

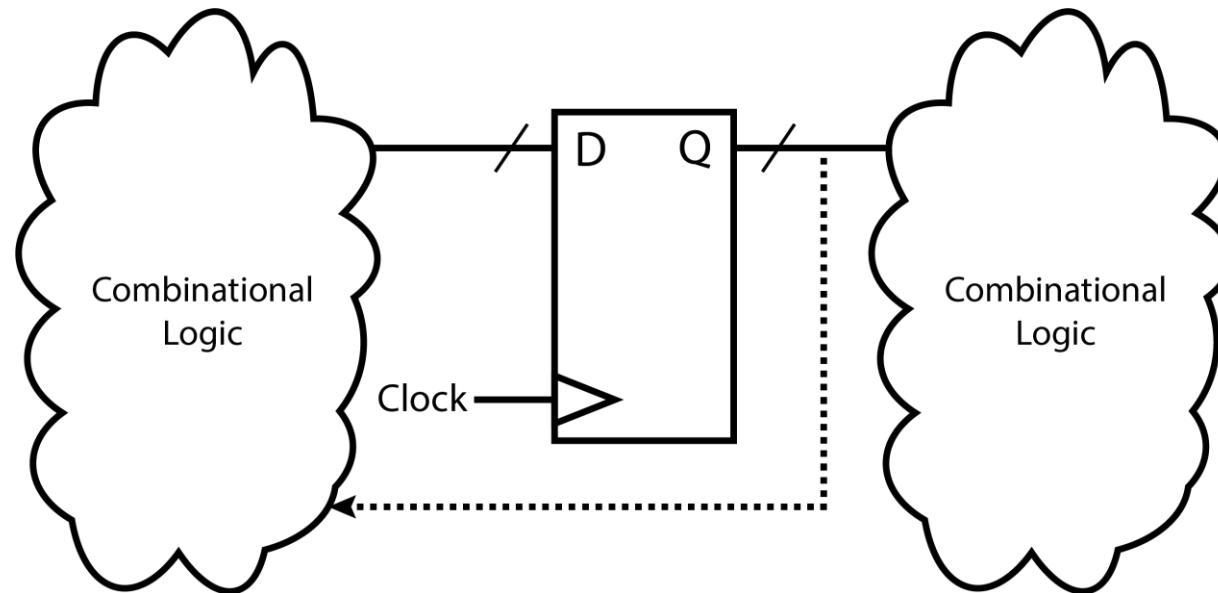
Discussion Section 2

Sean Huang

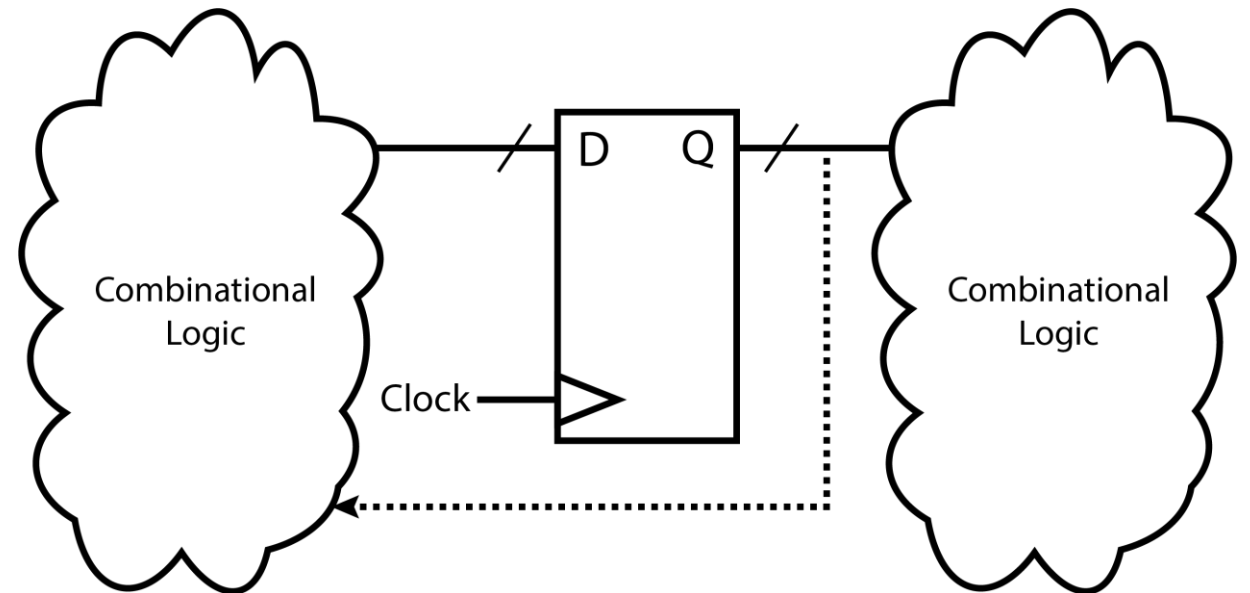
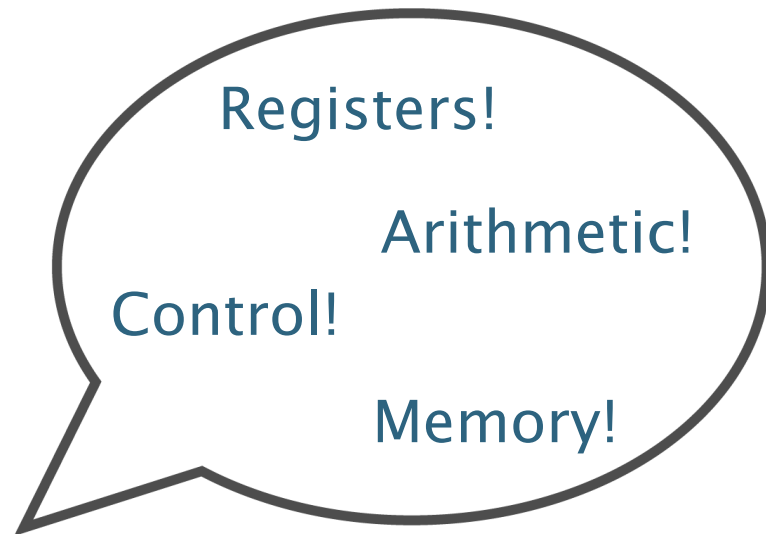
January 29, 2021

Hardware Description Languages (HDL)

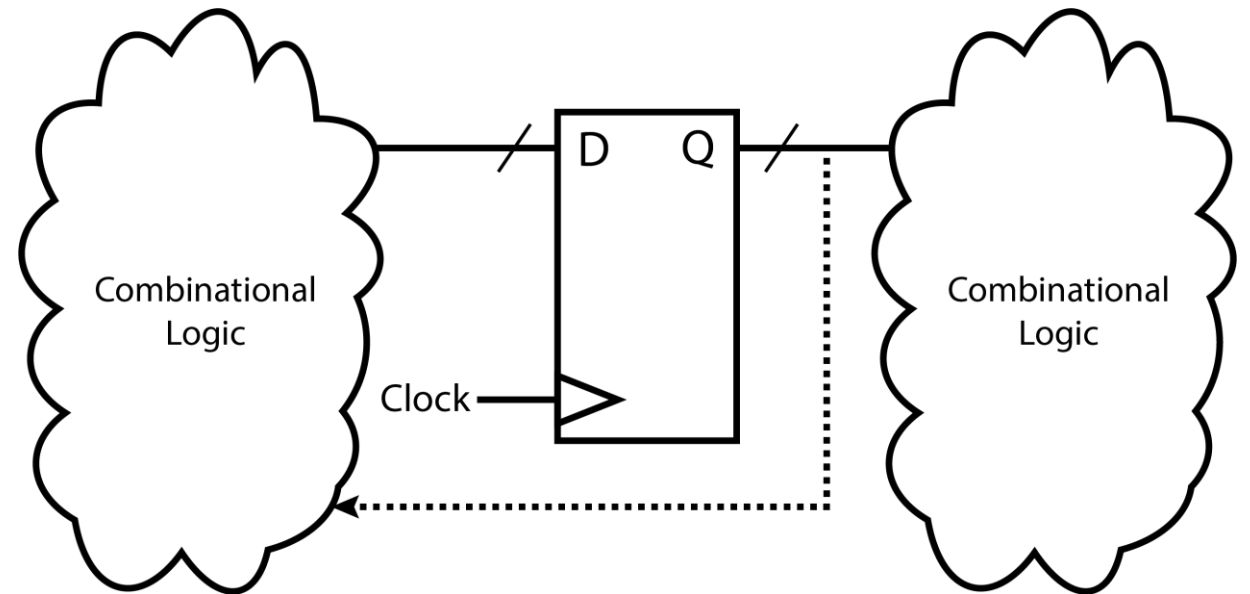
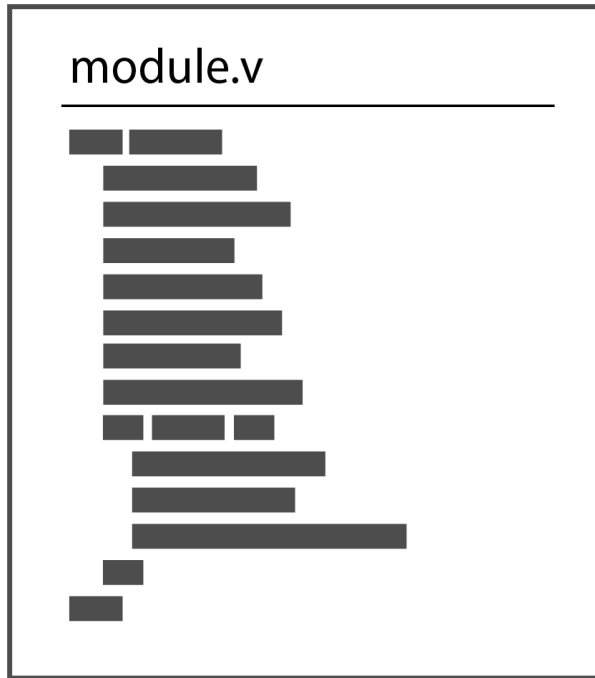
How do we describe this?



Hardware Description Languages (HDL)

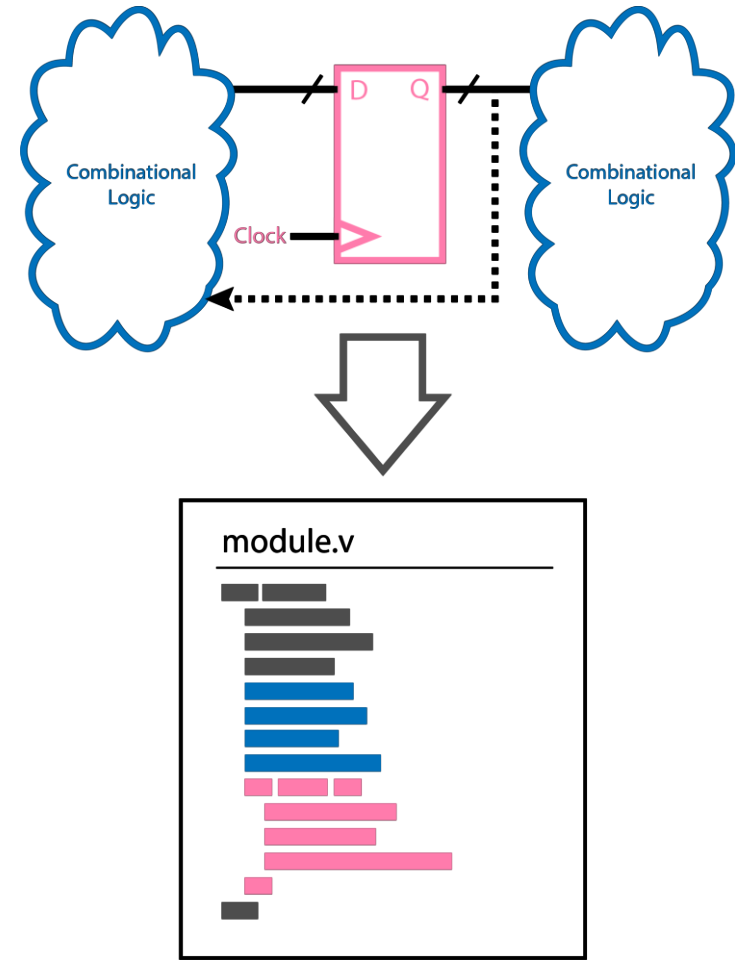


Hardware Description Languages (HDL)



Hardware Description Languages (HDL)

- Standard for describing and representing digital systems
- Contains all information necessary to build entire digital system
- Apply RTL abstraction for combinational and state elements



Hardware Description Languages (HDL)

- Verilog ← We'll be using this one in the class!
- VHDL
- SystemVerilog
- BlueSpec
- Chisel

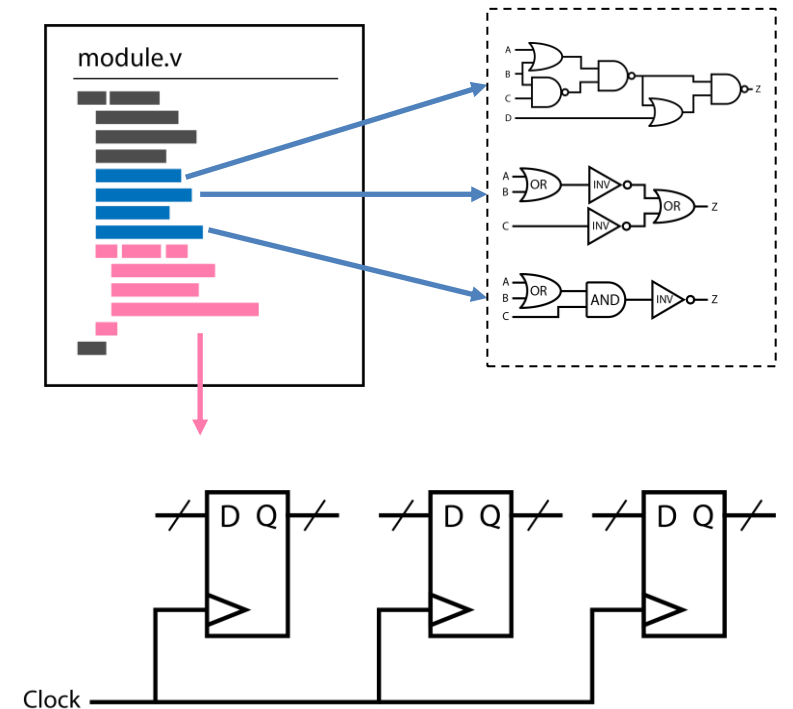
Digital Design in Verilog

Verilog Basics

- Not a programming language!!!
 - Only the syntax is based on C for familiarity
 - Circuits are not programs and follow different rules
- Learn Verilog from scratch
 - Think about it from a circuit perspective
 - Not "programming a circuit"
 - Writing a description

Verilog Basics

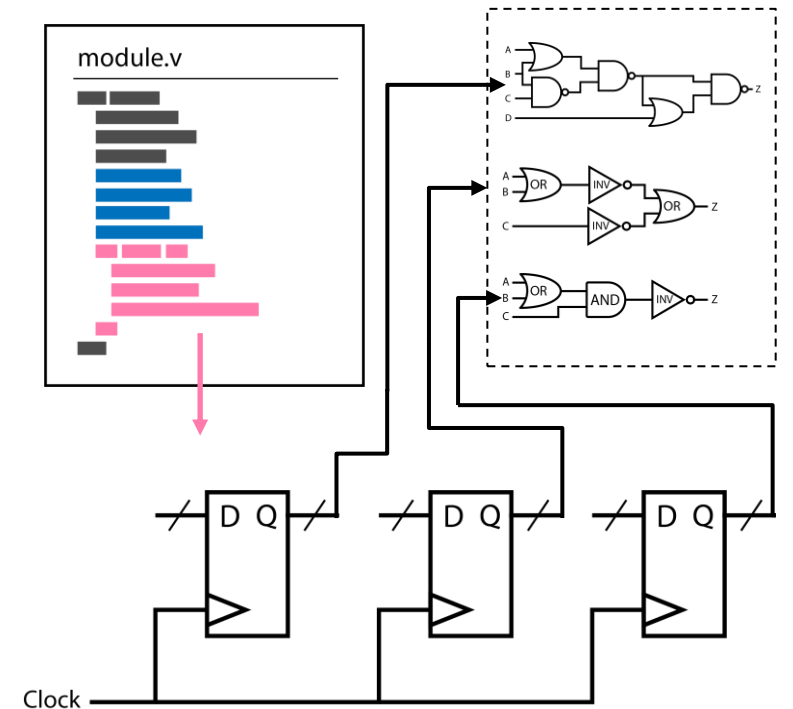
- Examples of differences
 - Combinational logic
 - All combinational blocks are always running in parallel
 - Output updates immediately* with input
 - Just because something is assigned at a later line doesn't mean it runs later!
 - Sequential logic
 - Many registers can update on same clock edge



* In RTL simulation. In gate-level simulation, there will be some gate delay before the output updates

Verilog Basics

- Examples of differences
 - Combinational logic
 - All combinational blocks are always running in parallel
 - Output updates immediately* with input
 - Just because something is assigned at a later line doesn't mean it runs later!
 - Sequential logic
 - Many registers can update on same clock edge
 - Usually drive combinational blocks
 - Need to be careful not to have conflicts!



* In RTL simulation. In gate-level simulation, there will be some gate delay before the output updates

Inputs, Outputs, Wires, and Regs

- Signals in Verilog are of 2 flavors
 - `wire`
 - `reg`
 - Used in always blocks
- I/O declared at beginning of module
 - Follow same signal types (wires/regs)
 - Not specifying `reg` implies `wire`
 - Inputs are wires
 - `input reg` doesn't really make sense
- Internal wires and regs are declared after I/O
 - Not visible outside module

```
module Example (a, b, c, status, s);  
  input a;  
  input b;  
  input c;  
  output reg status;  
  output s;  
  
  wire internal;  
  reg internal_state;
```

Wire vs. Reg

wire

- Continuous assignment
`assign internal = a & b;`
- Interconnections between modules

reg

- These are not registers themselves!!!
- Signals in `always` blocks must be reg
- Actual registers are made by instantiating modules from the register library

The `always @` block

- Block delineated by `always @(…)` `begin … end` keywords
- The `@ (…)` indicates the *sensitivity list* of the `always` block
 - Signals listed within the parentheses are those the `always` block is sensitive to
 - Indicates assignments in `always` block take effect when signals in sensitivity list update
 - Otherwise, signals in `always` block hold last output
 - Hardware is defined to only depend on signals in sensitivity list

```
always @ ( … ) begin
    …
end
```

The `always @ block` - Combinational Logic

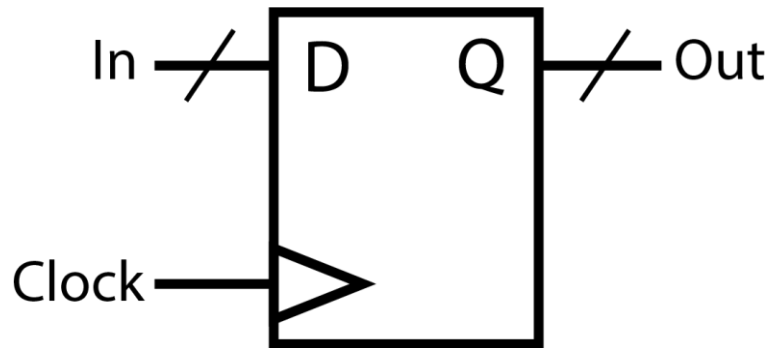
- Make sure all dependent signals are in sensitivity list
 - Signals on right side of assignment
 - Signals in conditional statements
- Missing sensitivities can result in unexpected behavior!
- `always @(*)`
 - Sensitivity list inferred from contents of block
 - Use this when using `always` blocks for combinational logic

```
wire a, b, s;  
reg out;
```

```
always @(*) begin  
    if (s) begin  
        out = a;  
    end else begin  
        out = b;  
    end  
end
```

The always @ block - Sequential Logic, Part 1

- The classic way to denote a register
- **posedge** sensitivity
 - Sensitive to rising edge of clk
 - Output updates only at 0→1 transition of clk
 - Assigns Q = D when this condition is met



```
wire clk, D;  
reg Q;
```

```
always @ ( posedge clk )  
begin  
    Q <= D;  
end
```


The always @ block - Sequential Logic, Part 2

- Registers are common subcircuits
 - Pretty much digital system in itself
- Define dedicated register module
 - Has I/O like higher level modules
 - Same **always** block definition as before
 - In this case there is a synchronous reset
- Clearer where registers are in top level design

```
module REGISTER_R(q, d, rst, clk);  
    parameter N = 1;  
    parameter INIT = {N{1'b0}};  
    output reg [N-1:0] q;  
    input [N-1:0] d;  
    input rst, clk;  
    always @(posedge clk)  
        if (rst) q <= INIT;  
        else q <= d;  
endmodule
```

Notice this parameter definition

The always @ block - Sequential Logic, Part 2

```
module Counter(val, en, reset, clk);  
    input en, reset, clk;  
    output [3:0] val;  
    wire [3:0] nxt;  Overwrite N=1 from module definition with new bit width  
    REGISTER_R #(.N(4)) state (.clk(clk), .d(nxt), .q(val), .reset(reset));  
    assign nxt = val + 1;  
endmodule
```

- Register clearly instantiated
 - Cleaner than having **always** blocks everywhere
 - Design kits may have predefined register standard cells

Multiple Assignments

- Cannot continuously assign wire to two other wires

```
wire a, b, c;
```

```
assign a = c;
```

```
assign a = b;
```

Don't do this!

- Two wires driving 1 wire?
 - Ambiguous what final value will be
 - No “half values” in digital!

- Can assign different values to reg at different points in `always` block

```
wire b, c, clk;
```

```
reg a;
```

```
always @ ( posedge clk ) begin
```

```
  a <= c;
```

```
  if (b == 1'b1) begin
```

```
    a <= b;
```

```
  end
```

```
end
```

This is fine.

- Last value overrides all previous values
- Can be used to set default values for registers

generate Loops

- Not a C for loop!
 - Not describing iterations of logic operation
 - Shorthand for repeating the same logic circuit in the design
- Looping operations requires you to design the control logic yourself

```
genvar i;  
wire [3:0] a;  
wire [7:0] b;  
generate  
  for(i = 0, i < 3, i = i+1) begin:bit  
    shift_r inst(.a(a[i]), .b(b[i]), .c(b[i+1]));  
  end  
endgenerate
```

Equivalent to



```
wire [3:0] a;  
wire [7:0] b;  
shift_r inst_bit_0(.a(a[0]), .b(b[0]), .c(b[1]));  
shift_r inst_bit_1(.a(a[1]), .b(b[1]), .c(b[2]));  
shift_r inst_bit_2(.a(a[2]), .b(b[2]), .c(b[3]));  
shift_r inst_bit_3(.a(a[3]), .b(b[3]), .c(b[4]));
```

Simulation in Verilog

Verilog Simulation

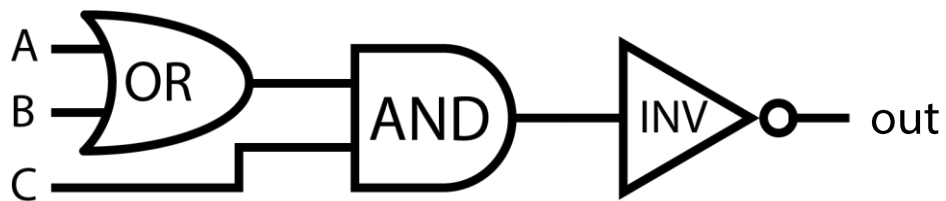
- Slightly more programmatic than the hardware description itself
- Can use some more software-like constructs
 - for loops
 - Subroutine calls
 - Print statements

Verilog Simulation

- Testbenches
 - Instance your module as Device Under Test (DUT)
 - Use **initial** blocks to make test vectors
- **initial** blocks
 - “snapshots” of simulation
 - Separate with delay timesteps between timeframes
 - Denoted with #N timesteps

Verilog Simulation

- Here's simple testbench for 4 test inputs to an OAI circuit



```
`timescale 1ns / 1ns

module oai_tb;

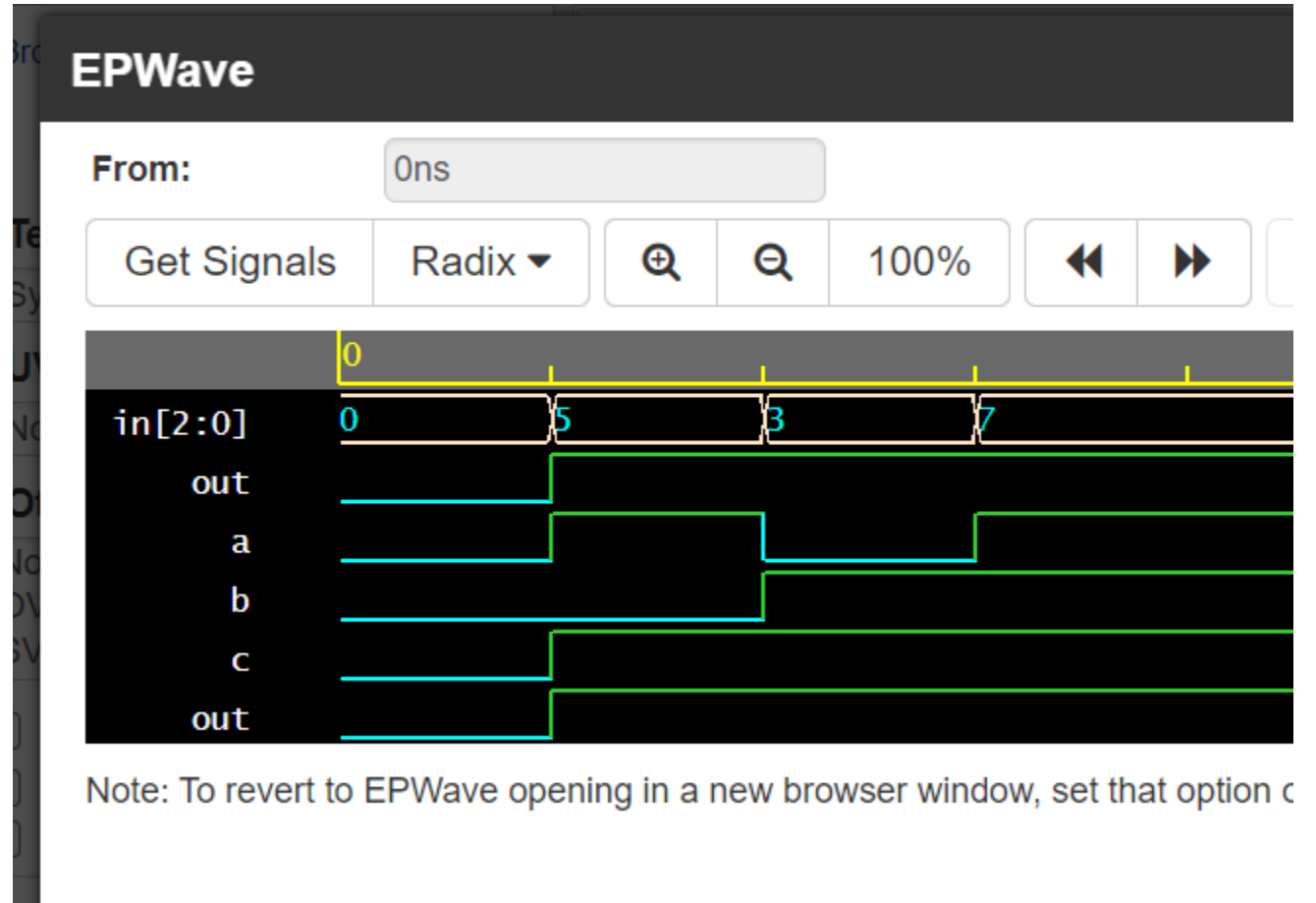
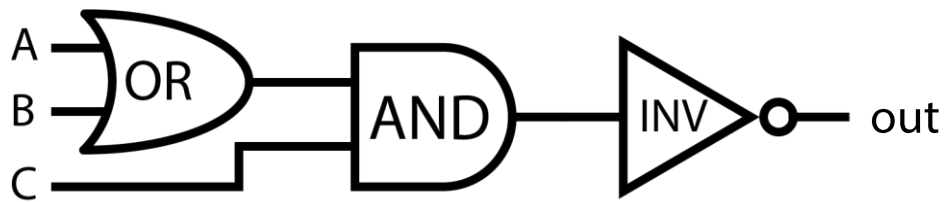
    reg [2:0] in;
    wire out;

    OAI dut (.a(in[2]), .b(in[1]), .c(in[0]), .out(out));

    initial begin
        $dumpfile("dump.vcd");
        $dumpvars;
        in = 3'b000;
        $strobe("a:%b, b:%b, c:%b, out:%b", in[2], in[1], in[0], out);
        #1;
        in = 3'b101;
        $strobe("a:%b, b:%b, c:%b, out:%b", in[2], in[1], in[0], out);
        #1;
        in = 3'b011;
        $strobe("a:%b, b:%b, c:%b, out:%b", in[2], in[1], in[0], out);
        #1;
        in = 3'b111;
        $strobe("a:%b, b:%b, c:%b, out:%b", in[2], in[1], in[0], out);
        #1;
        $finish();
    end
endmodule
```

Verilog Simulation

- Here's simple testbench for 4 inputs to an OAI circuit
- Outputs can be visualized as a waveform



Useful Commands

- `$strobe("format string", values);`
 - Prints values to console. Executed at end of current cycle (i.e. "simulation time")
- `$time`
 - Get current simulation time
- `$display("format string", values);`
 - Similar to `$strobe` but not guaranteed to execute at end of cycle
- `$finish()`
 - End simulation

Format String

| | |
|----------|------------------------|
| %d or %D | Decimal format |
| %b or %B | Binary format |
| %h or %H | Hexadecimal format |
| %o or %O | Octal format |
| %c or %C | ASCII character format |
| %v or %V | Net signal strength |
| %m or %M | Hierarchical name |
| %s or %S | As a string |
| %t or %T | Current time format |

Printing Every Cycle

```
module oai_tb;

    reg [2:0] in;
    wire out;

    OAI dut (.a(in[2]), .b(in[1]), .c(in[0]), .out(out));

    initial begin
        $dumpfile("dump.vcd");
        $dumpvars;
        in = 3'b000;
        #5;
        in = 3'b101;
        #5;
        in = 3'b111;
        #5;
        in = 3'b110;
        #5
        in = 3'b000;
        $finish();
    end
    initial begin
        forever begin
            $strobe("a:%b, b:%b, c:%b, out:%b", in[2], in[1], in[0], out);
            #1;
        end
    end
endmodule
```

outputs



```
a:0, b:0, c:0, out:0
a:0, b:0, c:0, out:0
a:0, b:0, c:0, out:0
a:0, b:0, c:0, out:0
a:1, b:0, c:1, out:1
a:1, b:0, c:1, out:1
a:1, b:0, c:1, out:1
a:1, b:0, c:1, out:1
a:1, b:1, c:1, out:1
a:1, b:1, c:1, out:1
a:1, b:1, c:1, out:1
a:1, b:1, c:1, out:1
a:1, b:1, c:1, out:1
a:1, b:1, c:0, out:0
a:1, b:1, c:0, out:0
a:1, b:1, c:0, out:0
a:1, b:1, c:0, out:0
a:1, b:1, c:0, out:0
```

Monitoring: An Alternative

```
module oai_tb;

  reg [2:0] in;
  wire out;

  OAI dut (.a(in[2]), .b(in[1]), .c(in[0]), .out(out));

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars;
    in = 3'b000;
    #5;
    in = 3'b101;
    #5;
    in = 3'b111;
    #5;
    in = 3'b110;
    #5
    in = 3'b000;
    $finish();
  end
  initial begin
    $monitor("a:%b, b:%b, c:%b, out:%b", in[2], in[1], in[0], out);
  end
endmodule
```

outputs



```
a:0, b:0, c:0, out:0
a:1, b:0, c:1, out:1
a:1, b:1, c:1, out:1
a:1, b:1, c:0, out:0
```

Simulators

- Synopsys VCS
 - Used for ASIC Lab
 - On INST machines
- ModelSim-Altera
 - On INST machines
 - Educational Version download available for Windows
- Vivado Simulator
 - Used in FPGA Lab
 - On INST machines
 - Free version available for Windows and Linux (will need VM for Mac)
- ...and many more!

Simulators

- EDA Playground (<https://www.edaplayground.com/>)
 - HIGHLY recommended for this homework!
 - Free web-based simulator
 - Can use a variety of simulation engines for many different HDLs
 - Built-in waveform viewer
 - Using proprietary simulators (e.g. VCS, Xcelium, etc.) requires registration

EDA Playground Example

- Exhaustively test a 4-bit wrap-around counter
 - Count from 0000 to 1111
 - Overflow wraps around to 0000 at next cycle

```
module Counter #(
    parameter bits = 1)(
    input reset,
    input clk,
    output [bits-1:0] val
);

    wire [bits-1:0] next = val + 1;

    REGISTER_R #(.N(bits), .INIT(0)) count_reg(.d(next), .q(val), .rst(reset), .clk(clk));

endmodule
```

EDA Playground Example – For Loop

```
`timescale 1ns/1ps

module counter_testbench;
    // Initial signal and parameter definitions
    parameter bits = 4;
    reg reset, clk;
    wire [bits-1:0] val;

    // Instantiate DUT
    Counter #(.bits(bits)) dut (.reset(reset), .clk(clk),
    .val(val));

    // Set initial clk state
    initial clk = 0;
    integer i = 1;

    // Every 1 time step, toggle clk
    always #(1) clk <= ~clk;

    // Begin test vector
    initial begin
        $dumpfile("counter_tb.vcd");
        $dumpvars;
        // Reset Counter
        clk = 1'b0;
        reset = 1'b1;
        #2;
        // Start Counting
        reset = 1'b0;
        for (i=0; i<32; i= i + 1) begin
            if (i == 16) $display("Wrapping!");
            $display("time:%4d, val: %b, clk: %b", $time, val, clk);
            #2;
        end
        $finish();
    end
endmodule
```


EDA Playground Example – Task

```
`timescale 1ns/1ps

module counter_testbench;
    // Initial signal and parameter definitions
    parameter bits = 4;
    reg reset, clk;
    wire [bits-1:0] val;

    // Instantiate DUT
    Counter #(.bits(bits)) dut (.reset(reset), .clk(clk),
    .val(val));

    // Set initial clk state
    initial clk = 0;
    integer i = 1;

    // Every 1 time step, toggle clk
    always #(1) clk <= ~clk;

    // Begin test vector
    initial begin
        $dumpfile("counter_tb.vcd");
        $dumpvars;
        // Reset Counter
        clk = 1'b0;
        reset = 1'b1;
        toggle_clk;
        // Start Counting
        reset = 1'b0;
        for (i=0; i<32; i= i + 1) begin
            if (i == 16) $display("Wrapping!");
            $display("time:%4d, val: %b, clk: %b", $time, val, clk);
            toggle_clk;
        end
        $finish();
    end

    task toggle_clk;
    begin
        #1 clk = ~clk;
        #1 clk = ~clk;
    end
endtask
endmodule
```

Questions?