N26F300 VLSI SYSTEM DESIGN (GRADUATE LEVEL)

Fall 2010

Synthesis for Verilog (I)

Outline

- Synthesis Tasks
- Unsupported Verilog Constructs
- Supported Verilog Constructs
- Synthesis of Combinational Blocks

Things That Synthesis Does

- Detect and eliminate redundant logic
- Detect combinational feedback loops
- Exploit don't care conditions
- Detect unused states
- Detect and collapse equivalent states
- Make state assignments
- Synthesize optimal, multilevel realizations of logic subject to constraints in a physical technology

Synthesizable Verilog Codes

- Synopsys can’t accept all kinds of Verilog constructs
- Synopsys can only accept a subset of Verilog syntax and this subset is called “Synthesizable Verilog Code”
- Understanding how to write synthesis-friendly Verilog models is the key to automated design methods.

Unsupported Verilog Constructs (1/5)

- **Definitions and Declarations**
  - **primitive** definition
  - **real** declaration
  - **time** declaration
  - **event** declaration
  - **triand**, **trior**, **tri1**, **tri0**, and **trireg** net types
  - ranges and arrays for **integers**
  - **specify** and **endspecify**

Unsupported Verilog Constructs (2/5)

- Statements (1/2)
  - **defparam**
  - **initial**
  - **repeat** (they are not supported because equivalent functionality can be constructed by other constructs)
  - **delay** (delay-based timing control, intra-assignement timing control)
  - **event** (named event control)
  - **wait** (level-sensitive timing control)
  - **forever**

Unsupported Verilog Constructs (3/5)

- Statements (2/2)
  - **fork/join** (helpful in modeling complex waveform in testbenches and abstract models of behavior)
  - **assign/deassign** for register type (commonly used for controlled periods of time inside a procedural block)
  - **force/release** for registers and nets (typically used in the interactive debugging process)
    - * Recommended to be used in stimulus or as debug statement.
  - Assignment statements with a variable used as a bit-select on the left side of the equal sign.

Unsupported Verilog Constructs (4/5)

- Operators
  - case equality (**===**) and case inequality (**!==**)
  - division and modulus operators for variables (?)
- Miscellaneous constructs
  - hierarchical names within a module
  - **‘ifdef**, **‘endif**, and **‘else** compiler directives

Unsupported Verilog Constructs (5/5)

- Switch-level Constructs
  - **nmos, pmos, cmos**
  - **rnmos, rpmos, rcmos**
  - **tran, tranif0, trnaif1**
  - **rtran, rtranif0, rtranif1**
  - **pullup, pulldown**

Language Constructs commonly supported by Synthesis Tools (I)

- Module declaration
- Port name: **input**, **output**, and **inout**
- Port binding by name
- Port binding by position
- Local parameter declaration
- Connectivity nets: **wire**, **tri**, **wand**, **wor**, **supply0**, and **supply1**
- Register variables: **reg**, **integer**
- integer types in binary, decimal, octal, hex format
- Scalar and vector nets
- Sub_range of vector nets on RHS of assignment
- Module and macro_module instantiation
- Primitive instantiation

Language Constructs commonly supported by Synthesis Tools (II)

- Continuous assignments
- Shift operator
- Conditional operator
- Concatenation operator (including nesting)
- Arithmetic, bitwise, reduction, logical and relational operators
- Procedure block assignments (**begin … end**)
- **case**, ~~**casex**,~~ **casez**, **default**
- Branching: **if, if … else, if … else … if**
- **disable** (of procedure block)
- **for** loop
- Tasks: **task … endtask** (no timing or event control)
- Functions: **function … endfunction**

Practical considerations

- Non-blocking & blocking statements
- Resource sharing
- Conditional operator
- If, case, for statements
- Parenthesis – multi-level circuit
- Minimizing registers
- Don’t care inference
- Sharing complex operators
- Finite state machine
- Latches & Flip-flops
- Synopsys compiler directives
- Some Synthesis tips

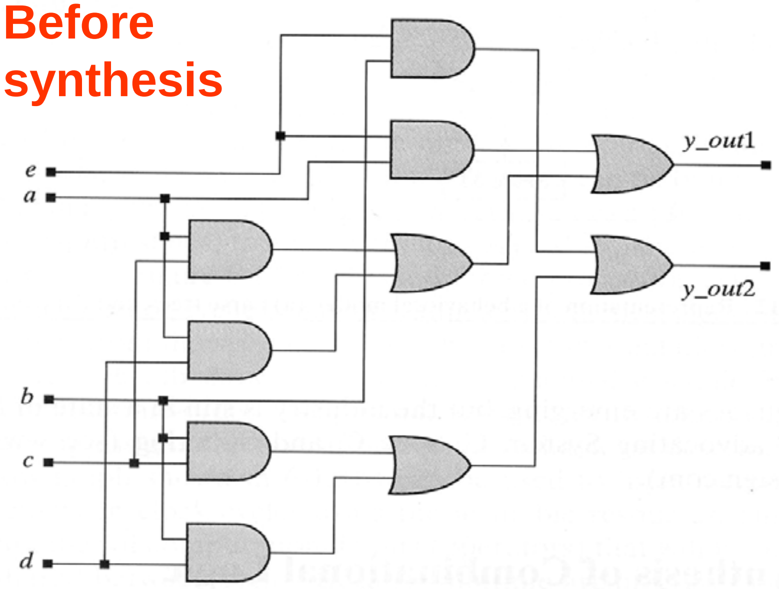
Synthesis of Combinational Logic

Synthesizable Statements for Combinational Logic

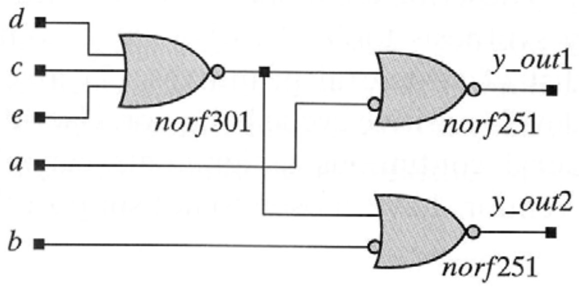
- a netlist of structural primitives
  - A design that is expressed as a netlist of primitives should be synthesized to remove any redundant logic before mapping the design into a technology.
- a set of continuous assignment statements
  - The expression that assigns value to a net variable in a continuous assignment statement will be translated by the synthesis tool to an equivalent Boolean equation.
- a level-sensitive cyclic behavior (e.g., always @(a or b))
  - It will be synthesized to combinational logic if it assigns a value to each output for every possible value of its inputs.

Structural primitives

Before synthesis



After synthesis



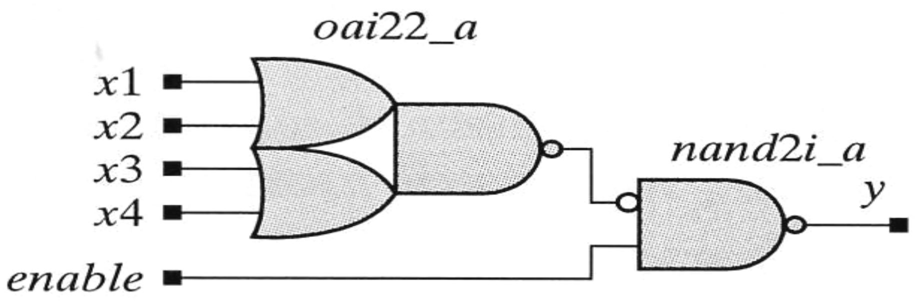
```
module  boole_opt  (y_out1, y_out2, a,
b, c, d, e);
 output        y_out1, y_out2;
  input        a, b, c, d, e;

  and      (y1, a, c);
  and      (y2, a, d);
  and      (y3, a, e);
  or       (y4, y1, y2);
  or       (y_out1, y3, y4);
  and      (y5, b, c);
  and      (y6, b, d);
  and      (y7, b, e);
  or       (y8, y5, y6);
  or       (y_out2, y7, y8);
endmodule
```

Continuous Assignment

```
module or_nand (y, enable, x1, x2, x3, x4);
  output y;
  input     enable, x1, x2, x3, x4;

  assign y = ~(enable & (x1 | x2) & (x3 | x4));
endmodule
```

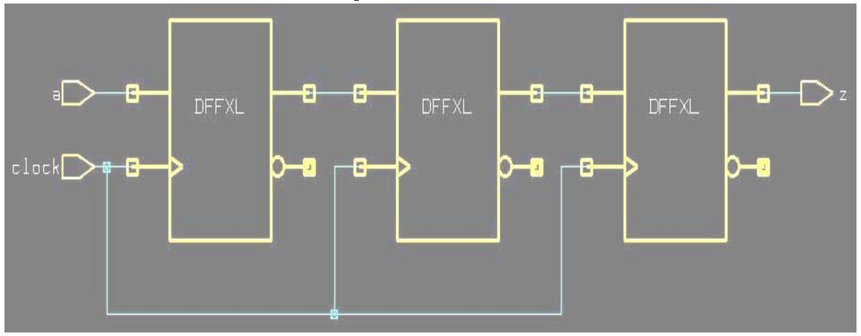


Blocking and non-Blocking (1/2)

- Use non_blocking assignments within sequential always block.

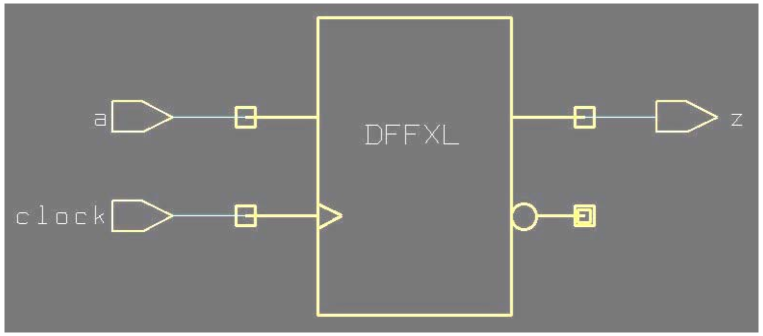
```
always @(posedge clock) begin
   x <= a;
   y <= x;
   z <= y;
end
```

expect



```
always @(posedge clock) begin
   x = a;
   y = x;
   z = y;
end
```

may not expect

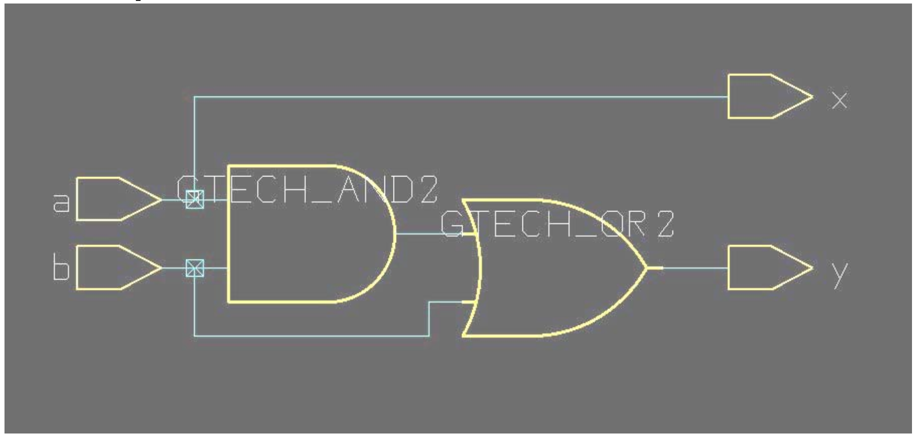


Blocking and non-Blocking (2/2)

- Use blocking assignments within combinational always block

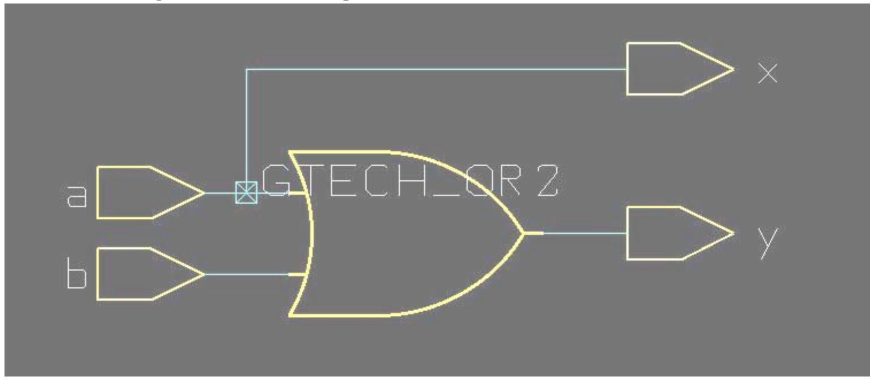
expect

```
always @(a or b or x) begin
   x = a & b;
   y = x | b;
   x = a;
end
```



may not expect

```
always @(a or b or x) begin
   x <= a & b;
   y <= x | b;
   x <= a;
end
```

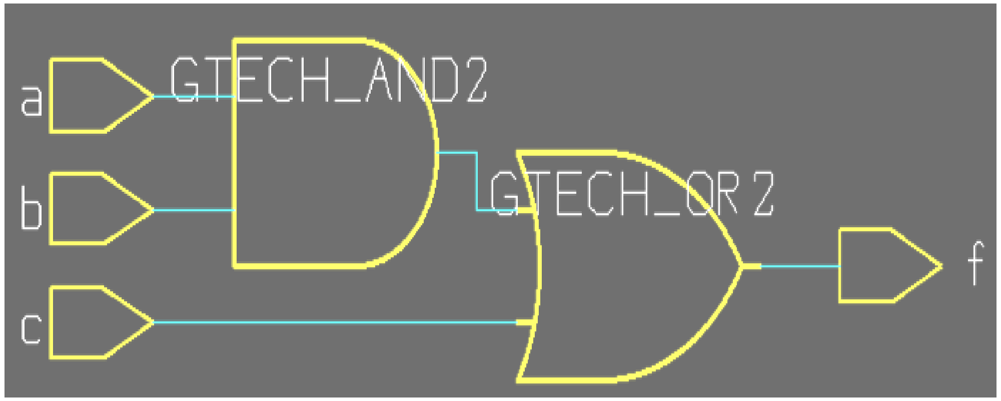


Combinational Always Block

- Sensitivity list must be specified completely, otherwise synthesis may mismatch with simulation

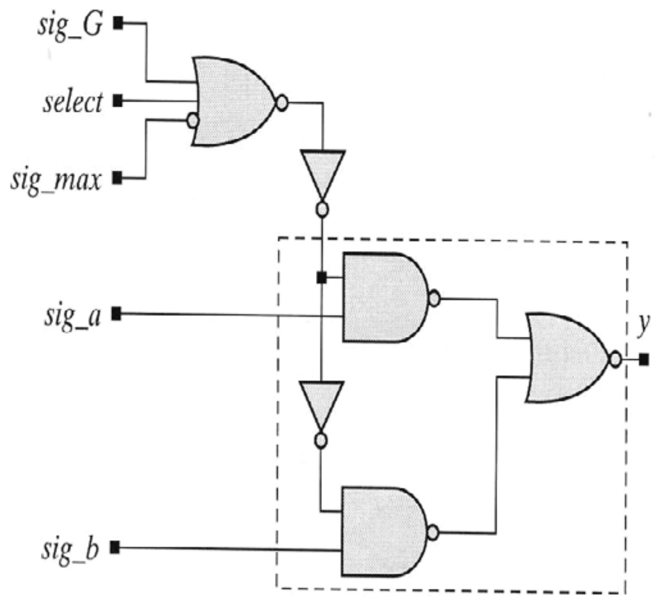
```
always @(a or b or c)
   f = a&b|c;
```

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



```
Warning: Variable ‘c’ is being read
        in routine train line 6 in file but does not occur in the timing
        control of the block which begins there. (HDL-180)
```

MUX with Selector Logic



```
module mux_logic (y, select, sig_G, sig_max,
sig_a, sig_b);
  output y;
  input   select, sig_G, sig_max, sig_a, sig_b;

  assign y = (select == 1) || (sig_G ==1) ||
(sig_max == 0) ? sig_a : sig_b;

endmodule
```

Note: Mux_logic has logic to determine where *sig_a* or *sig_b* is selected.

If Statement (1/2)

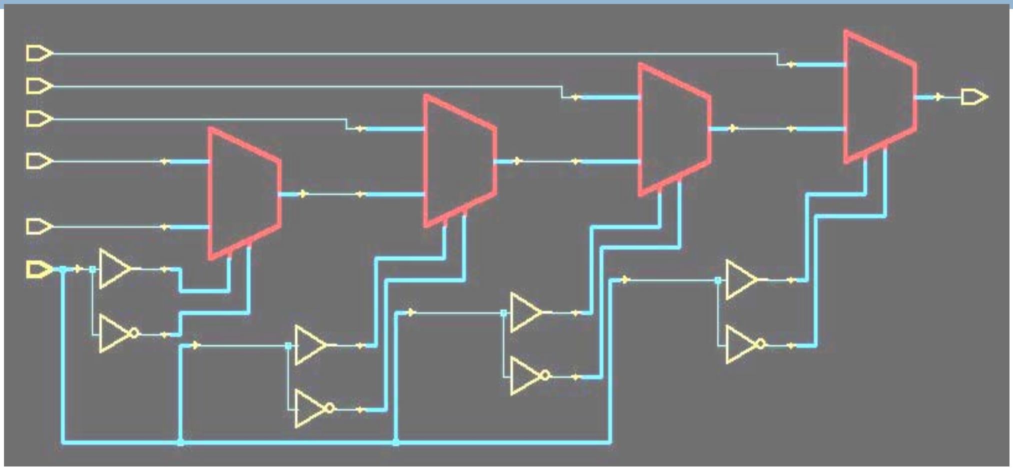
- What's the difference between these two coding styles?

```
module mult_if(a, b, c, d, e, sel, z);
input a, b, c, d, e;
input [3:0] sel;
output z;
reg z;
always @(a or b or c or d or e or sel)
begin
   z = e;
   if(sel[0]) z = a;
   if(sel[1]) z = b;
   if(sel[2]) z = c;
   if(sel[3]) z = d;
end
endmodule
```

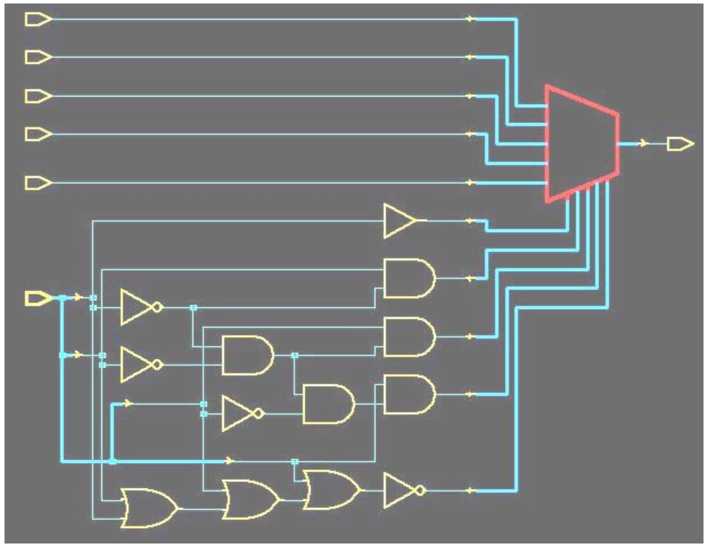
```
module single_if(a, b, c, d, e, sel, z);
input a, b, c, d, e;
input [3:0] sel;
output z;
reg z;
always @(a or b or c or d or e or sel)
begin
   z = e;
   if(sel[3]) z = d;
   else if(sel[2]) z = c;
   else if(sel[1]) z = b;
   else if(sel[0]) z = a;
end
endmodule
```

If Statement (2/2)

mult_if



single_if



Priority / Non-Priority Circuit

priority encoded

```
always @(X or Y or Z) begin
  if (Z) value = result4;
    else if (Y) value = result3;
      else if (X) value =
result2;
        else value = result1;
end
```

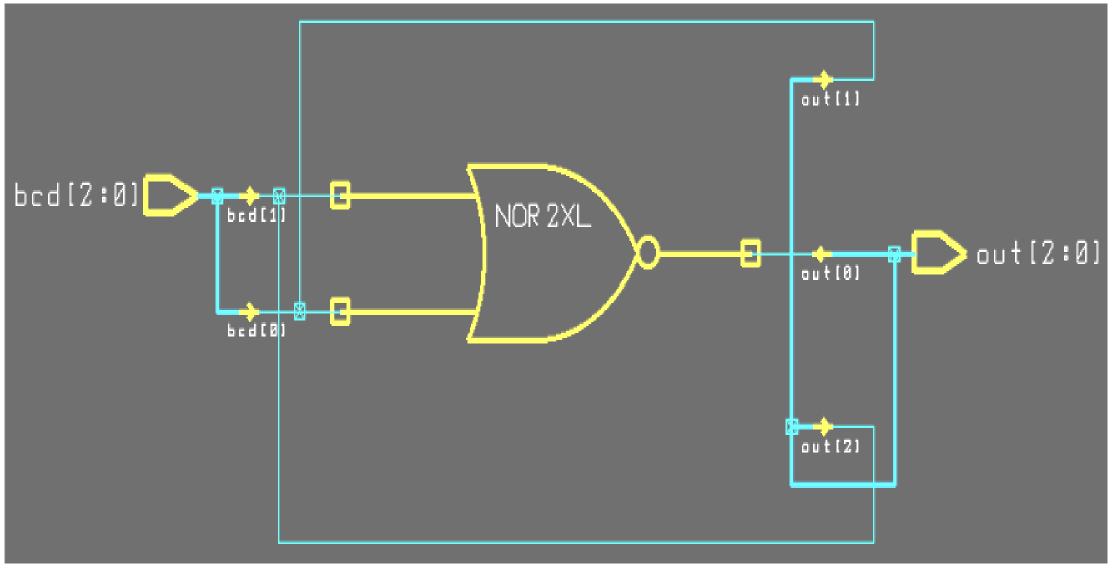
No priority encoded

```
always @(X or Y or Z)
case(1`b1) //synopsys parallel_case
  X: value = result2;
  Y: value = result3;
  Z: value = result4;
  default: value = result1;
endcase
```

Case Statement (1/4)

- A case statement is called a full case if all possible branches are specified

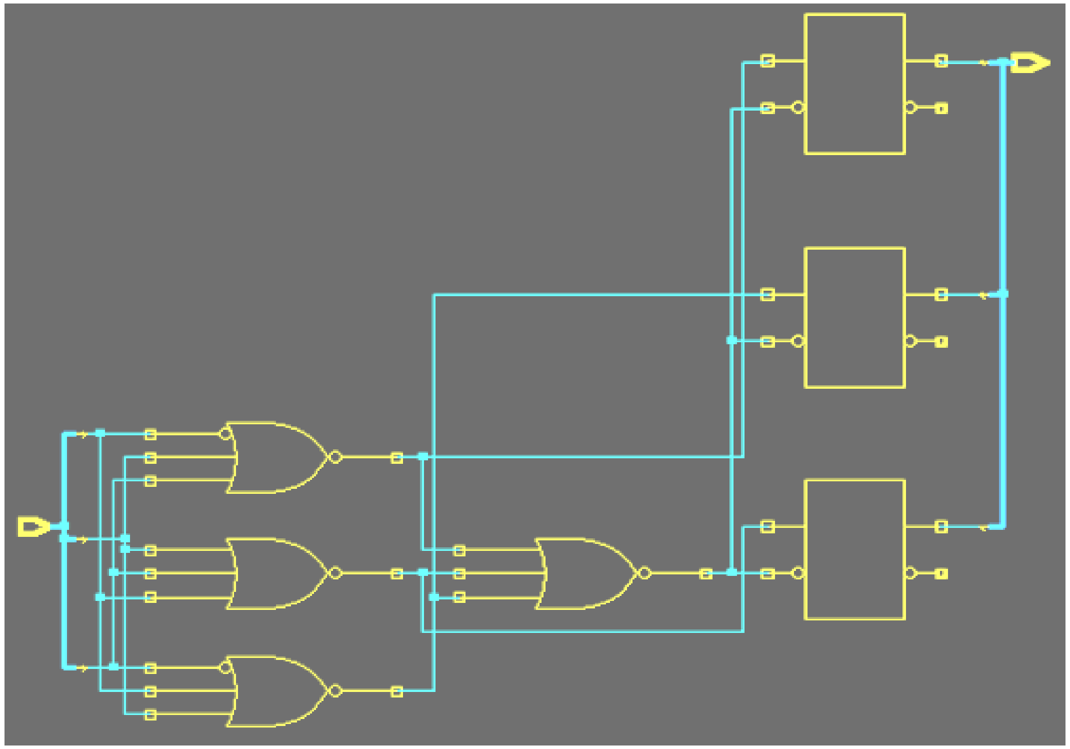
```
always @(bcd) begin
   case (bcd)
      4'd0: out = 3'b001;
      4'd1: out = 3'b010;
      4'd2: out = 3'b100;
      default: out = 3'bxxx;
   endcase
end
```



Case Statement (2/4)

- If a case statement is not a full case, it will infer a *latch* to retain its previous value (as inferred by the semantic).

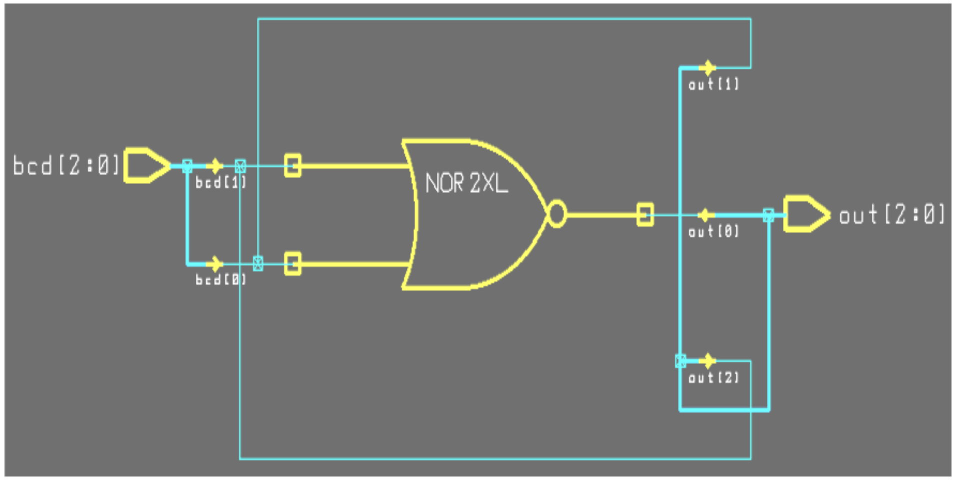
```
always @(bcd) begin
   case (bcd)
      4’d0: out = 3’b001;
      4’d1: out = 3’b010;
      4’d2: out = 3’b100;
   endcase
end
```



Case Statement (3/4)

- If you do not specify all possible branches, but you know the other branches will never occur, you can use "*//synopsys full_case*" directive to specify full case

```
always @(bcd) begin
   case (bcd) //synopsys full_case
   4'd0:out=3'b001;
   4'd1:out=3'b010;
  4'd2:out=3'b100;
   endcase
end
```



Case Statement (4/4)

- **Note:** the second case item does not modify *reg2*, causing it to be inferred as a latch (to retain last value).

```
case (cntr_sig) // synopsys full_case
2'b00 : begin
                reg1 = 0 ;
                reg2 = v_field ;
           end
2'b01 : reg1 = v_field ; /* latch will be inferred for reg2*/
2'b10 : begin
                reg1 = v_field ;
                reg2 = 0 ;
            end
endcase
```

Logical Don't-Care Conditions

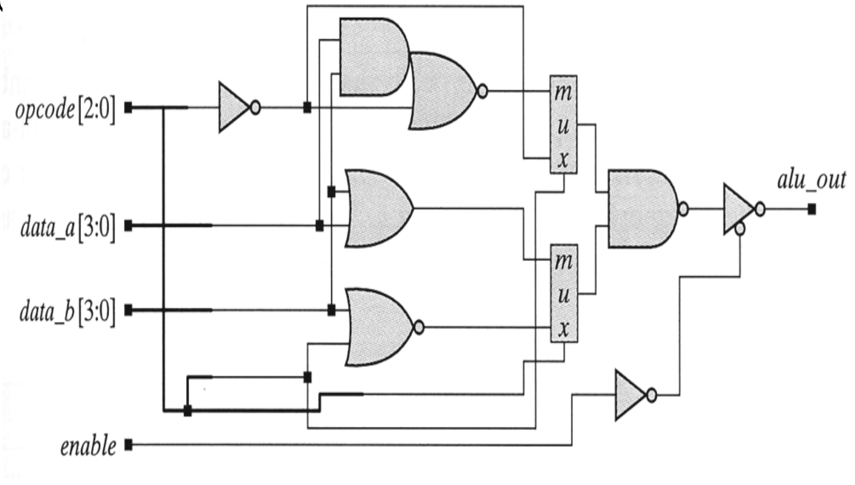
- If the default or branch statement is an explicit assignment of an *X* or a *Z*
  - Simulation results may different
  - Treated as don't care condition ➔ lead to a more compacted circuit

ALU with Z1

```
module alu_with_z1 (alu_out, data_a, data_b,
enable, opcode);
  input    [2: 0]  opcode;
  input    [3: 0]  data_a, data_b;
  input            enable;
  output           alu_out;   // scalar for illustration
  reg      [3: 0]  alu_reg;

  assign alu_out = (enable == 1) ? alu_reg : 4'bz;

  always @  (opcode or data_a or data_
    case (opcode)
      3'b001:   alu_reg = data_a | data_b;
      3'b010:   alu_reg = data_a ^ data_b;
      3'b110:   alu_reg = ~data_b;
      default:  alu_reg = 4'b0;
    endcase
endmodule
```

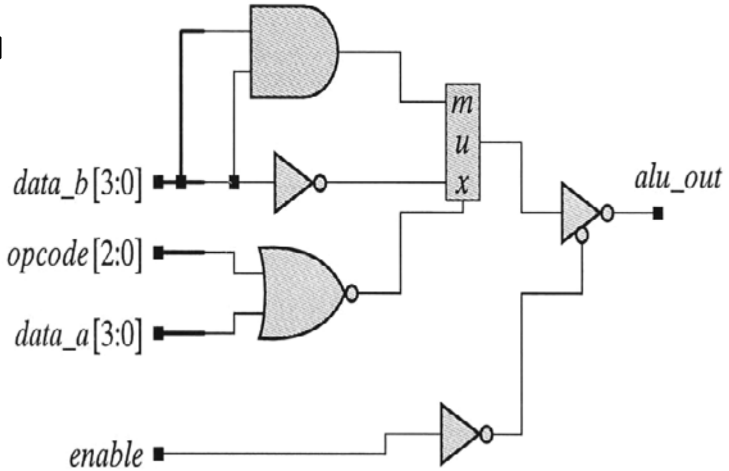


ALU with Z2

```
module alu_with_z1 (alu_out, data_a, data_b,
enable, opcode);
  input     [2: 0]  opcode;
  input     [3: 0]  data_a, data_b;
  input          enable;
  output              alu_out;        // scalar for illustration
  reg       [3: 0]  alu_reg;

  assign alu_out = (enable == 1) ? alu_reg : 4'bz;

  always @  (opcode or data_a or data_b
    case (opcode)
      3'b001:   alu_reg = data_a | data_b;
      3'b010:   alu_reg = data_a ^ data_b;
      3'b110:   alu_reg = ~data_b;
     default: alu_reg = 4'bx;
    endcase
endmodule
```



Notes for Don’t Care Inference

- In some cases, using don’t care values as default assignments can cause these problems:
  - Don’t care values create a greater potential for mismatches between simulation and synthesis.
  - Defaults for variables can hide mistakes in the Verilog code.
  - To a simulator, a don’t care is a distinct value, different from a 1 or a 0.
  - In synthesis, a don’t care becomes a 0 or a 1 (and hardware is built that treats the don’t care value as either a 0 or a 1).
- The safest way to use don’t care values is to
  - Assign don’t care values only to output ports.
  - Make sure that the design never reads those don’t care values.