6.205
(aka 6.111)

Sequential Logic I

Fall 2024

Administrative

- Week 01 Due Last Night
- Week 02 Out After Class…due next Wednesday

The Third Way (Oboe)

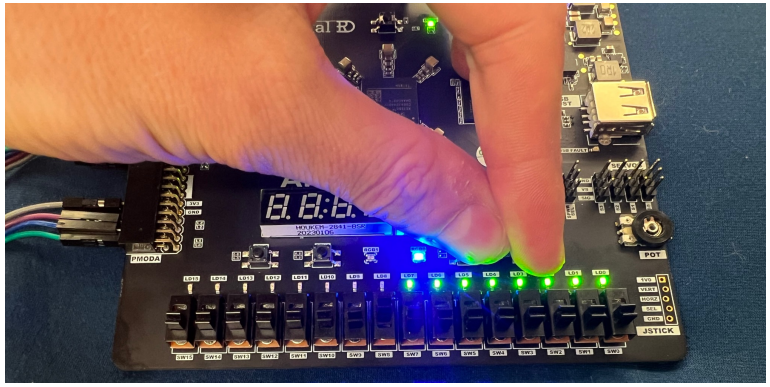
- In addition to GTKWave and Surfer, there's a MIT-based startup doing some EDA tool development that has offered to let us try/test out some of their tools.
- One that is relevant for 6.205 is **an online waveform viewer**.
- I've updated the documentation on the waveform page about this tool.
- Tool is called called Oboe:
- Site is here: https://digital.oboe.ai
- Google Docs instructions are here:
  https://docs.google.com/document/d/15R-8Jdouyi_TnCMoiqlFLgkRkacpWUujNlQElTLM0QI/edit

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 xxxxxxxxxxx as a result.
- If you tried that on your board then, you'd see something like:



"Huh. 0 divided by 0 is 255. Checks out."

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. ===

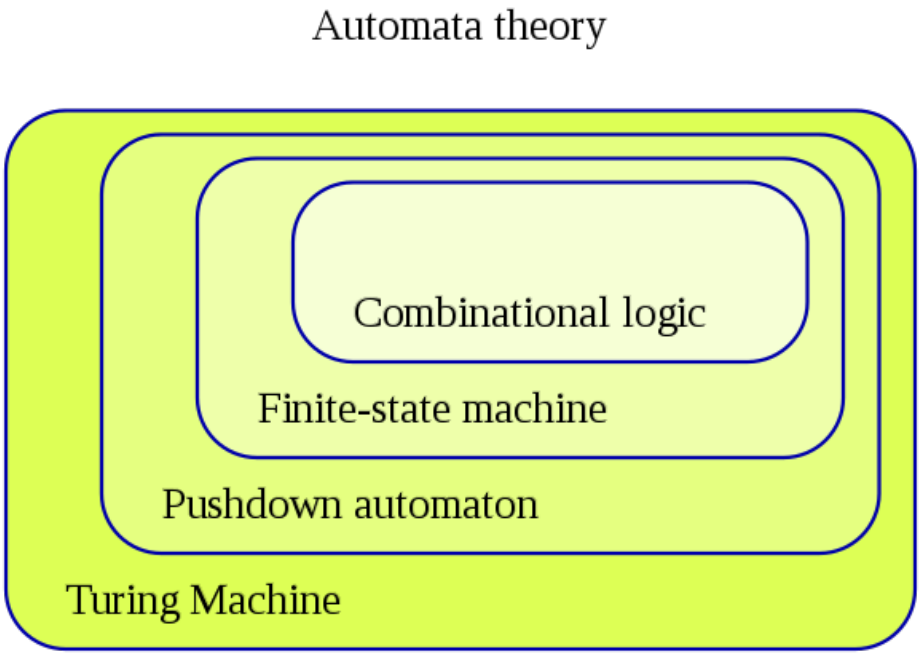
- In general equality checks in Verilog will pad 0s (or 1s) as needed (so you're not really doing checks on the size of the array)
- There's two types of equality checks in SystemVerilog:
  - == compares 1's to 0's only
  - === compares against all four types in an array:

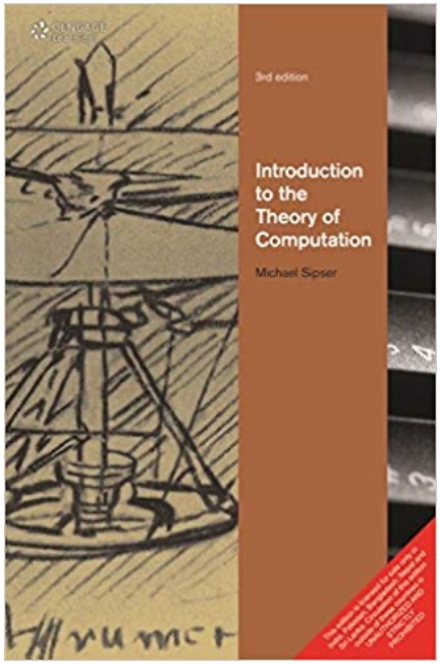
```
logic a,y;
initial begin
$display(5'b00001 == 8'b0000_00001); //eval to true
$display(5'b00001 === 8'b0000_00001); //eval to true
$display(5'b00001 == 8'b1000_0001); //eval to false!!
$display(5'b00001 === 8'b1000_0001); //eval to false!!
$display(a==y); //eval to undefined
$display(a===y); //eval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

Anywhoo....

- Moving on…

Levels of Complexity in Computation



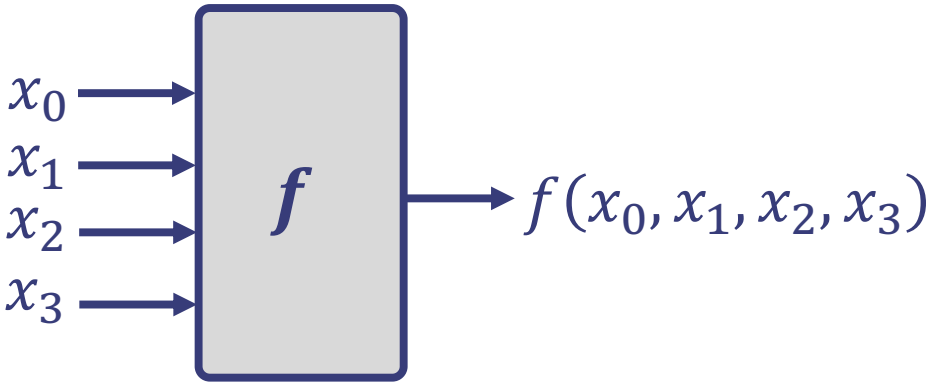


Sipser's Boo

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

Two Broad Types of Digital Logic

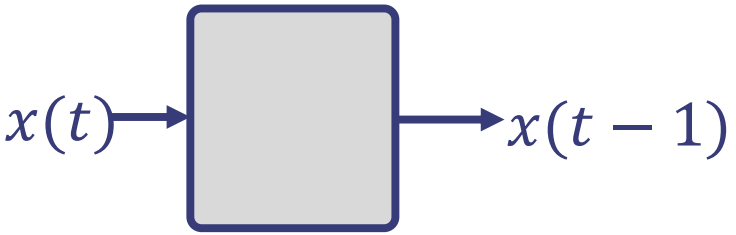
Functions:



Stateless

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

Storage:

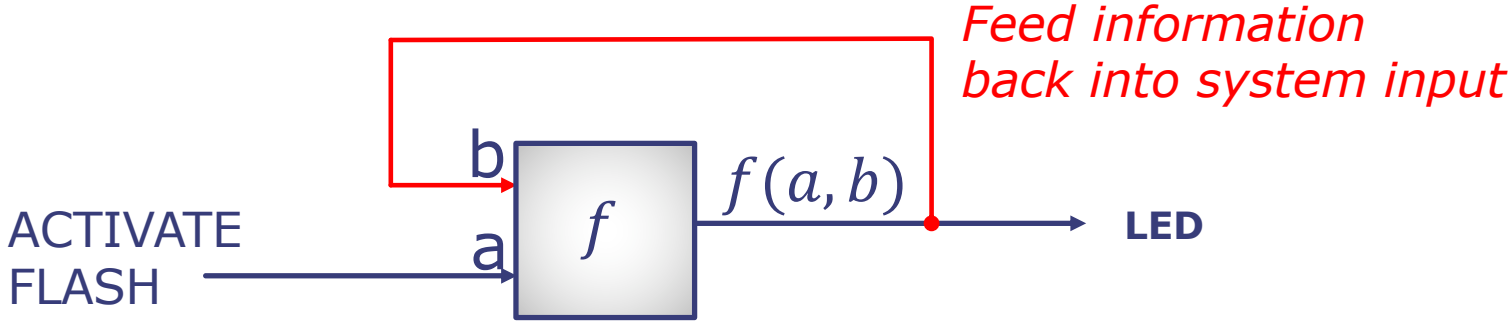


Stateful

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

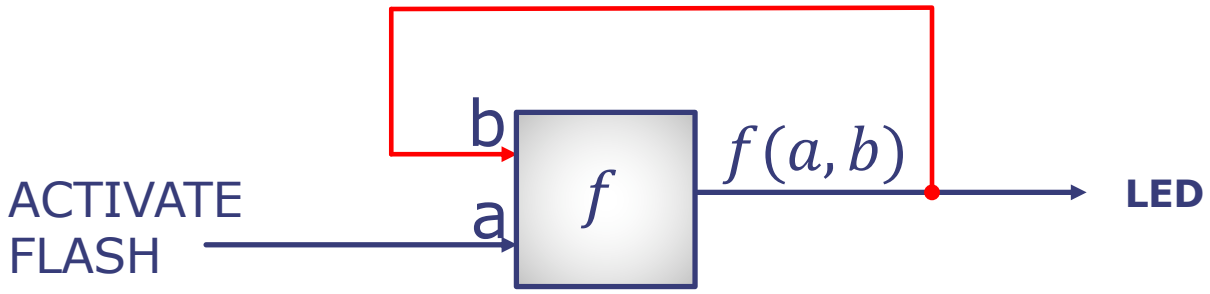
- One Attempt at Doing this:

```
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



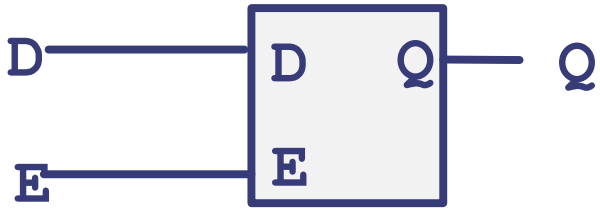
- a,b → f(a,b),a→ f(a,b),a→ f(a,b),a→ f(a,b),a→kaboom
- The assumptions we make about digital systems fall apart in this form

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

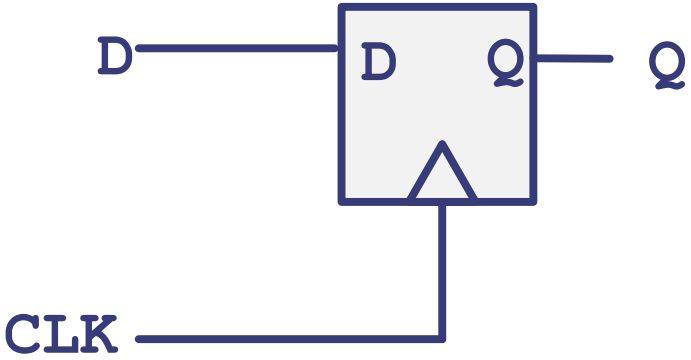
Level-Triggered Sample-and-Hold Device



"store D when E is high"

D Flip-Flop

Edge-Triggered Sample-and-Hold Device



"store D when clk rises"

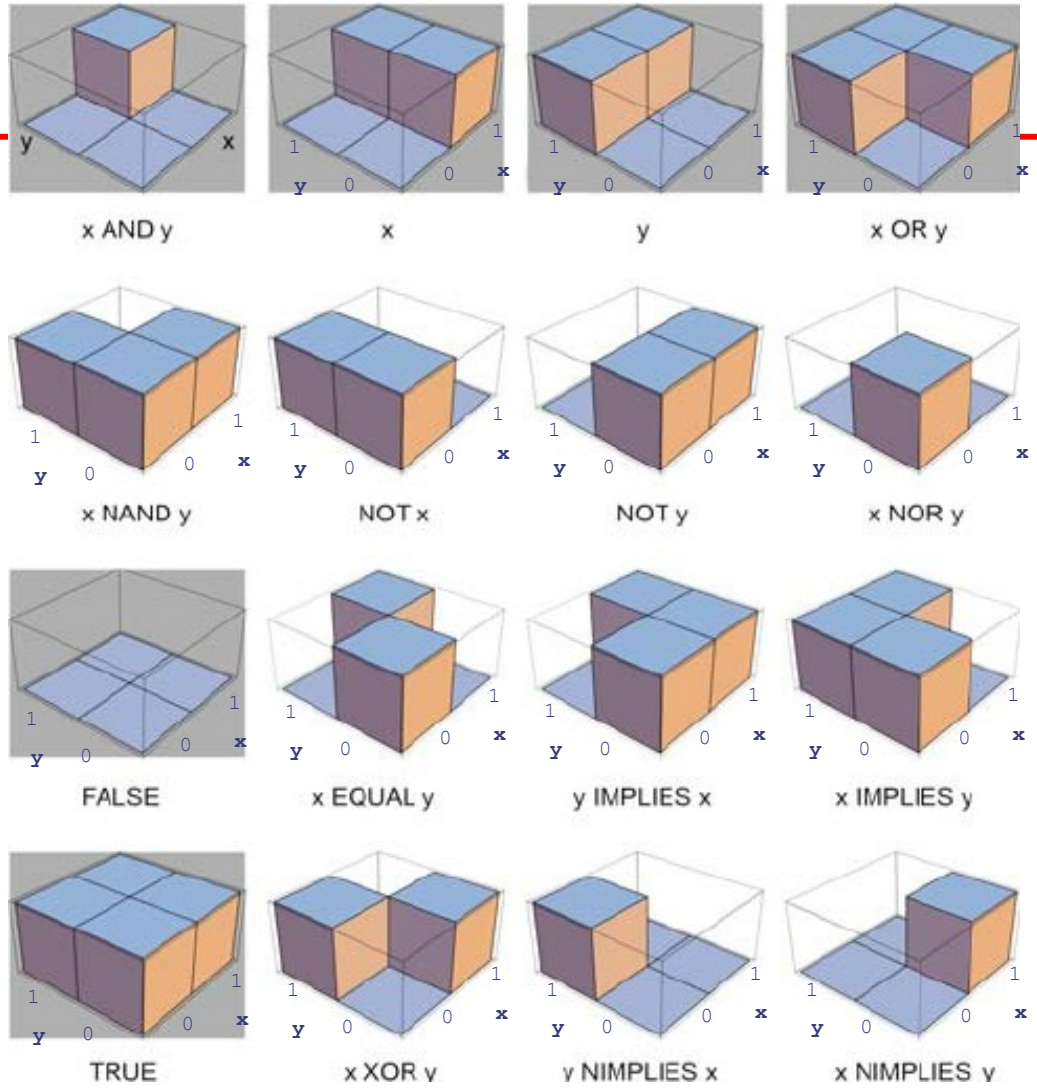
2-bit functions:

$f(x,y)$

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

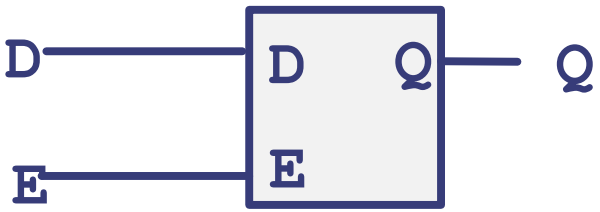
Stated another way: there are 16 unique 1-0 combinations for: $f(0,0)$, $f(0,1)$, $f(1,0)$, and $f(1,1)$



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/journ

D-Latch

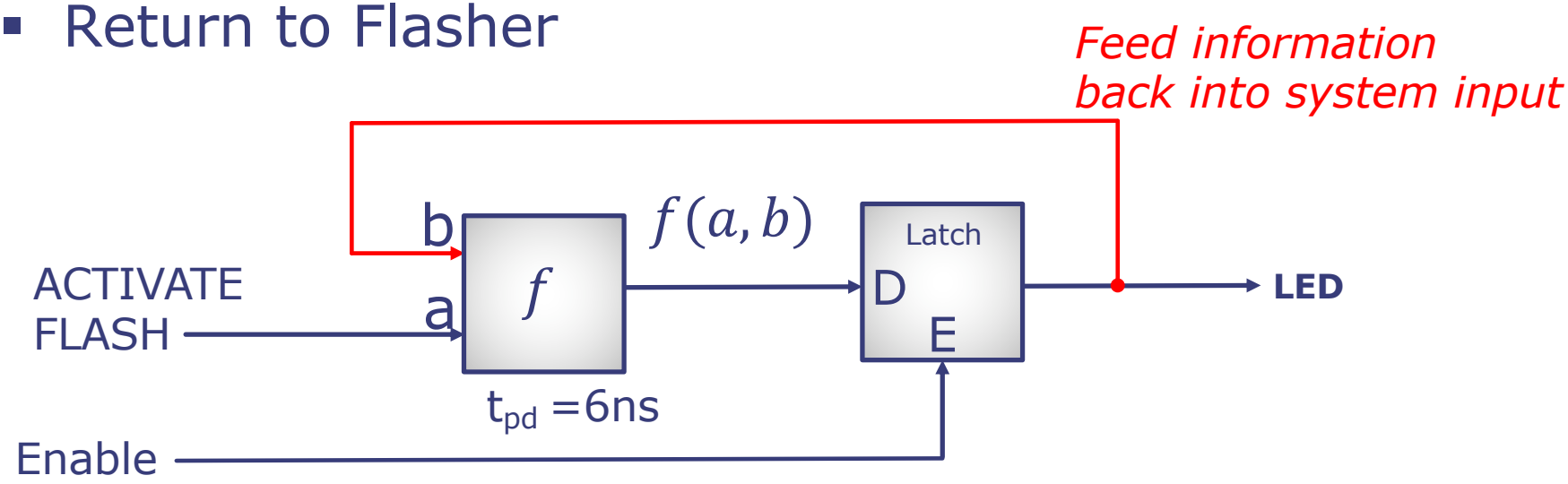
- A level-based sample-and-hodling device
- Turn on E, Whatever is at D, shows up at Q
- When E is on, the latch is "***transparent***"
- When E turns off, the last value that D had before turning off is locked in and held at Q output



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

Is Level Storing Good?

- Return to Flasher

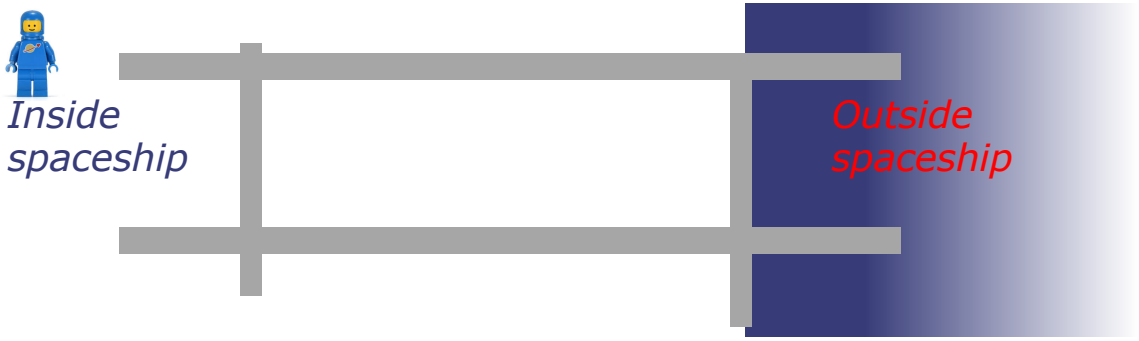


- The latch can store information, but when transparent (E = 1) the latch is allowing a combinational loop to exist:
  - a,b → f(a,b),a→ f(a,b),a→ f(a,b),a→ f(a,b),a→puke

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 latch can’t do that. It is like a door. You open a door and things and insects can flow in/out all crazy.
- We need something that will act as a digital air-lock or “sally port”

An Air Lock

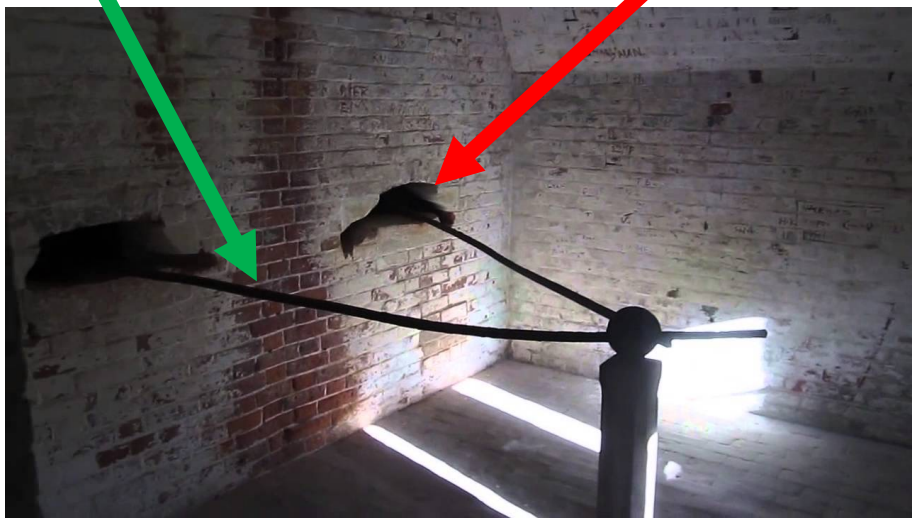


Open/close door to outside castle

Open/close door to inside castle

Never open both at same time!!!

- Prevent unregulated/ dangerous transfer of electrical signals forward (and also backwards too)

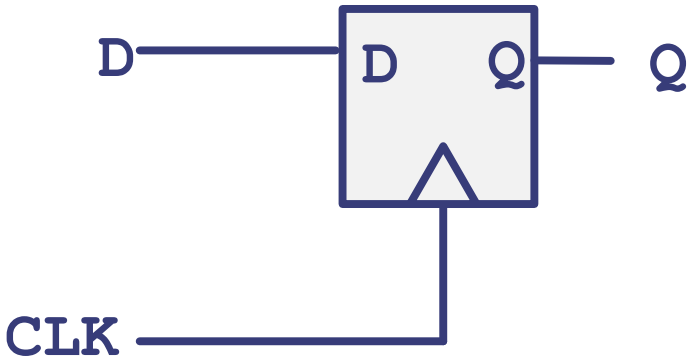


Sally Port Dover Castle, England

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



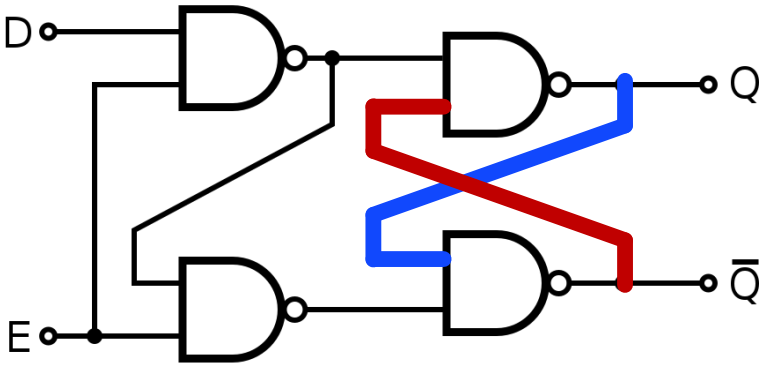
"store D when clk rises"

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 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

D = "Data"

E = "Enable"

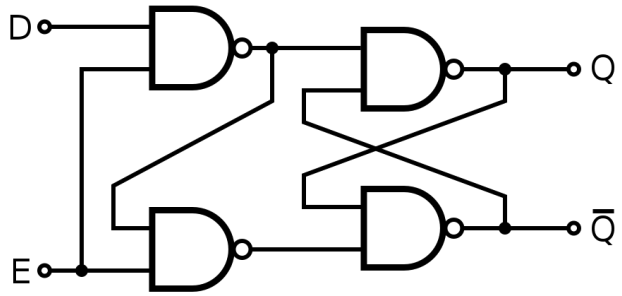
Q = not sure meaning, but it is the output

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

https://fpga.

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

D = "Data"

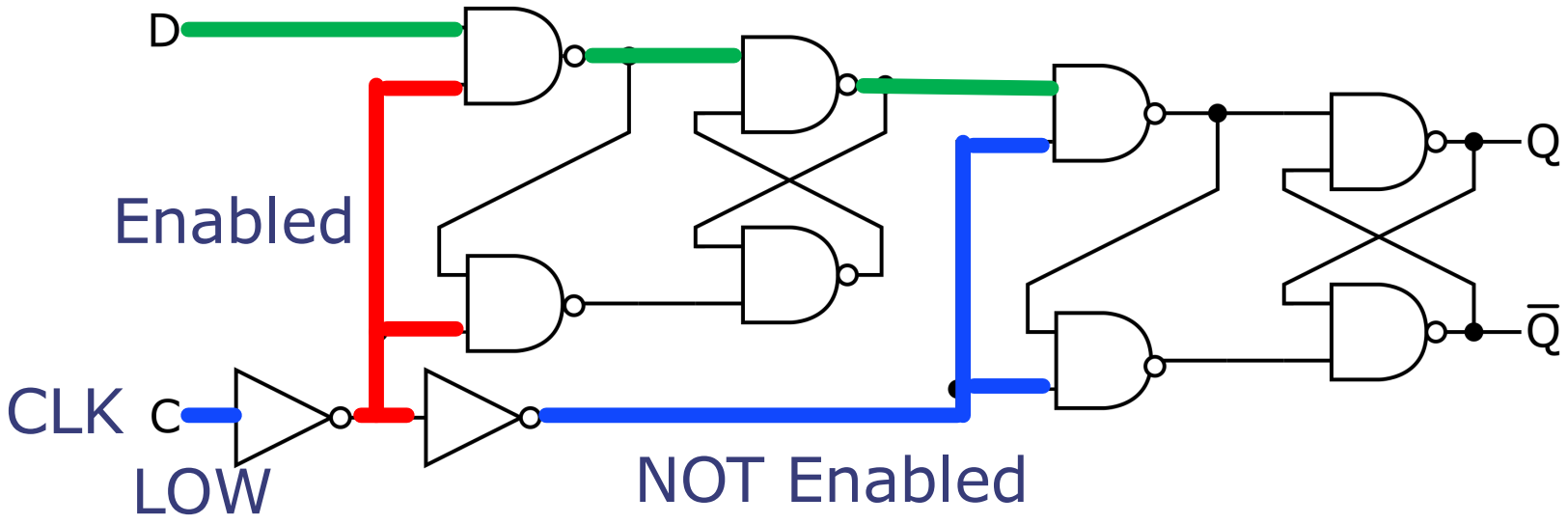
E = "Enable"

Q = not sure meaning, but it is the output

The D Flip-Flop (Reg)

Two D-Latches in Series driven with opposite enable signals

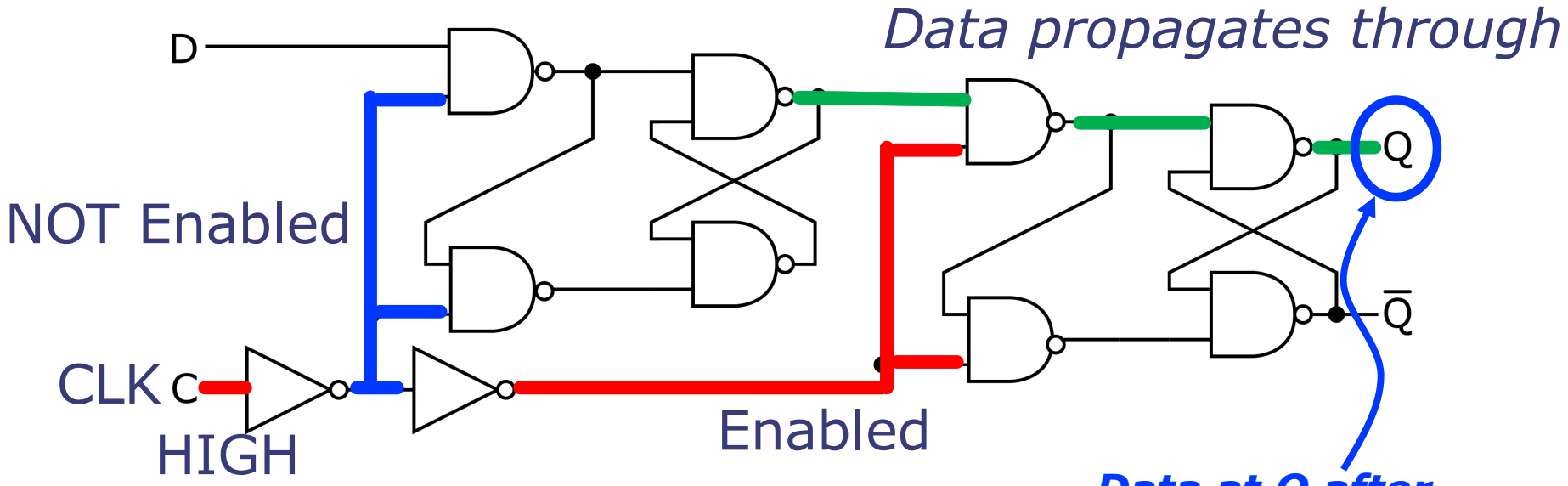
Data propagates through first D Latch



CLK LINE is LOW

The D Flip-Flop (Reg)

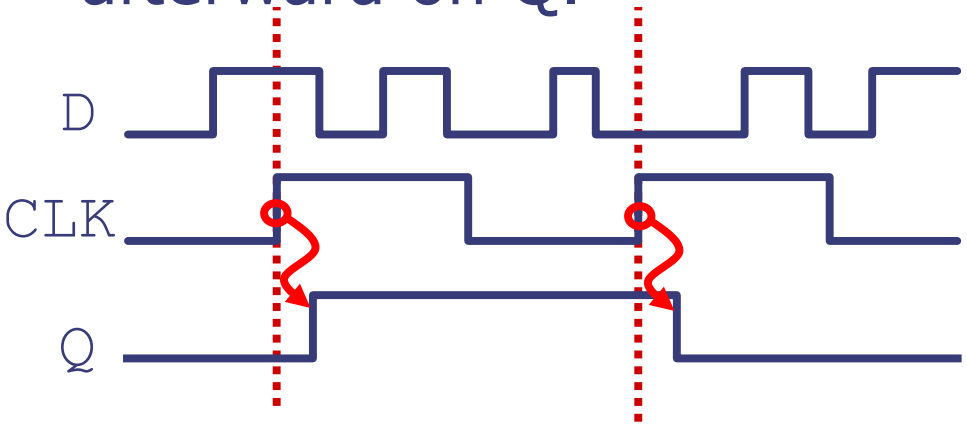
Two D-Latches in Series driven with opposite enable signals

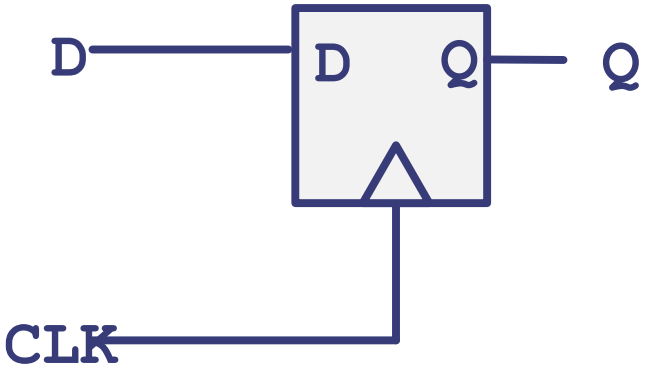


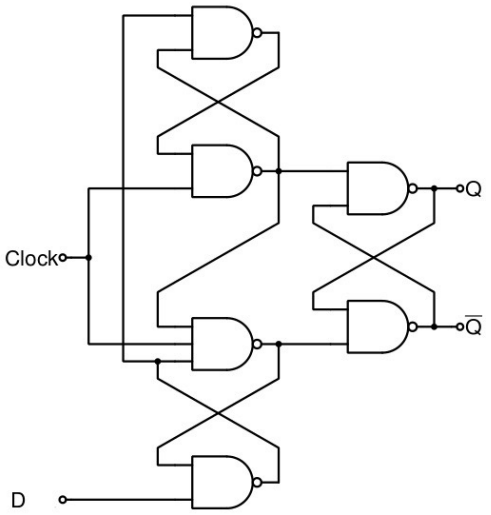
CLK LINE rises to HIGH

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

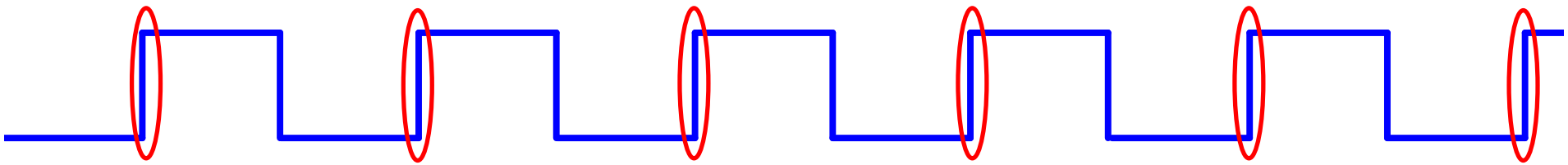
When you simplify some common/redundant logic between the two stages, you get to about ~25 transistors

A Clock Signal

- We 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

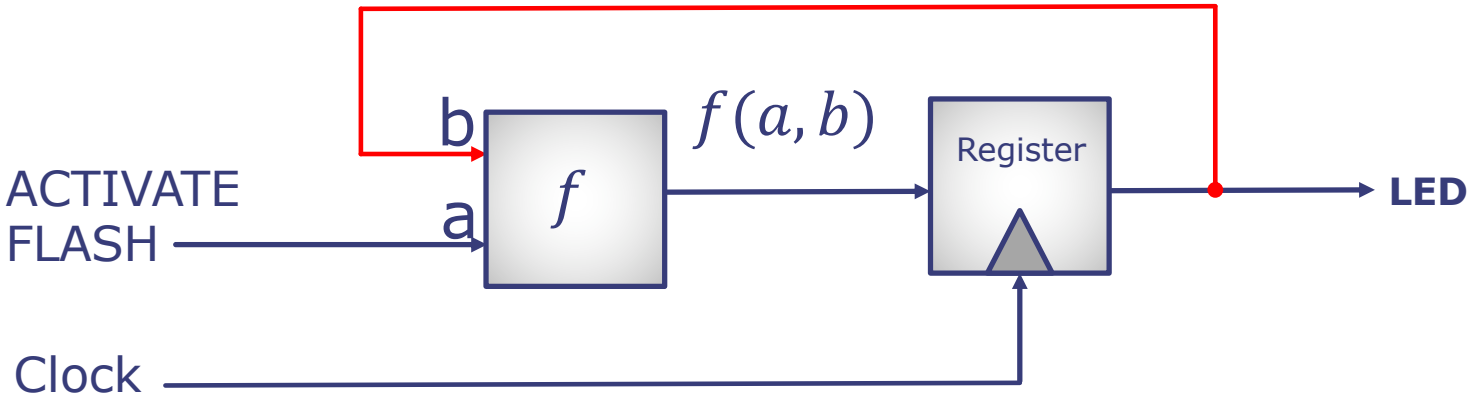
Clock signal:

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 writing Combinational Verilog modules (at least in lecture)*.
- How do we start to express these new ideas of sequential logic in Verilog?

*yes you used them a bit in week

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:

```
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

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 <=

- Blocking Assignments lend themselves to thinking about time
- `a <= 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 (and in fact can be confusing)
  - 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

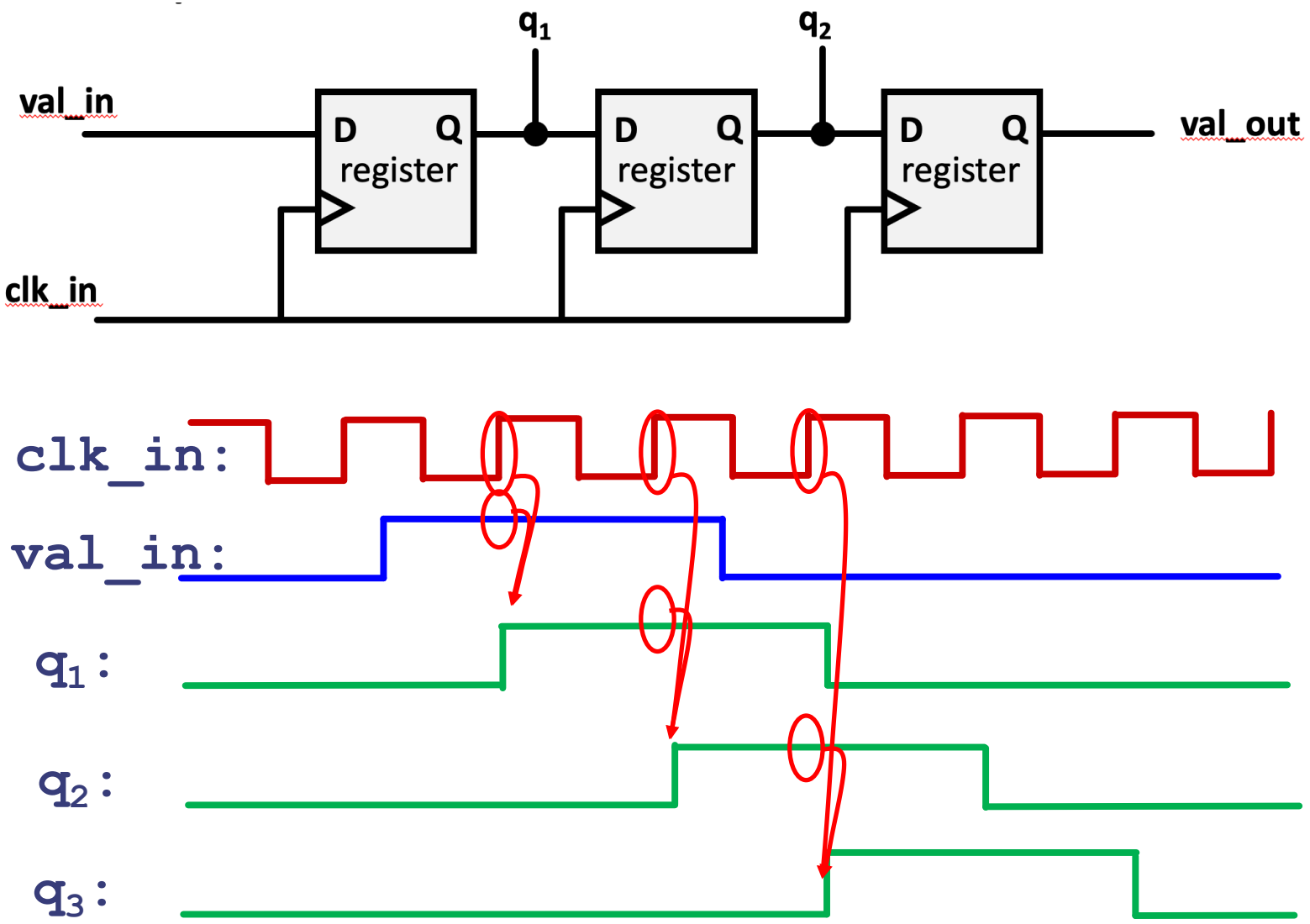
```
always_comb begin
  x = a | b; // 1. evaluate a|b, assign result to x
  y = a ^ b ^ c; // 2. evaluate a^b^c, assign result to y
  z = b & ~c; // 3. evaluate b&(~c), assign result to z
end
```

- ***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*)

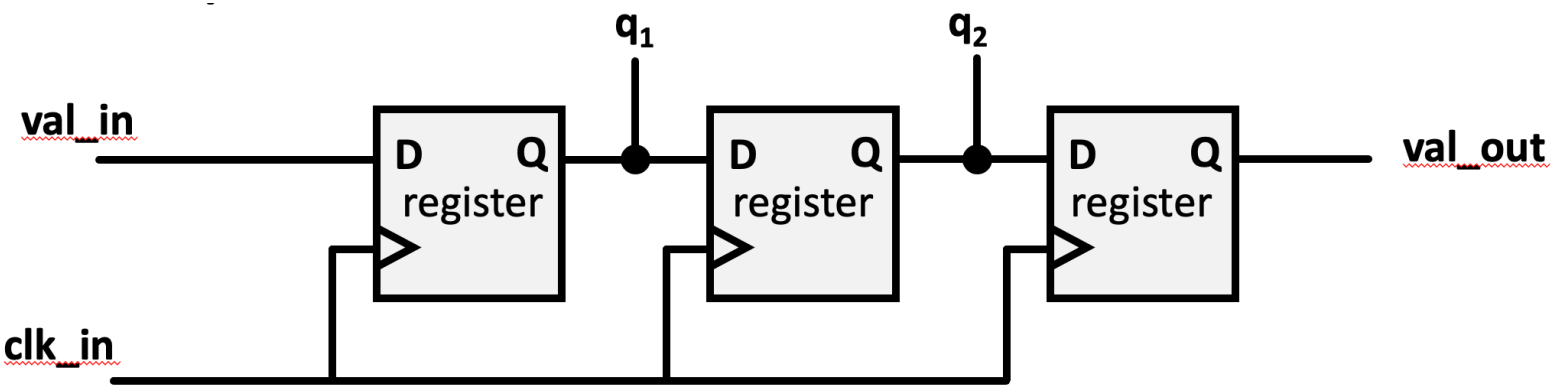
```
always_comb begin
  x <= a | b; // 1. evaluate a|b, but defer assignment to x
  y <= a ^ b ^ c; // 2. evaluate a^b^c, but defer assignment to y
  z <= b & ~c; // 3. evaluate b&(~c), but defer assignment to z
  // 4. end of time step: assign new values to x, y and z
end
```

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

Consider this Chained Flip Flop Circuit



Assignment Style for Sequential Logic

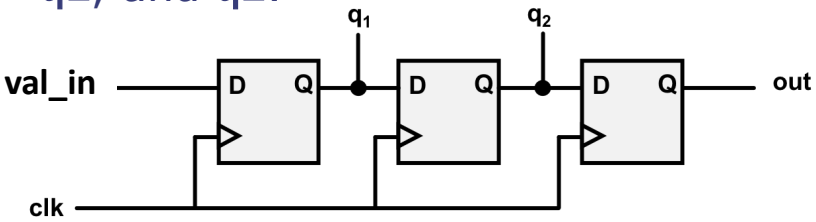


- Suppose we want to specify that circuit above in Verilog:
- Will nonblocking and blocking assignments both produce the desired result? ("old" means value **before** clock edge, "new" means the value **after** most recent assignment)

Use **Nonblocking** for Sequential Logic

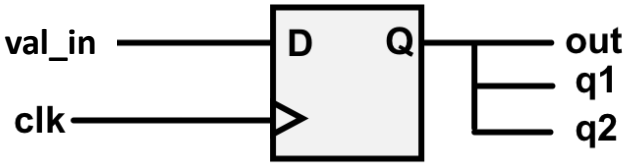
```
module nonblocking(
         input wire val_in, clk_in,
         output logic val_out
         );
  logic q1, q2;
  always_ff @(posedge clk_in) begin
    q1 <= val_in;
    q2 <= q1; // uses old q1
    val_out <= q2; // uses old q2
  end
endmodule
```

"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`)

- Avoid mixing 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

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.
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

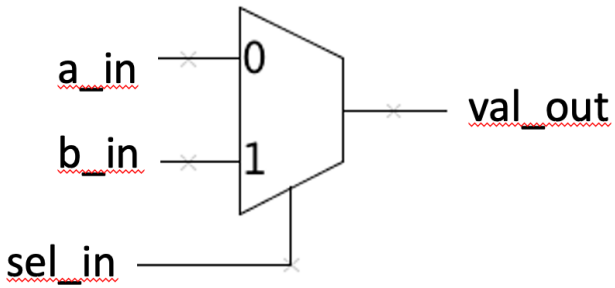
”Combinational vs Sequential”

- In combinational logic, you are making just combinational logic
- In sequential logic specification, you’re often specifying both combinational and sequential logic.

Examples

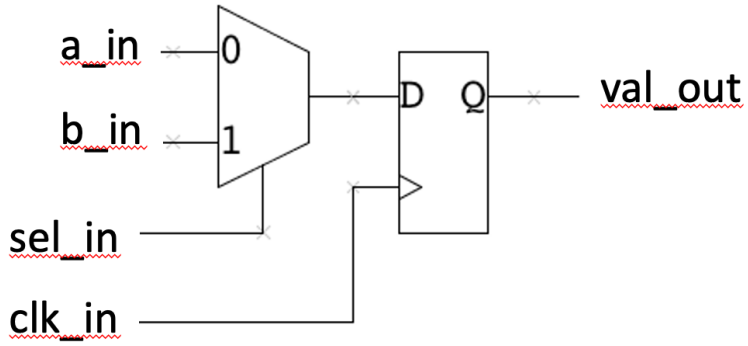
```
module blob(input wire a_in, b_in,sel_in,
            output logic val_out);
  always_comb begin
    if (sel_in)
      val_out = b_in;
    else
      val_out = a_in;
  end
endmodule
```

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
```

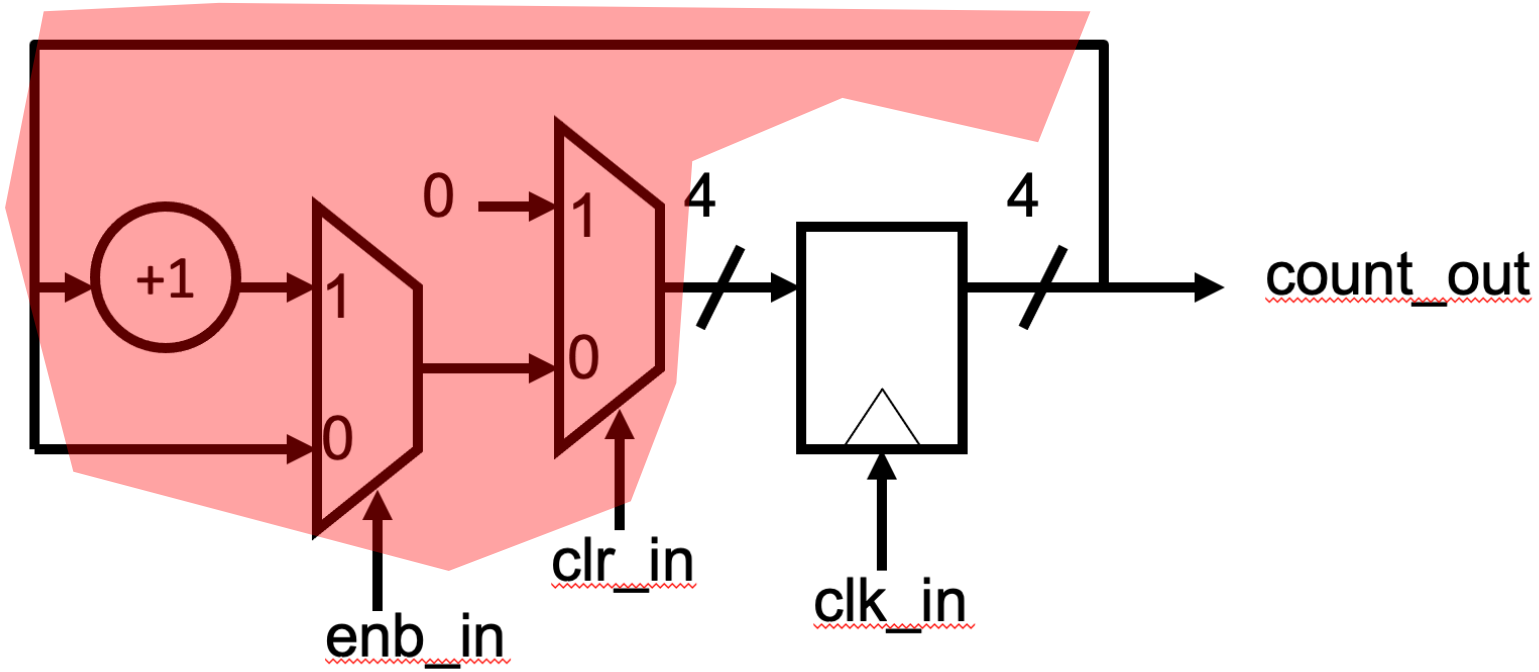
Makes:



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
```

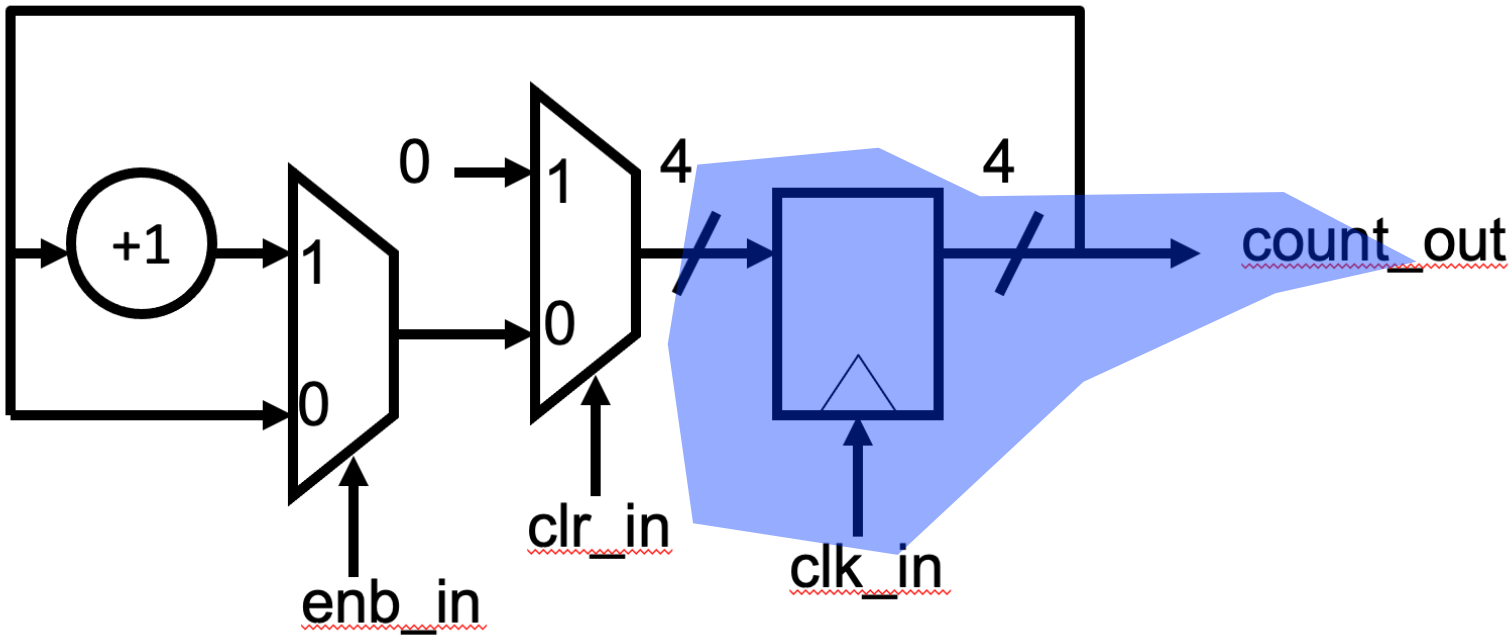
Combinational



Example: A Counter

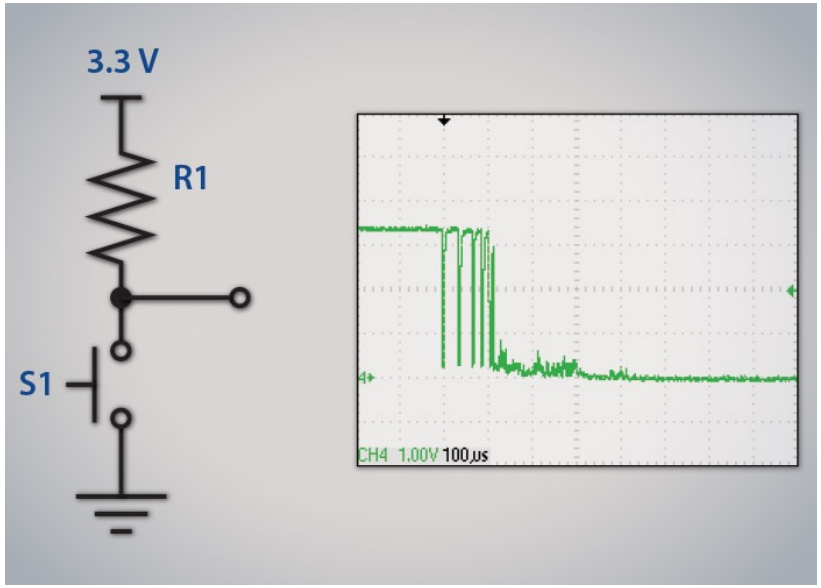
```
// 4-bit counter with enable and synchronous clear
module counter(   input wire clk_in, enb_in, clr_in,
                  output logic [3:0] count_out);
  always_ff @(posedge clk_in) begin
    count_out <= clr_in ? 4'b0 : (enb_in ? count_out+1 : count_out);
  end
endmodule
```

Sequential



Let's Build a Debouncer

- 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

Bouncing Switch

- Let us build and test a debouncer together…