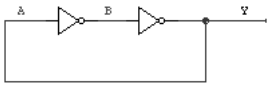
Today: Sequential Logic and Verilog

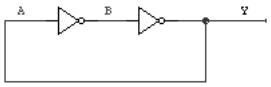
- Latches and Flip-flops
  - Review of latches
  - Verilog and sequential logic
- Shift Register Example in Verilog (which works in DW)
  - Sample code which **WORKS**
- More Advanced Verilog Features
  - $display() and $time() statements

Cascaded inverters



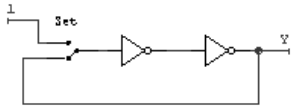
- **IF** A is "1", then B is "0", which forces A again to "1", which forces B again to "0", and so on. Thus, the ouput Y is "1", and stays "1" forever. This is a steady state (contrast this to the oscillator you did in your assignment).
- Similarly, **IF** A is "0", then Y will stay at "0" forever.
- Wow, this looks like a bit of memory (if you ignore the magical **IF**…)
- But, wait a second… How can this circuit store a value forever, it doesn't seem to be using any power: after all we are not applying any inputs…
- This is a common fallacy! Remember, we don't draw the power supplies, but they are ASSUMED to be there.

Cascaded inverters (cont'd)



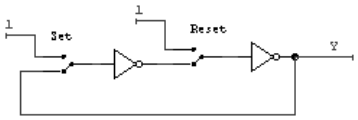
- In describing this circuit, we used the magical **IF**. But what happens if you build this in the lab? Which state will it go in? Will it go into one of the two states described before (Y = "0", or Y = "1")? Or is there a possibility that A and B will just remain undefined (maybe both will stay at 2.5 Volts…)
- Well, in theory this circuit has an undefined behavior. But if you build it, it WILL go into one of the two states. Why?
  - Say the circuit lands in an undefined state, maybe A=2.5 Volts, and B = 2.5 Volts.
  - The only way you'll stay in this state is if you are in perfect equilibrium.
  - But then, any perturbation in A or B (caused maybe by electromagnetic radiation from the moon…) will cause your circuit to spiral towards one of the two states.
- What's worse is that we **don't know** what state it will go in! That's not good… We need a way to guarantee that we land in the state we **WANT**.

Cascaded inverters with "Set"

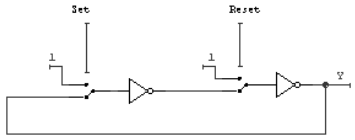


- Think of the switch as a push button, with the default state as shown. When you activate the button, it connects a "1" to the input of the first inverter. When you release the button, it bounces back to the default state.
- When you push the button, the loop is broken. The effect is that it sets Y to "1". When we release the button, the value of "1" at the output remains there.
- We'll call this a "Set" (we're setting the output…). This is good, but this means we can only store a "1". What about a "0"?

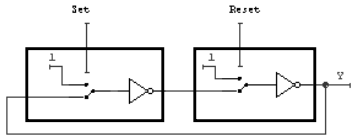
Cascaded inverters with "Set" and "Reset"



- Add a "Reset" switch, which makes the output Y go to 0. Again, when the switch is released, Y stays at 0.
- What happens if none of the switches are pressed? Well, the value of Y that was there before will stay there. This is called a "Hold".
- So, we have a one bit of memory that we can set, reset, or just leave as is.
- Now, pushing switches is not a scalable solution. We need to control our memory cell with

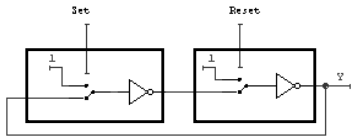


Cascaded inverters with "Set" and "Reset"

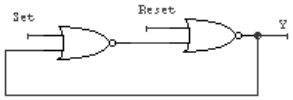


- Now, take a look at the black boxes above. Each has two inputs. When the first input is 1, the output should go to 0, whereas when the first input is 0, then the output should be the negation of the second input. What is that?

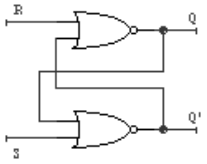
Cascaded inverters with "Set" and "Reset"



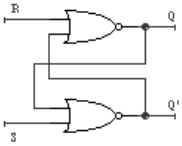
- The black boxes are NOR! So we can redraw the circuit with NORs instead of the black boxes:



Rearranging
and renaming



Cross-coupled NOR gates



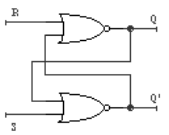
- This is called an RS latch. We can build a table that shows how this circuit behaves:

S	R	Q
0	0	hold
0	1	0
1	0	1
1	1	???

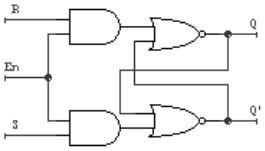
- What happens in the 1, 1 state? At first, both Y and Y' will be 0 (that's already a problem, since that means we can't call them Y and Y'...). But, then, if you "release" the S and R inputs (thus, setting S=0, and R=0), Y and Y' will oscillate. Forever!
- So… **We DISALLOW this state.**

RS Latch

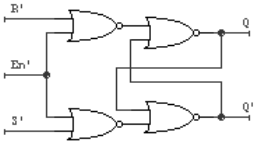
- R-S latch always samples its inputs.
- That means that a glitch in the inputs is fatal (called the 1's catching problem). Wouldn't it be great if the latch didn't always sample its inputs? That way, we wouldn't be required to build hazard free circuits (which lessens not only the design time, but also the hardware).
- First attempt at solving this: use an enable signal.
  - When latch is enabled, Set and Reset work.
  - When latch is disabled, hold.



Note inverted inputs



Bubble introduction and propagation



Incorrect Flip-flop in Verilog

- Use always block's sensitivity list to wait for clock to change

```
module dff (CLK, d, q);

    input  CLK, d;
    output q;
    reg    q;

    always @(CLK)
       q = d;

endmodule
```

Not correct! Q will change whenever the clock changes, not just on the edge.

Correct Flip-flop in Verilog

- Use always block's sensitivity list AND the posedge keyword to wait for clock edge

```
module dff (CLK, d, q);

    input  CLK, d;
    output q;
    reg    q;

    always @(posedge CLK)
       q = d;

endmodule
```

More Flip-flops

- Synchronous/asynchronous reset/set
  - single thread that waits for the clock
  - three parallel threads – only one of which waits for the clock

Synchronous

```
module dff (CLK, s, r, d, q);
    input  CLK, s, r, d;
    output q;
    reg    q;

    always @(posedge CLK)
        if (r)    q = 1'b0;
        else if (s) q = 1'b1;
        else        q = d;

endmodule
```

Asynchronous

```
module dff (CLK, s, r, d, q);
    input  CLK, s, r, d;
    output q;
    reg    q;

    always @(posedge r)
        q = 1'b0;
    always @(posedge s)
        q = 1'b1;
    always @(posedge CLK)
        q = d;

endmodule
```

Sample Shift Register

```
module ShiftReg(Clk, ShiftIn,
                D0, D1, D2, D3,
                S0, S1,
                Q0, Q1, Q2, Q3);

  input        Clk;
  input        ShiftIn;
  input        D0;
  input        D1;
  input        D2;
  input        D3;
  input        S0;
  input        S1;
  output  Q0;
  output  Q1;
  output  Q2;
  output  Q3;

  reg QO, Ql, Q2, Q3;

  reg[3:0] indata;
  reg[3:0] out;
  reg[3:0] nextout;
  reg[1:0] sel;

  assign indata = {D3,D2,D1,D0};
  assign sel  = {S1,S0};
```

```
 // State Definitions
  `define  Hold           2'b00
  `define  Load           2'b01
  `define  ShiftRight     2'b10
  `define  ShiftLeft      2'b11

  /* clock driven "state" transitions */
  always @(posedge Clk) begin
    out = nextout;
    $display("Input  is %b, Sel is %d, Output is %b",
              indata, sel, out);
  end

  /* combinational logic to determine next output */
  always @(out or sel or indata) begin
    case(sel)
      `Hold: nextout = out;
      `Load: nextout = indata;
      `ShiftRight: nextout = {1'b0, out[3:1]};
      `ShiftLeft: nextout = {out[2:0], 1'b0};
    endcase
  end

  assign Q3  = out[3];
  assign Q2  = out[2];
  assign Q1  = out[1];
  assign Q0  = out[0];

endmodule
```

$display and $time statements

- Documentation in the online manual (p. 56)
- Doesn't synthesize to anything!
- Formats similar to printf() in C
  - %h Hex, %d Decimal, %o Octal, %b Binary, %% Display a "%" sign
- Examples of $display()
  - $display("output %d is %h", i, vec[i]);
  - $display("%d%% completed", (count * 100) / max_count);
- The $time function returns system simulation time as a 32-bit integer
  - $display("Got an event at time %d", $time);