Verilog II

Ramkumar Ravi
03/09

Datatype Categories

- Net
  - Represents a physical wire
  - Describes structural connectivity
  - Assigned to in continuous assignment statements
  - Outputs of primitives and instantiated sub-modules
- Variables
  - Used in behavioral procedural blocks
  - Depending on how used, can represent either synchronous registers or wires in combinational logic

Variable Datatypes

- **reg** – scalar or vector binary values
- **integer** – 32 or more bits
- **time – time values represented in 64 bits (unsigned)**
- **real** – double precision values in 64 or more bits
- **realtime** – stores time as real (64-bit +)

- Assigned values only within a behavioral block
- CANNOT USE AS:
  - Output of primitive gate or instantiated submodue
  - LHS of continuous assignment
  - Input or inout port within a module

wire vs. **reg**

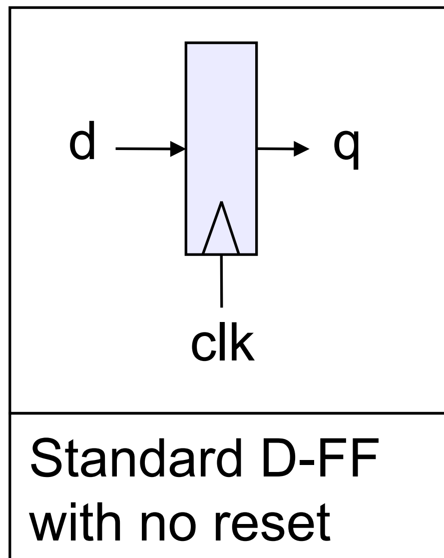
- Same “value” used both as 'wire and as 'reg'

```
module dff(q, d, clk);
      output reg q;                            // reg declaration
      input wire d, clk;                       // wire declaration, since module inputs
      always @ (posedge clk) q <= d;           // why is q reg and d wire
endmodule

module t_dff;
      wire q, clk;                             // now declared as wire
      reg d;                                   // now declared as reg
      dff FF (q, d, clk);                      // why is d reg and q wire
      clockgen myclk(clk);
      initial begin
            d = 0;
            #5 d = 1;
      end
endmodule
```

Memories and Multi-Dimensional Arrays

- A memory is an array of n-bit registers
  - **reg** [15:0] mem_name [0:127] // 128 16-bit words
  - **reg** array_2D [15:0] [0:127] // 2D array of 1-bit regs
- Can only access full word of memory
  - mem_name [122] = 35; // assigns word
  - mem_name [13][5] = 1; // illegal
  - array_2D[122] = 35 // illegal – *causes compilation error*
  - array_2D[13][5] = 1 // assigns bit
- Can use continuous assign to read bits
  - **assign** mem_val = mem[13] // get word in slot 13
  - **assign** out = mem_val[5] // get value in bit 5 of word
  - **assign** dataout = mem[addr];
  - **assign** databit = dataout[bitpos];

Implying Flops



Standard D-FF with no reset

```
reg q;

always @(posedge clk)
  q <= d;
```

It can be A vector too

```
reg [11:0] DAC_val;

always @(posedge clk)
  DAC_val <= result[11:0];
```

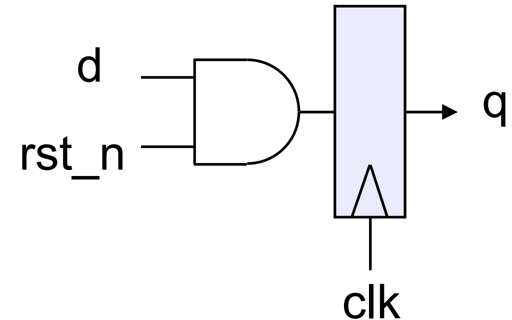
Be careful… Yes, a non–reset flop is smaller than a reset Flop, but most of the time you need to reset your flops.

Always error on the side of reseting the flop if you are at all uncertain.

Implying Flops (synchronous reset)

```
reg q;

always @(posedge clk)
  if (!rst_n)
    q <= 1'b0;      //synch reset
  else
    q <= d;
```

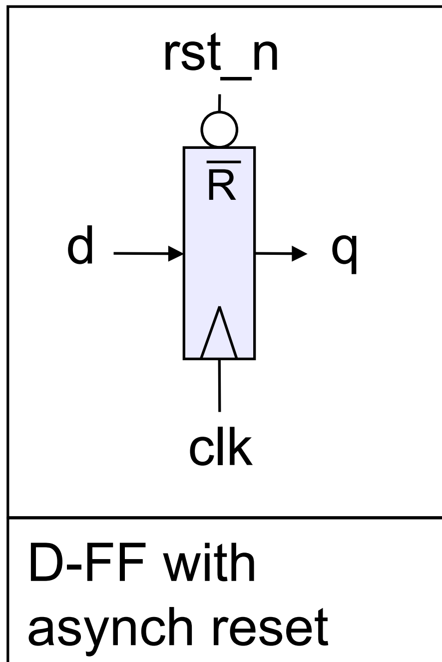
How does this synthesize?



Cell library might not contain a synch reset flop. Synthesis might combine 2 standard cells

Many cell libraries don't contain synchronous reset flops. This means the synthesizer will have to combine 2 (or more) standard cell to achieve the desired function… Hmmm? Is this efficient?

Implying Flops (asynch reset)




```
reg q;

always @(posedge clk or negedge rst_n)
   if (!rst_n)
      q <= 1'b0;
   else
      q <= d;
```

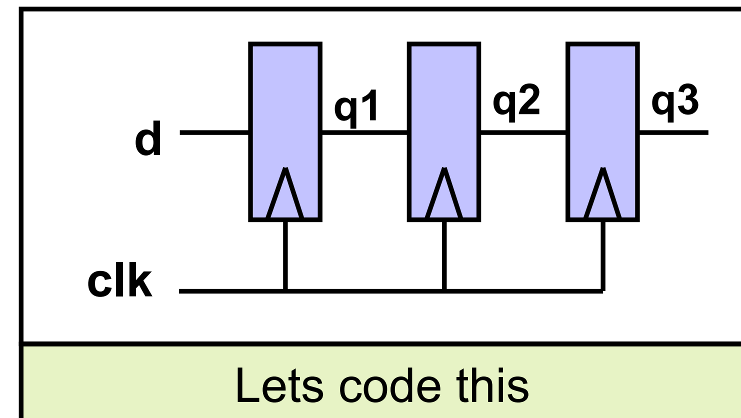
Cell libraries will contain an asynch reset flop. It is usually only slightly larger than a flop with no reset. This is probably your best bet for most flops.

Reset has its affect asynchronous from clock. What if reset is deasserting at the same time as a + clock edge? Is this the cause of a potential meta-stability issue?

Blocking assignment example

- Called blocking because….
  - The evaluation of subsequent statements <RHS> are **blocked**, until the <LHS> assignment of the current statement is completed.

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

input clk,d;
output q;
reg q;

always @(posedge clk) begin
  q1 = d;
  q2 = q1;
  q3 = q2;
end

endmodule
```



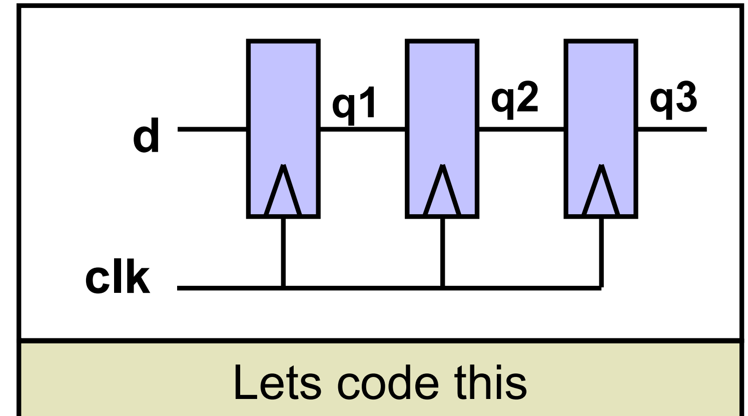
Simulate this in your head…

Remember blocking behavior of:
<LHS> assigned before
<RHS> of next evaluated.

Does this work as intended?

More on Non-Blocking

- Lets try that again



Lets code this

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

input clk,d;
output q;
reg q;

always @(posedge clk) begin
  q1 <= d;
  q2 <= q1;
  q3 <= q2;
End

endmodule;
```

With non-blocking statements the <RHS> of subsequent statements are **not blocked**. They are all evaluated simultaneously.

The assignment to the <LHS> is then scheduled to occur.

This will work as intended.

Verilog Stratified Event Queue [2]

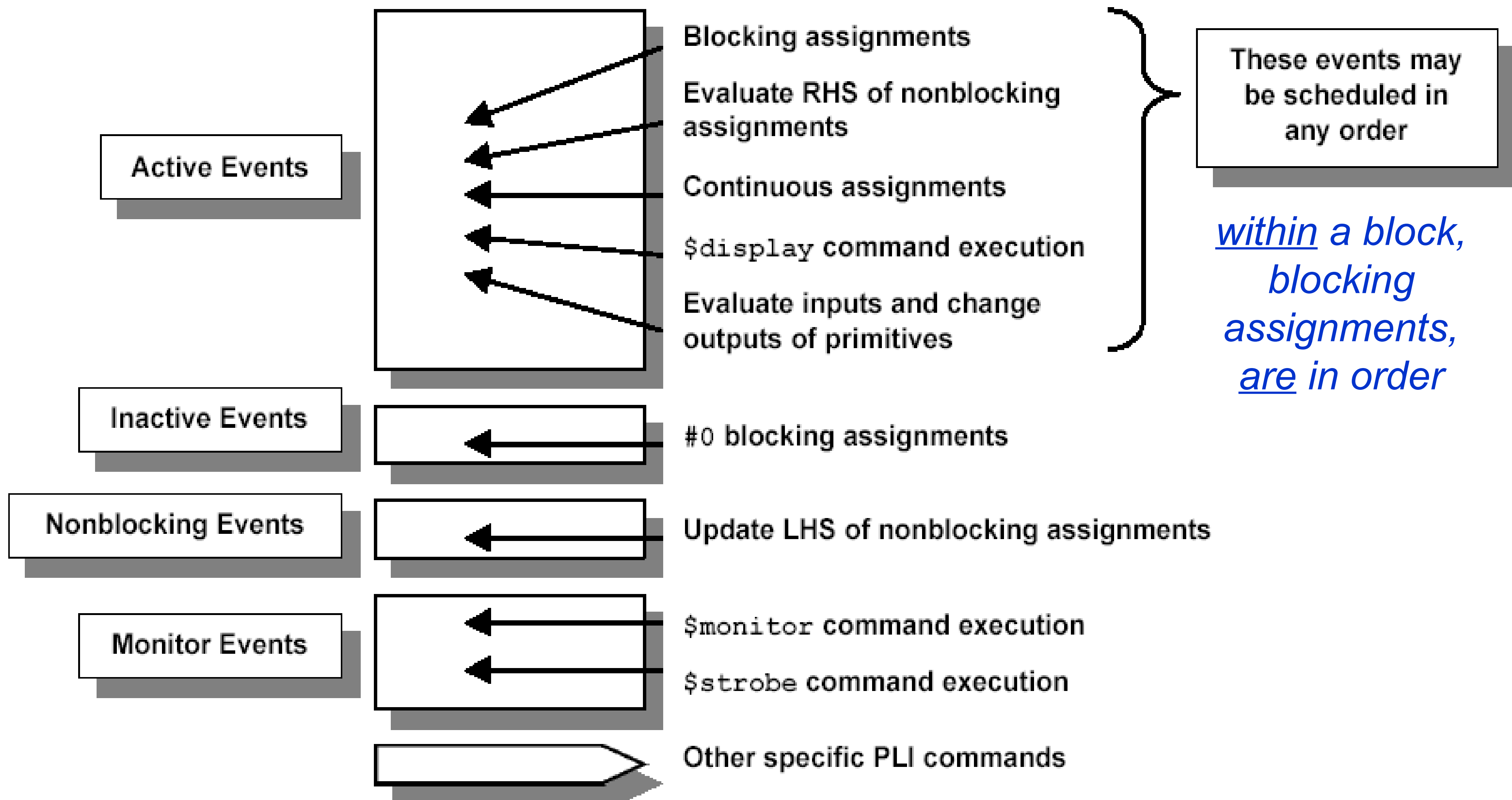


Figure 1 - Verilog "stratified event queue"

So Blocking is no good and we should always use Non-Blocking??

- Consider combinational logic

```
module ao4(z,a,b,c,d);

input a,b,c,d;
output z;

reg z,tmp1,tmp2;

always @(a,b,c,d) begin
  tmp1 <= a & b;
  tmp2 <= c & d;
  z <= tmp1 | tmp2;
end
endmodule
```

The inputs (a,b,c,d) in the sensitivity list change, and the always block is evaluated.

New assignments are scheduled for tmp1 & tmp2 variables.

A new assignment is scheduled for z using the **previous** tmp1 & tmp2 values.

Does this work?

If and Else If constructs

- In the next few slides we will introduce the if and else if constructs in verilog
- However note that Vcheck does not permit the use of these statements for this course
- Knowledge on **if** and **else if** constructs will help us in understanding the **case** statements which will be introduced later

if…else if…else statement

- General forms…

```
If (condition) begin
   <statement1>;
   <statement2>;
end
```

```
If (condition)
  begin
    <statement1>;
    <statement2>;
  end
else
  begin
    <statement3>;
    <statement4>;
  end
```

```
If (condition)
  begin
    <statement1>;
    <statement2>;
  end
else if (condition2)
  begin
    <statement3>;
    <statement4>;
  end
else
  begin
    <statement5>;
    <statement6>;
  end
```

Of course the compound statements formed with **begin/end** are optional.

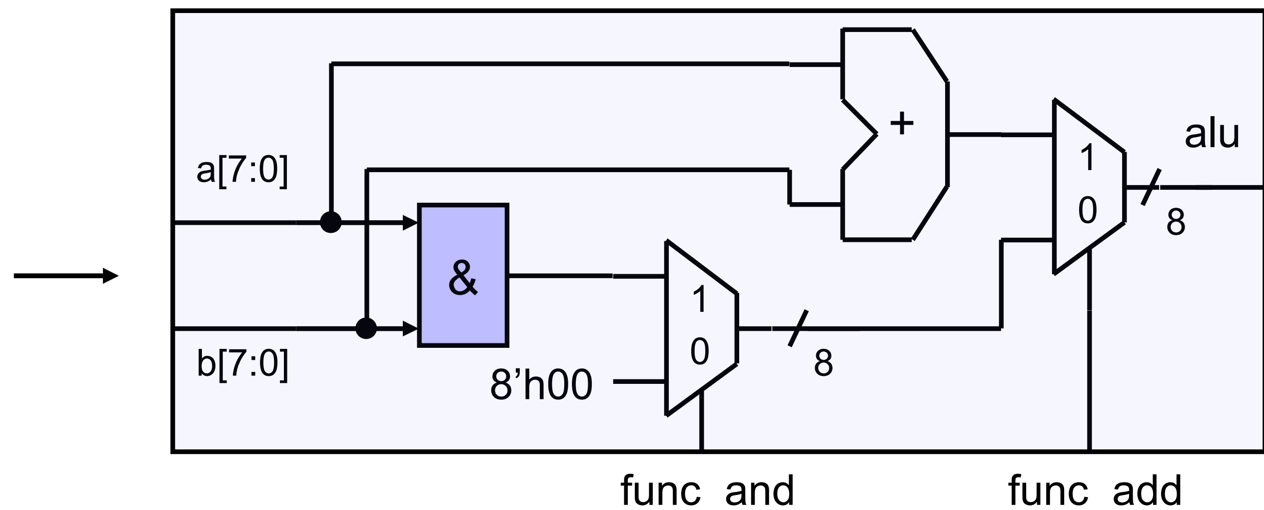
Multiple else if's can be strung along indefinitely

How does and **if…else if…else** statement synthesize?

- Does not conditionally "execute" block of "code"
- Does not conditionally create hardware!
- It makes a <u>multiplexer</u> or selecting logic
- Generally:
  - ✓ Hardware for both paths is created
  - ✓ Both paths "compute" simultaneously
  - ✓ The result is selected depending on the condition

```
If (func_add)
   alu = a + b;
else if (func_and)
   alu = a & b;
Else
   alu = 8'h00;
```

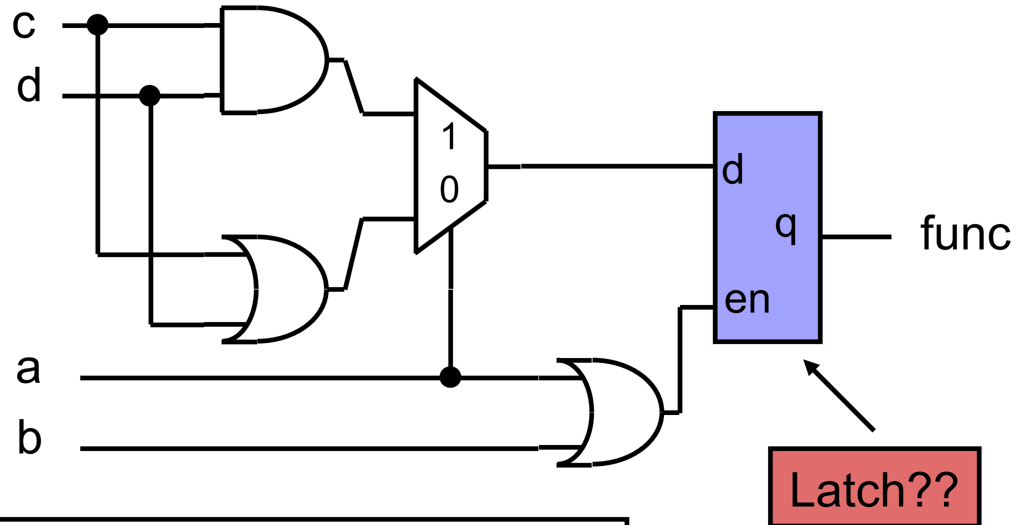
→



if statement synthesis (continued)

```
if (a)
  func = c & d;
else if (b)
  func = c | d;
```

How does this synthesize?



Latch??

What you ask for is what you get!

func is of type register. When neither **a** or **b** are asserted it didn't not get a new value.

That means it must have remained the value it was before.

That implies memory…i.e. a **latch!**

Always have an **else** to any **if** to avoid unintended latches.

case Statements

- Verilog has three types of case statements:
  - **case**, **casex**, and **casez**

- Performs bitwise match of expression and case item
  - Both must have same bitwidth to match!
- **case**
  - Can detect **x** and **z**! (good for testbenches)
- **casez**
  - Uses **z** and **?** as “don’t care” bits in case items and expression
- **casex**
  - Uses **x**, **z**, and **?** as “don’t care” bits in case items and expression

Case statement (general form)

```
case (expression)
   alternative1 : statement1; // any of these statements could
   alternative2 : statement2; // be a compound statement using
   alternative3 : statement3; // begin/end
   default : statement4       // always use default for synth stuff
endcase
```

```
parameter AND = 2'b00;
parameter OR = 2'b01;
parameter XOR = 2'b10;

case (alu_op)
   AND      : alu = src1 & src2;
   OR       : alu = src1 | src2;
   XOR      : alu = src1 ^ src2;
   default  : alu = src1 + src2;
endcase
```

Why always have a default?

Same reason as always having an else with an if statement.

All cases are specified, therefore no unintended latches.

Using **case** To Detect **x** And **z**

- Only use this functionality in a testbench!

- Example taken from Verilog-2001 standard:

```
case (sig)
        1'bz:           $display("Signal is floating.");
        1'bx:           $display("Signal is unknown.");
        default:        $display("Signal is %b.", sig);
endcase
```

casex Statement

- Uses **x**, **z**, and **?** as single-bit wildcards in case item and expression
- Uses first match encountered

```
always @ (code) begin
        casex (code)                    // case expression
                2'b0?: control = 8'b00100110; // case item1
                2'b10: control = 8'b11000010; // case item 2
                2'b11: control = 8'b00111101; // case item 3
        endcase
end
```

- What is the output for code = 2'b01?
- What is the output for code = 2'b1x?

casez Statement

- Uses z, and ? as single-bit wildcards in case item and expression

```
always @ (code) begin
   casez (code)
        2'b0?: control = 8'b00100110; // item 1
        2'bz1: control = 8'b11000010; // item 2
   default: control = 8b'xxxxxxxx; // item 3
   endcase
end
```

- What is the output for code = 2b'01?
- What is the output for code = 2b'zz?

Synthesis Of **x** And **z**

Only allowable uses of **x** is as “don’t care”, since **x** cannot actually exist in hardware

in **casex**

in defaults of conditionals such as **:**

- The **else** clause of an **if** statement
- The **default** selection of a **case** statement

Only allowable use of **z**:

Constructs implying a 3-state output

- Of course it is helpful if your library supports this!

Don’t Cares

x, **?**, or **z** within case item expression in **casex**

Does <u>not</u> actually output “don’t cares”!

Values for which input comparison to be ignored

Simplifies the case selection logic for the synthesis tool

```
casex (state)
   3'b0??: out = 1'b1;
   3'b10?: out = 1'b0;
   3'b11?: out = 1'b1;
endcase
```

state[2] \ state[1:0]	00	01	11	10
0	1	1	1	1
1	0	0	1	1

$$out = \overline{state[0]} + state[1]$$

Use of Don't Care in Outputs

Can really reduce area

```
case (state)
   3'b001: out = 1'b1;
   3'b100: out = 1'b0;
   3'b110: out = 1'b1;
   default: out = 1'b0;
endcase
```

state[2] \ state[1:0]	00	01	11	10
0	0	1	0	0
1	0	0	0	1

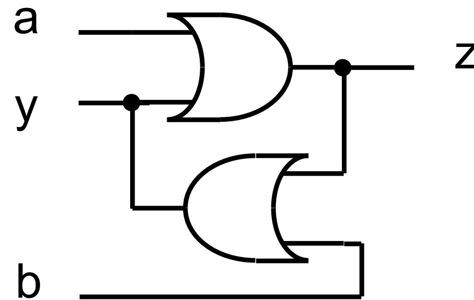
```
case (state)
   3'b001: out = 1'b1;
   3'b100: out = 1'b0;
   3'b110: out = 1'b1;
   default: out = 1'bx;
endcase
```

state[2] \ state[1:0]	00	01	11	10
0	x	1	x	x
1	0	x	x	1

Unintentional Latches

- Avoid structural feedback in continuous assignments, combinational always

```
assign z = a | y;

assign y = b | z;
```



- Avoid incomplete sensitivity lists in combinational **always**
- For conditional assignments, either:
  - Set default values before statement
  - Make sure LHS has value in every branch/condition
- For warning, set hdlin_check_no_latch true before compiling

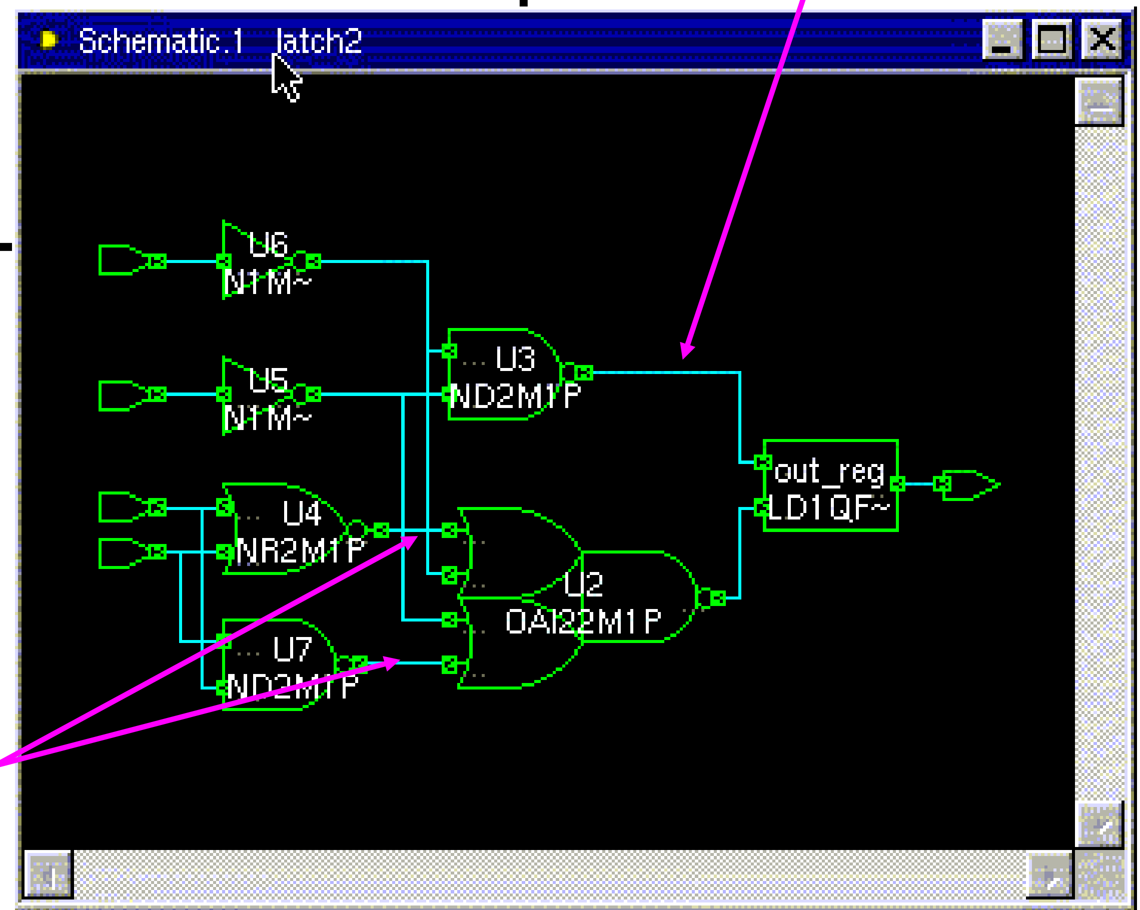
Synthesis Example [1]

```
module Hmmm(input a, b, c, d, output reg out);
always @(a, b, c, d) begin
   if (a) out = c | d;
   else if (b) out = c & d;
end
endmodule
```

a|b enables latch

How will this synthesize?

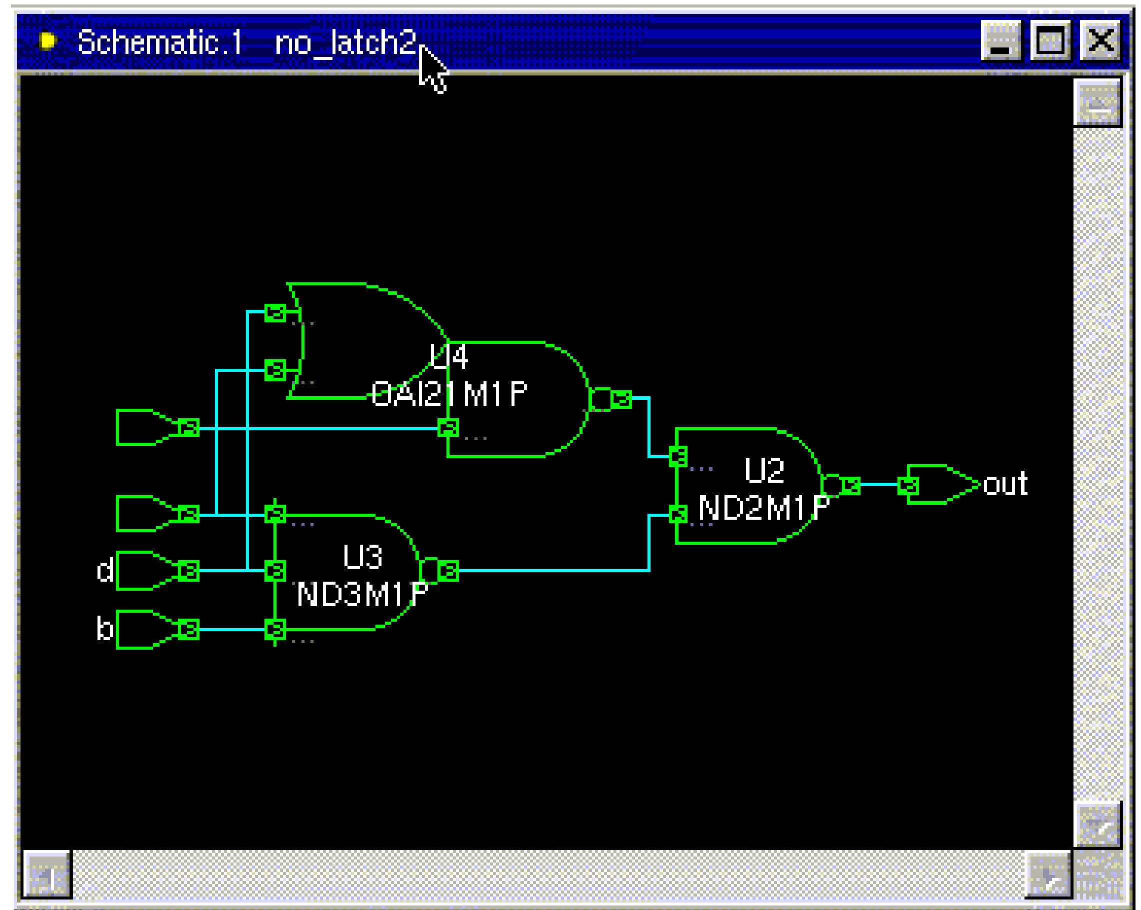
Area = 44.02



Either c|d or c&d are passed through an inverting mux depending on state of a / b

Synthesis Example [2]

```
module Better(input a, b, c, d, output reg out);
always @(a, b, c, d) begin
   if (a) out = c | d;
   else if (b) out = c & d;
   else out = 1'b0;
end
endmodule
```



Perhaps what you meant was that if not a or b then out should be zero??

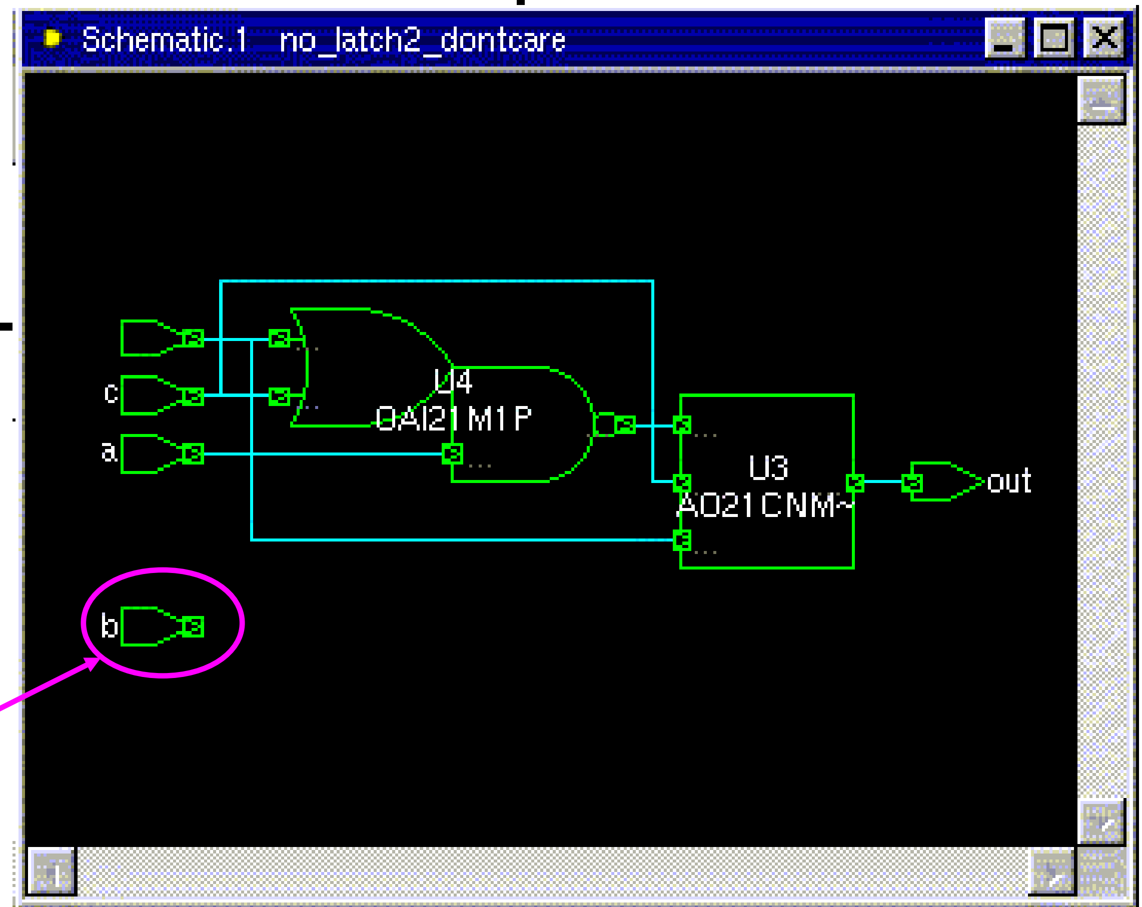
Area = 16.08

Does synthesize better…no latch!

Synthesis Example [3]

Area = 12.99

```
module BetterYet(input a, b, c, d, output reg out);
always @(a, b, c, d) begin
   if (a) out = c | d;
   else if (b) out = c & d;
   else out = 1'bx;
end
endmodule
```

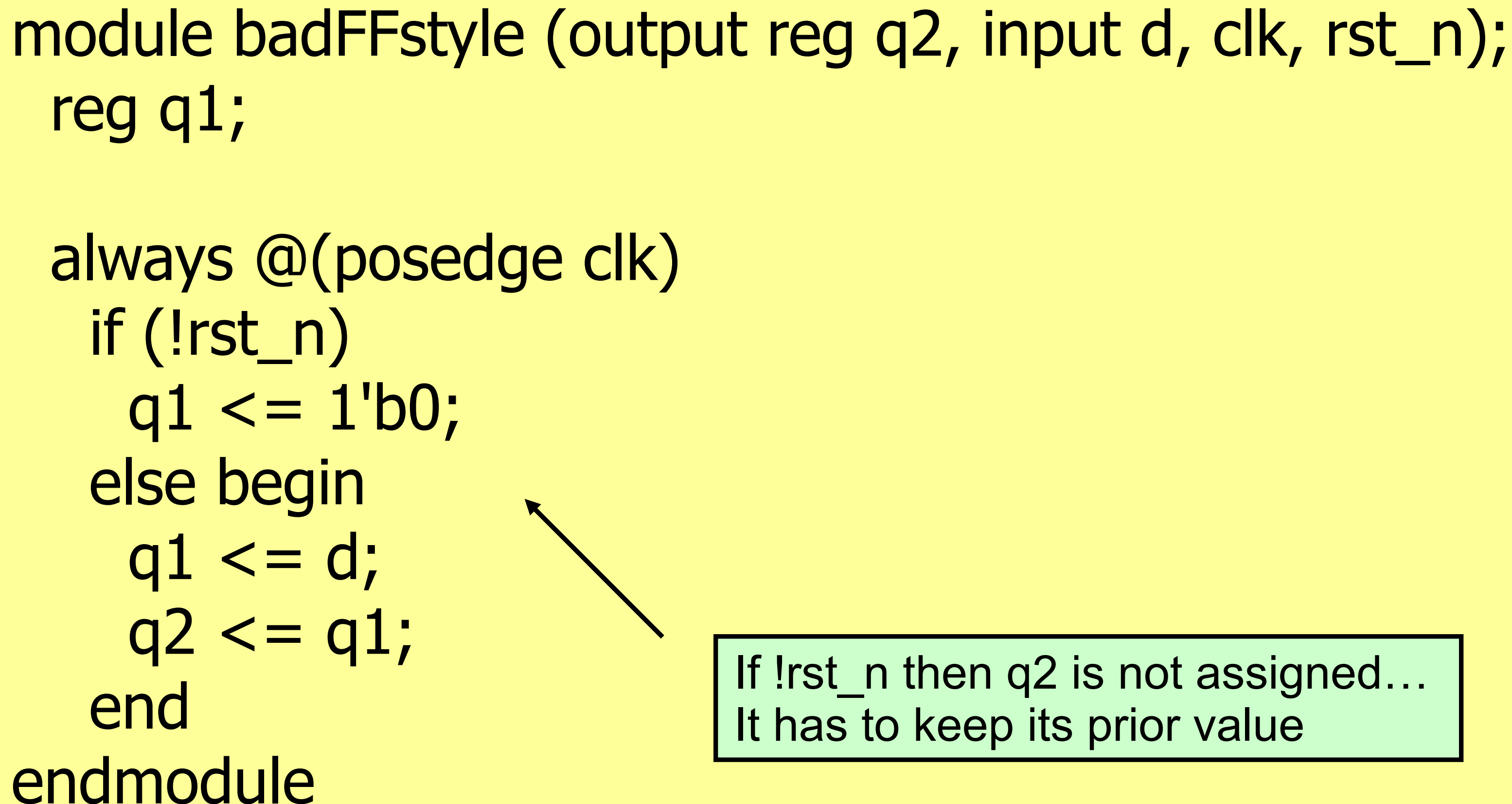


But perhaps what you meant was if neiter a nor b then you really don't care what out is.

Hey!, Why is b not used?

Mixing Flip-Flop Styles (1)

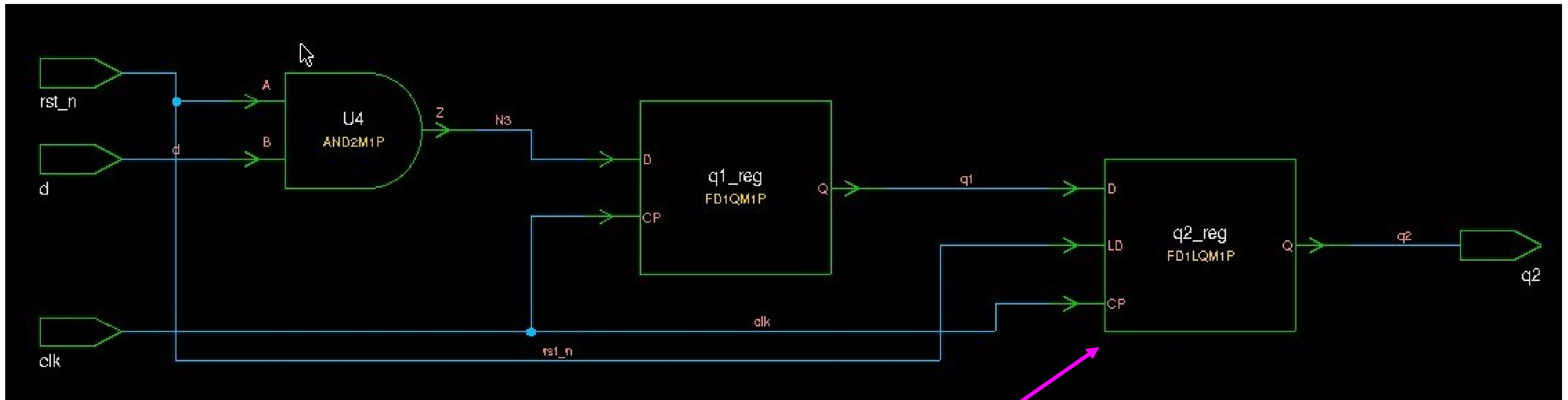
- What will this synthesize to?



```
module badFFstyle (output reg q2, input d, clk, rst_n);
  reg q1;

  always @(posedge clk)
    if (!rst_n)
      q1 <= 1'b0;
    else begin
      q1 <= d;
      q2 <= q1;
    end
endmodule
```

Flip-Flop Synthesis (1)

- Area = 59.0



Note: q2 uses an enable flop (has mux built inside) enabled off rst_n

Mixing Flip-Flop Styles (2)

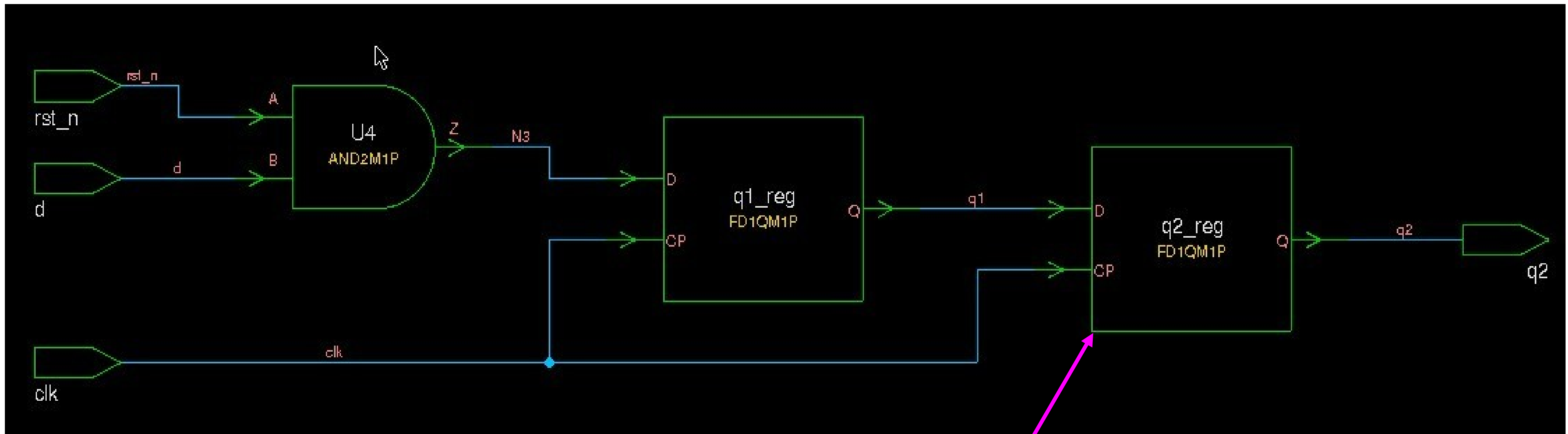
```
module goodFFstyle (output reg q2, input d, clk, rst_n);
  reg q1;

  always @(posedge clk)
    if (!rst_n) q1 <= 1'b0;
    else q1 <= d;

  always @(posedge clk)
    q2 <= q1;

endmodule
```

Only combine like flops (same reset structure) in a single always block.

If their reset structure differs, split into separate always blocks as shown here.

Flip-Flop Synthesis (2)

- Area = 50.2 (85% of original area)



Note: q2 is now just a simple flop as intended

What Have We Learned?

1) Sequential elements (flops & latches) should be inferred using non-blocking “**<=**“ assignments

1) Combinational logic should be inferred using blocking “**=**“ statements.

1) Blocking and non-Blocking statements should not be mixed in the same **always** block.

1) Plus 5 other guidelines of good coding outlined in the Cummings SNUG paper.

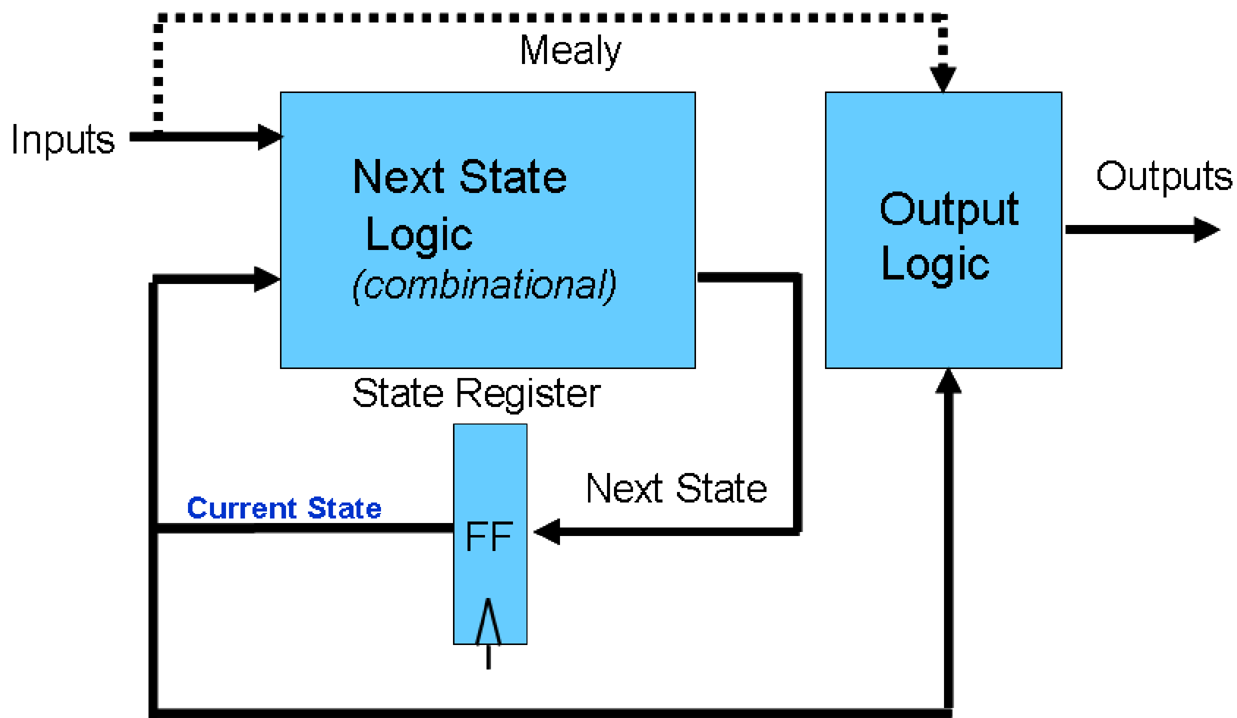
Parameters

```
module adder(a,b,cin,sum,cout);

parameter WIDTH = 8; // default is 8

input [WIDTH-1:0] a,b;
input cin;
output [WIDTH-1:0] sum;
output cout;

assign {cout,sum} =  a  +  b + cin

endmodule
```

```
module alu(src1,src2,dst,cin,cout);
  input [15:0] src1,src2;
  …
  //////////////////////////////////
  // Instantiate 16-bit adder //
  //////////////////////////////////
  adder #(16) add1(.a(src1),.b(src2),
                      .cin(cin),.cout(cout),
                      .sum(dst));

  …
endmodule
```

Instantiation of module can override a parameter.

State Machines



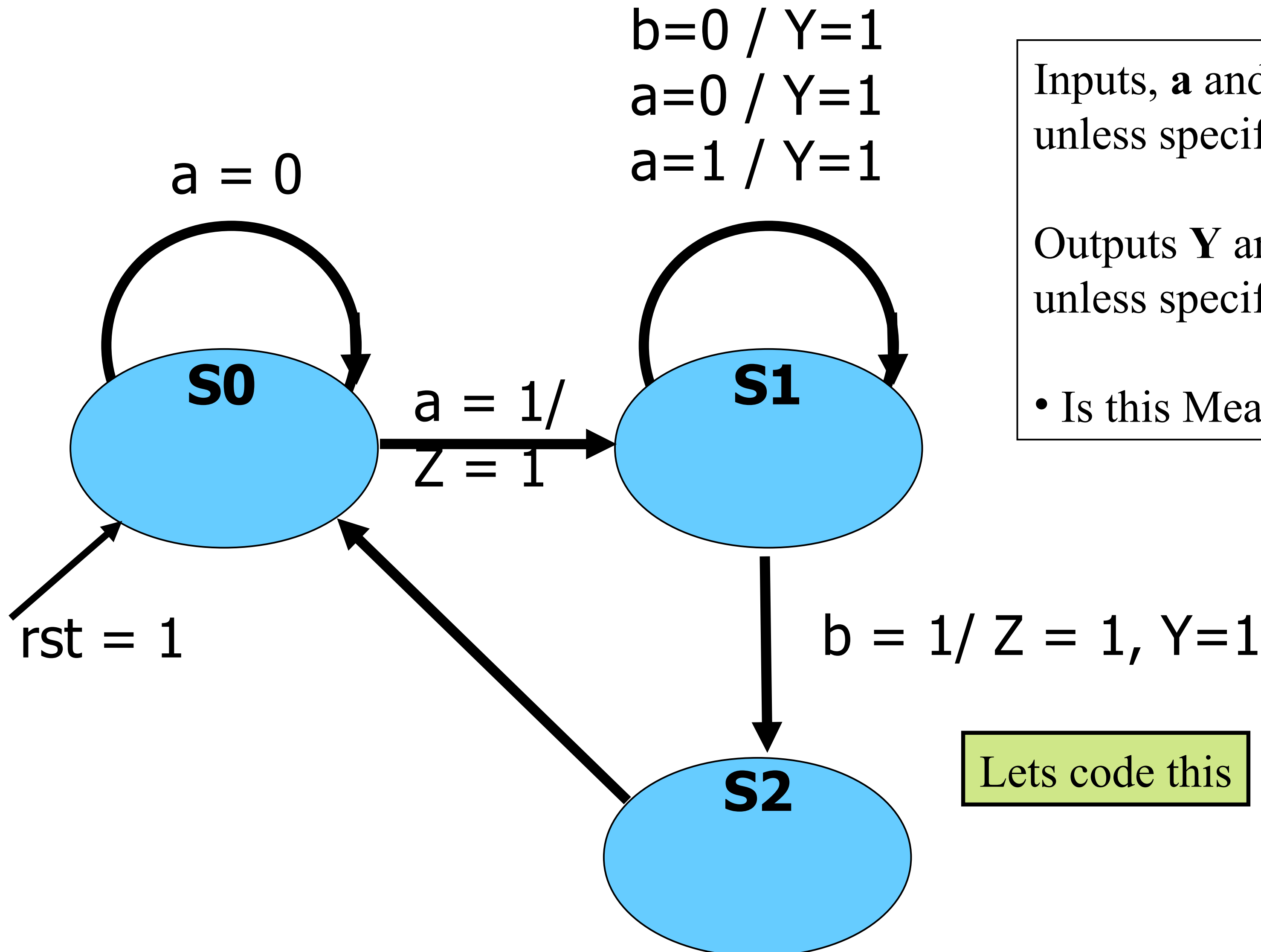
State Machines:

- Next State and output logic are combinational blocks, which have outputs dependent on the current state.

- The current state is, of course, stored by a FF.

- **What is the best way to code State Machines?:**
  - ✓ Best to separate combinational (blocking) from sequential (non-blocking)
  - ✓ Output logic and state transition logic can be coded in same always block since they have the same inputs
  - ✓ Output logic and state transition logic are ideally suited for a case statement

State Diagrams



Inputs, **a** and **b** are 0, unless specified otherwise

Outputs **Y** and **Z** are 0, unless specified otherwise.

- Is this Mealy or Moore?

Lets code this

SM Coding

```
module fsm(clk,rst,a,b,Y,Z);

input clk,rst,a,b;
output Y,Z;

parameter S0 = 2'b00,
          S1 = 2'b01,
          S2 = 2'b10;

reg [1:0] state,nxt_state;

always @(posedge clk, posedge rst)
  if (rst)
    state <= S0;
  else
    state <= nxt_state;
```

What problems do we have here?

```
always @ (state,a,b)
  case (state)
    S0 : if (a) begin
              nxt_state = S1;
              Z = 1; end
         else
              nxt_state = S0;
    S1 : begin
           Y=1;
           if (b) begin
              nxt_state = S2;
              Z=1; end
           else
              nxt_state = S1;
         end
    S2 : nxt_state = S0;
  endcase
endmodule
```

SM Coding (2nd try of combinational)

```
always @ (state,a,b)
  nxt_state = S0;  // default to reset
  Z = 0;           // default outputs
  Y = 0;           // to avoid latches

  case (state)
    S0 : if (a) begin
           nxt_state = S1;
           Z = 1;
         end
    S1 : begin
           Y=1;
           if (b) begin
             nxt_state = S2;
             Z=1; end
           else nxt_state = S1;
         end
    default : nxt_state = S0;
  endcase
endmodule
```

Defaulting of assignments and having a default to the case is highly recommended!

SM Coding Guidlines

1) Keep state assignment in separate always block using non-blocking “<=“ assignment
2) Code state transition logic and output logic together in a always block using blocking assignments
3) Assign default values to all outputs, and the *nxt_state* registers. This helps avoid unintended latches
4) Remember to have a default to the case statement.
   - Default should be (if possible) a state that transitions to the same state as reset would take the SM to.
   - Avoids latches
   - Makes design more robust to spurious electrical/cosmic events.

Priority Encoder With **casex**

```
module priority_encoder (output reg [2:0] Code, output valid_data,
   input [7:0] Data);

assign valid_data = |Data; // "reduction or" operator
always @  (Data)
  // encode the data
  casex (Data)
      8'b1xxxxxxx   : Code = 7;
      8'b01xxxxxx   : Code = 6;
      8'b001xxxxx   : Code = 5;
      8'b0001xxxx   : Code = 4;
      8'b00001xxx         : Code = 3;
      8'b000001xx         : Code = 2;
      8'b0000001x   : Code = 1;
      8'b00000001   : Code = 0;
      default    : Code = 3'bxxx; // should be at least one 1, don't care
  endcase
endmodule
```

Exhaustive testing with for loops

For combinational designs w/ up to 8 or 9 inputs

Test ALL combinations of inputs to verify output

Could enumerate all test vectors, but don't…

Generate them using a "for" loop!

```
 reg [4:0] x;
 initial begin
         for (x = 0; x <  16; x = x + 1)                    #5;
// need a delay here!
 end
```

Need to use "reg" type for loop variable? Why?

Why Loop Vector Has Extra Bit

Want to test all vectors 0000 to 1111

```
reg [3:0] x;
initial begin
        for (x = 0; x <  16; x = x + 1)
        #5;   // need a delay here!
    end
```

If x is 4 bits, it only gets up to 1111 => <u>15</u>

1100 => 1101 => 1110 => 1111 => 0000 => 0001

x is never >= 16… so loop goes forever

while loops

- Executes until boolean condition is not true
  - If boolean expression false from beginning it will never execute loop

```
reg [15:0] flag;
reg [4:0] index;

initial begin
  index=0;
  found=1'b0;
  while ((index<16) && (!found)) begin
    if (flag[index]) found = 1'b1;
    else index = index + 1;
  end
  if (!found) $display("non-zero flag bit not found!");
  else $display("non-zero flag bit found in position %d",index);
end
```

Handy for cases where loop termination is a more complex function.

Like a search

repeat Loop

- Good for a fixed number of iterations
  - Repeat count can be a variable but…
    - It is only evaluated when the loops starts
    - If it changes during loop execution it won't change the number of iterations

- Used in conjunction with @(posedge clk) it forms a handy & succinct way to wait in testbenches for a fixed number of clocks

```
initial begin
  inc_DAC = 1'b1;
  repeat(4095) @(posedge clk);   // bring DAC right up to point of rollover
  inc_DAC = 1'b0;
  inc_smpl = 1'b1;
  repeat(7)@(posedge clk);   // bring sample count up to 7
  inc_smpl = 1'b0;
end
```

forever loops

- We got a glimpse of this already with clock generation in testbenches.
- Only a **$stop**, **$finish** or a specific **disable** can end a **forever** loop.

```
initial begin
  clk = 0;
  forever #10 clk = ~ clk;
end
```

Clock generator is by far the most common use of a forever loop