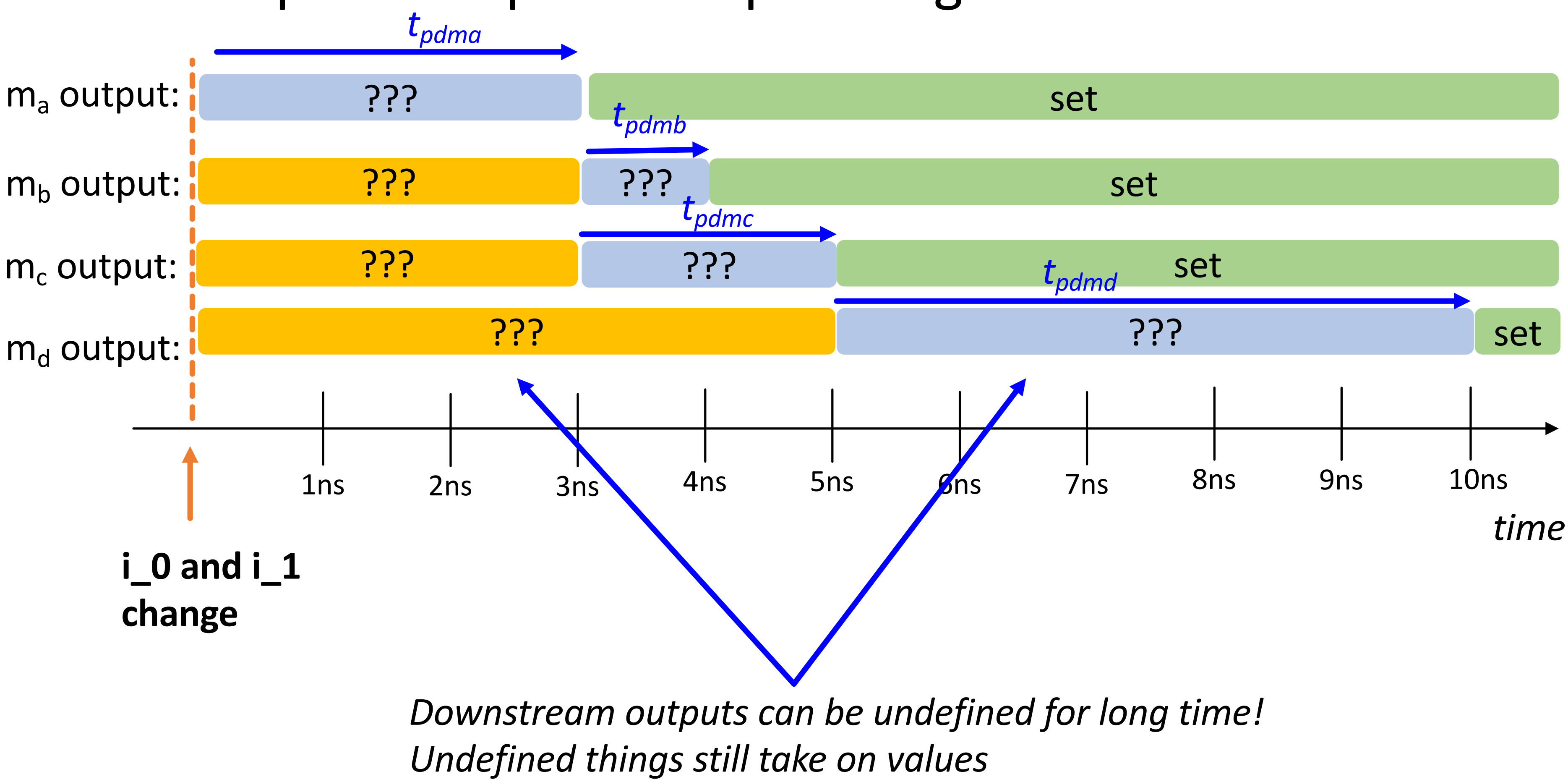
- The t_{pd} on any stage/module can't start being used until all inputs to it are set/stable:



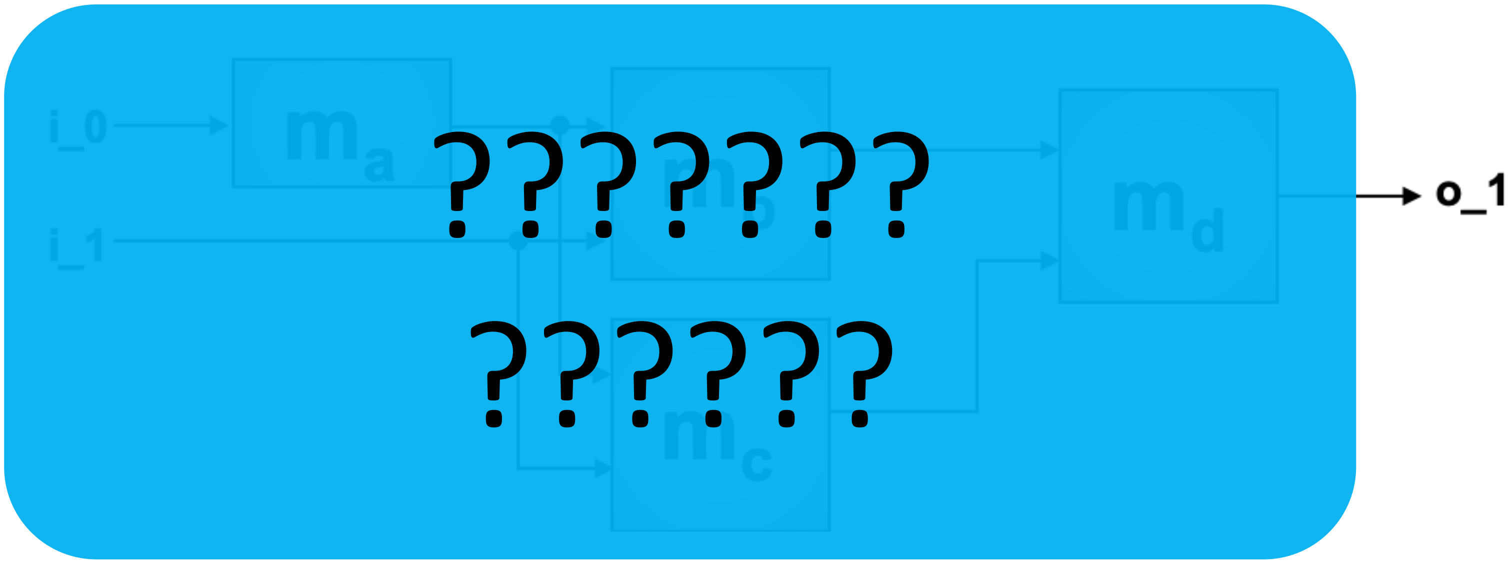
Timing Diagram



- Additionally: the "unknown" periods for subsequent outputs are quite large



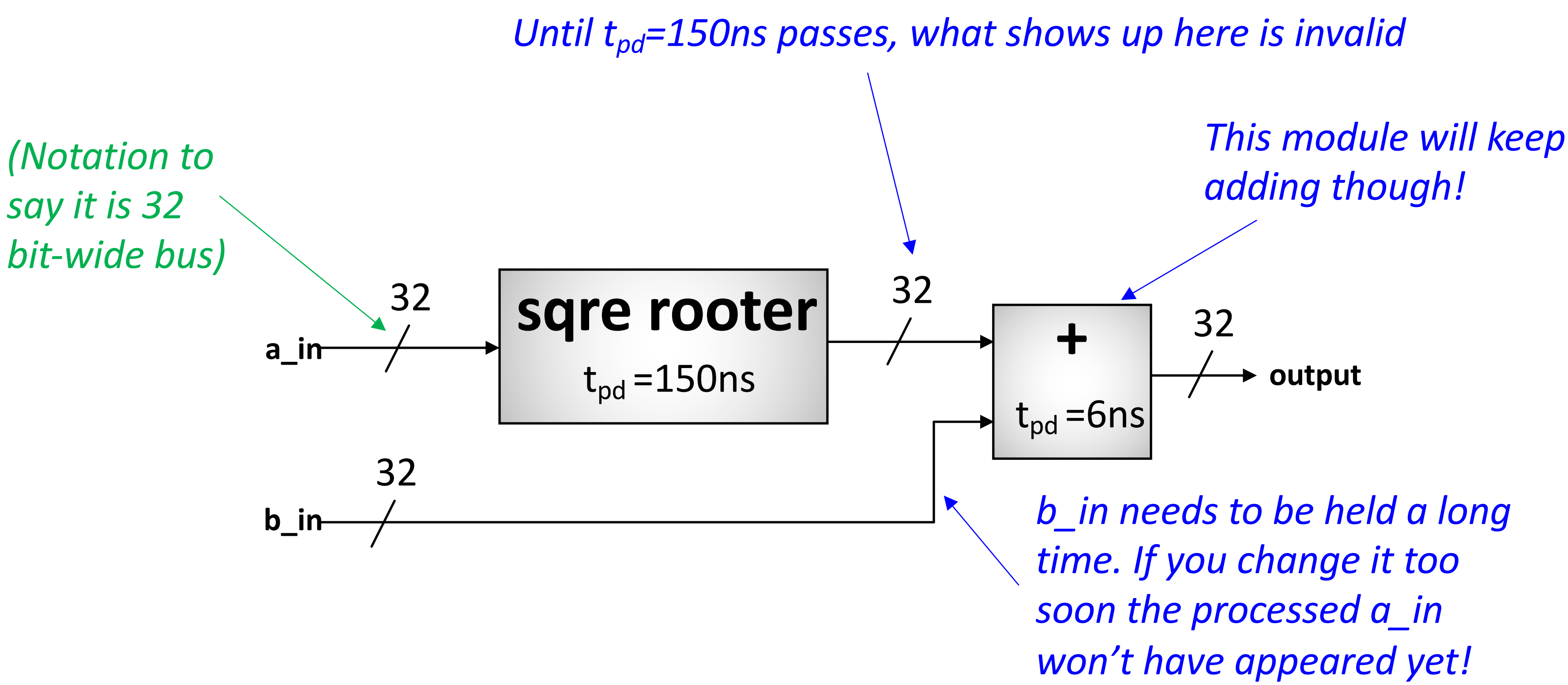
From the Outside



- If all you see is *o_1* how can you actually determine what is valid and what isn't?
- Is that **1'b1** on *o_1* valid or invalid? Who knows?
- Unless you know when you put in values and know the total t_{pd} of system very hard to discern what is good and what isn't.

Another Way to Look At Problem

- You have a system, takes in two numbers, a_in and b_in and produces an output.
- System calculates square root of a_in and then adds b_in to it



Another Problem (being a real downer today, I know…)

- Consider simple addition in binary (or any base):

```
 11
 10110
+00011
------
 11001
```

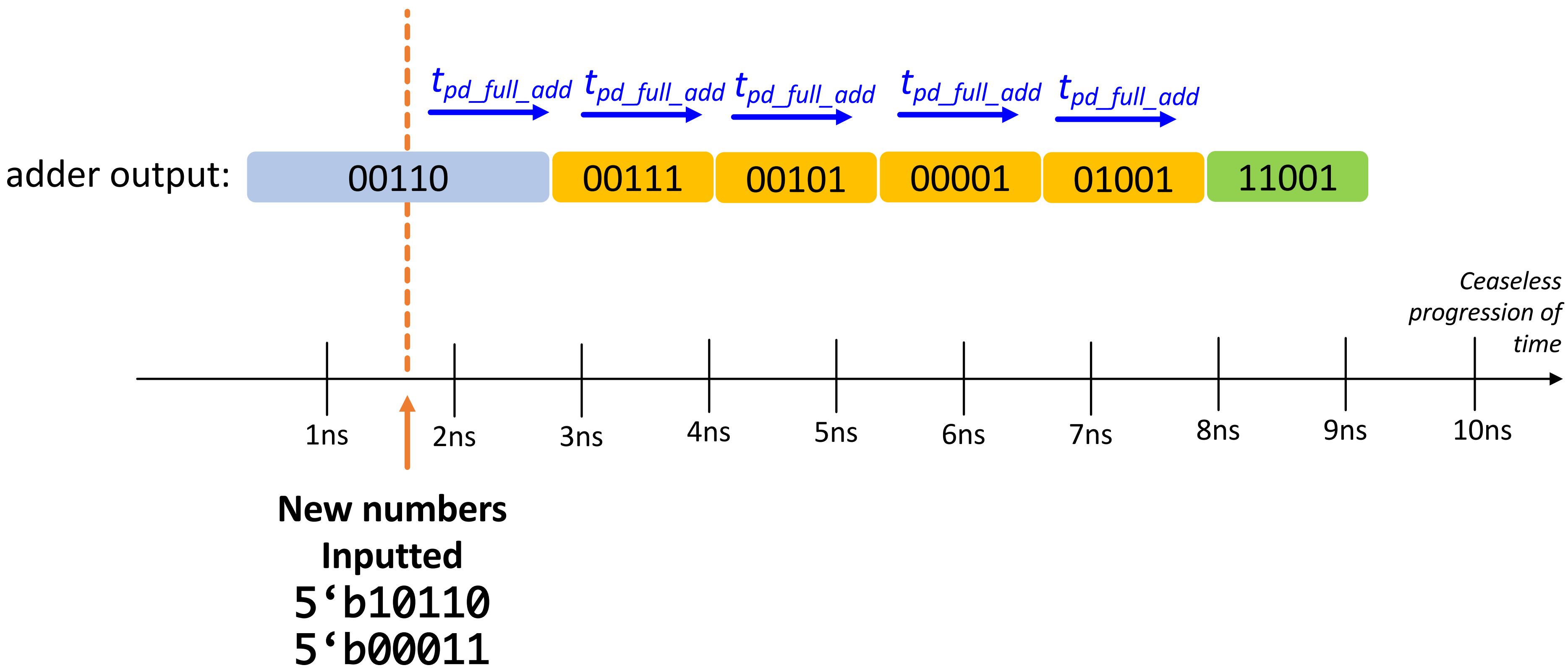
Math checks out:

```
  22
+  3
------
  25
```

- Notice how we need to calculate the lower digits first before we calculate the upper digits?
- Uh Oh…

Timing Diagram of Add

- Lots of invalids before the valid result!



What if we then had this circuit?

i_0
i_1
+
val

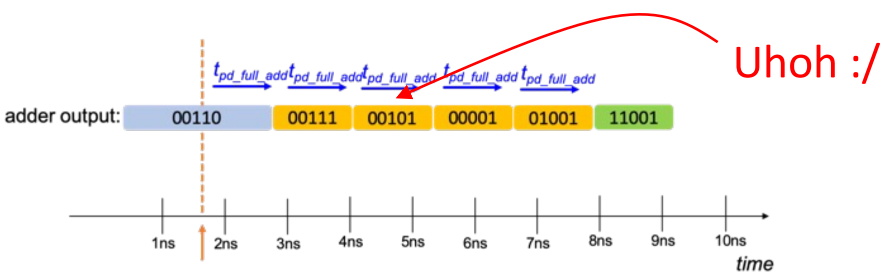
```
module grand_deity (input wire [4:0] val,
                    output logic destroy_world);

 assign destroy_world = val == 5'b00101;

endmodule
```

destroy_world

Adder from previous page

0

1



Combinational Glitches!

- Combinational glitches arise when outputs transition through unintended outputs in response to transitioning inputs
- Caused by differences in overall **OR** internal delays of logic

a_in	b_out
0	1
1	1

System should always have 1 as output, but during transitions from 0 → 1 or 1→0, b_out will glitch to 0.

Glitches Will Happen

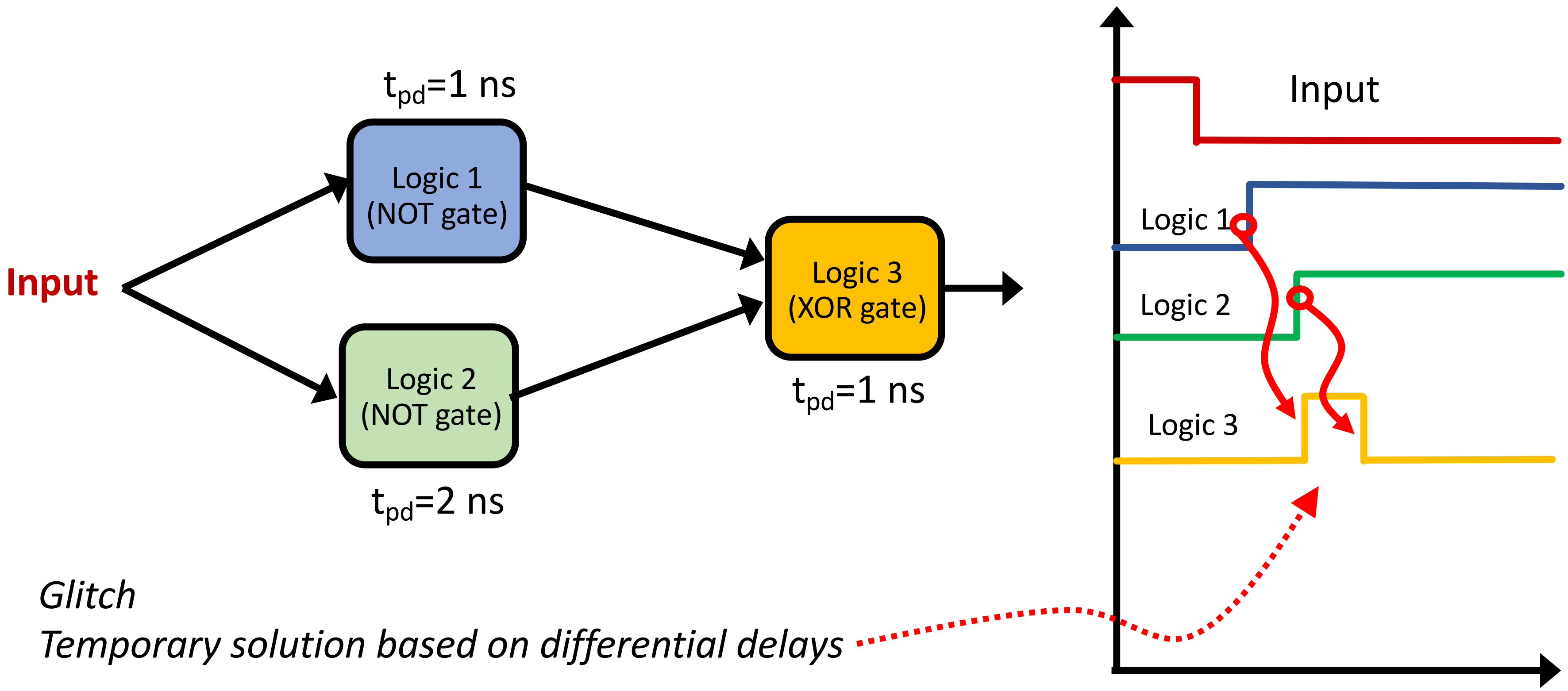
- Inherent when you have complex and DEEP combinational logic!
- Perform calculations on irrelevant information
  - Waste energy
  - Very hard to debug
- Extremely difficult to design with reliably at scale
  - Too many related time constants
  - Too many invalid values
- Our only hope is to encode our data in glitch-minimizing ways and limit the range that combinational glitches can propagate (next up)

Glitches can be hard to find

- Let’s say t_{pd} = 1ns (conservative)
- Human is the consumer
- You push the button…
  - System stabilizes before the photons emitted from the LEDs have even reached your eye
  - Human eye can only detect up to ~0.01s phenomena...lol 6 or 7 orders of magnitude difference
- Basically we can’t appreciate the glitches…but they can be there.

So How to Fix this?

- Every combinational circuit has delays regarding how slowly (or quickly) its outputs change in response to inputs, and this varies based on design/complexity
  - t_{cd} minimum time input takes to start to change output
  - t_{pd} maximum time input takes to finish changing output

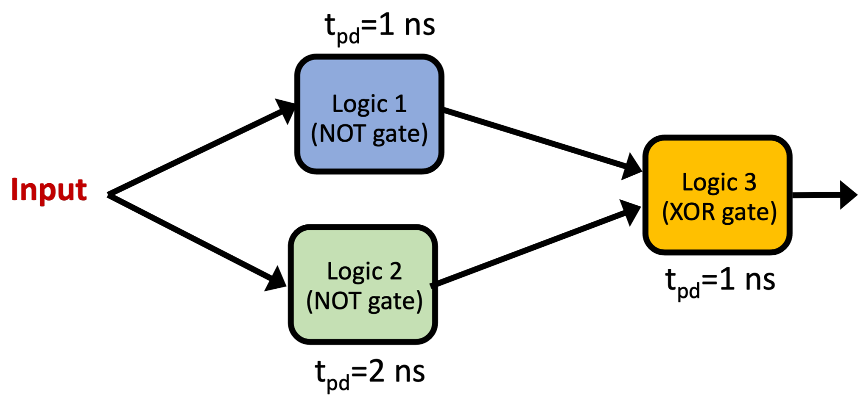


Glitch

Temporary solution based on differential delays

SV

Previous page:

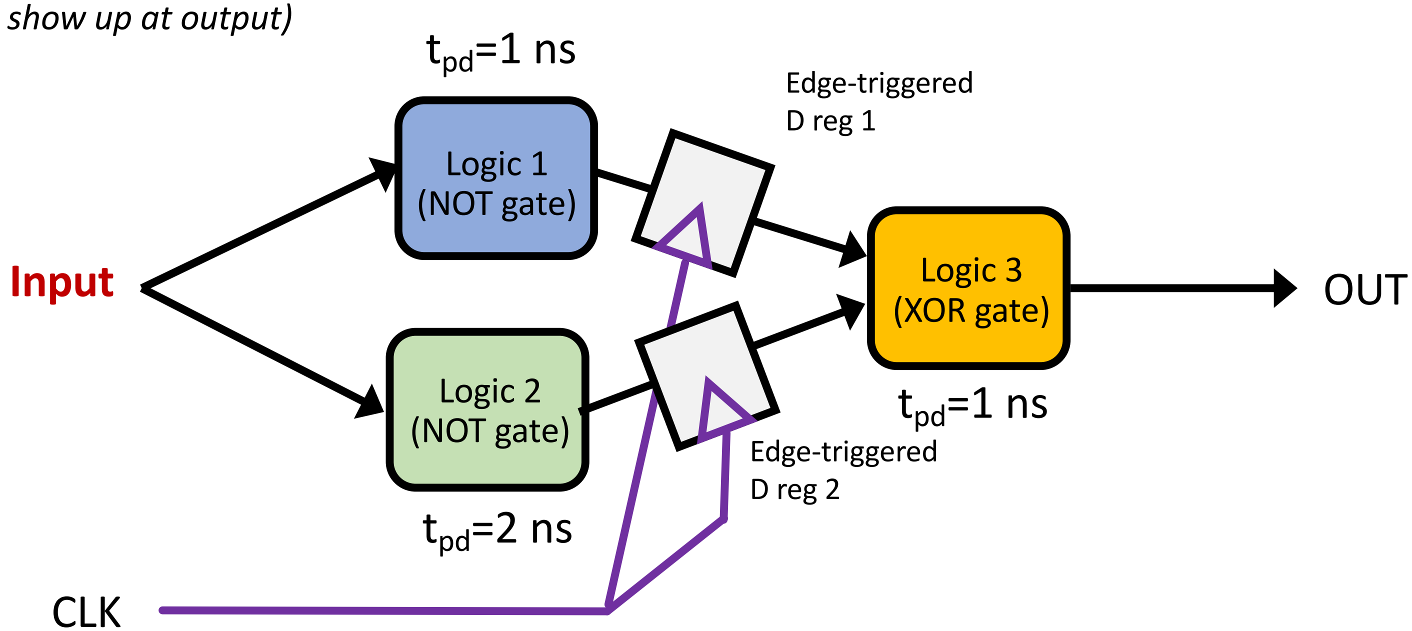



```
logic inputt, outputt;
logic o1, o2, o3;
//assume modules are combinational:
not_gate_a nga(.val_in(inputt), .val_out(o1));
not_gate_b ngb(.val_in(inputt)), .val_out(o2));
xor_gate xg (.vala_in(o1), .valb_in(o2), val_out(o3));

assign outputt = o3;
```

This is How We Fix This

- Registers let us isolate/limit signal propagation and synchronize stages

t_{pd} is propagation delay (how long input takes to show up at output)

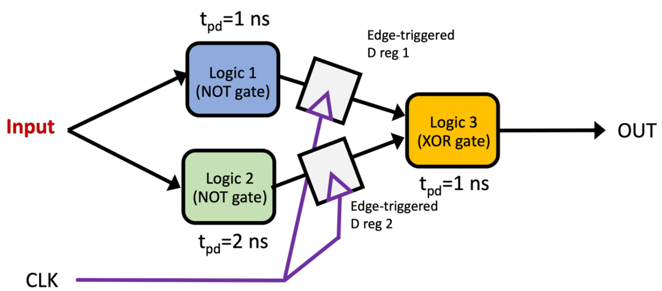


CLK is a synchronization signal

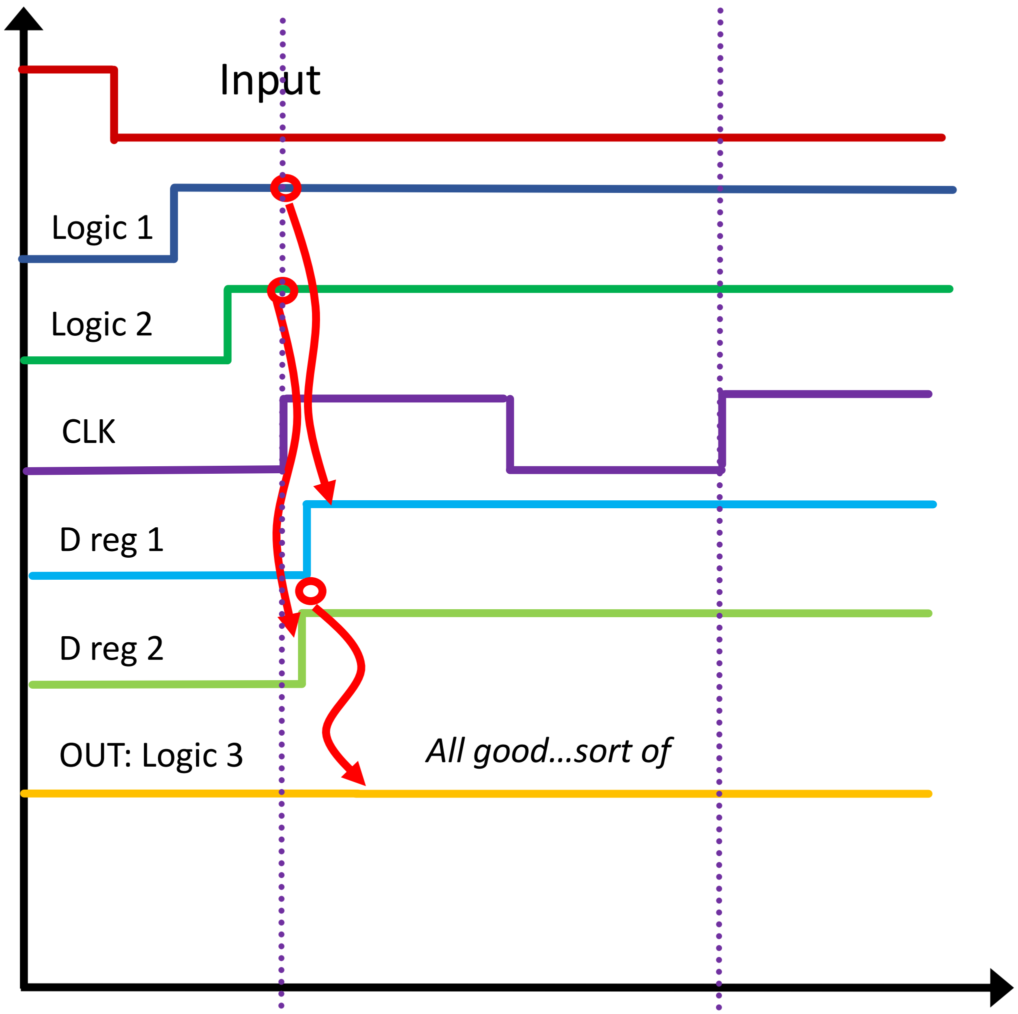
Solution 1

- Balance output

Previous page:



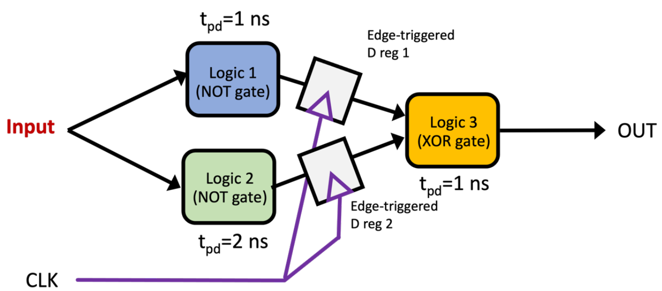
Intermediate glitches are minimized and suppressed in output

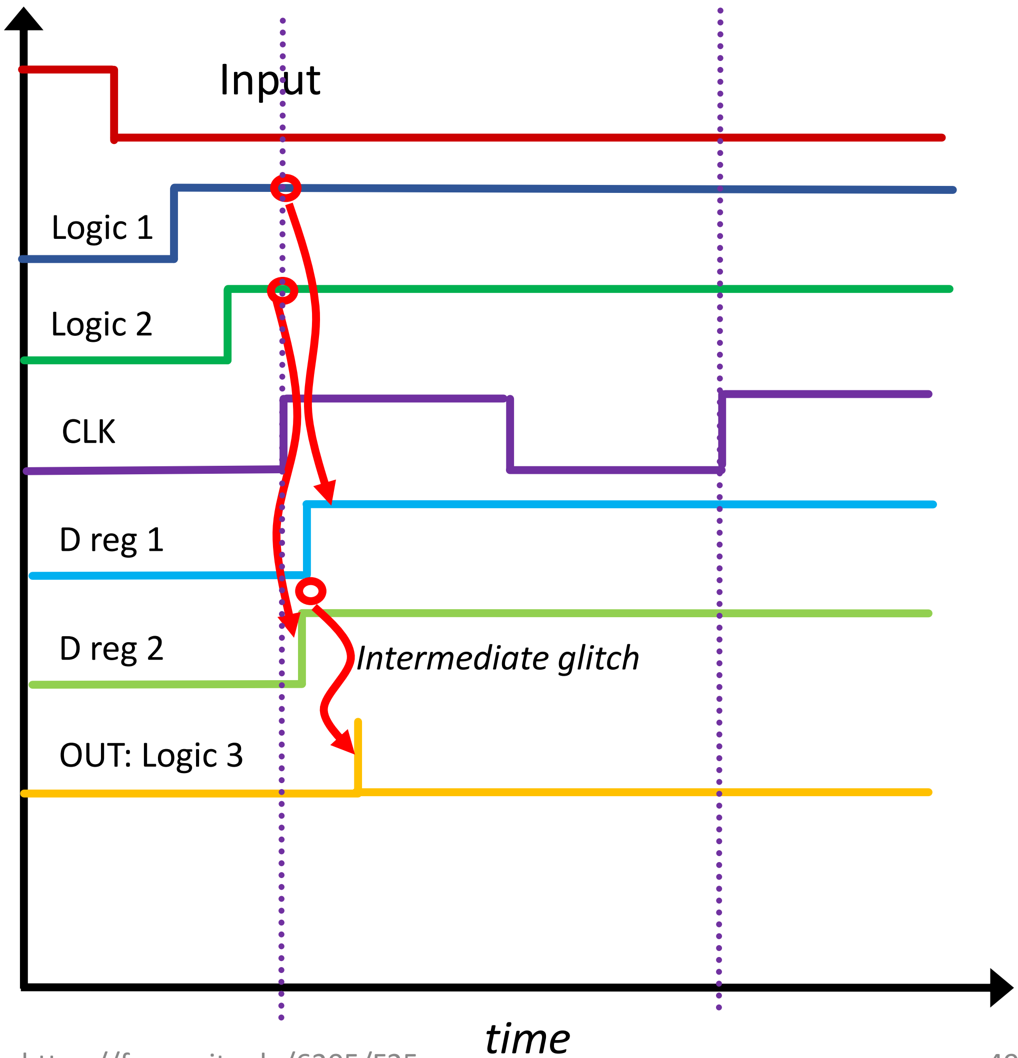


Combinational Logic At Output

- There still could be slight differences in register to XOR routing

Previous page:

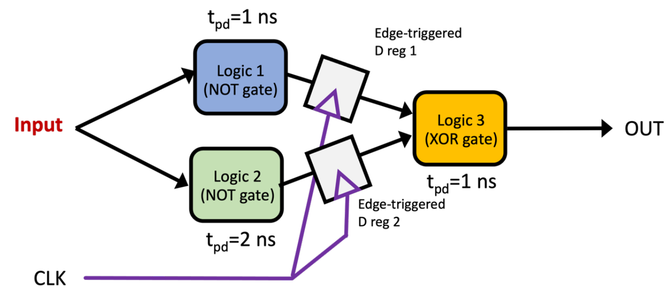




Intermediate glitches are minimized and suppressed in output

SV

Previous page:

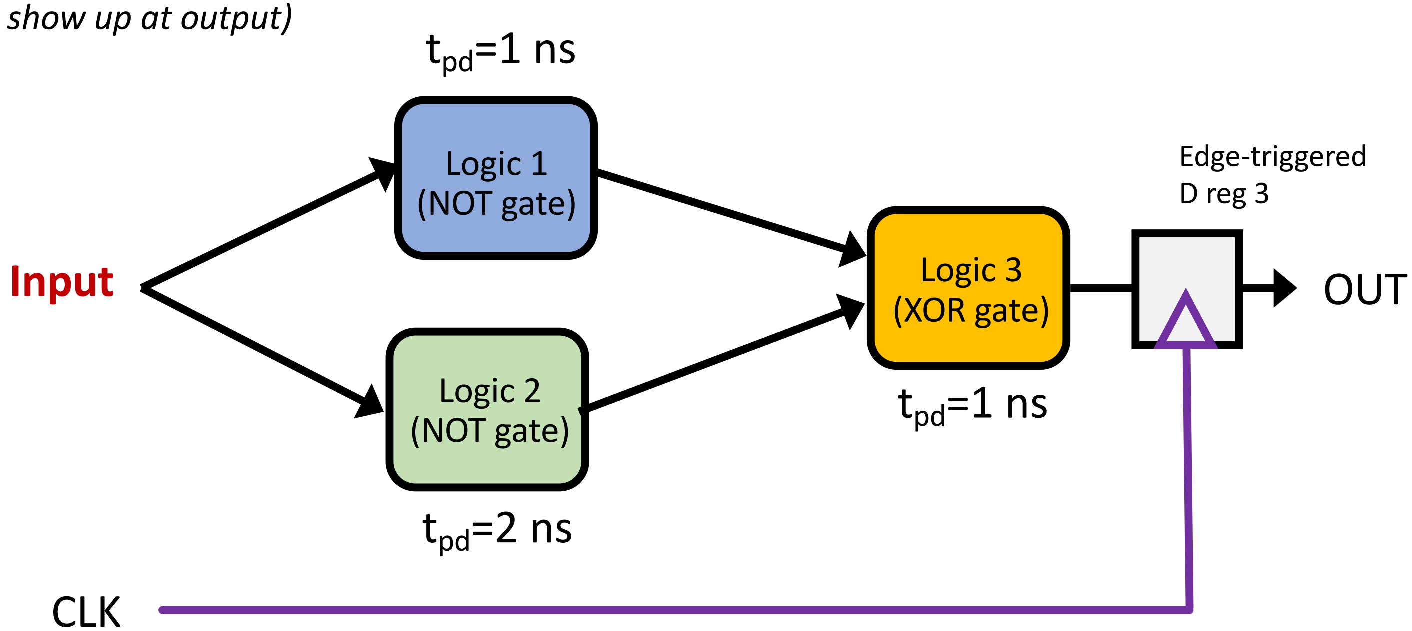



```
logic inputt, outputt;
logic o1, o2, o3;
logic o1r, o2r;

not_gate_a nga(.val_in(inputt), .val_out(o1));
not_gate_b ngb(.val_in(inputt)), .val_out(o2));

always_ff@(posedge clk_in)begin
  o1r <= o1;
  o2r <= o2;
end

xor_gate xg (.vala_in(o1r), .valb_in(o2r), val_out(o3));

assign outputt = o3;
```

This is How We Fix This

- Registers let us isolate/limit signal propagation and synchronize stages

t_{pd} is propagation delay (how long input takes to show up at output)

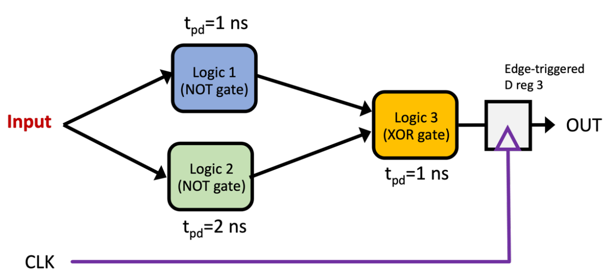


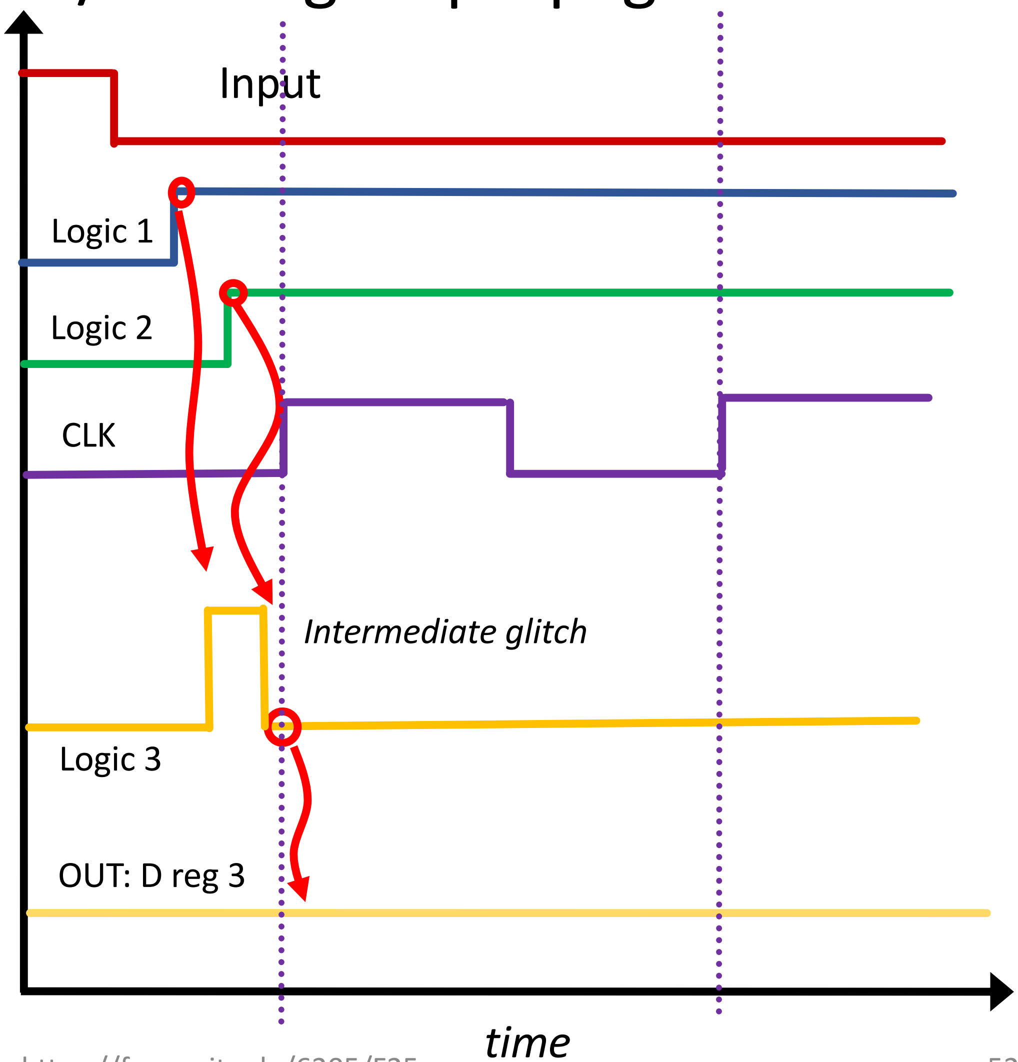
CLK is a synchronization signal

Remember about Delays in Logic

- Registers let us isolate/limit signal propagation and synchronize stages

Previous page:



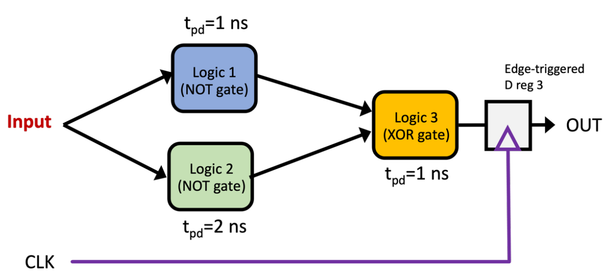


Intermediate glitches are minimized and suppressed in output

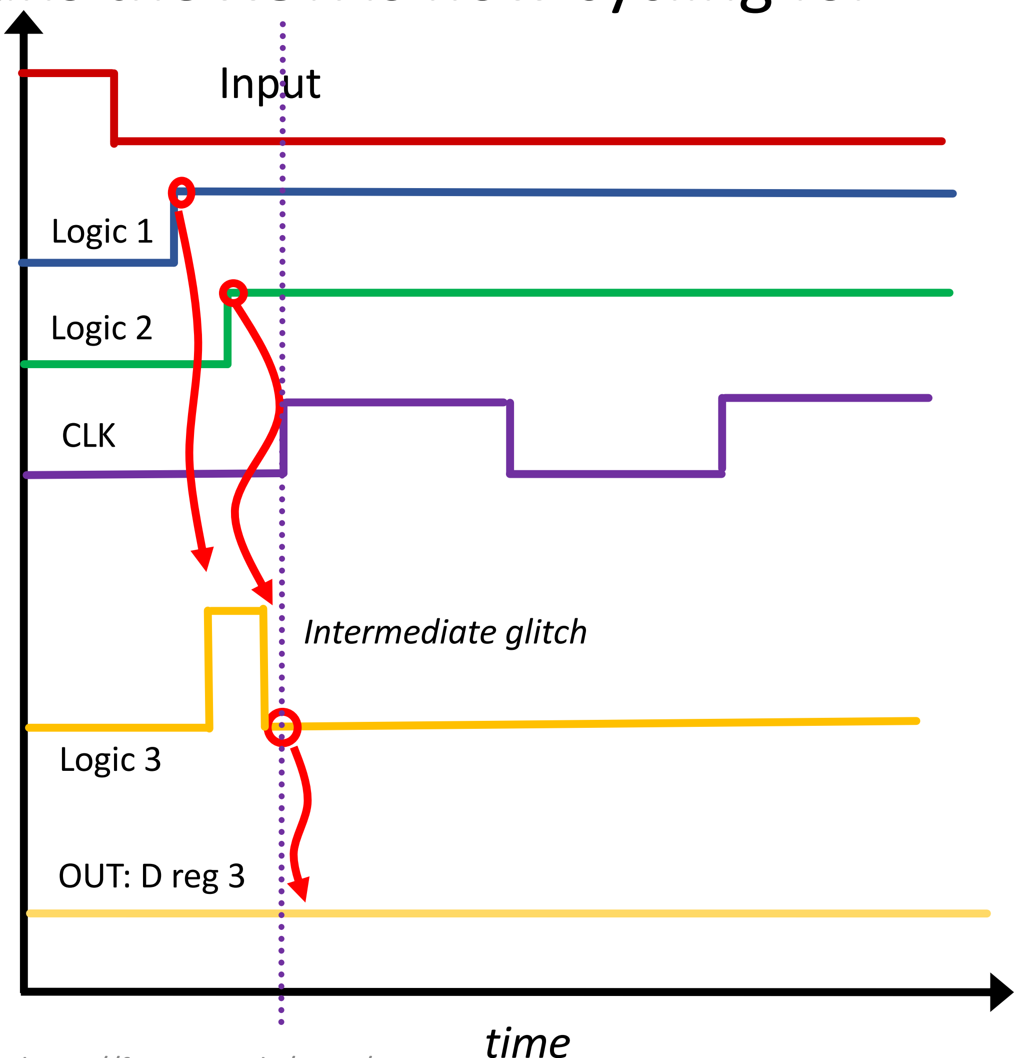
Intermediate Glitches

- Even though that glitch is now internal, the fact that it happens means the XOR is now cycling for no reason…

Previous page:

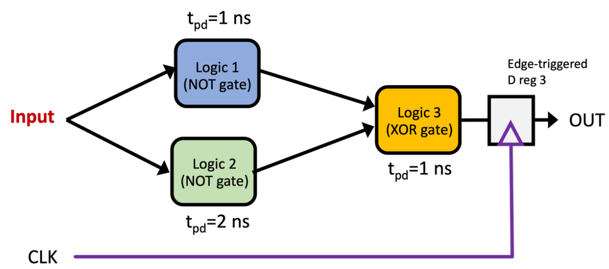


May not be happy with having large internal logic needlessly flipping bits for no reason



SV

Previous page:

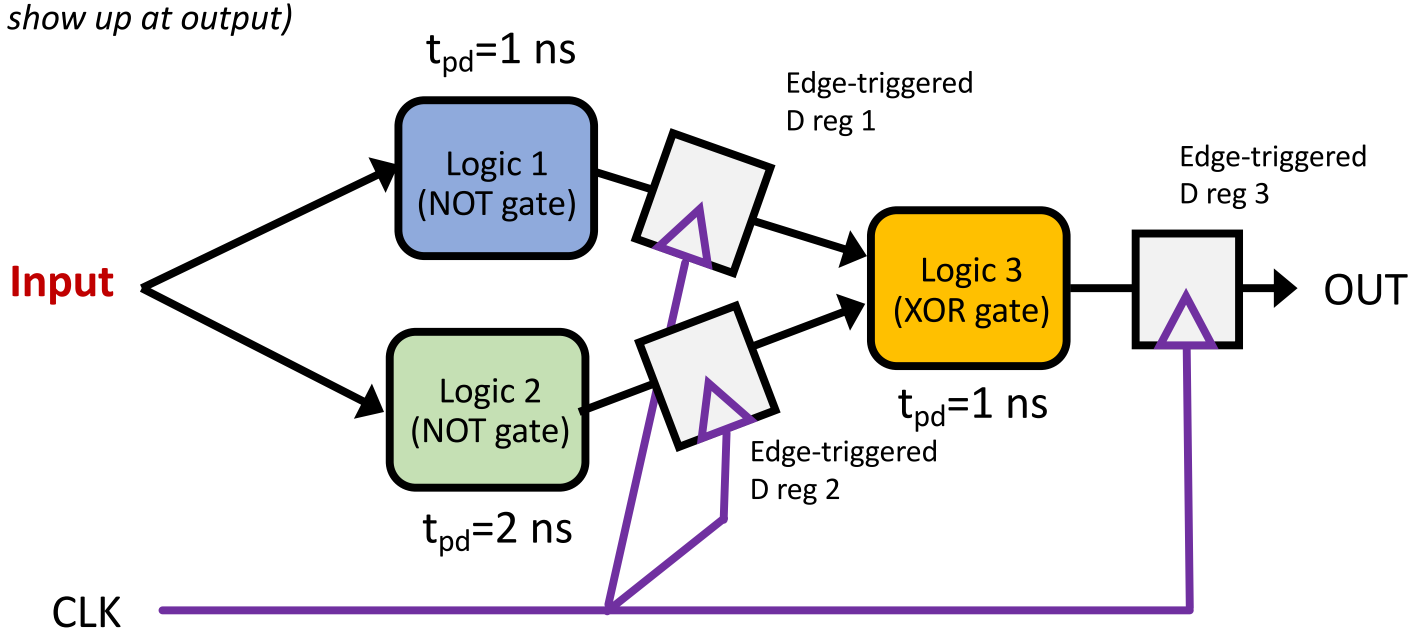



```
logic inputt, outputt;
logic o1, o2, o3;

not_gate_a nga(.val_in(inputt), .val_out(o1));
not_gate_b ngb(.val_in(inputt)), .val_out(o2));
xor_gate xg (.vala_in(o1), .valb_in(o2), val_out(o3));

always_ff@(posedge clk_in)begin
  outputt <= o3;
end
```

Add more

- Registers let us isolate/limit signal propagation and synchronize stages

t_{pd} is propagation delay (how long input takes to show up at output)

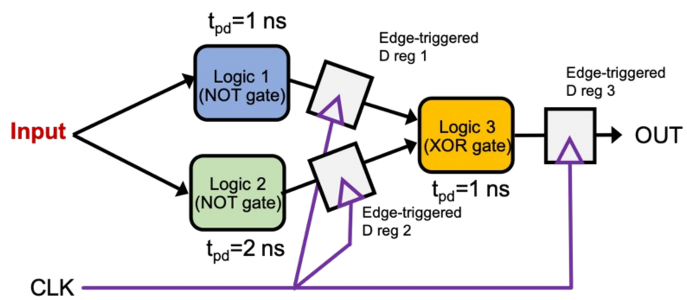


CLK is a synchronization signal

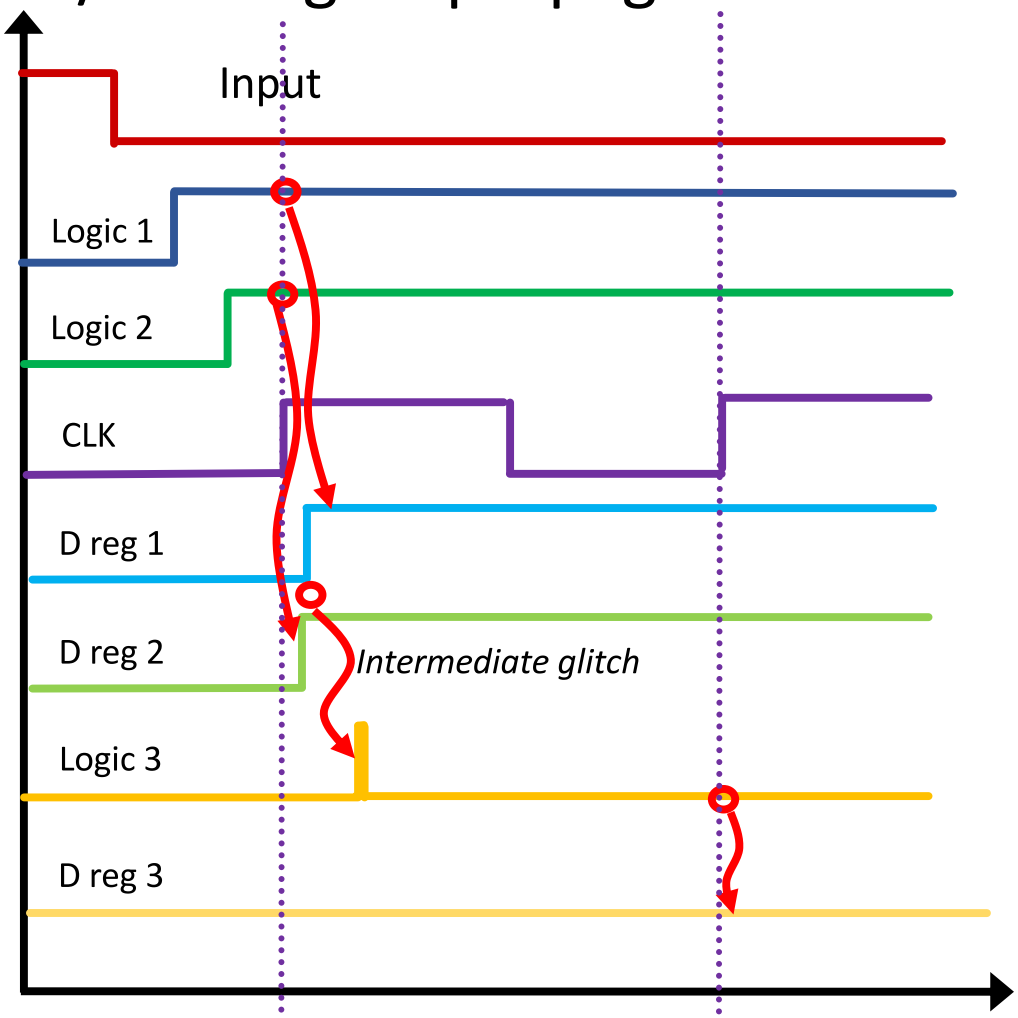
Remember about Delays in Logic

- Registers let us isolate/limit signal propagation and synchronize stages

Previous page:

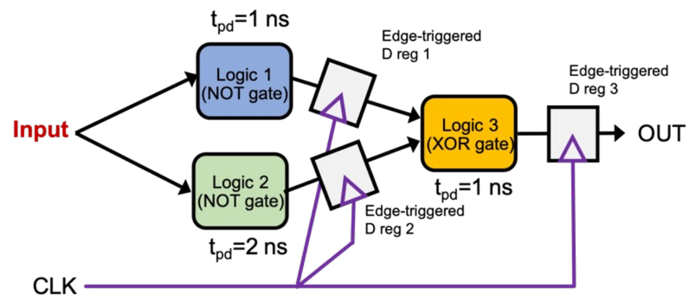


Intermediate glitches are minimized and suppressed in output



SV

Previous page:

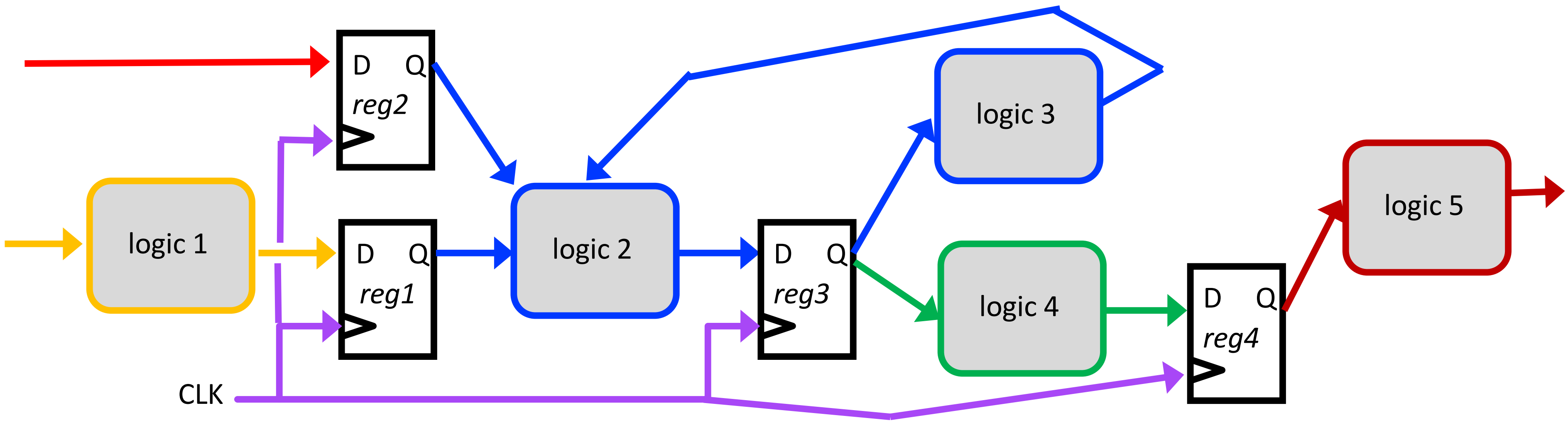



```
logic inputt, outputt;
logic o1, o2, o3;
logic o1r, o2r;

not_gate_a nga(.val_in(inputt), .val_out(o1));
not_gate_b ngb(.val_in(inputt)), .val_out(o2));

always_ff@(posedge clk_in)begin
  o1r <= o1;
  o2r <= o2;
  output <= o3;
end

xor_gate xg (.vala_in(o1r), .valb_in(o2r), val_out(o3));
```

Tradeoff for all this “protection”?

- No free lunch!
- Use More resources
- Have More latency

Design Complex Logic In Stages!



- D flip-flops regulate signal propagation!
- Design complex logic systems in stages
- Worry only about affects of delays (t_{pd} and t_{cd}) and glitches within a given stage, rather than how they all interplay!

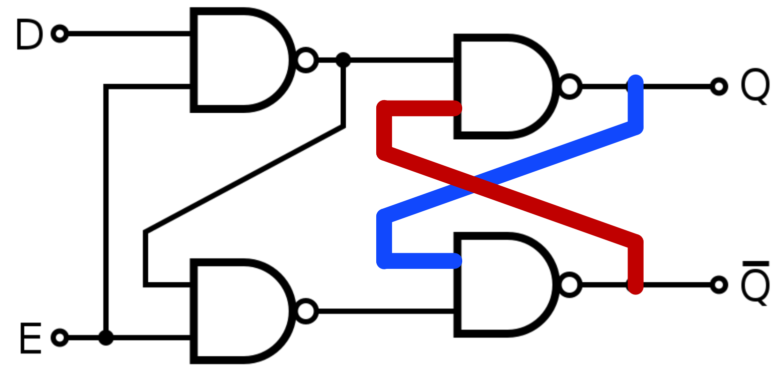
Is that All there is To It?

- No. No there’s not
- Let’s return to how Latches and Flip Flops actually work

The D Latch

- Made of gates (which are made of transistors, which are made of sand(currently))
- Something different though…what is it?

"latch" means it holds whatever value was already present…basically: "Previous Q"

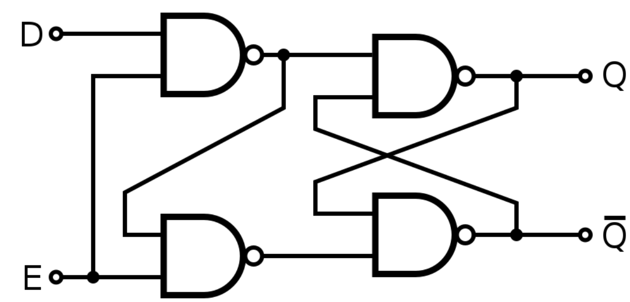


E	D	Q	$\overline{Q}$
0	0	latch	latch
0	1	latch	latch
1	0	0	1
1	1	1	0

E = "Enable"   D = "Data"   Q = not sure, but it is the output

https://www.allaboutcircuits.com/textbook/digital/chpt-10/d-latch/

The D Latch Provides Memory!



1. Set E=1
2. Set your D value
3. Set E=0
4. Whatever D was is stored at Q forever until E is 1 again!
5. **Can we do better/different?**

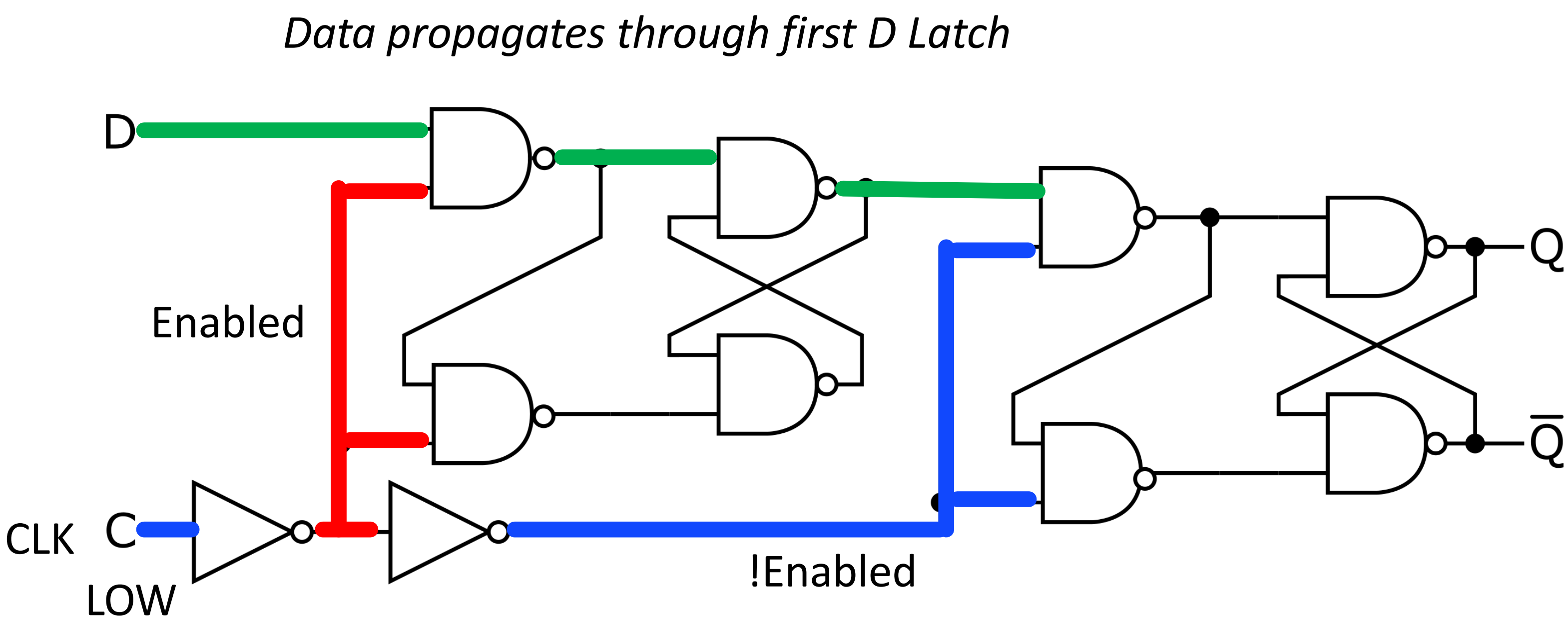
E	D	Q	$\overline{Q}$
0	0	latch	latch
0	1	latch	latch
1	0	0	1
1	1	1	0

E = “Enable” D = “Data”

Q = not sure, but it is the output

The D Flip-Flop (Reg)

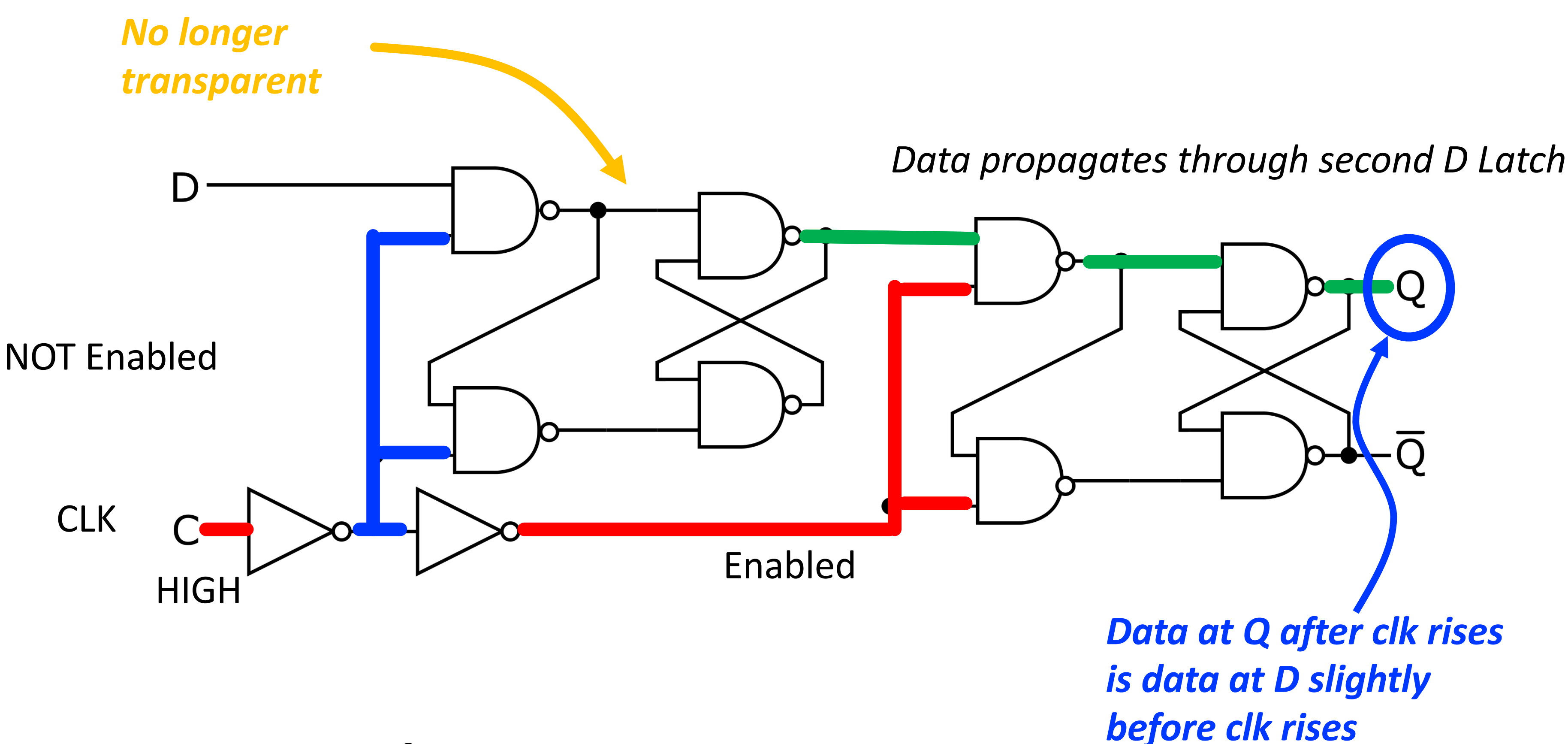
Two D-Latches in Series driven with opposite enable signals



<u>CLK LINE is LOW</u>

The D Flip-Flop (Reg)

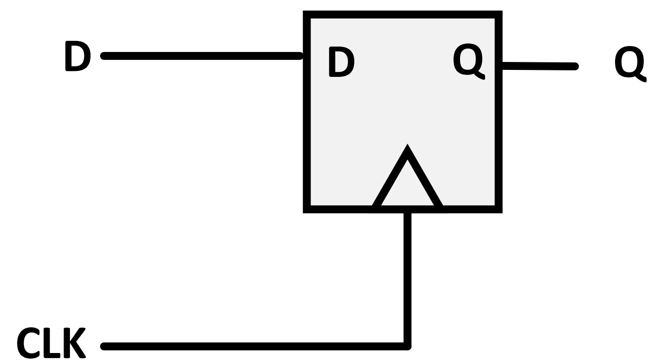
Two D-Latches in Series driven with opposite enable signals

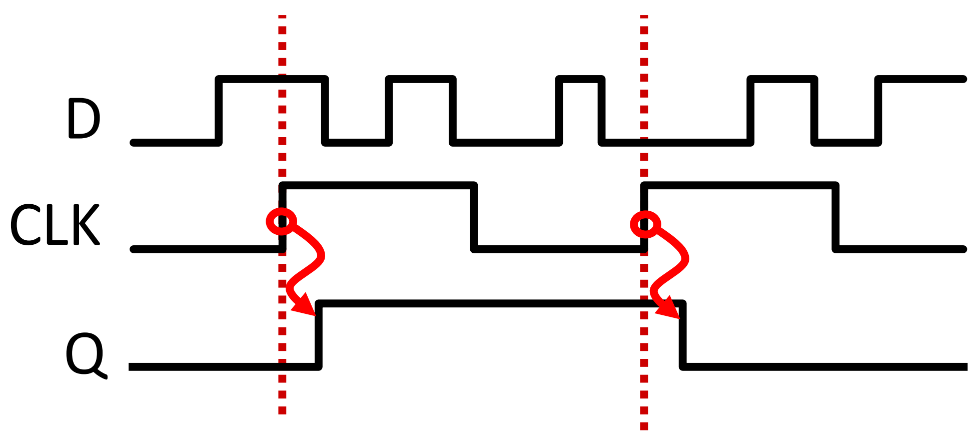


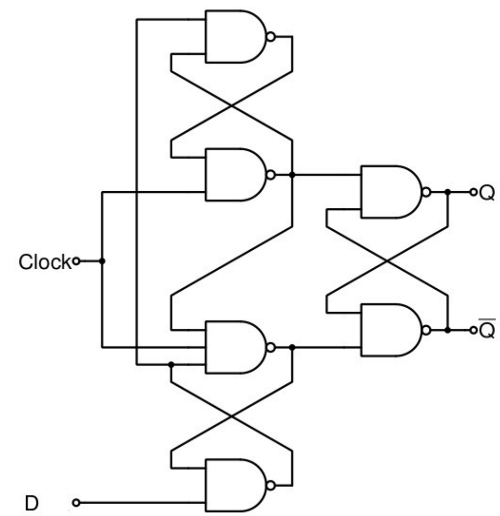
<u>CLK LINE rises to HIGH</u>

The Result: the D Flip-Flop

- The edge-triggered D register: *on the rising edge of CLK*, the value of D is saved in the register and then appears shortly afterward on Q.







Example: 74LS74 internals

When you simplify some common/redundant logic between the two stages, you get to about ~25 transistors

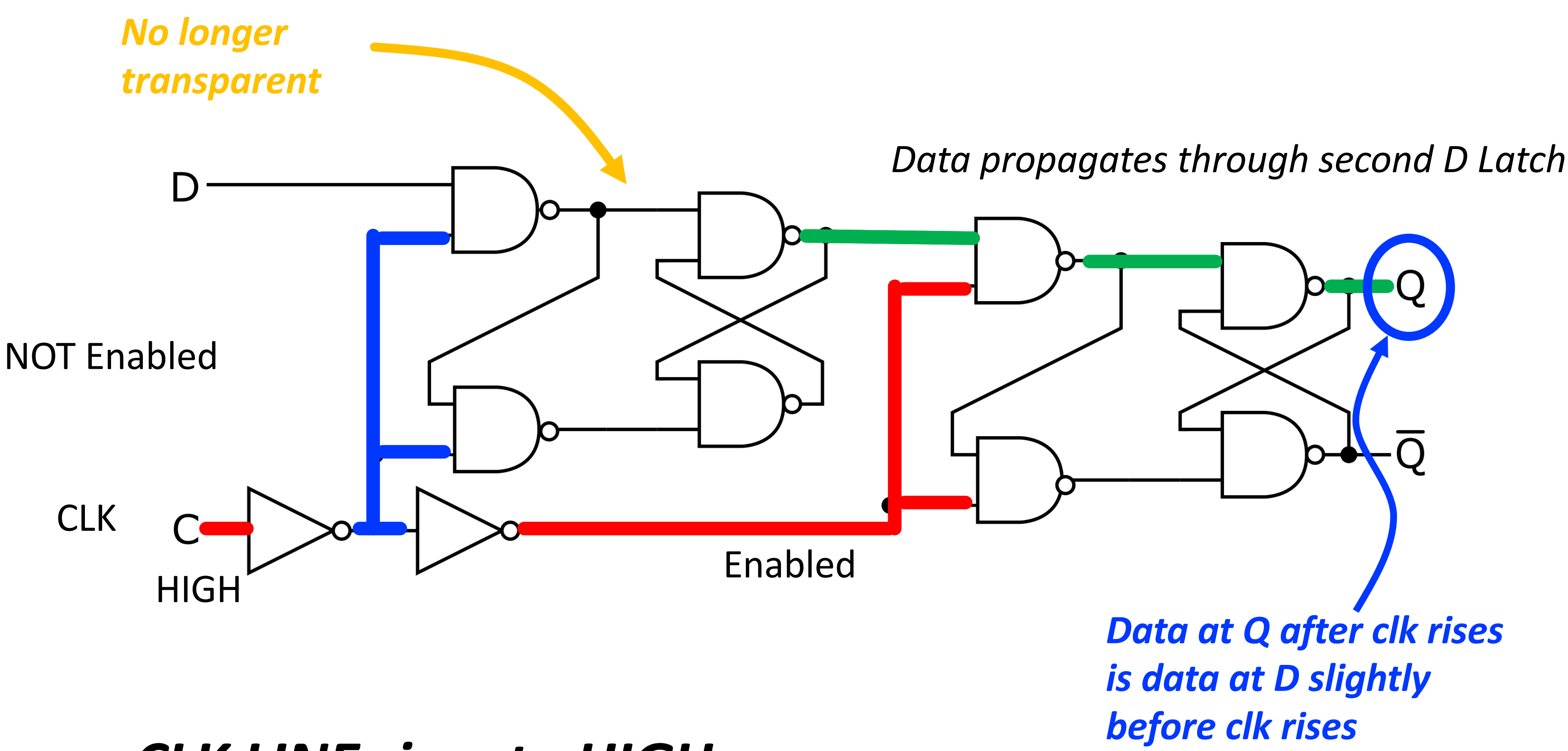
What about...?

- What about on the falling edge?

- What about it?

The D Flip-Flop (Reg)

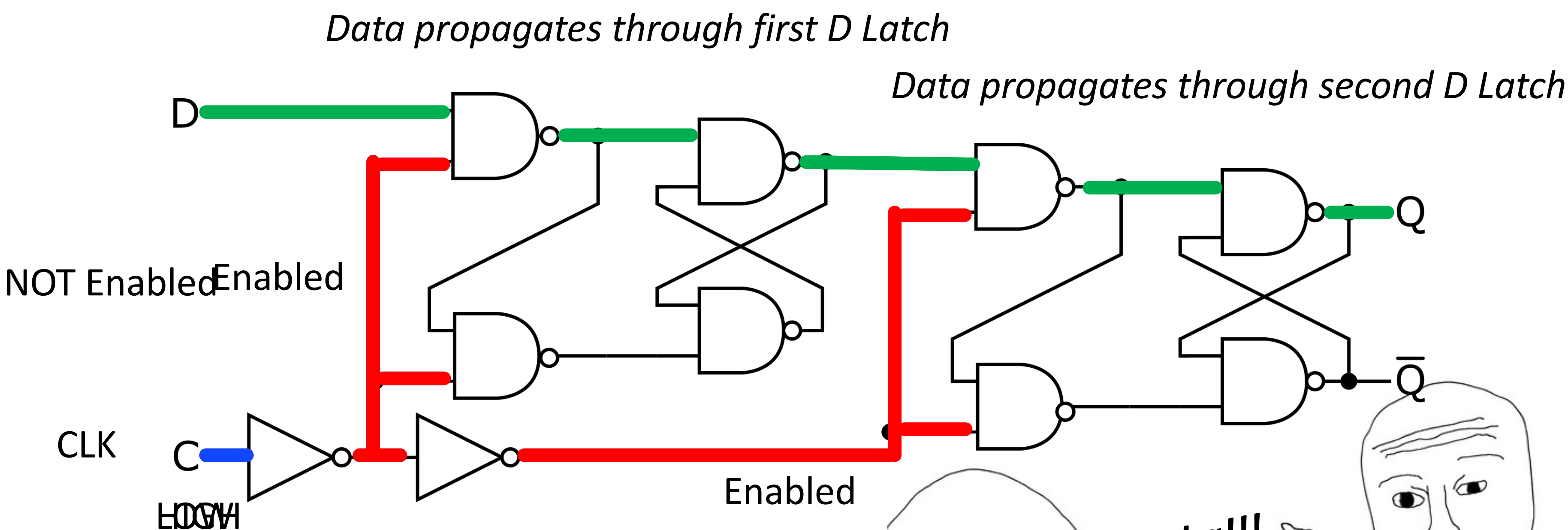
Two D-Latches in Series driven with opposite enable signals

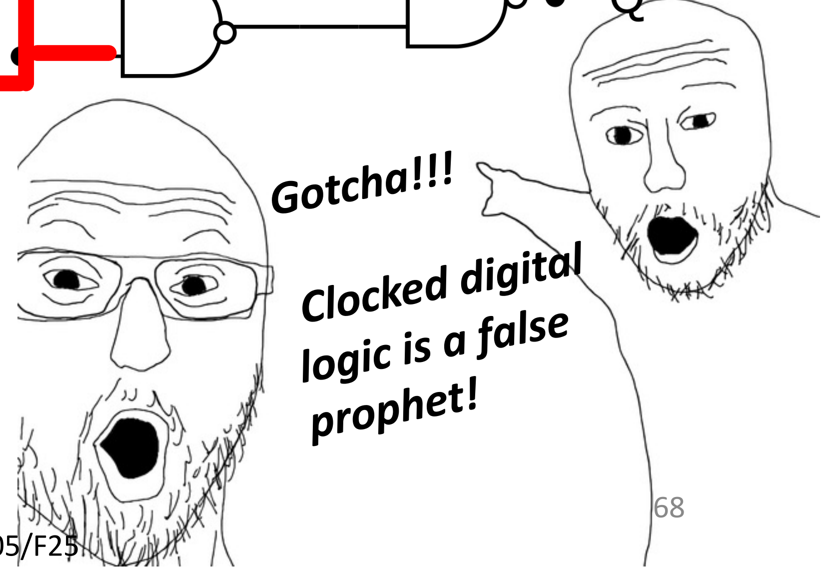


CLK LINE rises to HIGH

The D Flip-Flop (Reg)

Two D-Latches in Series driven with opposite enable signals





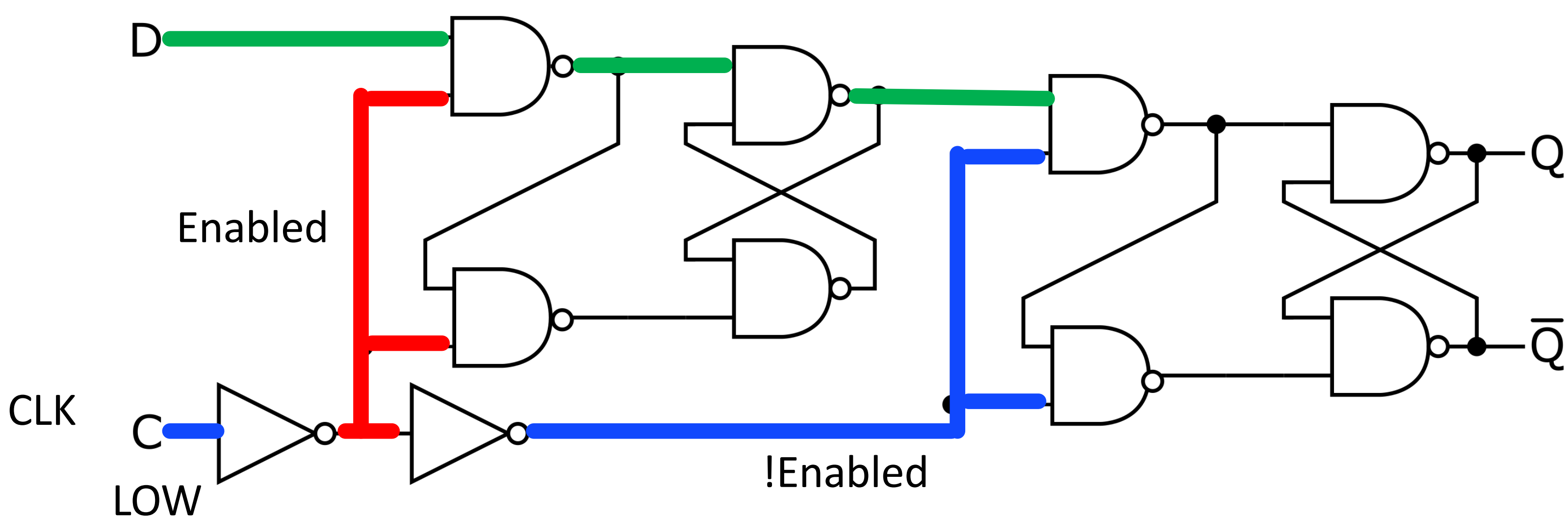
CLK LINE falls to LOW

In Reality

- The cartoon gates are not real. They are models
- Electricity doesn’t really flow like magical juice
- The gates don’t really act like valves to electricity, but moreso valves to information...and even then they’re not valves...they’re amplifiers
- Even if there is some momentary transparency, it doesn’t really matter since it would be for a very short time (much faster than any outside thing can take advantage of it).
- So you have:

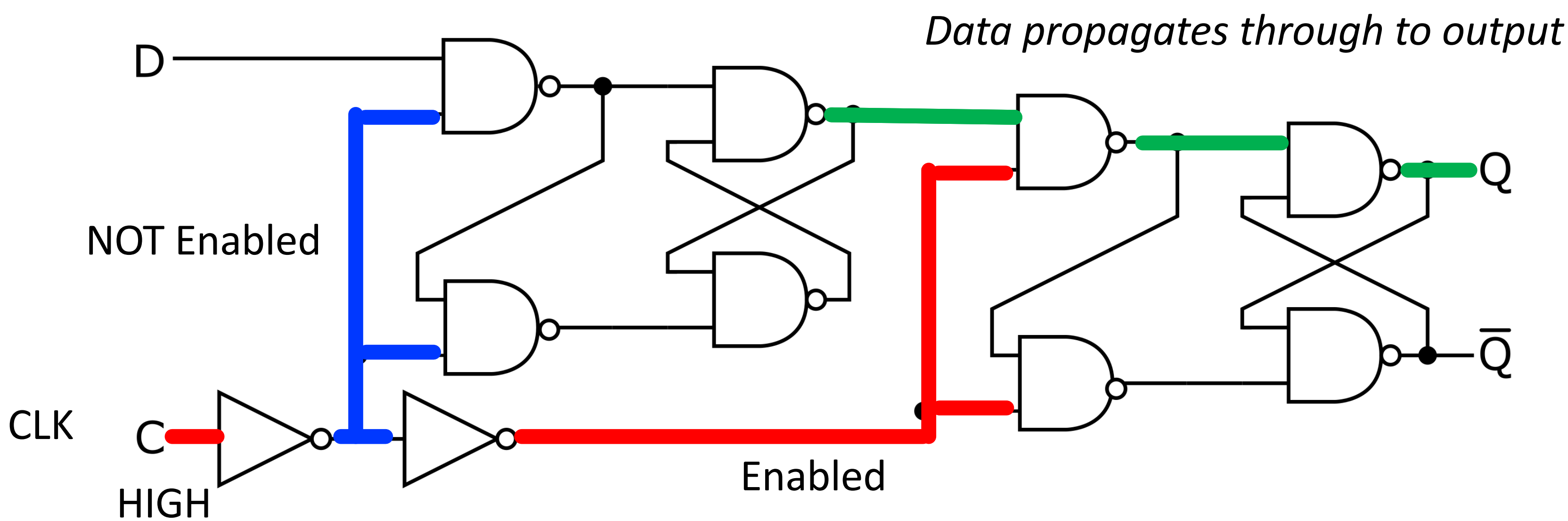
The D Flip-Flop (Reg)

Data propagates through first D Latch



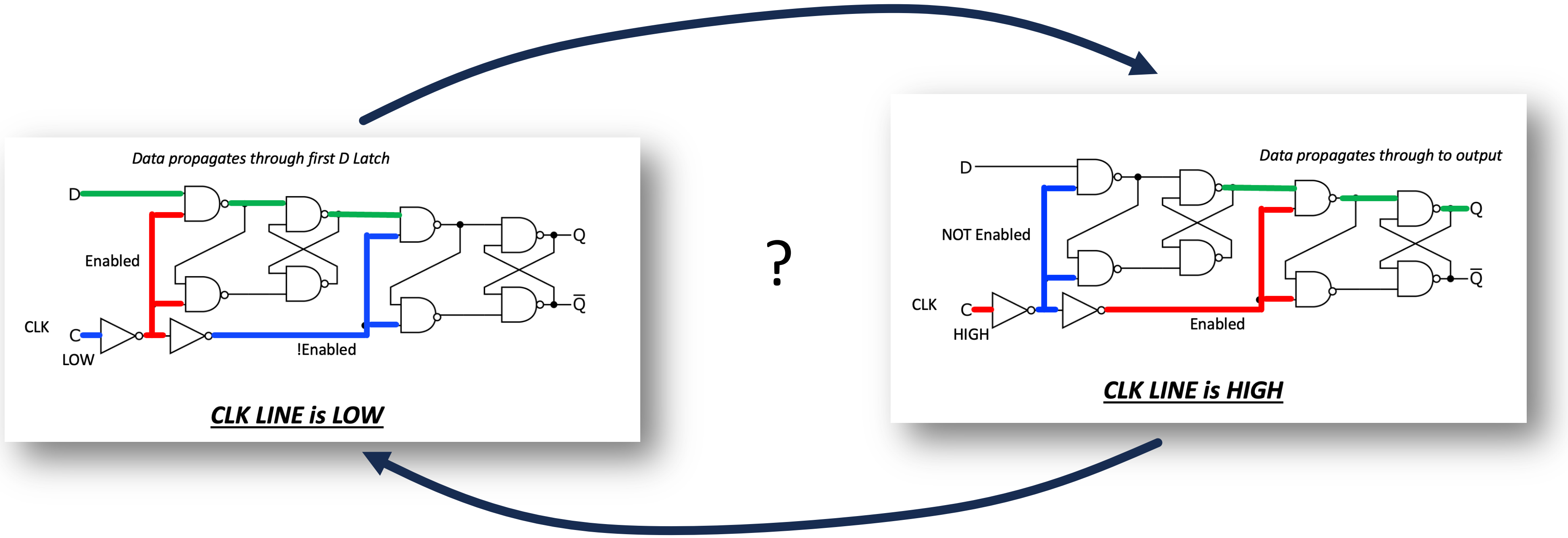
<u>CLK LINE is LOW</u>

The D Flip-Flop (Reg)



CLK LINE is HIGH

And it cycles back and forth



- The exact behavior during the transition is a little vague...
- And it does need to be considered

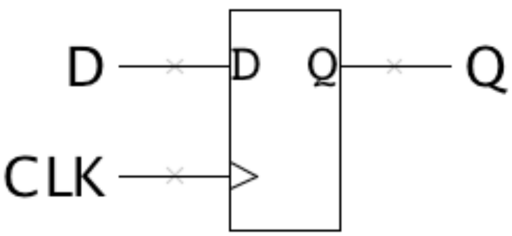
Things to Keep in Mind: Delay

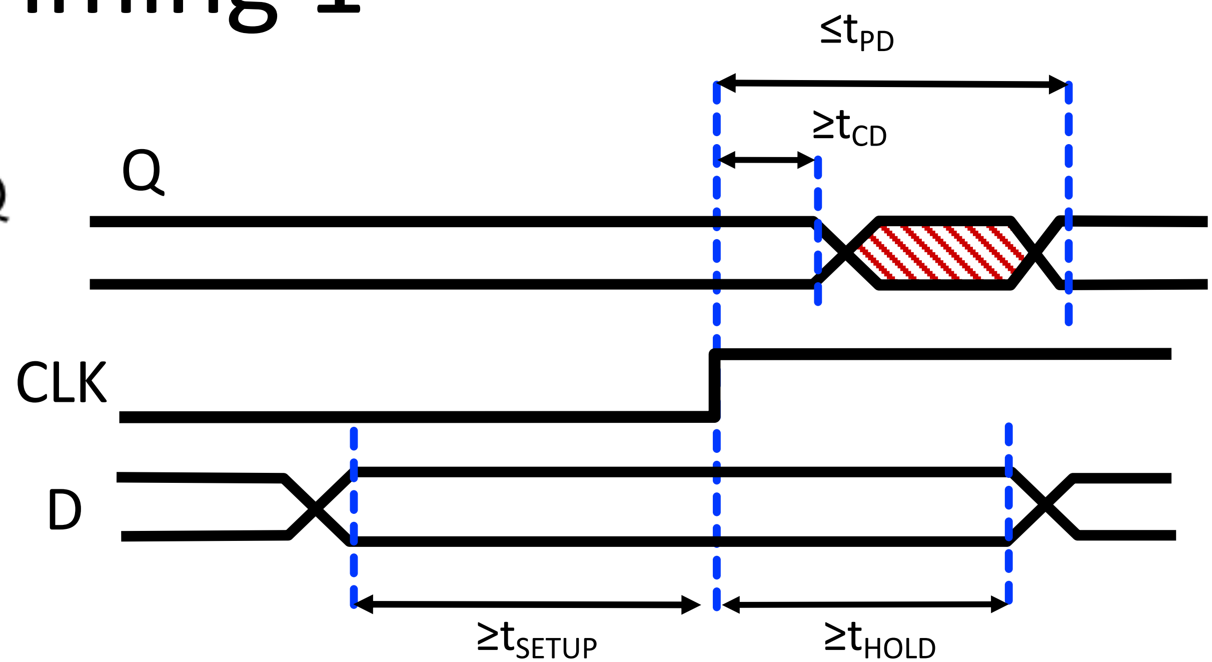
- Flip-flops are circuits at the end of the day...they have contamination and propagation delays

- Just like regular combinational circuits!!!

Replace "Stars" with "Flip-flops" and "Us" "Combinational Logic" for joke to operate

D-Register Timing 1



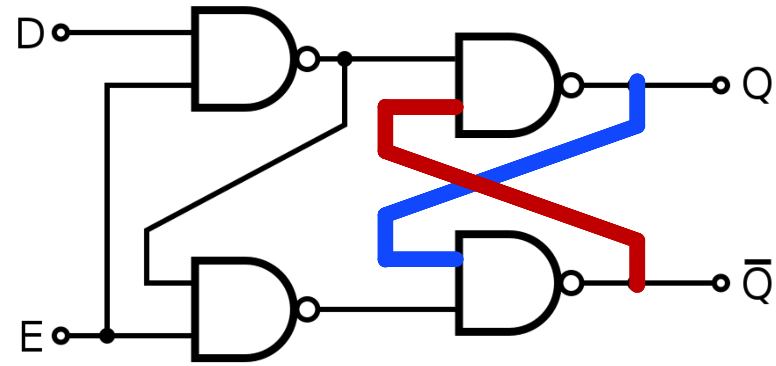


IMPORTANT:

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

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

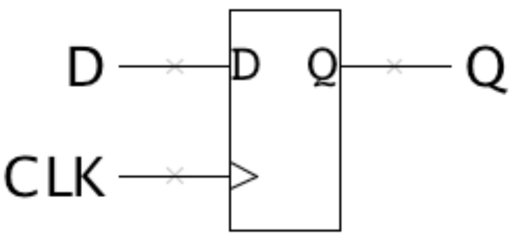
Vulnerability

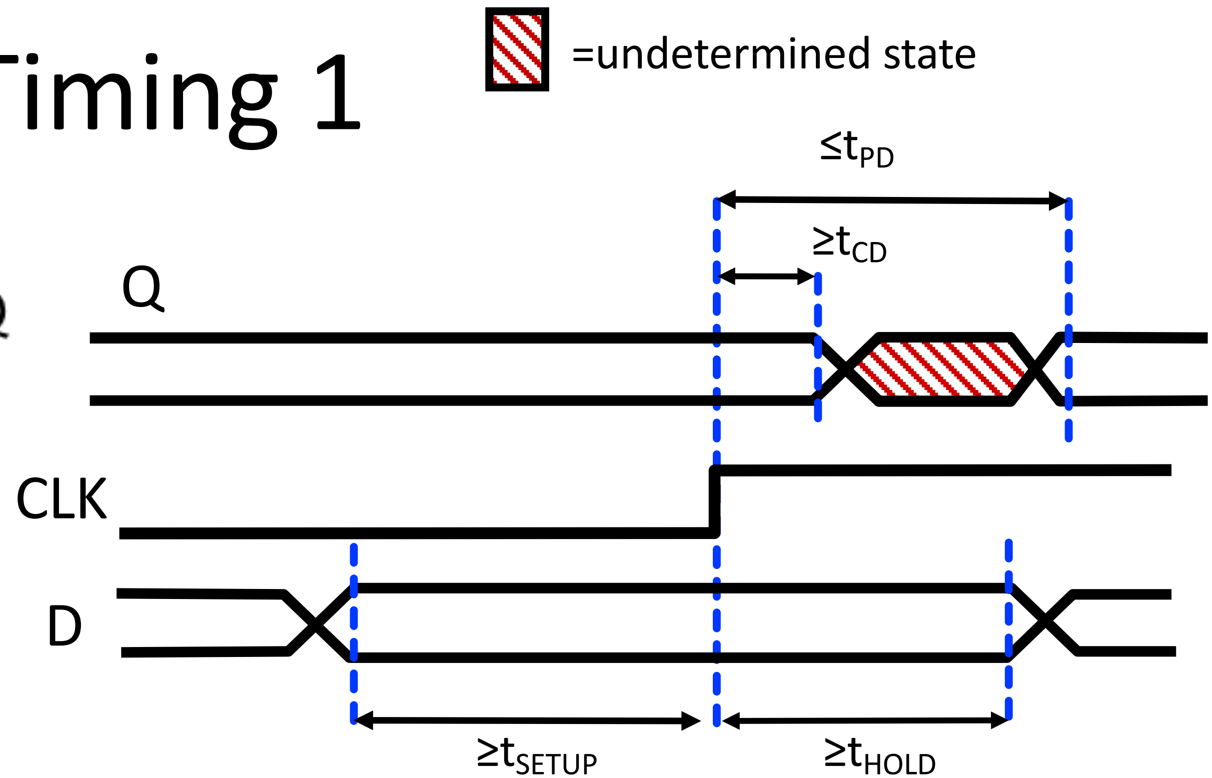


- During the transition period Flip-flops are vulnerable during so we need to help out by treating the device in certain ways!
- Digital circuits are high-gain amplifiers
- There is also feedback present
- Messing with the flip-flop's inputs during a transition period can be catastrophic*
- So we must give the flops their space when they are most vulnerable!!!

**metastability...talked about in future class.*

D-Register Timing 1





IMPORTANT:

t_{PDr}: maximum propagation delay, @posedge CLK D →Q

__Maximum__ time it takes for Q to change after rising edge of CLK

t_{CDr}: minimum contamination delay, @posedge CLK D →Q

__Minimum__ time it takes for Q to start to change after rising edge of CLK

t_{SETUP}: setup time

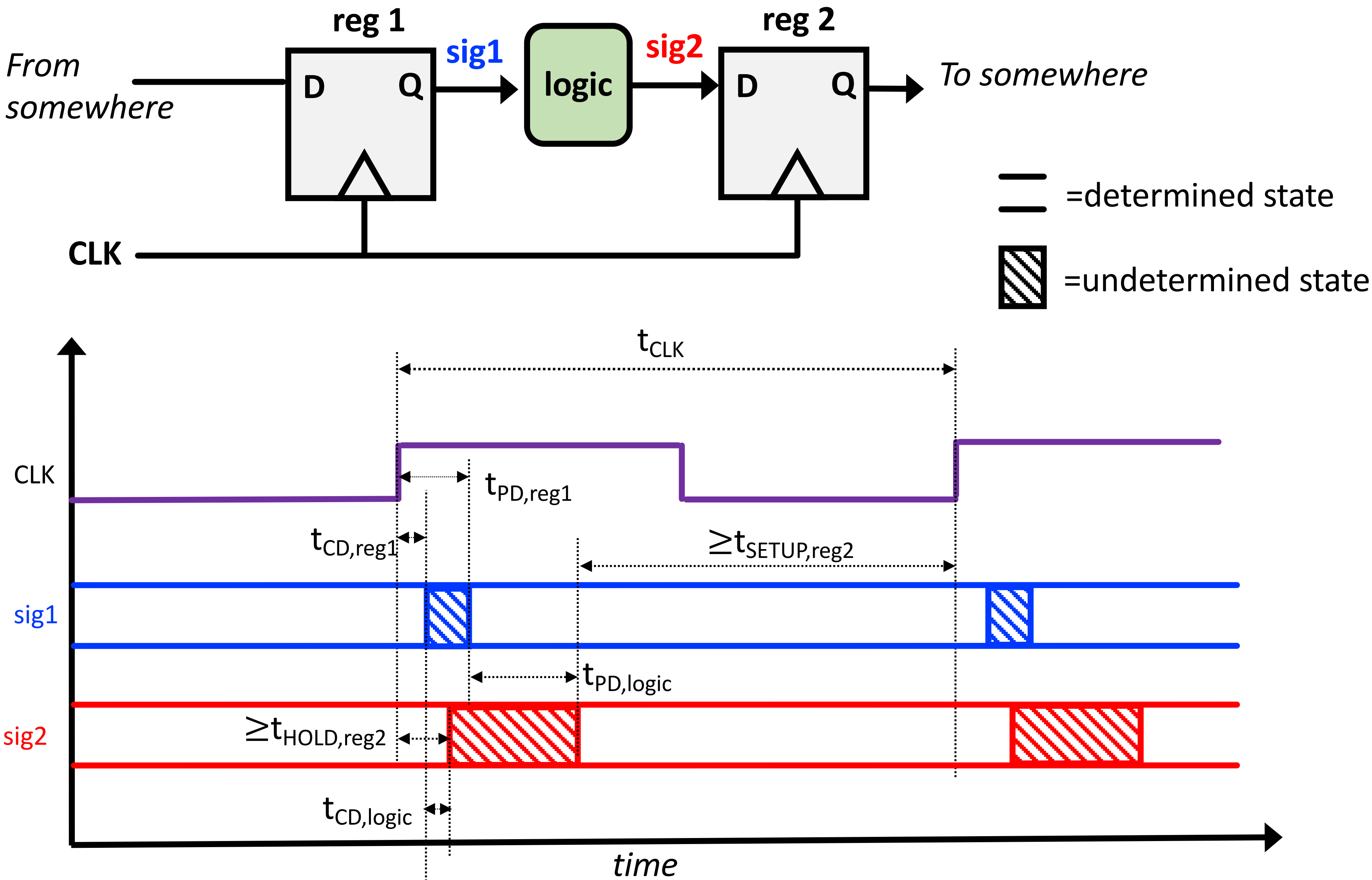
How long D (input) must be stable __before__ the rising edge of CLK

t_{HOLD}: hold time

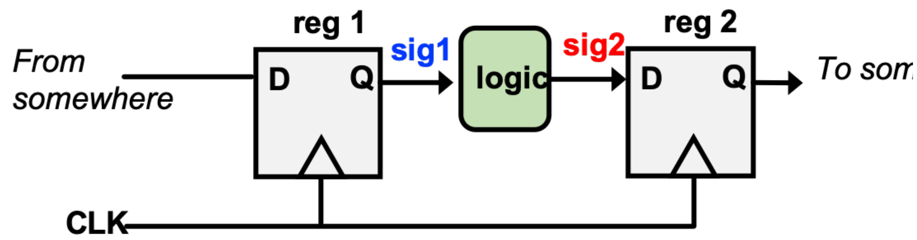
How long D (input) must be stable __after__ the rising edge of CLK

New timing attributes for registers

Register-to-Register Timing

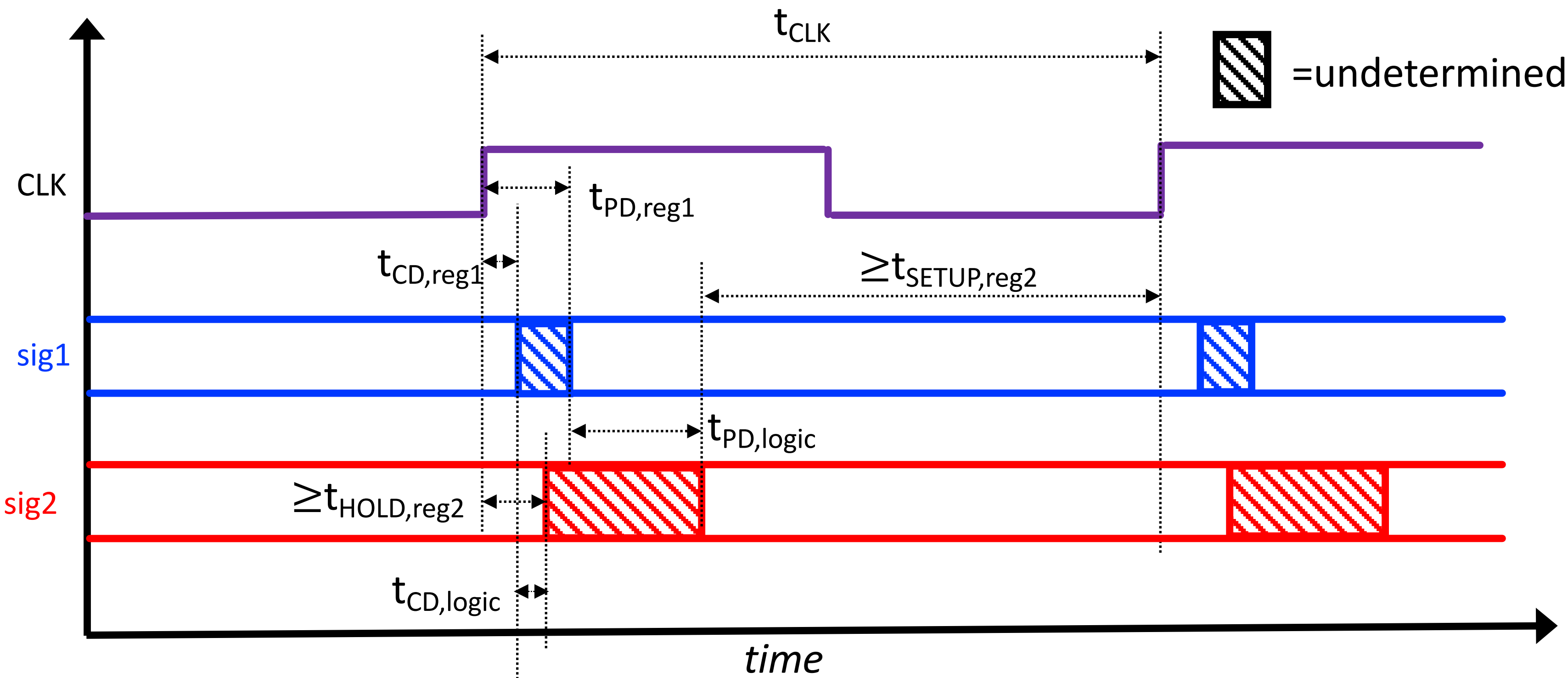


Register-to-Register Timing



=determined state

=undetermined state



Two Requirements/ Conclusions:

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

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

D Register Timing Conclusions

$t_{PD,reg}$, $t_{SETUP,reg}$, $t_{CD,reg}$, $t_{HOLD,reg}$, and $t^*_{CD,logic}$ are all roughly fixed/ unchangeable

*We may/will encounter this in 6.205! If we try to make our combinational logic **tooooo complex** and we won't satisfying timing. How do we fix? Two options:*

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

<u>6.205 Design Space:</u>

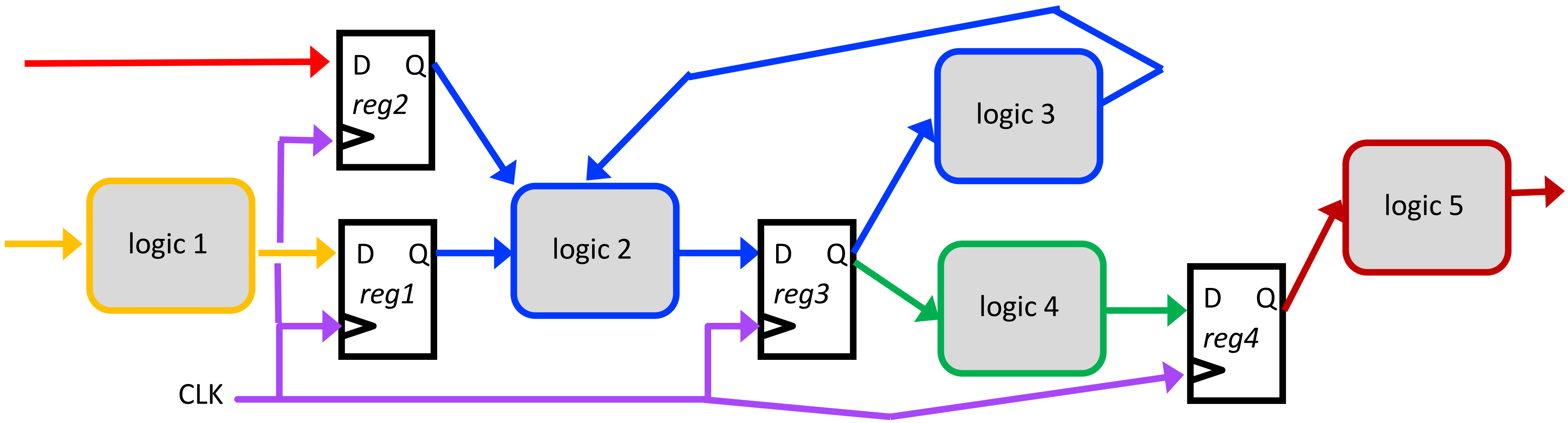
Slow down clock: $\uparrow t_{CLK}$

Shorten combinational logic: $\downarrow t_{PD,logic}$

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

*If you violate this, you have to change your design. This is more an issue for the device engineers…on our FPGAs the contamination delays (min change times) are usually longer than HOLD times, so it is hard for **<u>us</u>** to run into this problem in 6.205 (though it is a very real problem for people laying out circuits)*

Design Complex Logic In Stages!



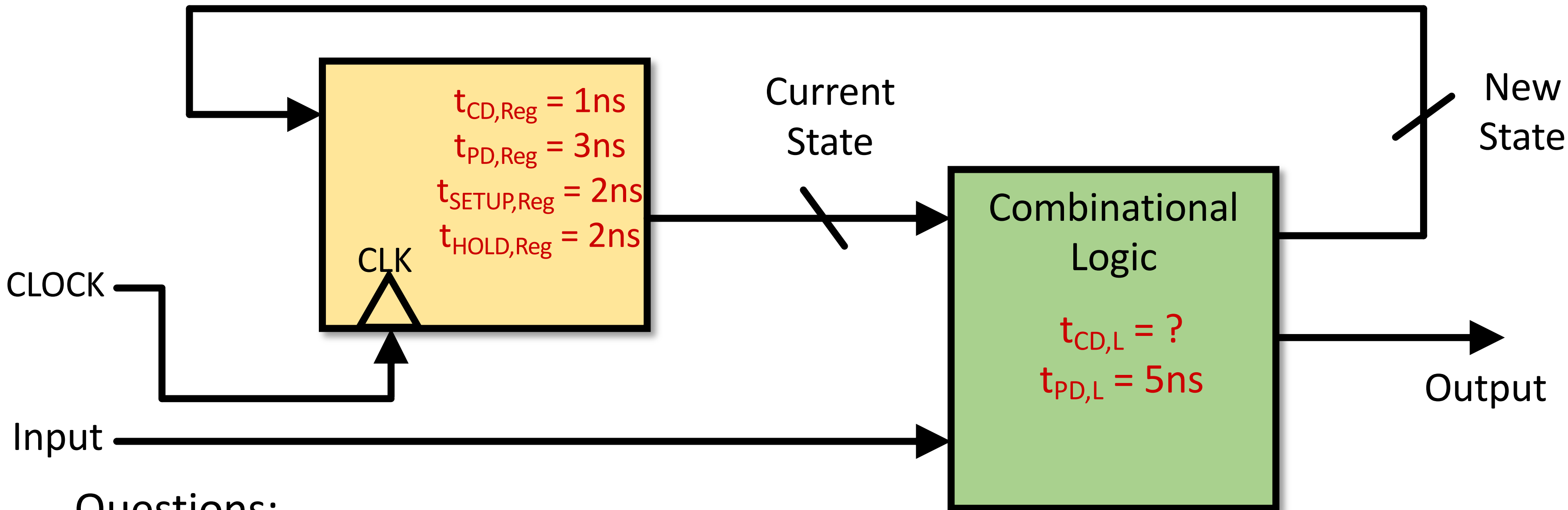
- Design complex logic systems in stages
- Worry only about effects of delays (t_{pd} and t_{clk}) within a given stage, rather than how they all interplay!

Single Clock Synchronous Discipline

- The timing requirements are already complicated enough with one clock. Avoid multiple clocks at all cost! <u>DO NOT clock flip flops on non-clock lines.</u>
- 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 combinational delay
- Change saved state after noise-inducing logic changes have stopped!

Sequential Circuit Timing

Assume input is also coming from a clocked system



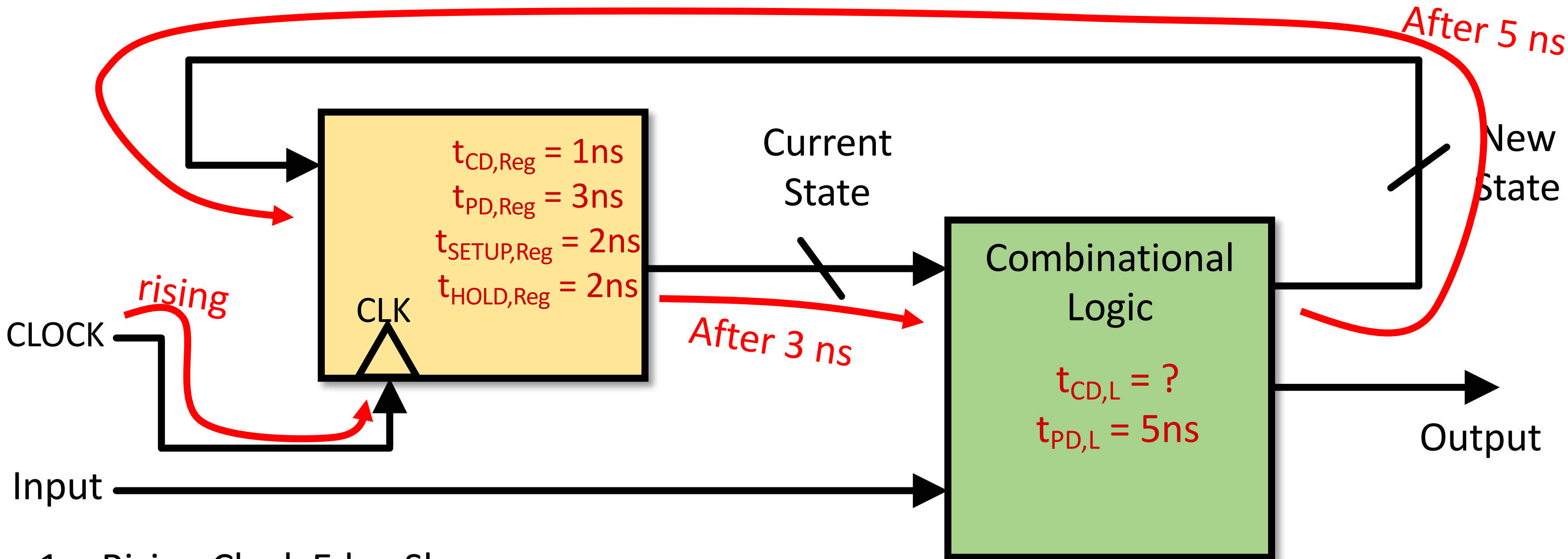
Questions:

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

This is a simple *Finite State Machine (next lecture)*

Minimum Clock Period?

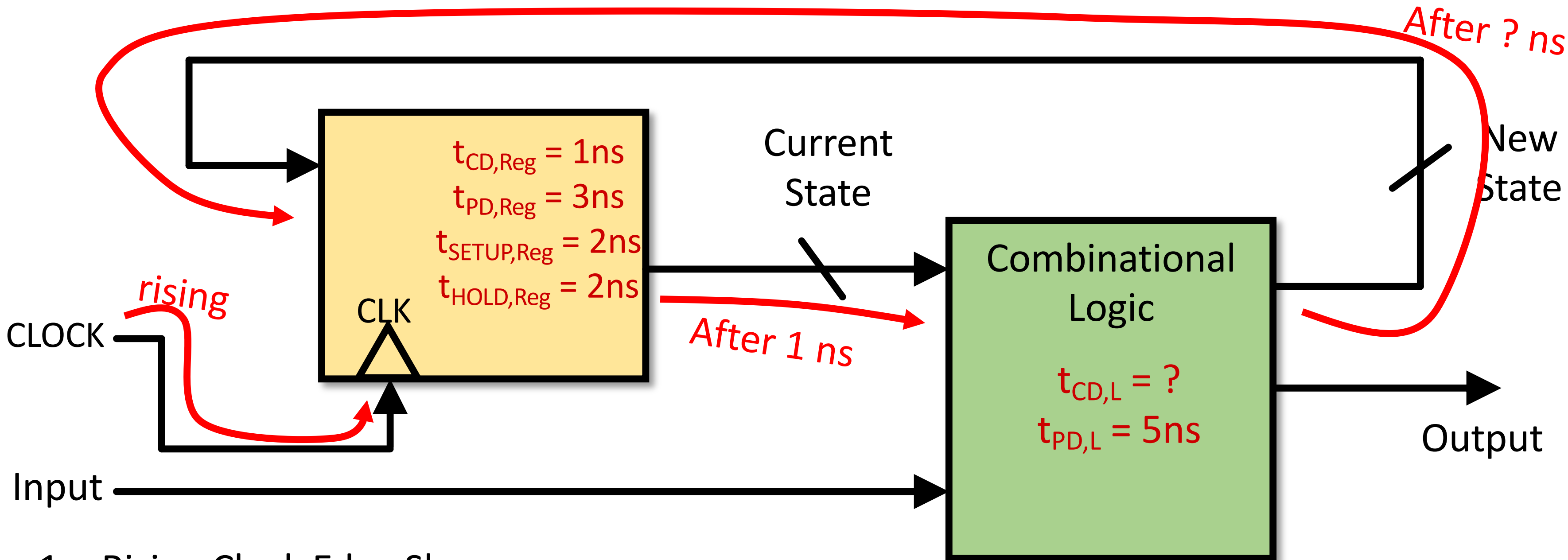
Assume input is also coming from a clocked system



1. Rising Clock Edge Shows up…
2. It is $t_{PD,Reg}$ = 3ns until flop has finalized changing
3. It is then 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

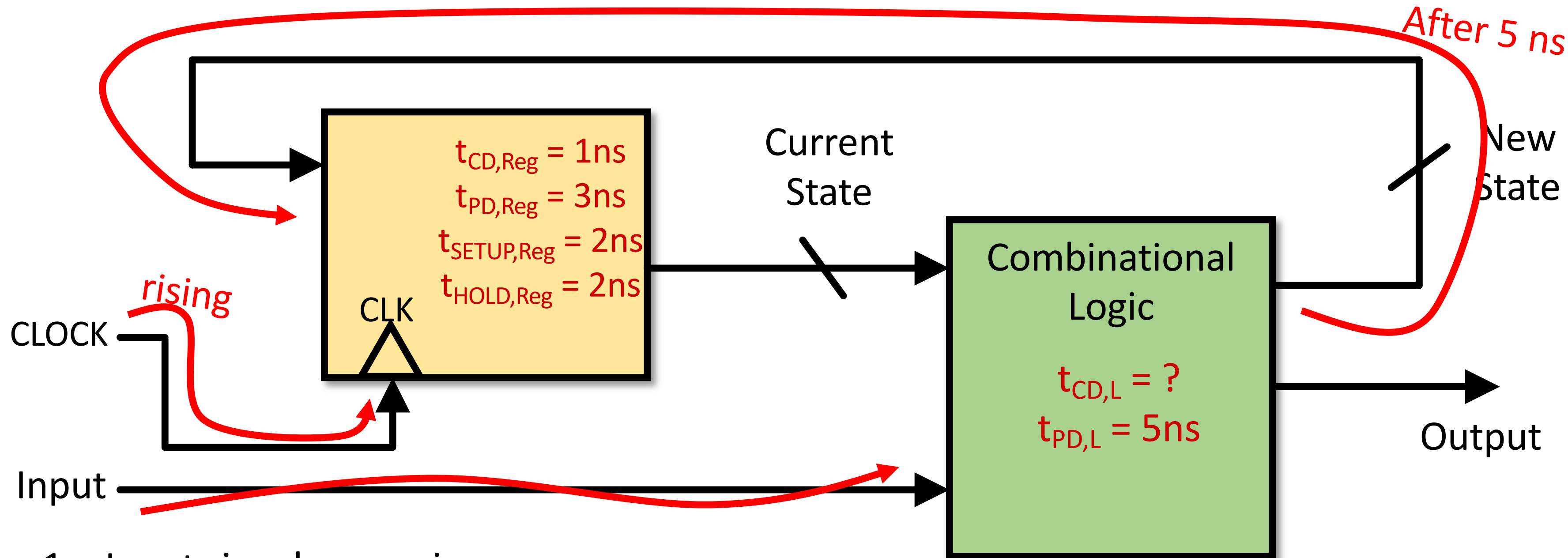
$t_{CD,L}$ Constraints?

Assume input is also coming from a clocked system



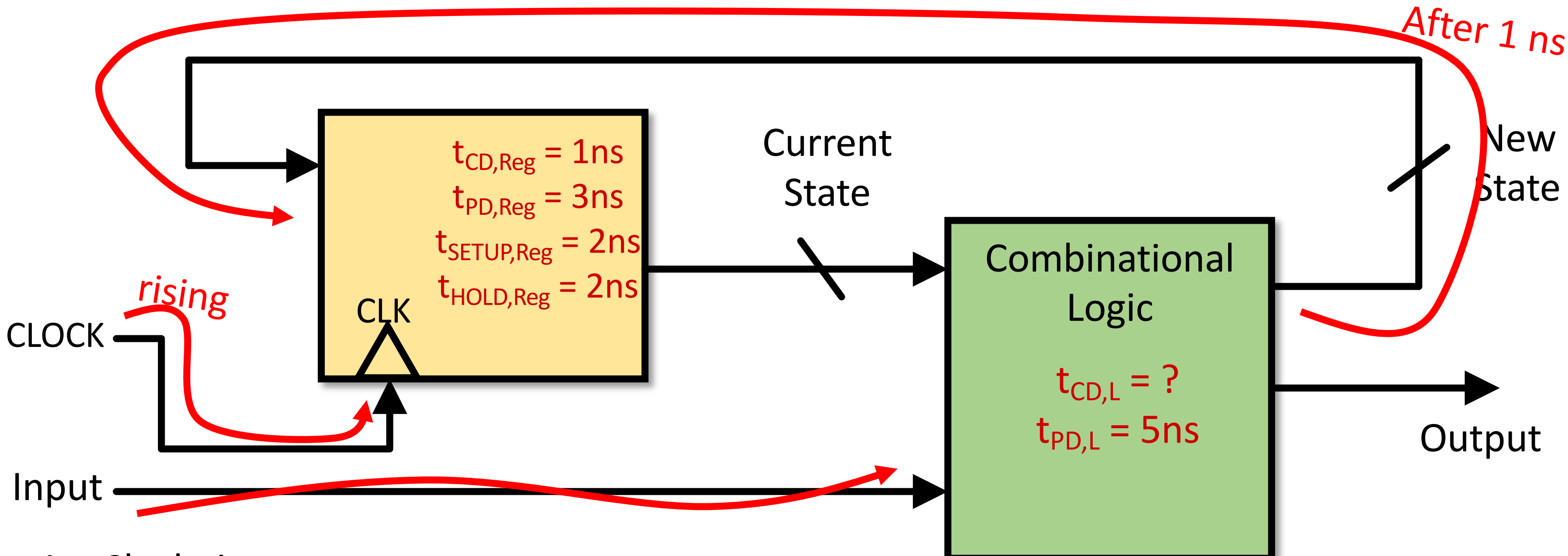
1. Rising Clock Edge Shows up…
2. It is $t_{CD,Reg}$ = 1ns until flop output starts changing
3. It is then additional $t_{CD,L}$ = ? Until feedback wire starts changing
4. That change cannot be happening until at least $t_{HOLD,Reg}$ = 2ns has passed from clock edge

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 done at least $t_{SETUP,Reg}$ = 2ns before the next rising clock edge

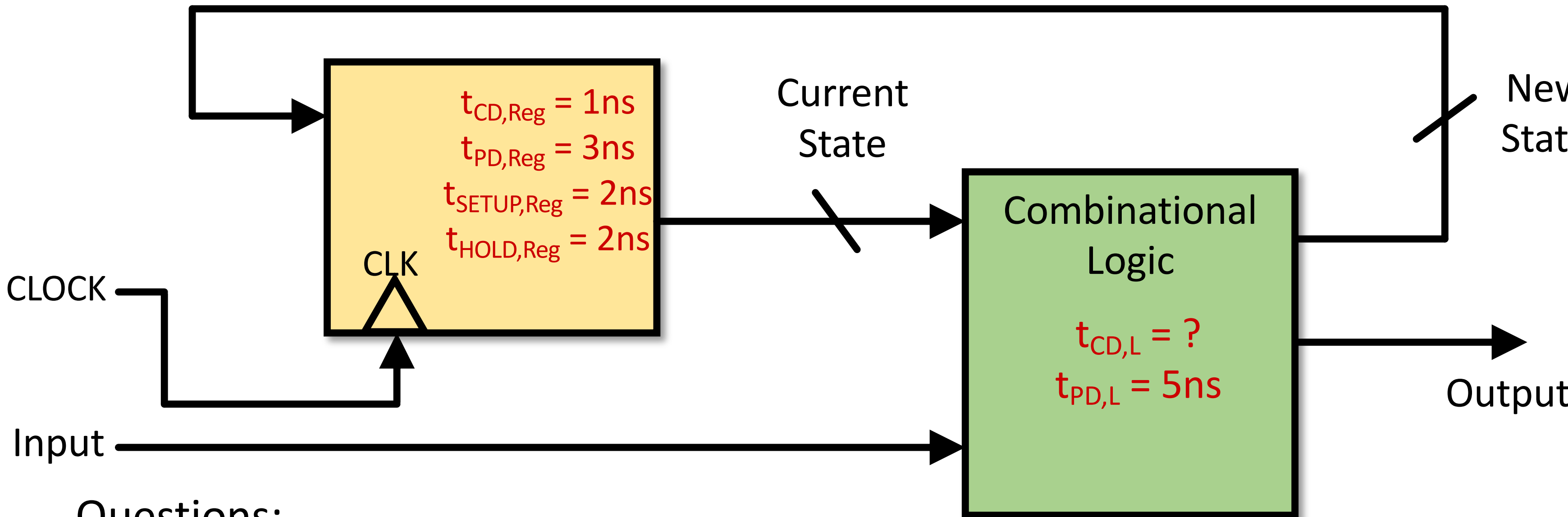
Hold Time for System Input?



1. Clock rises
2. Input signal comes in
3. It is additional $t_{CD,L}$ = 1ns until comb logic starts to feed back…
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?

This is a simple Finite State Machine …
which we will cover formally on Thursday