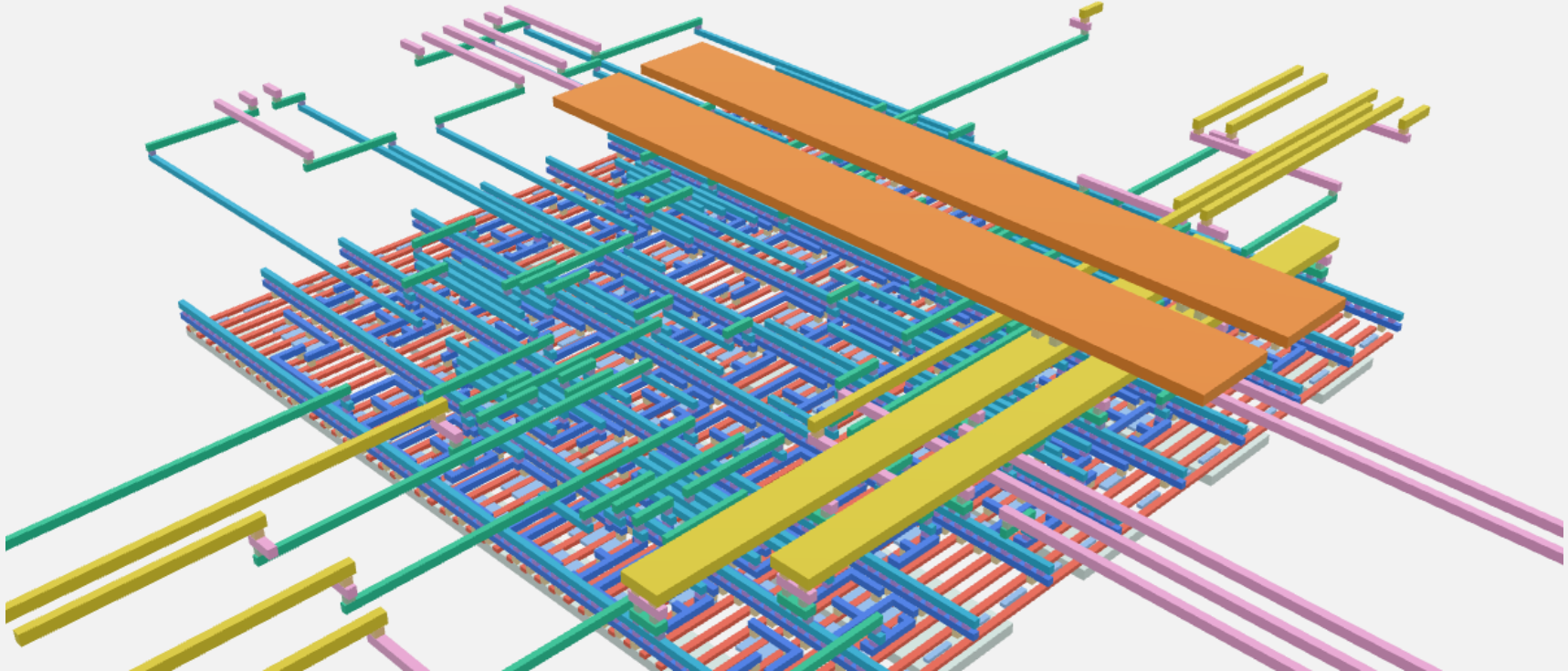
3.2
CSE140 – Digital Design

2026 Summer Session 1

Abarajithan G



Daily reminder:

- Open FeedbackFruits
  Canvas → CSE140 → Assignments → Slides 3.2

- Attendance is mandatory

- Midterm exam:
  - 45 minutes
  - 26 MCQs (1 bonus)
  - You need to know every line of your PA submission
  - No questions during exam – if you think Q is wrong, write in the back
  - Closed book, no phones, no calculators, one cheat sheet

Previous Feedback

I liked how we walked thorugh the ALU questions. I was a little confused at first but the slow walkthrough was helpful

Learned things that I kinda knew about but not at this level so it was very interesting

I like that how we went through the code today.

explaining by drawing on the black board

Better explain the SV code, because sometimes I get stuck trying to read and understand the SV code

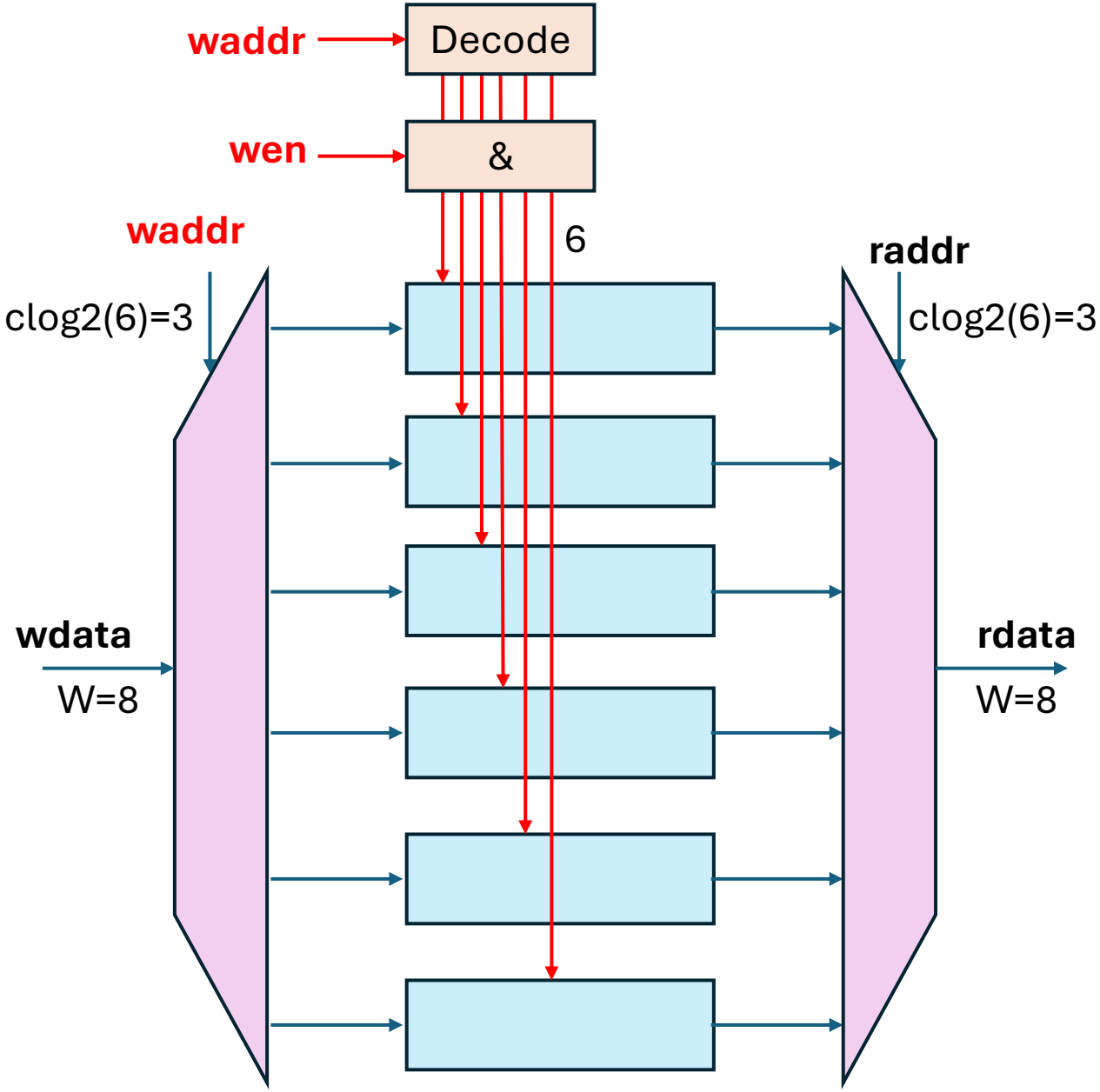
it would probably have been better to teach verilog by first going into the basics, instead of just diving straight into lots of examples and using them to teach the basics.

Register File Example

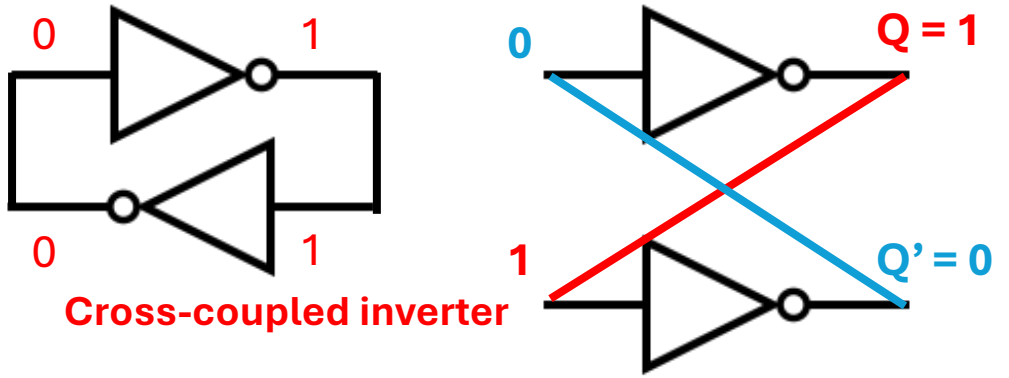
- N registers, each W bits wide
- Read/write address: clog2(N) bits
- wen=1: regs[waddr] <= wdata
- rdata = regs[raddr]

```
module register_file #(
  parameter int  W_DATA = 8, N_REGS = 6,
  localparam int W_ADDR = $clog2(N_REGS)
)(
  input  logic clk, rstn, wen
  input  logic [W_ADDR-1:0] raddr,
  input  logic [W_ADDR-1:0] waddr,
  input  logic [W_DATA-1:0] wdata,
  output logic [W_DATA-1:0] rdata
);
  logic [15:0] regs [N_REGS];

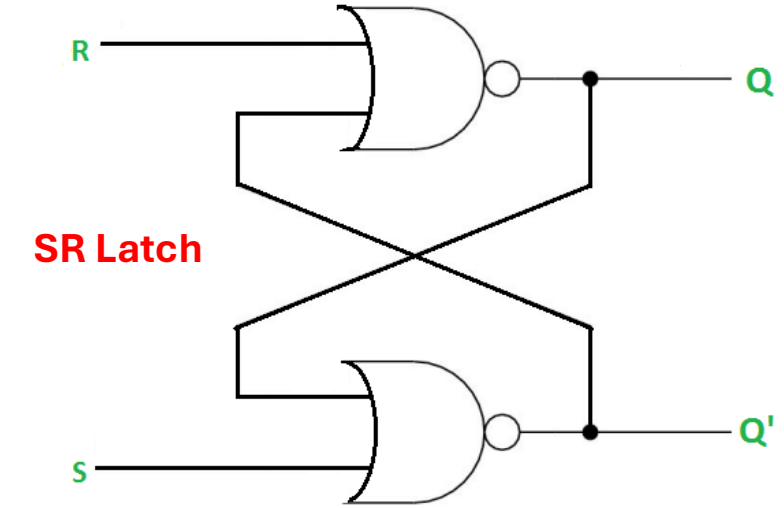
  always_ff @(posedge clk)
    if(!rstn) begin
      for (int i=0; i<N_REGS; i++) regs[i] <= '0;
    end else begin
      if (wen) regs[waddr] <= wdata;
    end
  always_comb rdata = regs[raddr];
endmodule
```

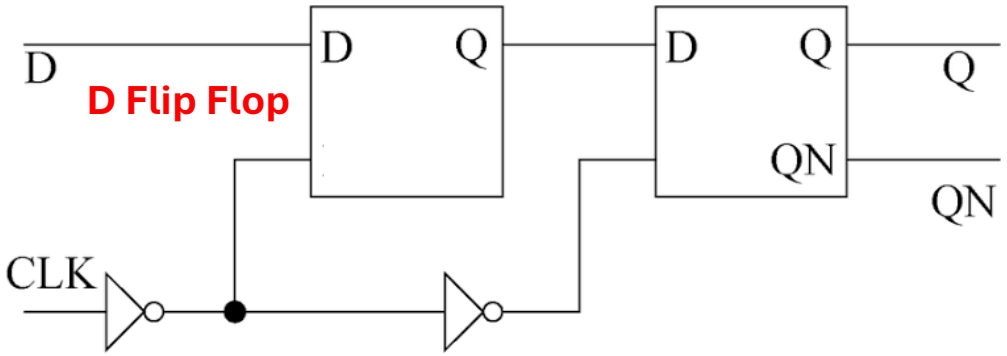


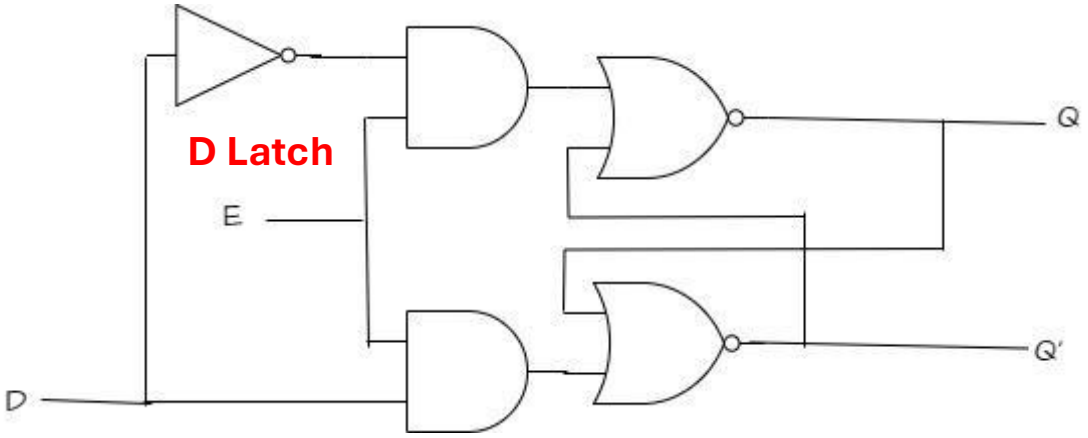
Storing Data



- How to store a bit – two NOT gates reinforcing each other
- But no inputs? – Add set & reset inputs! SR Latch!
- S=R=0 latches, but S=R=1 causes next Q to be non-deterministic
- Make S=D & R=!D - But can't latch (no S=R=0) – Add E
- D-Latch - E=1: transparent, E=0: store data (opaque)
- But we want to capture data at an edge
- D Flop – Two D latches, each store data in each phase (0,1) of a clock cycle. Data captured at every clock edge and held for a clock cycle







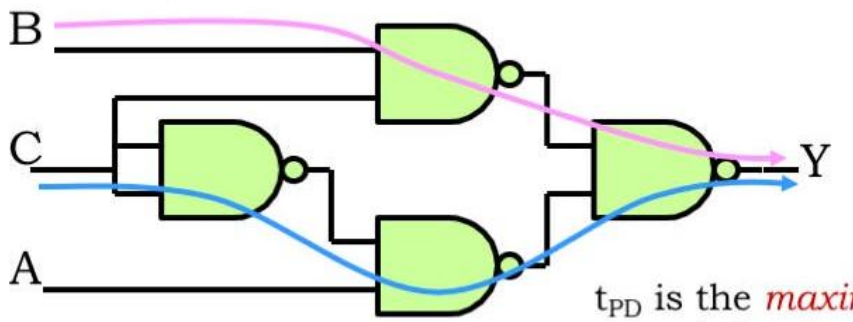
Find tCD and tPD

If NAND gates have a t_{PD} = 4nS and t_{CD} = 1nS

t_{PD} = 12 nS

t_{CD} = 2 nS

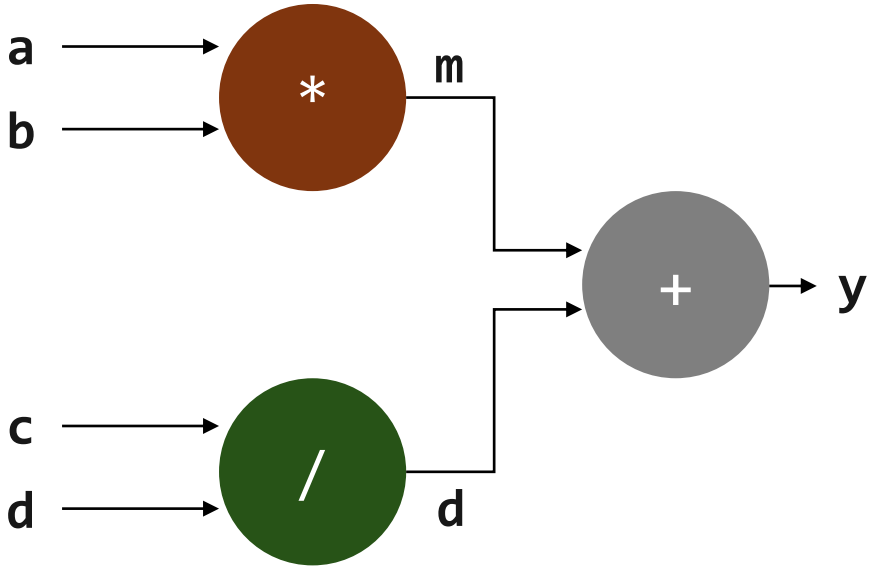
t_{CD} is the *minimum* cumulative contamination delay over all paths from inputs to outputs



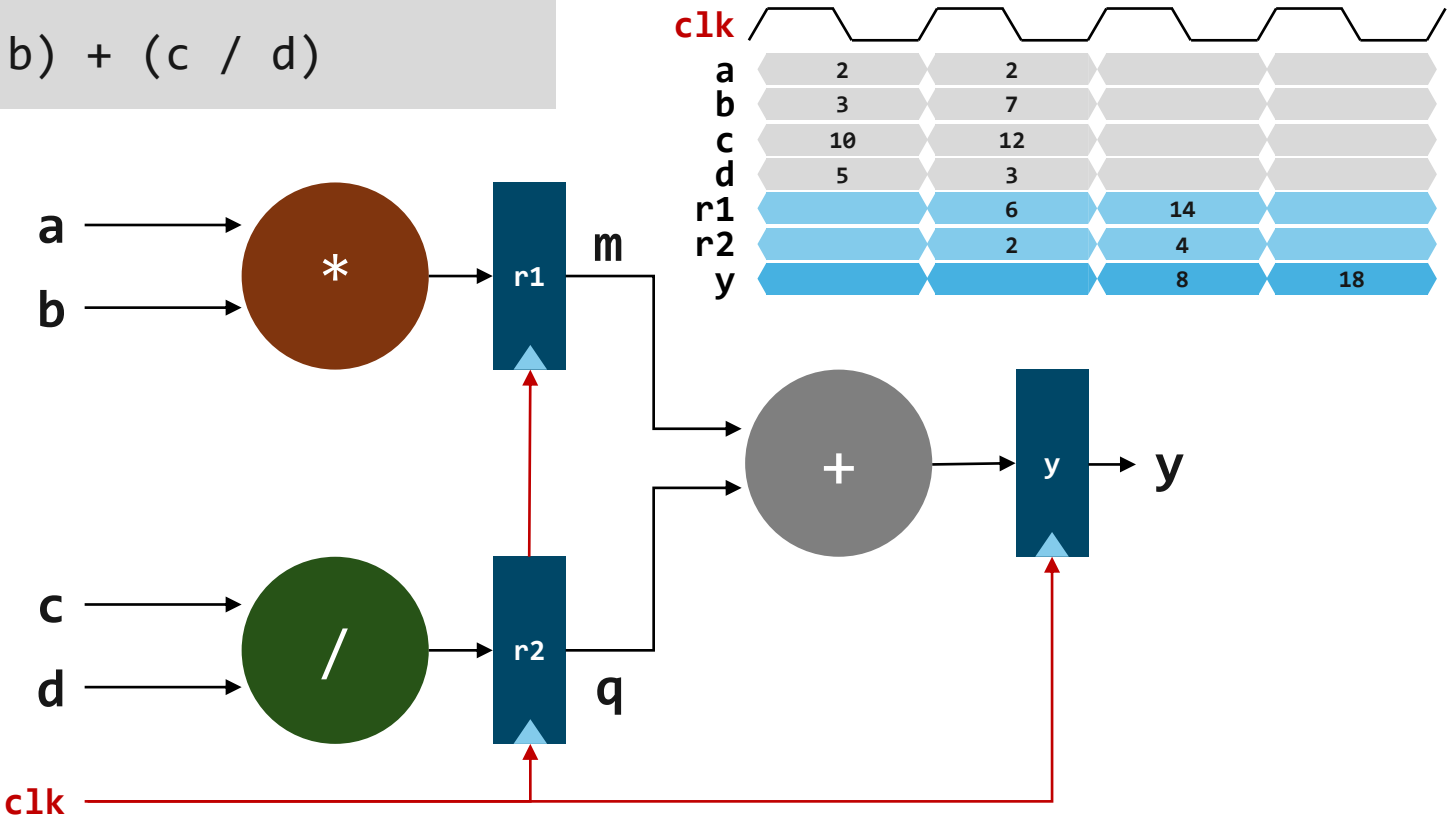
t_{PD} is the *maximum* cumulative propagation delay over all paths from inputs to outputs

Why do we need clock?

```
y = (a * b) + (c / d)
```

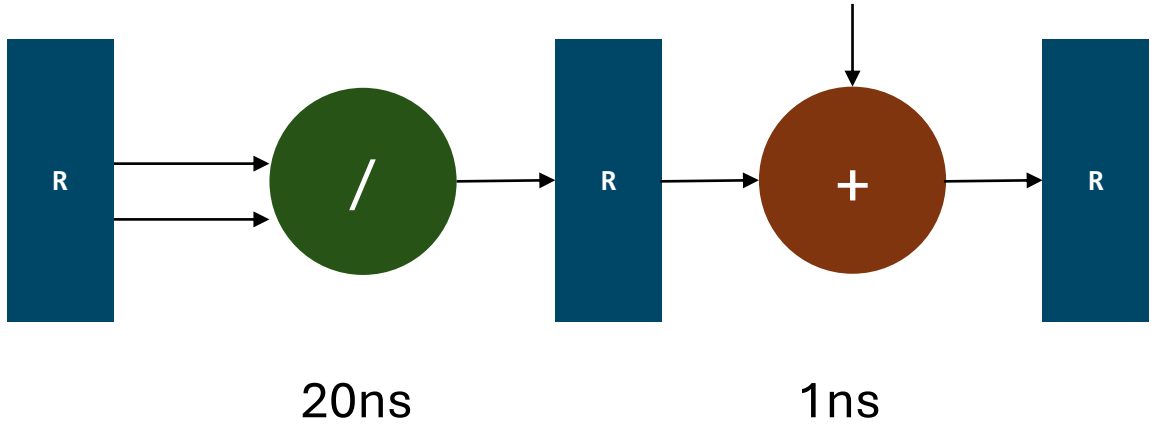


- Signals take time to run through wires
- Different delays in multiplier and divider
- m, d stabilize at different times
- We are not sure:
  - when to read y
  - when to change a,b,c,d
- Difficult to design anything complex

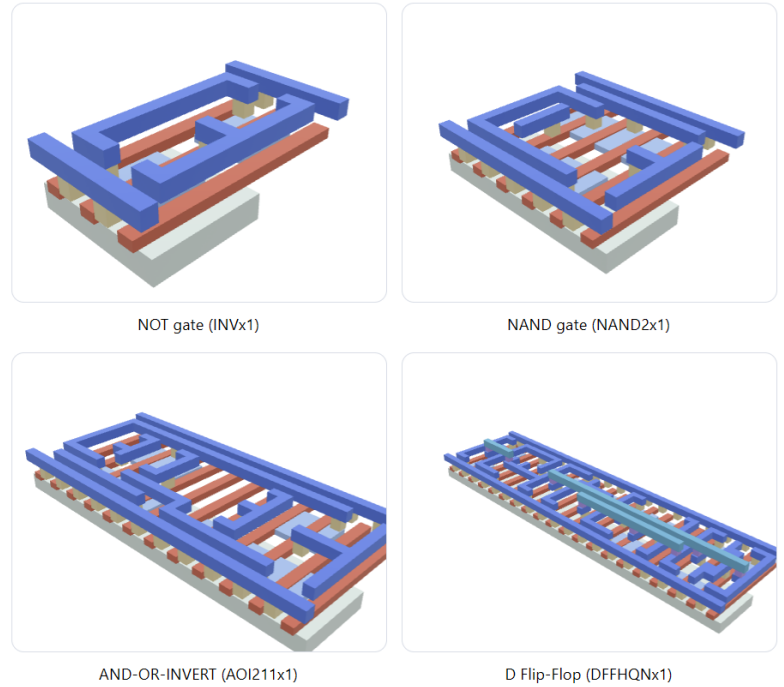


- Clock is a drum beat, that “synchronizes” other signals
- Registers, made of flip flips are sequential. They capture data at an **edge of clock signal** and hold it unchanged through the cycle.
- Synthesis, P&R software will try, such that all signals take less than a clock period (which we set) to go from one register to another.
- Each signal should stabilize before next posedge, to be accurately read.

Timing closure



- Critical path: 20ns
- Ignoring other irregularities:
- Maximum clock frequency: $1/(20 \times 10^{-9})$s = 50 MHz
- Even if all other paths are 1ns (=1 GHz)
- Trade-off:
  - Make sure the combinational circuits between two registers/input/output aren't too big
  - Should not register too often – flip flops are big!

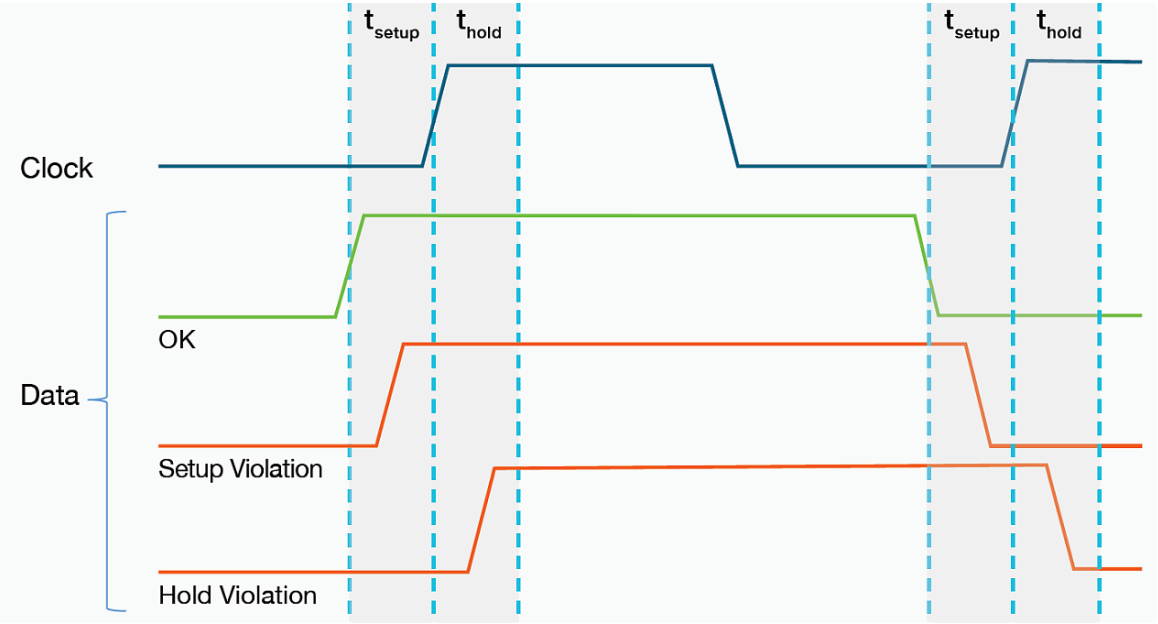


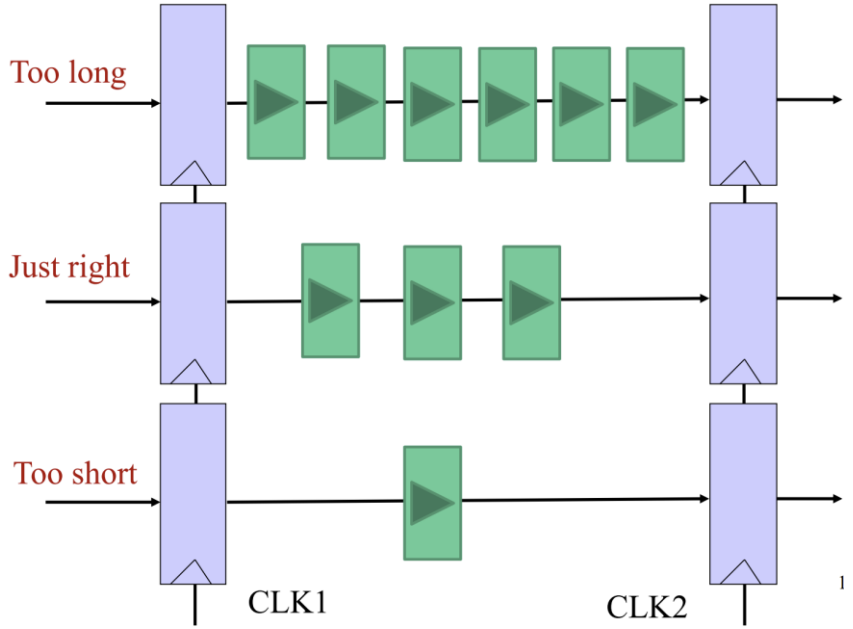
Clock-to-Q Time

- Flip flops are also not ideal
- Tpcq – clock-to-q propagation delay of a flip flop (not Tpd)
  - Upper bound on the time after which FF's output data will be stable
- Tccq – clock-to-q contamination delay of a flip flop (not Tcd)
  - Minimum time after which FF's output may begin changing

Setup and Hold Time

- Flip flops don't exactly sample at the edge
- The data needs to be stable/unchanged over a period of time around the clock edge
- Setup time (before the clock edge)
  - Violation – too much combinational logic before FF
- Hold time (after the clock edge)
  - Violation – too little combinational logic before FF
- Synthesis/PNR tools try to move things around, add buffers…etc. to fix these, as much as they can.
- Beyond that, we must change the RTL.





https://jlcpcb.com/blog/digital-circuit-timing-basics

cseweb.ucsd.edu/classes/wi16/cse140-a/slides/lec10timing.pdf

Setup Time Constraint

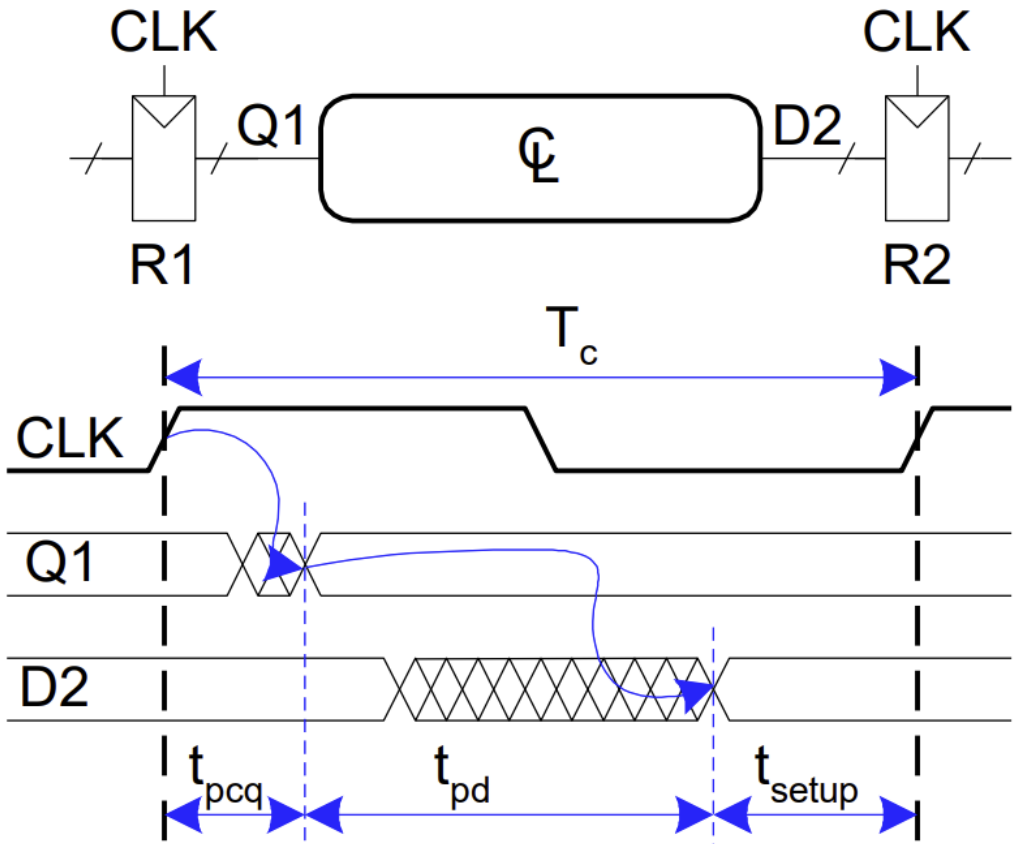
Setup slack =
Required time – Arrival time

Negative is bad

$$T \geq t_{pcq} + t_{pd} + t_{setup}$$

PI Q: As a designer, which of the following parameters would you modify to meet the set up time constraint?

A. The clock period, T
B. The prop. delay of R1, t_{pcq}
C. The prop. delay of CL, t_{pd}
D. The setup time of R2, t_{setup}
E. All of the above



cseweb.ucsd.edu/classes/wi16/cse140-a/slides/lec10timing.pdf

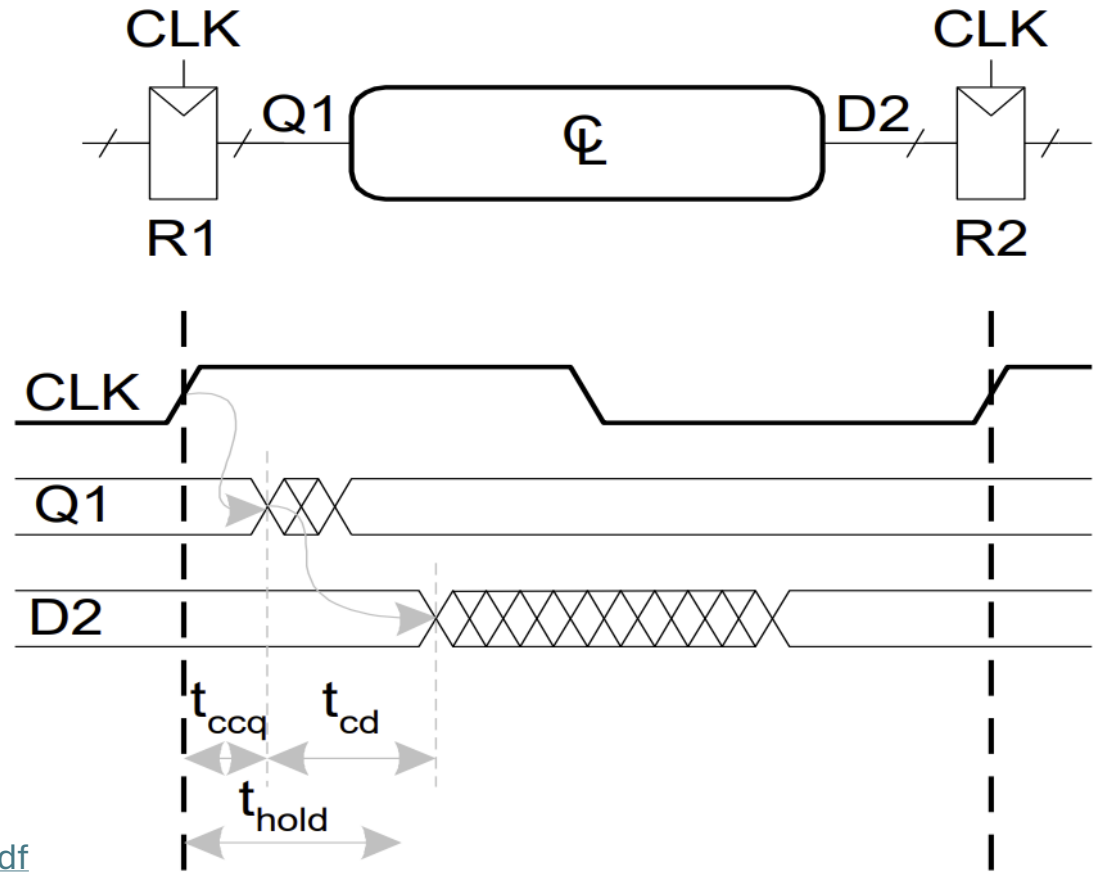
Hold Time Constraint

Hold slack =
Arrival time - Required time

Negative is bad

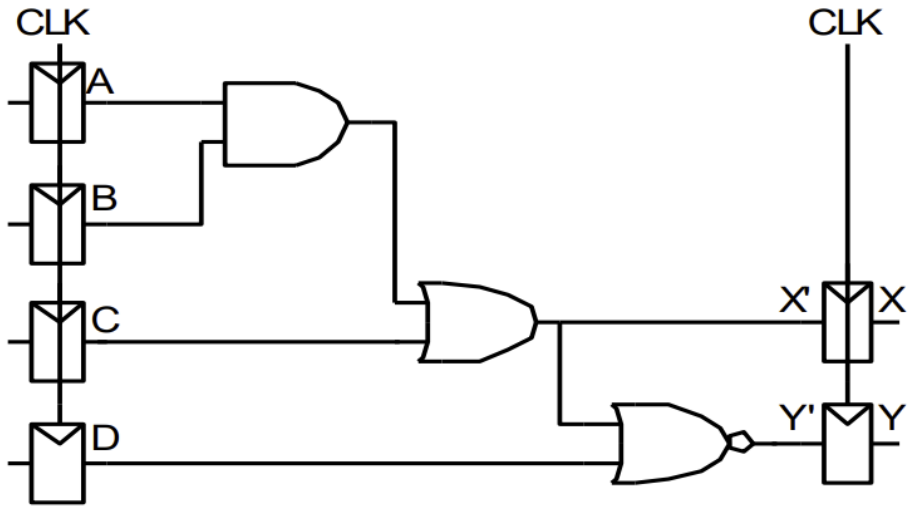
$$t_{\text{hold}} < t_{ccq} + t_{cd}$$

$$t_{cd} > t_{\text{hold}} - t_{ccq}$$



cseweb.ucsd.edu/classes/wi16/cse140-a/slides/lec10timing.pdf

Timing Analysis: Example



Timing Characteristics

FFs t_{ccq} = 30 ps

t_{pcq} = 50 ps

t_{setup} = 60 ps

t_{hold} = 70 ps

Gates $t_{pd}(g)$ = 35 ps

t_{cd}(g)= 25 ps

t_{pd}(com) =

t_{cd}(com) =

Setup time constraint:

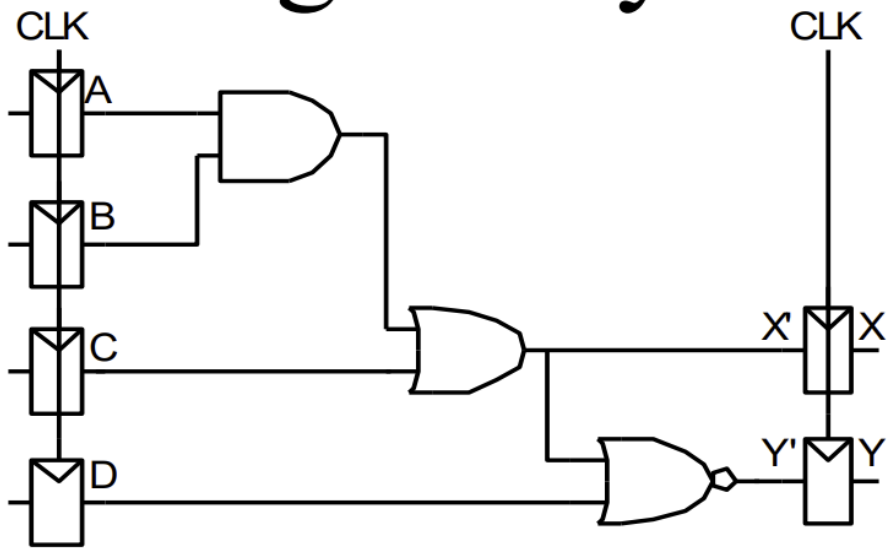
$T_c \geq$

$f_c = 1/T_c =$

Hold time constraint:

$t_{ccq} + t_{cd}(com) > t_{hold}$?

Timing Analysis: Example



FFs t_{ccq} = 30 ps

t_{pcq} = 50 ps

t_{setup} = 60 ps

t_{hold} = 70 ps

Gates t_{pd}(g)= 35 ps

t_{cd}(g)= 25 ps

t_{pd}(com)= 3 x 35 ps = 105 ps

t_{cd}(com)= 25 ps

Setup time constraint:

$T \geq t_{pcq} + t_{pd}(\text{com}) + t_{\text{setup}}$

=50 + 105 + 60 = 215 ps

$f_c = 1/T_c$ = 4.65 GHz

Hold time constraint:

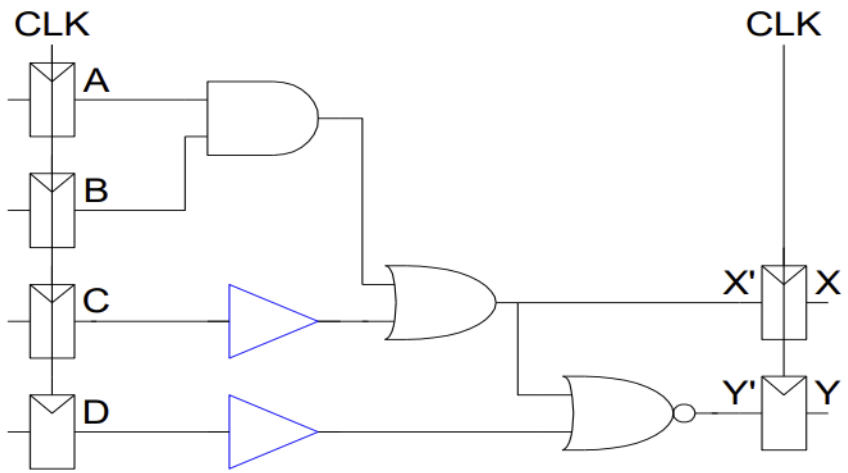
$t_{\text{ccq}} + t_{cd}(\text{com}) > t_{\text{hold}}$?

(30 + 25) ps > 70 ps ?

No!

Example: Fix Hold Time Violation

Add buffers to the short paths:



t_{pd}(com)= 3 x 35 = 105 ps

t_{cd}(com)= 2 x 25 = 50 ps

Setup time constraint:

$T \geq 50 + 105 + 60 = 215$ ps

$f_c = 1/T_c = 4.65$ GHz

FFs t_{ccq} = 30 ps

t_{pcq} = 50 ps

t_{setup} = 60 ps

t_{hold} = 70 ps

Gates t_{pd}(g)= 35 ps

t_{cd}(g)= 25 ps

Hold time constraint:

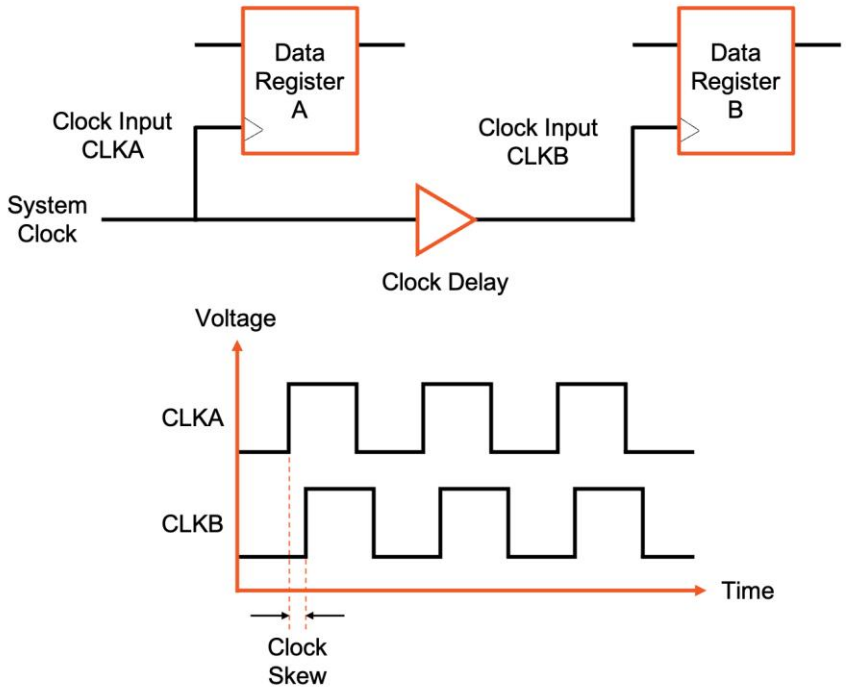
$t_{ccq} + t_{cd}$(com)> t_{hold} ?

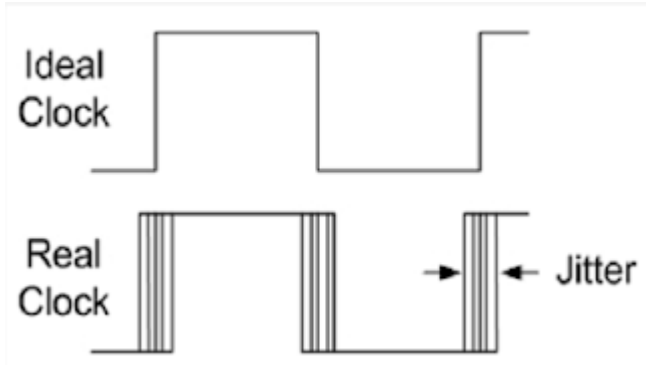
(30 + 50) ps > 70 ps ?

Yes!

Other non-idealities

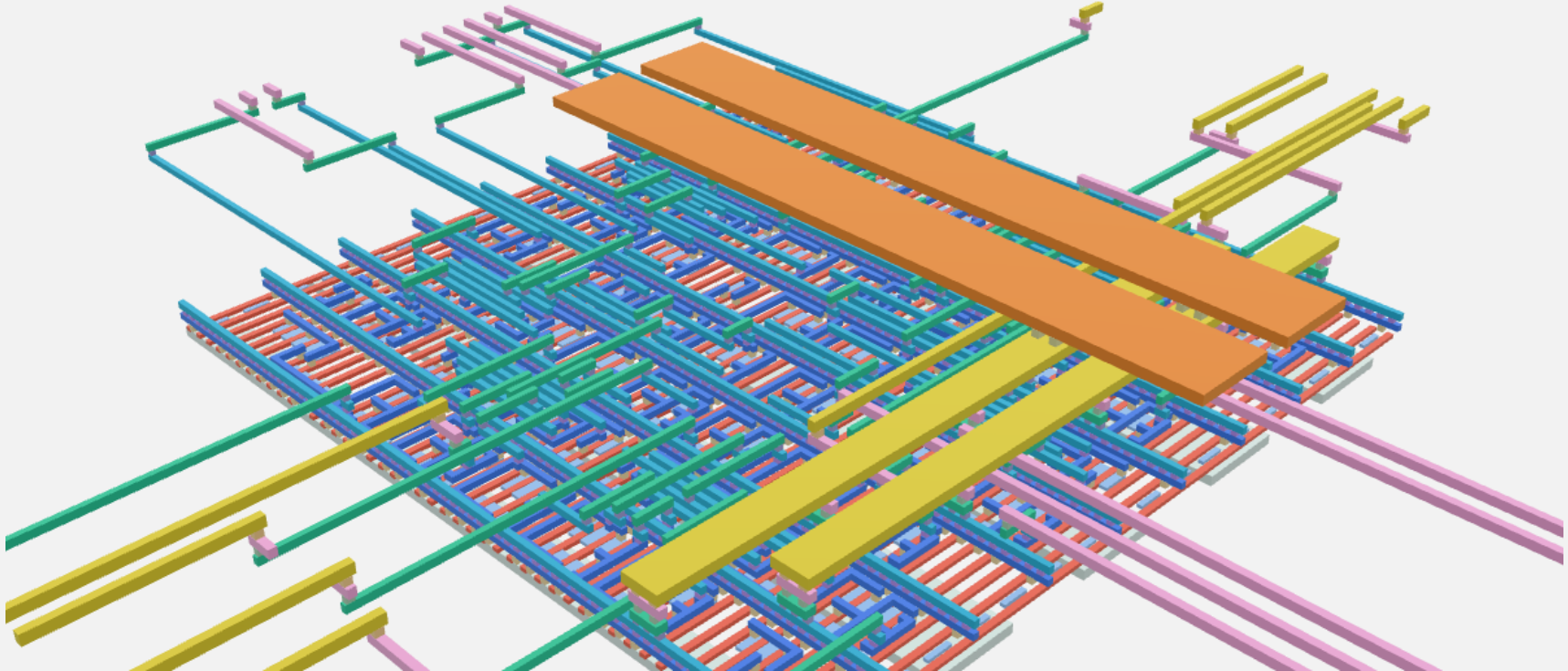
- Clock skew – clock doesn't arrive at all flops at the same time
- Clock jitter – clock isn't perfectly periodic





https://www.allaboutcircuits.com/technical-articles/what-is-clock-skew-understanding-clock-skew-and-clock-distribution-networks/

Live coding SystemVerilog



Pipelined Y = AX + B

- A module to perform AX+B
- Quick: two files anywhere




```
# in any folder, create these files:
#   ax_plus_b.sv
#   tb_ax_plus_b.sv

verilator --binary --timing --trace-vcd and_tb.sv   # compile sim

./obj_dir/Vand_tb                                   # run sim

gtkwave waveform.vcd                                # view wave
```

- Our flow: Makefile

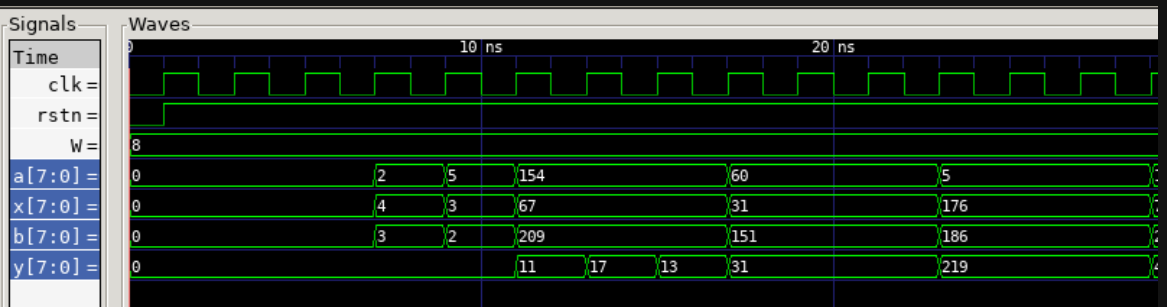
```
# RTL: material/rtl/reference/ax_plus_b.sv
# TB: material/tb/reference/tb_ax_plus_b.sv
# Filelist: material/designs/reference/ax_plus_b.f


make sim     DESIGN=ax_plus_b
make gtkwave DESIGN=ax_plus_b
```

Pipelined Y = AX + B

```
module ax_plus_b #(parameter W=8)(
  input clk, rstn,
  input  logic [W-1:0] x, a, b,
  output logic [W-1:0] y
);
  localparam  WS  = 2*W+1;
  localparam logic [W-1:0]  MIN = 1 << (W-1),
                            MAX = MIN - 1'b1;
  logic [2*W-1:0] mul;
  logic [WS-1 :0] sum;
  logic [W-1  :0] b_1;

  always_ff @(posedge clk)
    if (!rstn) {mul, y, b_1, sum} <= '0;
    else begin
      mul <= $signed(a) * $signed(x);
      b_1 <= b;
      sum <= WS'($signed(mul)) + WS'($signed(b_1));
      y   <=  $signed(sum) < $signed(MIN) ? MIN :
              $signed(sum) > $signed(MAX) ? MAX :
              W'(sum);
    end
endmodule
```



Clock generation, period=2ns

You can do this

Always write #1ps after the clock edge

```
// material/tb/reference/tb_ax_plus_b.sv

`timescale 1ns/1ps
module tb_ax_plus_b;
  localparam W = 8;
  localparam WS = 2*W+1;

  logic clk=0, rstn=0;
  logic [W-1:0] a, x, b, y, y_exp; logic signed [WS-1:0] sum_exp;

  initial forever #1 clk = !clk;

  ax_plus_b #(.W(W)) dut (.*);
  initial begin
    $dumpfile(`FST_PATH); $dumpvars;

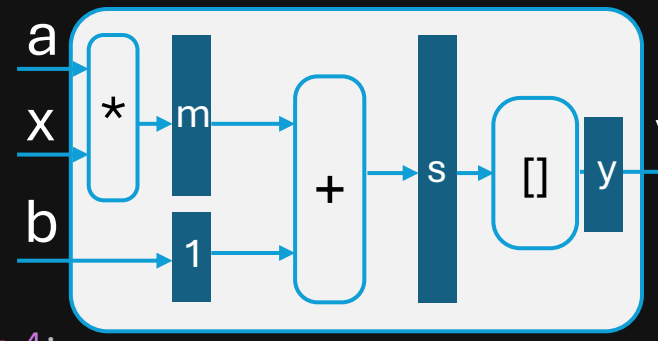
    @(posedge clk) rstn = 1;
    repeat(2) @(posedge clk);
    @(posedge clk) #1ps a = 2; b = 3; x = 4;
    @(posedge clk) #1ps a = 5; b = 2; x = 3;
    @(posedge clk) #1ps a = 5; b = 3; x = 2;

    repeat(10) begin
      #1ps a = W'($urandom);
           x = W'($urandom);
           b = W'($urandom);
      y_exp = W'(WS'($signed(a)) * WS'($signed(x)) + WS'($signed(b)));
      y_exp = sum_exp < WS'($signed(MIN)) ? MIN :
              sum_exp > WS'($signed(MAX)) ? MAX : W'(sum_exp);
      repeat(3) @(posedge clk);
      ;
      assert (y == y_exp) $display("OK");
      else $display("Error: y_exp=%d, y=%d", y_exp, y);
    end
    $finish();
  end
endmodule
```

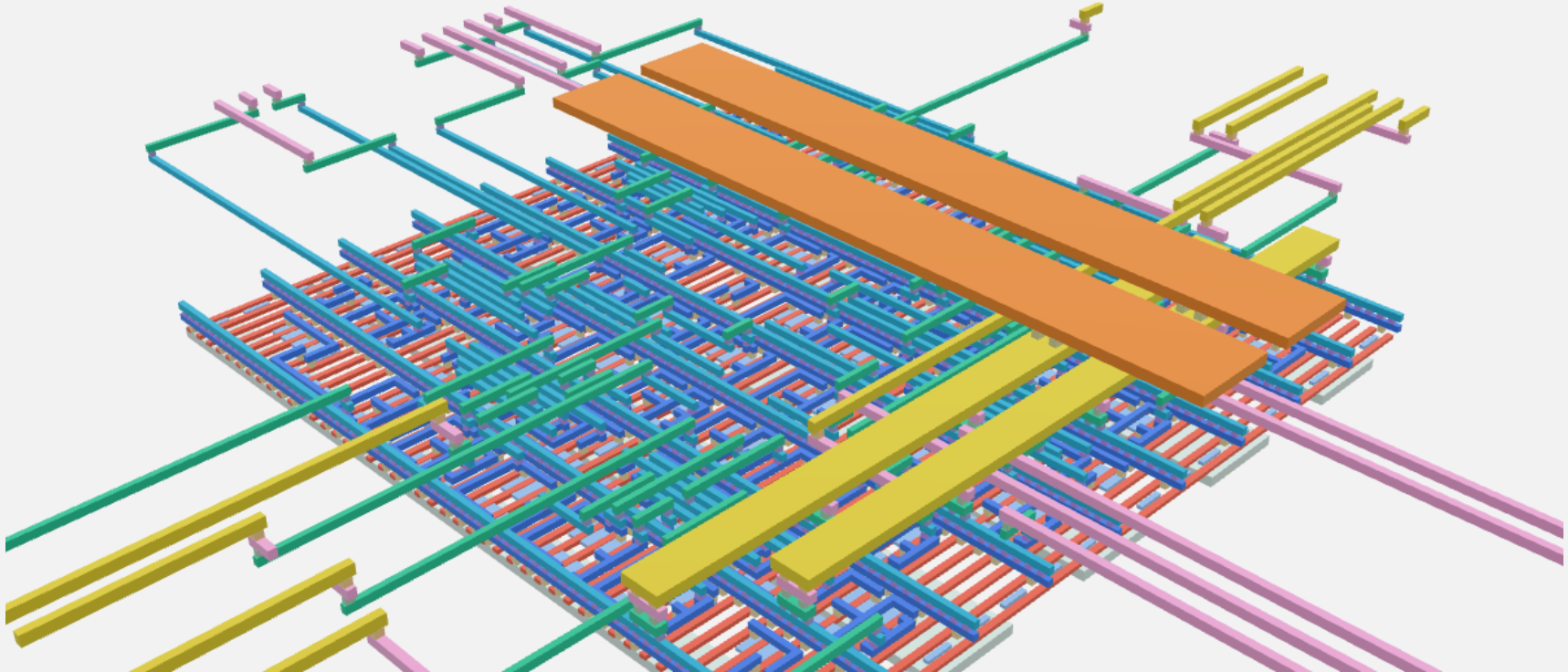
Pipeline latency = 3

```
make sim     DESIGN=ax_plus_b
make gtkwave DESIGN=ax_plus_b

# finds: material/designs/reference/ax_plus_b.f
```

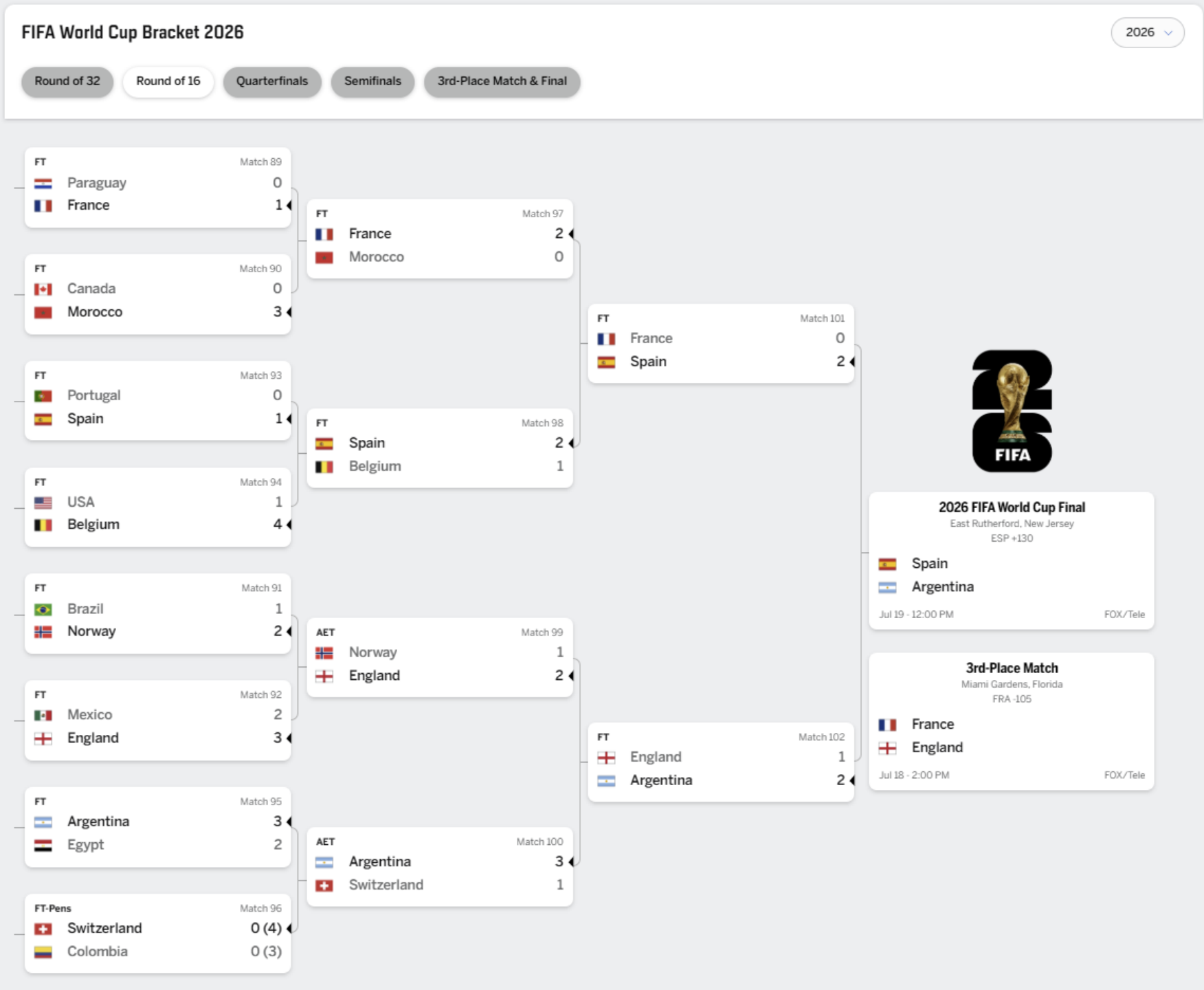


Binary Reduction Tree



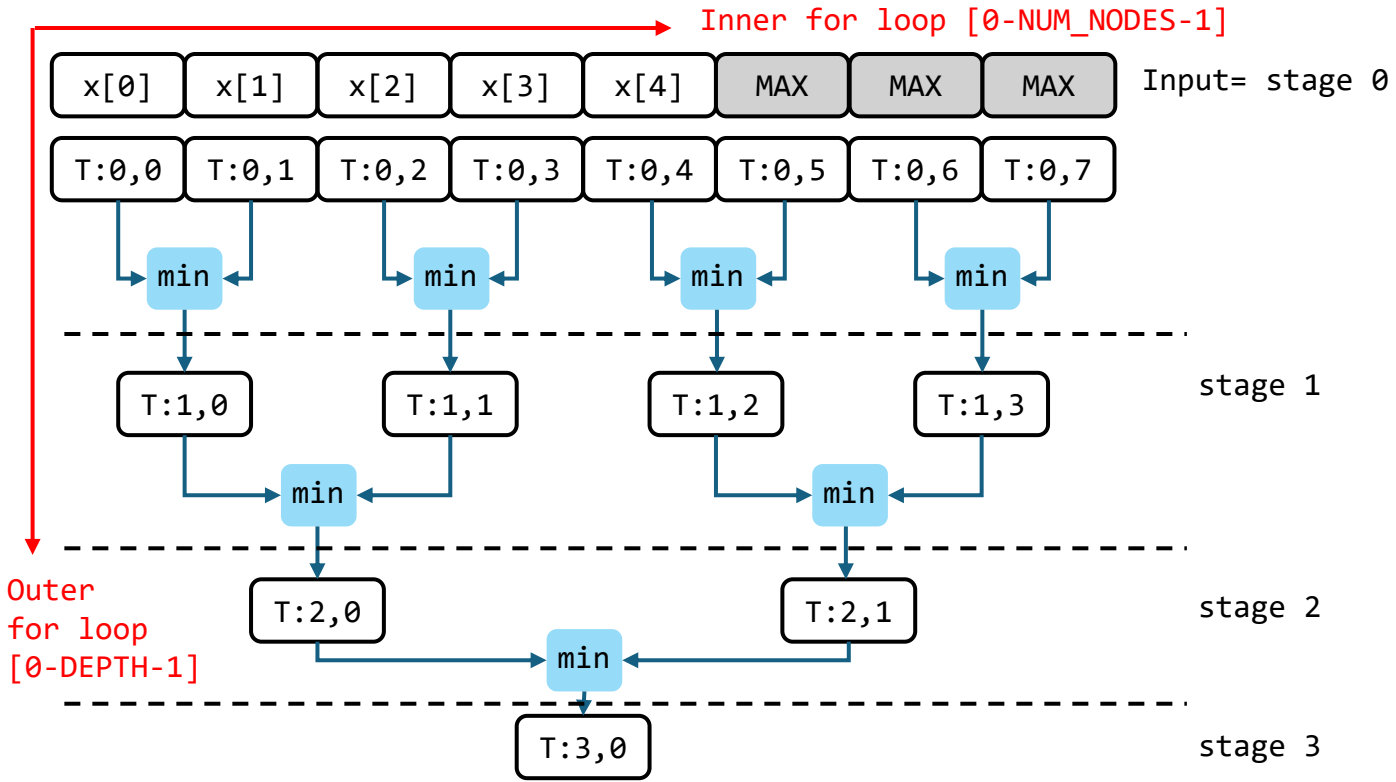
Binary Reduction Tree

- Reduction – combining multiple numbers into one
- Binary Reduction – combine two into 1 in each level
- E.g. FIFA Brackets



Binary Reduction Tree

- Compute y = min(X), X is an array
- Each stage cuts the arrays by half
- N_PAD = N padded to power of two
- DEPTH = clog2(N) stages
- Each stage has N_PAD >> (stage+1) nodes




```
module reduction_tree_min #(parameter int N = 5, W_X = 8)(
    input  logic clk, rstn, cen,
    input  logic [N-1:0][W_X-1:0] x,
    output logic        [W_X-1:0] y
  );
  localparam int DEPTH = $clog2(N), N_PAD = 2**DEPTH;
  localparam logic [W_X-1:0] MAX = (W_X'(1) << (W_X-1))-1'b1;

  genvar stage, node;
  logic [DEPTH:0][N_PAD-1:0][W_X-1:0] tree;

  always_comb begin
    for (int i = 0; i < N_PAD; i++)
      tree[0][i] = i < N ? x[i] : MAX;
  end

  for (stage = 0; stage < DEPTH; stage++) begin : g_stage
    localparam int NUM_NODES = N_PAD >> (stage + 1);

    for (node = 0; node < NUM_NODES; node++) begin : g_node
      logic [W_X-1:0] left, right;

      always_comb begin
        left  = tree[stage][2*node];
        right = tree[stage][2*node+1];
      end

      always_ff @(posedge clk or negedge rstn)
        if (!rstn)     tree[stage+1][node] <= '0;
        else if (cen)  tree[stage+1][node] <=
            $signed(left) < $signed(right)
              ? left : right;
    end
  end

  always_comb y = tree[DEPTH][0];

endmodule
```

genvar – used in generate for loops

Procedural for loop, use int

Assign input / MAX (padding) to tree

Nested generate loops

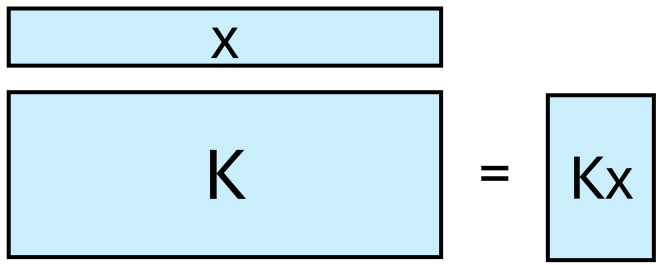
In every "min" – left/right nodes

Output of min is registered (tree)

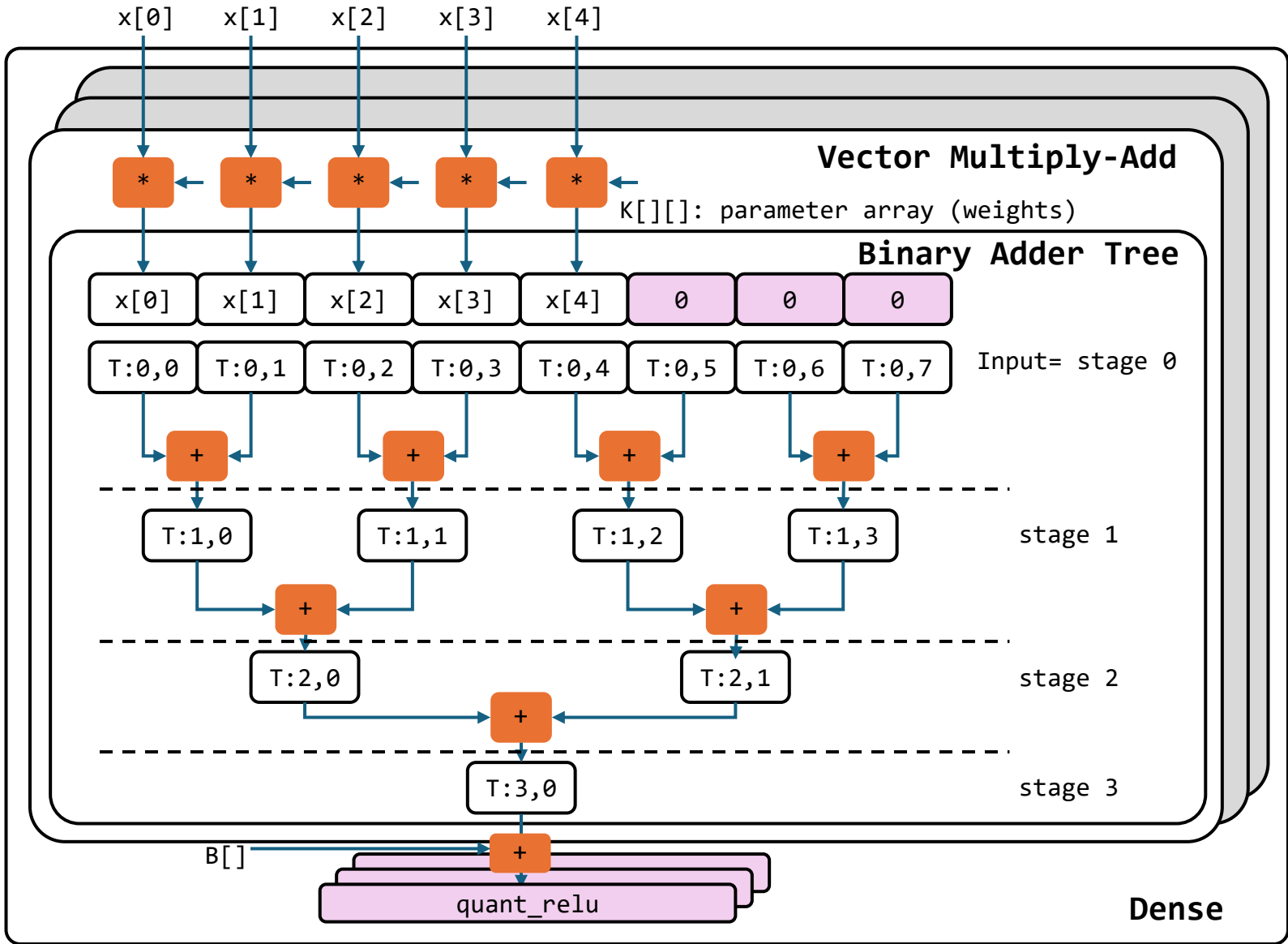
Output

Assignment 3

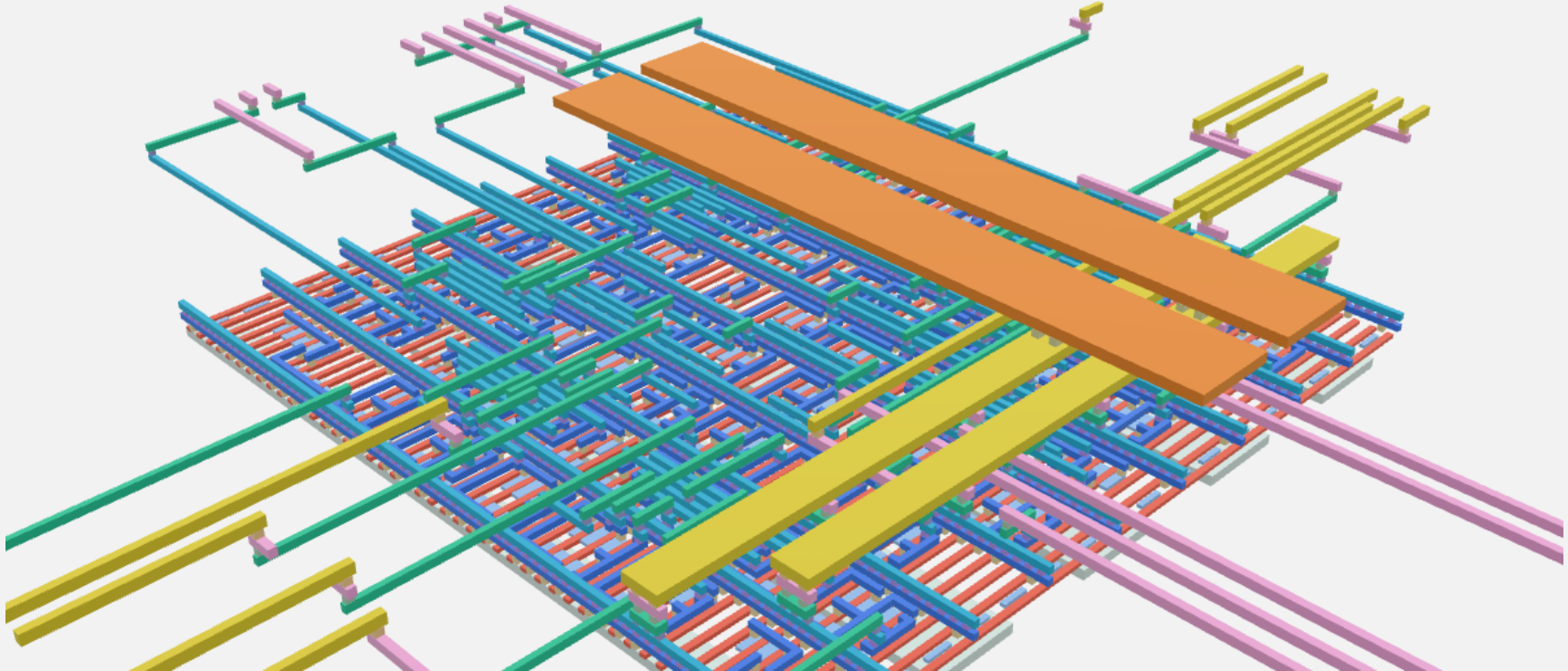
- Dense layer
- y = ReLU(Kx + B)



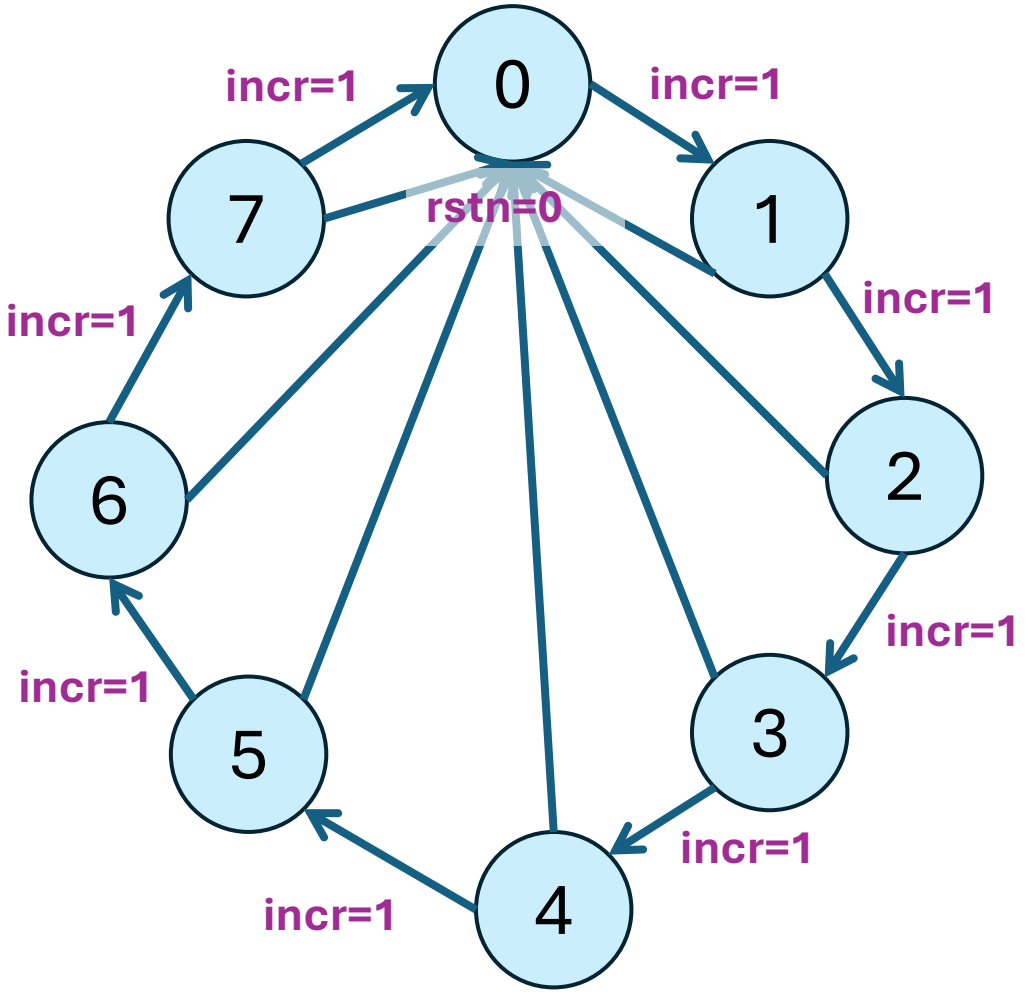




Finite State Machines



Counter

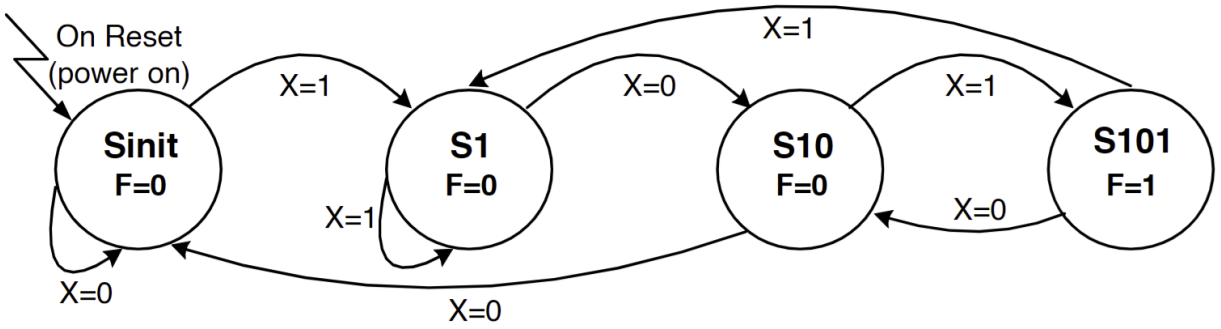



```
module up_counter #(parameter WIDTH = 3)(
  input  logic clk, rstn, incr,
  output logic [WIDTH-1:0] count
);
  always_ff @(posedge clk or negedge rstn) begin
    if      (!rstn) count <= 0;
    else if (incr)  count <= count + WIDTH'(1'b1);
  end
endmodule
```

- Moore machine
- Output only depends on state

Pattern Detector

"101" Sequence Detector should output F=1 when sequence 101 is found in consecutive order



State Diagram for "101" Sequence Detector

```
module detector_101 (
  input  logic clk, rstn, x,
  output logic f
);
```

typedef, enum

```
typedef enum logic [1:0]
        {SINIT, S1, S10, S101} state_t;
state_t state, next;
```

```
always_ff @(posedge clk or negedge rstn)
  if (!rstn) state <= SINIT;
  else       state <= next;
```

State sequencing

State transition

```
always_comb begin
  case (state)
    SINIT: next = x ? S1   : SINIT;
    S1:    next = x ? S1   : S10;
    S10:   next = x ? S101 : SINIT;
    S101:  next = x ? S1   : S10;
    default: next = SINIT;
  endcase
end
```

```
assign f = (state == S101);
```

State decoding

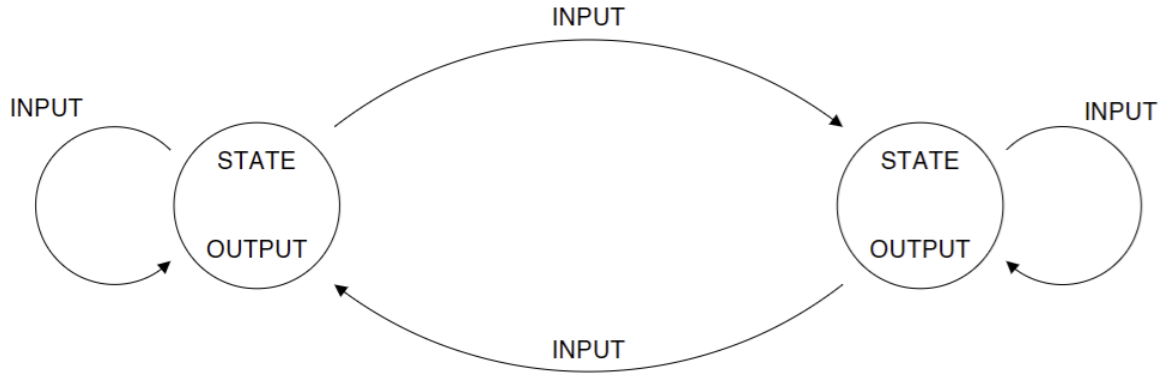
```
endmodule
```

https://ee.usc.edu/~redekopp/ee209/slides/EE209Spiral1-7.pdf

Mealy & Moore Machines

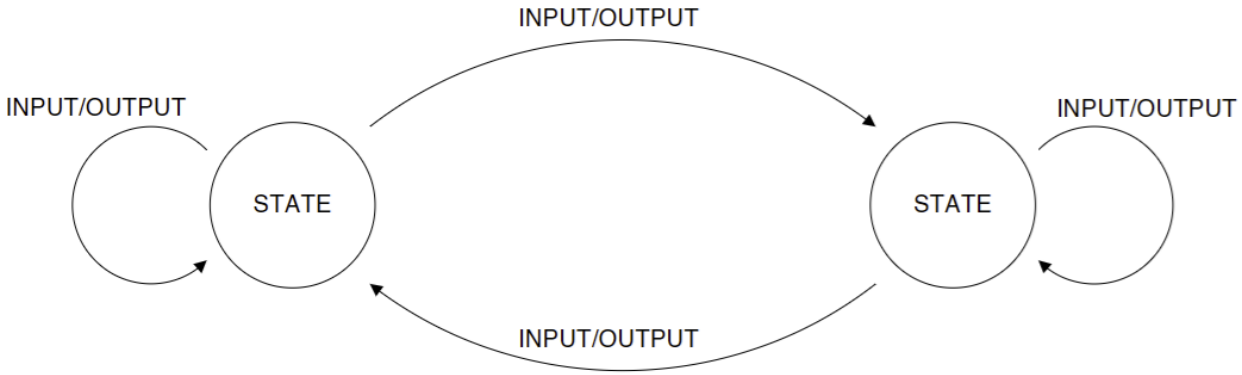
- Moore machine: out = f(state)

```
module up_counter #(parameter WIDTH = 3)(
  input  logic clk, rstn, incr,
  output logic [WIDTH-1:0] count,
  output logic less_than_3
);
  always_ff @(posedge clk or negedge rstn)
    if      (!rstn) count <= 0;
    else if (incr)  count <= count + WIDTH'(1'b1);
  always_comb less_than_3 = count < 3;
endmodule
```



- Mealy machine: out = f(state, inputs)

```
module up_counter #(parameter WIDTH = 3)(
  input  logic clk, rstn, incr,
  output logic [WIDTH-1:0] count, count_next
);
  always_ff @(posedge clk or negedge rstn)
    if      (!rstn) count <= 0;
    else if (incr)  count <= count + WIDTH'(1'b1);
  always_comb count_next = count + incr;
endmodule
```



Mealey & Moore Machines

	Moore Machine	Mealy Machine
Output depends on	Current state only	Current state and current input
Output location in diagram	Inside states	On transitions
Transition label	Input	Input / output
Output timing	Changes after state changes	Can change immediately when input changes
Number of states	Often more	Often fewer
Output stability	More stable	More sensitive to input glitches
Response speed	Usually one clock later	Can respond in the same clock cycle
Typical output logic	`z = f(state)`	`z = f(state, input)`

Down Counter

- Counting to zero and comparing with zero is slightly more efficient, a common pattern.
- Last and last clk are useful signals

```
module down_counter #(parameter WIDTH = 8)(
  input  logic clk, rstn, en, clear,
  input  logic [WIDTH-1:0] max_in,
  output logic [WIDTH-1:0] count,
  output logic last, last_clk
);
  logic [WIDTH-1:0] max;
  logic [WIDTH-1:0] count_next;

  always_ff @(posedge clk or negedge rstn) begin
    if (!rstn) {count, max, last} <= '0;
    else if (clear) begin
      count  <= max_in;
      max    <= max_in;
      last   <= max_in == 0;
    end
    else if (en) begin
      last   <= count_next == 0;
      count  <= count_next;
    end
  end

  always_comb begin
    count_next = last ? max : count - 1;
    last_clk = en && last && rstn && !clear;
  end

endmodule
```

Daily Feedback

- One thing I liked about today’s class:
- One thing that could be improved:
- One place where I felt lost (if any):