

Introduction to SystemVerilog

Nima Honarmand

(Slides adapted from Prof. Milder's ESE-507 course)

First Things First

- SystemVerilog is a superset of Verilog
 - The SystemVerilog subset we use is 99% Verilog + a few new constructs
 - Familiarity with Verilog (or even VHDL) helps but is not necessary
- SystemVerilog resources and tutorials on the course “Assignments” web page

Hardware Description Languages (HDL)

- HDLs are used for a variety of purposes in hardware design
 - Functional simulation
 - Timing simulation
 - Hardware synthesis
 - Testbench development
 - ...
- Many different features to accommodate all of these
 - We focus on functional simulation
- With HDLs, you describe hardware in one of two styles (usually)
 - Structural model (network of gates and transistors)
 - Behavioral model (high-level statements such as assignments, if, while, ...)
- We use behavioral modeling for the course project
 - Much simpler than designing with gates

HDLs vs. Programming Languages (1)

- Have syntactically similar constructs:
 - Data types, variables, operators, assignments, if statements, loops, ...
- But very different mentality and semantic model
- Statements are evaluated in parallel (unless specified otherwise)
 - Statements model hardware
 - Hardware is inherently parallel

***Reset your mind! You are a HW developer now.
Stop thinking like a SW programmer!***

HDLs vs. Programming Languages (2)

- Software programs are organized as a set of ***subroutines***
 - Subroutines ***call*** each other, passing ***arguments and return values***
 - When in callee, caller's execution is paused
- Hardware descriptions are organized as a hierarchy of hardware ***modules***
 - A ***hierarchy*** of module instances ***connected*** to each other using ***wires***
 - Modules are active at the same time

Modules

- The basic building block in SystemVerilog
 - Interfaces with outside using **ports**
 - Ports are either **input** or **output** (for now)

all ports declared here

module name

declare which
ports are inputs,
which are outputs

```
module mymodule(a, b, c, f);  
    output f;  
    input a, b, c;  
    // Description goes here  
endmodule  
  
// alternatively  
module mymodule(input a, b, c, output f);  
    // Description goes here  
endmodule
```

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
```

name of
instance

connect the ports

- You can instantiate your own modules or pre-defined gates
 - Always inside another module
- Predefined: `and`, `nand`, `or`, `nor`, `xor`, `xnor`
 - for these gates, port order is <output, input(s)>
- For your modules, port order is however you defined it

Connecting Ports

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

```
module mod1(input a, b, output f);  
    // ...  
endmodule  
  
// by order  
module mod2(input c, d, output g);  
    mod1 i0(c, d, g);  
endmodule  
  
// by name  
module mod3(input c, d, output g);  
    mod1 i0(.f(g), .b(d), .a(c));  
endmodule
```

Advice: Use **by-name** connections (where possible)

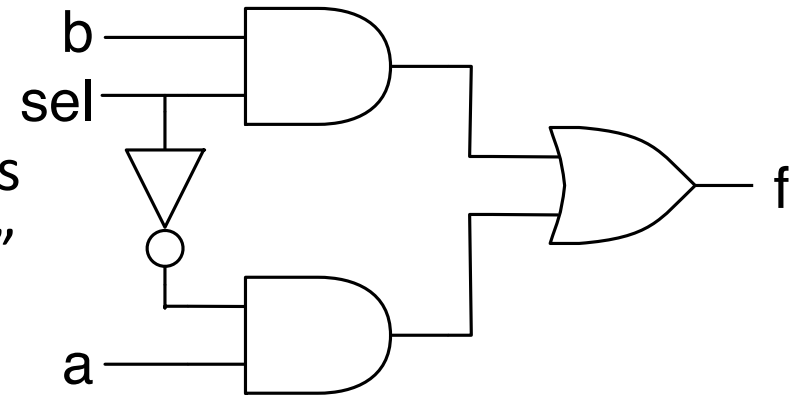
Review: Combinational vs. Sequential Logic

- In combinational logic, circuit outputs are pure function of circuit inputs
 - i.e., output values only determined by input values
 - Examples: and, or, multiplexer, adder, etc.
- In sequential logic, there are “state” elements that can “hold” their old values regardless of the input changes
 - Example: any circuit with a latch, flip-flop or any other “memory” element in it

Combinational Logic Description

Structural Description

- Example: multiplexor
 - Output equals one of the inputs
 - Depending on the value of “sel”



```

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

  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

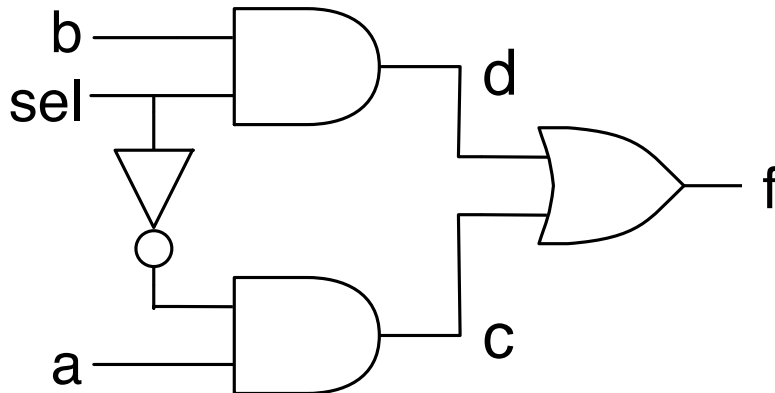
```

datatype for describing Boolean logic

Built-in gates:
port order is:
<output, input(s)>

Behavioral: Continuous Assignment

- Specify logic *behaviorally* by writing an expression to show how the signals are related to each other.
 - assign statement



```
module mux2(a, b, sel, f);
    output f;
    input a, b, sel;
    logic c, d;

    assign c = a & (~sel);
    assign d = b & sel;
    assign f = c | d;

    // or alternatively
    assign f = sel ? b : a;
endmodule
```

Behavioral: Procedural Block

- Can use `always_comb` procedural block to describe combinational logic using *a series of sequential statements*
- All `always_comb` blocks are independent and parallel to each other

```
module mymodule(a, b, c, f);  
    output f;  
    input a, b, c;  
  
    always_comb begin  
        // Combinational logic  
        // described  
        // in C-like syntax  
    end  
endmodule
```

Procedural Behavioral Mux Description

```
module mux3(a, b, sel, f);  
    output logic f;  
    input a, b, sel;  
  
    always_comb begin  
        if (sel == 0) begin  
            f = a;  
        end  
        else begin  
            f = b;  
        end  
    end  
end  
endmodule
```

If we are going to drive f this way, need to declare it as `logic`

Important: for behavior to be combinational, every output (f) must be assigned in all possible control paths

Why? Otherwise, would be a latch and not combinational logic.

Avoid Accidental Latch Description

```
module bad(a, b, f);
  output logic f;
  input a, b;

  always_comb begin
    if (b == 1) begin
      f = a;
    end
  end
end
endmodule
```

- This is not combinational, because for certain values of b, f must **remember** its previous value.
- This code describes a latch. (If you want a latch, you should define it using `always_latch`)

Avoid Multiply-Assigned Values

```
module bad2 (...);  
  ...  
  always_comb begin  
    b = ... something ...  
  end  
  always_comb begin  
    b = ... something else ...  
  end  
endmodule
```

- Both of these blocks execute ***concurrently***
- So what is the value of b?
We don't know!

Don't do this!

Multi-Bit Values

- Can define inputs, outputs, or logic with multiple bits
 - Also called *bit vectors*

```
module mux4(a, b, sel, f);  
    output logic [3:0] f;  
    input [3:0] a, b;  
    input sel;  
  
    always_comb begin  
        if (sel == 0) begin  
            f = a;  
        end  
        else begin  
            f = b;  
        end  
    end  
end  
endmodule
```

Multi-Bit Constants and Concatenation

- Can give constants with specified number bits
 - In binary, decimal or hexadecimal
- Can concatenate with { and }

```
logic [3:0] a, b, c;
logic signed [3:0] d;
logic [7:0] e;
logic [1:0] f;
assign a = 4'b0010; // four bits, specified in binary
assign b = 4'hC; // four bits, specified in hex == 1100
assign c = 3; // == 0011
assign d = -2; // 2's complement == 1110 as bits
assign e = {a, b}; // concatenate == 0010_1100
assign f = a[2 : 1]; // two bits from middle == 01
```

Case Statements and “Don’t-Cares”

```
module newmod(out, in0, in1, in2);
  input in0, in1, in2;
  output logic out;

  always_comb begin
    case({in0, in1, in2})
      3'b000: out = 1;
      3'b001: out = 0;
      3'b010: out = 0;
      3'b011: out = x;
      3'b10x: out = 1;
      default: out = 0;
    endcase
  end
endmodule
```

output value is
undefined in this case

Last bit is a “don’t
care” -- this line will
be active for 100 OR
101

default gives “else”
behavior. Here active
if 110 or 111

Arithmetic Operators

- Standard arithmetic operators defined: + - * / %
- Many subtleties here, so be careful:
 - four bit number + four bit number = five bit number
 - Or just the lower four bits
 - arbitrary division is difficult

Addition and Subtraction (1)

- Be wary of overflow!

```
logic [3:0] d, e, f;  
  
assign f = d + e;
```

$4'b1000 + 4'b1000 = 4'b0000$
In this case, overflows to zero

```
logic [3:0] a, b;  
logic [4:0] c;  
  
assign c = a + b;
```

Five-bit output can prevent overflow:
 $4'b1000 + 4'b1000$ gives $5'b10000$

Addition and Subtraction (2)

- Use “signed” if you want values as 2’s complement

`i == 4'b1010 == -6` ←
`j == 5'b11010 == -6`

```
logic signed [3:0] g, h, i;  
logic signed [4:0] j;  
  
assign g = 4'b0001; // == 1  
assign h = 4'b0111; // == 7  
assign i = g - h;  
assign j = g - h;
```

Multiplication

- Multiply k bit number with m bit number
 - How many bits does the result have? $k+m$

```
logic signed [3:0] a, b;  
logic signed [7:0] c;  
assign a = 4'b1110; // -2  
assign b = 4'b0111; // 7  
assign c = a*b;
```

$c = 8'b1111_0010 == -14$

- If you use fewer bits in your code
 - Gets least-significant bits of the product

```
logic signed [3:0] a, b, d;  
assign a = 4'b1110; // -2  
assign b = 4'b0111; // 7  
assign d = a*b;
```

$d = 4'b0010 == 2$

Design Example

- Let's say we want to compute $f = a + b * c$
 - b and c are 4 bits, a is 8 bits, and f is 9 bits
- Let's a combinational circuit using `always_comb`

```
module MAF(f, a, b, c);  
    input [7:0] a  
    input [3:0] b, c;  
    output logic [9:0] f;  
  
    always_comb begin  
        f = a + b * c;  
    end  
endmodule
```

OR

```
module MAF(f, a, b, c);  
    input [7:0] a  
    input [3:0] b, c;  
    output logic [9:0] f;  
  
    logic [7:0] temp;  
  
    always_comb begin  
        temp = b * c;  
        f = a + temp;  
    end  
endmodule
```


Design Example 2

- Let's say we want to compute $f = (a ? b + 1 : c + 2) * d$
 - a is 1 bit; b, c and d are 4 bits
 - How wide should f be to avoid any overflows?
- Let's a combinational circuit using `always_comb`

Sequential Logic Description

Sequential Design

- Everything so far was purely combinational
 - Stateless
- What about *sequential* systems?
 - flip-flops, registers, finite state machines
- New constructs
 - `always_ff @ (posedge clk)`
 - non-blocking assignment `<=`

Edge-Triggered Events

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

```
always_ff @(posedge clk, negedge reset_n) begin

    // This block will be evaluated
    // anytime clk goes from 0 to 1
    // or anytime reset_n goes from 1 to 0

end
```

Flip Flops (1)

- q remembers what d was at the 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
```

Flip Flops (2)

- 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
```

Flip Flops (3)

- 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
```

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
```


Interlude: Module Parameters

- Parameters allow modules to be easily changed

```
module my_flipflop(d, q, clk, rst_n);  
    parameter WIDTH=16;  
    input [WIDTH-1:0] d;  
    input clk, rst_n;  
    output logic [WIDTH-1:0] q;  
    ...  
endmodule
```

← default value set to 16

- Instantiate and set parameter:

```
my_flipflop f0(d, q, clk, rst_n);
```

← uses default value

```
my_flipflop #(12) f0(d, q, clk, rst_n);
```

← changes parameter to 12 for this instance

```
my_flipflop #(.WIDTH(12)) f0(d, q, clk, rst_n);
```

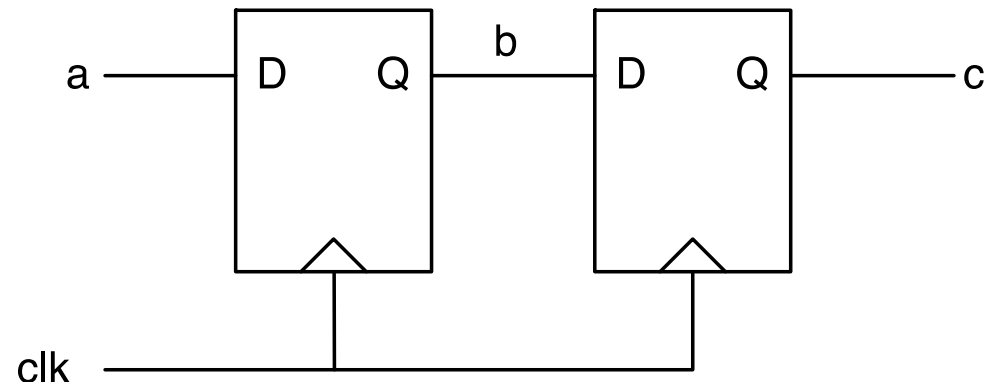
Non-Blocking Assignment $a \leq b$

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

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

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

- This models synchronous logic!



Non-Blocking vs. Blocking (1)

- 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
```

Non-Blocking vs. Blocking (2)

- Blocking models flow of values in wires and through gates in a combinational circuit
 - Output of multiplier is input to adder
 - That's why with blocking, (2) is evaluated after (1)
- Non-blocking assignments model relation between input and output of flip-flops
 - All FFs clocked together → all outputs take new values together
 - That's why (3) and (4) are evaluated in parallel

```
always_comb begin
  (1) temp = b * c;
  (2) f = a + temp;
end
```

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

Non-Blocking vs. Blocking (3)

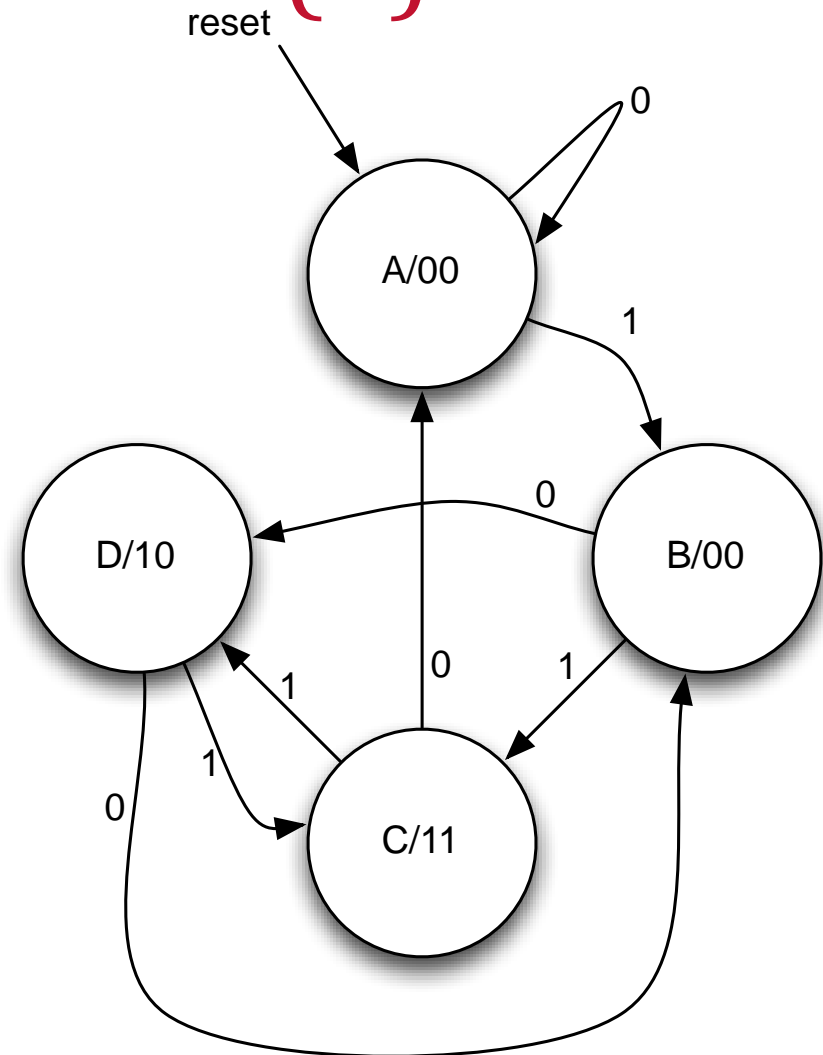
- Do not mix blocking and non-blocking assignments
- Use only blocking assignments in `always_comb`
- Use only non-blocking assignments in `always_ff`
- And keep their differences in mind

Design Example — Sequential

- Recall our previous example: $f = a + b * c$
 - b and c are 4 bits, a is 8 bits, and f is 9 bits
 - We built it as a combinational circuit
- Now, let's add registers at its inputs and outputs

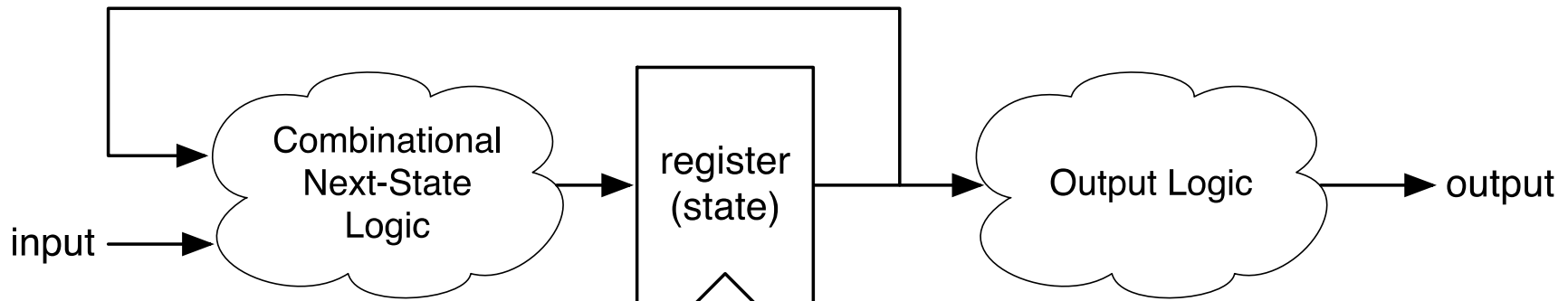
Finite State Machines (1)

- State names
- Output values
- Transition values
- Reset (initial) state



Finite State Machines (2)

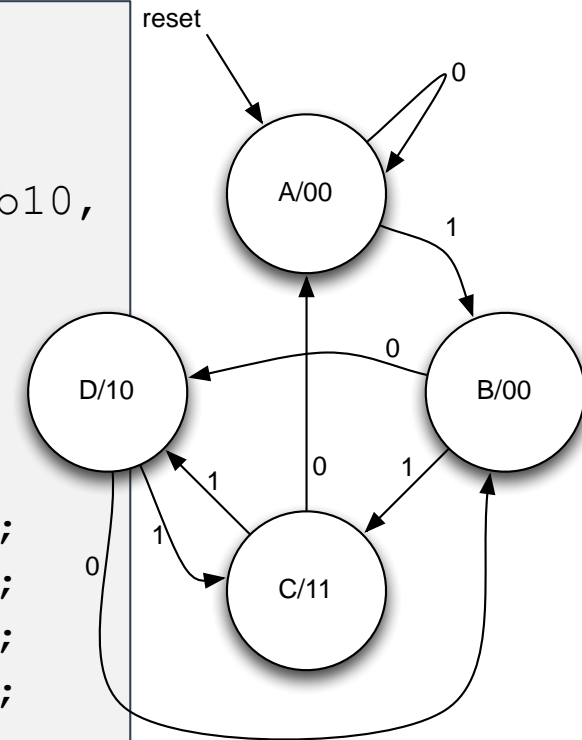
- What does an FSM look like when implemented in HW?



- Combinational logic and registers (things we already know how to do!)

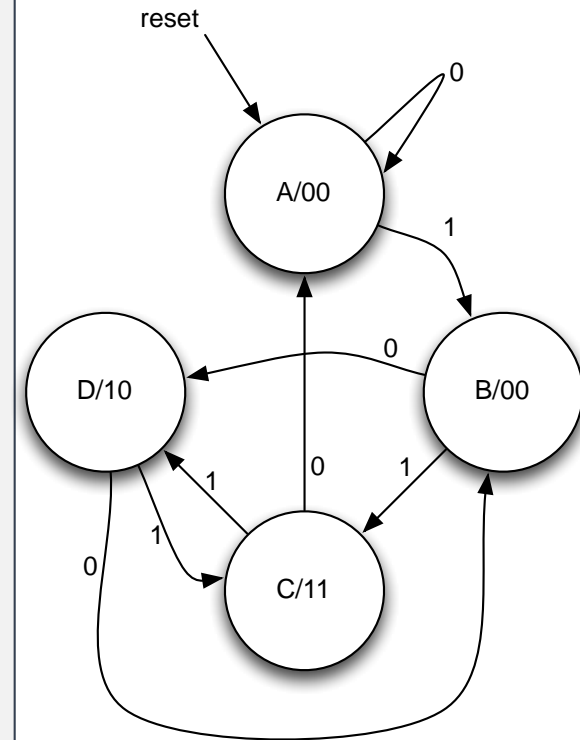
Full FSM Example (1)

```
module fsm(clk, rst, x, y);  
  input clk, rst, x;  
  output logic [1:0] y;  
  enum { STATEA=2'b00, STATEB=2'b01, STATEC=2'b10,  
        STATED=2'b11 } state, next_state;  
  
  // next state logic  
  always_comb begin  
    case(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  
  
  // ... continued on next slide
```



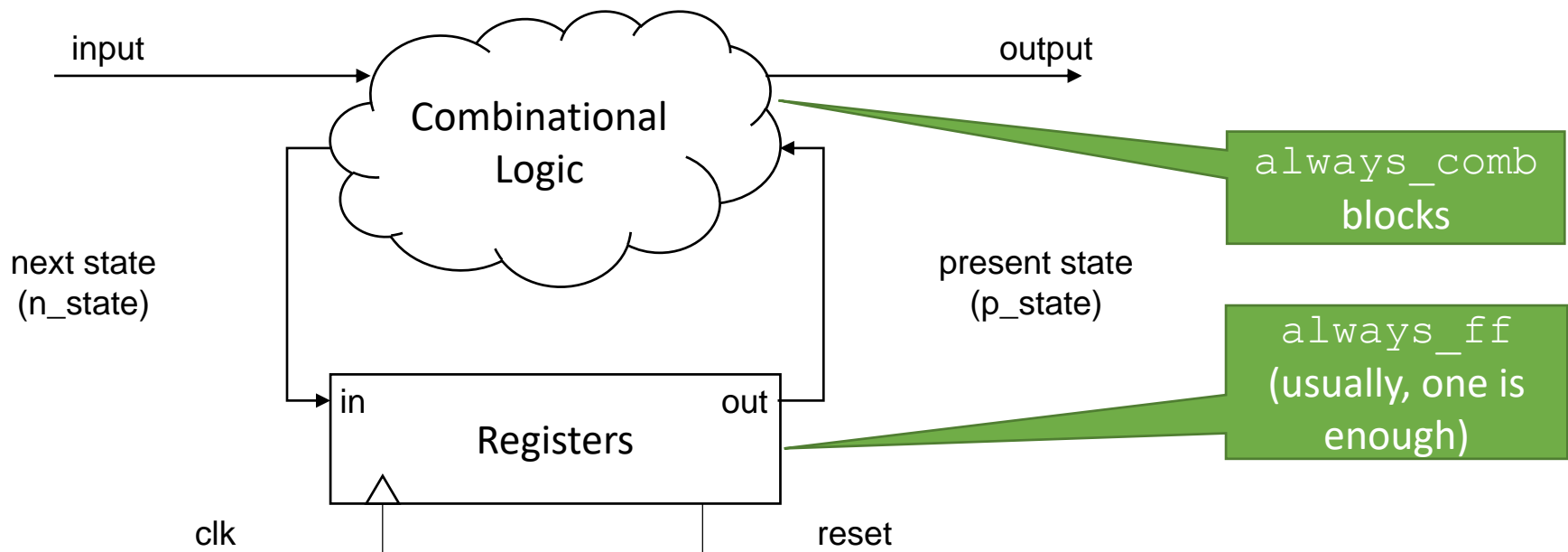
Full FSM Example (2)

```
// ... continued from previous slide
// register
always_ff @(posedge clk) begin
    if (rst)
        state <= STATEA;
    else
        state <= next_state;
end
// Output logic
always_comb begin
    case(state)
        STATEA: y = 2'b00;
        STATEB: y = 2'b00;
        STATEC: y = 2'b11;
        STATED: y = 2'b10;
    endcase
end
endmodule
```



Huffman Partitioning

- In my experience, for anything other than memories (SRAM arrays), you should code according to **Huffman Partitioning** of your module



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
```

display
(SystemVerilog's
printf)

Memory (Combinational read)

```
module mymemory(clk, data_in, data_out,  
                r_addr, w_addr, wr_en);  
    parameter WIDTH=16, LOGSIZE=8;  
    localparam SIZE=2**LOGSIZE;  
    input  [WIDTH-1:0] data_in;  
    output logic [WIDTH-1:0] data_out;  
    input  clk, wr_en;  
    input  [LOGSIZE-1:0] r_addr, w_addr;  
  
    logic [WIDTH-1:0] mem [SIZE-1:0];  
  
    assign data_out = mem[r_addr];  
  
    always_ff @(posedge clk) begin  
        if (wr_en)  
            mem[w_addr] <= data_in;  
    end  
endmodule
```

Combinational read

Synchronous write

Memory (Synchronous read)

```
module mymemory2(clk, data_in, data_out,
                 r_addr, w_addr, wr_en);
    parameter WIDTH=16, LOGSIZE=8;
    localparam SIZE=2**LOGSIZE;
    input  [WIDTH-1:0] data_in;
    output logic [WIDTH-1:0] data_out;
    input  clk, wr_en;
    input  [LOGSIZE-1:0] r_addr, w_addr;

    logic [WIDTH-1:0] mem [SIZE-1:0];

    always_ff @(posedge clk) begin
        data_out <= mem[r_addr];
        if (wr_en)
            mem[w_addr] <= data_in;
    end
endmodule
```

Synchronous read

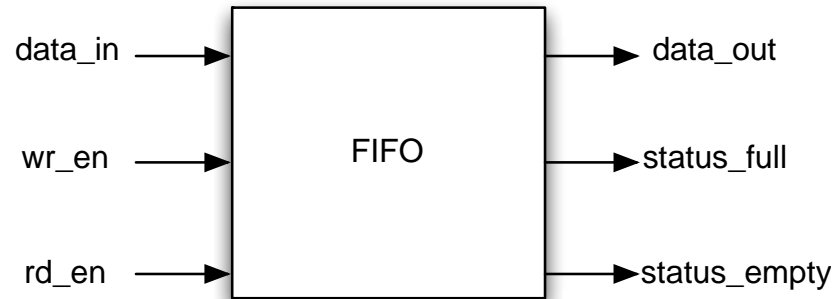
What happens if we try to read and write the same address?

Assertions

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

Example: A 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

Assertions

- A procedural statement that checks an expression when statement is executed

Use `$error`
to print error,
or `$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
```

- SV also has **Concurrent Assertions** that are continuously monitored and can express temporal conditions
 - Complex but very powerful
 - See <http://www.doulos.com/knowhow/sysverilog/tutorial/assertions/> for an introduction

DOs and DON'Ts to Keep in Mind (1)

- 1) Always try to picture the hardware that corresponds to your Verilog code (especially, `always` blocks)
 - If you can't, you're probably doing something wrong
 - Each hardware component is simple; the power is in their connection and parallelism
- 2) Have a reset signal that is connected to all your flip-flops
 - Do not make any assumptions about the initial state of your flip-flops
 - Instead, reset them explicitly
- 3) Avoid using loops to implement hardware functionality
 - Okay to use them for `display` or `assert` statements
 - But not for hardware functionality

DOs and DON'Ts to Keep in Mind (2)

- 4) Do not mix blocking and non-blocking assignments
 - Only use blocking assignments in `always_comb`
 - Only use non-blocking assignments in `always_ff`
 - And keep their differences in mind

- 5) Do not put any combinational logic in `always_ff`
 - `always_ff` should only model flip flops
 - Follow Huffman Partitioning rules

DOs and DON'Ts to Keep in Mind (3)

- 6) Big modules and `always` blocks are sign of bad design
 - Just like big functions
 - Keep each module simple to make it easy to test individually

- 7) Test, Test, Test
 - Test each module independently before connecting it to others
 - If a module's functionality is not independently-testable, it is probably a bad design