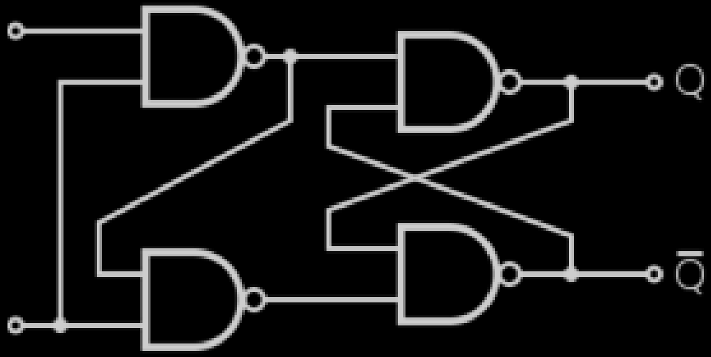
Lab 3 Preparation

Parts of Lab 3



- Part 1: Make a latch out of NAND gates.
  - Back to the breadboard!
- Part 2: Add shift operations to your ALU.
- Part 3: Make a shift register out of shift bits
  - Like making a ripple carry adder out of full adder units.

Shift Operations

- Logic Shift (left or right)
  - Verilog Operators: << , >>
  - `X >> N`
    - Produces a new vector with value of `X` shifted right
    - The `N` most-significant bits of the new vector are filled with zeros.
  - `X << N`
    - Produces a new vector with the value of `X` shifted left by `N` bits.
    - The `N` least significant bits of the new vector are filled with zeros.

```
wire [2:0] a, b;
wire c;
assign c = 1'b1;
assign a = (3'b011 >> 1'b1);
assign b = a << c;
```

- Example:
  - `3’b100 >> 2` will produce `3’b001`
  - `3’b100 << 2` will produce `3’b000`

Why is Shift Important?

- What is the sum of `01101101` and `01101101`?

`11011010`

Note what's happening here!

- Try the following:
  - `00110 << 1`
  - `00110 >> 1`

`A << N` results in $A*2^N$

`A >> N` results in $A/2^N$

Logic vs. Arithmetic Shift

- Arithmetic right shifts replicate the sign bit instead of using zero to fill in the most-significant bit(s).
  - Needed if dealing with signed numbers (e.g., 2's complement notation)
- Examples:
  - Arithmetic Right Shift:
    - `3’b100 >>> 2` will produce `3’b111`
  - Logic Right Shift:
    - `3’b100 >> 2` will produce `3’b001`

Sign Extension – Why?

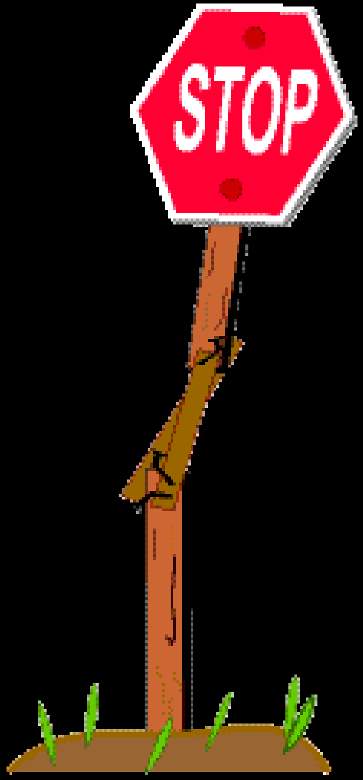
- Used in binary arithmetic when we want to increase the # bits used to represent a number while maintaining its sign/value.

- Let's say you want to add these two signed numbers . How would you do that?
  - `0011_1101`
  - `0110`
  - What if the second number was `1110` instead?

Sign extend this one!

Sign Extension – How?

- You need to replicate the sign (i.e. the most significant bit in 2's complement form)
  - Replicate 0 for positive numbers
  - Replicate 1 for negative numbers



Implementing D-FF in Verilog

```
module my_dff (clk, reset_n, d, q);
    input clk;
    input d;
    input reset_n;
    output q;
    reg q;

    always @(*) begin
        q <= d;
    end
endmodule
```

Need to change this so that `q` follows `d` on the positive or negative edge of the clock.

The (<=) operator is for non-blocking assignments. Use this for sequential circuits.

Implementing D-FF in Verilog

```
module my_dff (clk, reset_n, d, q);
    input clk;
    input d;
    input reset_n;
    output q;
    reg q;

    always @(posedge clk) begin
        q <= d;
    end
endmodule
```

The sensitivity list is now correct. We'll fix the body of the always block next.

Could use **negedge** keyword for negative edge-triggered behaviour.

D Flip-Flop w/ a Reset Signal

- Reset: This is how you put your hardware in a known initial state!

```
always @(posedge clk) begin
        if (reset_n == 1'b0)
            q <= 0;
        else
            q <= d;
    end
```

`if-else` used within an `always` block. Synthesizes to a multiplexer.

Note: Reset is usually **active-low** (meaning it triggers when `reset_n` is low). Here we have an **active-low synchronous reset** signal.

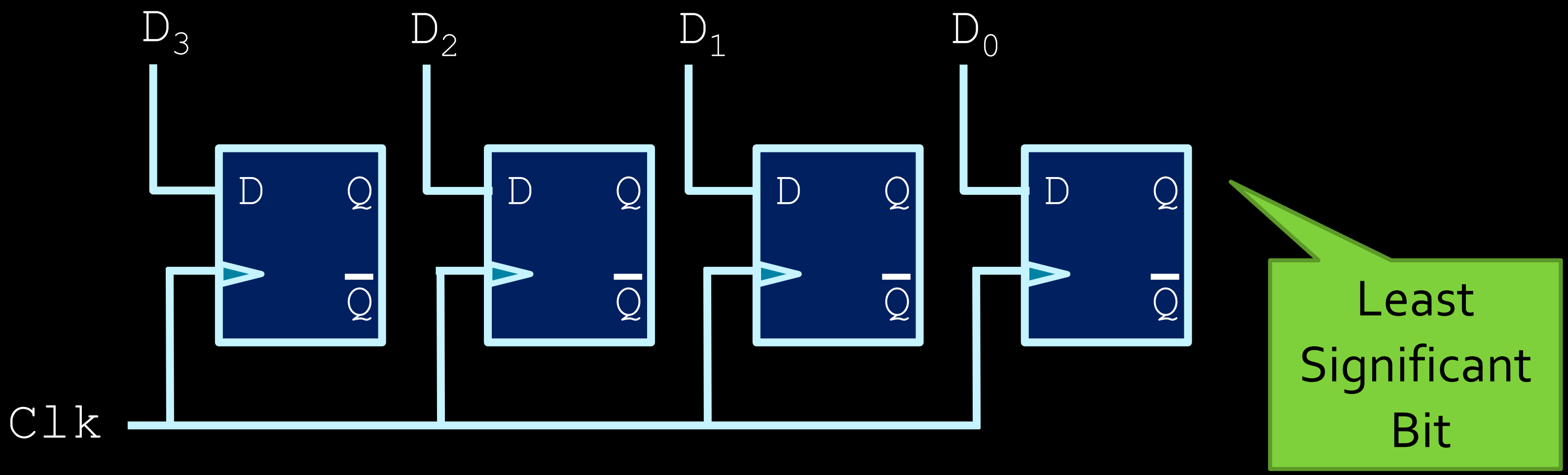
When you test/demo your design

- Synchronous Reset
  - Needs to be `0` @ the active clock edge.

- Be careful with KEYs and active low signals.
  - A KEY on the DE1-SoC board is 0 when pressed.
  - Here's an example
    - Assume I have `KEY[0]` as my clock and `KEY[1]` as a signal that is active-low.
    - How can you test for a scenario where `KEY[1]` is low at the positive edge of your clock?
      - Think about how you will need to press these two keys.

Load register

- N-bit number => n D-flipflops with same clock signal
- You can load a register's value (all bits at once), by feeding signals into each flip-flop:
  - In this example: a 4-bit load register.



Design Guidelines

- Combinational Circuits (e.g., `always @(*)`)
  - Use blocking assignment statements `=`
- Sequential Circuits (e.g., `always @(posedge clock)`)
  - Use non-blocking assignment statements `<=`
- Don't mix assignment types in the same `always` block! ☺
- Order of `always` blocks doesn't matter; neither does the order of `always` blocks and `assign` statements.