

6.205
(aka 6.111)

Sequential Logic I

Fall 2026

Administrative

- Week 01 Due Last Night
- Week 02 Out After Class...due next Wednesday...
- Week 02's lab take more time than Week 01. I would recommend you start on its pieces early.

THIS WEEKEND

COMPUTERS - HI TECH - HAM RADIO

GIANT SWAPFEST www. .us

AT
M
I
T

NOW the THIRD SUNDAY of EACH MONTH ALL SUMMER
(April to October)

BARGAIN ELECTRONICS - COMPUTERS - RADIO PARTS
MECHANICAL TOOLS - ENGINEERING TEST EQUIPMENT - RARE COMPONENTS

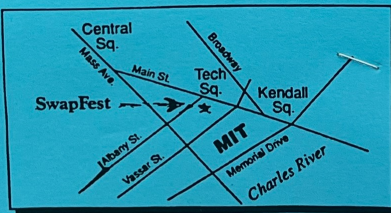
Back after the pandemic!
Now in our 38th year.

Sunday September 20th

- Buy Sell Swap - 9 AM to 2 PM

Albany and Main Streets, Cambridge
TAILGATE ROOM FOR OVER 200 SELLERS
Rain or Shine — Garage Available

BUYERS \$6
FREE for MIT & Harvard STUDENTS with ID
SELLERS \$20
per space includes 1 admission
Bring a table or sell right out of your trunk!
Additional vendor spaces are \$15.



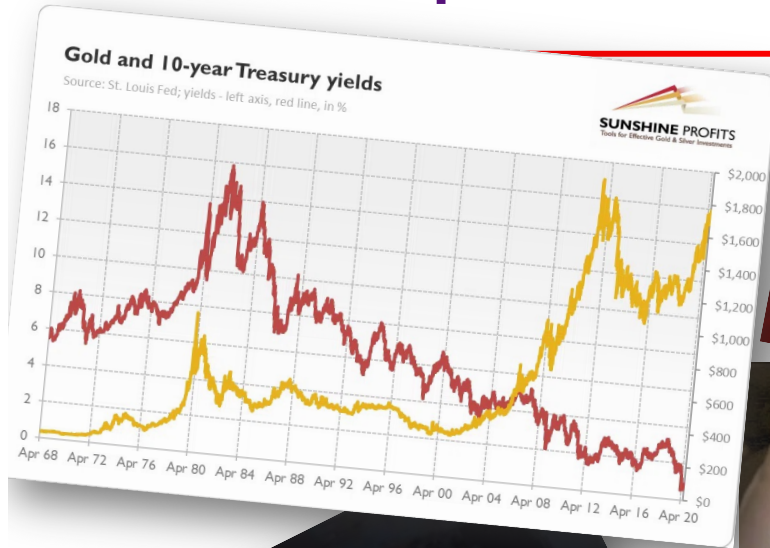
FREE!
Buyers Parking
VENDORS
from all of
New England

*APVD
HWC*

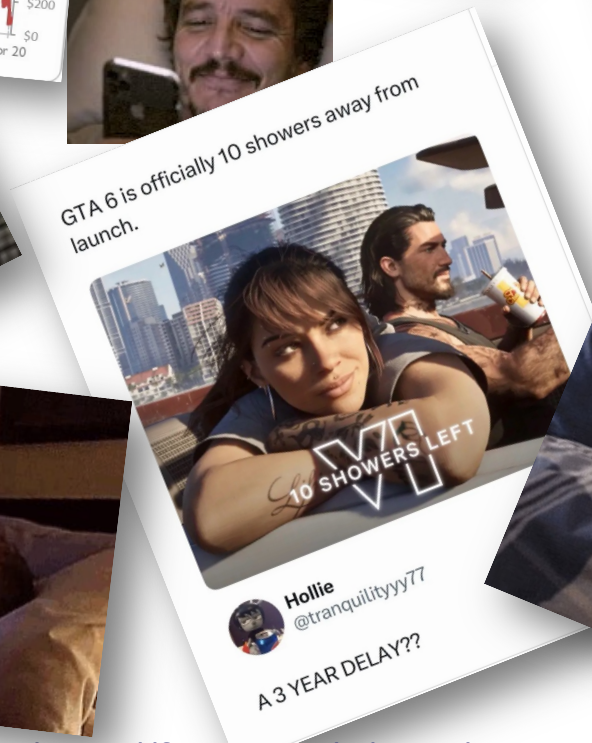
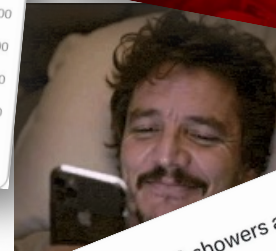
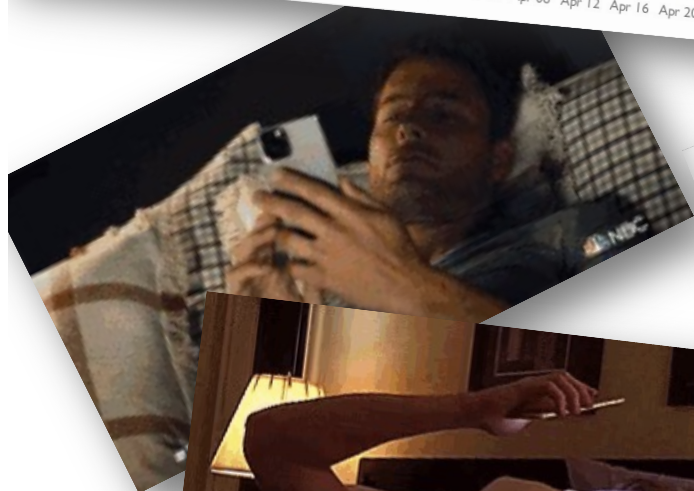
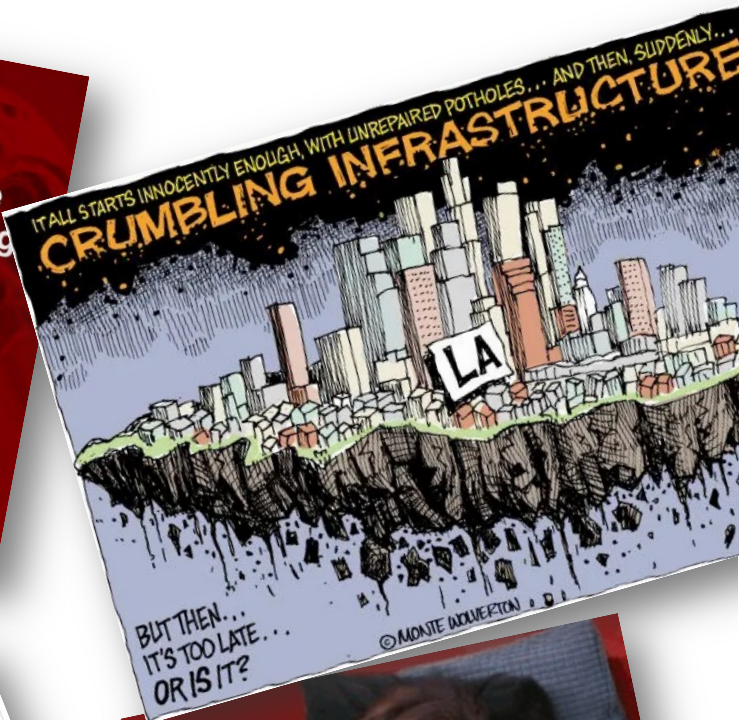
Come early for BEST BARGAINS

Sponsored By: MIT Radio Society W1MX, MIT Electronics Research Society, and MIT UHF Repeater Association W1XM
In Memory of Steve W1GSL

Hellscape



I saw the headline
"Is AI Actually Going
to Kill Us All?"
But the article was
behind a paywall,
so I'm just going to
take my chances.



Touch Some Grass...Talk to
People...Get some Sweet Deals

COMPUTERS - HI TECH - HAM RADIO
GIANT SWAPFEST AT
www. .us

Sunday September 20th

Back at the place
the pandemic
Now in our
38th year.
Sunday September 20th
- Buy Sell Swap - 9 AM to 2 PM

Albany and Main Streets, Cambridge

TAILGATE ROOM FOR OVER 200 SELLERS

Rain or Shine - Garage Available

**FREE for MIT & Harvard
STUDENTS with ID**

Look at this sweet Memory Board I got this summer at SWAPFEST

- OMG

Administrative

- Week 01 Due Last Night
- Week 02 Out After Class...due next Wednesday...
- Week 02's lab take more time than Week 01. I would recommend you start on its pieces early.

Week 2 Hint

- When it comes to the SPI module you need to write, don't approach it as a software engineer with only arrays and indexing/index variables and iterators. Yuck!
- Try to think about how shifting bits ($\gg$ or $\ll$) can get you what you want instead!

Rule about For Loops

- Inside an `always_comb` (or `always_ff?`):
 - Use *regular for loop*
- Want to make multiple assign statements? Or Multiple `always`-type blocks? or multiple modules? Want to *generate* higher-level structures?:
 - Use a *generate loop*!
- In both instances, the iterating variables of the loop have no intrinsic hardware meaning...they exist as a helper variable during specification (a copy-paster thingie)

Regular for loop

- If you are in an always block and just need to replace a bunch of repetitive lines, a for loop can help
- Let's say I had to do some annoying operation a bunch of times with some variables:

```
logic [7:0] b [63:0];  
logic [7:0] c [63:0];  
logic [63:0] a;  
  
//assume b and c are large enough  
always_comb begin  
    for(integer i =0; i<64; i= i+1)begin  
        a[i] = b[i]>c[63-i];  
    end  
end
```

Generate For Loops

An Example:

```
generate
  genvar i;
  for(i=0; i<5; i=i+1)begin: myloop // label myloop
    logic[31:0] hi;
    assign hi = 32'hAAAAAAAA ^ i;
  end
endgenerate
//outside of generate, those logics can be accessed with:
// myloop[2].hi for example
// this is needed since the logic hi needs more
// specificity than provided otherwise.
```

wire vs. logic. vs. reg

- **wire** Can only be signal flow ("nets"). From perspective of a module, signals coming into module are conveyed by wires. In other usage, declared wires can only be given values with **assign** statement. A wire can also be associated with combinational logic.
- **reg** Ideally represents a flipflop or latch (storage mechanism), but in reality can also turn into a net (in other words a wire)/ combinational logic based on usage (cover more on Thursday in Lec 03). Only given values with **always-family** blocks. **DO NOT USE IN 6.205**
- **logic** Can represent all datatypes. Its usage dictates what it ultimately represents (combinational logic or Flip Flops). Can be worked with **assign** and **always-family** blocks

Why logic?

- In addition to allowing us to just use one general type rather than two, the **logic** datatype has stricter protections against multi-driven nets

```
module thing(input wire [3:0] a,b,
output wire [3:0] c);
    //some behavior goes here
endmodule

module main_module();
    logic[3:0] a,b,c;
    thing my_thing(.a(a), .b(b), .c(c));
    assign c = 4'b1010; //whoops might miss checks in Vivado (multi-driven net)
endmodule
```

- Logic on output should prevent:

```
module thing(input wire [3:0] a,b,
output logic [3:0] c);
    //some behavior goes here
endmodule

module main_module();
    logic[3:0] a,b,c;
    thing my_thing(.a_in(a), .b_in(b), .c_in(c));
    assign c = 4'b1010; //should get caught on synthesis
endmodule
```

So why still use `wire` in module definitions at all?

```
module thing(input wire [3:0] a_in,  
             input wire [3:0] b_in,  
             output wire [3:0] c_out);  
    //stuff  
endmodule
```

- Seems excessive...Let's just use logic for everything.
- We would...but...
- This is a thing we do in 6.205 to help us with our Vivado toolchain since it is a picky, picky child.

If we had a module definition like this:

```
module thing(input logic [3:0] a,  
             input logic [3:0] b,  
             output logic [3:0] c);  
    assign c = a ^ b;  
endmodule
```

- This would run just fine in all honesty.
- In the scope of this module, what logic “is” isn’t specified, but by default Verilog interpreters tend to assign the “wire” attribute to unspecified things (like these logic’s would be).

The Problem...

```
module thing(input logic [3:0] a,
             input logic [3:0] b,
             output logic [3:0] c);

    assign d = 4; //new line
    assign c = a ^ b ^ d;
endmodule
```

- This verilog will synthesize perfectly happy for the same reason the verilog on the previous page did. Why???
- `d` is undeclared, but Verilog just assumes it is a “wire” by default...a one-bit wire. omg why.

To Protect Ourselves...

- we will (and should) often add the a directive at the top of our file to protect against this:

```
`default_nettype none
```

- This forces Verilog to treat everything not explicitly declared as a none entity and working with those **will** throw an error

The downside of doing that though...

- All inputs and outputs are of type `logic`
- A `logic` is an abstract type whose physical realization is determined through usage.

```
`default_nettype none
module thing(input logic [3:0] a,
             input logic [3:0] b,
             output logic [3:0] c);
    assign c = a ^ b;
endmodule
```

- In the scope of this module, how these inputs should get their values is never specified so their actual manifestation is left undefined and defaults to a none and an error gets thrown (annoying!)

So must make the inputs into wires

- All inputs are now wires

```
`default_nettype none
module thing(input wire [3:0] a,
             input wire [3:0] b,
             output logic [3:0] c);
    assign c = a ^ b;
endmodule
```

- In the scope of this module, inputs a and b are explicitly known to be wires (things that convey signals) so there's no ambiguity
- Still have that great protection against accidental variables since they'll be of type none!

But That's Not Enough

- It just keeps going...
- A lot of Vivado's legacy source files and modules *rely* on the default nettype being wire (I know who would do that?).
- So after our module we need to set things back to defaulting to wire
- So we'll tack this on at the *end of files*:

```
`default_nettype wire
```

To Satisfy All of These Issues

- Declare inputs to a module as wires since it will fully specify what they are
- Begin and end all modules with nettype compiler directives. This will protect you from implicit declarations by Vivado

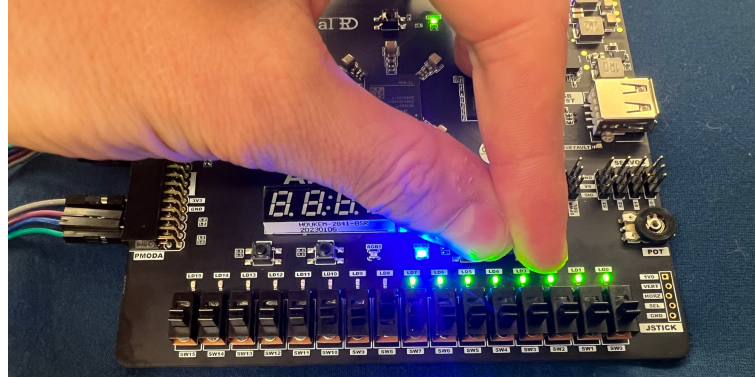
```
`default_nettype none
module thing(input wire [3:0] a,
             input wire [3:0] b,
             output logic [3:0] c);
    assign c = a ^ b;
endmodule
`default_nettype wire
```

Values in Verilog

- There are four different values in SystemVerilog:
 - 0: **logical 0**: self-explanatory
 - 1: **logical 1**: self-explanatory
 - **X: undefined**: meaning can't be determined by simulation
 - **Z: high-impedance**: meaning it is driven by something else (you'll see this with things that are inouts in shared wire bus architectures). To be honest not very widely used.

Simulation vs. Reality

- If you were to try to build a hardware divider in Verilog it can work.
- If you simulated the module with 0/0, you should have seen xxxxxxxxxxxx as a result.
- If you tried that on your board then, you'd see something like:



"Huh. 0 divided by 0 is 255. That checks out. I'll use that as a firm foundation upon which to build my high-speed trading design "

Simulation vs. Reality

- You need to be very careful about looking out for undefined (X) values in simulation. **They should not be ignored.**
- In fact, simulations are great because they let us see those clearly.
- Digital circuits are extremely robust and are NOT HAPPY in some in-between state. They **will** resolve to having a 1 or a 0 at their output.
- This means that in real-life undefined values only show up as 1's or 0's...so they can be hidden amongst a sea of legit 1's and 0's.

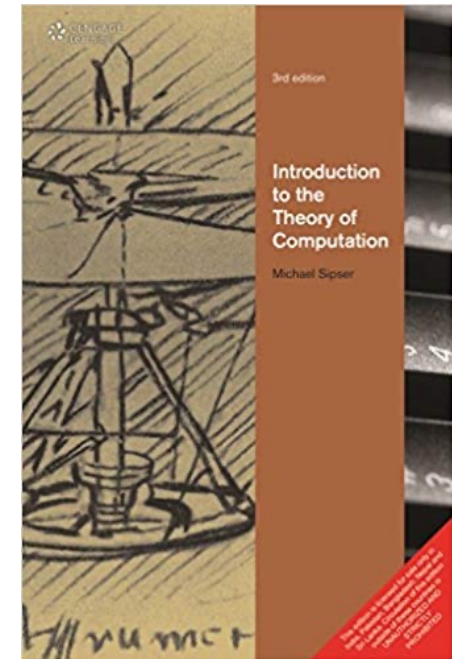
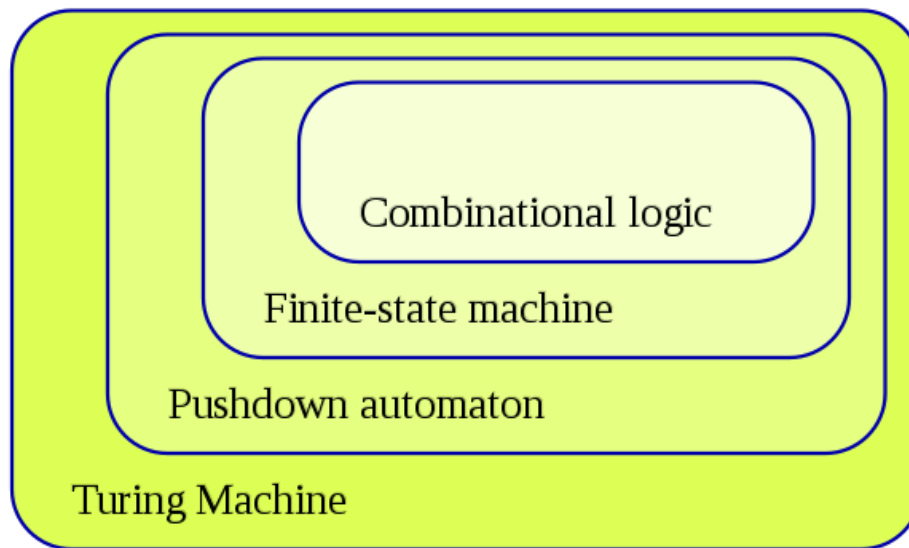
== VS. ===

- In general equality checks in Verilog will pad 0s (or 1s) as needed (so you're not really doing checks on the size of the array)
- There's two types of equality checks in SystemVerilog:
 - == compares 1's to 0's only
 - === compares against all four types in an array:

```
logic a,y;
initial begin
$display(5'b00001 == 8'b0000_00001); //eval to true
$display(5'b00001 === 8'b0000_00001); //eval to true
$display(5'b00001 == 8'b1000_0001); //eval to false!!
$display(5'b00001 === 8'b1000_0001); //eval to false!!
$display(a==y); //eval to undefined
$display(a===y); //eval to true (Since both are themselves X (undefined))
end
```

Levels of Complexity in Computation

Automata theory

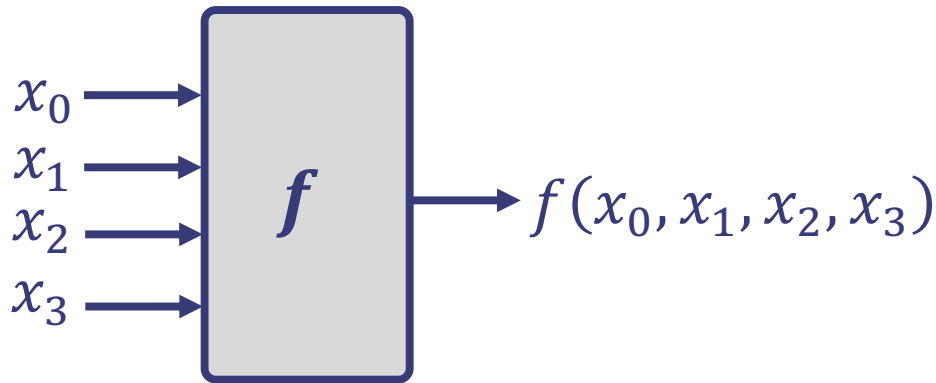


Sipser's Book

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

Two Broad Types of Digital Logic

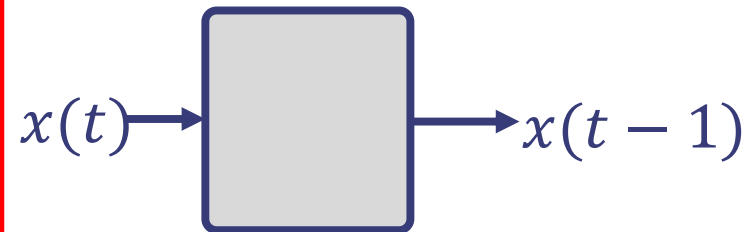
Functions:



Stateless

Current Output is
based ONLY on
current Inputs
NOT a function of
time

Storage:

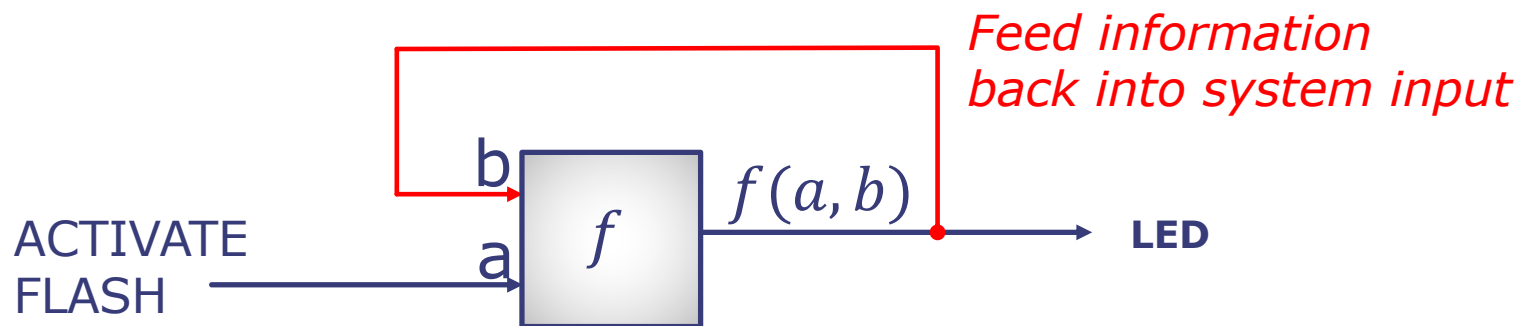


Stateful

Current Output
is based past
Input

A Problem

- Some things that we want to do in life need “history” to work (statefulness)
- Consider this attempt at an LED flasher:



- How fast will this flash?

IUNNO

a	b	$f(a, b)$
0	0	0
0	1	0
1	0	1
1	1	0

An Attempt

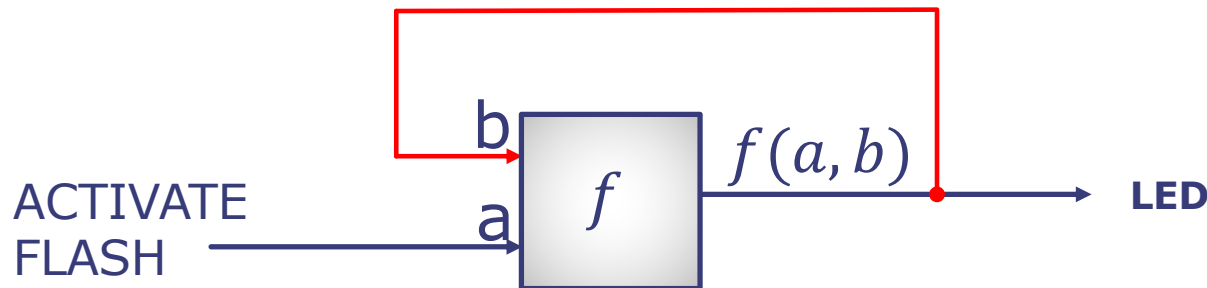
- One Attempt at Doing this:

```
module flasher (input wire a_in,  
                output logic led)  
    logic b;  
    always_comb begin  
        if (a_in) begin  
            b = ~b;  
        end else begin  
            b = 0;  
        end  
    end  
    assign led = b;  
endmodule
```

- This will fail

Need a Way to Regulate Information Flow

- The feedback loop on the previous slide is a **combinational loop**.
- These are very difficult to get to behave
- Vivado will actually fail to build if you get one of these



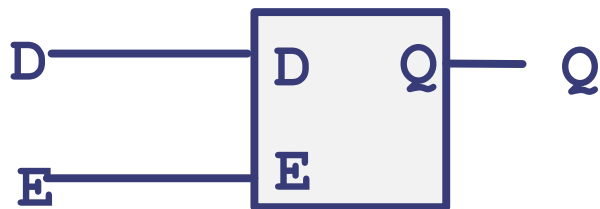
- $a, b \rightarrow f(a, b), a \rightarrow f(a, b), a \rightarrow f(a, b), a \rightarrow f(a, b), a \rightarrow kaboom$
- The assumptions we make about digital systems fall apart in this form

A part that remembers

- All the parts on the previous page have outputs based only on inputs.
- What we need are parts that do more than that.
- This arrives in the form of two components:

D Latch

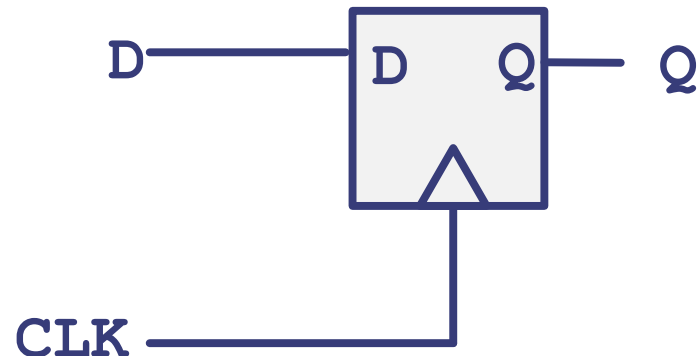
Level-Triggered Sample-and-Hold Device



"store D when E is high"

D Flip-Flop

Edge-Triggered Sample-and-Hold Device



"store D when clk rises"

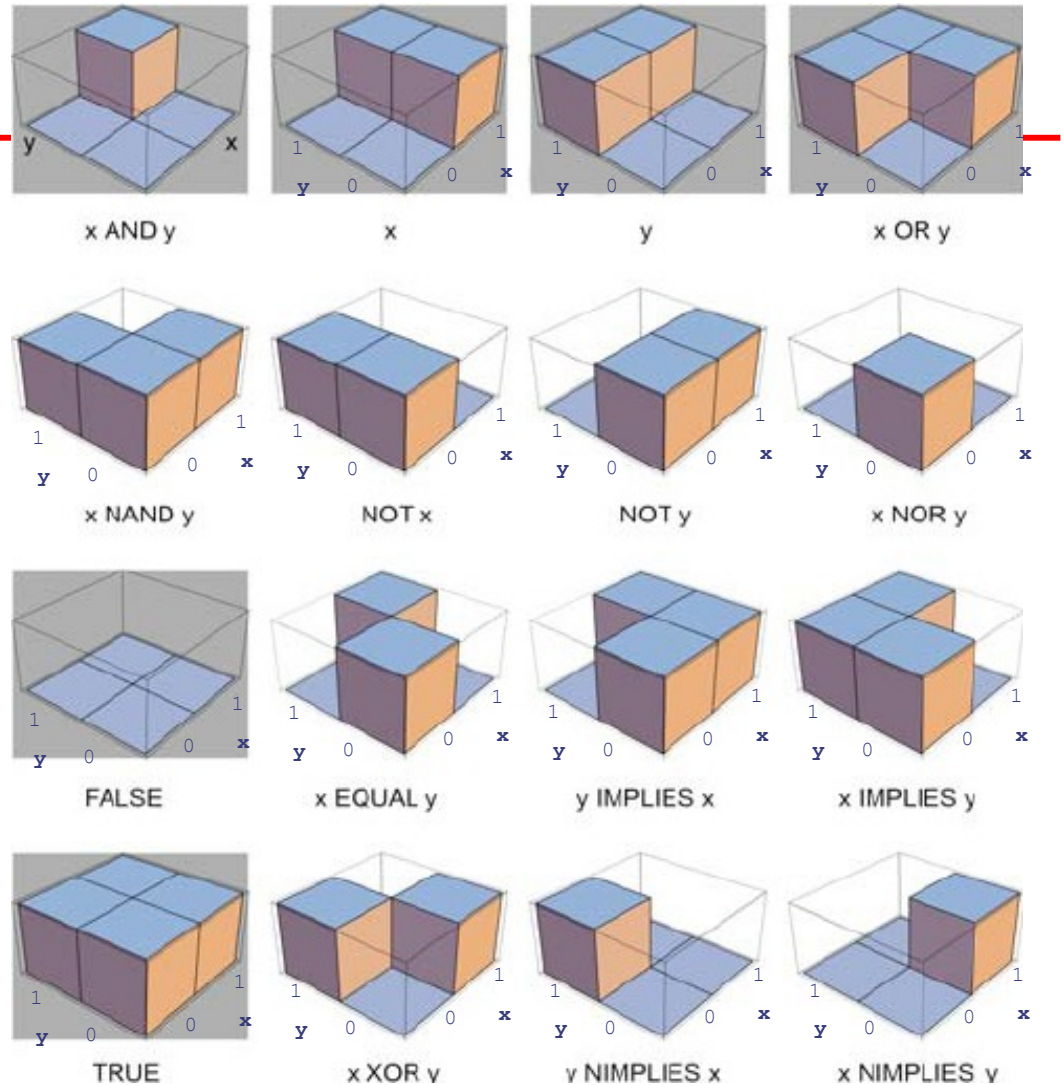
2-bit functions:

$$f(x, y)$$

x	y	$f(x, y)$
0	0	$f(0,0)$
0	1	$f(0,1)$
1	0	$f(1,0)$
1	1	$f(1,1)$

$2^4 = 16$ possible functions exist

*Stated another way:
there are 16 unique 1-0 combinations for:
 $f(0,0)$, $f(0,1)$, $f(1,0)$, and $f(1,1)$*



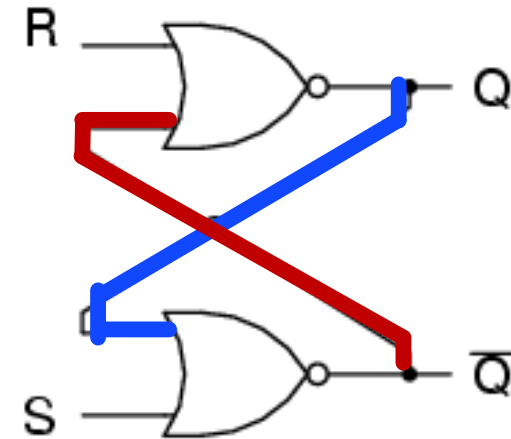
Mayo, Avi & Setty, Yaki & Shavit, Seagull & Zaslaver, Alon & Alon, Uri. (2006).
Plasticity of the cis-Regulatory Input Function of a Gene. *PLoS biology*. 4. e45. 10.1371/jour

SR-Latch

NOR		
A	B	Output
0	0	1
1	0	0
0	1	0
1	1	0

- 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”*

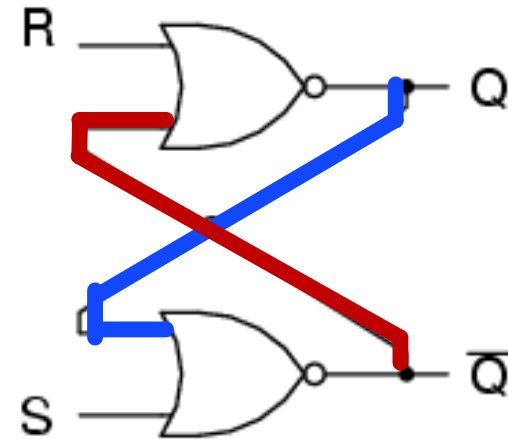


S	R	Q	$\bar{Q}$
0	0	latch	latch
0	1	0	1
1	0	1	0
1	1	0	0

Q and $\bar{Q}$ is just that these two signals are *always* opposite

But *are* they?

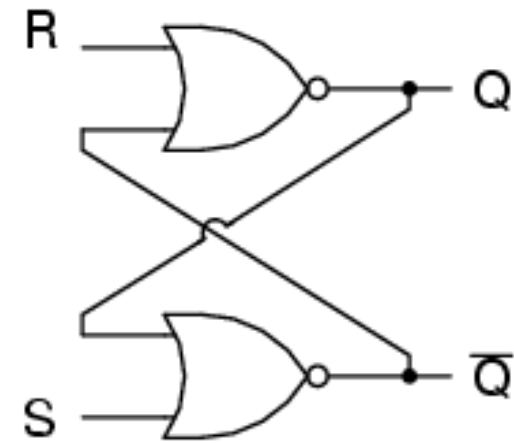
- If both S and R are asserted, both Q and $\bar{Q}$ **are held at 0**
- However in the first three pairs of the input space, Q and $\bar{Q}$ **must resolve to be opposite values.**



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

Input transitions

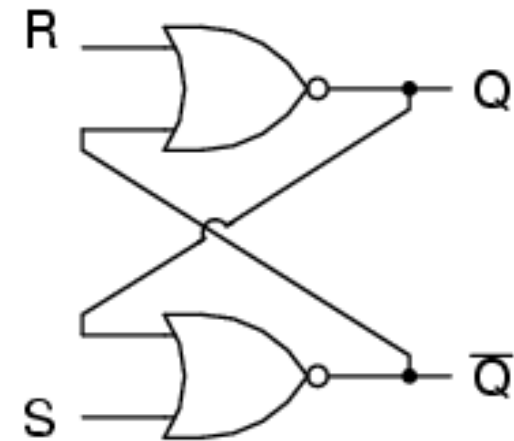
- Fine to move from:
 - $S = 0, R = 1 \rightarrow S = 0$ and $R = 0$
 - $S = 1, R = 0 \rightarrow S = 0$ and $R = 0$
- In both situations, the latching phenomenon works.
- You can set Q to 1 with $S = 1$ and $R = 0$ and then remember that you set it in the future
- Or you can “reset” Q to 0 with $S = 0$ and $R = 1$ and then remember that in the future



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

But but but....What if...what IF?

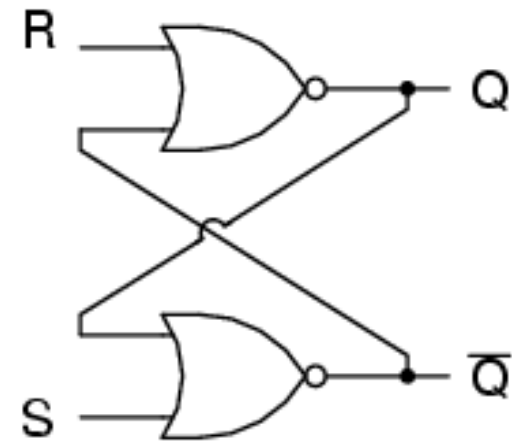
- What if you do this:
 - $S = 1, R = 1 \rightarrow S = 0$ and $R = 0$
- You're moving from a state where Q and $\bar{Q}$ are the same to a state where they must be different.
- But they can't inherit a user-specified difference since there wasn't one.
- Is it latching to:
 - $Q, \bar{Q} = 1, 0$ or
 - $Q, \bar{Q} = 0, 1$????????



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

Undefined Behavior

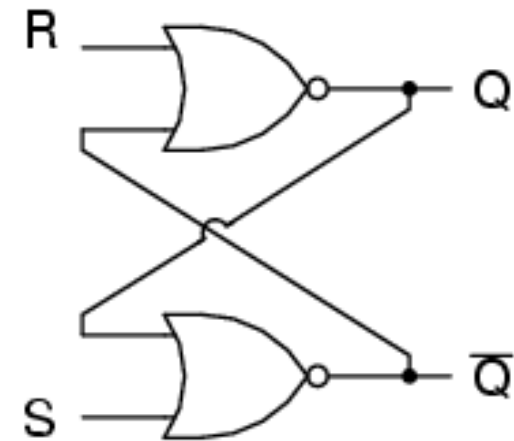
- If you do this to an SR-latch, its behavior will be **undefined**!
- The output will eventually resolve to either 0,1 or 1,0, but which one it does will be non-deterministic
- This arises from the feedback in the circuit!!!
- And the feedback may make the undefined nature take a while to resolve



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

Life and Death

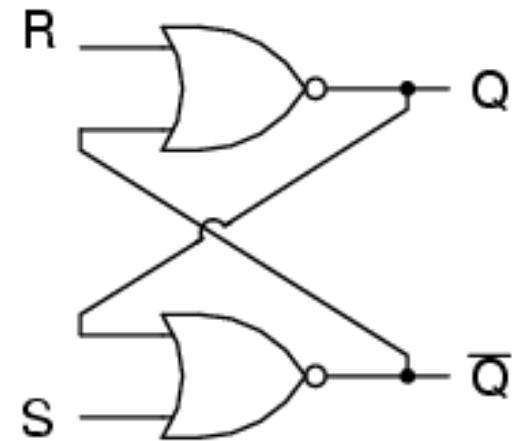
- With no reliable pattern to inherit going into the latch state, the two NOR gates are forced into a life-and-death race-condition struggle where one of them will eventually come out on top.



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

Undefined Behavior

- If you do this to an SR-latch, its behavior will be **undefined**!
- The output will eventually resolve to either 0,1 or 1,0, but which one it does will be non-deterministic
- This arises from the feedback in the circuit!!!

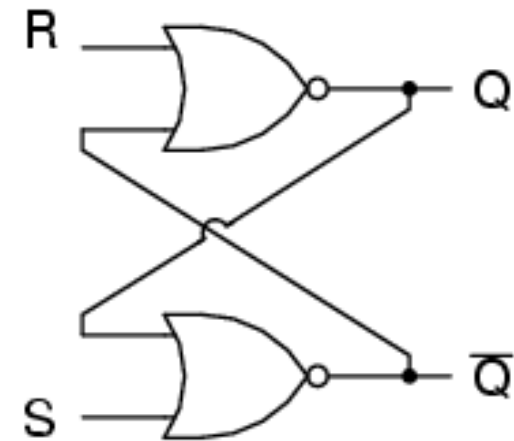


- Solution could be to avoid ever setting
 - $S = 1, R = 1$
- But more specifically...never do a $S = 1, R = 1 \rightarrow S = 0, R = 0$ transition

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

Other Problems with SR Latch?

- When used properly, the SR latch will remember for us, but what it remembers is based off of a command (value of S,R).
- It would be nicer if we could have a memory module that could just remember data.
- So ignoring that transition issue, it isn't the most useful

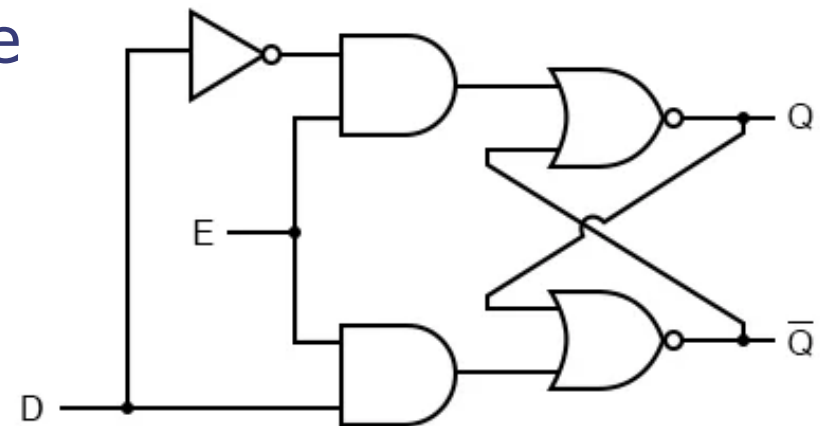


S	R	Q	$\bar{Q}$
0	0	latch	latch
0	1	0	1
1	0	1	0
1	1	0	0

verboten

The D-Latch

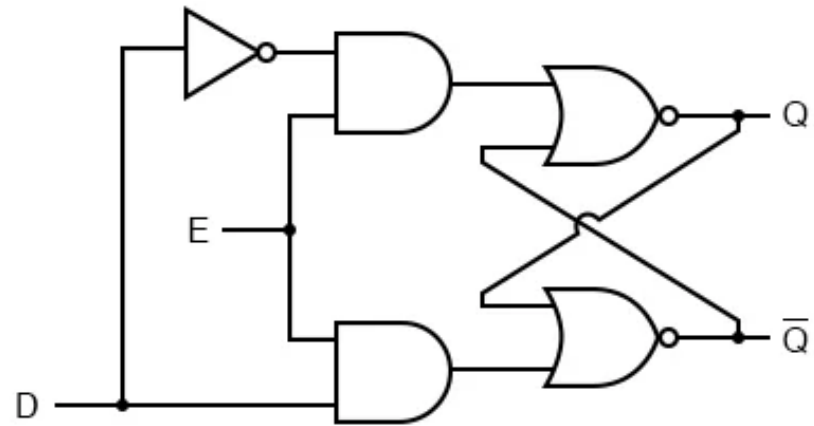
- The solution is to add some input control logic to control the S and R signals to repackage as a different memory storage unit.
- Still entirely made of combinational logic gates, but now with a much more usable behavior:...



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

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. The decision about what to store is data-dependent



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

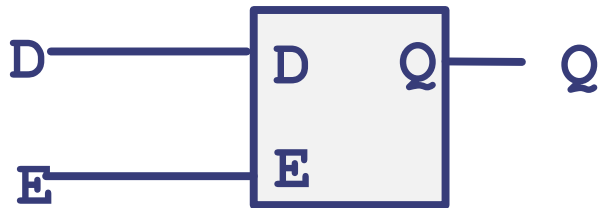
D = “Data”

E = “Enable”

Q = is the output

D-Latch

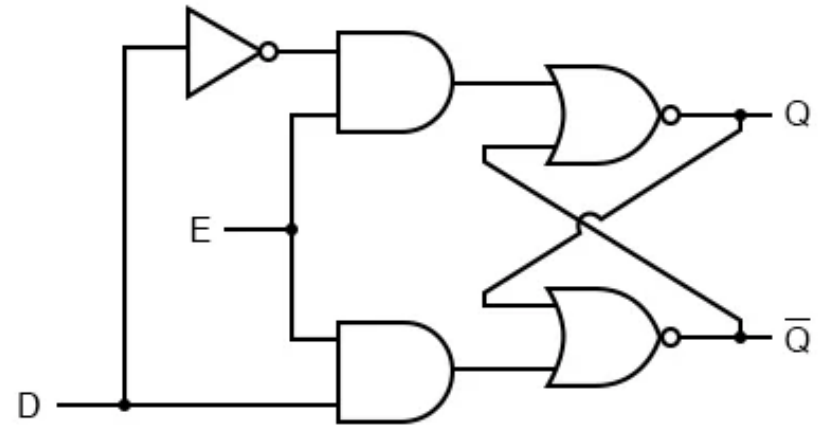
- A level-based sample-and-holding device
- Turn on E, Whatever is at D, shows up at Q
- When E is on, the latch is “***transparent***”
 - D directly controls what Q is in this state...
- When E turns off, the last value that D had before turning off is locked in and held at Q output



D	E	Q
0	0	Previous Q
0	1	0
1	0	Previous Q
1	1	1

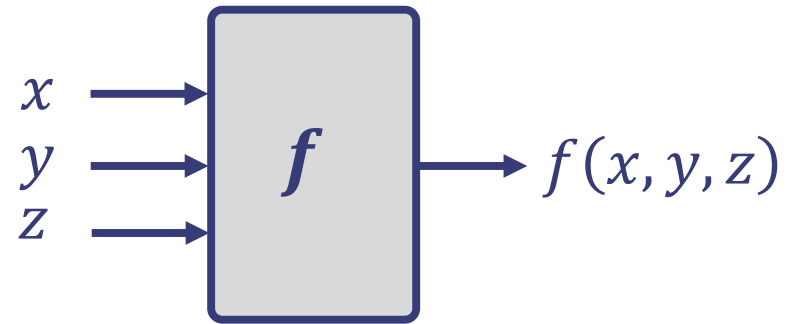
Good Truth Table.

- At a high level, there is no inherent input-to-input transition problem like you get with the SR-latch.
- But....



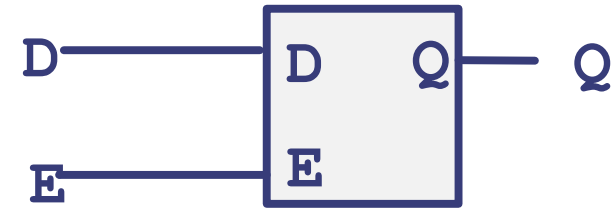
E	D	Q	$\bar{Q}$
0	0	latch	latch
0	1	latch	latch
1	0	0	1
1	1	1	0

Delays



- Two big numbers for a Digital Function:
 - t_{cd} : **Contamination Delay:** The minimum time it takes from an input change on a function to appear at output of function
 - t_{pd} : **Propagation Delay:** The maximum time it takes from an input change on a function to appear at output of function
- Therefore, when a new input is presented to a digital (combinational function), it will take between t_{cd} and t_{pd} for it to calculate!
- Can also think of these as best/worst case response times
- During t_{pd} **the input must be held stable**

Implications



- The D-Latch is just made up of logic gates so it will have its own t_{pd} and t_{cd} in operation, meaning it will take time to respond to input changes!
- This is good to note down and when we design, we need to take that into account...

D	E	Q
0	0	Previous Q
0	1	0
1	0	Previous Q
1	1	1

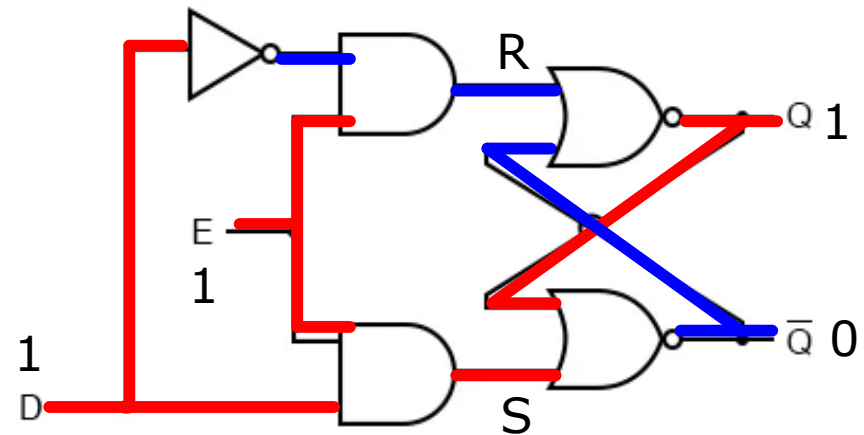


Replace "Stars" with "Latches" and "Us"
"Combinational Logic" for joke to operate

But more dangerously...

1 ——— red
0 ——— blue

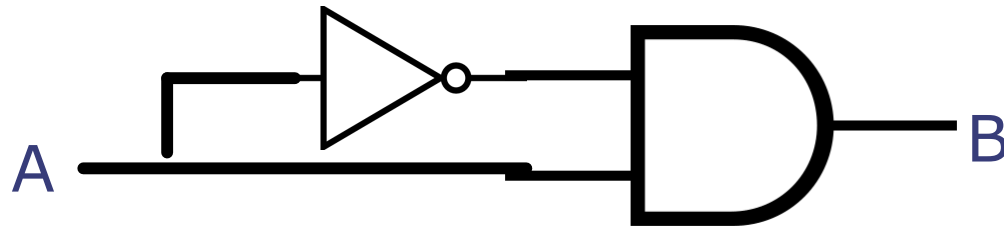
- The various internal components also have their own t_{pd} and t_{cd}
- And this can open up the chance for glitches to appear!



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

What is a glitch?

- Consider this circuit...



A	B
0	0
1	0

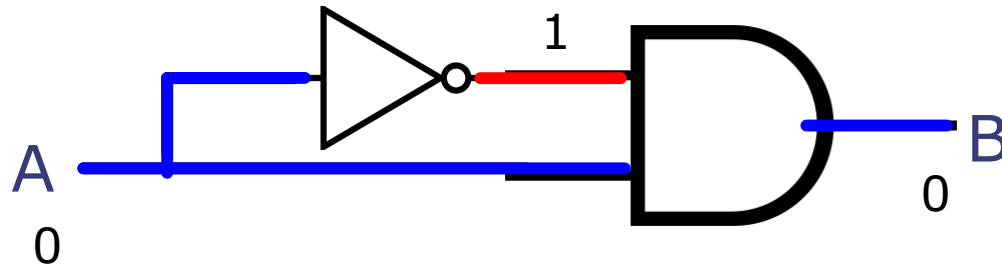
- For all input combinations, there should be no way for this to ever have anything but 0 at the output
- BUT

What is a glitch?

1 
0 

- Consider a $0 \rightarrow 1$ transition

A	B
0	0
1	0



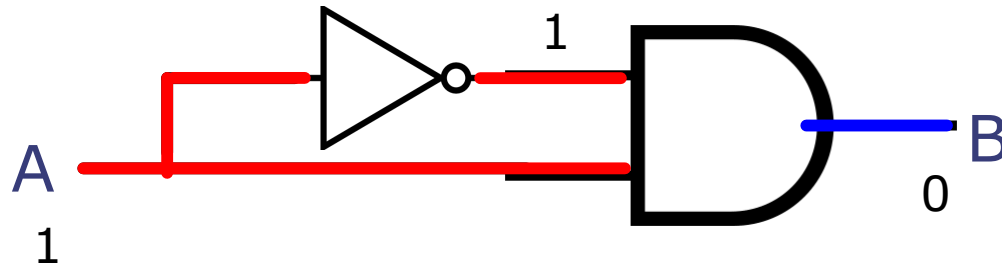
- First A transitions high...

What is a glitch?

1 
0 

- Consider a $0 \rightarrow 1$ transition

A	B
0	0
1	0



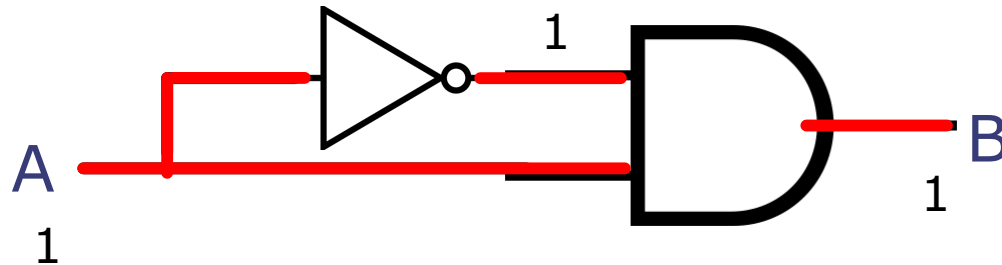
- The AND gate is now exposed to a 1,1 input. The Inverter is now exposed to a 1. They both start reacting...

What is a glitch?

1 
0 

- Consider a $0 \rightarrow 1$ transition

A	B
0	0
1	0



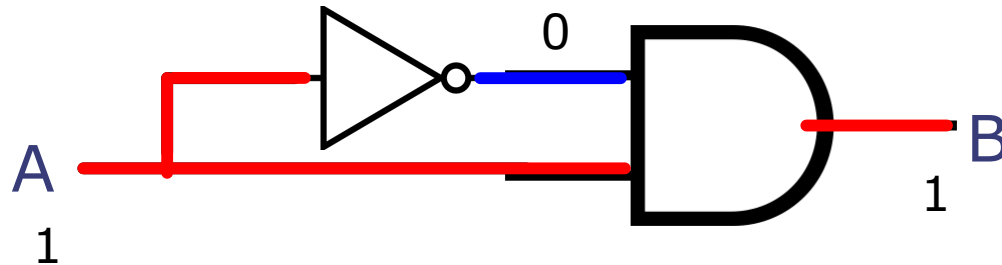
- Depending on the different paths' and components' t_{pd} and t_{cd} it is entirely possible that the AND gate starts to change at its output to a 1.....

What is a glitch?

1 
0 

- Consider a $0 \rightarrow 1$ transition

A	B
0	0
1	0



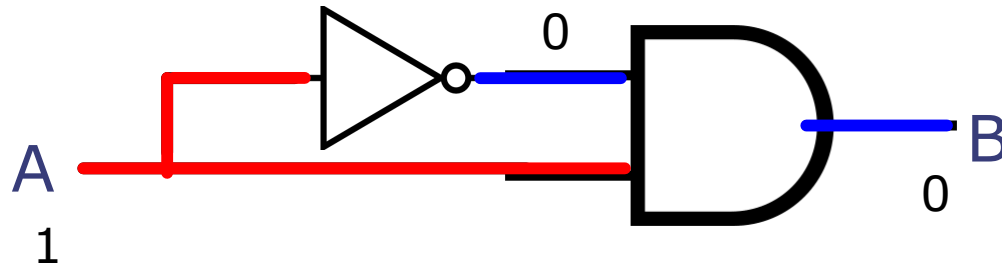
- Before the inverter catches up...

What is a glitch?

1 
0 

- Consider a $0 \rightarrow 1$ transition

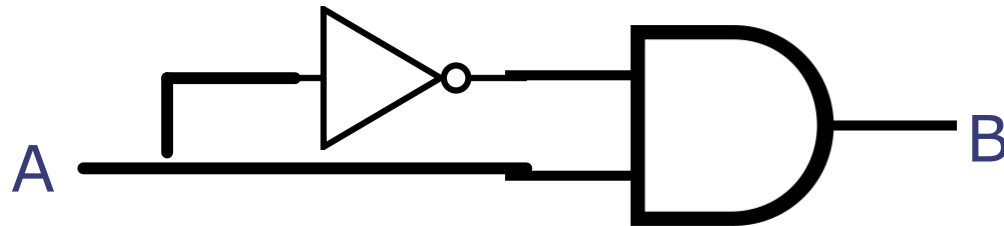
A	B
0	0
1	0



- Before the inverter catches up...

But nevertheless

- Consider this circuit...



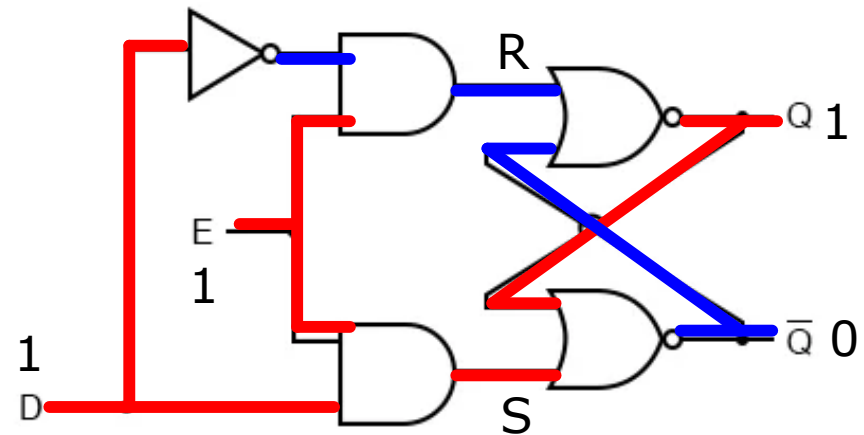
A	B
0	0
1	0

- This circuit can make a 1 at its output!!! Surprise.
- In this situation it is funny haha thing

In a D Latch

1 ——— red
0 ——— blue

- Certain input transition combinations can cause particularly troubling internal signal transitions.
- In particular...changing both D and E at the same time can result in possible temporary internal states where the R,S pins go $(1,1) \rightarrow (0,0)$



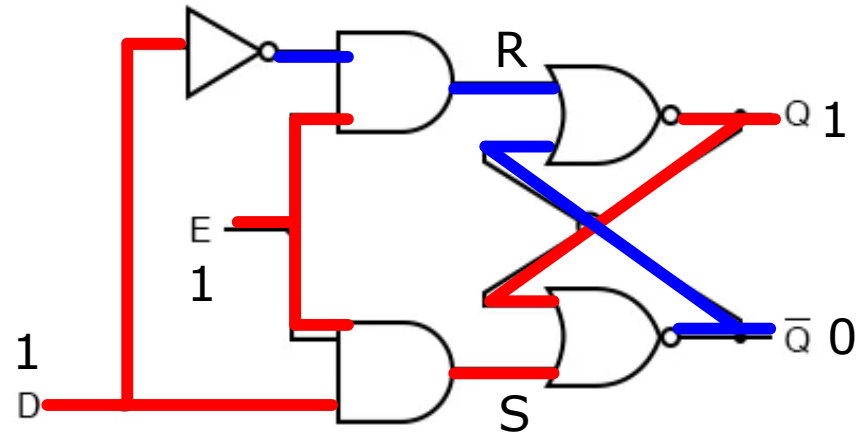
E	D	Q	$\bar{Q}$
0	0	latch	latch
0	1	latch	latch
1	0	0	1
1	1	1	0

And if the D latch does that

- Its output will no longer be reliable.
- It will find itself in a metastable state where the output is unusable.

So restrict D-latch transitions to...

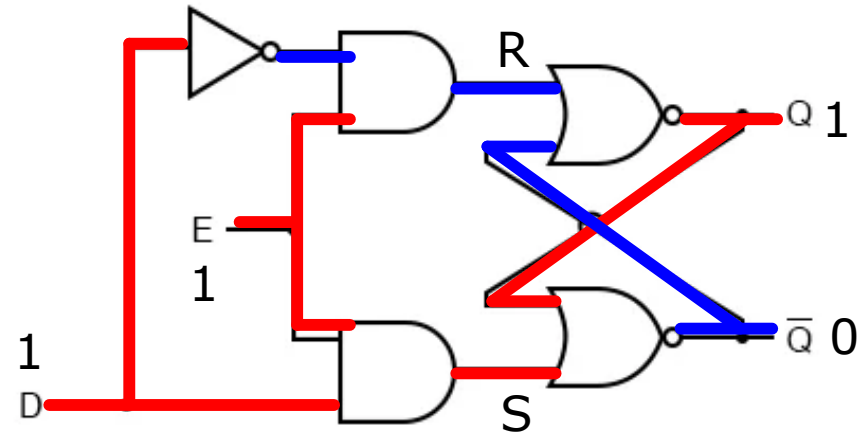
- Keep D the same, and change E
- Keep E the same, and change D
- Transitions like this will never have intermediate states that expose a
 - R,S (1,1)→(0,0) transition



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

Setup and Hold Times

- The avoidance of these dangerous transitions is going to lead us into the need for **setup and hold times** which we will obsess a lot about!



NO:

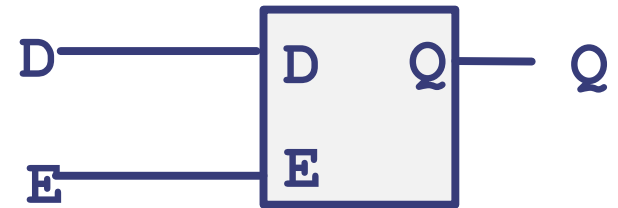
E	D	Q	$\bar{Q}$
0	0	latch	latch
0	1	latch	latch
1	0	0	1
1	1	1	0

YES:

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

Now

- Let's assume we can follow that rule about not changing D and E at the same time...let's just return to the latch....

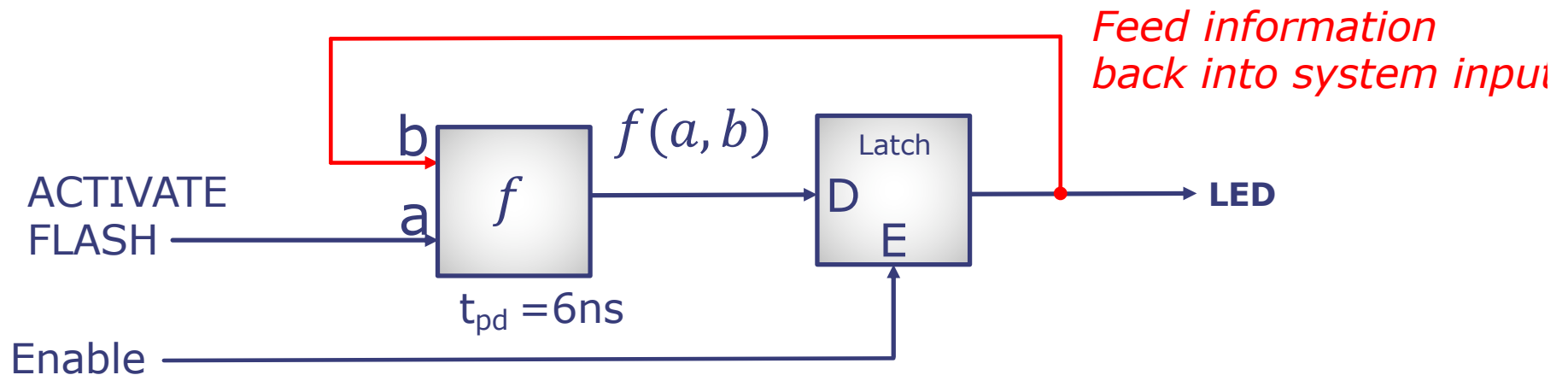


- The D-latch gives us the chance to remember data. This is great!!!!

D	E	Q
0	0	Previous Q
0	1	0
1	0	Previous Q
1	1	1

Is Level Storing Good?

- Return to Flasher (now with latch):



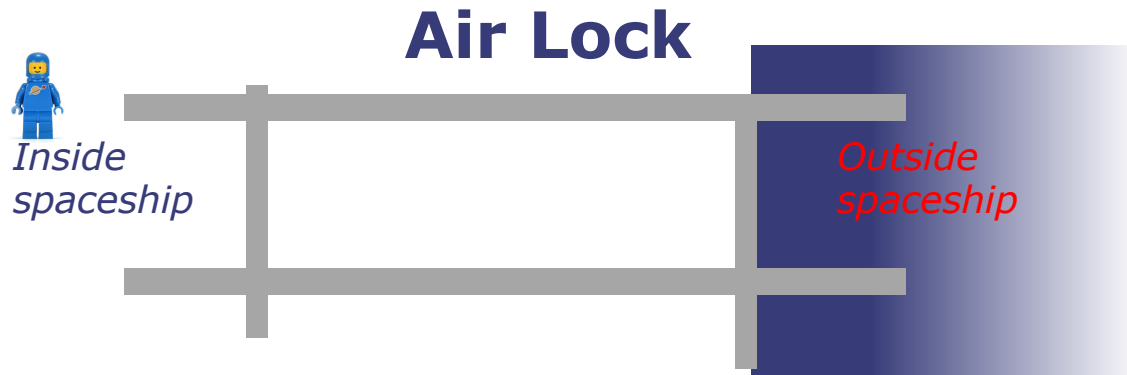
- The latch can store information, but when transparent ($E = 1$) the latch is allowing a combinational loop to exist:
 - $a, b \rightarrow f(a, b), a \rightarrow f(a, b), a \rightarrow f(a, b), a \rightarrow f(a, b), a \rightarrow \text{puke}$

What Do We Want?

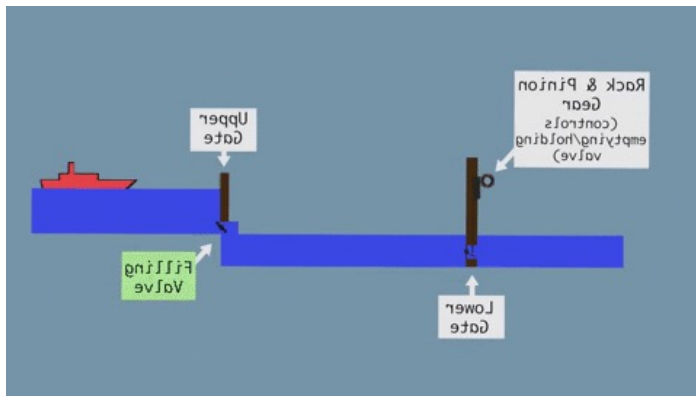
- We want to store the past info
- Also want to prevent unregulated flow of information (the transparent nature of a latch)
- A D-latch can't do that. It is like a door. You open a door and things and insects and pollen and people who live on your hall that you don't like talking to can flow in/out all crazy.
- We need something that will act as a digital air-lock or "sally port" or canal lock

What do we want?

Prevent unregulated/ dangerous transfer of things forward (and also backwards too)



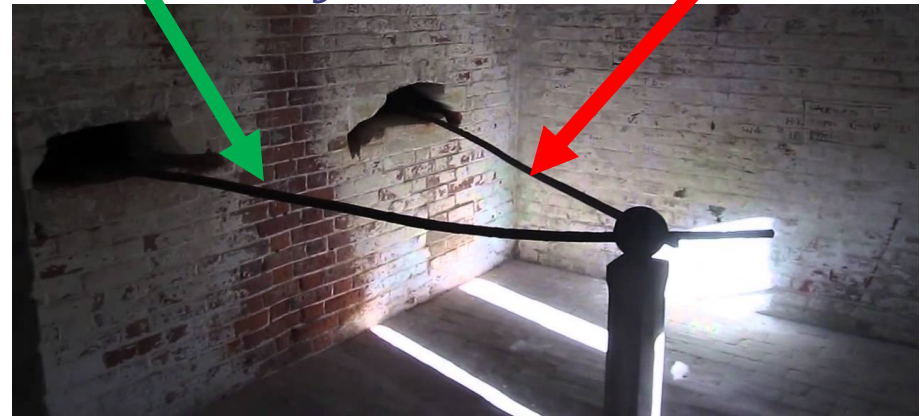
Canal Lock



Open/close door to inside castle

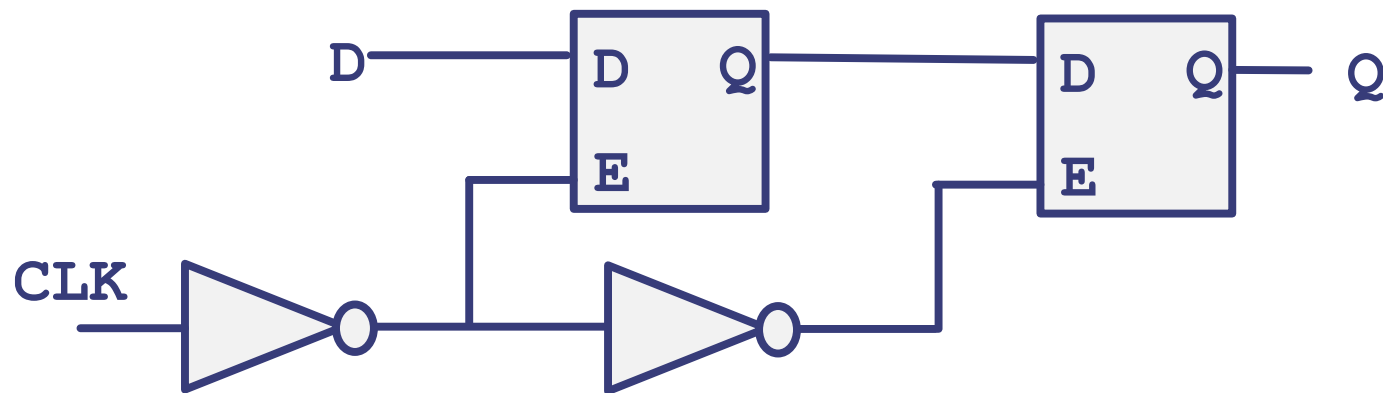
Never open both at same time!!!

Open/close door to outside castle



Sally Port Dover Castle, England

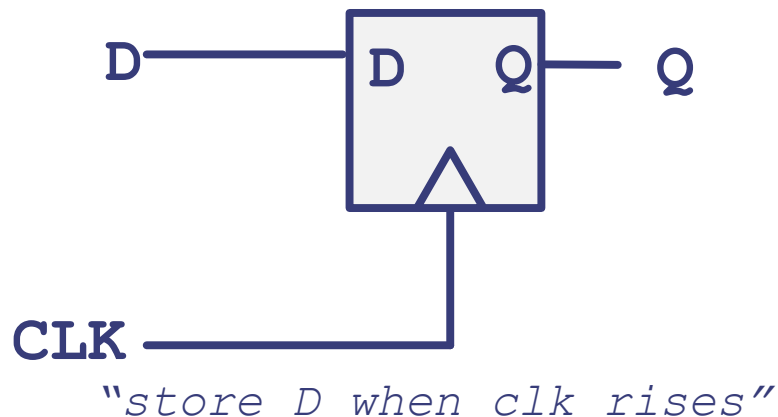
Basically put two D-latches in series with alternate controls...



The D Flip-Flop

- A Flip-Flop (aka "register") only samples the value of D when clk goes 0→1. Shortly after that it transfer the captured value of D to Q. And holds it there under all other conditions
- Since rising edge of clk is short, no opportunity for transparency

Edge-Triggered Sample-and-Hold Device



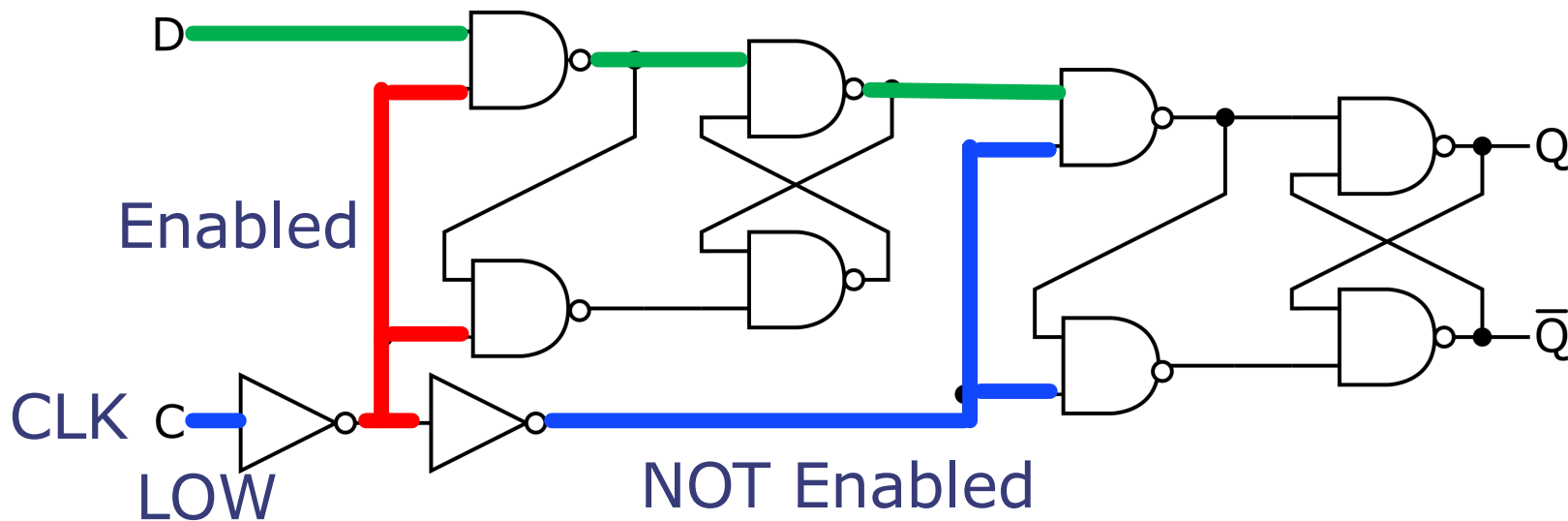
D	CLK	Q
0	Rising	0
1	0	Previous Q
0	1	Previous Q
0	Falling	Previous Q
1	Rising	1
1	0	Previous Q
1	1	Previous Q
1	Falling	Previous Q

The D Flip-Flop (Reg)

Sorry I know these are NAND gate D-latches, but I messed up my animations for the NOR gate latches.

Two D-Latches in Series driven with opposite enable signals

Data propagates through first D Latch



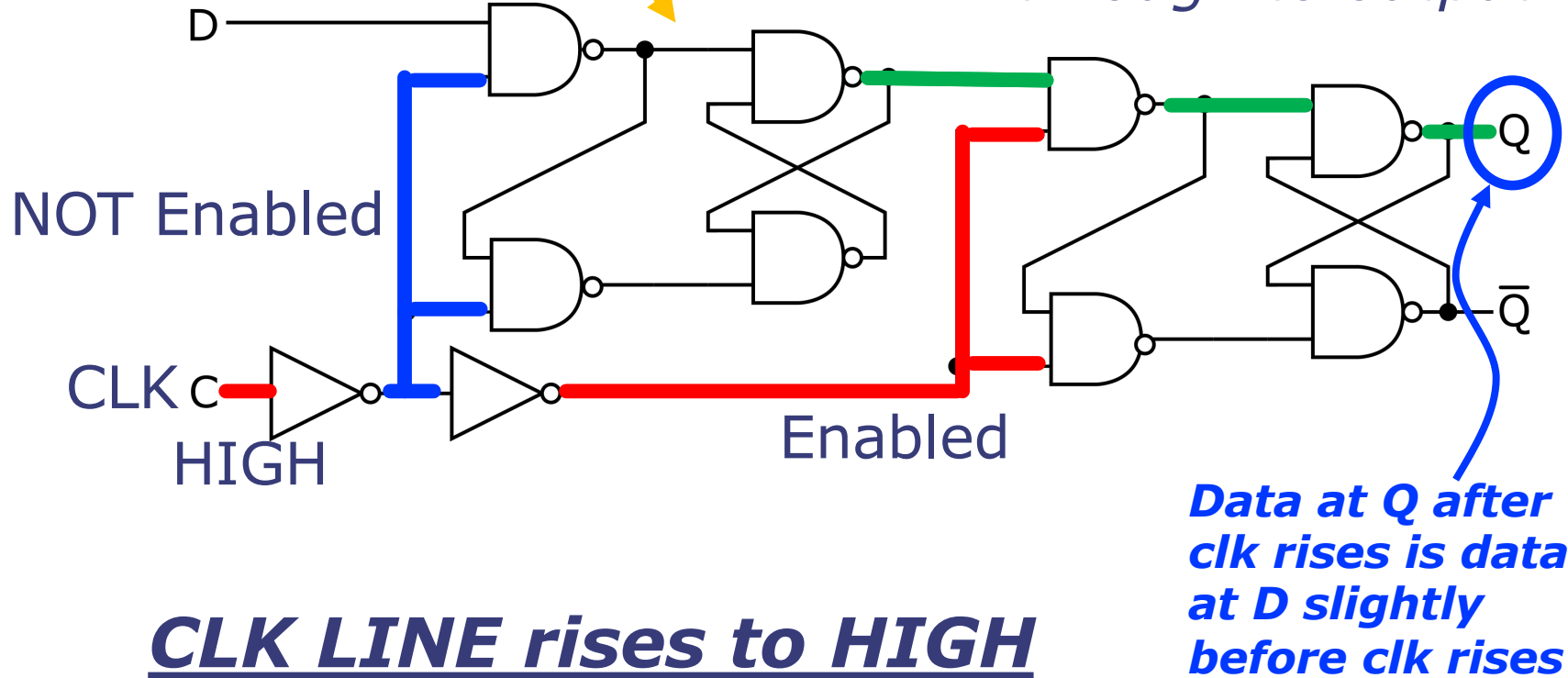
CLK LINE is LOW

The D Flip-Flop (Reg)

Two D-Latches in Series driven with opposite enable signals

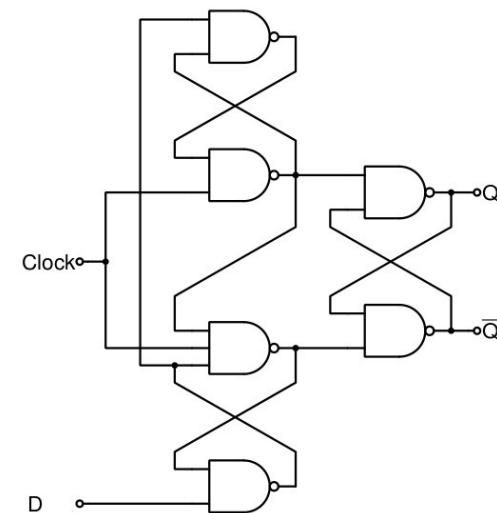
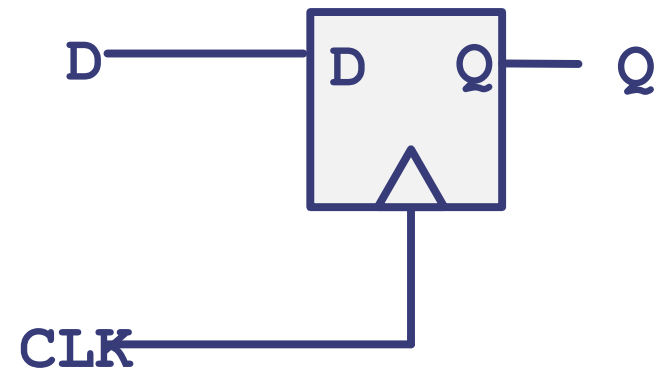
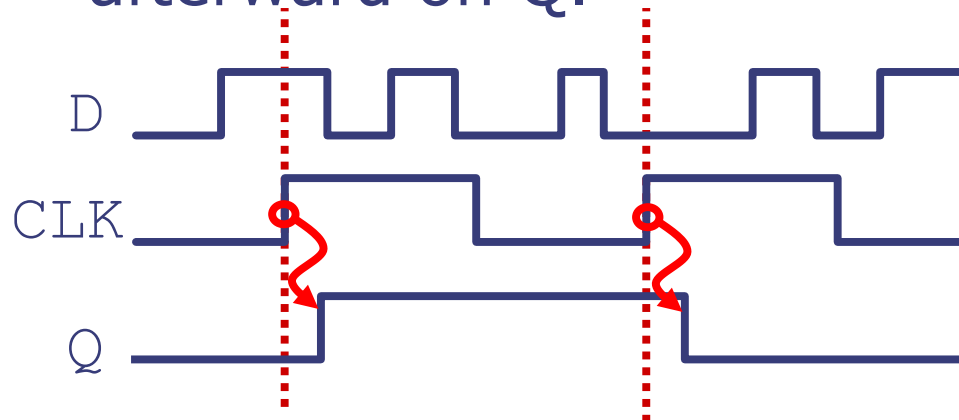
No longer transparent

Data propagates through to output



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

Stackable

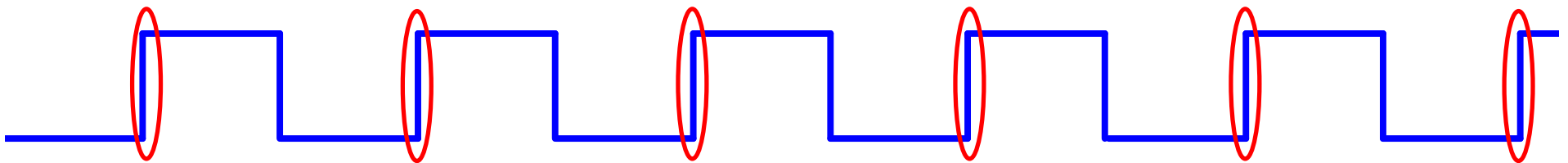
- Because Flip-Flops are *never transparent* like a latch, we can have all of them flip or flop at the same time and there's no risk of any brief momentary shoot-through/transparency!
- This is NOT something you can do with latches!

A Clock Signal

- We therefore usually use Flip Flops in conjunction with periodic signals of fixed frequency
- Call these Clock Signals and they regulate how often the input is sampled and transferred to the output
- The default frequency on our FPGA is 100 MHz

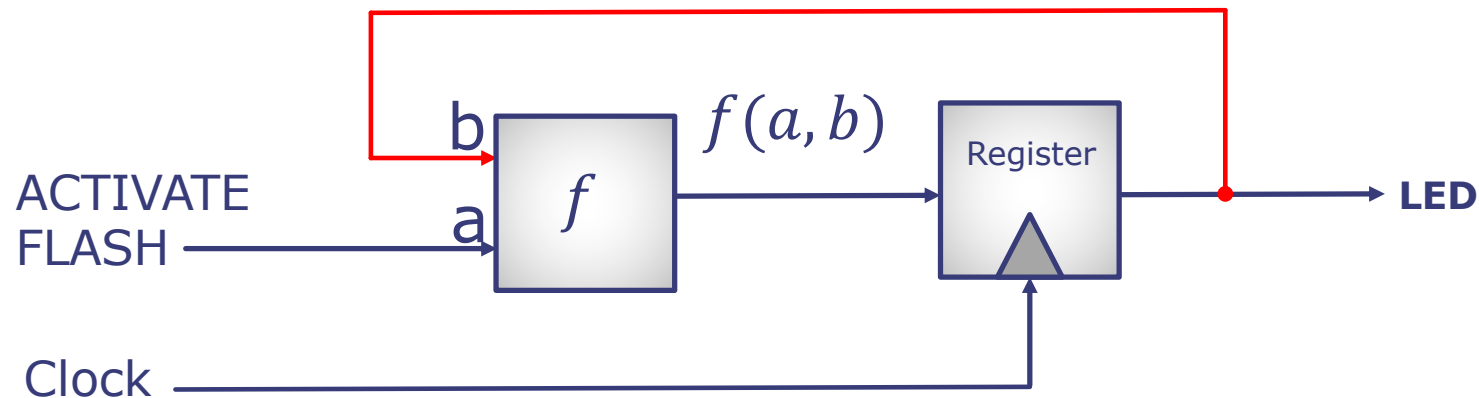
Clock signal:

Usually we will "step" on the rising edge of a clock signal



Solving the Flasher

- Let's put a FlipFlop in the Feedback Loop



- A flip flop is **NEVER** transparent like a latch!
- It allows us to store and regulate the flow of information in a system.
- Make clock a periodic signal and then things start to work

Sequential Verilog

- So far we've mostly been seeing Combinational Verilog modules*.
- How do we start to express these new ideas of sequential logic in Verilog?

*yes you used them a bit in week 1 lab

Always

- In Verilog the `always` keyword is a way to specify logic (sequential, combinational) that is caused by an event (clock edge, change of state, etc)
- Very similar to an asynchronous callback in Javascript etc:
 - “When an event happens, do a certain thing:”
- Historically there was one `always` word and you would then specify a sensitivity list:
 - `always @(x)` = “when x changes”
 - `always @(*)` = “when anything changes (combinational)”
 - `always @(posedge clk)` = “when clk edge rises”
 - Etc...

Regs, Wires, Logics, and Life

- Original Verilog had two main datatypes
 - **wire**: Used for continuous assignment (combinational)
 - **reg**: Used to “store” values
- Despite its name being short for “register” a **reg** might not actually mean the design will synthesize to an actual *register*...It depended on usage in the Verilog.
- In particular it mostly depended on your sensitivity list in your **always** block and if you used block or non-blocking assignments (= or <=):
 - *posedge?* Make a flip flop
 - *values?* Make it a combinational
 - *or possibly a latch*

SystemVerilog

- Mostly drop the reg/wire terminology, just have **logic** and let compiler figure out if it becomes an actual register (flip-flop) or wire/net from use
- Use is specified more clearly now by replacing ambiguousness of generic **always** with specific use cases:
 - **always_comb**: build using combinational logic
 - **always_ff**: build using D-flip-flops (edge-trig sequential)
 - **always_latch**: build using D-latch (level-trig logic)
- What is synthesized is NOT “inferred” and more clearly based on user specification! 😊

Why Logic?

- In addition to allowing us to just use one general type rather than two, the **logic** datatype has stricter protections against multi-driven nets

```
module thing(input wire [3:0] a_in,b_in,
output wire [3:0] c_out);
    //stuff that changes c_out
endmodule

module main_module();
    logic[3:0] a,b,c;
    thing my_thing(.a_in(a), .b_in(b), .c_out(c));
    assign c = 4'b1010; //whoops might make it through (multi-driven net)
endmodule
```

- Logic on output should prevent this:

```
module thing(input wire [3:0] a_in,b_in,
output logic [3:0] c_out);
    //stuff
endmodule

module main_module();
    logic[3:0] a,b,c;
    thing my_thing(.a_in(a), .b_in(b), .c_out(c));
    assign c = 4'b1010; //should get caught on synthesis
endmodule
```

We use an always_ff to make flipflops

```
always_ff @(posedge clk)begin
    //do things ON the rising edge of clk
end
```

- You use @(posedge clk) to specify that the flipflop is triggered on the positive (rising) edge of the clk signal
- Can also do negedge (negative/falling edge) if you want, but we'll rarely need that in 6.205.

Blocking vs. Nonblocking Assignment

- Within any type of **always** block you can assign things in two different ways:
- In both ways, you don't need the keyword **assign**
- *Blocking assignment (=)*: evaluation and assignment are immediate; subsequent statements affected. (ORDER MATTERS)
- *Nonblocking assignment (<=)*: all assignments deferred to end of simulation time step after all right-hand sides have been evaluated (*even those in other active always blocks*) (ORDER DOESN'T MATTER)

Blocking Assignments $\leq$

- Blocking Assignments lend themselves to thinking about time
- $a \leq a + 1$; will allow us to express the idea of future a will be based on current $a + 1$
- This is *not* something we ever want to do in purely combinational logic
 - We want combinational logic to express instantaneous relationships of causality
- But we do often want to do that in sequential logic!

Blocking vs. Nonblocking Assignment in Combinational

- Verilog supports two types of assignments within `always-type` blocks, with subtly different behaviors.
- Blocking assignment (=)**: evaluation and assignment are immediate

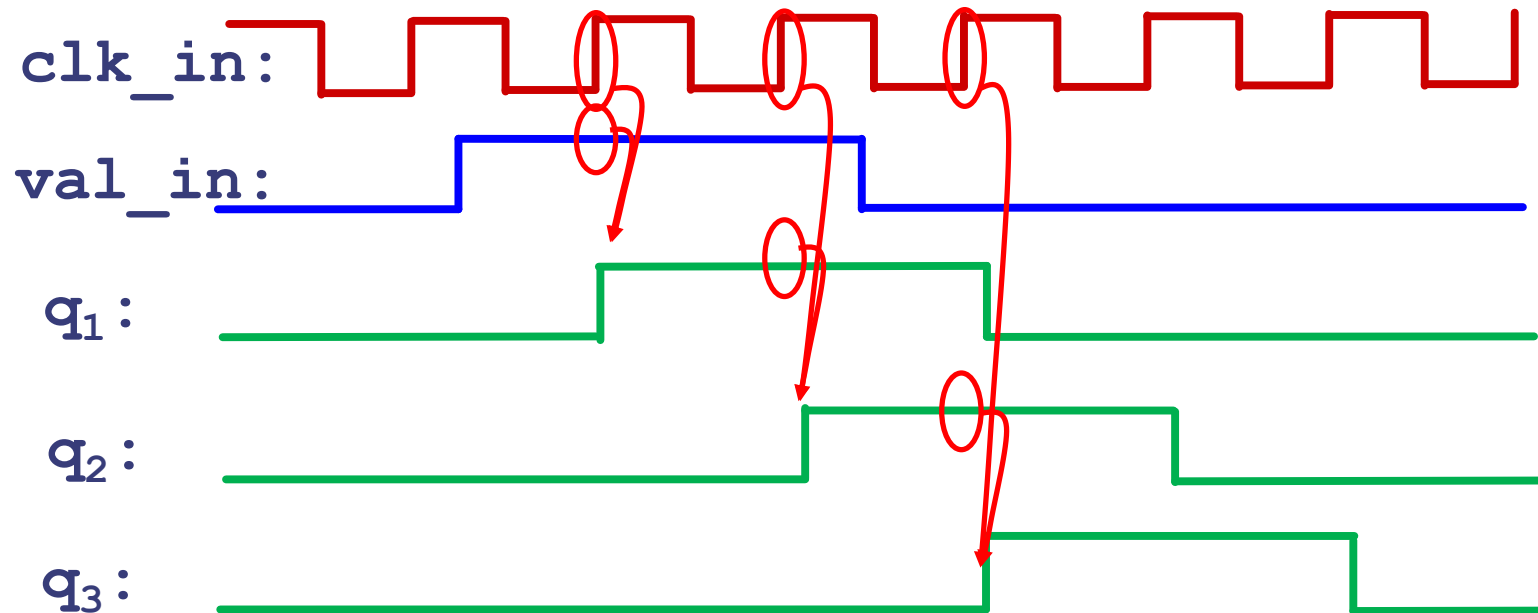
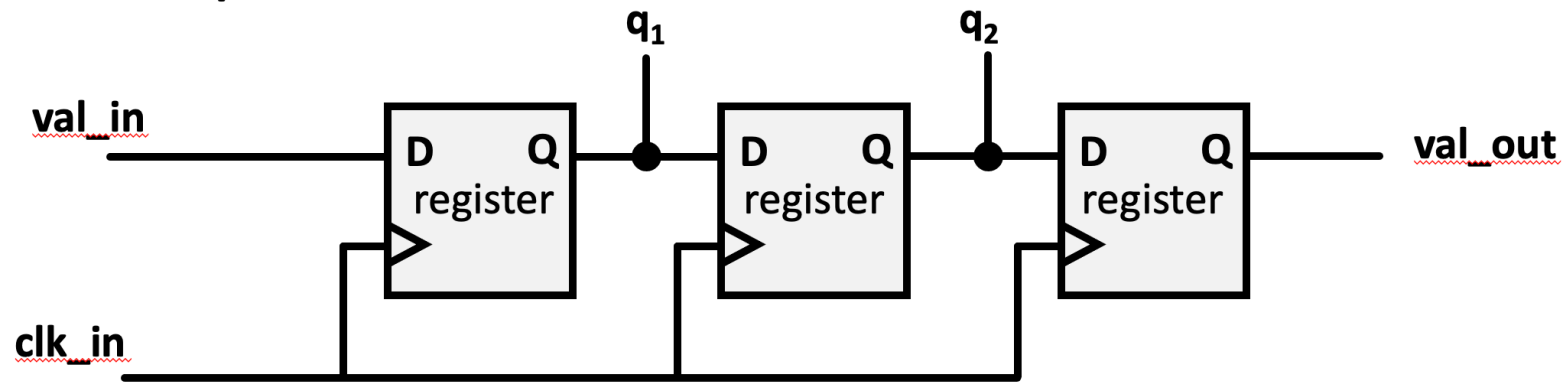
```
always_comb begin
  x = a | b; // 1. evaluate a|b, assign result to x
  y = a ^ b ^ c; // 2. evaluate a^b^c, assign result to y
  z = b & ~c; // 3. evaluate b&(~c), assign result to z
end
```

- Nonblocking assignment (<=)**: all assignments deferred to end of simulation time step after all right-hand sides have been evaluated (even those in other active `always` blocks)

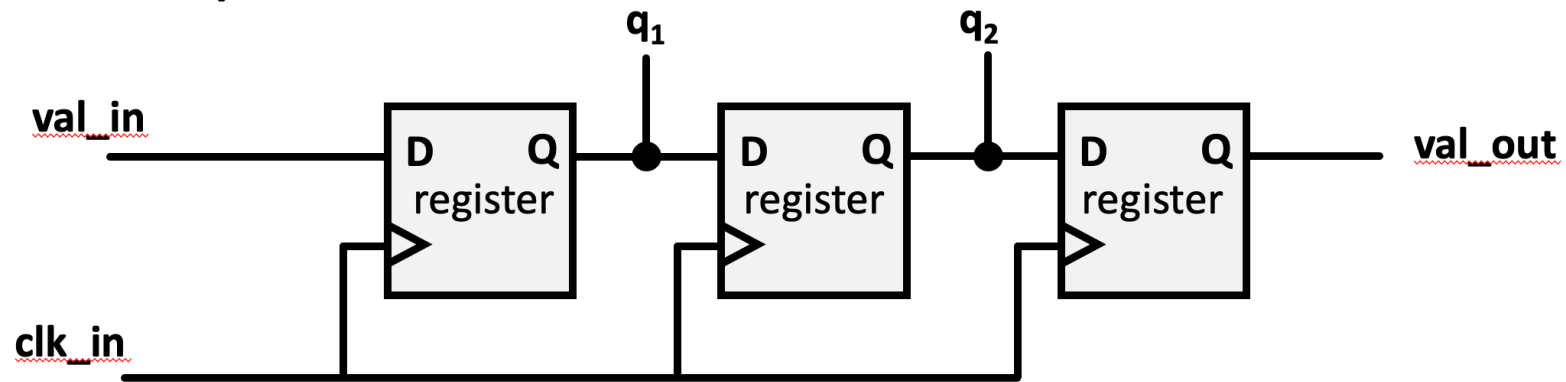
```
always_comb begin
  x <= a | b; // 1. evaluate a|b, but defer assignment to x
  y <= a ^ b ^ c; // 2. evaluate a^b^c, but defer assignment to y
  z <= b & ~c; // 3. evaluate b&(~c), but defer assignment to z
  // 4. end of time step: assign new values to x, y and z
end
```

*Sometimes, as above, both produce the same result. **Sometimes, not!***

Consider this Chained Flip Flop Circuit



Assignment Style for Sequential Logic

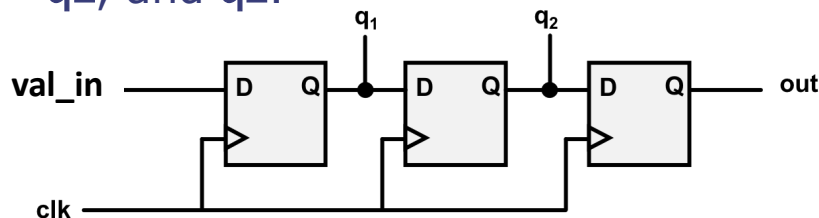


- Suppose we want to specify that circuit above in Verilog:
- Will nonblocking and blocking assignments both produce the desired result? ("old" means value **before** clock edge, "new" means the value **after** most recent assignment)

Use Nonblocking for Sequential Logic

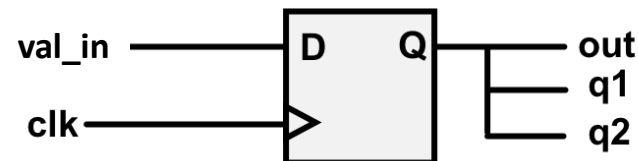
```
module nonblocking(  
    input wire val_in, clk_in,  
    output logic val_out  
);  
    logic q1, q2;  
    always_ff @(posedge clk_in) begin  
        q1 <= val_in;  
        q2 <= q1; // uses old q1  
        val_out <= q2; // uses old q2  
    end  
endmodule
```

“At each rising clock edge, **q1**, **q2**, and **out** **simultaneously** receive the old values of **vin**, **q1**, and **q2**.”



```
module blocking(  
    input wire val_in, clk_in,  
    output logic val_out  
);  
    logic q1, q2;  
    always_ff @(posedge clk_in) begin  
        q1 = val_in;  
        q2 = q1; // uses new q1  
        val_out = q2; // uses new q2  
    end  
endmodule
```

“At each rising clock edge, **q1 = vin**.
After that, **q2 = q1**.
After that, **out = q2**.
Therefore **out = vin**.”



Strong Guidelines

- Blocking assignments (=) more closely align with how combinational works (use only in `always_comb`)
- Non-blocking assignments (<=) more closely align with how sequential logic works (use only in `always_ff`)
- Do NOT mix blocking and non-block assignments within one block!
 - Something will synthesize, but sometimes simulation *may* differ from what gets synthesized (built)
 - Really hard to comprehend for our limited human minds...so debugging is a nightmare
 - Do not do it. We will make you change your code if we see it and/or send a letter the Committee on Discipline

Coding Guidelines

- The following are helpful guidelines passed down by people smarter than me. If followed, they ensure your simulation results will match what they synthesized hardware will do:
 1. When modeling sequential logic, use and **always_ff** with **nonblocking assignments (<=)**.
 2. When modeling combinational logic with an always block, use **always_comb** with **blocking assignments (=)**.
 3. When modeling both sequential and “combinational” logic within the same always block, use nonblocking assignments.
 4. Do not mix blocking and nonblocking assignments in the same always block. Just don't. You're not that good.
 5. Do not make assignments to the same variable from more than one always block (this should throw errors, but might not if using blocking assignments)
- **A Big thing we will be checking in your Verilog code!**

Taken from: “Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!” by Clifford E. Cummings

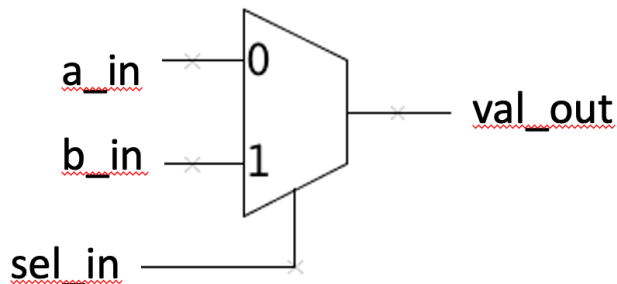
"Combinational vs Sequential"

- In combinational logic (`always_comb`, `assign`), you are making just combinational logic
- In sequential logic specification (`always_ff`), you're often specifying both combinational and sequential logic.

Examples

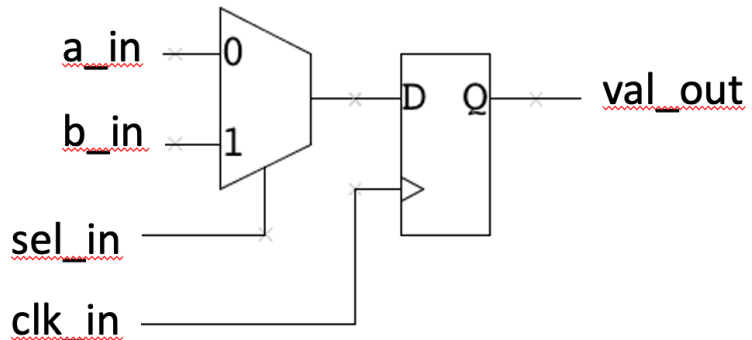
```
module blob(input wire a_in, b_in, sel_in,
            output logic val_out);
    always_comb begin
        if (sel_in)
            val_out = b_in;
        else
            val_out = a_in;
        end
    endmodule
```

Makes:



```
module blob(input wire a_in, b_in,
            sel_in, clk_in,
            output logic val_out);
    always_ff @(posedge clk_in) begin
        if (sel_in)
            val_out <= b_in;
        else
            val_out <= a_in;
        end
    endmodule
```

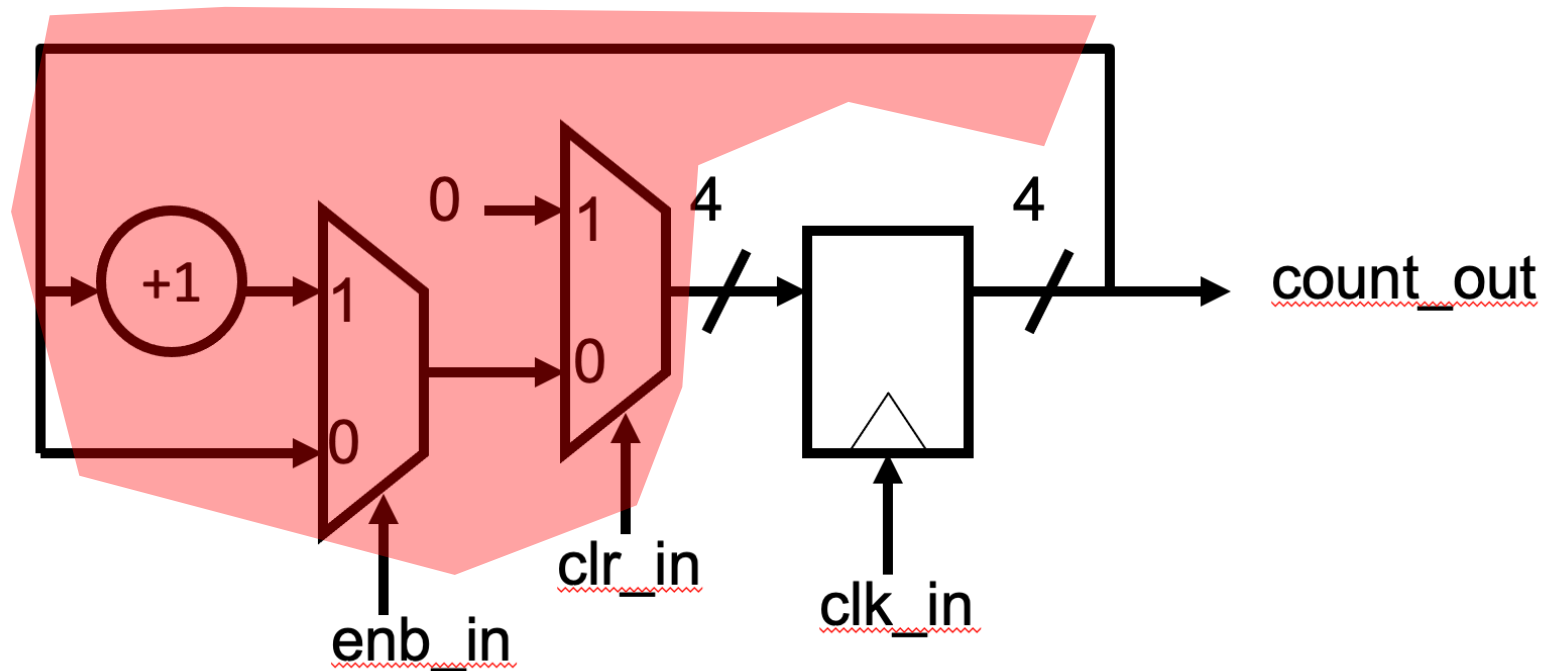
Makes:



Example: A Counter

```
// 4-bit counter with enable and synchronous clear
module counter(  input wire clk_in, enb_in, clr_in,
                 output logic [3:0] count_out);
  always_ff @(posedge clk_in) begin
    count_out <= clr_in ? 4'b0 : (enb_in ? count_out+1 : count_out);
  end
endmodule
```

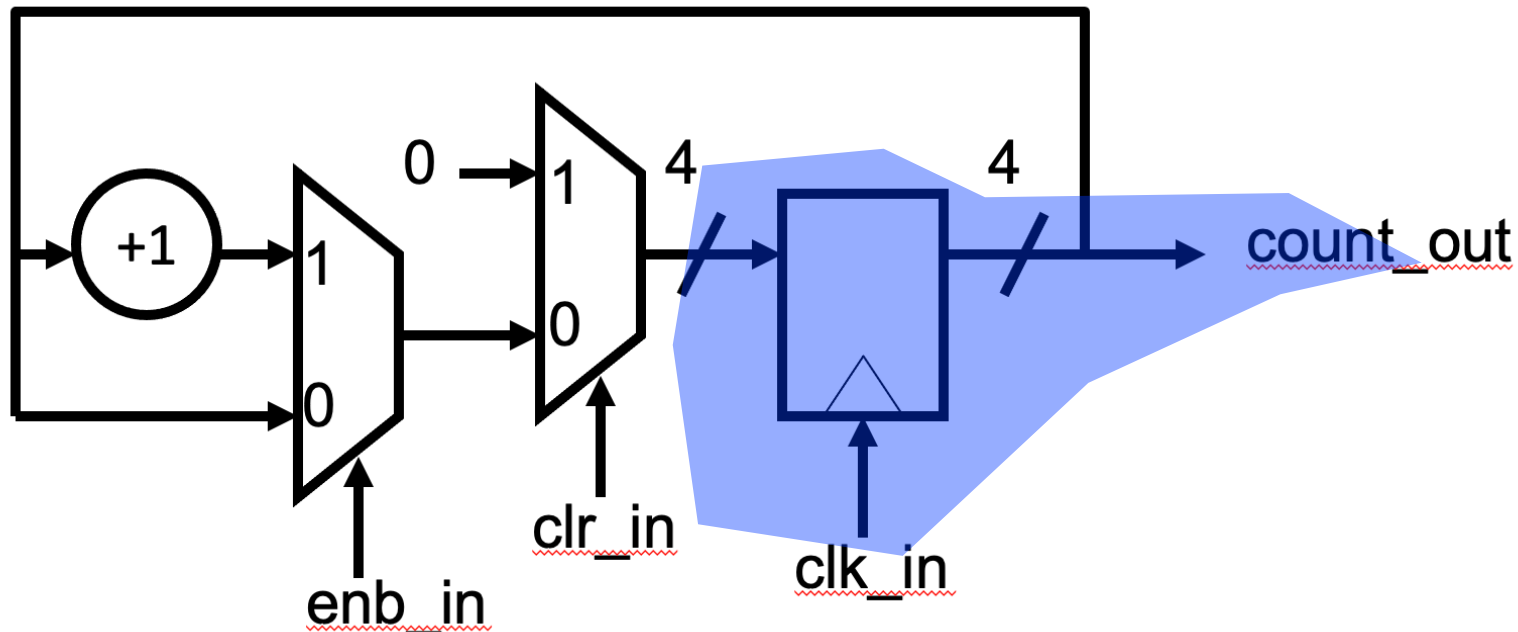
Combinational



Example: A Counter

```
// 4-bit counter with enable and synchronous clear
module counter(  input wire clk_in, enb_in, clr_in,
                 output logic [3:0] count_out);
  always_ff @(posedge clk_in) begin
    count_out <= clr_in ? 4'b0 : (enb_in ? count_out+1 : count_out);
  end
endmodule
```

Sequential



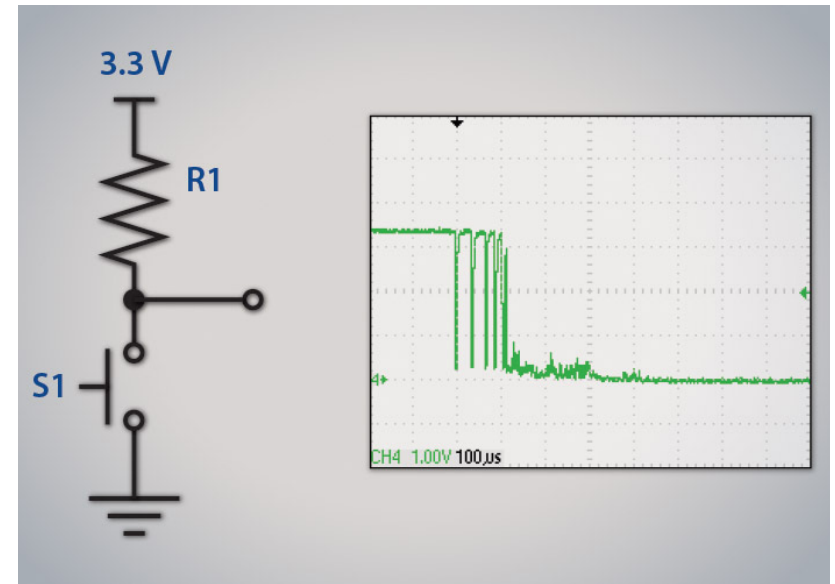
Week 2 Hint

- When it comes to the SPI module you need to write, don't approach it as a software engineer with only arrays and indexing/index variables and iterators. Yuck!
- Try to think about how shifting bits ($\gg$ or $\ll$) can get you what you want instead!

```
always_ff @(posedge clk_in) begin
    data_buffer <= {data_buffer[30:0], new_data};
    // OR...
    data_buffer <= (data_buffer<<1) | new_data;
end
```

Let's Build a Debouncer... Maybe

- Switches are mechanical devices. When they close and open, they can “bounce”
- Humans can't see, but electronics can since they are fast.
- A debouncer protects logic from these types of artifacts
- Only transfers its input to its output if it has been stable for a long time



<https://www.nuvation.com/resources/article/switch-debouncing-electronic-product-designs>