CSE 30321 Verilog Review

Presented by Aaron Dingler
Original Slides by Jay Brockman
Department of Computer Science and Engineering
Department of Electrical Engineering
University of Notre Dame

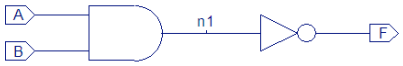
Signal Assignment in Verilog

Beware

- There are lots of ways of describing the same piece of hardware (correctly) in Verilog.
  - Different textbooks, Xilinx “wizards”, etc. adopt different coding styles.
- Some approaches are more error-prone than others.
  - Let's take an approach based on paranoia
  - Pick language features to use in a given situation based on avoiding subtle bugs

NAND Gate: Structural Model

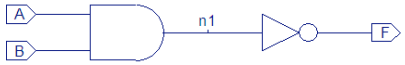
```
module nand_structural(A, B, F);
       input A, B;
       output F;

       wire n1;

       not (F, n1);
       and (n1, A, B);

endmodule
```



NAND Gate: 2 Concurrent **always** Statements

```
module nand_always(A, B, F);
       input A, B;
       output F;

       reg    F;
       reg n1;

       always @(n1)
              F = ~n1;

       always @(A, B)
              n1 = A & B;

endmodule
```

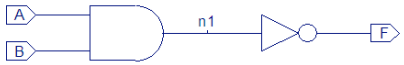
always statements run concurrently



NAND Gate: 2 Concurrent **assign** Statements

```
module nand_assign(A, B, F);
       input A, B;
       output F;

       wire n1;

       assign F = ~n1;
       assign n1 = A & B;

endmodule
```

assign statements run concurrently



Blocking vs. Non-Blocking Assignment

blocking: complete each assignment before moving on to next statement

```
begin
       LHS1 = RHS1;
       LHS2 = RHS2;
       LHS3 = RHS3;
end
```

evaluate RHS_1, assign to LHS_1
evaluate RHS_2, assign to LHS_2
evaluate RHS_3, assign to LHS_3

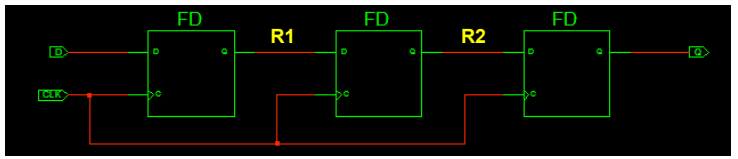
non-blocking: evaluate each RHS without waiting for assignment, then make all assignments concurrently

```
begin
       LHS1 <= RHS1;
       LHS2 <= RHS2;
       LHS3 <= RHS3;
end
```

evaluate RHS_1
evaluate RHS_2
evaluate RHS_3
assign right-hand sides to left-hand-sides

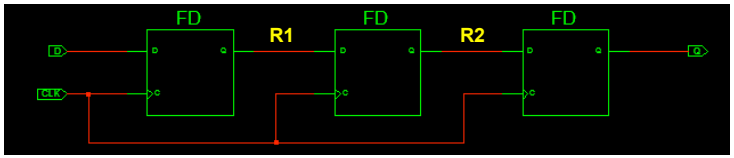
Non-Blocking Registers

```
module register_nonblock_fwd(D, CLK, Q);
       input D, CLK;
       output Q;

       reg Q, R1, R2;

       always @(posedge CLK)
              begin
                     R1 <= D;
                     R2 <= R1;
                     Q  <= R2;
              end
endmodule
```



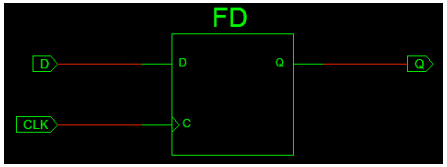
Non-Blocking Registers (Reversed Order)

```
module register_nonblock_fwd(D, CLK, Q);
       input D, CLK;
       output Q;

       reg Q, R1, R2;

       always @(posedge CLK)
              begin
                     Q  <= R2;
                     R2 <= R1;
                     R1 <= D;
              end
endmodule
```



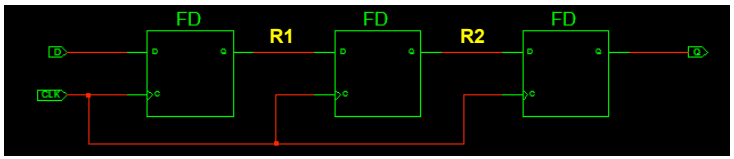
Blocking Registers

```
module register_nonblock_fwd(D, CLK, Q);
       input D, CLK;
       output Q;

       reg Q, R1, R2;

       always @(posedge CLK)
              begin
                     R1 = D;
                     R2 = R1;
                     Q  = R2;
              end
endmodule
```



Blocking Registers (Reversed Order)

```
module register_nonblock_fwd(D, CLK, Q);
       input D, CLK;
       output Q;

       reg Q, R1, R2;

       always @(posedge CLK)
              begin
                     Q  = R2;
                     R2 = R1;
                     R1 = D;
              end
endmodule
```



Cyclic Assignment NAND Gates

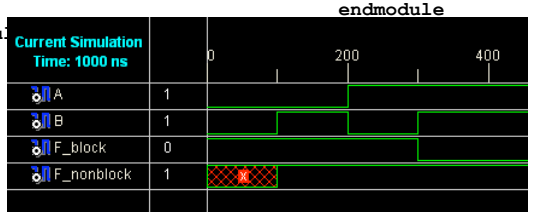
```
module nand_block(A, B, F);
       input A, B;
       output F;

       reg n1, F;

       always @(A, B)
              begin
              n1 = A & B;
              F = ~n1;
              end

endmodule
```

```
module nand_nonblock(A, B, F);
       input A, B;
       output F;

       reg n1, F;

       always @(A, B)
              begin
              n1 <= A & B;
              F <= ~n1;
              end

endmodule
```

Cyclic Assignment NAND Gates

```
module nand_block(A, B, F);
        input A, B;
        output F;

        reg n1, F;

        always @(A, B)
                begin
                        n1 = A & B;
                        F = ~n1;
                end
endmodule
```

```
module nand_nonblock(A, B, F);
        input A, B;
        output F;

        reg n1, F;

        always @(A, B)
                begin
                        n1 <= A & B;
                        F <= ~n1;
                end
endmodule
```

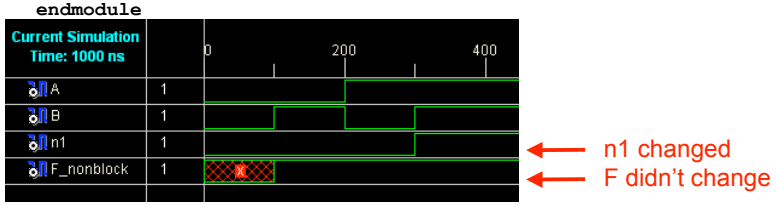


A Closer Look

```
module nand_nonblock(A, B, F);
        input A, B;
        output F;

        reg n1, F;

        always @(A, B)
                begin
                        n1 <= A & B;
                        F <= ~n1;
                end

endmodule
```

needs to be sensitive to n1

```
always @(A, B, n1)
```



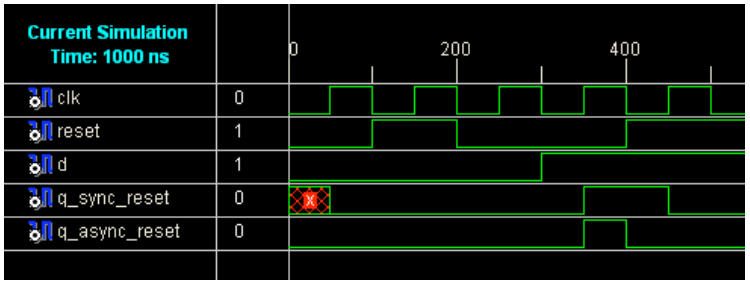
Flip Flops with Reset

```
module flipflop_sync_reset(d, clk, reset, q);
        input d, clk, reset;
        output q;
        reg q;

        always @(posedge clk)
                if (reset) q <= 0;
                else q <= d;
endmodule

module flipflop_async_reset(d, clk, reset, q);
        input d, clk, reset;
        output q;
        reg q;

        always @(posedge clk, reset)
                if (reset) q <= 0;
                else q <= d;
endmodule
```

Flip Flop Testbench

```
module flipflop_compare_tb();
        reg d, clk, reset;
        wire q_sync_reset, q_async_reset;

        flipflop_sync_reset(d, clk, reset, q_sync_reset);
        flipflop_async_reset(d, clk, reset, q_async_reset);

        always
                #50 clk = ~clk;

        initial begin
                d = 0;
                clk = 0;
                reset = 0;
                #100 reset = 1;
                #100 reset = 0;
                #100 d = 1;
                #100 reset = 1;
        end

        initial #10000 $finish;
endmodule
```


Flip Flop Simulation Results



Bottom Line

- Combinational logic
  - non-blocking assignment =
- Sequential logic (flip flops)
  - blocking assignment <=

Helps preserve your sanity!

Questions

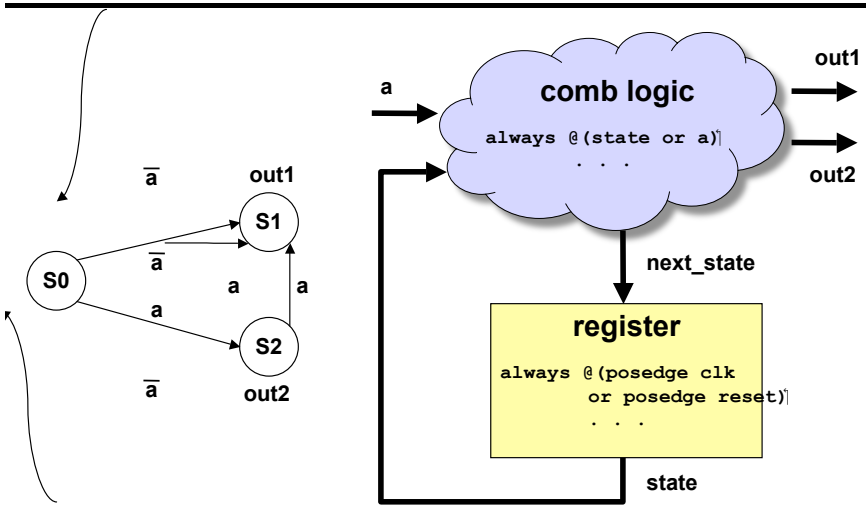
- Are there any questions on signal assignment?

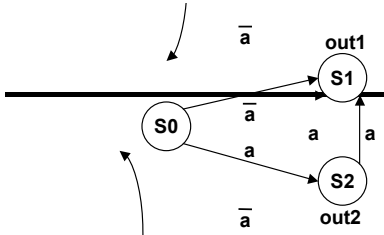
Finite State Machines in Verilog

What we (should) know so far

- Finite state machines
  - Binary encoded state names
  - Next state depends on
    - current state
    - input
  - Outputs depend on
    - Moore machine: state only
    - Mealy machine: state and inputs
- Verilog
  - Combinational logic
    - sensitive to changes in signal levels
  - Sequential logic (registers)
    - sensitive to changes in clock edges (and reset)

Describing FSM in Verilog

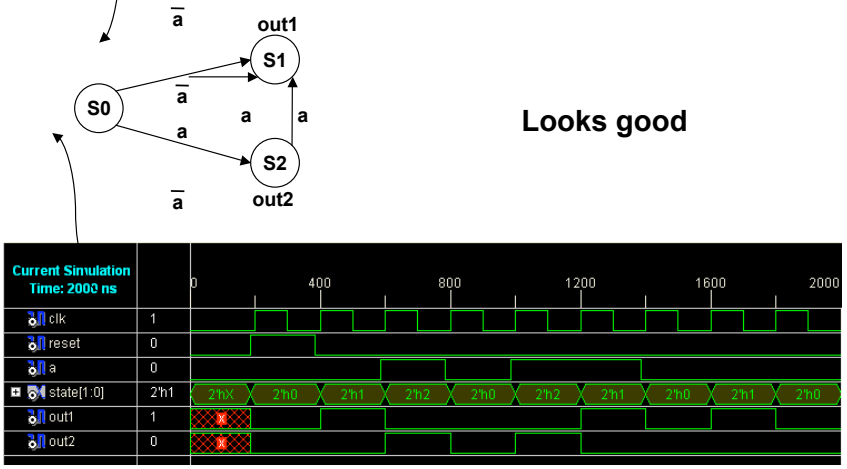





```
module fsm(clk, reset, a, out1, out2);
    input clk;
    input reset;
    input a;
    output reg out1;
    output reg out2;

        reg [1:0] state;
        reg [1:0] next_state;

        parameter s0 = 2'b00;
        parameter s1 = 2'b01;
        parameter s2 = 2'b10;

        always @(posedge clk)
                if (reset)
                        state <= s0;
                else
                        state <= next_state; endmodule
```

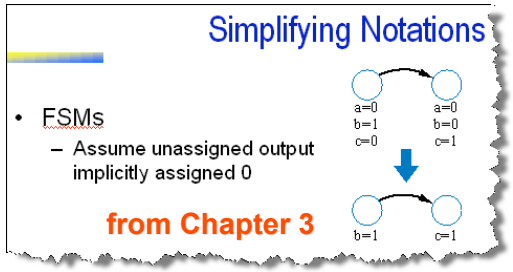
```
always @(state, a) begin
  out1 = 0; out2 = 0;
  case (state)
    s0: begin
      if (a == 0)
        next_state = s1;
      else
        next_state = s2;
      out1 = 0;
      out2 = 0;
    end
    s1: begin
      if (a == 0)
        next_state = s0;
      else
        next_state = s2;
      out1 = 1;
      out2 = 0;
    end
    s2: begin
      if (a == 0)
        next_state = s0;
      else
        next_state = s1;
      out1 = 0;
      out2 = 1;
    end
  endcase
end
```

Simulation Results



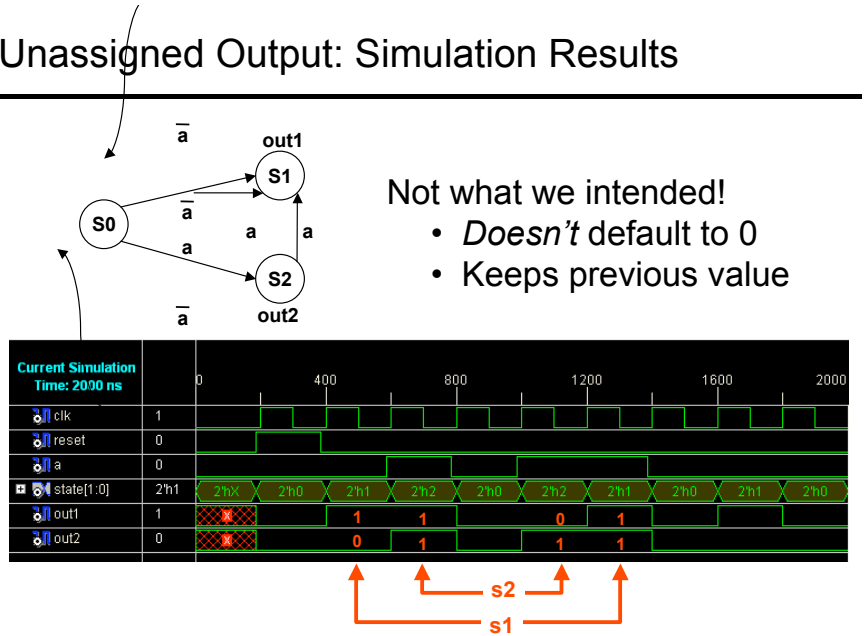
Looks good

Unassigned Output (Beware!)



```
always @(state, a)
  case (state)
    s0: begin
      if (a == 0)
        next_state = s1;
      else
        next_state = s2;
      out1 = 0;
      out2 = 0;
    end
    s1: begin
      if (a == 0)
        next_state = s0;
      else
        next_state = s2;
      out1 = 1;
    end
    s2: begin
      if (a == 0)
        next_state = s0;
      else
        next_state = s1;
      out2 = 1;
    end
  endcase
endmodule
```


Unassigned Output: Simulation Results

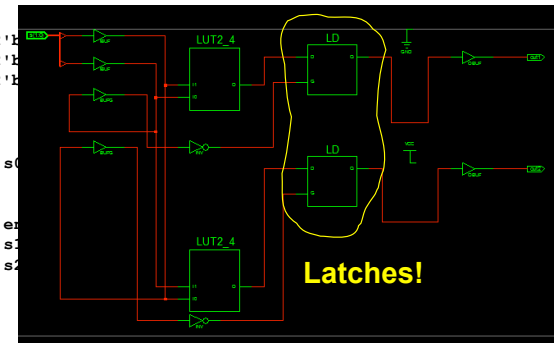


Not what we intended!

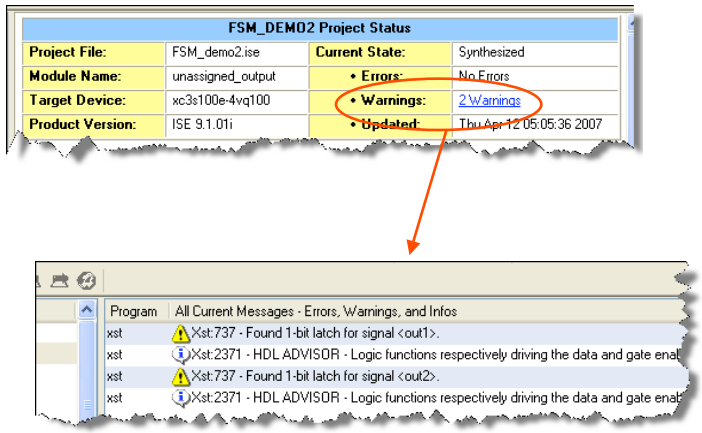
- *Doesn't* default to 0
- Keeps previous value

Synthesis of Unassigned Output

```
module unassigned_output(s, out1, out2);
    input [1:0] s;
    output reg out1;
    output reg out2;

        parameter s0 = 2'b
        parameter s1 = 2'b
        parameter s2 = 2'b

        always @(s)
                case (s)
                        s0
                        en
                        s1
                        s2
                endcase
endmodule
```

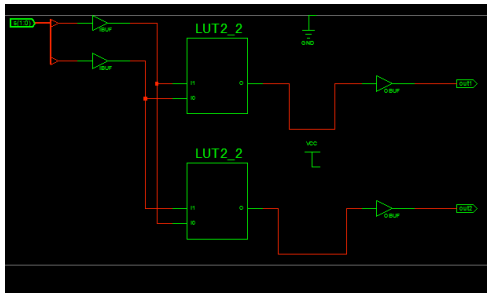


Detecting Unassigned Output/Latch Problem



Always assign all outputs!
Don't leave undefined input cases!

```
always @(s) begin
 out1 = 0; out2 = 0
 case (s)
   s0: begin
     out1 = 0;
     out2 = 0;
   end
   s1: begin
     out1 = 1;
     out2 = 0;
   end
   s2: begin
     out1 = 0;
     out2 = 1;
   end
   default: begin
     out1 = 0;
     out2 = 0;
   end
 endcase
end
```



Define default input case so FSM can't get "stuck" (Also a latch problem)

Questions

- Are there any questions on creating FSMs in Verilog?