EECS 151 Disc 2 (Verilog Tutorial)

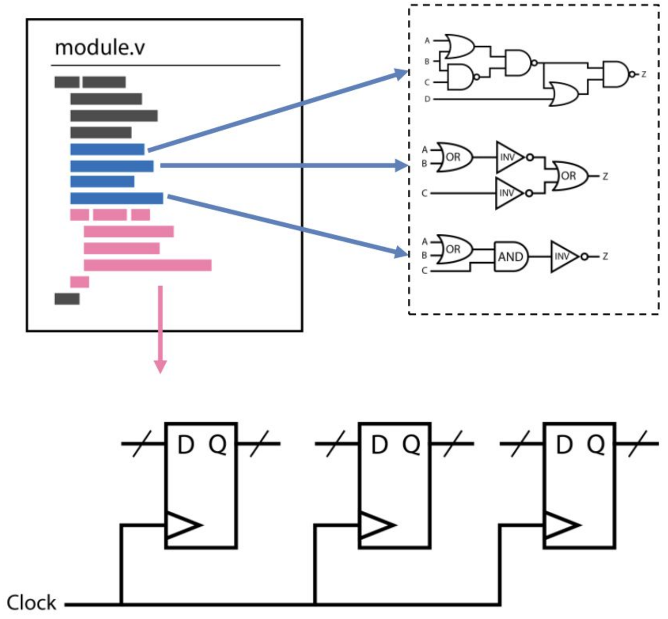
Rahul Kumar (session 1)
Yukio Miyasaka (session 2)

Contents

Acknowledgement: Some materials were taken from EECS 151 sp21/fa22 discussion 2 slides

Hardware Description Language (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
- We use Verilog in this class



Berkeley
UNIVERSITY OF CALIFORNIA

Verilog Basics

Not a programming language!

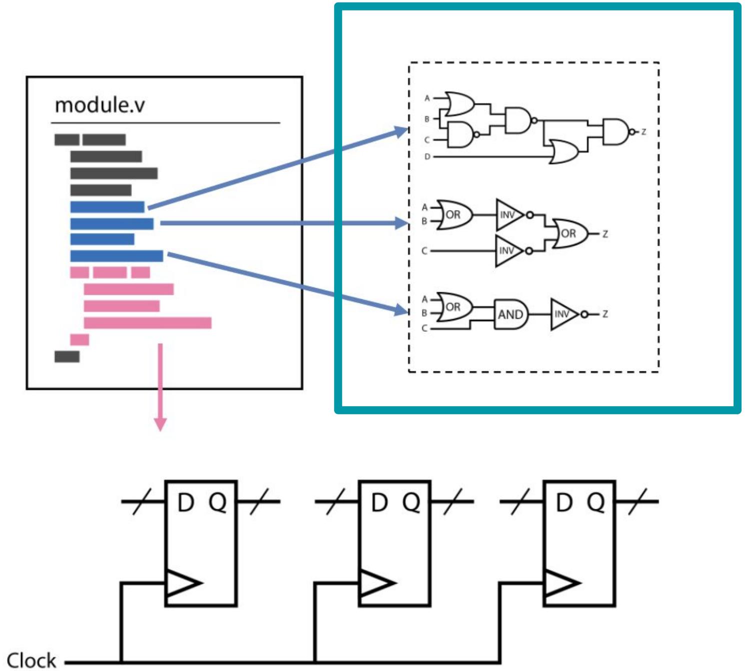
- C-like syntax
- However, we’re writing a description of hardware
- All elements are working concurrently

Combinational Logic

Combinational logic

- Output updates (almost*) immediately with input
- Just because something is assigned at a later line doesn't mean it runs later!

(*) There are some tricky things when doing simulation



Inputs, Outputs, Wires, and Regs

Signals in Verilog are of 2 flavors

- wire
- reg

I/Os declared at beginning of module

- Followed by signal types
  - wire [default] / reg
- input reg doesn't make sense

Internal wires and regs declared after

- Not visible outside module

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

        wire internal_1;
        reg   internal_2;
...
endmodule
```

Inputs, Outputs, Wires, and Regs

Signals in Verilog are of 2 flavors

- wire
- reg

I/Os declared at beginning of module

- Followed by signal types
    - wire [default] / reg
- input reg doesn't make sense

Internal wires and regs declared after

- Not visible outside module

```
module Example (a, b, c, status, s);
        input a;
        input b;
        input c;
        output reg stat
        output s;

        wire internal_1
        reg   internal_2
...
endmodule
```

Alternative

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

        wire internal_1;
        reg   internal_2;
...
endmodule
```

Berkeley
UNIVERSITY OF CALIFORNIA

Multi-Bit Signals

Width of multi-bit signals are declared between type and name

- By convention we use [Width-1:0]
- We can declare multiple signals after type and width
- We can extract a bit or a range using [i] or [i: j]

```
module Example2 (a, b, c, d, e);
        input [3:0] a;
        input [1:0] b, c;
        output [1:0] d, e;

        assign d[0] = a[0];
        assign d[1] = b[1];
        assign e = a[2:1] & c;
...
endmodule
```

Wire vs. Reg

wire

- Continuous assignment

```
wire internal;
assign internal = a & b;
```

- Output of instances (structural Verilog)

```
wire internal;
and (internal, a, b);
```

reg

- These are not actual registers!!!
- Signals in always blocks must be reg

```
reg internal;
always @(*)
    internal = a & b;
```

- Used in Behavioral Verilog except when using continuous assignment

Both can be used as input of operators/instances

Berkeley
UNIVERSITY OF CALIFORNIA

The always @(*) Block

- Used to describe combinational logic in the behavioral Verilog
- We can use various statements inside, such as if, case, etc.
- begin-end is equivalent to {-} in C
    - Necessary to put more than one statements

```
module Example3 (a, b, s, out);
        input a, b, s;
        output reg out;

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

Berkeley
UNIVERSITY OF CALIFORNIA

Multiple Assignment

- Cannot continuously assign wire to two other wires

```
assign a = b;
assign a = c;
```

This is illegal!

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

```
always @(*) begin
        a = b;
        a = c;
end
```

This is okay.
a = c.

- Compilers interpret it and use the last value to generate hardware
    - Previous values are overwritten
- Useful to handle complex conditions

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

Berkeley
UNIVERSITY OF CALIFORNIA

Inferred Latches

How does this example work?

(Note: {a, b} is concatenation)

```
module Example4 (a, b, out);
        input a, b;
        output reg out;

        always @(*) begin
                case ({a, b})
                2'b01: out = 1;
                2'b10: out = 1;
                endcase
        end
endmodule
```

Berkeley
UNIVERSITY OF CALIFORNIA

Inferred Latches

This is not XOR!

- The output will hold a value if there is no matching cases
- Once 2'b10 or 2'b01 happen, the output will stay 1 forever

Make sure all regs are assigned a value for all cases
(Same thing happens when using if statements and not covering all cases)

```
module Example4 (a, b, out);
        input a, b;
        output reg out;

        always @(*) begin
                case ({a, b})
                2'b01: out = 1;
                2'b10: out = 1;
                endcase
        end
endmodule
```

Sequential Logic

In this class, we use a register library

Whenever you need a register, you have to instantiate one from the library

Specification:

- Stores an N-bit value
- Outputs the value as q
- At each positive edge of clk,
  - Resets to INIT if rst
  - Updates to d if !rst & ce
  - Hold the value otherwise

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

Berkeley
UNIVERSITY OF CALIFORNIA

Generators

```
wire [4:0] f;                                  Usage
wire g;
naryand #(.N(5)) and5(.in(f), .out(g));
```

Receives a parameter and creates various instances

We can use **for** loops

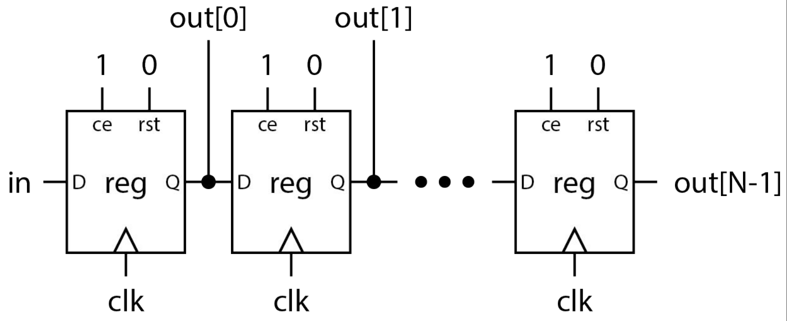
- Loop variables must be **genvar** when creating instances or **always** blocks inside
- (We can use **integer** in other places)

Parameters are given between module name and instance name with #()

```
module naryand (in, out);
    parameter N = 1;
    input [N-1:0] in;
    output out;
    wire [N-1:0] tmp;
    buf(tmp[0], in[0]);
    buf(out, tmp[N-1]);
    genvar i;
    generate for(i = 1; i < N; i = i + 1) begin : ands
        and(tmp[i], in[i], tmp[i-1]);
    end endgenerate
endmodule
```

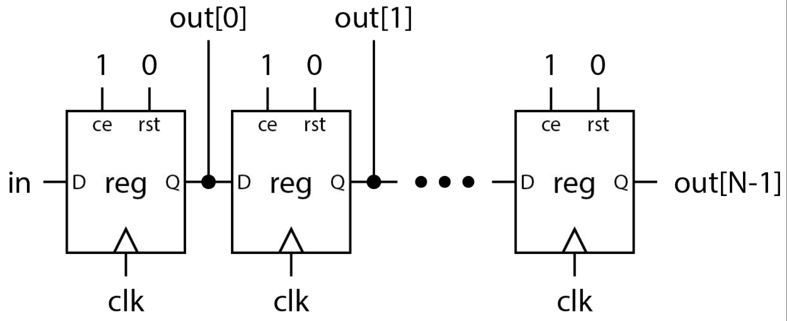
Berkeley
UNIVERSITY OF CALIFORNIA

Example: Shift-Register



```
module shift_register(clk, in, out);
  parameter N = 1;
  input clk, in;
  output [N-1:0] out;
  wire [N-1:0] tmp;

  REGISTER_R_CE #(.N(N))
    r(.q(out), .d(tmp), .rst(0), .ce(1), .clk(clk));

  assign tmp = {out, in};
endmodule
```

Example: Shift-Register



```
module shift_register(clk, in, out);
  parameter N = 1;
  input clk, in;
  output [N-1:0] out;
  wire [N-1:0] tmp;

  REGISTER_R_CE #(
    r(.q(out), .d(tmp), .rst(0), .ce(1), .clk(clk));

  assign tmp = {out, in};
endmodule
```

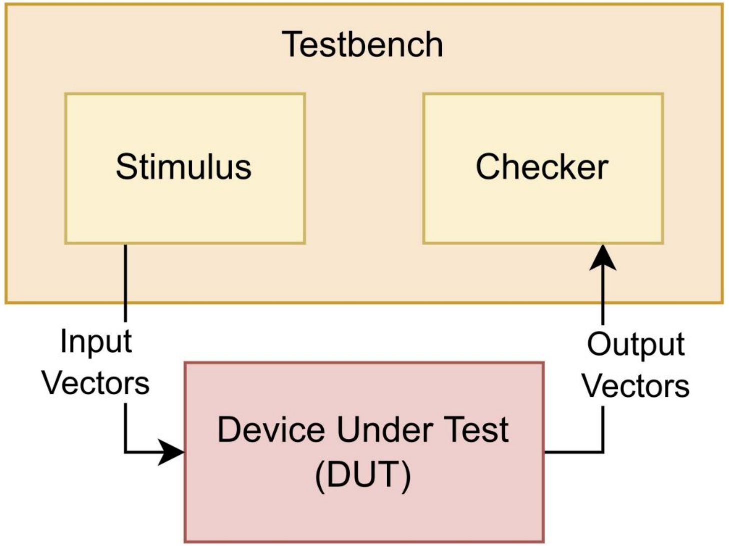
Alternative

```
module shift_register
       #(parameter N = 1)
       (clk, in, out);
  input clk, in;
...
```

Simulation

What’s a Testbench?

- Description of tests to verify that designs behave as specified
- Generate input to drive designs
- Compare output against expected values



Testbench Structure

Testbenches are also written in Verilog

- wire for test output
- reg for test input and others

initial blocks

- Used only for the purpose of simulation
- No hardware implementation

Simulation commands

- $dumpfile(dump.vcd); $dumpvars; generates a waveform
- $finish(); ends simulation

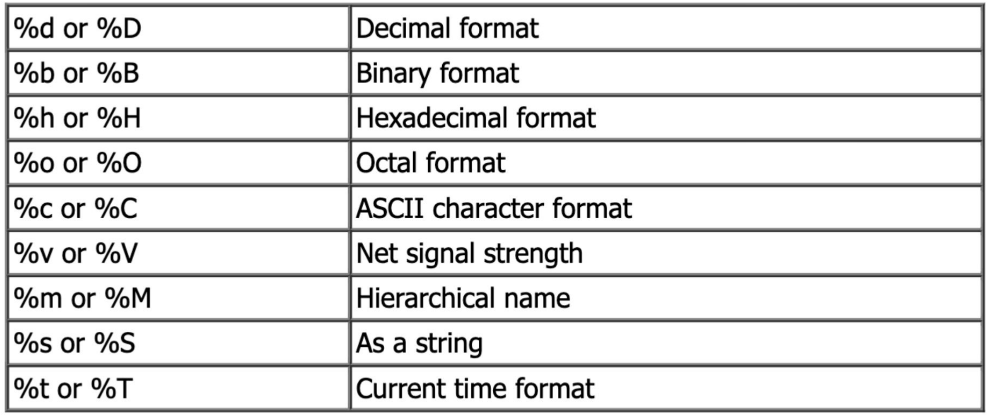
```
module my_tb;
  reg in;
  wire out;
  reg expected;

  DUT dut(.in(in), .out(out));

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars;
    /* Drive inputs and check here */
    $finish();
  end
endmodule
```

Berkeley
UNIVERSITY OF CALIFORNIA

Printing Signals

`$display("format string", values);`

- Example: `$display("in: %b, out: %b, expected: %b", in, out, expected);`



%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

Simulation Order

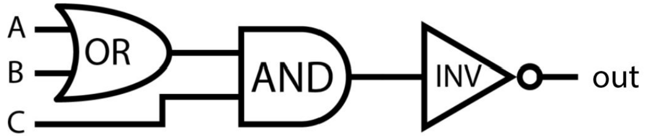
In simulation, DUT output may not update immediately

- Why? - Simulator may execute next line in testbench before
- The execution order is arbitrary

How can we get it updated?

- Simulation timestep advances after executing all waiting updates
- Delay testbench by #num after changing input
- It postpones the execution of the current sequence by num timesteps

Example: OAI



EDA Playground (https://www.edaplayground.com/)

- Free web-based simulator
- Built-in waveform viewer
- HIGHLY recommended for this homework!
- (Select Icarus Verilog 0.9.7 for simulator)

OAI design and testbench:

https://www.edaplayground.com/x/Egiz

Berkeley
UNIVERSITY OF CALIFORNIA

Testing Sequential Circuits

Clock signal

- Initialized to 0
- always #1 flips it every timestep

task block

- Defines a procedure
- Can be called from anywhere in testbench
- task check advances simulation by two timesteps, and compare output with expected values
- Why two timesteps?

```
module my_tb;
...
  reg clk;
  initial clk = 0;
  always #1 clk = ~clk;
...
  task check; begin
    #2;
    if(out !== expected) begin
      $display("FAILED");
      $finish();
    end
  end endtask
...
```

Example: Shift-Register

https://www.edaplayground.com/x/jrmq

*Disregard the warning about timescale for now.