

ECE 551

System on Chip Design

Introduction to SystemVerilog

Garrett S. Rose
Fall 2018

First Things First

- SystemVerilog is a superset of Verilog
 - The SystemVerilog subset we use is 99% Verilog + a few new constructs
 - Familiarity with Verilog (which you have) helps a lot
- Useful SystemVerilog resources and tutorials on Canvas

More Thoughts on HDLs

- Used for a variety of purposes in hardware design
 - High-level behavioral modeling
 - Register transfer level (RTL) behavioral modeling
 - Gate level netlists
 - Timing models for timing simulation
 - Design verification and testbench development
- Many different and useful features
- We tend to focus on RTL modeling and “synthesizable HDL”

HDL vs. Programming

- Syntactically similar:
 - Data types, variables, assignments, if statements, loops, ...
- Very different mentality and semantic models: everything runs concurrently, unless specified otherwise
 - Statements model hardware
 - Implement structure in space not actions in time
- SW programs are composed of *subroutines* or *functions*
 - Subroutines *call* each other
 - Calling subroutine pauses when called subroutine runs
- HW descriptions are composed of *modules*
 - *Hierarchy* of modules *connected* to each other
 - Modules are active at same time – they're *concurrent*

SystemVerilog Modules

- Look familiar? This is basically Verilog!
- Modules interface with top level constructs via ports
- Ports are usually input or output

```
module mymodule
    (a, b, c, f);

    output f;
    input  a, b, c;

    // Description goes here
endmodule: mymodule

// alternatively
module mymodule
    (input logic a , b, c,
     output logic f);

    // Description goes here
endmodule: mymodule
```

Module Instantiation

Name of
module to
instantiate

```
module mymodule  
    (a, b, c, f);
```

```
    output f;  
    input a, b, c;
```

```
module_name inst_name(port_connections);  
endmodule: mymodule
```

Instance name

Connect the ports

- You can instantiate your own modules or pre-defined gates
 - Always instantiated inside another module
- Predefined: and, nand, or, nor, xor, xnor
 - For predefined gates port order is (output, input(s))
- For user defined modules, port order determined by user

Options for Connecting Modules

- For module instantiation, can specify port connections by name or by order

```
module mod1  
  (input a, b, output f);
```

```
  // ...  
endmodule: mod1
```

```
// by order  
module mod2  
  (input c, d, output g);
```

```
  mod1 i0(c, d, g);  
endmodule: mod2
```

```
// by name  
module mod3  
  (input c, d, output g);
```

```
  mod1 i0(.f(g), .b(d), .a(c));  
endmodule: mod3
```

By name connections
can save headaches

Example Structural Design

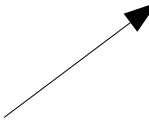
- This is SystemVerilog... you've seen this before with Verilog

```
module mux
  (input a, b, sel output f);

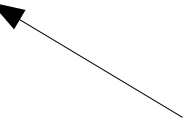
  logic c, d, not_sel;

  not gate0(not_sel, sel);
  and gate1(c, a, not_sel);
  and gate2(d, b, sel);
  or gate3(f, c, d);
endmodule: mux
```

Datatype for
describing
logical value



Built-in gates:
Port order is
output(s), input(s)



Edge-Triggered Events

- Variant of always block called **always_ff**
 - Indicated that block will be sequential logic (flip flops)
- Procedural block occurs only on signal's edge
 - **@(posedge ...)** or **@(negedge ...)**

```
always_ff @(posedge clk, negedge reset_n) begin  
    // This procedure will be executed  
    // anytime clk goes from 0 to 1  
    // or anytime reset_n goes from 1 to 0  
end
```

Flip Flops

- Output q “remembers” what input d was at last clock edge
 - One bit of memory
- Without reset:

```
module flipflop
  (d, q, clk);

  input d, clk;
  output logic q;

  always_ff @(posedge clk) begin
    q <= d;
  end
endmodule: flipflop
```

Flip Flops

- Output q “remembers” what input d was at last clock edge
 - One bit of memory
- With asynchronous reset:

```
module flipflop_asyncr
    (d, q, clk, rst_n);

    input d, clk, rst_n;
    output logic q;

    always_ff @(posedge clk, negedge rst_n) begin
        if (rst_n == 0)
            q <= 0;
        else
            q <= d;
        end
    endmodule: flipflop_asyncr
```

Flip Flops

- Output q “remembers” what input d was at last clock edge
 - One bit of memory
- With synchronous reset:

```
module flipflop_syncr
    (d, q, clk, rst_n);

    input d, clk, rst_n;
    output logic q;

    always_ff @(posedge clk) begin
        if (rst_n == 0)
            q <= 0;
        else
            q <= d;
        end
    endmodule: flipflop_syncr
```


Multi-Bit Flip Flop

```
module flipflop_asyncr
  (d, q, clk, rst_n);

  input [15:0] d,
  input clk, rst_n;
  output logic [15:0] q;

  always_ff @(posedge clk, negedge rst_n) begin
    if (rst_n == 0)
      q <= 0;
    else
      q <= d;
  end
endmodule: flipflop_asyncr
```

Recognized by type



Digression: Module Parameters

- Parameters allow modules to be easily changed

```
module my_flipflop
  #(parameter WIDTH=16)
    (d, q, clk, rst_n);

    input [WIDTH-1:0] d;
    input clk, rst_n;
    output logic [WIDTH-1:0] q;
    ...
endmodule: my_flipflop
```

- Instantiate and set parameter:

```
// To use the default parameter value:
my_flipflop f0(d, q, clk, rst_n);

// Change parameter to 12 for this instance:
my_flipflop #(12) f0(d, q, clk, rst_n);
```

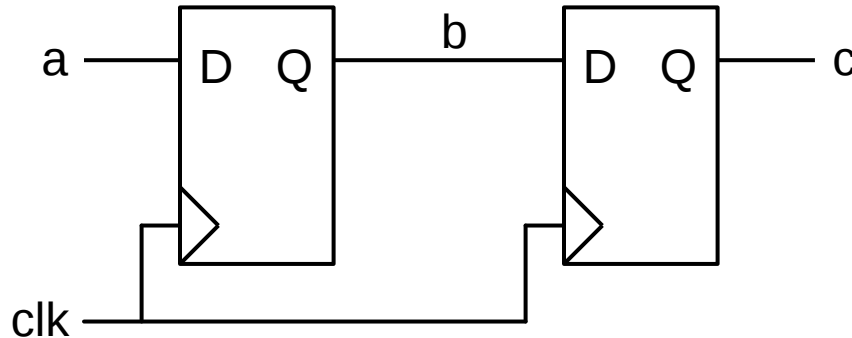
Non-Blocking Assignment $a \leq b$;

- \leq is the non-blocking assignment operator
 - All left-hand side values take new values *concurrently*

```
always_ff @(posedge clk) begin  
    b  $\leq$  a;  
    c  $\leq$  b;  
end
```

c gets *old* value of b, not
value assigned just above

- This models synchronous logic



Non-Blocking vs. Blocking

- Use non-blocking assignment `<=` to describe edge-triggered (synchronous) assignments

```
always_ff @(posedge clk) begin  
    b <= a;  
    c <= b;  
end
```

- Use blocking assignment `=` to describe combinational assignment

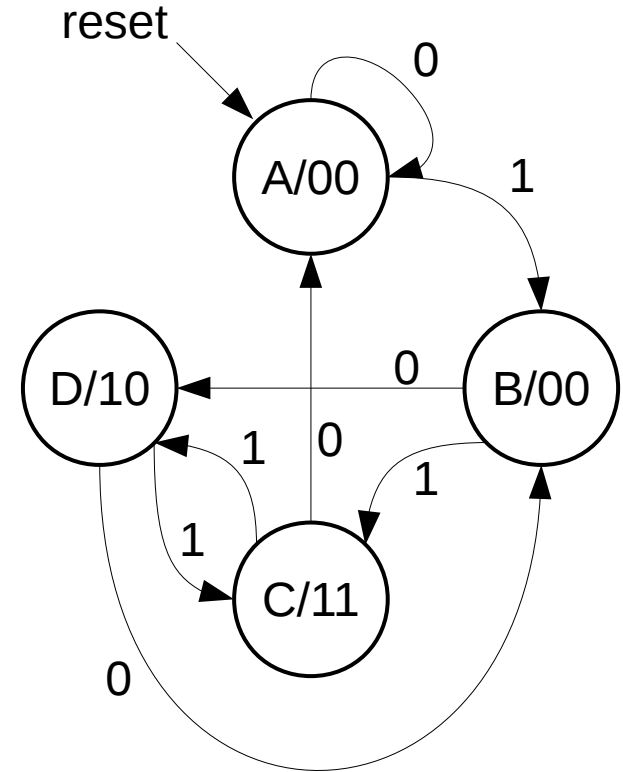
```
always_comb begin  
    b = a;  
    c = b;  
end
```


Design Example

- Let's design a module to compute $f = a + b * c$
 - b and c are 4 bits, a is 8 bits, and f is 9 bits
- First, we will build this as a combinational circuit
- Then, we will add registers at the inputs and outputs

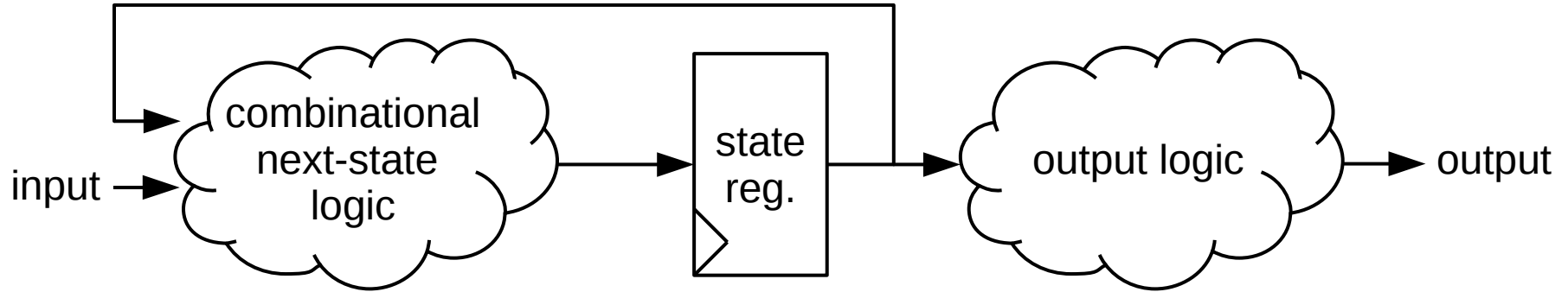
Finite State Machine

- State names
- Output values
- Transition values
- Reset state



Finite State Machines Continued

- An FSM look like when implemented



- Combinational logic and registers (things we know how to implement using SystemVerilog)

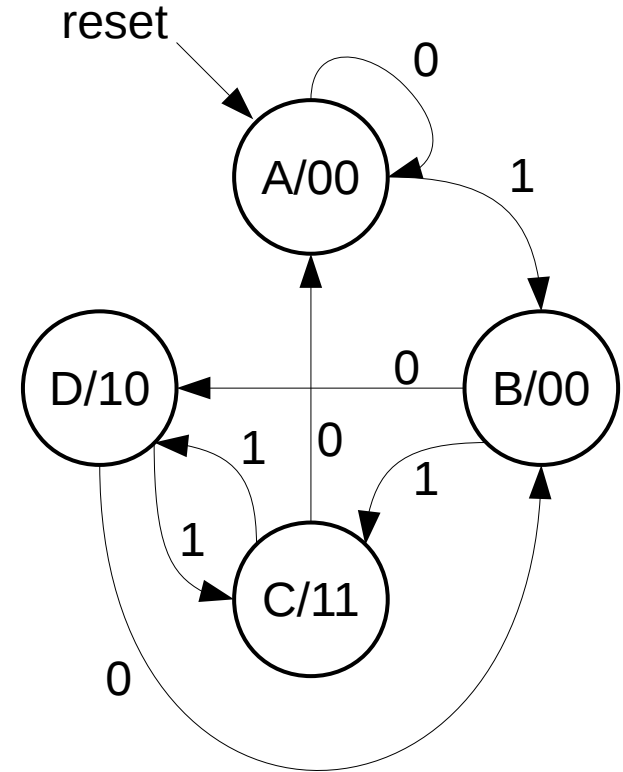
Full FSM Example

```
module fsm
  (clk, rst, x, y);

  input clk, rst, x, y;
  output logic [1:0] y;
  enum { STATEA=2'b00, STATEB=2'b01, STATEC=2'b10,
        STATED=2'b11 } curr_state, next_state;

  // next state logic
  always_comb begin
    case(curr_state)
      STATEA: next_state = x ? STATEB : STATEA;
      STATEB: next_state = x ? STATEC : STATED;
      STATEC: next_state = x ? STATED : STATEA;
      STATED: next_state = x ? STATEC : STATEB;
    endcase
  end

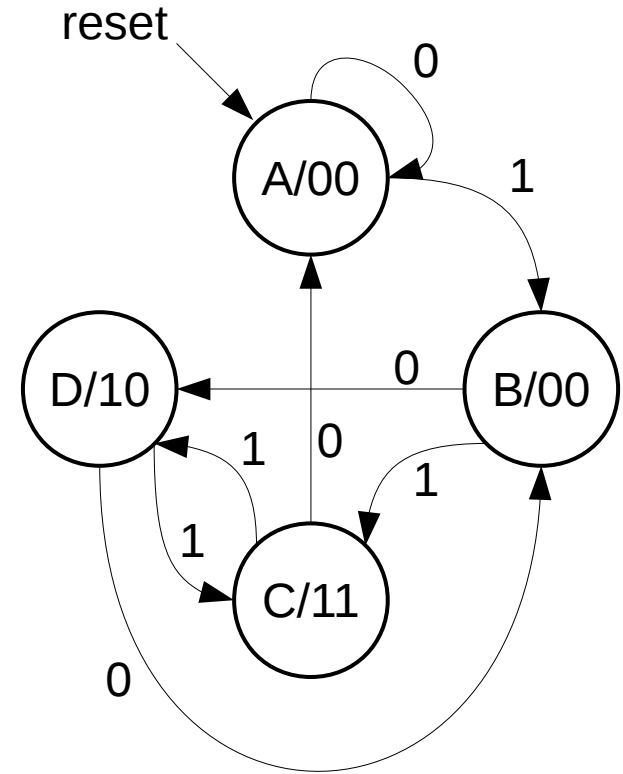
  // to be continued ...
```



Full FSM Example

```
// now concluded ...
// register for state memory
always_ff @(posedge clk) begin
    if (rst)
        curr_state <= STATEA;
    else
        curr_state <= next_state;
end

// output logic
always_comb begin
    case(curr_state)
        STATEA: y = 2'b00;
        STATEB: y = 2'b00;
        STATEC: y = 2'b11;
        STATED: y = 2'b10;
    endcase
end
endmodule: fsm
```



Arrays

```
module multidimarraytest;
    logic [3:0] myarray [2:0];

    assign myarray[0]      = 4'b0010;
    assign myarray[1][3:2] = 2'b01;
    assign myarray[1][1]   = 1'b1;
    assign myarray[1][0]   = 1'b0;
    assign myarray[2][3:0] = 4'hC;

    initial begin
        $display("myarray      == %b", myarray);
        $display("myarray[2:0] == %b", myarray[2:0]);
        $display("myarray[1:0] == %b", myarray[1:0]);
        $display("myarray[1]   == %b", myarray[1]);
        $display("myarray[1][2] == %b", myarray[1][2]);
        $display("myarray[2][1:0] == %b", myarray[2][1:0]);
    end
endmodule: multidimarraytest
```

Assertions

- Assertions are test constructs
 - Automatically validated as design is simulated
 - Written for properties that must always be true
- Makes it easier to test design
 - Don't have to manually check for these conditions

Example: Good Place for Assertions

- Imagine you have a FIFO queue
 - When queue is full, it sets status_full to true
 - When queue is empty, it sets status_empty to true



- When status_full is true, wr_en must be false
- When status_empty is true, rd_en must be false

Assertion Syntax

- Assertion: a procedural statement that checks an expression when statement is executed

Use \$display to
print test, \$error
to print error, or
\$fatal to print
and halt
simulation

```
// general form
assertion_name: assert(expression) pass_code;
else fail_code;

// example
always @(posedge clk) begin
    assert ((status_full == 0) || (wr_en == 0))
    else $error("Tried to write to FIFO when full.");
end
```

- SystemVerilog includes *concurrent assertions* that are continuously monitored and can express temporal conditions
 - Complex but very powerful
 - See: <http://www.doulos.com/knowhow/sysverilog/tutorial/asserstions>