

Combinational and Sequential Logic

Lecture 02

Josh Brake
Harvey Mudd College

Administrative

- Grading discussion
- Example specs
- AI policy
- Office hours

Outline

- Combinational Logic Refresh
 - Sum of products form
- Sequential Logic Refresh
 - D latch vs. D Flip-flop
 - Different flavors of flip-flops
 - Counters
- Strategies to avoid the asynchronous trap

Learning Objectives

By the end of this lecture you should be able to...

- Recall how to go from a truth table to a sum-of-products Boolean equation.
- Recall the difference between D latches and flip flops.
- Recall the Verilog idioms for different types of flip flops.
- Recall the importance of designing synchronous sequential circuits.

Combinational Logic

Truth Table

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

Sum of products form

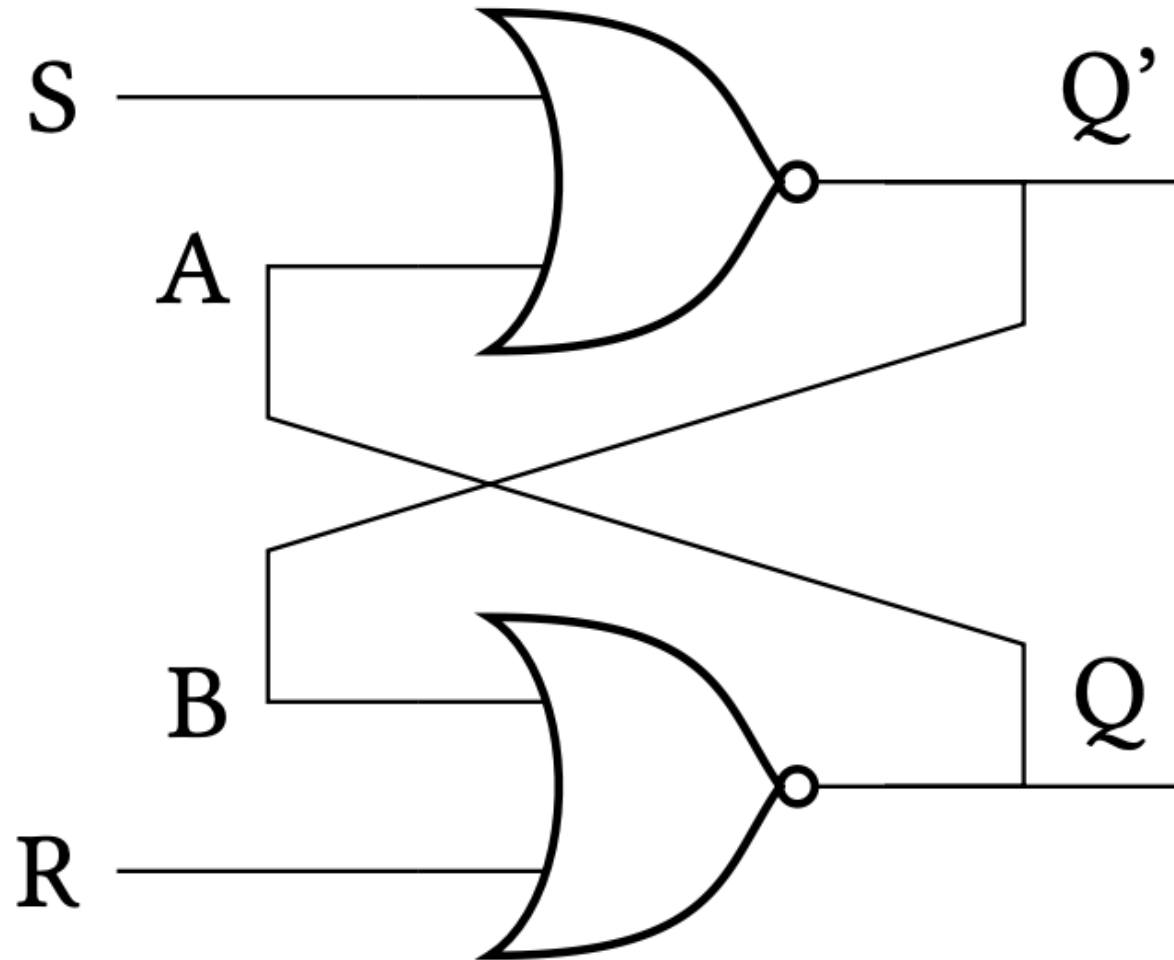
$$\begin{aligned} Y &= \bar{A}\bar{B}C + A\bar{B}\bar{C} + A\bar{B}C \\ &= (\bar{A} + A)\bar{B}C + A\bar{B}\bar{C} \\ &= \bar{B}C + A\bar{B}\bar{C} \end{aligned}$$

Combinational Logic

- Combinational: Outputs depend only on current inputs.
- Sequential: Outputs depends on current as well as older inputs (state).
- Make sure you know what you are trying to write!

D Latch and Flip-flop

SR latch with cross-coupled NOR gates



Truth Table

S	R	Q
0	0	Q
0	1	0
1	0	1
1	1	illegal

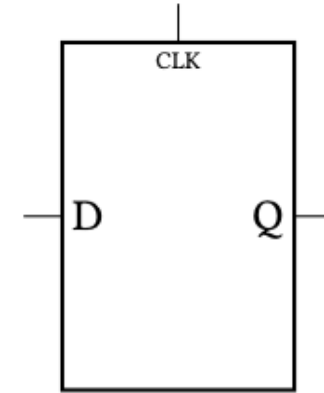
D Latch

- The D Latch is **transparent** when CLK is high.
- **Danger:** Not many good reasons to use these because they are asynchronous.

```
1 module latch(input logic clk,  
2               input logic [N-1:0] d,  
3               output logic [N-1:0] q);  
4     always_latch  
5       if (clk) q <= d;  
6 endmodule
```

Truth Table

CLK	D	Q
0	0	Q
0	1	Q
1	0	0
1	1	1



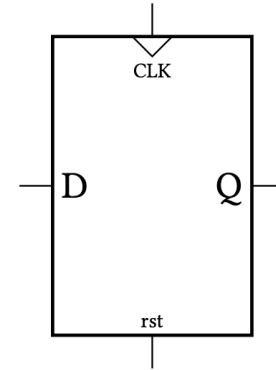
D Flip-flop

- Flip-flop is **edge-triggered**. So, we say Q gets D on the rising edge of the clock.
- Several different flavors: standard, with reset (async or sync), with enable.

```
1 // Register
2 module flop #(parameter N=4)
3     (input logic      clk,
4      input logic [N-1:0] d,
5      output logic [N-1:0] q);
6
7     // Standard D Flip-flop
8 endmodule
```

Code inside always block:

```
1 always_ff @(posedge clk)
2     q <= d;
```

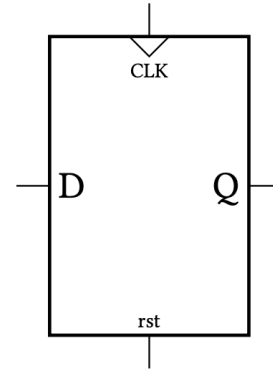


D Flip-flop: Asynchronous Reset

```
1 // Resettable Register with Asynchronous Reset
2 module flopr #(parameter N=4)
3     (input logic      clk,
4         input logic [N-1:0] d,
5         output logic [N-1:0] q);
6
7     // asynchronous reset
8
9
10 endmodule
```

Code inside always block:

```
1 always_ff @(posedge clk, posedge reset)
2     if(reset) q <= 0;
3     else     q <= d;
```

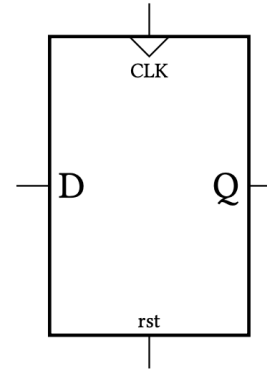


D Flip-flop: Synchronous Reset

```
1 // Resettable Register with Synchronous Reset and Enable
2 module flopenr #(parameter N=4)
3     (input logic      clk,
4       input logic [N-1:0] d,
5       output logic [N-1:0] q);
6
7     // synchronous reset with enable
8
9
10
11 endmodule
```

Code inside always block:

```
1 always_ff @(posedge clk)
2   if(reset)  q <= 0;
3   else if (en) q <= d;
```

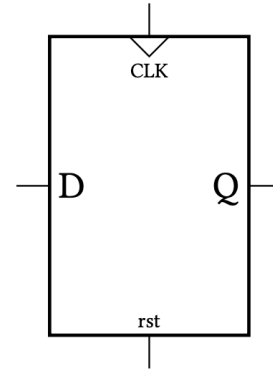


D Flip-flop: Synchronous Reset & Enable

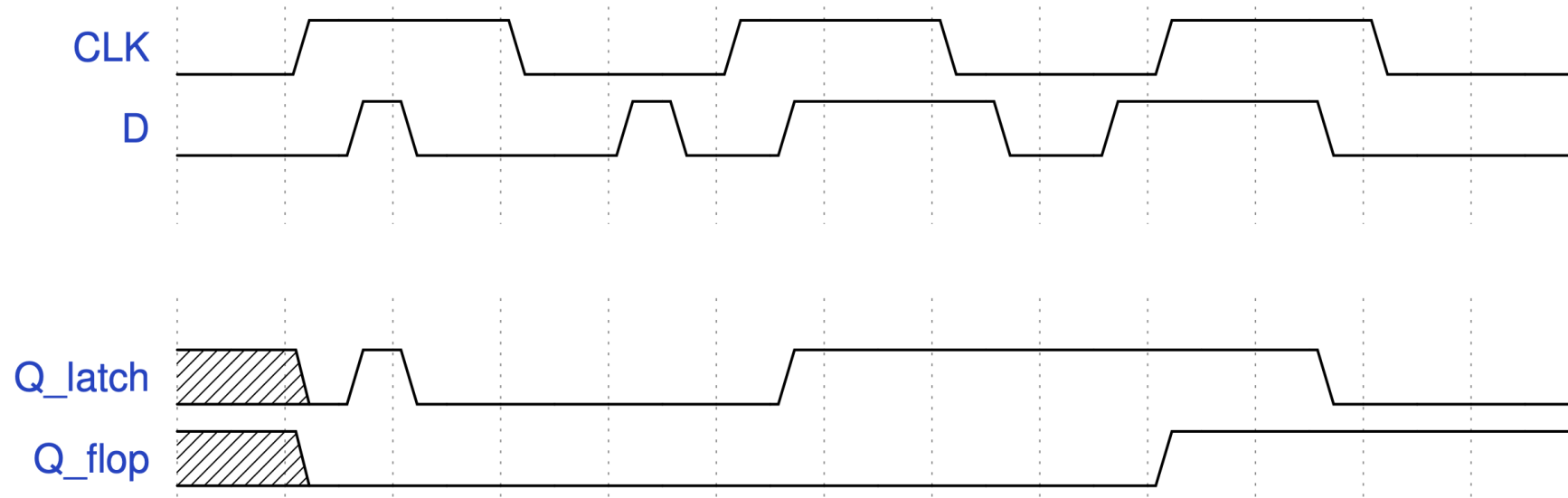
```
1 // Resettable Register with Synchronous Reset and Enable
2 module flopenr #(parameter N=4)
3     (input logic      clk,
4         input logic [N-1:0] d,
5         output logic [N-1:0] q);
6
7     // synchronous reset with enable
8
9
10
11 endmodule
```

Code inside always block:

```
1 always_ff @(posedge clk)
2     if(reset)    q <= 0;
3     else if (en) q <= d;
```

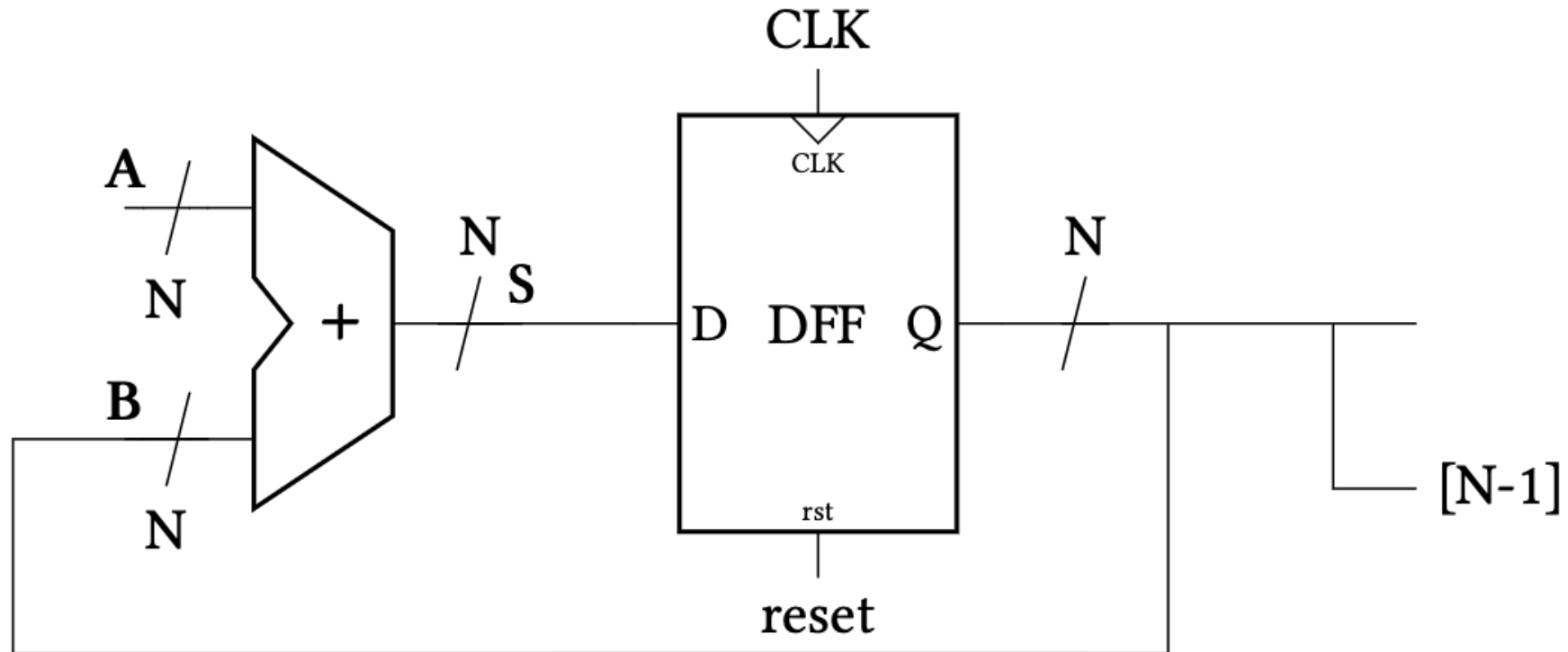


Comparing D Latch and D Flip-flop Waveforms



Counters

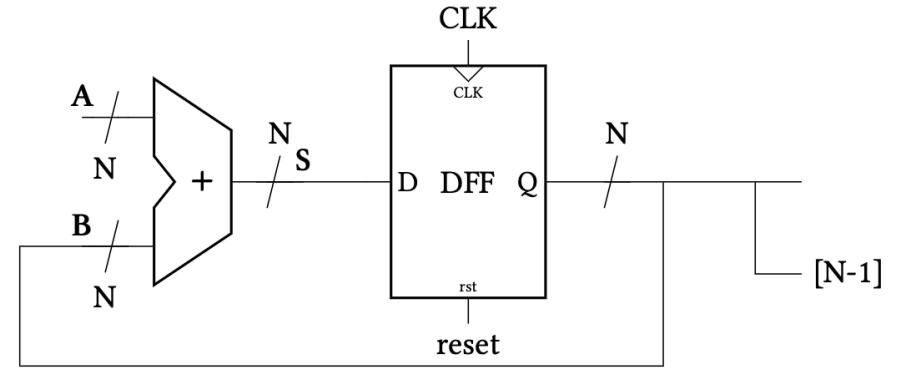
- Don't forget a reset! Otherwise, the counter may start at a random value.
- Be careful about using initial statements. These are not synthesizable (i.e., only work in simulation). Better to use a reset signal and have your testbench toggle it in sim.



Counter Verilog

```
1 // Counter
2 module counter #(parameter N=12)
3     (input  logic    clk, reset,
4      output logic [N-1:0] count);
5
6     // What goes here?
7 endmodule
```

```
1 always_ff @(posedge clk) begin
2     if(reset == 0) count <= 0;
3     else          count <= count + 1;
4 end
```



Running Logic Slower than the System Clock

What if I have some logic that I want to run at a slower clock speed?

- Clock divider?
 - Clock skew problems
 - Slows the whole circuit down or is asynchronous!
- Better solution? A strobe or pulse signal.

```
1 // Strobe signal
2 always_ff @(posedge clk)
3   ck_stb <= (counter == THRESHOLD-1'b1);
4
5 always_ff @(posedge clk)
6   if (ck_stb)
7     begin
8       // Build your logic this way instead
9       end else if (some_other_condition)
10      begin
11      end else if ...
```

See <https://zipcpu.com/blog/2017/06/02/generating-timing.html> for more detailed discussion.

Wrap Up

Combinational Logic

- Driven by truth table. Then use sum of products to derive Boolean expression and simplify. Or just let the synthesis tool simplify for you!
- Output is a function of current input only.

Sequential Logic

- Output is a function of current and past inputs.
- Past inputs are known as state.
- We use D Flip-flops to store state. Follow the dynamic discipline and constrain yourself to synchronous sequential design (edge-triggered flops only).
- Finite State Machines (FSMs) are the tool for helping us design synchronous systems.
- Beware the accidental creation of asynchronous circuits.

Announcements & Reminders

- Schedule check off time if you haven't already.
- Start on Lab 1.
- See tutorials on the website for some examples to get started with Radiant.
- More lab demo/office hour time this afternoon during the first hour of the lab block

Appendix: Strobe vs. Slow Clock

```
1 module strobe_example #(parameter N=3, parameter STROBE_THRESHOLD=3)
2 (
3     input  logic nreset,
4     output logic led_slow_clk, led_strobe
5 );
6
7     logic int_osc;           // Clock signal from internal oscillator output
8     logic [N-1:0] counter, strobe_counter; // Counter registers
9     logic ck_stb;          // Clock strobe signal
10    logic led_state;       // Register to store led_state.
11    logic slow_clk;       // Slow clock signal
12
13    // Internal high-speed oscillator
14    HSOSC #(.CLKHF_DIV(2'b01))
15        hf_osc (.CLKHFPU(1'b1), .CLKHFEN(1'b1), .CLKHF(int_osc));
16
17    ...
```

Appendix: Strobe vs. Slow Clock

```
1 ...
2
3 // Counter
4 always_ff @(posedge int_osc) begin
5     if(!nreset) counter <= 0;
6     else counter <= counter + 1;
7 end
8
9 assign slow_clk = counter[N-1];
10
11 // Strobe counter
12 always_ff @(posedge int_osc) begin
13     if(!nreset | ck_stb) strobe_counter <= 0;
14     else strobe_counter <= strobe_counter + 1;
15 end
16
17 ...
```

Appendix: Strobe vs. Slow Clock

```
1  ...
2
3  // Strobe generation
4  always_ff @(posedge int_osc) begin
5      ck_stb <= (strobe_counter == STROBE_THRESHOLD - 1'b1);
6  end
7
8  always_ff @(posedge int_osc) begin
9      if(!nreset) led_state <= 0;
10     else if(ck_stb) led_state = ~led_state;
11 end
12
13 // Assign LED output
14 assign led_slow_clk = slow_clk;
15 assign led_strobe = led_state;
16
17 endmodule
```

Appendix: Strobe vs. Slow Clock

