

Digital Design with Verilog

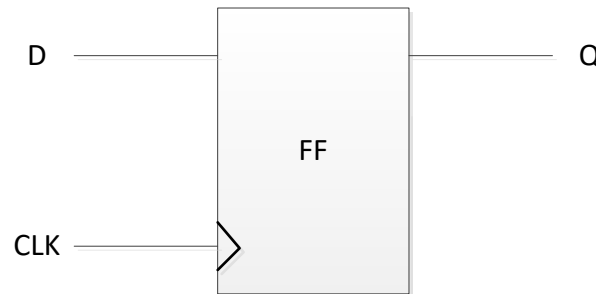
Design Mantras

- ❑ One clock, one edge, Flip-flops only
- ❑ Design BEFORE coding
- ❑ Behavior implies function
- ❑ Clearly separate control and datapath



Basic Verilog Constructs

- ❑ Flip Flop
 - Behavior
 - For every positive edge of the clock Q changes to become equal to D
- ❑ Write behavior as “code”
- ❑ `always @()`
 - Triggers execution of the following code block
 - `()` called “sensitivity list”
 - Describes when execution triggered



Mantra # 1

- ❑ Methodology for purposes of ASIC Design class
 - If at all possible one-edge of one clock
 - If you need multiple clocks, they must have a common root and be related by factors of 2
- ❑ E.g. Root clock : Tclock = 5 ns
 - This is BEST: Tclock only!!
 - This is OK
 - Tclock, Tclock10, Tclock20
 - Tclock10 = 10 ns (Tclock*2)
 - Tclock20 = 20 ns (Tclock*4)
 - This is NOT OK
 - Tclock, Tclock15, Tclock17
 - Tclock15 = 15 ns (Tclock*3)
 - Tclock17 = 17 ns (??)

Mantra # 3

❑ Behavior implies function

- Determine the behavior described by the Verilog code
- Choose the hardware with the matching behavior

```
always @(posedge clk)
    Q <= D;
```

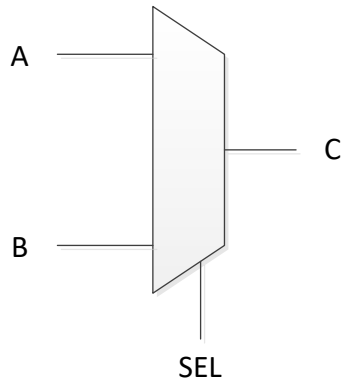
❑ Code behavior

- Q re-evaluated every time there is a rising edge of the clock
- Q remains unchanged between rising edges

❑ This behavior describes an edge-triggered Flip-Flop

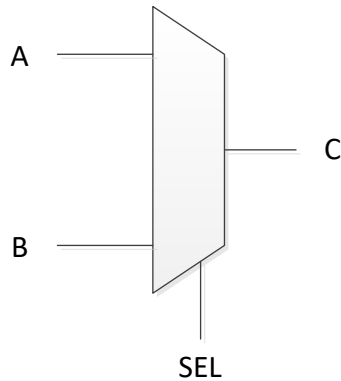
More Examples

- ❑ What is the behavior and code for this schematic?



More Examples

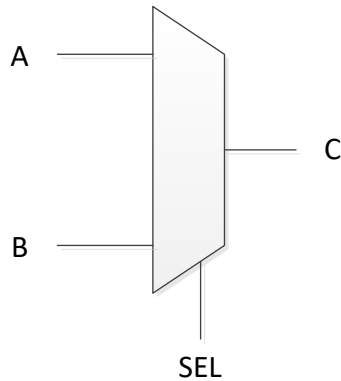
- ❑ What is the behavior and code for this schematic?



```
always @(A,B,SEL)
    if (SEL) C = A;
    else C = B;
```

More Examples

- ❑ What is the behavior and code for this schematic?



```
always @(A,B,SEL)
    if (SEL) C = A;
    else C = B;
```

```
assign C = (SEL)? A: B;
```


Flip - Flops

- ❑ Every variable assigned in a block starting with

```
always @(posedge clk) or  
always @(negedge clk)
```

becomes the output of an edge-triggered flip –flop

- ❑ This is the only way to build flip-flops

Verilog Module for Flip-Flop

```
module flipflop (d, clk, q);  
    input d, clk;  
    output q;  
    reg ff;  
  
    always @(posedge clk) begin  
        ff <= d;  
    end  
    assign q = ff;  
endmodule
```

Module name

Connected ports

Port declarations

Local variables

Code segments

End module

Verilog Module for Flip-Flop

```
module flipflop (d, clk, q);  
  input d, clk;  
  output q;  
  reg ff;  
  
  always @(posedge clk) begin  
    ff <= d;  
  end  
  assign q = ff;  
endmodule
```



Module name

Connected ports

Port declarations

Local variables

Code segments

End module

Verilog Module for Flip-Flop

```
module flipflop (d, clk, q);  
  input d, clk;  
  output q;  
  reg ff;  
  
  always @(posedge clk) begin  
    ff <= d;  
  end  
  assign q = ff;  
endmodule
```

Module name

Connected ports

Port declarations

Local variables

Code segments

End module

Verilog Module for Flip-Flop

```
module flipflop (d, clk, q);  
  input d, clk;  
  output q;  
  reg ff;  
  
  always @(posedge clk) begin  
    ff <= d;  
  end  
  assign q = ff;  
endmodule
```

Module name

Connected ports

Port declarations

Local variables

Code segments

End module

Verilog Module for Flip-Flop

```
module flipflop (d, clk, q);  
  input d, clk;  
  output q;  
  reg ff;  
  
  always @(posedge clk) begin  
    ff <= d;  
  end  
  assign q = ff;  
endmodule
```

Module name

Connected ports

Port declarations

Local variables

Code segments

End module

Verilog Module for Flip-Flop

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

```
  input d, clk;
```

```
  output q;
```

```
  reg ff;
```

```
  always @(posedge clk) begin
```

```
    ff <= d;
```

```
  end
```

```
  assign q = ff;
```

```
endmodule
```

Module name

Connected ports

Port declarations

Local variables

Code segments

End module

Verilog Module for Flip-Flop

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

```
  input d, clk;
```

```
  output q;
```

```
  reg ff;
```

```
  always @(posedge clk) begin
```

```
    ff <= d;
```

```
  end
```

```
  assign q = ff;
```

```
endmodule
```

Module name

Connected ports

Port declarations

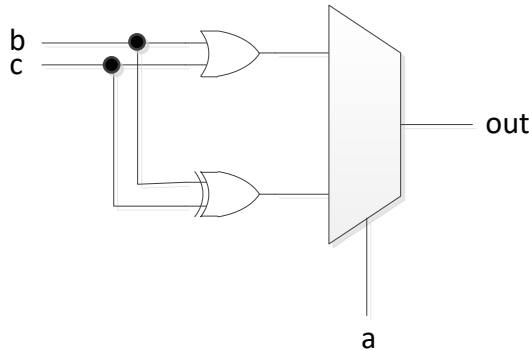
Local variables

Code segments

End module

Combinational Logic

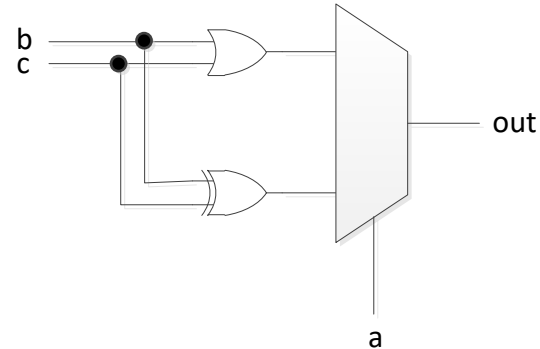
- ❑ Combinational logic example
- ❑ How would you describe the behavior of this function in words?



- ❑ And in code?

Behavior → Function

```
always @(a or b or c)
  if (a) out = b ^ c;
  else out = b | c;
```

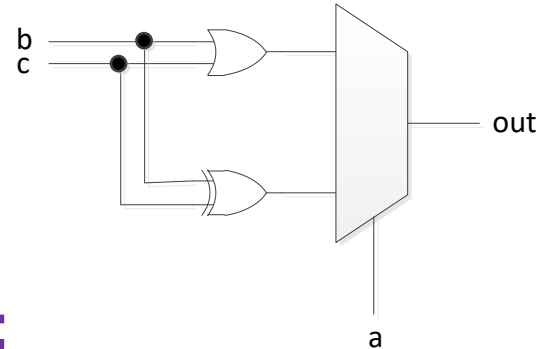


- ❑ All logical inputs in sensitivity list
- ❑ If, else → multiplexer
- ❑ Behavior: whenever input changes $out = \text{mux XOR or OR}$
- ❑ Same behavior as combinational logic

Behavior → Function

- ❑ Verilog has a short hand way to capture combinational logic
- ❑ Called “continuous assignment”

`assign foo = a ? b^c : b | c;`



- ❑ LHS re-evaluated whenever anything in RHS changes

**`f = a ? d : e; same as`
`if (a) f=d else f=e;`**

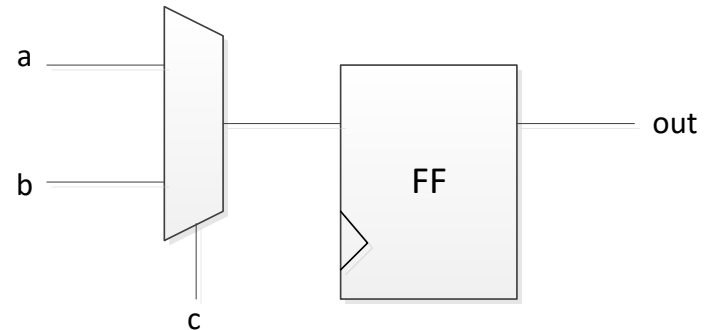
Procedural blocks

- ❑ Statement block starting with an “always@” statement is called a procedural block
- ❑ Statements in block are generally executed in sequence (i.e. procedurally)

Input Logic to Flip-Flops

- ❑ Can include some combinational logic in FF procedural block

```
always @(posedge clk)
    if (c)
        out <= b;
    else
        out <= a;
```



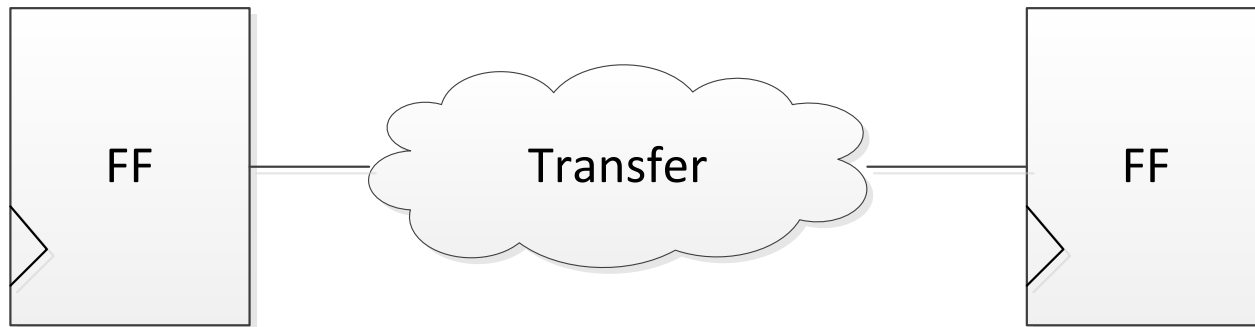
- ❑ Behavior → function
 - Out is reevaluated on every clock edge → output of FF
 - If; else → MUX

RTL Coding Styles

- ❑ Three coding styles
 - always @(? edge clock) → FF and input logic
 - always @(*) → Combinational logic (CL)
 - assign a = → Combinational logic (CL)
- ❑ The hard part is **not** coding but design

Always Design Before Coding

- ❑ Must code at register transfer level
- ❑ Register and “transfer” (combinational) logic must be worked out before coding can start



Design Before Coding

- ❑ Automatic synthesis does not relieve you of logic design
 - it does relieve you of
 - Logic optimization
 - Timing calculations and control
 - In many cases, detailed logic design
- ❑ If you don't design before coding, you are likely to end up with the following
 - A very slow clock (long critical path)
 - Poor performance and large area
 - Non-synthesizable Verilog
 - Many HDL lint errors

Avoid Temptation

❑ Temptation # 1

- “Verilog looks like C, so I’ll write the algorithm in C and turn it into Verilog with a few always @ statements”
- Usual results: Synthesis problems, unknown clock level timing, too many registers

Avoid Temptation

❑ Temptation # 1

- “Verilog looks like C, so I’ll write the algorithm in C and turn it into Verilog with a few always @ statements”
- Usual results: Synthesis problems, unknown clock level timing, too many registers

❑ Temptation # 2

- “I cant work out how to design it, so I’ll code up something that looks right and let Synthesis fix it”
- Usual result: Synthesis does not fix it

Avoid Temptation

❑ Temptation # 1

- “Verilog looks like C, so I’ll write the algorithm in C and turn it into Verilog with a few always @ statements”
- Usual results: Synthesis problems, unknown clock level timing, too many registers

❑ Temptation # 2

- “I cant work out how to design it, so I’ll code up something that looks right and let Synthesis fix it”
- Usual result: Synthesis does not fix it

❑ Temptation #3

- “Look at these neat coding structures available in Verilog, I’ll write more elegant code and get better results”
- Usual result: Synthesis problems for neophytes
- Better logic, not better code, gives a better design

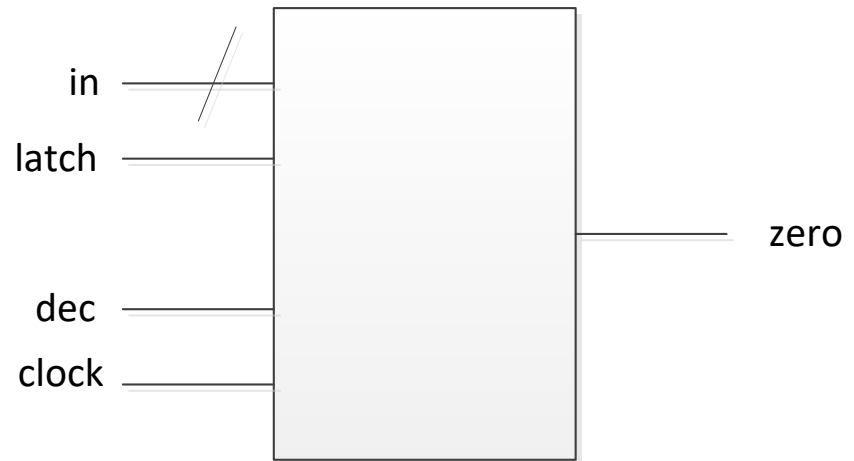
Design Before Coding – Steps in Design

1. Work out the hardware algorithm and overall strategy
2. Identify and name all registers (flip – flops)
 - Determine system timing while doing that
3. Identify the behavior of each combinational logic “cloud”
4. Translate design to RTL
5. Verify design
6. Synthesize design

Example: Count Down Timer

□ Specification

- 4 bit counter
- “count” value loaded from “in” on a positive clock edge when “latch” is high
- “count” value decremented by 1 on a positive clock edge when “dec” is high
- Decrement stops at 0
- “zero” flag active high whenever count value is 0



What not to do

- ❑ Coding before design

```
always @(posedge clock)
    for (value = in; value >= 0; value --)
        if (value == 0) zero = 1
        else zero = 0;
```

- ❑ or

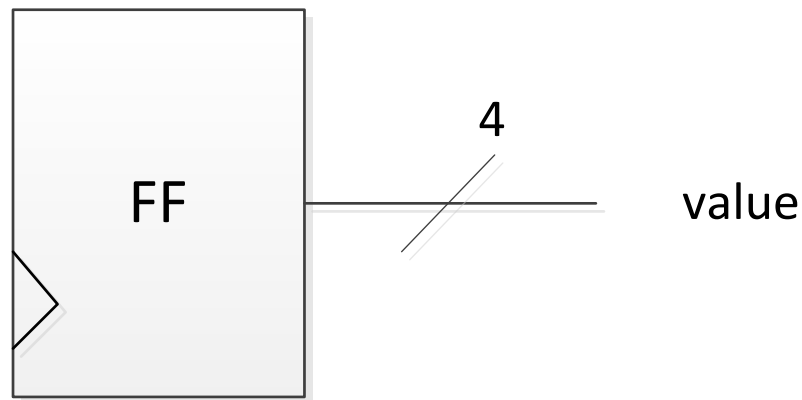
```
always @(posedge clock)
    for (value = in; value >= 0; value --)
        @ (posedge clock)
            if (value == 0) zero = 1
            else zero = 0;
```

Strategy

- ❑ 1. Work out the hardware algorithm and overall strategy
 - Strategy
 - Load “in” into a register
 - Decrement value of register while “dec” is high
 - Monitor register value to determine when zero

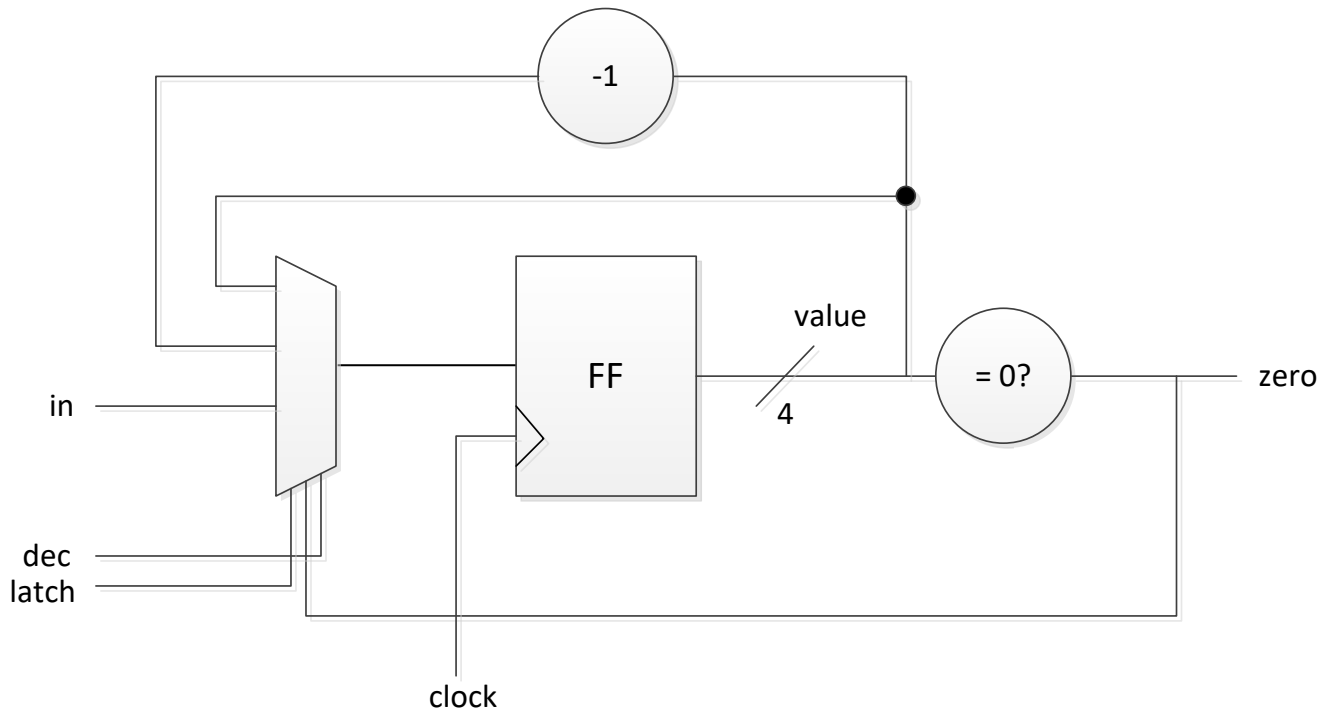
Design

- ❑ 2. Identify and name all the registers (flip – flops)



Design (cont'd)

- 3. Identify the behavior of each “cloud” of combinational logic



4. Translate Design into RTL

```
module counter (clock, in, latch, dec, zero);

    input clock;           //clock
    input [3:0] in;        //starting count
    input latch;           //latch 'in' when high
    input dec;             //decrement count when 'dec' high
    output zero;           //high when count down to zero

    reg [3:0] value        //current count value

    always @(posedge clock) begin
        if (latch) value <= in;
        else if (dec && !zero) value <= value - 1'b1;
    end
    assign zero = (value == 4'b0);

endmodule
```

Intrinsic parallelism

- How Verilog models the intrinsic parallelism of hardware

```
always @(posedge clk)
    A <= C & D);
```

```
always @(A)
    G = |A;
```

```
assign F = E ^ A;
```

Intrinsic parallelism

- How Verilog models the intrinsic parallelism of hardware

```
always @(posedge clk)  
    A <= C & D);
```



```
always @(A)  
    G = |A;
```

```
assign F = E ^ A;
```

Intrinsic parallelism

- How Verilog models the intrinsic parallelism of hardware

```
always @(posedge clk)  
    A <= C & D);
```



```
always @(A) ← triggers  
    G = |A;
```

```
assign F = E ^ A;
```

Intrinsic parallelism

❑ Algorithm for

```
always @(A)
    G = |A;
assign F = E ^ A;
```

❑ when A changes

- In same time step
 - $\text{nextG} = |A;$
 - $\text{nextF} = E \wedge A;$
- At the end of time step
 - $G = \text{nextG};$
 - $F = \text{nextF};$

5. Verify Design

- ❑ Achieved by designing a “test fixture” to exercise design
- ❑ Verilog in test fixture is not highly constrained
 - See more Verilog features in test fixture than in RTL

Verilog 2001 Test Fixture

```
module test_fixture;
    reg                clock100 = 0 ;
    reg                latch = 0;
    reg                dec = 0;
    reg                [3:0] in = 4'b0010;
    wire zero;
    initial //following block executed only once
    begin
        $dumpfile("count.vcd"); // waveforms in this file..
                                   // Note Comments from previous example
        $dumpvars; // saves all waveforms
        #16 latch = 1;                // wait 16 ns
        #10 latch = 0;                // wait 10 ns
        #10 dec = 1;
        #100 $finish;                //finished with simulation
    end

    always #5 clock100 = ~clock100; // 10ns clock

    // instantiate modules -- call this counter u1
    counter u1( .clock(clock100), .in(in), .latch(latch), .dec(dec), .zero(zero));
endmodule /*test_fixture*/
```


Verilog 2001 Test Fixture

```
module test_fixture;
  reg      clock100 = 0 ;
  reg      latch = 0;
  reg      dec = 0;
  reg      [3:0] in = 4'b0010;
  wire     zero;
  initial  //following block executed only once
  begin
```

```
    $dumpfile("count.vcd"); // waveforms in this file..
```

// Note Comments from previous example

```
    $dumpvars; // saves all waveforms
```

```
    #16 latch = 1;
```

// wait 16 ns

```
    #10 latch = 0;
```

// wait 10 ns

```
    #10 dec = 1;
```

```
    #100 $finish;
```

//finished with simulation

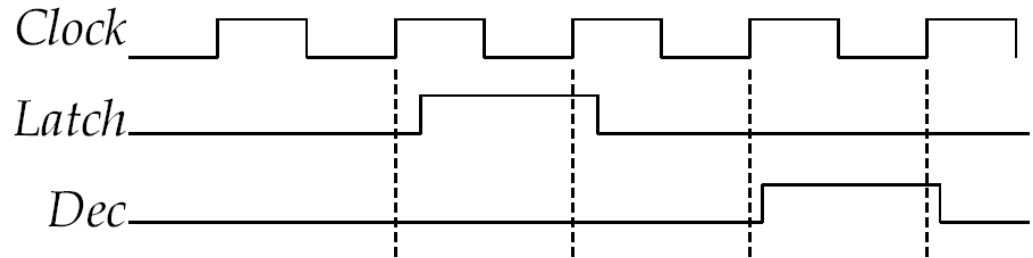
```
  end
```

```
  always #5 clock100 = ~clock100; // 10ns clock
```

```
  // instantiate modules -- call this counter u1
```

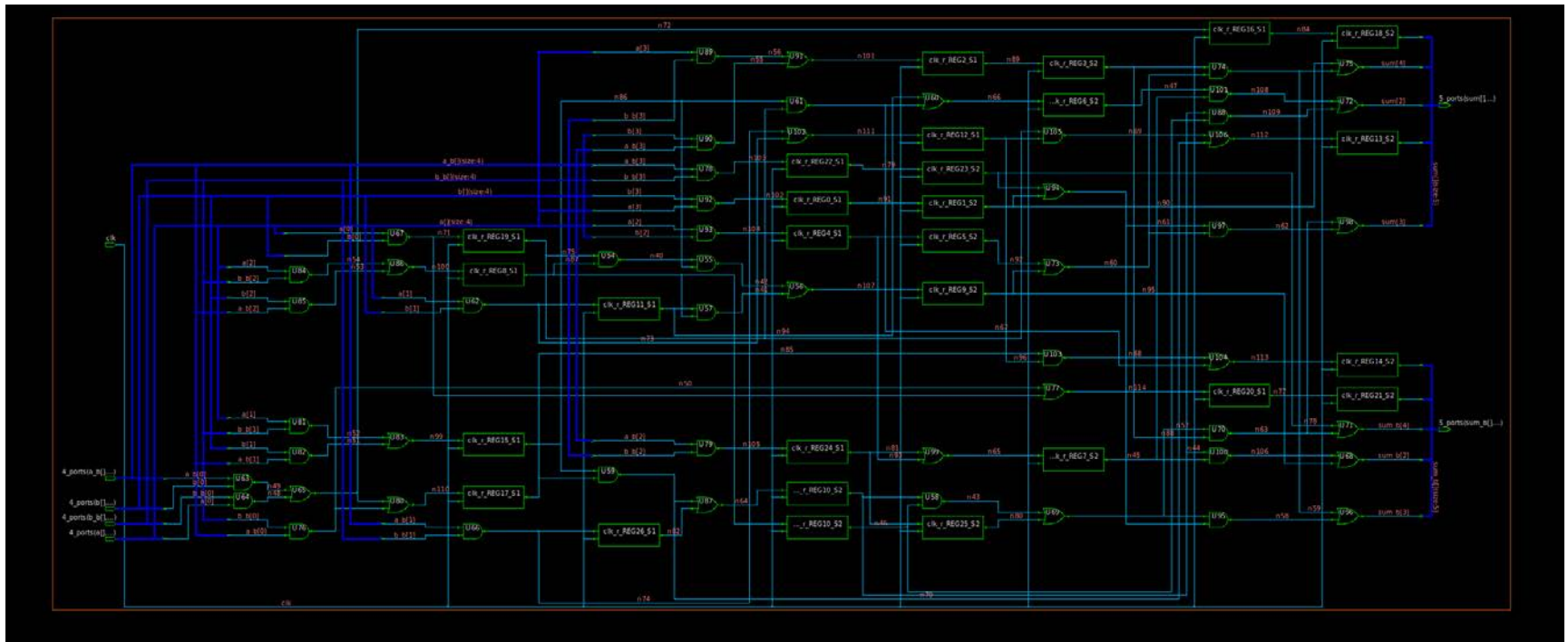
```
  counter u1( .clock(clock100), .in(in), .latch(latch), .dec(dec), .zero(zero));
```

```
endmodule /*test_fixture*/
```



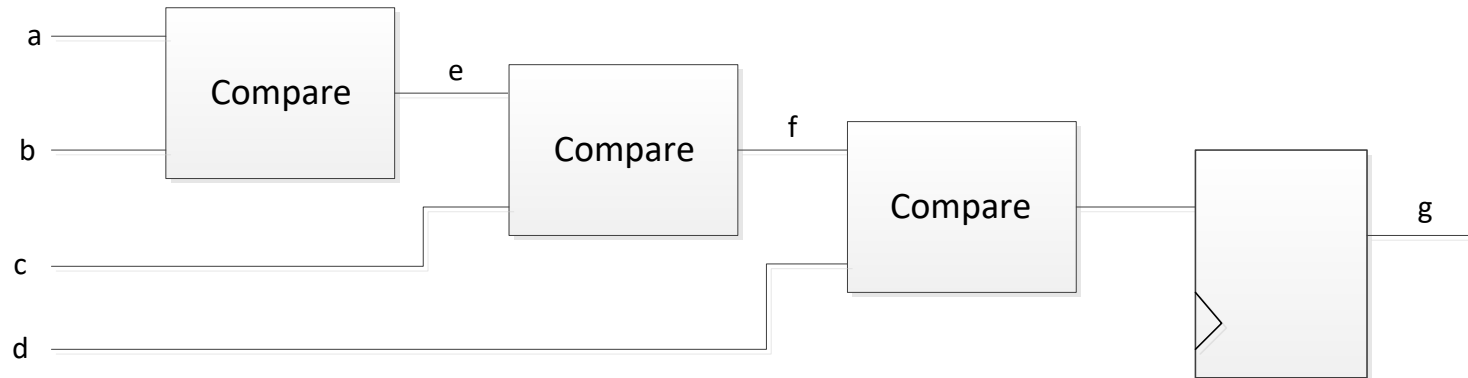
Synthesis

- ❑ After verifying correctness, the design can be synthesized to optimized logic
- ❑ The result is a gate level design (netlist)



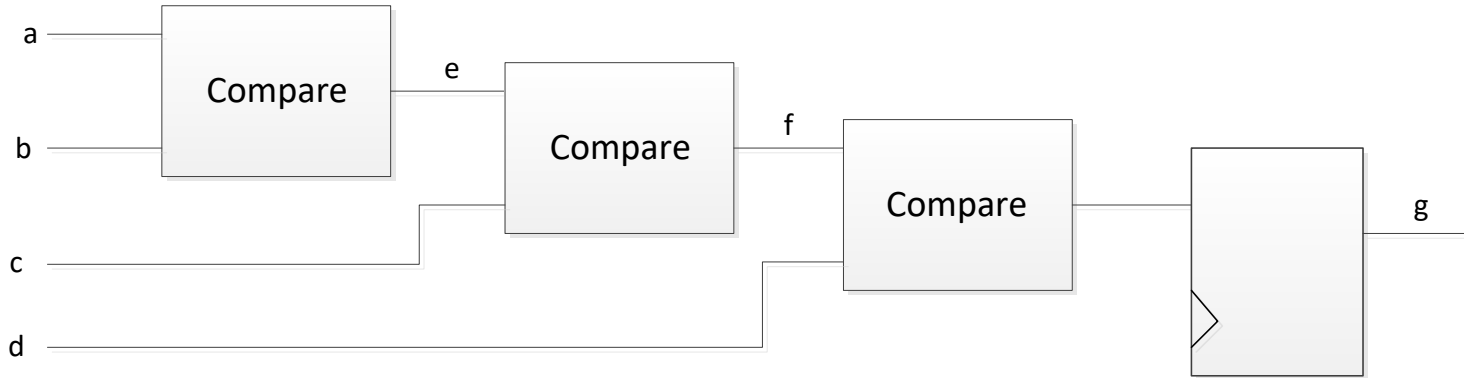
Exercise

- What do these look like in Verilog



Exercise

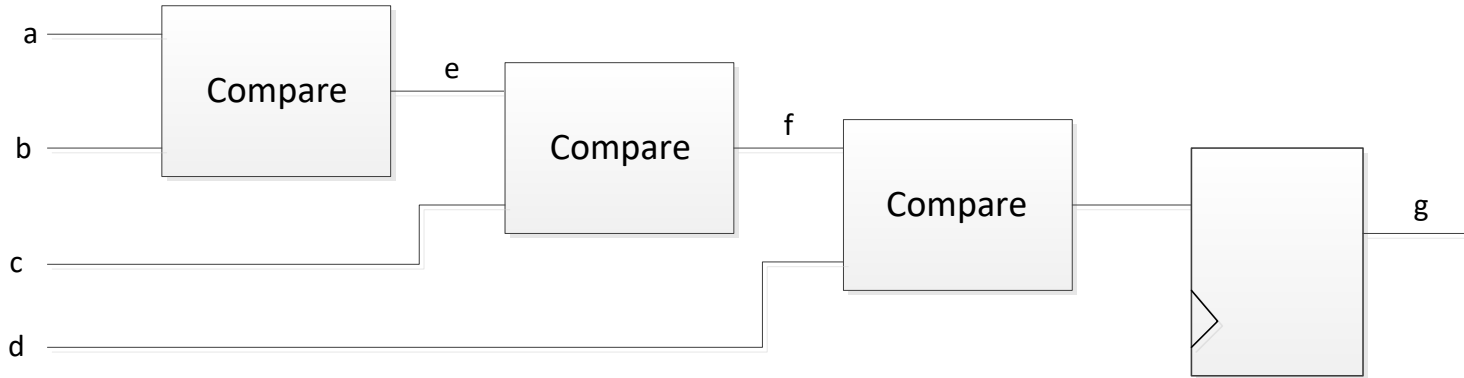
- What do these look like in Verilog



```
always @(a or b or c) begin
    if (a > b) e = a; else e = b;
    if (c > e) f = e; else f = c;
end
always @(posedge clock)
    if (d > f) g <= d; else g <= f;
```

Exercise

- What do these look like in Verilog

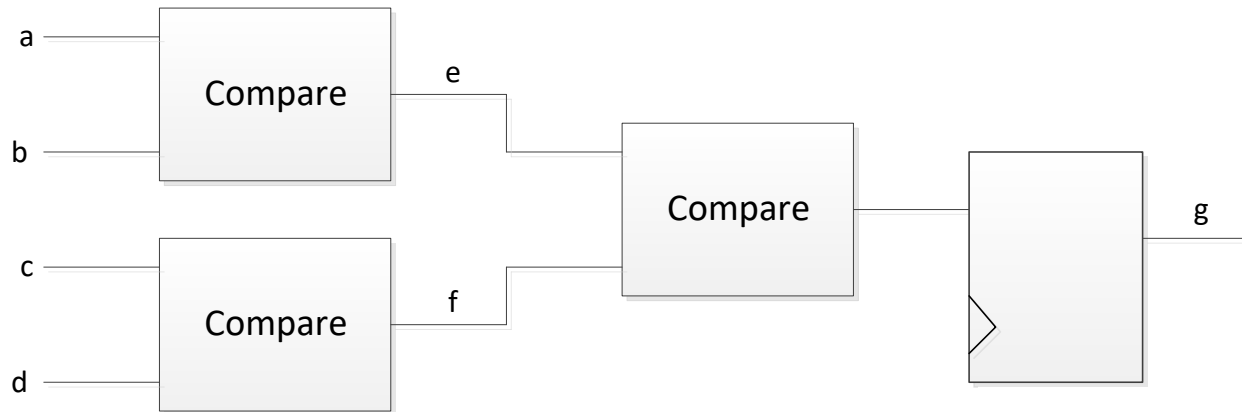


```
always @(a or b or c) begin
    if (a > b) e = a; else e = b;
    if (c > e) f = e; else f = c;
end
always @(posedge clock)
    if (d > f) g <= d; else g <= f;
```

Why 2 always?

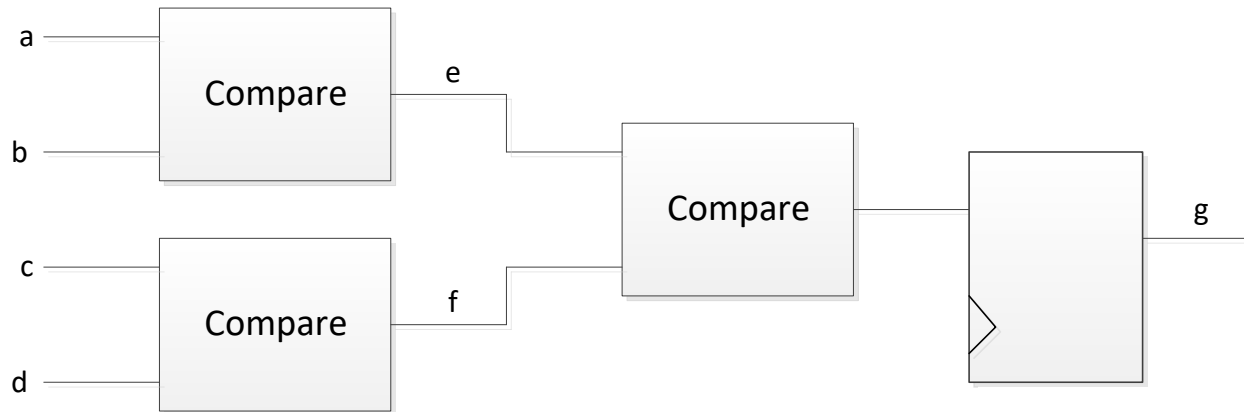
Exercise

- Produce a Verilog code fragment for the following schematic (use continuous assignment)



Exercise

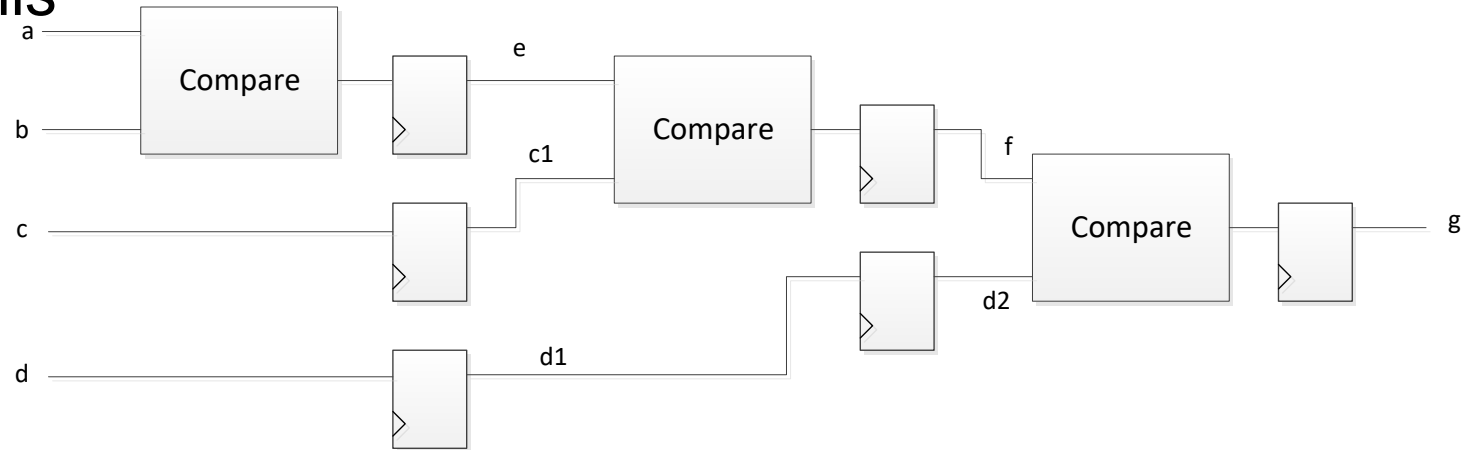
- Produce a Verilog code fragment for the following schematic (use continuous assignment)



```
assign e = (a>b)? a : b;  
assign f = (c>d)? c : d;  
always @(posedge clock)  
    if (e > f) g <= e; else g <= f;
```

Exercise

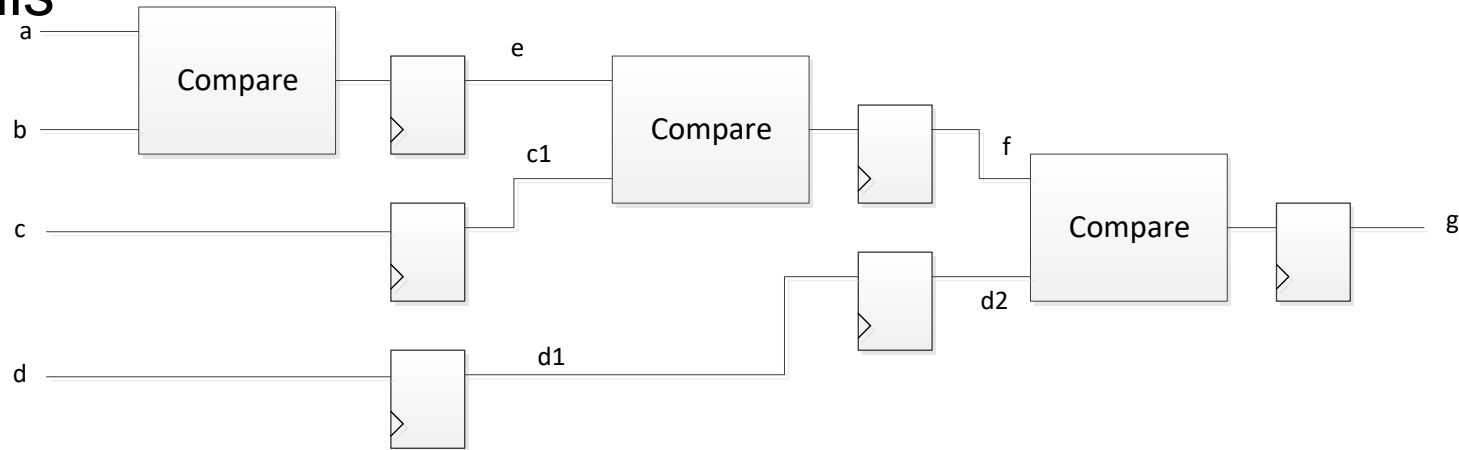
□ And for this



*Note: All FFs
have to be
named*

Exercise

□ And for this

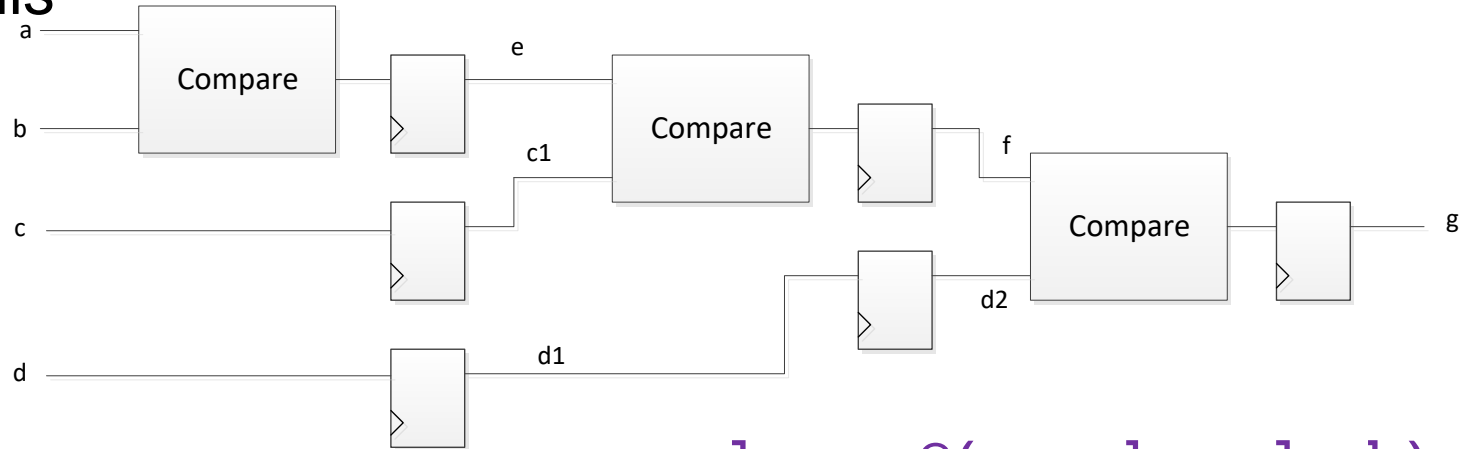


*Note: All FFs
have to be
named*

```
always @(posedge clock)
begin
    if (a > b) e <= a;
    else e <= b;
    c1 <= c;
    d1 <= d;
end
```

Exercise

□ And for this



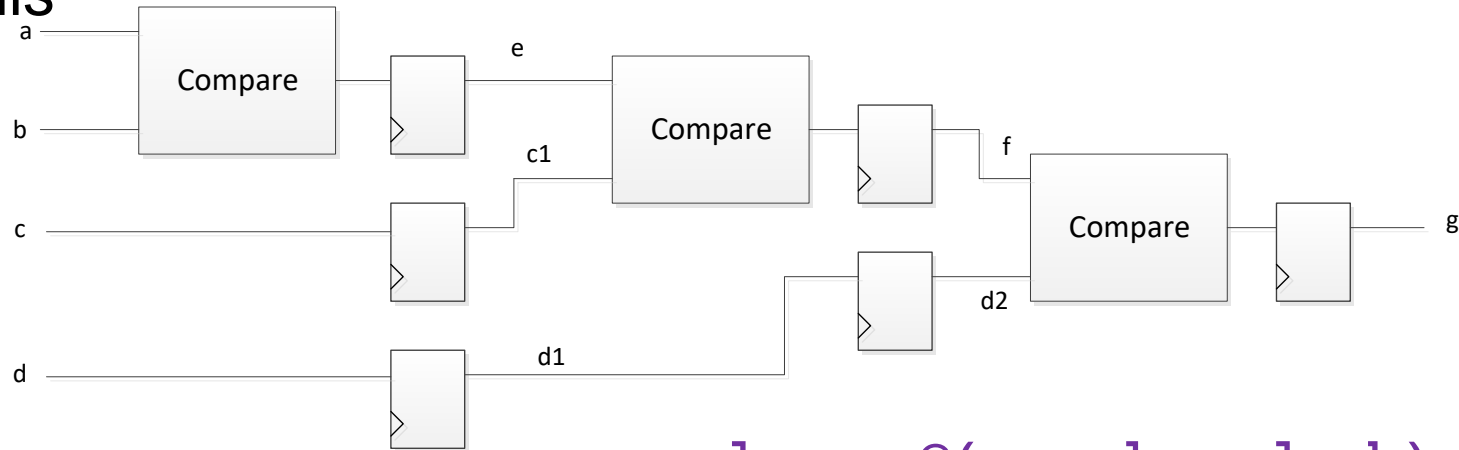
*Note: All FFs
have to be
named*

```
always @(posedge clock)
begin
    if (a > b) e <= a;
    else e <= b;
    c1 <= c;
    d1 <= d;
end
```

```
always @(posedge clock)
begin
    if (e > c1) f <= e;
    else f <= c1;
    d2 <= d1;
end
```

Exercise

□ And for this



*Note: All FFs
have to be
named*

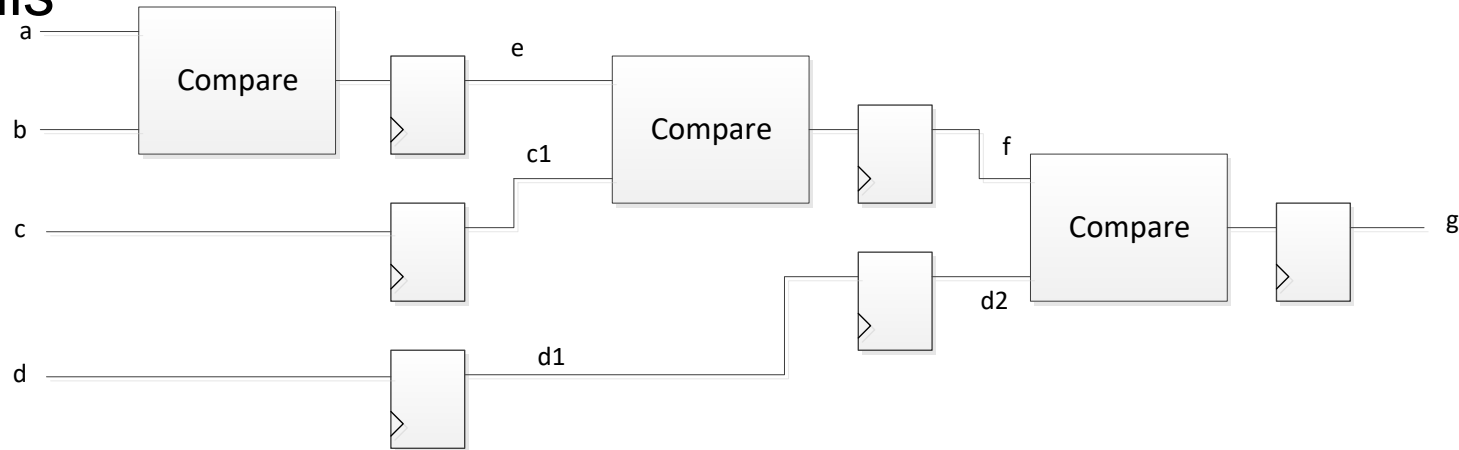
```
always @(posedge clock)
begin
    if (a > b) e <= a;
    else e <= b;
    c1 <= c;
    d1 <= d;
end
```

```
always @(posedge clock)
begin
    if (e > c1) f <= e;
    else f <= c1;
    d2 <= d1;
end
```

```
always @(posedge clock)
    if (f > d2) g <= f; else g <= d2;
```

Exercise

□ And for this



*Note: All FFs
have to be
named*

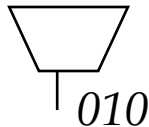
```
always @(posedge clock) begin
    if (a > b) e <= a; else e <= b;
    if (e > c1) f <= e; else f <= c1;
    if (f > d2) g <= f; else g <= d2;
    c1 <= c;
    d1 <= d;
    d2 <= d1;
end
```

Minimizing Power Consumption

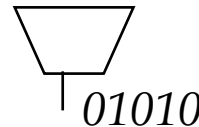
- Will go over in a later set of notes, but here is the logic design impact...
 - In general, at the logic level, the energy required to complete a complex task is roughly proportional to:

- $\sum_{\text{nodes}} 01 \text{ and } 10 \text{ logic transitions}$

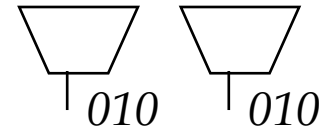
- E.g.



“1 unit of energy”



“2 units of energy”



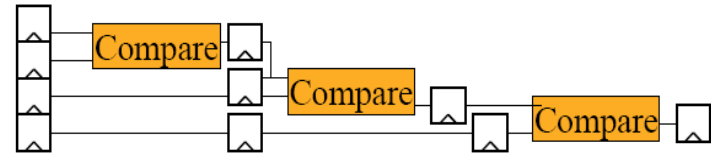
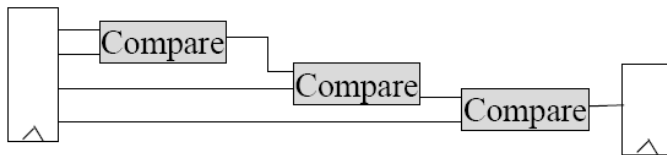
“2 units of energy”

- Note:
 - Complex logical units (e.g. Multiplier) have a lot more internal nodes than simpler logical units
 - And thus consume more energy per operation

Minimizing Power Consumption

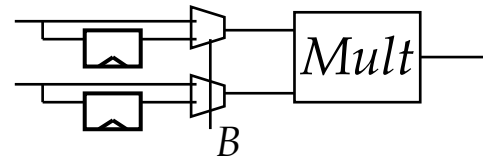
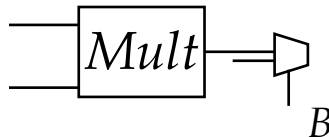
- Simpler, smaller design will often be more energy efficient
- There is often a speed-power tradeoff

E.g. Which design is more energy efficient?



- Try to eliminate useless toggling

E.g. Which design is LESS energy efficient if B mostly DESELECTS mult output?



Minimizing Power Consumption

- Memory accesses are particularly energy hungry, especially with larger memories
- Complex data motion is particularly power hungry
 - E.g. Long range on-chip interconnect
 - Off-chip interconnect
 - Using an on-chip network to move data, especially a store and forward network

Let's be formal now

Lexical Conventions in Verilog

❑ Logic Values

Logic Value	Description
0	Zero, low, false
1	One, high, true
Z or z or ?	High impedance, tri-state, or floating
X or x	Unknown, uninitialized, or don't care

Size'base value;
size = # of bits

❑ bases:

h = hexadecimal, d = decimal (default), o=octal, b=binary

❑ Examples:

10, 3'b1, 8'hF0, 8'hF, 5'd11, 2'b10, {n{1'b0}}, 32'hffff_efff

Procedural Blocks

- ❑ Code of the type

```
always @(input1 or input2 or ...) begin
    if or case statements
end
```

- ❑ is referred as procedural code
 - Statements between begin and end are executed procedurally, or in order
 - Variables assigned (on the left hand side) in procedural code must be of a register data type. Type reg is used
 - It does not mean it is a register or a FF
 - The procedural block is executed when triggered by the always@ statement
 - The statement in parenthesis are referred to as the sensitive list

Blocking vs. Non-Blocking

- ❑ Why use `<=` to specify flip-flops?
- ❑ Blocking

```
begin
    A = B;
    C = D;
end
```

- Assignment of C blocked until A = B completed → They executed in sequence

- ❑ Non Blocking

```
begin
    A <= B;
    C <= D;
end
```

- Assignment of C not blocked until A = B completed → They executed in parallel

Blocking vs. Non-Blocking

Blocking

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

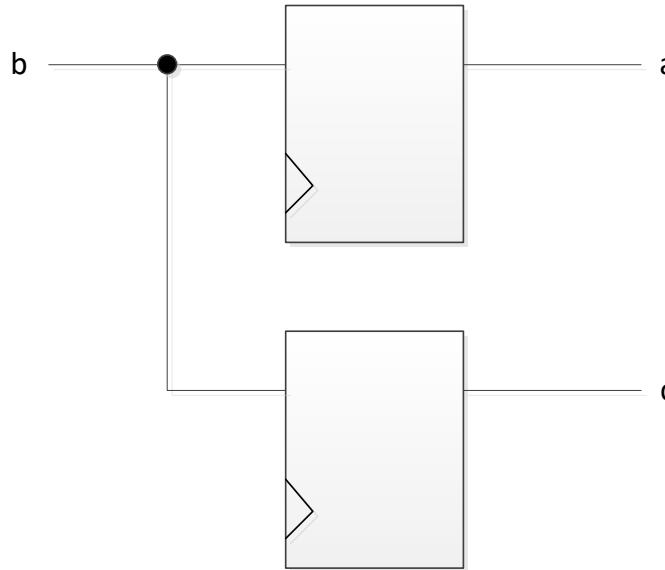
Non-Blocking

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

Blocking vs. Non-Blocking

Blocking

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



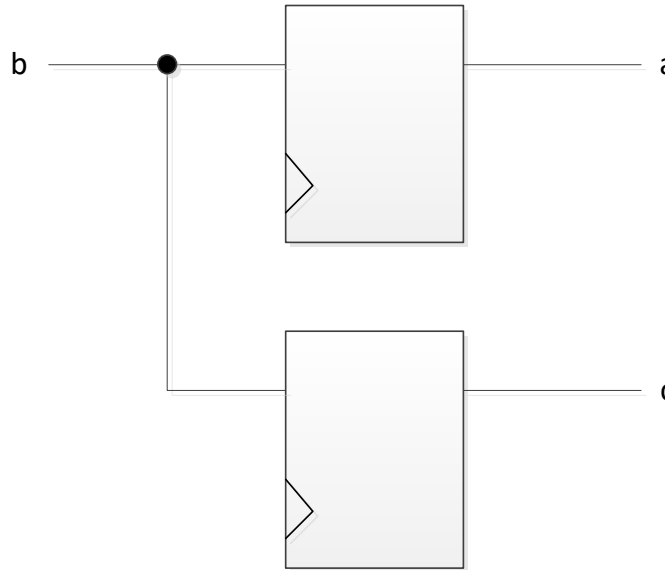
Non-Blocking

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

Blocking vs. Non-Blocking

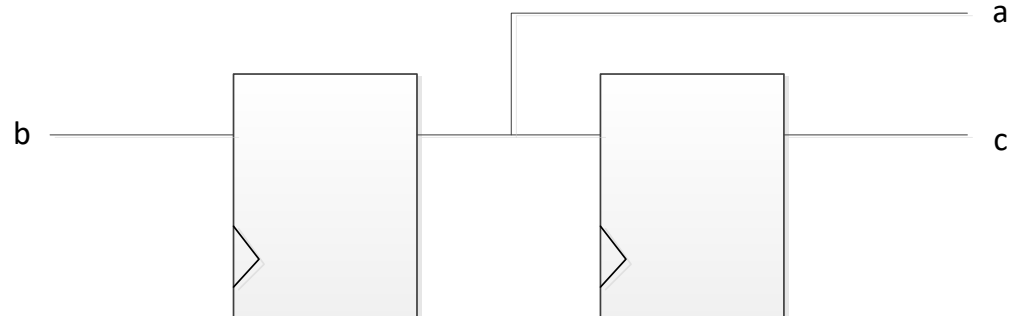
Blocking

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



Non-Blocking

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



Blocking vs. Non-Blocking

- ❑ Which describes better what you expect to see?
 - Non blocking assignment
- ❑ Note
 - Use non blocking for flip flops
 - Use blocking for combinational logic
 - Logic can be evaluated in sequence
 - Not synchronized to clock
 - Don't mix them in the same procedural block

Blocking Statements

- ❑ Hand execute the following

```
//test fixture
initial begin
    a = 4'h3; b = 4'h4;
end
//code
always @(posedge clock)
    c = a + b;
    d = c + a;
end
```

- ❑ Results?

Blocking Statements

- ❑ Hand execute the following

```
//test fixture
initial begin
    a = 4'h3; b = 4'h4;
end
//code
always @(posedge clock)
    c = a + b;      c=4'h7
    d = c + a;
end
```

- ❑ Results?

Blocking Statements

- ❑ Hand execute the following

```
//test fixture
initial begin
    a = 4'h3; b = 4'h4;
end
//code
always @(posedge clock)
    c = a + b;          c=4'h7
    d = c + a;          d=4'hA
end
```

- ❑ Results?

Non Blocking Statements

- ❑ Contrast it with this code

```
//test fixture
initial begin
    a = 4'h3; b = 4'h4; c = 4'h2;
end
//code
always @(posedge clock)
    c <= a + b;
    d <= c + a;
end
```

- ❑ Results?

Non Blocking Statements

- ❑ Contrast it with this code

```
//test fixture
initial begin
    a = 4'h3; b = 4'h4; c = 4'h2;
end
//code
always @(posedge clock)
    c <= a + b;      c=4'h7
    d <= c + a;      d=4'h5
end
```

- ❑ Results?

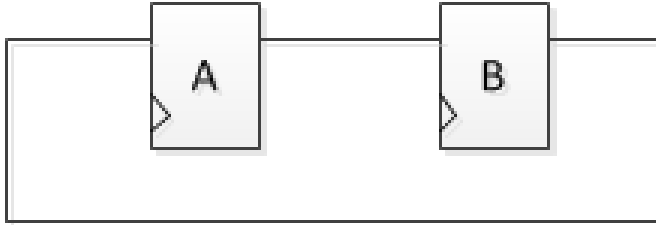
Non Blocking Statements

- ❑ Contrast it with this code

```
//test fixture
initial begin
    a = 4'h3; b = 4'h4; c = 4'h2;
end
//code
always @(posedge clock)
    c <= a + b;      c=4'h7      c=4'h7
    d <= c + a;      d=4'h5      d=4'hA
end
```

- ❑ Results?

Exercise



- A.

```
always @(posedge clock) begin
    A = B; B = A;
end
```
- B.

```
always @(posedge clock) begin
    B = A; A = B;
end
```
- C.

```
always @(posedge clock) begin
    C = B; D = A; B = D; A = C;
end
```
- D.

```
always @(posedge clock) begin
    A = B = A;
end
```

Behavior → Function

- What do the following code fragments synthesize to?

```
reg out;  
always @(a or b or c) begin  
    if (a)  
        out = b | c;  
    else  
        out = b ^ c;  
end
```

```
reg out;  
always @(clock or a) begin  
    if (clock)  
        out = a;  
end
```

Behavior → Function

- What do the following code fragments synthesize to?

```
reg out;  
always @(a or b or c) begin  
    if (a)  
        out = b | c;  
    else  
        out = b ^ c;  
end
```

mux

```
reg out;  
always @(clock or a) begin  
    if (clock)  
        out = a;  
end
```


Behavior → Function

- What do the following code fragments synthesize to?

```
reg out;  
always @(a or b or c) begin  
    if (a)  
        out = b | c;  
    else  
        out = b ^ c;  
end
```

mux

```
reg out;  
always @(clock or a) begin  
    if (clock)  
        out = a;  
end
```

latch

Behavior → Function

- What is the logic being described?

```
input [1:0] sel;  
input [3:0] a;  
reg y;  
  
always @(sel or a)  
    casex (sel)  
        0: y = a[0];  
        1: y = a[1];  
        2: y = a[2];  
        3: y = a[3];  
        default: y = 1'bx;  
    endcase
```

Behavior → Function

- What is the logic being described?

```
input [1:0] sel;  
input [3:0] a;  
reg y;
```

mux

```
always @(sel or a)  
  casex (sel)  
    0: y = a[0];  
    1: y = a[1];  
    2: y = a[2];  
    3: y = a[3];  
    default: y = 1'bx;  
  endcase
```

Behavior → Function

- What is the logic being described?

```
input [3:0] a;  
reg [3:0] y;  
always @(a)  
    casex (a)  
        8'b0001: y = 0;  
        8'b0010: y = 1;  
        8'b0100: y = 2;  
        8'b1000: y = 3;  
        default: y = 2'bx;  
    endcase
```

Behavior → Function

- ❑ What is the logic being described?

```
input [3:0] a;  
reg [3:0] y;  
always @(a)  
    casex (a)  
        8'b0001: y = 0;  
        8'b0010: y = 1;  
        8'b0100: y = 2;  
        8'b1000: y = 3;  
        default: y = 2'bx;  
    endcase
```

encoder

Continuous Assignment

- Sketch the logic being specified

```
input [3:0] a, b;  
wire [3:0] c, e;  
wire d, f, g;  
  
assign c = a ^ b;  
assign d = |a;  
assign e = {{2a[3]}, a[2:1]}  
assign f = a[0] ? b[0] : b[1];  
assign g = ( a == b);
```

Continuous Assignment

- Sketch the logic being specified

```
input [3:0] a, b, c;
```

```
wire [3:0] f, g;  
wire h;
```

```
assign f = a + b + c + d;  
assign g = (a + b) + ( c +d);  
assign h = c[a[1:0]];
```

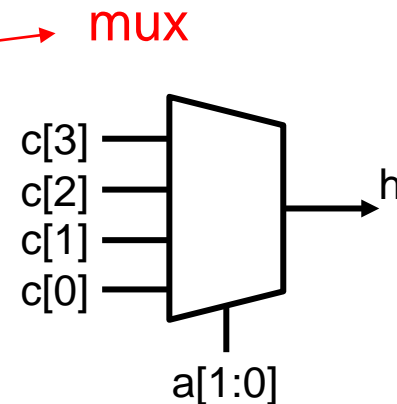
Continuous Assignment

- Sketch the logic being specified

```
input [3:0] a, b, c;
```

```
wire [3:0] f, g;  
wire h;
```

```
assign f = a + b + c + d;  
assign g = (a + b) + ( c +d);  
assign h = c[a[1:0]];
```



Continuous Assignment

- Sketch the logic being specified

```
input [7:0] a, b, c, d;  
input e;
```

```
wire [8:0] f, g, h;
```

```
assign f = a + b;  
assign g = c + d;  
assign h =(e)? f: g;
```

```
input [7:0] a, b, c, d;  
input e;
```

```
wire [7:0] f, g;  
wire [8:0] h;
```

```
assign f =(e)? a : c;  
assign g =(e)? b : d;  
assign h =f+g;
```

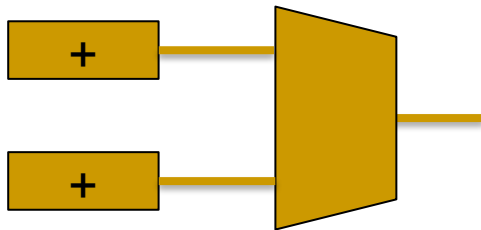
Continuous Assignment

- Sketch the logic being specified

```
input [7:0] a, b, c, d;  
input e;
```

```
wire [8:0] f, g, h;
```

```
assign f = a + b;  
assign g = c + d;  
assign h = (e)? f : g;
```



```
input [7:0] a, b, c, d;  
input e;
```

```
wire [7:0] f, g;  
wire [8:0] h;
```

```
assign f = (e)? a : c;  
assign g = (e)? b : d;  
assign h = f + g;
```

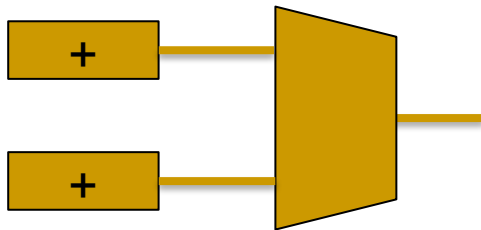
Continuous Assignment

- Sketch the logic being specified

```
input [7:0] a, b, c, d;  
input e;
```

```
wire [8:0] f, g, h;
```

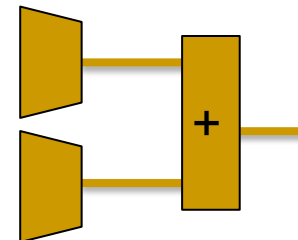
```
assign f = a + b;  
assign g = c + d;  
assign h = (e)? f : g;
```



```
input [7:0] a, b, c, d;  
input e;
```

```
wire [7:0] f, g;  
wire [8:0] h;
```

```
assign f = (e)? a : c;  
assign g = (e)? b : d;  
assign h = f + g;
```



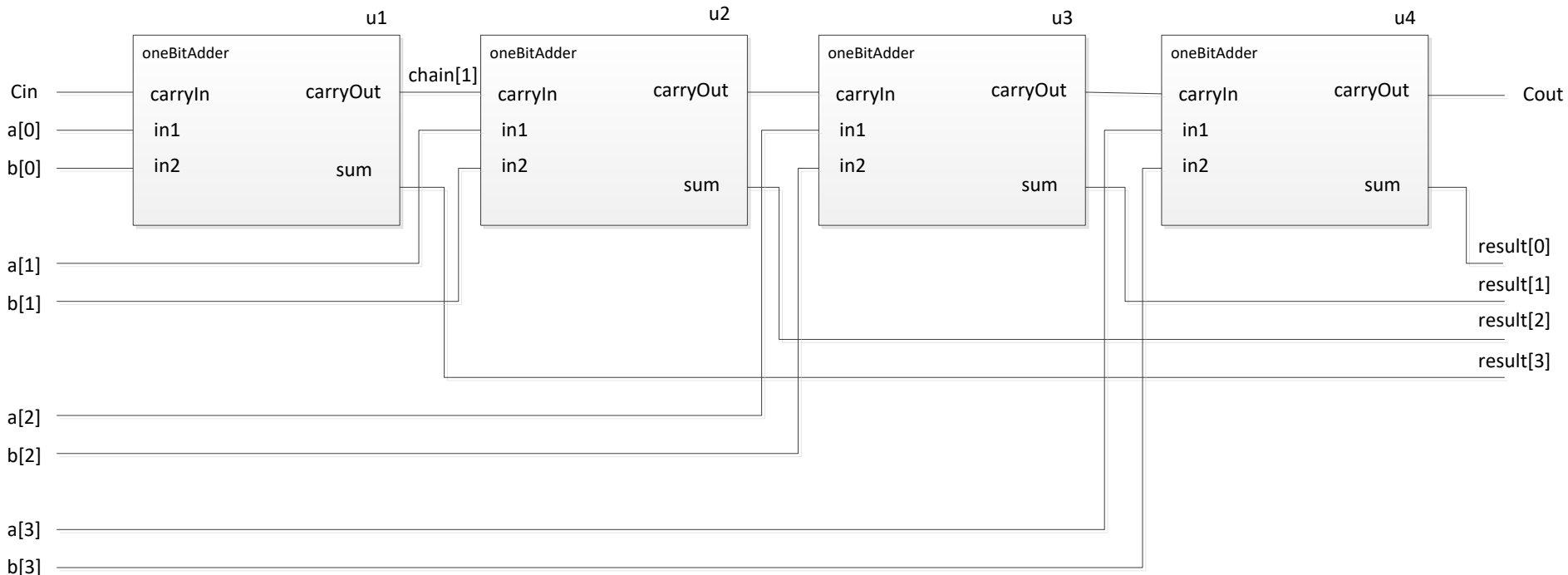
Structural Verilog

- ❑ Complex modules can be put together by building (instantiating) a number of small modules

```
module oneBitAdder (carryIn, in1, in2, sum, carryOut);  
module fourBitAdder (Cin, a, b, result, Cout);  
    input Cin;  
    input [3:0] a, b;  
    output [3:0] result;  
    output Cout;  
    wire [3:1] chain;  
    oneBitAdder u1 (.carryIn(Cin), .in1(a[0], .in2(b[0],  
.sum(result[0]), .carryOut(chain[1]));  
    oneBitAdder u2 (.carryIn(chain[1]), .in1(a[1],  
.in2(b[1], .sum(result[1]), .carryOut(chain[2]));  
    oneBitAdder u3 (.carryIn(chain[2]), .in1(a[2],  
.in2(b[2], .sum(result[2]), .carryOut(chain[3]));  
    oneBitAdder u4 (Chain[3], a[3], b[3], result[3], Cout);  
endmodule
```

Structural Example

❑ Schematic



Structural Verilog

❑ Features

- Four copies of the same module (oneBitAdder) are built (instantiated) each with a unique name (u1, u2, u3, and u4)
- Module instance syntax

```
oneBitAdder u1 (.carryIn(Cin), ...
```

- All nets connecting to outputs of modules must be of wire type

Applications of Structural Verilog

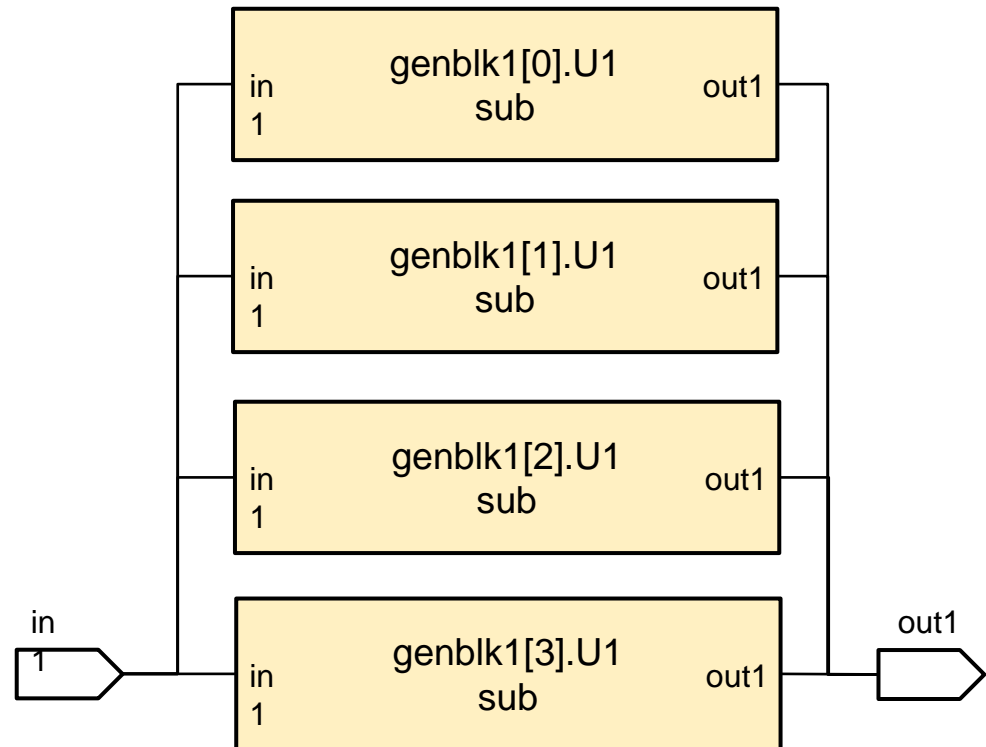
- ❑ To assemble modules together in a hierarchical design
- ❑ Final gate set written out in this format (netlist)
- ❑ Design has to be implemented as a module in order to integrate with the test fixture

Generate blocks

- ❑ Allow to generate Verilog code dynamically at elaboration time
 - Facilitated the parameterized model generation
 - Required the keywords `generate` – `endgenerate`
 - Generate instantiations can be
 - Module instantiations
 - Continuous assignments
 - `initial` / `always` blocks
 - Typically used are generate loop and conditional generate

Generate loop

```
module top( input  [0:3] in1,
            output [0:3] out1);
// genvar control the loop
genvar I;
generate
for(I=0; I<=3; I=I+1 )
begin
    sub U1 (in1[I], out1[I]);
end
endgenerate
endmodule
```



Conditional generate

```
module top #(parameter POS=0)
    (input in, clk, output reg out);

generate
if(POS==1)
always @ (posedge clk)
    out = in;
else
always @ (negedge clk)
    out = in;
endgenerate
endmodule
```

Common Problems and Fixes

❑ Unintentional latches

- Detected after reading the design
- How to fix: make sure every variable is assigned for every way code is executed (except for flip flops)
- If unfixed: you can have glitches on “irregular clock” to latch cause set up and hold problems in actual hardware (transient failures)

Problem code

```
always @(a or b) begin
    if (a) c = ~b;
    else d = |b;
end
```

Common Problems and Fixes

- ❑ Incomplete sensitivity list
 - Detected after reading the design
 - How to fix: all logic inputs have to appear in sensitivity list or use always @(*) (Verilog 2001)
 - If unfixed: since simulation results won't match what actual hardware will do, bugs can remain undetected

Problem code

```
always @(a or b) begin
    if (a) c = b ^ a;
    else c = d & e;
    f = c | a;
end
```

Common Problems and Fixes

- ❑ Unintentional wired-or logic
 - Detected after reading the design, variables cannot be assigned in more than one block
 - How to fix: redesign hardware so that every signal is driven by only one piece of logic (or redesign s a tri-state buf if that is the intention)
 - If unfixed: unsynthesizable, this is a symptom of not designing before coding

Problem code

```
always @(a or b)
    c = |b;
```

```
always @(d or e)
    c = ^e;
```

Common Problems and Fixes

- ❑ Improper startup
 - Cannot be detected
 - How to fix: make sure don't cares are propagated
 - If unfixed: possible undetected bug in reset logic

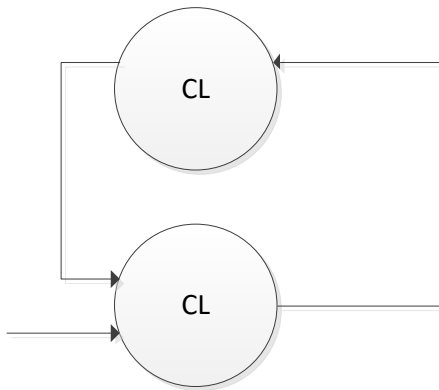
Problem code

```
always @(posedge clock)
    if (a) q <= d;
```

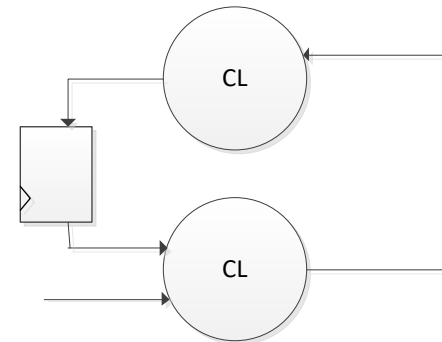
```
always @(q or e)
    case (q)
        0: f = e;
        default: f = 1;
    endcase
```

Common Problems and Fixes

- ❑ Feedback in combinational logic
 - Either results in
 - Latches, when the feedback path is short
 - Timing Arcs when feedback path is convoluted
 - Fix by redesigning logic to remove feedback
 - Feedback can only be through FFs



WRONG



OK

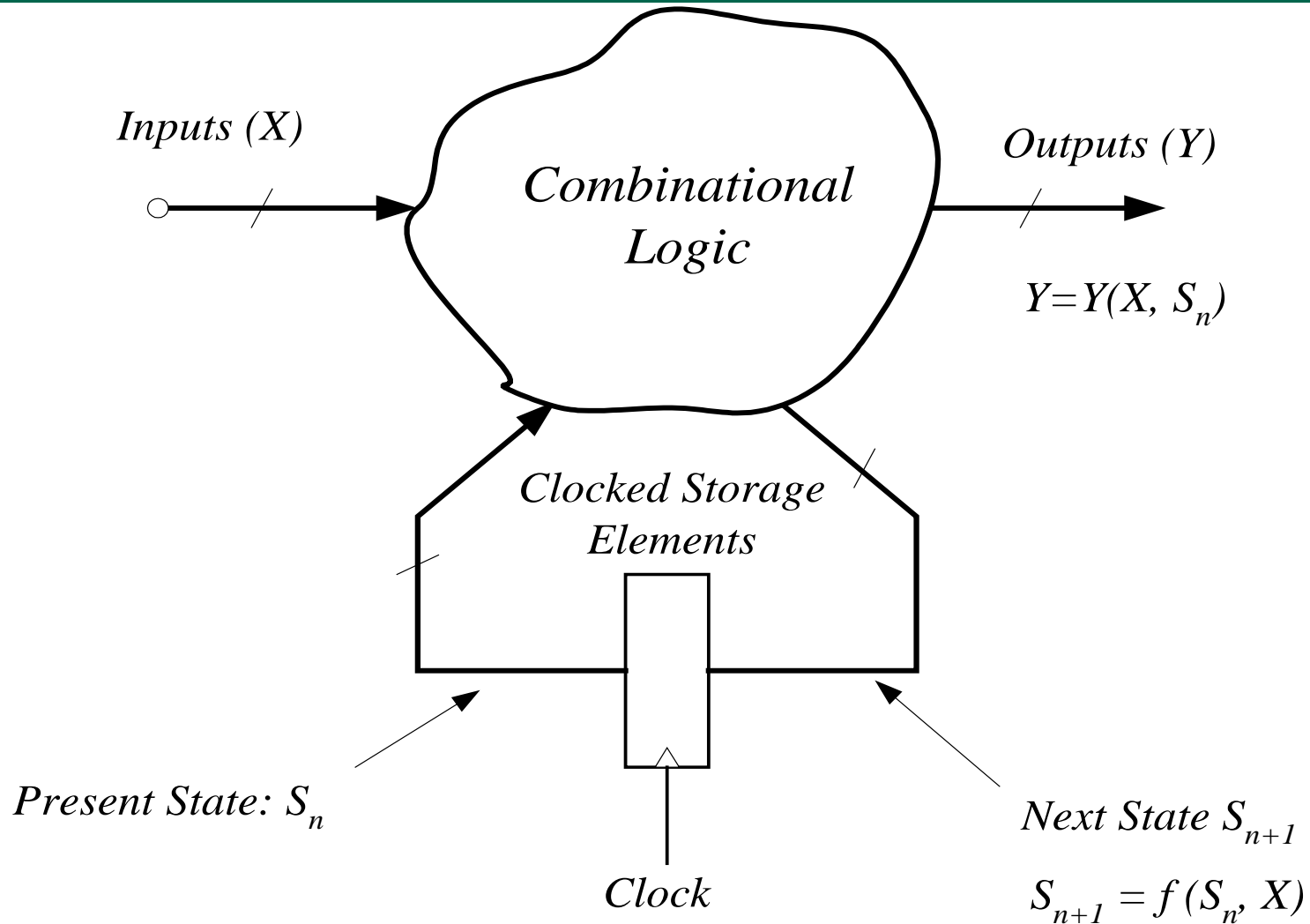
Common Problems and Fixes

- ❑ Incorrect use of FOR loops
 - Only correct use is to iterate through an array of bits
 - If in doubt, do not use it
- ❑ Unconstrained timing
 - To calculate permitted delay, synthesis must know where the FFs are
 - If you have a path from input port to output port that does not pass through a FF, synthesis cannot calculate timing
 - To fix: revisit modules partitioning to include FFs in all paths

Debug

- How to prevent a lot of need to debug:
 - Carefully think through design before coding
 - Simulate “in your head”
- How to debug:
 - Track bug point back in design and back in time
 - Check if each “feeding” signal makes sense
 - Compare against a “simulation in your head”
 - If all else fails, recode using a different technique

Finite-State Machine Abstraction



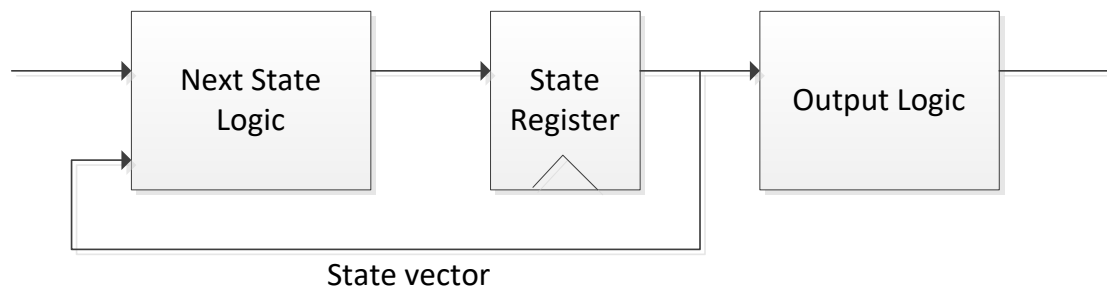
Finite-State Machine Abstraction

- ❑ Clocked Storage Elements: Flip-Flops and Latches should be viewed as **synchronization elements**, not merely as **storage elements** !
- ❑ Their main purpose is to synchronize fast and slow paths:
 - Prevent the fast path from corrupting the state
- ❑ Function of clock signals is to provide a reference point in time when the FSM changes states

Finite-State Machine – Types

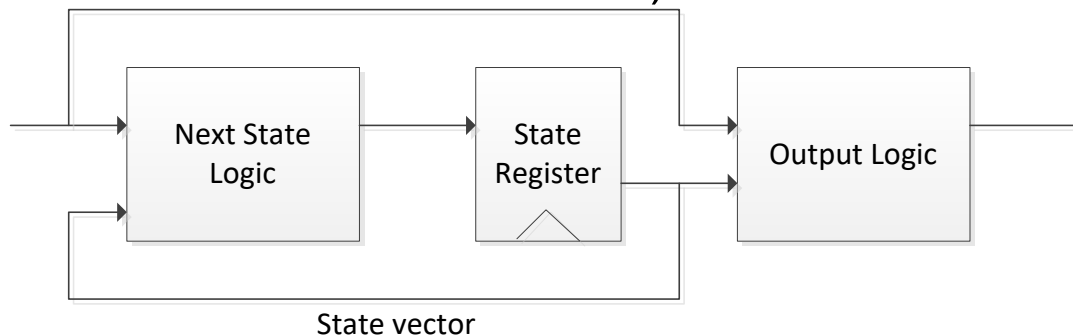
❑ Moore machine

- Outputs depend solely on state vector (simplest to design)



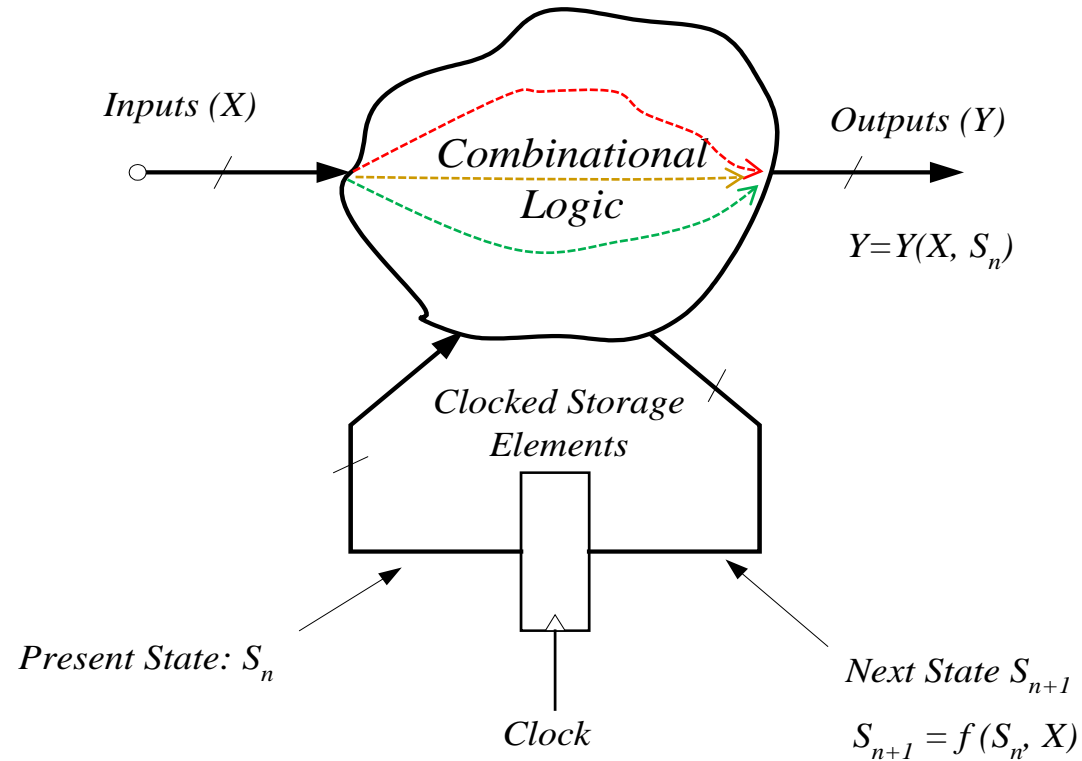
❑ Mealy machine

- Outputs depend on inputs and state vector (only use it if smaller or faster machines are needed)



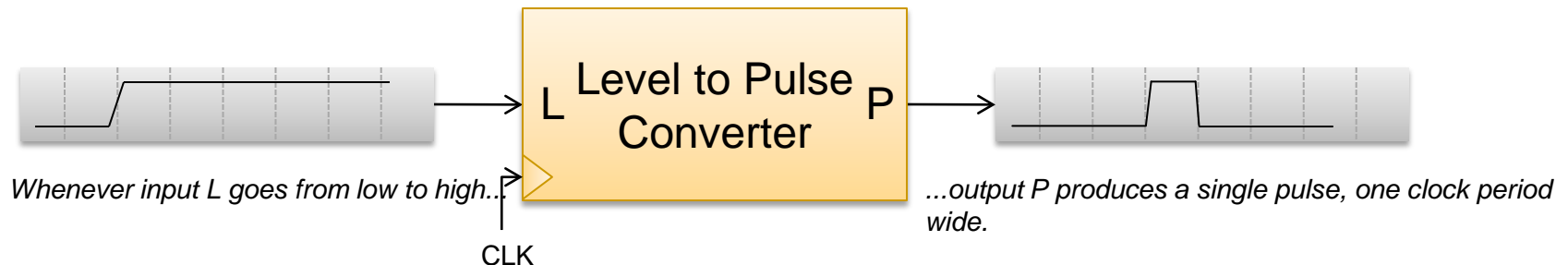
Finite-State machine – Critical Path

Critical path is defined as the chain of gates in the longest (slowest) path through the logic



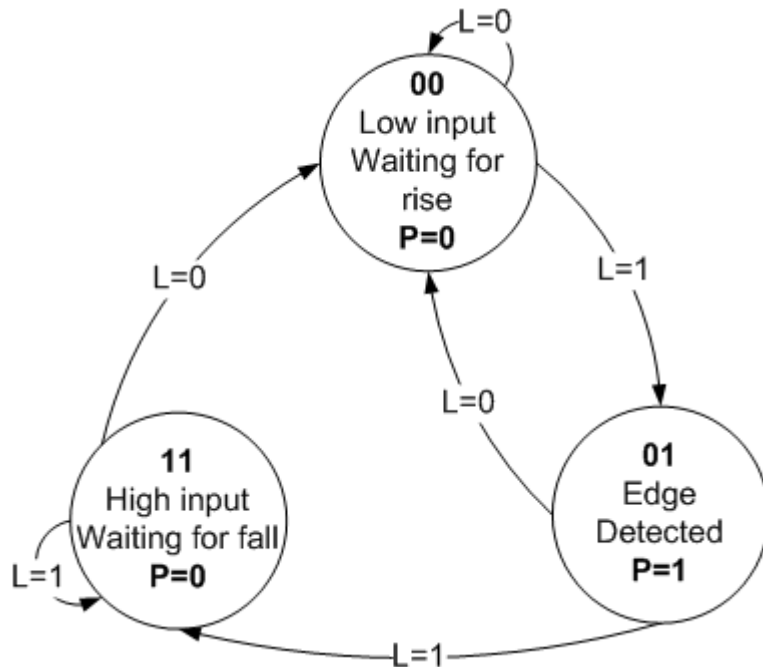
FSM Implementation – Design example

- ❑ A level-to-pulse converter produces a single-cycle pulse each time its input goes high.
- ❑ In other words, it's a synchronous rising-edge detector.
- ❑ Sample uses:
 - Buttons and switches pressed by humans for arbitrary periods of time
 - Single-cycle enable signals for counters



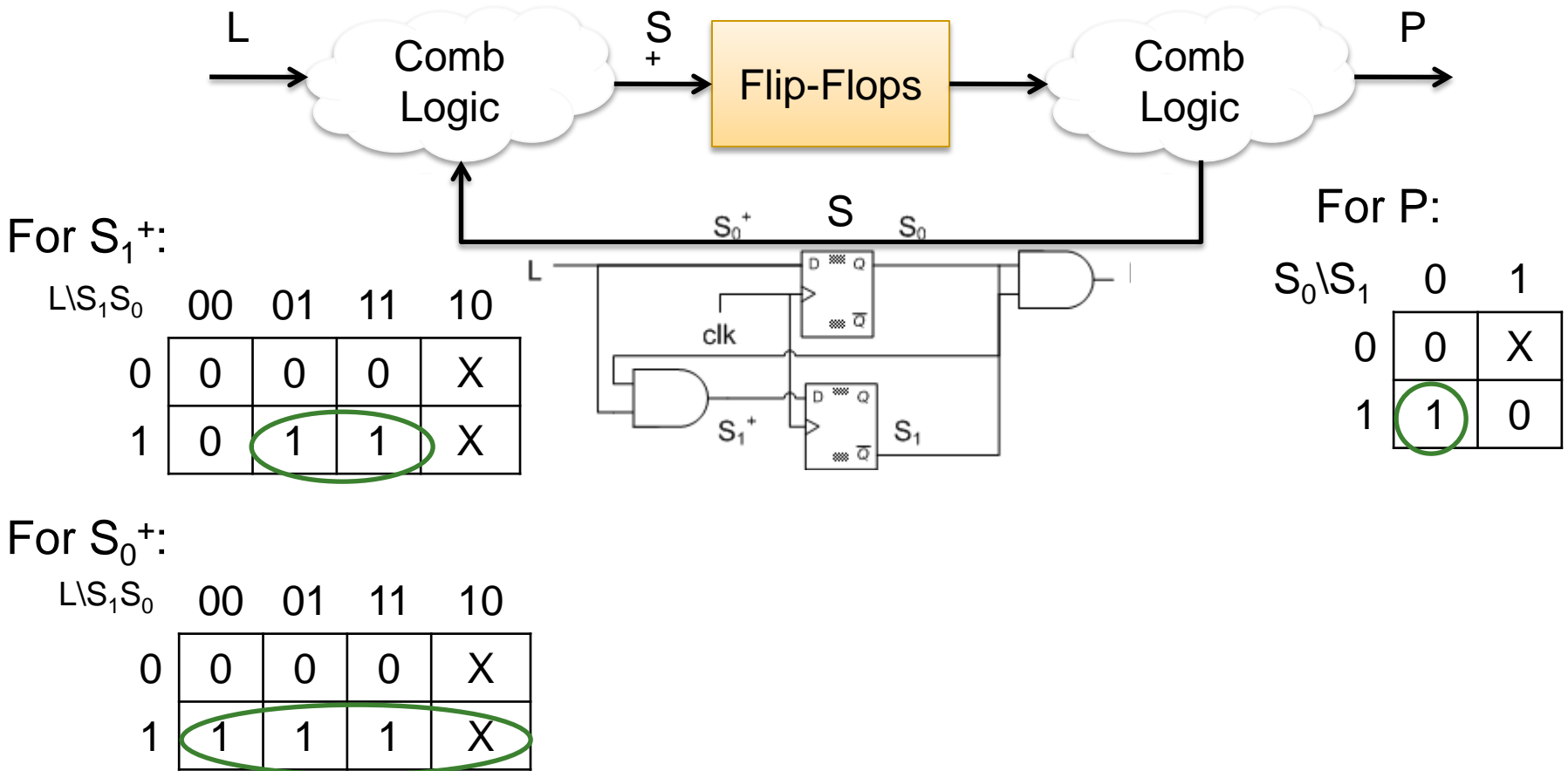
FSM Implementation – State Diagram

❑ Moore implementation



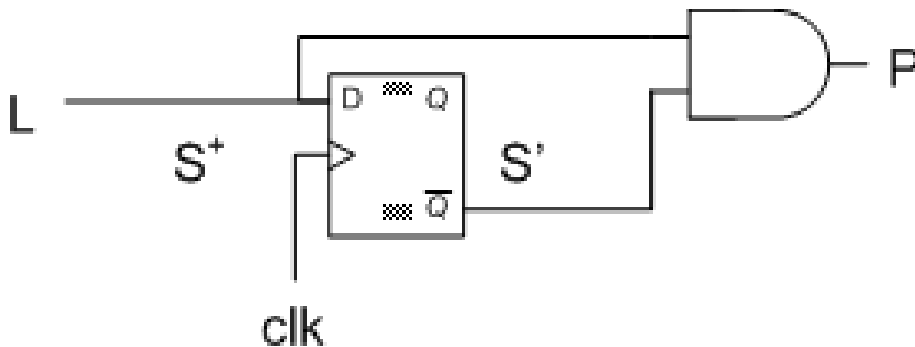
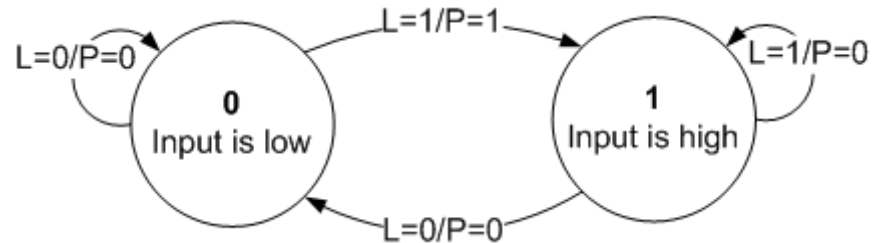
Current State		In	Next State		Out
S_1	S_0	L	S_1^+	S_0^+	P
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	1
1	1	0	0	0	0
1	1	1	1	1	0

FSM Implementation – Logic Implementation



FSM Implementation Mealy implementation

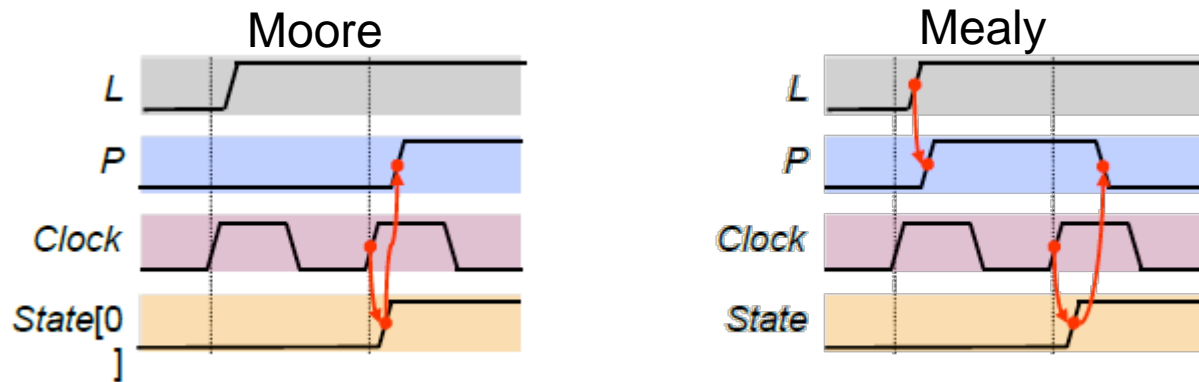
- Since outputs are determined by state and inputs, Mealy FSMs may need fewer states than Moore FSM implementations



Pre State	In	Next State	Out
S	L	S ⁺	P
0	0	0	0
0	1	1	1
1	0	0	0
1	1	1	0

FSM Implementation Moore/Mealy trade-off

- Remember that the difference is in the output:
 - Moore outputs are based on state only
 - Mealy outputs are based on state and input
 - Mealy outputs generally occur one cycle earlier than a Moore:



- Compared to a Moore FSM, a Mealy FSM might...
 - Be more difficult to conceptualize and design
 - Have fewer states

HDL FSM Implementation Binary Encoding

- ❑ Straight encoding of states

$S_0 = \text{"00"} \quad S_1 = \text{"01"} \quad S_2 = \text{"10"} \quad S_3 = \text{"11"}$

- ❑ For n states, there are $\lfloor \log_2(n) \rfloor + 1$ flip-flops needed
- ❑ This gives the least numbers of flip-flops
- ❑ Good for “Area” constrained designs
- ❑ Drawbacks:
 - Multiple bits switch at the same time = Increased noise and power
 - Next state logic is multi-level = Increased power and reduced speed

HDL FSM Implementation Gray-Code Encoding

- ❑ Encoding using a gray code where only one bit switches at a time

$S_0 = "00"$ $S_1 = "01"$ $S_2 = "11"$ $S_3 = "10"$

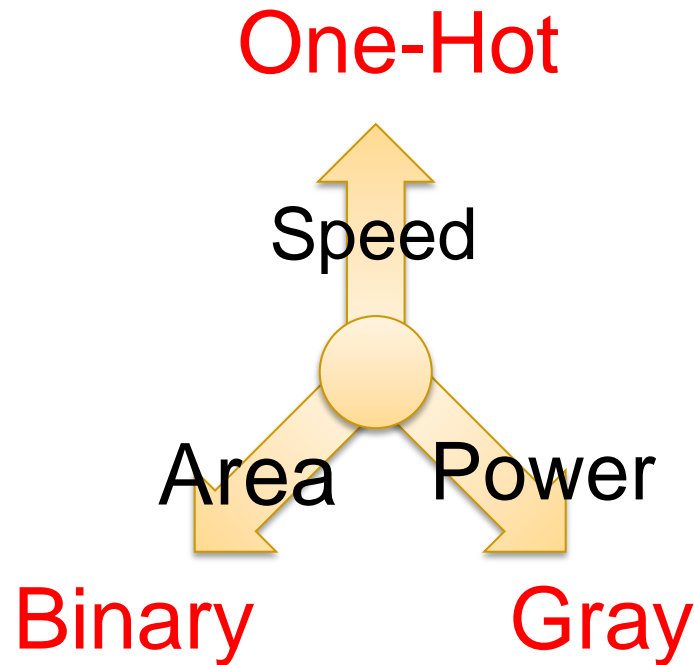
- ❑ For n states, there are $\lfloor \log_2(n) \rfloor + 1$ flip-flops needed
- ❑ This gives low power and noise due to only one bit switching
- ❑ Good for “power/noise” constrained designs
- ❑ Drawbacks:
 - The next state logic is multi-level = Increased power and reduced speed

HDL FSM Implementation *One-Hot Encoding*

- ❑ Encoding one flip-flop for each state
 $S_0 = "0001"$ $S_1 = "0010"$ $S_2 = "0100"$ $S_3 = "1000"$
- ❑ For n states, there are n flip-flops needed
- ❑ The combination logic is one level (i.e., a decoder)
- ❑ Good for speed
- ❑ Especially good for FPGA due to "Programmable Logic Block"
- ❑ Number of possible illegal states = $2^n - n$
- ❑ Drawbacks:
 - Takes more area

HDL FSM Implementation *State Encoding Trade-Offs*

- Typically trade off Speed, Area, and Power



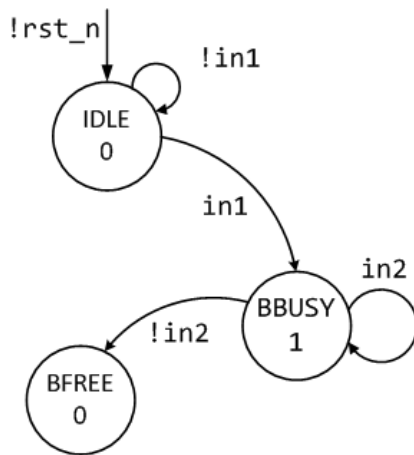
HDL FSM Implementation FSM Coding goals

- ❑ The FSM coding style should be easily modified to change state encodings and FSM styles
- ❑ The coding style should be compact
- ❑ The coding style should be easy to code and understand
- ❑ The coding style should facilitate debugging
- ❑ The coding style should yield efficient synthesis results

HDL FSM Implementation

Two Always Block FSM Style (Good Style)

- ❑ One of the best Verilog coding styles
- ❑ Code the FSM design using two always blocks,
 - One for the sequential state register
 - One for the combinational next-state and combinational output logic.



```
parameter [1:0]
IDLE=2'b00, BBUSY=2'b01, BFREE=2'b10;
reg [1:0] state, next;

always @(posedge clk or negedge rst_n)
if (!rst_n) state <= IDLE;
else      state <= next;

always @(state or in1 or in2) begin
next = 2'bx;  out1 = 1'b0;
case (state)
  IDLE : if (in1) next = BBUSY;
         else      next = IDLE;
  BBUSY: begin
         out1 = 1'b1;
         if (in2) next = BBUSY;
         else      next = BFREE;
         end
  //...
endcase
end
```


HDL FSM Implementation

Two Always Block FSM Style (Good Style)

1. Use parameters to define state names because the state name is a constant that applies only to the FSM module
2. The sequential always block is coded using nonblocking assignments

```
parameter [1:0]
IDLE=2'b00, BBUSY=2'b01, BFREE=2'b10;
reg [1:0] state, next;

always @(posedge clk or negedge rst_n)
if (!rst_n) state <= IDLE;
else      state <= next;

always @(state or in1 or in2) begin
next = 2'bx;  out1 = 1'b0;
case (state)
  IDLE : if (in1) next = BBUSY;
        else      next = IDLE;
  BBUSY: begin
        out1 = 1'b1;
        if (in2) next = BBUSY;
        else      next = BFREE;
        end
  //...
endcase
end
```



HDL FSM Implementation

Two Always Block FSM Style (Good Style)

3. The combinational always block sensitivity list is sensitive to changes on the state variable and all of the inputs referenced in the combinational always block.
4. Assignments within the combinational always block are made using Verilog blocking assignments

```
parameter [1:0]
IDLE=2'b00, BBUSY=2'b01, BFREE=2'b10;
reg [1:0] state, next;

always @(posedge clk or negedge rst_n)
if (!rst_n) state <= IDLE;
else      state <= next;

always @(state or in1 or in2) begin  3
next = 2'bx;  out1 = 1'b0;
case (state)
  IDLE : if (in1) next = BBUSY;
         else      next = IDLE;
  BBUSY: begin  4
    out1 = 1'b1;
    if (in2) next = BBUSY;
    else      next = BFREE;
  end
  //...
endcase
end
```

HDL FSM Implementation


Two Always Block FSM Style (Good Style)

5. Default output assignments are made before coding the case statement (this eliminates latches and reduces the amount of code required)
6. Placing a default next state assignment on the line immediately following the always block sensitivity list is a very efficient coding style



```
parameter [1:0]
IDLE=2'b00, BBUSY=2'b01, BFREE=2'b10;
reg [1:0] state, next;

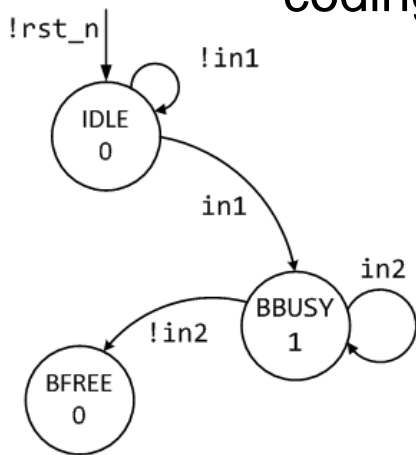
always @(posedge clk or negedge rst_n)
if (!rst_n) state <= IDLE;
else      state <= next;

always @(state or in1 or in2) begin
next = 2'bx; out1 = 1'b0; 
case (state)
  IDLE : if (in1) next = BBUSY;
         else   next = IDLE;
  BBUSY: begin
         out1 = 1'b1;
         if (in2) next = BBUSY;
         else   next = BFREE;
       end
  //...
endcase
end
```

HDL FSM Implementation

One Always Block FSM Style

- ❑ One of the most common FSM coding styles in use today
 - It is more verbose
 - It is more confusing
 - It is more error prone(comparable two always block coding style)

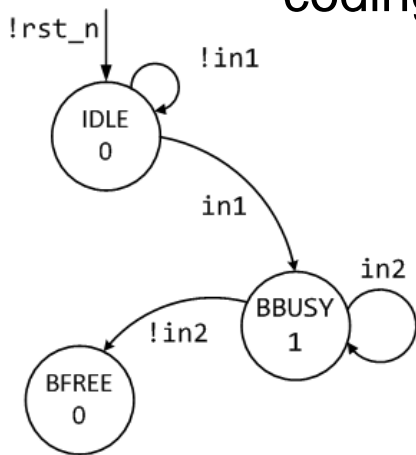


```
parameter [1:0]
IDLE=2'b00, BBUSY=2'b01, BFREE=2'b10;
reg [1:0] state;
always @(posedge clk or negedge rst_n)
if (!rst_n) begin
    state <= IDLE; out1 <= 1'b0;
end
else begin
    state <= 2'bx; out1 <= 1'b0;
    case (state)
        IDLE : if (in1) begin
                    state <= BBUSY;
                    out1 <= 1'b1;
                end
            else state <= IDLE;
        BBUSY: if (in2) begin
                    state <= BBUSY;
                    out1 <= 1'b1;
                end
            else state <= BFREE;
    endcase
end
```

HDL FSM Implementation

One Always Block FSM Style

- ❑ One of the most common FSM coding styles in use today
 - It is more verbose
 - It is more confusing
 - It is more error prone(comparable two always block coding style)



```
parameter [1:0]
IDLE=2'b00, BBUSY=2'b01, BFREE=2'b10;
reg [1:0] state;
always @(posedge clk or negedge rst_n)
if (!rst_n) begin
    state <= IDLE; out1 <= 1'b0;
end
else begin
    state <= 2'bx; out1 <= 1'b0;
    case (state)
        IDLE : if (in1) begin
                    state <= BBUSY;
                    out1 <= 1'b1;
                end
            else state <= IDLE;
        BBUSY: if (in2) begin
                    state <= BBUSY;
                    out1 <= 1'b1;
                end
            else state <= BFREE;
    endcase
end
```

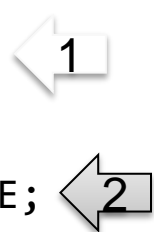
Avoid

HDL FSM Implementation

One Always Block FSM Style

1. A declaration is made for state. Not for next.
2. The state assignments do not correspond to the current state of the case statement, but the state that case statement is transitioning to. This is **error prone** (but it does work if coded correctly).

```
parameter [1:0]
IDLE=2'b00, BBUSY=2'b01, BFREE=2'b10;
reg [1:0] state;
always @(posedge clk or negedge rst_n)
if (!rst_n) begin
    state <= IDLE; out1 <= 1'b0;
end
else begin
    state <= 2'bx; out1 <= 1'b0;
    case (state)
        IDLE : if (in1) begin
                    state <= BBUSY;
                    out1 <= 1'b1;
                end
            else state <= IDLE;
        BBUSY: if (in2) begin
                    state <= BBUSY;
                    out1 <= 1'b1;
                end
            else state <= BFREE;
    endcase
end
```



HDL FSM Implementation

One Always Block FSM Style

3. There is just one sequential always block, coded using nonblocking assignments. Difficult to track the changes
4. All outputs will be registered. No asynchronous Mealy outputs can be generated from a single synchronous always block.

```
parameter [1:0]
IDLE=2'b00, BBUSY=2'b01, BFREE=2'b10;
reg [1:0] state;
always @(posedge clk or negedge rst_n)
if (!rst_n) begin
state <= IDLE;
out1 <= 1'b0;
end
else begin
state <= 2'bx; out1 <= 1'b0;
case (state)
IDLE : if (in1) begin
state <= BBUSY;
out1 <= 1'b1;
end
else state <= IDLE;
BBUSY: if (in2) begin
state <= BBUSY;
out1 <= 1'b1;
end
else state <= BFREE;
endcase
end
```

34

HDL FSM Implementation

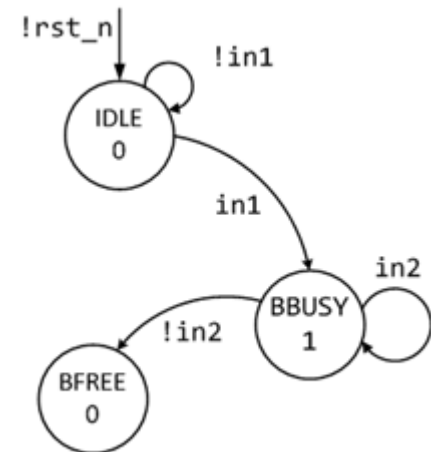
One-hot FSM Coding Style (Good Style)

```
parameter [1:0] IDLE=0, BBUSY=1,BFREE=2;
reg [2:0] state, next;

always @(posedge clk or negedge rst_n)
if (!rst_n) state <= 3'b001;
else
    state <= next;

always @(state or in1 or in2) begin
next = 3'b000;  out1 = 1'b0;
case (1'b1) //ambit synthesis case full, parallel
    state[IDLE] : if (in1) next[BBUSY] = 1'b1;
                  else      next[IDLE]  = 1'b1;
    state[BBUSY]: begin
        out1 = 1'b1;
        if (in2) next[BBUSY] = 1'b1;
        else    next[BFREE]  = 1'b1;
    end
//...
endcase
end
```

Efficient (small and fast) One-hot state machines can be coded using an reverse case statement



HDL FSM Implementation

One-hot FSM Coding Style (Good Style)

1. Index into the state register, not state encodings
2. One-hot requires larger declarations
3. State reset, set to 1 the IDLE bit
4. Next state assignment must make all-0's

```
parameter [1:0] IDLE=0, BBUSY=1, BFREE=2;
reg [2:0] state, next;

always @(posedge clk or negedge rst_n)
if (!rst_n) state <= 3'b001;
else      state <= next;

always @(state or in1 or in2) begin
next = 3'b000; out1 = 1'b0;
case (1'b1) //ambit synthesis case full, parallel
state[IDLE] : if (in1) next[BBUSY] = 1'b1;
               else      next[IDLE]  = 1'b1;
state[BBUSY]: begin
out1 = 1'b1;
if (in2) next[BBUSY] = 1'b1;
else      next[BFREE] = 1'b1;
end
//...
endcase
end
```

HDL FSM Implementation

One-hot FSM Coding Style (Good Style)

- 5. Reverse case statement usage
- 6. Case branch check state values
- 7. Added “full” and parallel case pragmas
- 8. Only the next state bit

```
parameter [1:0] IDLE=0, BBUSY=1, BFREE=2;
reg [2:0] state, next;

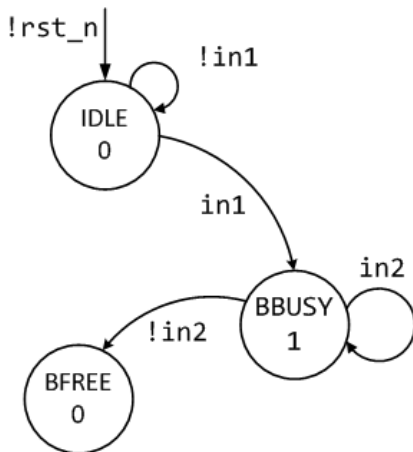
always @(posedge clk or negedge rst_n)
if (!rst_n) state <= 3'b001;
else      state <= next;

always @(state or in1 or in2) begin
next = 3'b000; out1 = 1'b0;
case (1'b1) //ambit synthesis case full, parallel
5 state[IDLE] : if (in1) next[BBUSY] = 1'b1;
6               else      next[IDLE]  = 1'b1;
state[BBUSY]: begin
7               out1 = 1'b1;
if (in2) next[BBUSY] = 1'b1;
else      next[BFREE] = 1'b1;
8               end
//...
endcase
end
```

HDL FSM Implementation

One-hot FSM Coding Style (Good Style)

- ❑ Registering the outputs of an FSM design
 - Insures that the outputs are glitch-free
 - Improves synthesis results by standardizing the output and input delay constraints of synthesized modules



```
output reg out1;
parameter [1:0] IDLE=2'b00,
               BBUSY=2'b01,BFREE=2'b10;
reg [1:0] state, next;
always @(posedge clk or negedge rst_n)
if (!rst_n) state <= IDLE;
else      state <= next;
always @(state or in1 or in2) begin
next = 2'bx;
case (state)
  IDLE : if (in1) next = BBUSY;
         else   next = IDLE;
  BBUSY: if (in2) next = BBUSY;
         else   next = BFREE;
endcase
end
always @(posedge clk or negedge rst_n)
if (!rst_n) out1 <= 1'b0;
else begin
  out1 <= 1'b0;
  case (next)
    IDLE, BFREE: ; // default outputs
    BBUSY, BWAIT: out1 <= 1'b1;
  endcase
end
```

Reset

- ❑ Reset is an important part of the control strategy
 - Used to initialize the chip to a known state
 - Distributed to registers that determine state
 - E.g. FSM state vector
 - Usually asserted on startup and reset
 - Globally distributed
 - Not a designer-controlled signal

