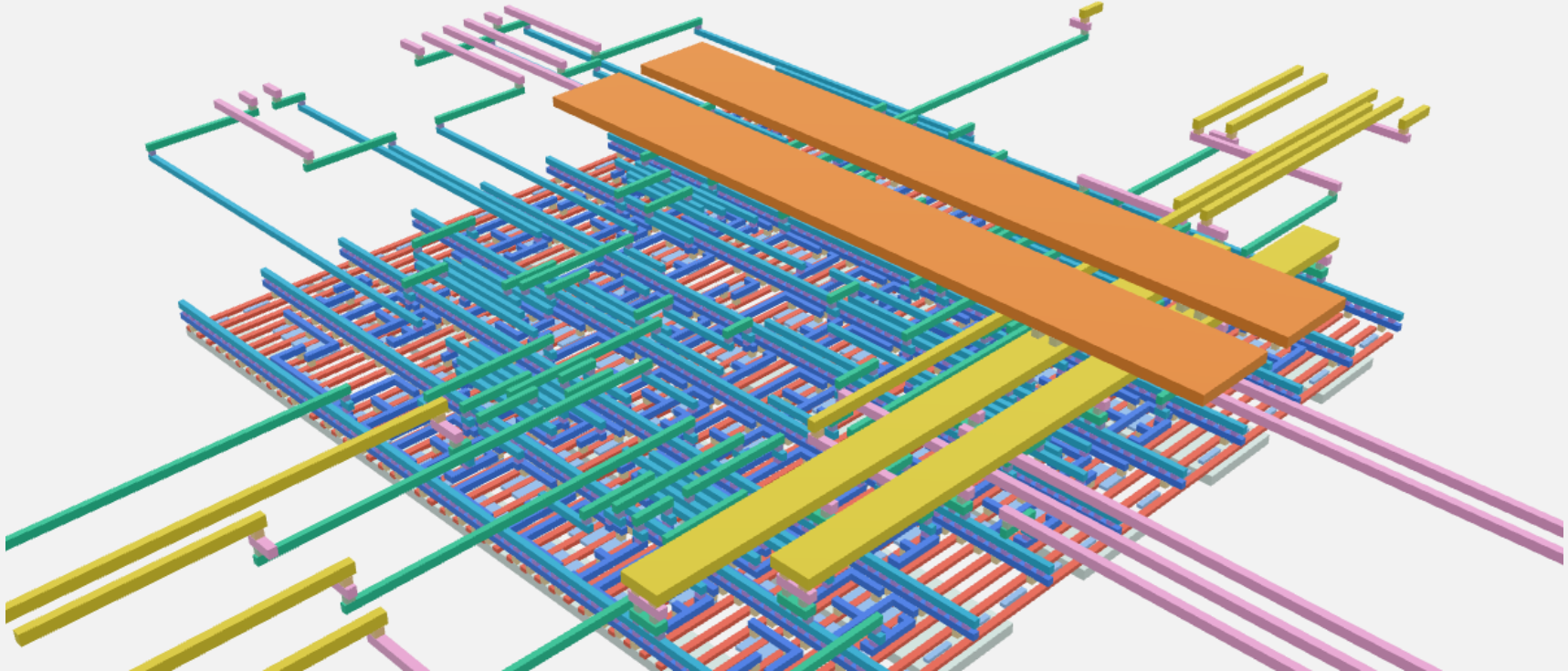
3.1
CSE140 – Digital Design

2026 Summer Session 1

Abarajithan G



Daily reminder:

- Open FeedbackFruits
  Canvas → CSE140 → Assignments → Slides 3.1

- Attendance is mandatory

- Midterm exam:
  - 45 minutes
  - ~20 MCQs
  - You need to know every line of your PA submission

Last Week’s Feedback

Seeing the corresponding SystemVerilog code next to the full adder example was pretty cool

Maybe going a little slower about Verilog?

i liked how you explained banker's rounding, and gave us a small example first

I think the pace should be faster. We spent almost 2 weeks out of 5 covering concepts already learned in prerequisite courses like CSE 30.

lot of examples solved through on board step by step, helped me understand better when hand solving the additions and multiplications

Don't rush the last topics, make ample time.

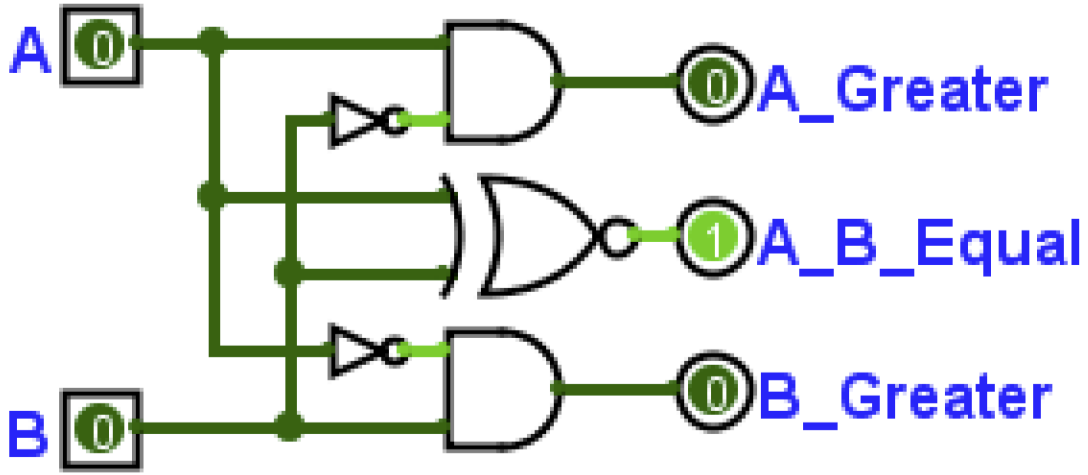
very fast paced today, struggled to keep up

Learning outcomes of this course

- Basic building blocks – Multiplexers, decoders, latches, flip flops
- SystemVerilog – designing more complex circuits & systems
- Math: How it translates to circuits & SystemVerilog features
- End-to-end flow, so you can see concepts like setup & hold time
- Physical implementation on FPGA

One-bit comparator

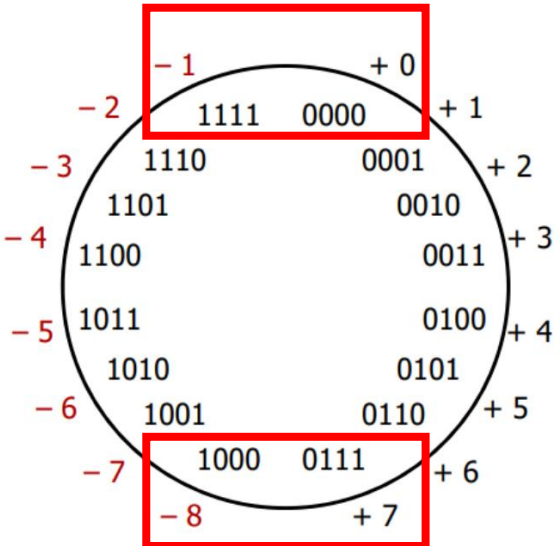
Inputs		Outputs		
B	A	$A > B$	$A = B$	$A < B$
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0



- Multiple bits? – Write the truth table, derive the circuit

Saturating Adder

- Detecting overflow
  - Hold the true sum in B+1 bits
  - If the sign bits of sum and true sum are different, overflow has occurred.
- E.g. 3-bit number, 4-bit true sum.




```
`timescale 1ns/1ps

module saturating_adder #(
  parameter W = 8
)(
  input  logic [W-1:0] a, b,
  output logic [W-1:0] s
);
  localparam logic [W-1:0] MIN = W'(1) << (W-1);   // 1000...0
  localparam logic [W-1:0] MAX = MIN - 1'b1;       // 0111...1


  logic [W:0] sum_ext;   // one extra bit holds the true sum
  logic overflow, negative;

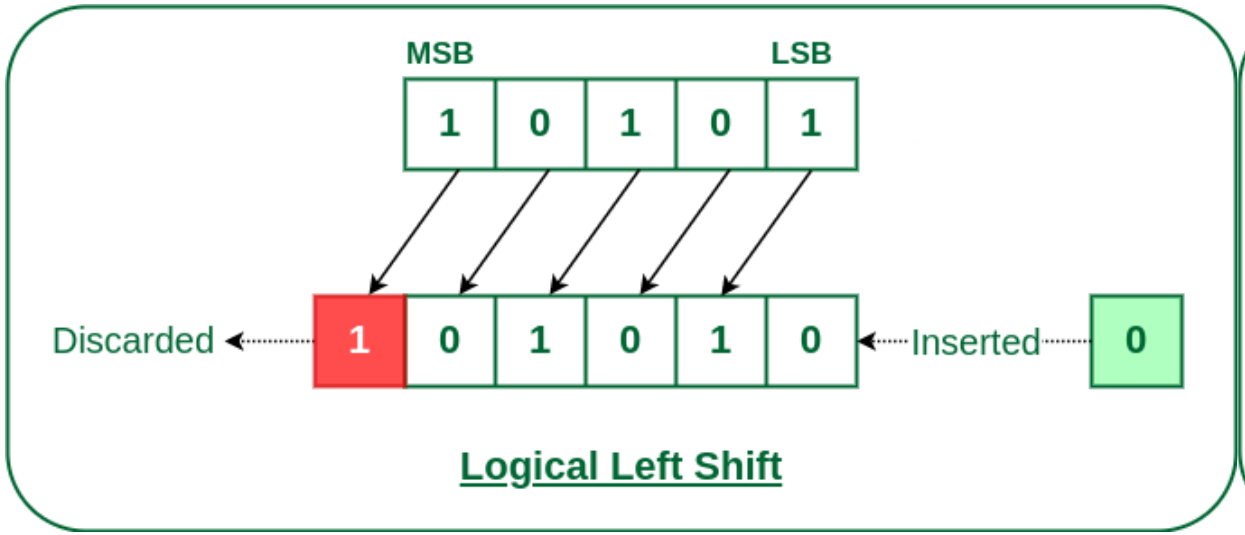

  always_comb begin
    sum_ext = $signed(a) + $signed(b);

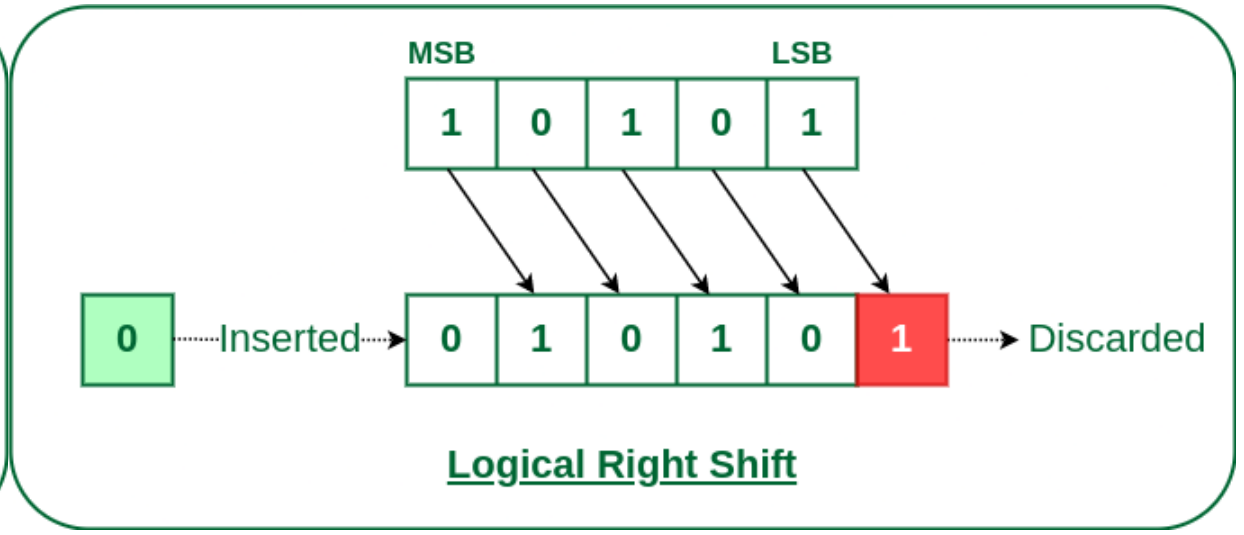

    overflow = (sum_ext[W] != sum_ext[W-1]);
    negative = sum_ext[W];

    if (overflow) s = negative ? MIN : MAX;
    else          s = sum_ext[W-1:0];
  end
endmodule
```

Logical Shifting

- Discard bits
- Insert Zeros





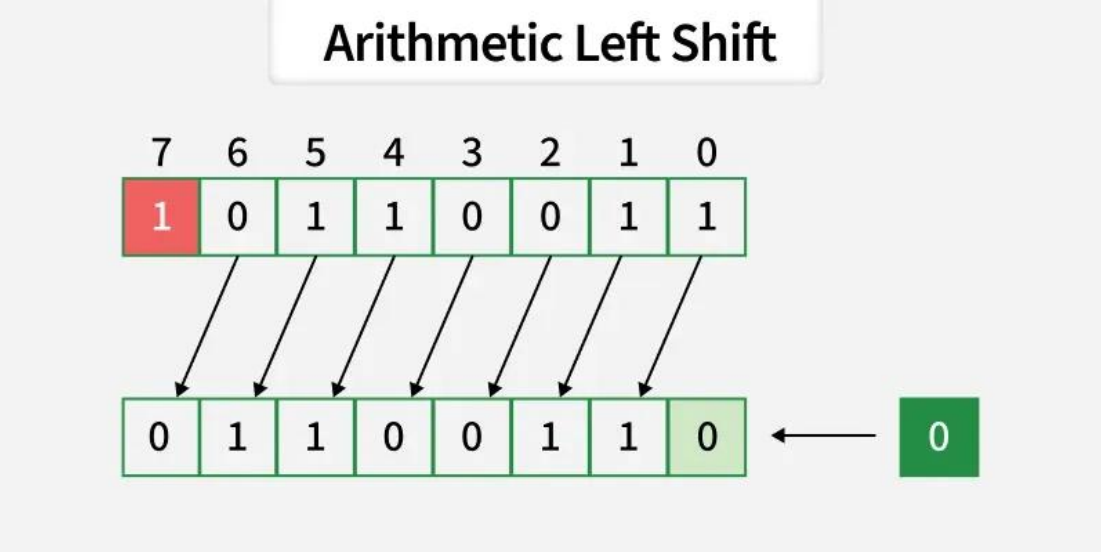

```
y = x << s
```

```
y = x >> s
```

Arithmetic Shifting

Left:

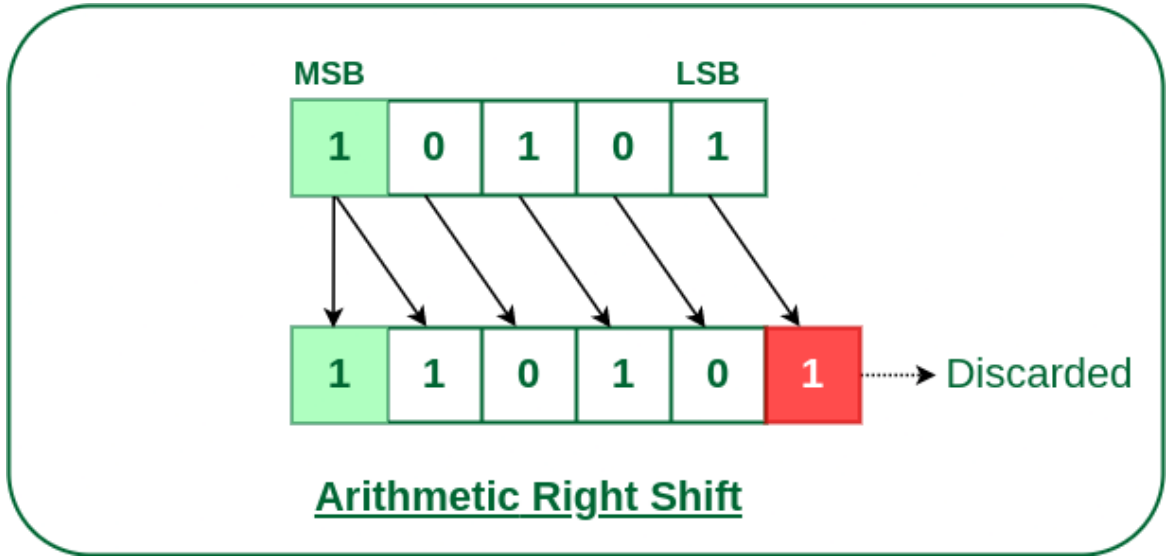
- Discard bits
- Insert Zeros




```
y = x <<< s
```

Right:

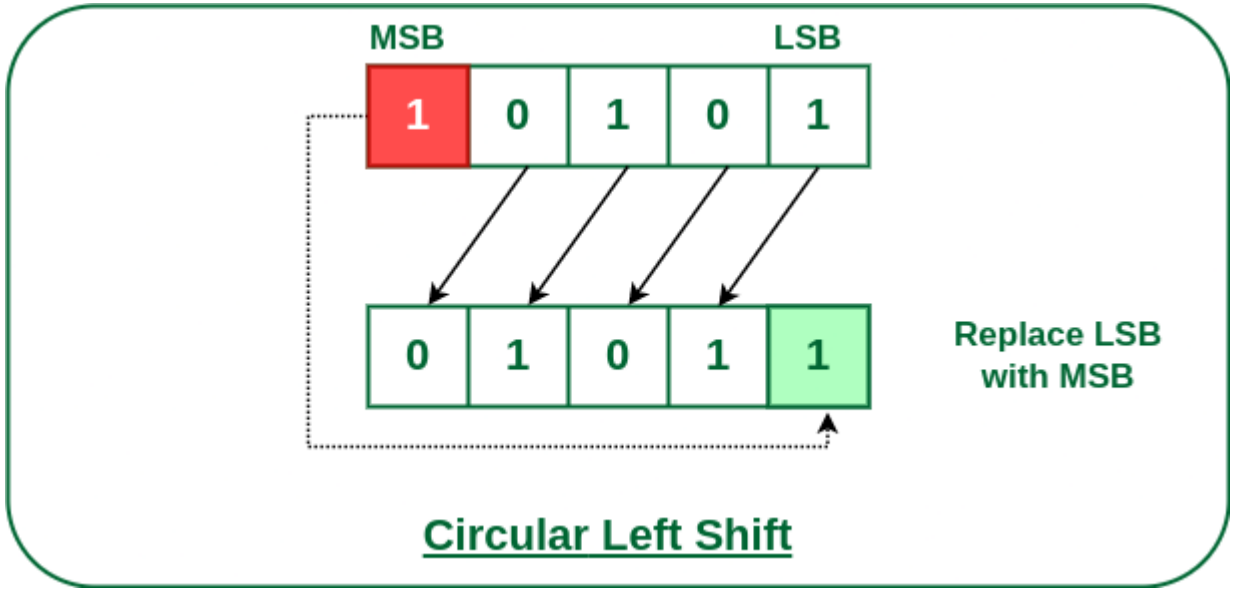
- Discard bits
- Insert **sign bit,** if x is signed

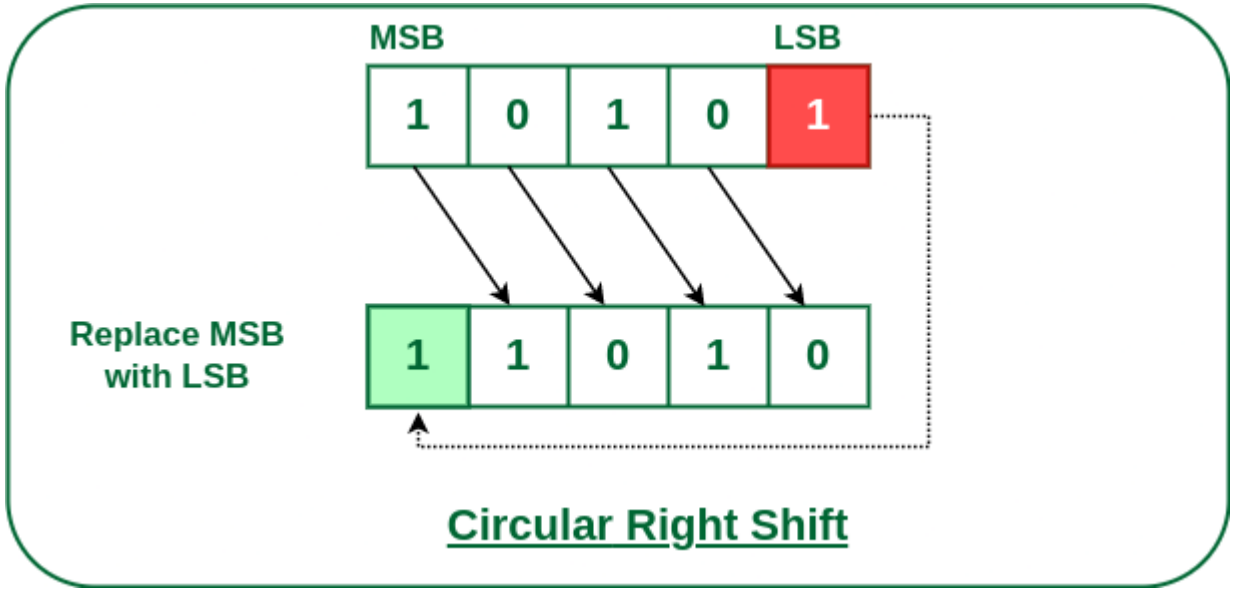



```
y = $signed(x) >>> s
```

Circular shifting

- Nothing is discarded
- Wrap around

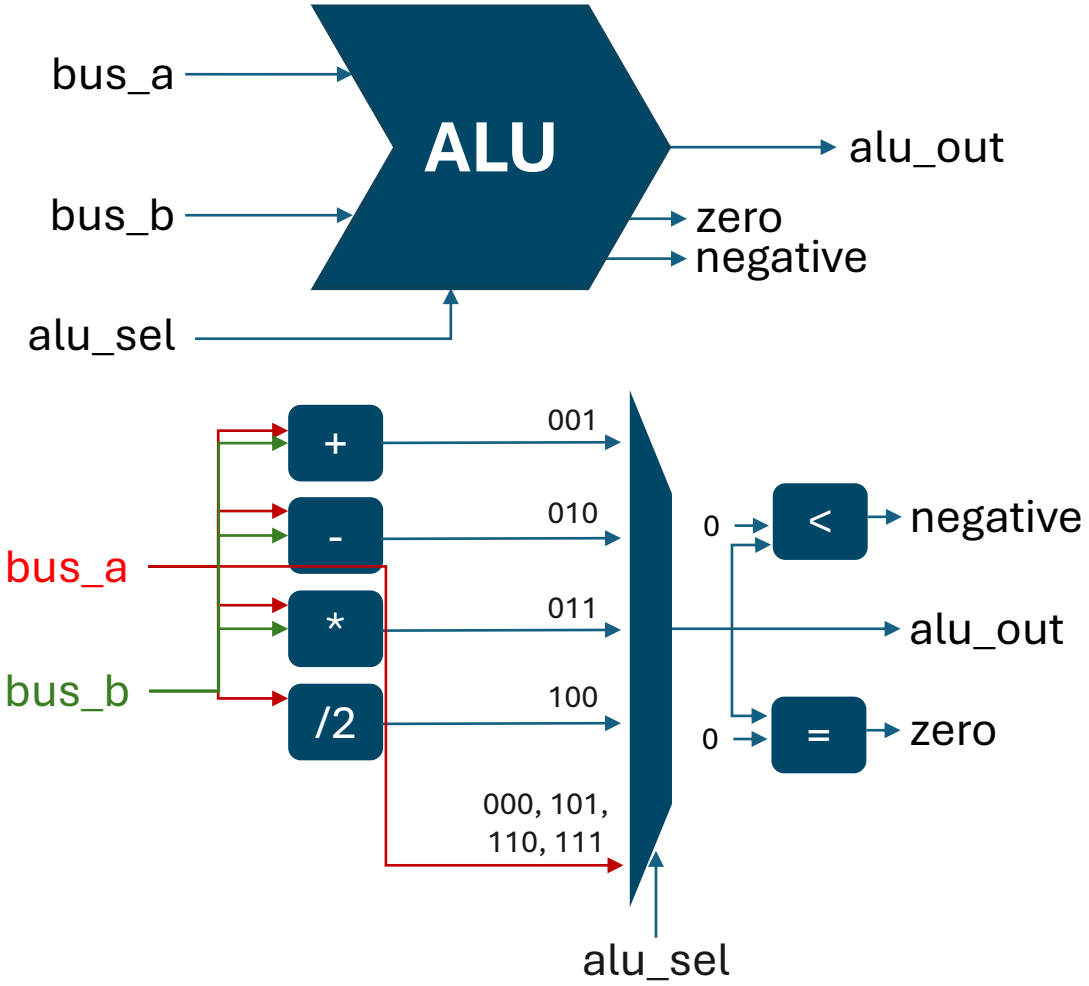





```
y = (x << s) | (x >> (W - s))
```

```
y = (x >> s) | (x << (W - s))
```

ALU




```
`timescale 1ns/1ps

module alu #(
  parameter  WIDTH     = 8,
  localparam W_ALU_SEL = 3
)(
  input  logic [WIDTH-1:0] bus_a,
  input  logic [WIDTH-1:0] bus_b,
  output logic [WIDTH-1:0] alu_out,
  input  logic [W_ALU_SEL   -1:0] alu_sel,
  output logic zero, negative
);

  always_comb begin

    unique case (alu_sel)
      'b001  : alu_out = $signed(bus_a) + $signed(bus_b);
      'b010  : alu_out = $signed(bus_a) - $signed(bus_b);
      'b011  : alu_out = $signed(bus_a) * $signed(bus_b);
      'b100  : alu_out = $signed(bus_a)/2;
      default: alu_out = bus_a; // pass a
    endcase

    zero     = (alu_out == 0);
    negative = $signed(alu_out) < 0;
  end

endmodule
```

case : multiplexer

$signed() – because signed / unsigned multiplication are different

Multiplication – truncates to WIDTH bits.

Careful - Even if one operand is unsigned, SV will make the operation unsigned.

Activity

1. What is **out** when
   `WIDTH =8; a=8'd16; b=8'd16; sel=3'b011;`
2. What is **out** (in binary, eg 4'bxxxx) when
   `WIDTH= 4; a= 4'b1000; b= 4'b1111; sel= 3'b011;`
3. What is **out,** if we use >> for shift, when
   `WIDTH= 4; a= 4'b1000; b= 4'b1111; sel= 3'b100;`
4. What is **out,** if we do **negative=out<0;** when
   `out = 4'b1111;`

```
`timescale 1ns/1ps


module alu #(
  parameter  WIDTH     = 8,
  localparam W_ALU_SEL = 3
)(
  input  logic [WIDTH-1:0] a,
  input  logic [WIDTH-1:0] b,
  output logic [WIDTH-1:0] out,
  input  logic [W_ALU_SEL   -1:0] sel,
  output logic zero, negative
);


  always_comb begin


    unique case (sel)
      'b001  : out = $signed(a) + $signed(b);
      'b010  : out = $signed(a) - $signed(b);
      'b011  : out = $signed(a) * $signed(b);
      'b100  : out = $signed(a) >>> 1;
      default: out = a; // pass a
    endcase


    zero      = (out == 0);
    negative  = $signed(out) < 0;
  end


endmodule
```

Activity

1. What is **out** (in decimal) when
   `WIDTH =8; a=8'd16; b=8'd16; sel=3'b011;` 0 - overflow
2. What is **out** (in binary, eg 4'bxxxx) when out = 4'b1000
   `WIDTH= 4; a= 4'b1000; b= 4'b1111; sel= 3'b011;`
3. What is **out,** if we use >> for shift, when out = 4'b0100 = 4
   `WIDTH= 4; a= 4'b1000; b= 4'b1111; sel= 3'b100;`
4. What is **out,** if we do **negative=out<0;** when
   `out = 4'b1111;` out = 15 <0 = 0 (false)

```
`timescale 1ns/1ps


module alu #(
  parameter  WIDTH     = 8,
  localparam W_ALU_SEL = 3
)(
  input  logic [WIDTH-1:0] a,
  input  logic [WIDTH-1:0] b,
  output logic [WIDTH-1:0] out,
  input  logic [W_ALU_SEL   -1:0] sel,
  output logic zero, negative
);


  always_comb begin


    unique case (sel)
      'b001  : out = $signed(a) + $signed(b);
      'b010  : out = $signed(a) - $signed(b);
      'b011  : out = $signed(a) * $signed(b);
      'b100  : out = $signed(a) >>> 1;
      default: out = a; // pass a
    endcase


    zero      = (out == 0);
    negative  = $signed(out) < 0;
  end


endmodule
```

ALU Testbench

```
`timescale 1ns/1ps

module tb_alu;
  localparam WIDTH = 8;

  logic [WIDTH-1:0] bus_a, bus_b, alu_out;
  logic [2:0] alu_sel;
  logic zero, negative;

  alu #(.WIDTH(WIDTH)) dut (.*);

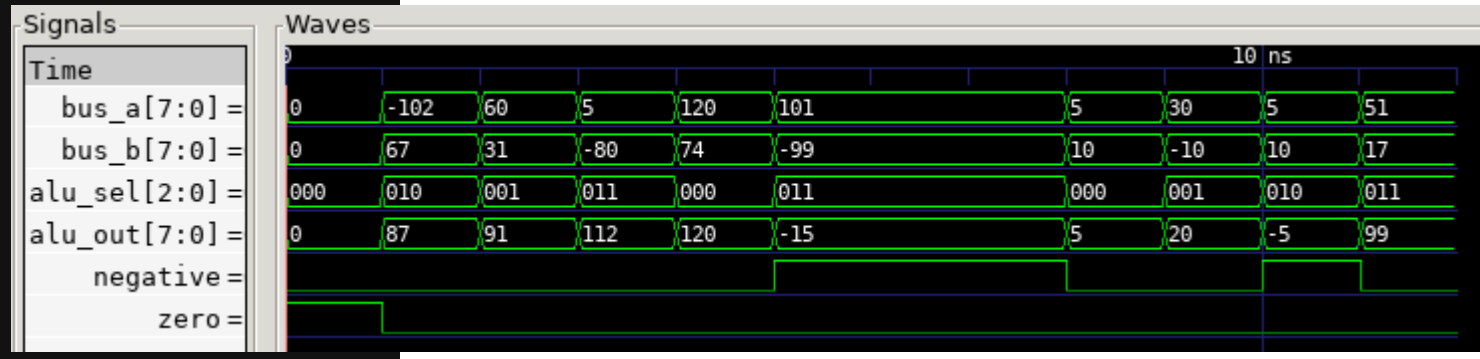
  initial begin
    $dumpfile(`FST_PATH); $dumpvars;

    repeat (5) begin
      #1;
      bus_a   = WIDTH'($urandom);
      bus_b   = WIDTH'($urandom);
      alu_sel = 3'($urandom_range(4, 0));
    end

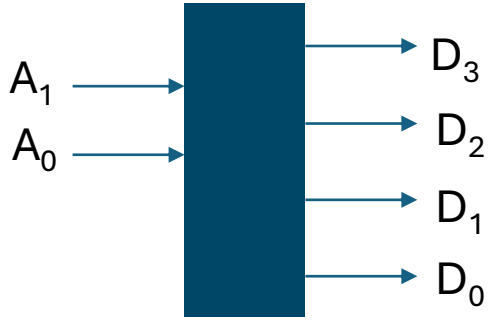
    #3 bus_a =  8'sd5;   bus_b =  8'sd10;  alu_sel = 3'b000;
    #1 bus_a =  8'sd30;  bus_b = -8'sd10;  alu_sel = 3'b001;
    #1 bus_a =  8'sd5;   bus_b =  8'sd10;  alu_sel = 3'b010;
    #1 bus_a =  8'sd51;  bus_b =  8'sd17;  alu_sel = 3'b011;
    #1 $finish;
  end
endmodule
```

```
# inside container (after make enter)

make sim gtkwave DESIGN=alu
```

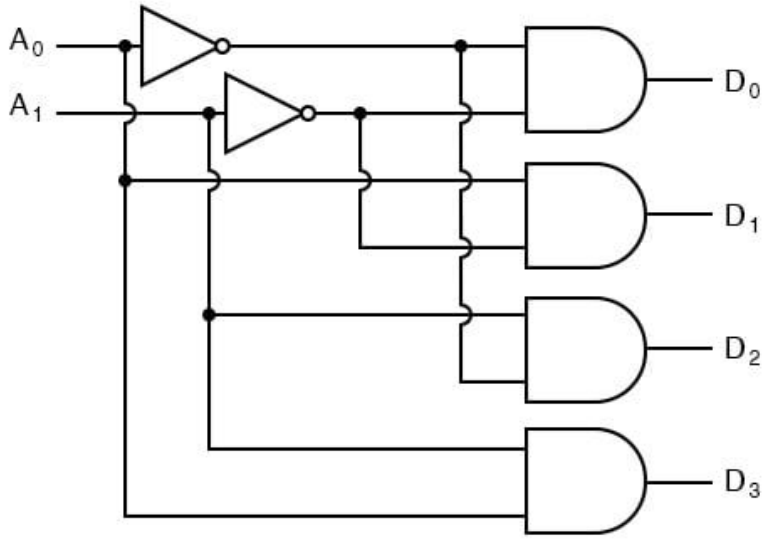


Decoder



- Converts a N bit input (code) into a 2^N (one-hot) output
- When A=n, only D_n=1, all other D_i are 0 – one hot
- Application – convert address to enable signals

i	A_1	A_0	D_3	D_2	D_1	D_0
0	0	0	0	0	0	1
1	0	1	0	0	1	0
2	1	0	0	1	0	0
3	1	1	1	0	0	0




```
`timescale 1ns/1ps


module decoder #(
  parameter N = 3
)(
  input  logic [N-1:0]    code,
  output logic [2**N-1:0] onehot
);


  always_comb begin

    onehot        = '0;
    onehot[code] = 1'b1;

  end

endmodule
```

Last assignment wins

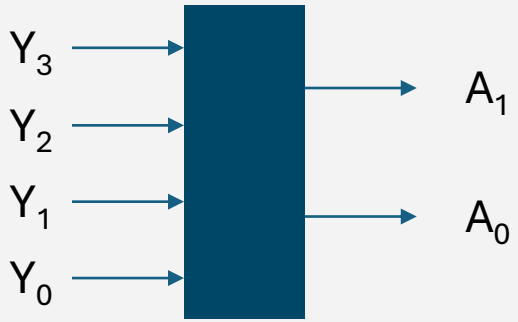
Encoder

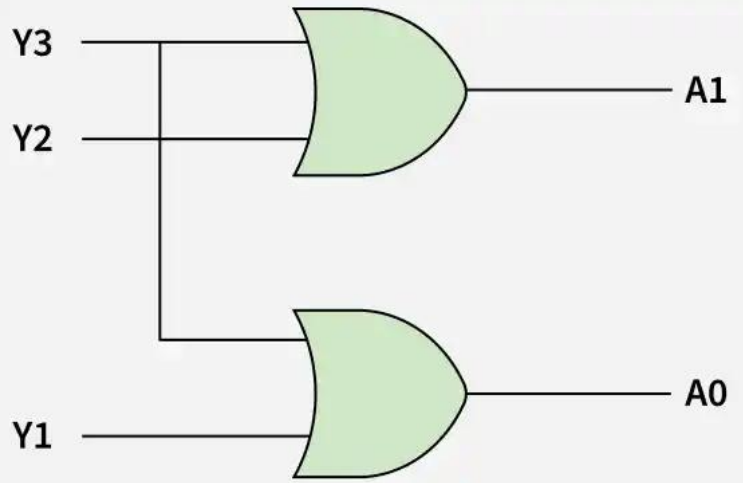
- Converts a 2^N (one-hot) into N-bit input (code) output

```
`timescale 1ns/1ps

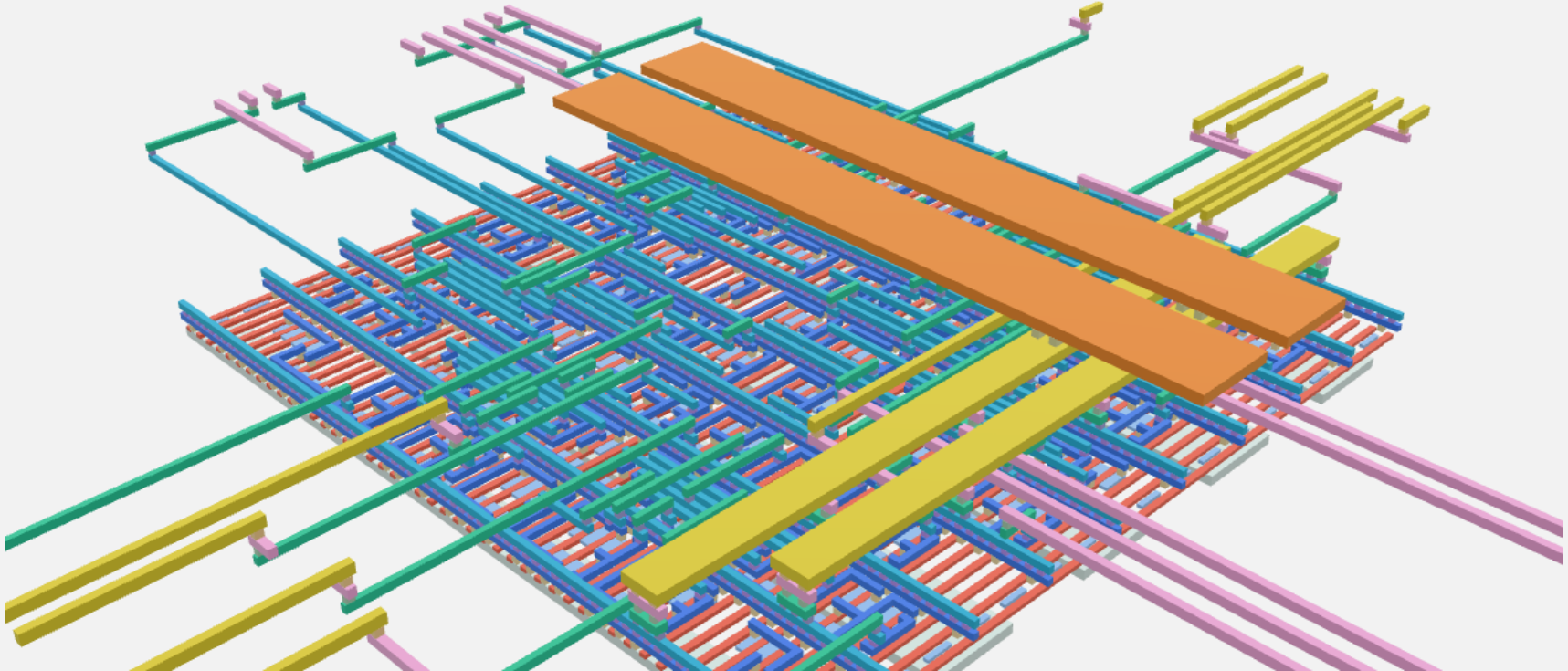
module encoder #(
  parameter N = 3
)(
  input  logic [2**N-1:0] onehot,
  output logic [N-1:0]    code
);
  always_comb begin
    code = '0;
    for (int i = 0; i < 2**N; i++)
      if (onehot[i]) code = N'(i);
  end
endmodule
```

i	Y_3	Y_2	Y_1	Y_0	A_1	A_0
0	0	0	0	1	0	0
1	0	0	1	0	0	1
2	0	1	0	0	1	0
3	1	0	0	0	1	1

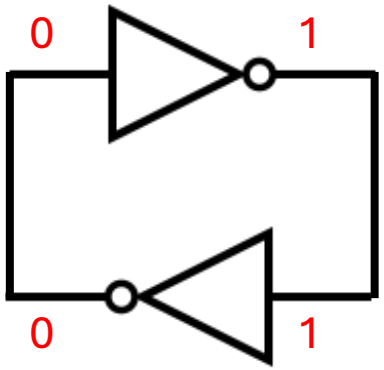


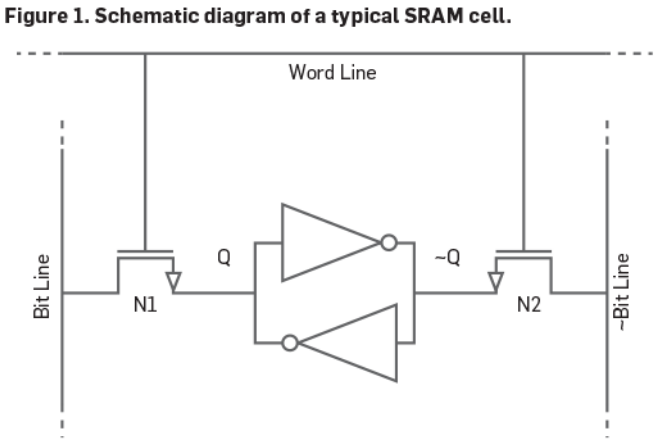


Sequential Elements



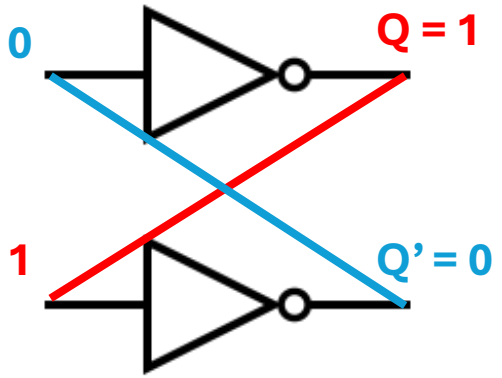
Storing a bit





https://mags.acm.org/communications/august_2025/MobilePagedArticle.action?articleId=2073744#articleId2073744

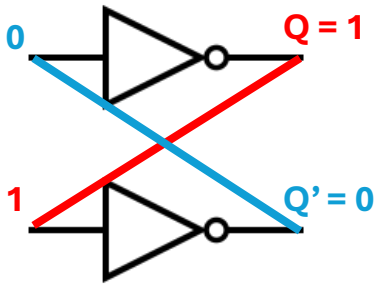
- Combinational circuits are memoryless
- How do we create memory?
- Feedback loops!
- This is how SRAM stores data – two NOT gates reinforcing each other
- But where are the inputs and outputs?
- In SRAM, they force an input with a strong pulse, but we need something better.



The loop will be drawn like this from now

- Note: Avoid unintentional combinational loops in your SV code.

SR Latch



A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

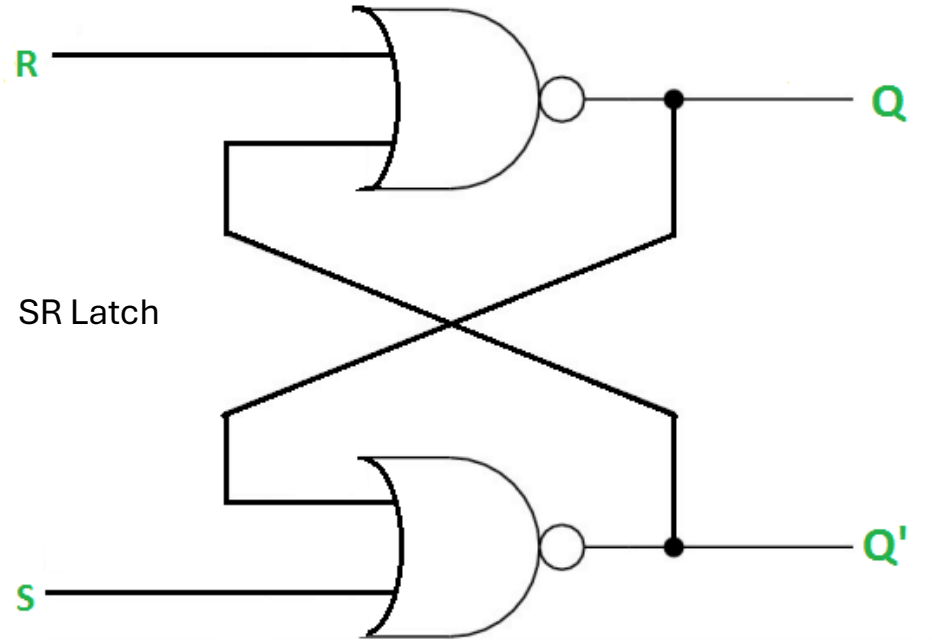
NOR Truth Table

S	R	Q	Q'
0	0	Latch	Latch
0	1	0	1
1	0	1	0
1	1	0	0

SR Latch
Truth Table

- We need inputs – replace NOT with NOR
- **S** sets the latch : S=1 → Q'=0 → Q=1
- **R** resets the latch : R=1 → Q =0
- When **S=0 && R=0**,
  - It acts like the NOT-loop
  - Holds ANY value – Latch!
- When **S=1 && R=1**,
  - Q=0, Q'=0 – impossible?
  - Metastability – indeterministic, not good!
  - How to prevent this?



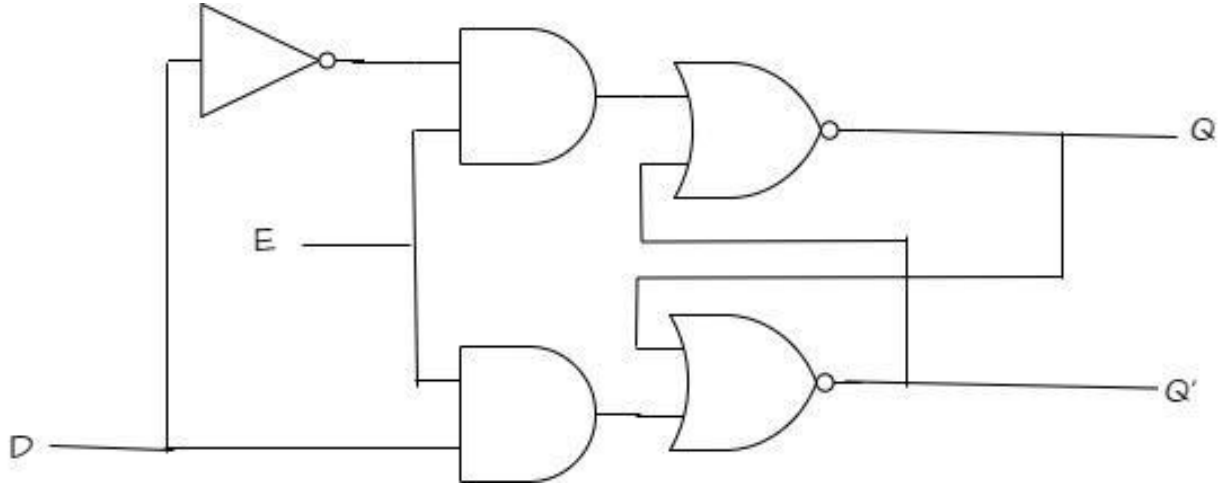


SR Latch

geeksforgeeks.org/digital-logic/latches-in-digital-logic/

D-Latch

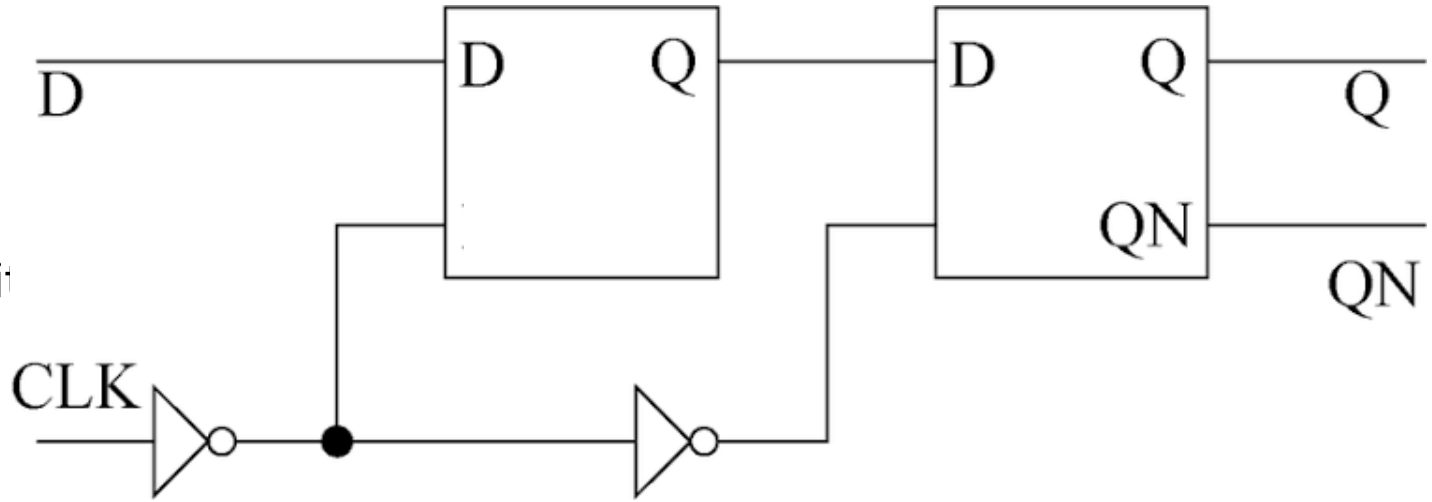
- How to prevent **S=1 && R=1**?
  - Give only one input – **D**
  - But then it won't "latch" (S=R=0)
- Another input – **E** (enable)
  - E=1: Latch is "transparent" to D
  - E=0: Latch is "opaque" / latched to past D
- D-latch is the standard & most common latch
- Problem with latch:
  - Level-sensitive
  - Edge-sensitive is better



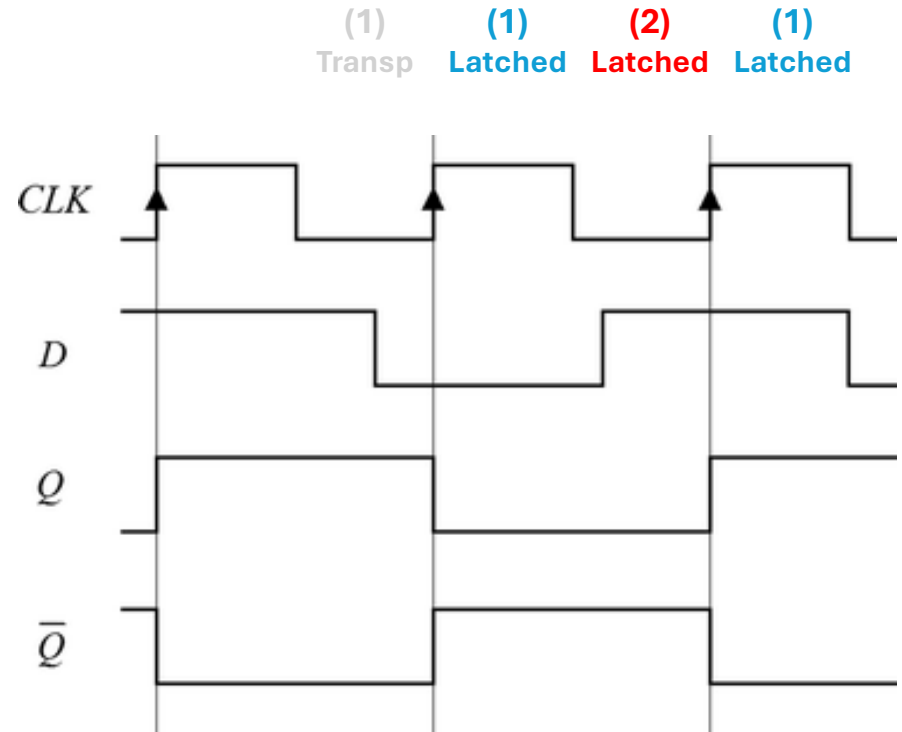
E	D	Q(next)	Q'(next)
0	0	Q(prev)	Q(prev)
0	1	Q(prev)	Q(prev)
1	0	0	1
1	1	1	0

D Latch
Truth Table

D-Flip Flop



- We want to capture a signal at a given (posi
  edge of a clock signal
- Why – its almost instantaneous
- How: two D-latches in series
- First latch is
  transparent when `clk=0`, latches when `clk =1`
- Second latch is
  transparent when `clk=1`, latches when `clk=0`
- Output is delayed by one “clock cycle”
- D-flip flop is the standard flip flop



Register

- Flip flop stores 1 bit
- Register stores N-bits (N flops)
- Accepts data at a clock edge (usually posedge)
- **always_ff:**
  - **Flip flop**
  - **Use ONLY non-blocking statements (<=)**
  - **In always_comb, use ONLY blocking statements (=)**
- Note: Never use latches, (unless you know what you're doing)
- Always use registers to store data temporarily

```
module flip_flop (
  input  logic clk,
  input  logic inp,
  output logic out
);
  always_ff @(posedge clk) begin
    out <= inp;
  end
endmodule
```

```
module register (
  input  logic clk,
  input  logic [31:0] inp,
  output logic [31:0] out
);
  always_ff @(posedge clk) begin
    out <= inp;
  end
endmodule
```

Registers: Common Patterns

- With clock enable
- With reset
  - FPGA, commonly:
    - active high – reset when reset=1
    - synchronous – reset also checked at clk edge, and will update in next cycle
  - ASIC, commonly:
    - Active low – reset when rstn=0 (n implies that)
    - Asynchronous – reset immediately
    - Depends on PDK
    - Active-low reset is convenient for power-on reset because rstn can be held low while the supply rises, then released high after the chip is stable.
- Can have reset + clock enable too

```
module register (
  input  logic clk, ce,
  input  logic [31:0] inp,
  output logic [31:0] out
);
  always_ff @(posedge clk)
    if(ce) out <= inp;
endmodule

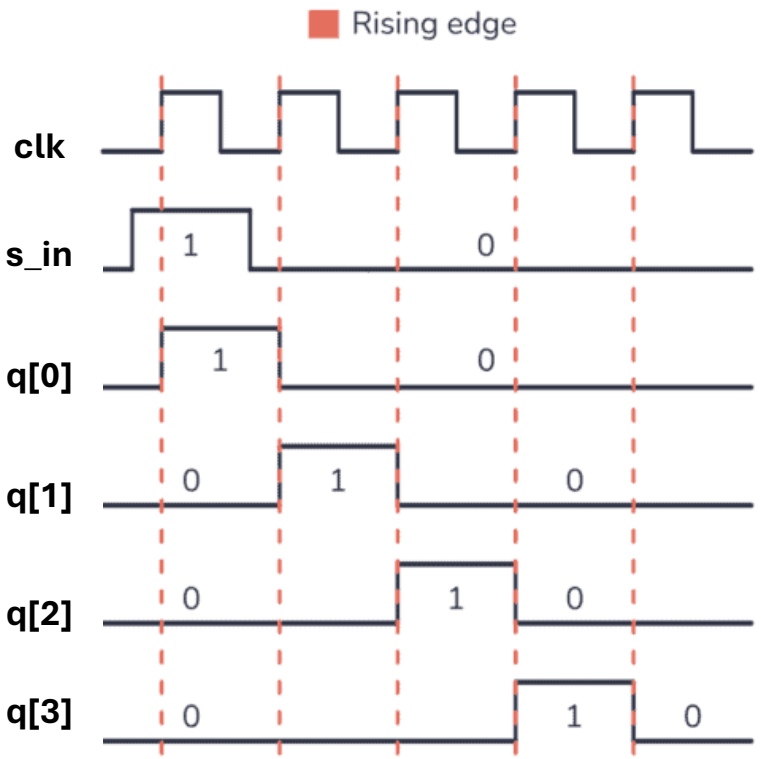
module register (
  input  logic clk, rst,
  input  logic [31:0] inp,
  output logic [31:0] out
);
  always_ff @(posedge clk)
    if(rst) out <= 0;
    else    out <= inp;
endmodule

module register (
  input  logic clk, rstn,
  input  logic [31:0] inp,
  output logic [31:0] out
);
  always_ff @(posedge clk or negedge rstn)
    if(!rstn) out <= 0;
    else      out <= inp;
endmodule

module register (
  input  logic clk, rstn, ce,
  input  logic [31:0] inp,
  output logic [31:0] out
);
  always_ff @(posedge clk or negedge rstn)
    if(!rstn)   out <= 0;
    else if(ce) out <= inp;
endmodule
```

Shift register

- Delay a signal over many clock cycles
- Get the value of a signal from the past N clocks




```
module shift_register (
  input  logic       clk, rst, s_in,
  output logic [3:0] q
);

  always_ff @(posedge clk) begin
    if (rst) q <= '0;
    else     q <= {q[2:0], s_in};
  end

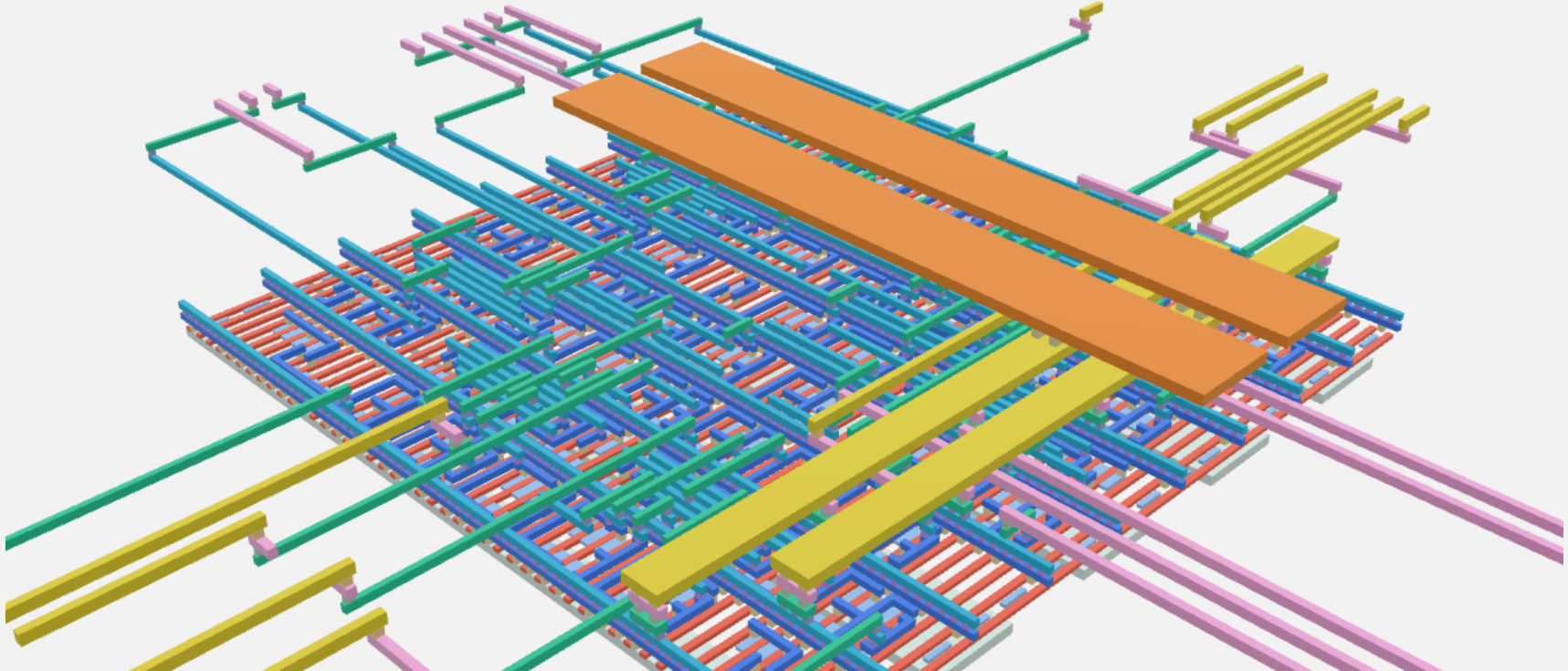
endmodule
```

```
module shift_register (
  input  logic       clk, rst, s_in,
  output logic [3:0] q
);

  always_ff @(posedge clk) begin
    if (rst) begin
      q <= '0;
    end else begin
      q[0] <= s_in;
      q[1] <= q[0];
      q[2] <= q[1];
      q[3] <= q[2];
    end
  end

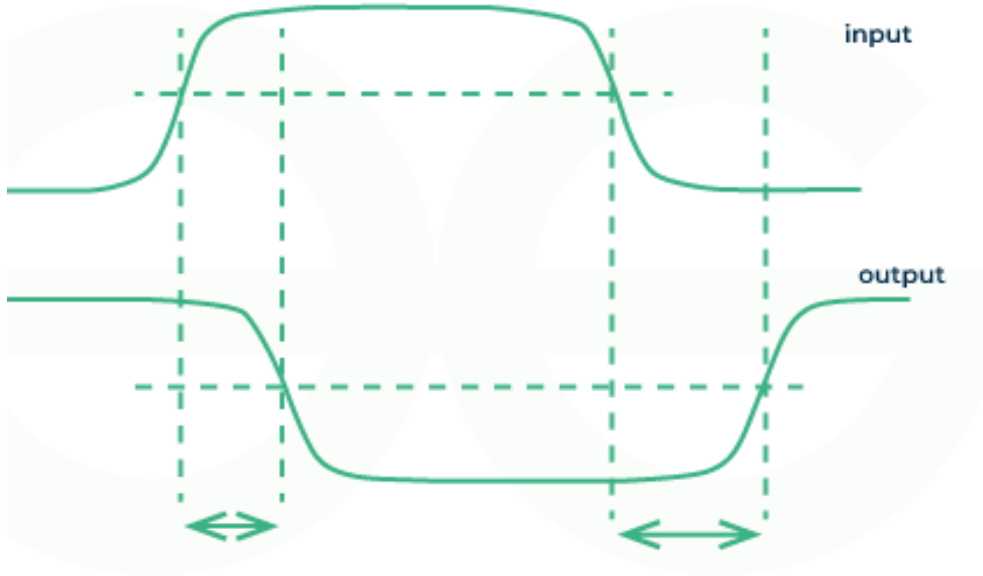
endmodule
```

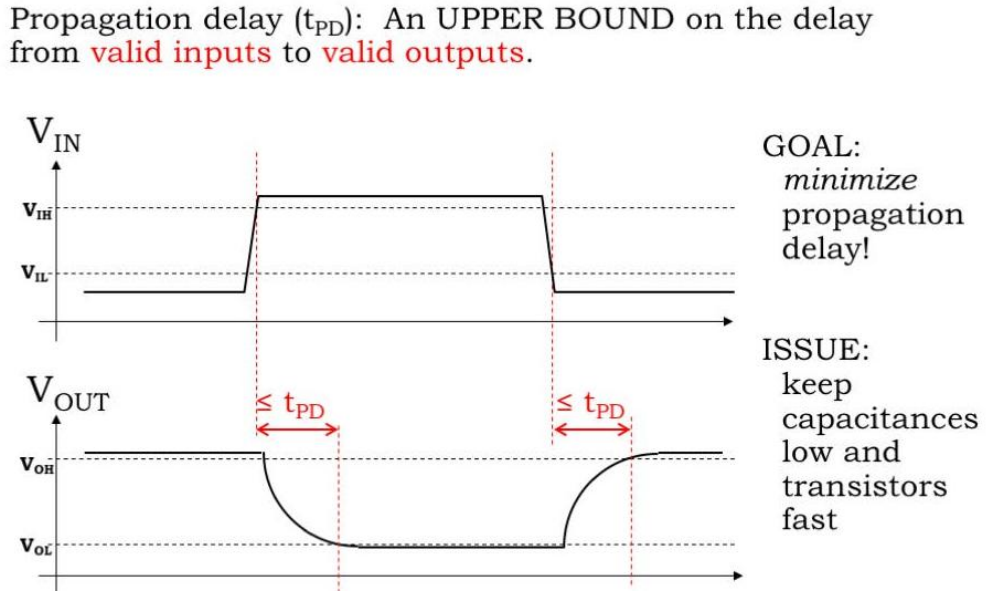
Timing Analysis



Propagation Delay

- Combinational circuits are not really instantaneous
- Gates have delays – transistors take time
- 0→1 and 1→0 can have different delays
- Propagation delay is the maximum time (upper bound) after an input change until the output is guaranteed to have reached its new valid value.



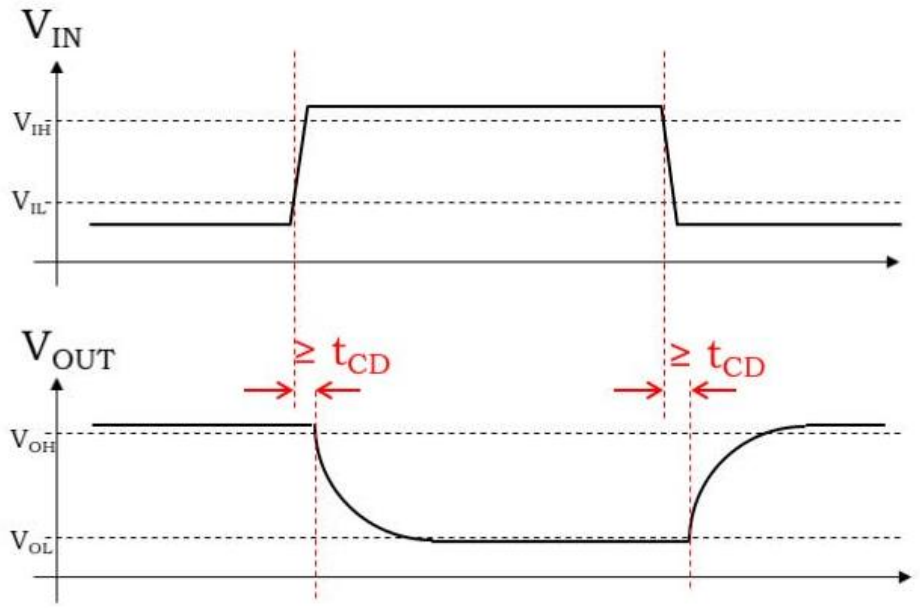


https://ocw.mit.edu/courses/6-004-computation-structures-spring-2017/

Contamination Delay

The minimum time after an input change before an output can begin changing.

Contamination delay (t_{CD}): A LOWER BOUND on the delay from any invalid input to an invalid output



Do we really need t_{CD}?

Usually not… it'll be important when we design circuits with registers (coming soon!)

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

https://ocw.mit.edu/courses/6-004-computation-structures-spring-2017/

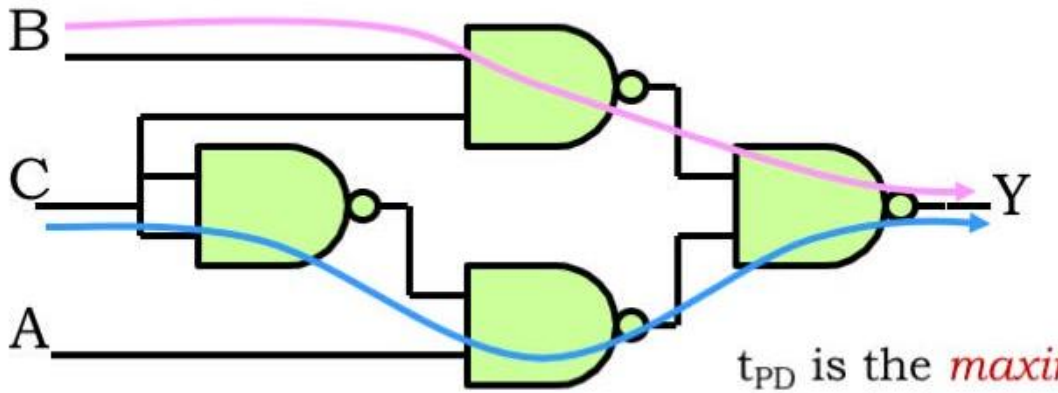
Find tCD and tPD

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

t_{PD} = _____ nS

t_{CD} = _____ 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

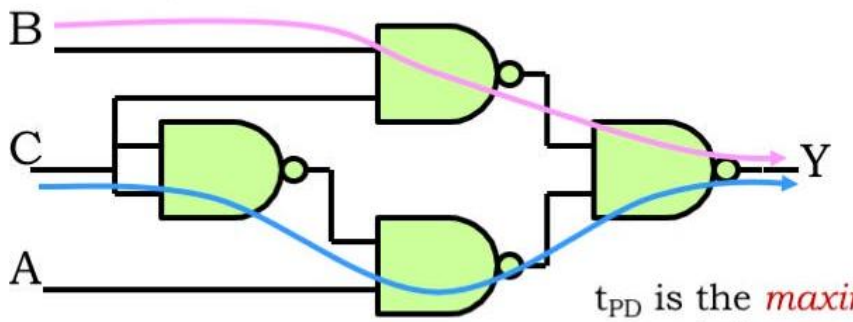
Find tCD and tPD

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

t_{PD} = 12 nS

t_{CD} = 2 nS

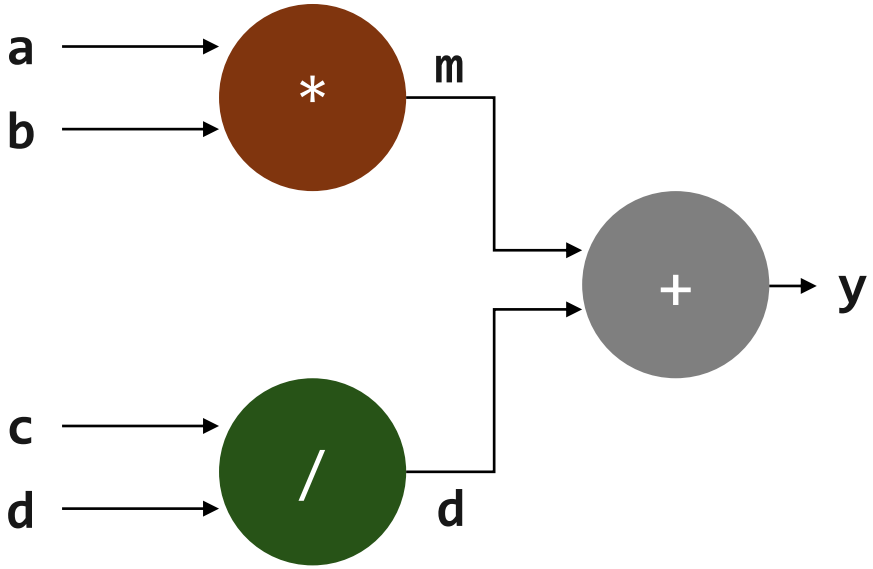
t_{CD} is the *minimum* cumulative contamination delay over all paths from inputs to outputs



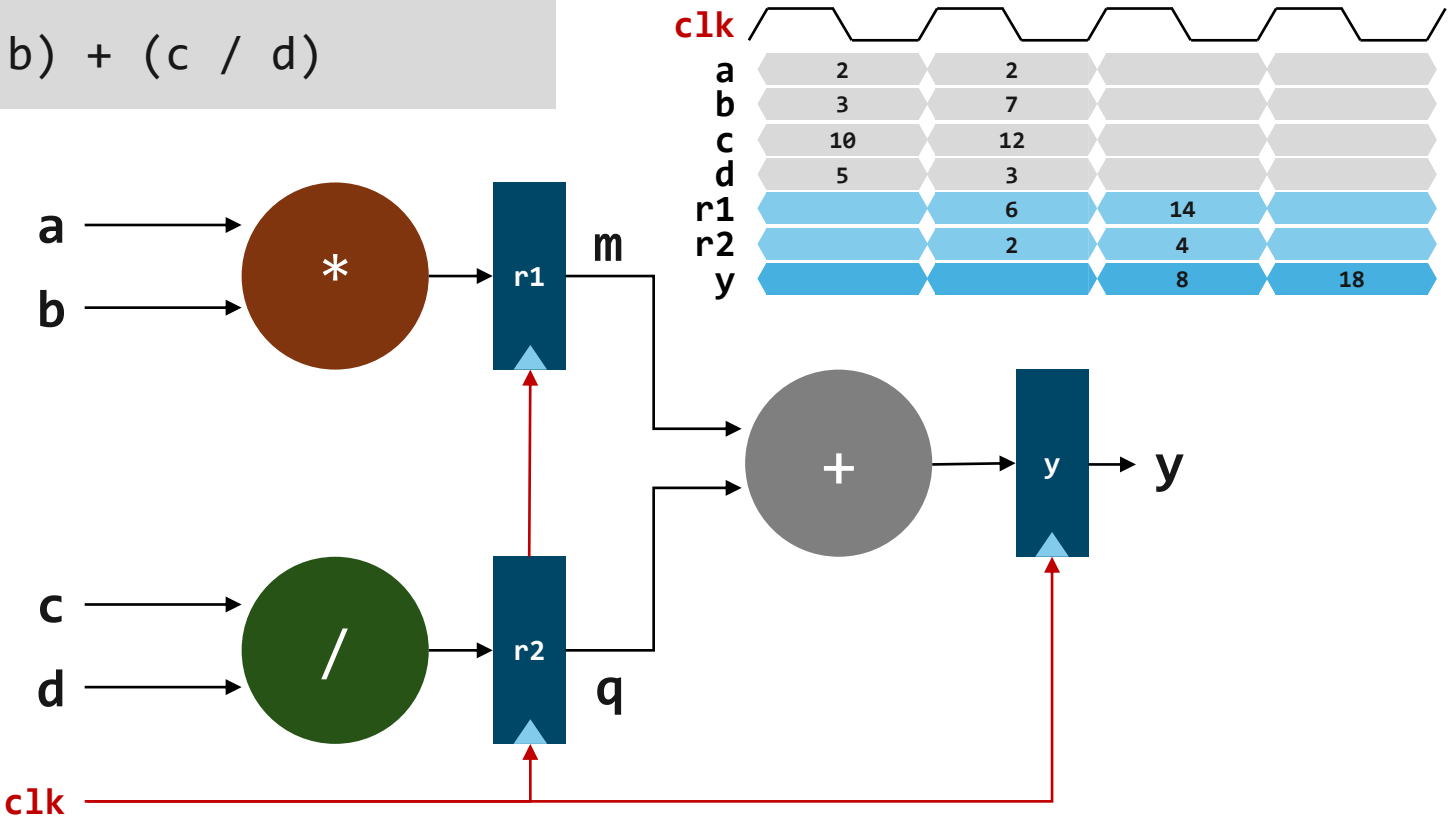
t_{PD} is the *maximum* cumulative propagation delay over all paths from inputs to outputs

Why do we need clock?

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

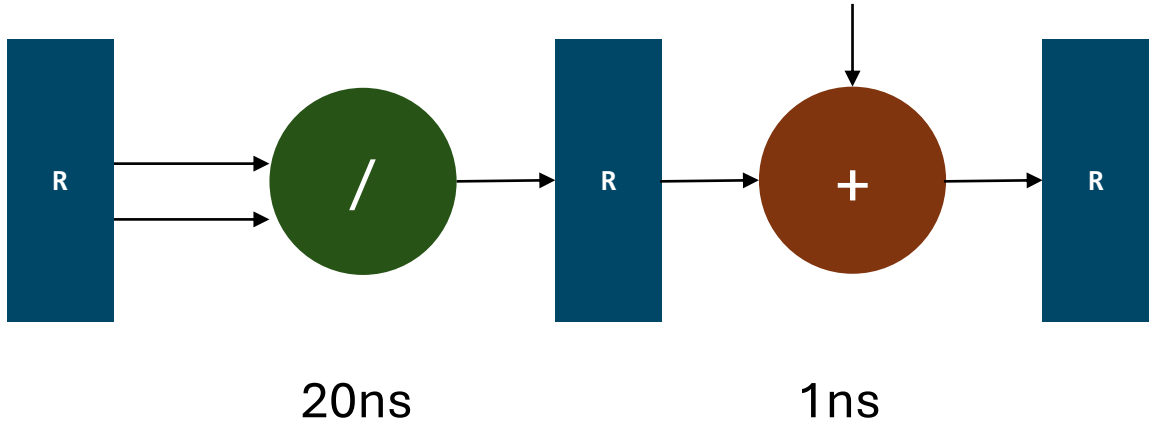


- Signals take time to run through wires
- Different delays in multiplier and divider
- m, d stabilize at different times
- We are not sure:
  - when to read y
  - when to change a,b,c,d
- Difficult to design anything complex



- 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.

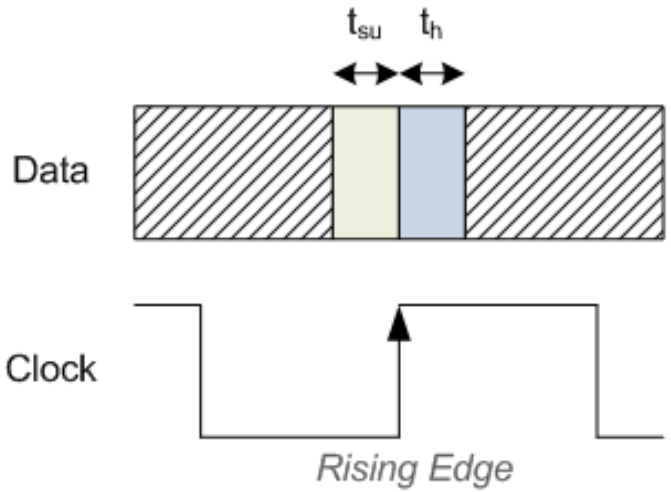
Timing closure



- Critical path: 20ns
- 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!

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 (yes!) 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 have to change the RTL.



nandland.com/lesson-12-setup-and-hold-time/

Daily Feedback

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