

CZ/CE 1005

Digital Logic

Recap and Discussion

Lecture 15 & 16

Verilog

Summary of Lecture 15

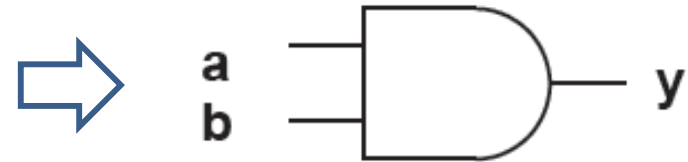
- Introduction to Verilog
 - Previously (in L14) we covered
 - Module Declaration;
 - Instantiating Gate-Level Primitives;
 - Module Instantiation
 - In Lecture 15 we covered
 - Verilog Assignments
 - Vectors in Verilog
 - Number literals and Parameters

Verilog Assignments

- Implementing larger circuits using individual gates can be tedious.

Instantiating a
gate primitive

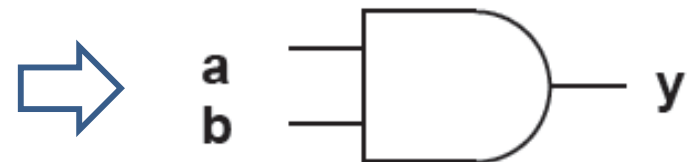
```
and (y, a, b);
```



- Verilog allows us to assign the result of a combinational Boolean expression to a signal through the **assign** keyword:

Continuous
assignment

```
assign y = a & b;
```



Operator	Name
~	bit-wise NOT
&	bit-wise AND
~&	bit-wise NAND
	bit-wise OR
~	bit-wise NOR
^	bit-wise XOR
~^	bit-wise XNOR

Precedence

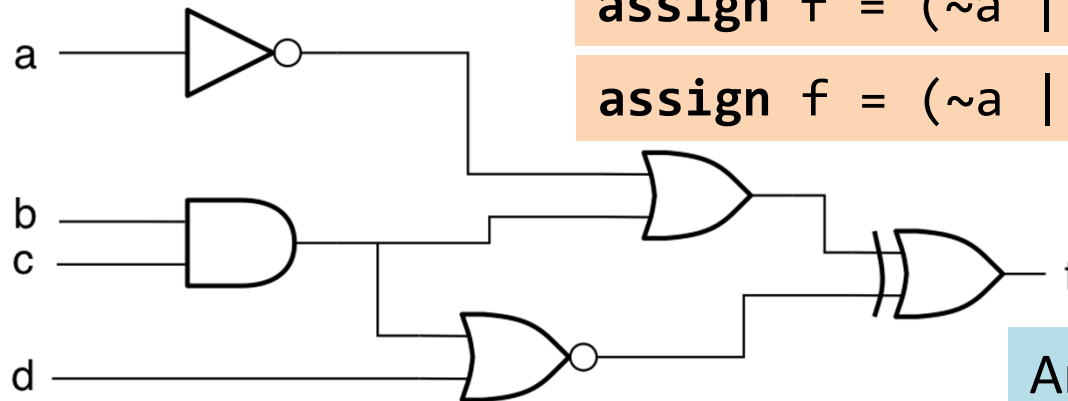
- The **assign statement** allows the use of more complex operators and operands

```
assign x = a&~b | (c^d)&e;
```

```
assign x = (a&(~b)) | ((c^d)&e);
```

Recap: Verilog Assignments

■ Continuous assignment



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

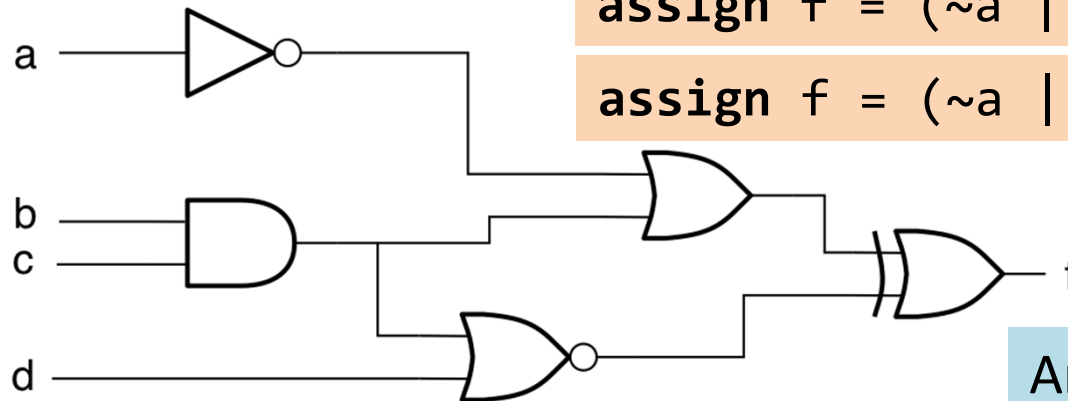
```
assign f = (~a | b&c) ^ ~(b&c | d);
```

Are the 2 expressions the same?

Note: **$a \sim \& b$** is the same as **$\sim(a \& b)$** , both will produce a NAND gate.
But **$a \& \sim b$** is NOT a NAND gate. It will produce an AND gate with a bubble input on b.

Recap: Verilog Assignments

■ Continuous assignment

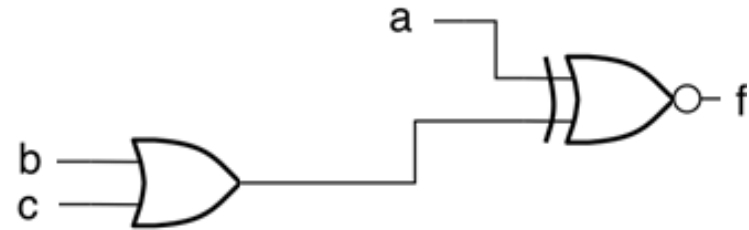
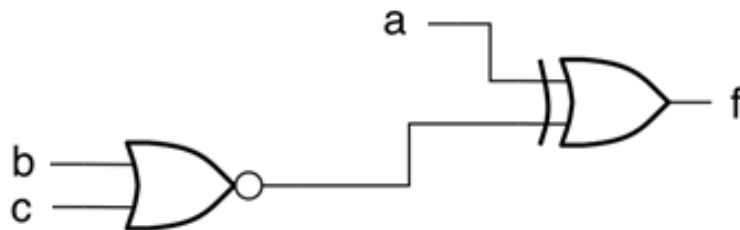


```
assign f = (~a | (b&c)) ^ ((b&c) ~| d);
```

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

Are the 2 expressions the same?

Is “ $a \wedge \sim(b \mid c)$ ” a NOR followed by an XOR or an OR followed by an XNOR?
Consider the following

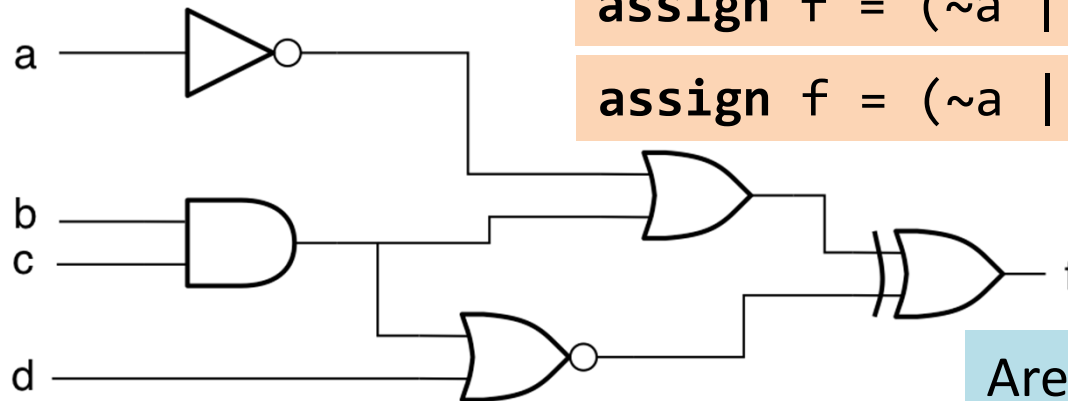


Is there any difference in functionality?

No... But be careful. Add brackets if you are worried.

Recap: Verilog Assignments

Continuous assignment



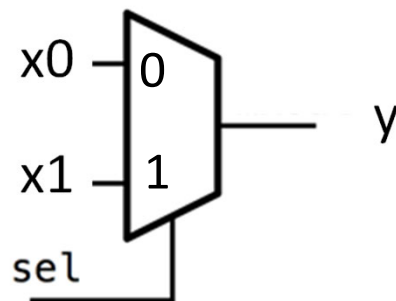
```
assign f = (~a | (b&c)) ^ ((b&c) ~| d);
```

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

Are the 2 expressions the same?

Conditional assignment

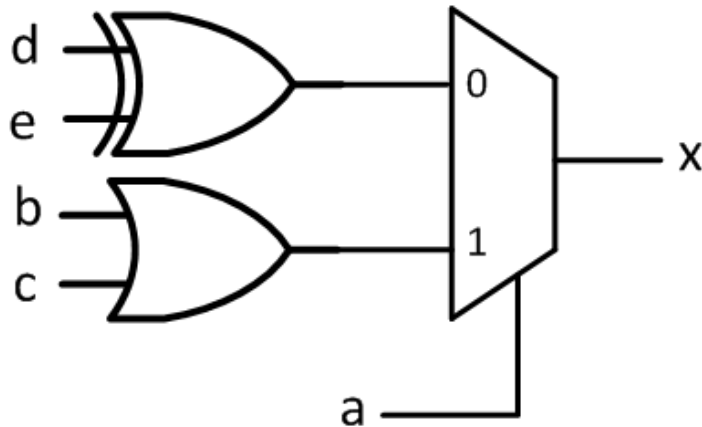
1-bit 2x1 mux



```
assign y = sel ? x1 : x0;
```

Exercise 1

- Which of the following is the correct continuous assignment for the circuit below?



- A. `assign x = a ? (b^c) : (d|e);`
- B. `assign x = a ? (d^e) : (b|c);`
- C. `assign x = a ? (b|c) : (d^e);`
- D. `assign x = a ? (d|e) : (b^c);`
- E. None of the above.



Vectors

- Verilog has a special construct for handling multi-bit signals (wires). **Formed by specifying a range:**
- By convention, we label the *most significant bit* (MSB) using the higher number, and the *least significant bit* (LSB) using zero

```
wire [31:0] databus;
```

- Also for specifying multi-bit module ports

```
module add16 (input [15:0] a, b,  
              output [15:0]sum,  
              output cout);  
  
    // add module internals here  
  
endmodule
```

$\overline{\text{32}}$ databus

is short
for

— databus[31]

— databus[30]

—

— databus[0]

Recap: Vectors and Arithmetic Operations

- Arithmetic: +, −, *
- Comparisons: <, <=, >, >=, ==, !=

```
module unsigned_4bit_adder (  
    input Cin,  
    input [3:0] A, B,  
    output Cout,  
    output [3:0] Sum);  
  
    assign {Cout, Sum} = A+B+Cin;  
  
endmodule
```

**We must declare
multi-bit signals**

**Even easier if we use
the concatenation
operator**

Recap: Vectors in Verilog

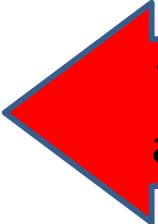
- We can select/assign to *individual bits* or a *range* of the vector

```
assign x_bit = y_bus[3];
```

```
assign x_bus[0]    = y_bus[1];  
assign x_bus[2:1]  = y_bus[3:2];
```

- Widths of vectors in assignments should match

```
assign x[2:0] = y[1];  
assign x[2:1] = a; // a 1-bit
```



This will not generate an error. So be careful

- Also check the Synthesis warnings for width mismatch

WARNING:HDLCompiler:189 "C:\path\myDesign.v" Line 25: Size mismatch in connection of port <sum>. Formal size is 16-bit, actual signal size is 1-bit.

Lecture 15 (Quiz)

Which of the following should be avoided in Verilog, assuming x is 4-bit signal?

- A. `assign y[4:0] = x;`
- B. `assign x = y[9:6];`
- C. `assign x[1:0] = y[4:3];`
- D. `assign y[3:1] = x[2:0];`



Ans:

This will not produce an error.
But what actually happens?

Ans: The most significant bit of y will be padded with a zero. This is a real problem if we are dealing with 2's complement numbers.

Worse is: `assign y[2:0] = x;`

This truncates and discards the most significant bit of x.

Number Literals

- Verilog allows us to use number *literals*:
 - `<size>'<radix><value>`
 - `<size>` is the width **in bits**
 - `<radix>`: b for binary, o for octal, h for hex, d for decimal
 - `<value>`: the number you want, with as many optional underscores as needed (for readability)
- Examples:
 - `4'b0000` (4 binary bits “0 0 0 0”)
 - `8'h4F` (= `8'b01001111`)
 - `8'b0100_1111` (Same as above. Note use of the underscore)
 - `1'b1` (a single “1”)

Verilog is fairly forgiving. For example:

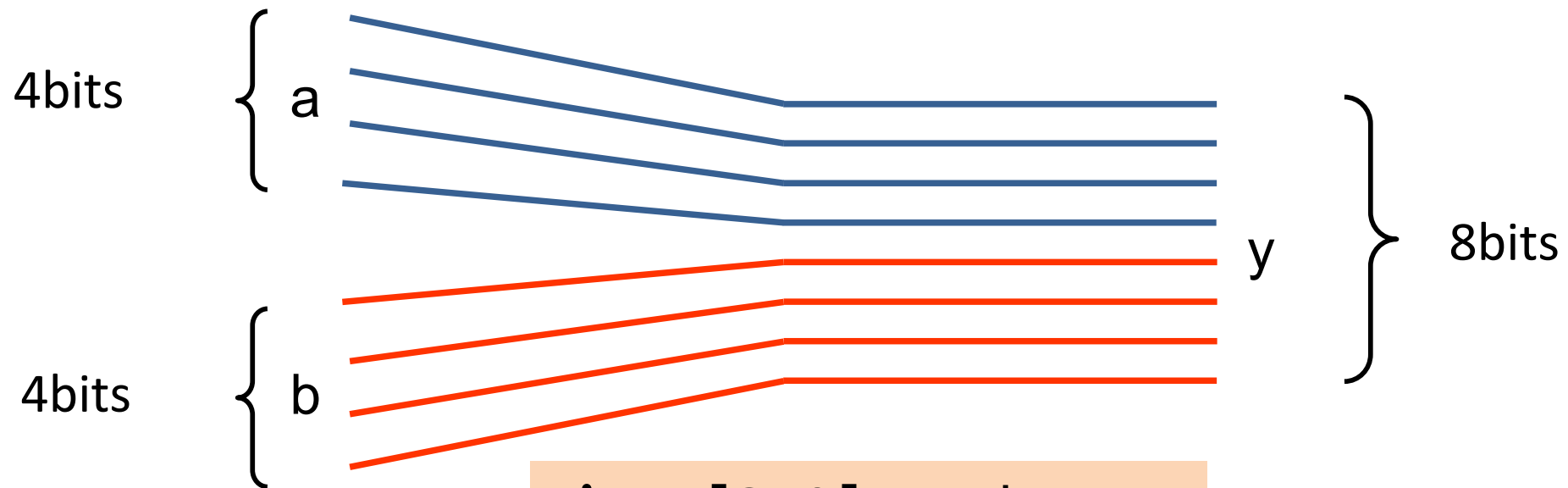
```
assign x = 1'b1;
```

```
assign x = 1;
```

achieve the same thing & both are valid. When in doubt use the top format.

Concatenation

- It is sometime very useful to be able to concatenate a number of signals into a single signal.
 - Concatenation is signified by curly brackets enclosing a list



```

wire [3:0] a, b;
wire [7:0] y;

...

assign y = {a, b};
    
```

Concatenation (Example)

```
// b = 2'b11, c = 2'b00, d = 3'b110
```

```
wire [3:0] x;
```

```
wire [7:0] y;
```

```
assign x = {b, c};
```

```
// x = 4'b1100
```

```
assign y = {b, d, 3'b011};
```

```
// y = 8'b11_110_011
```

Can also have:

```
assign {Cout, Sum} = A + B + Cin;
```

Exercise 2

- What is the resulting signal value of z after the following assign statement?

```
// a = 5'b01011, b = 2'b10, c = 2'b00, d = 3'b100  
  
wire [7:0] z;  
  
assign z = {a[3:0], b[1], c, d[2]};
```

- A. z = 8'b0101_0000
- B. z = 8'b1011_1001
- C. z = 8'b0101_1001
- D. z = 8'b1011_0000

Firstly, check there are the same number of bits on each side of the “=” in the assign statement.

The RHS is 4+1+2+1 = 8bits and z is 8bits, so no bit extension or truncation of z.

So Answer is: 1011_1_00_1 (Ans B).

Replication

- Replication uses braces with a preceding integer or variable representing an integer.

```
// a = 1'b1, c = 4'b1010
```

```
wire [3:0] x;
```

```
wire [7:0] z;
```

```
assign x = { 4{a} };
```

```
// x = 4'b1111
```

```
assign z = {{4{c[3]}}}, c[3:0]};
```

This is the preferred way of
sign extending a number

```
// z = 8'b1111_1010
```


- A **parameter** is a constant that is local to a module
 - Note the use of the **arithmetic addition operator (+)** to generate the sum
 - Note that the **concatenation** operator handles the 33-bit result produced by $A+B+C_{in}$

```
module adder #(parameter SIZE=32) (  
    input Cin,  
    input [SIZE-1:0] A, B,  
    output Cout,  
    output [SIZE-1:0] Sum);  
  
    assign {Cout, Sum} = A+B+Cin;  
  
endmodule
```

Parameters

- It is also possible to override a **parameter**
 - Consider the module (submod) with SIZE = 8:

```
module submod #(parameter SIZE = 8) (  
    input    [SIZE-1:0] X, Y, output [SIZE-1:0] Z);  
    // some statements in here  
endmodule
```

- When instantiating submod, it is possible to override SIZE as:

```
module top_mod #(parameter Const = 16) (  
    input    [Const-1:0] a, b, c, output [Const-1:0] D, E);  
  
    submod #(.SIZE(Const)) U1 (.X(a), .Y(b), .Z(D));  
    submod #(.SIZE(Const)) U2 (.X(c), .Y(b), .Z(E));  
  
endmodule
```

Parameters Override

- Consider the following sub-module

```
module adder #(parameter SIZE=32) (input Cin, input [SIZE-1:0] A, B,  
                                     output Cout, output [SIZE-1:0] Sum);  
    assign {Cout, Sum} = A + B + Cin;  
endmodule
```

- Now consider the adder2 module which instantiates 2 adder modules. Note one overrides SIZE to 6, while the other overrides SIZE to 12.

```
module adder2 (input Ci1, Ci0, input [11:0] A1, B1, input [5:0] A0, B0,  
               output Co1, Co0, output [11:0] S1, output [5:0] S0);  
    adder #(.SIZE(6)) u1 (.Cin(Ci0), .A(A0), .B(B0), .Cout(Co0), .Sum(S0));  
    adder #(.SIZE(12)) u2 (.Cin(Ci1), .A(A1), .B(B1), .Cout(Co1), .Sum(S1));  
endmodule
```

- Is this OK. That is, would it synthesise?

Yes, perfectly OK

As there are 2 separate
adder instances, u1 and u2

Summary of Lecture 16

■ So far we have only used **Structural Verilog** (for Combinational Circuits)

- Previously (in L14 and L15) we covered

- Module Declaration

- Instantiating Gate-Level Primitives

- Module Instantiation

- Verilog Assignments

- Vectors in Verilog

- Number literals and Parameters

	Structural Verilog
--	--------------------

■ Next we look at **Behavioural Verilog**

- Firstly, again just for Combinational Circuits

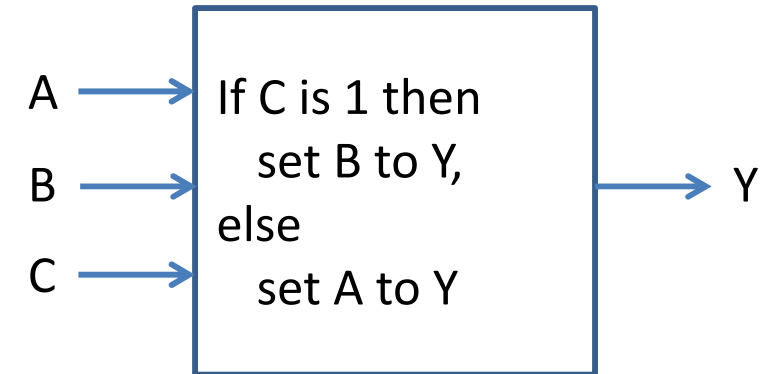
- Behavioural Modelling

- If and Case Statements

- Behavioural Verilog

Behavioral Modeling

- A more powerful level: we describe **how** the circuit **behaves**, not how it is *constructed*
- For combinational circuits we use a **combinational always block** to describe the behaviour of the desired hardware



```

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

BUT... this will not generate the expected output as it will not be triggered when input c changes

```

always @ *
begin
    x = a & b;
    y = a | b;
    z = a & c;
end
  
```

The "*" means all input signals inside the block.

This is the **PREFERRED** method. Whenever any signal changes (* means all input signals) the always block is invoked.

Reg

- In Structural Verilog an **assign** statement was used to assign to a signal
- In Behavioural Verilog, when you assign from within an ***always*** block, the signals MUST be declared as being of type: reg
- **reg** is synonymous with wire, but these signals can be assigned to from inside an always block, **wires** cannot!
 - A **wire** is simply a *connection*, it holds no value of its own
 - The **reg** type is more like a *variable* in programming
- Note: you *cannot* assign to a **reg** signal using an ***assign*** statement, or connect it to module instance outputs.

```
module temp (input a, b,  
             output out);  
  
    wire w1;  
  
    assign w1 = a & b;  
    assign out = w1;  
  
endmodule
```

```
module temp (input a, b,  
             output out);  
  
    reg w1;  
  
    assign out = w1;  
    always @ *  
        w1 = a & b;  
  
endmodule
```

Exercise 4

- State **all** signals in the following Verilog code segment that must be declared as **reg**.

```
module temp (...);  
    ...  
    assign w1  = a ^ b;  
    always @ *  
    begin  
        r1      = a & b;  
        out_reg = a | w1;  
        out_bus = 5'd25;  
    end  
    assign out = r1;  
endmodule
```

Ans: B

The 3 signals that are assigned to from inside the always block. That is : r1, out_reg, out_bus.

- A. out_reg, out_bus, a, b
- B. r1, out_reg, out_bus
- C. out_reg, out_bus, a, b
- D. r1, out_reg, out_bus, w1

Recap: If Statement

- If there is more than 1 statement in a combinational always block you must use **begin** and **end**

```
always @ *  
begin  
    if (alarm == 1'b1)  
        begin  
            if (after_hours)  
                siren = 1'b0;  
            else  
                siren = 1'b1;  
            light = 1'b1;  
        end  
    else  
        begin  
            siren = 1'b0;  
            light = 1'b0;  
        end  
end
```

These are the same

```
always @ *  
begin  
    if (x < 6)  
        alarm = 1'b0;  
    else  
        alarm = 1'b1;  
end
```

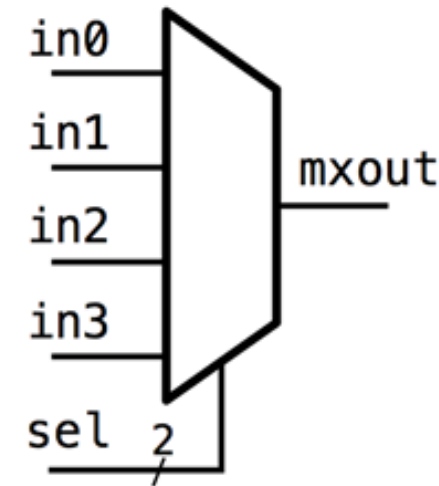
```
always @ *  
if (x < 6)  
    alarm = 1'b0;  
else  
    alarm = 1'b1;
```

But, it is good practice to use **begin** and **end** to delineate the always block

Recap: Case Statement

```
module mux4 (input [3:0] in,  
             input [1:0] sel,  
             output reg mxout);  
  
    always @ * begin  
        case (sel)  
            2'b00 : mxout = in[0];  
            2'b01 : mxout = in[1];  
            2'b10 : mxout = in[2];  
            2'b11 : mxout = in[3];  
        endcase  
    end  
endmodule
```

1-bit 4x1 mux



Need to ensure that the output(s) are assigned to for all possible cases (of sel).
Better to use a default assignment.

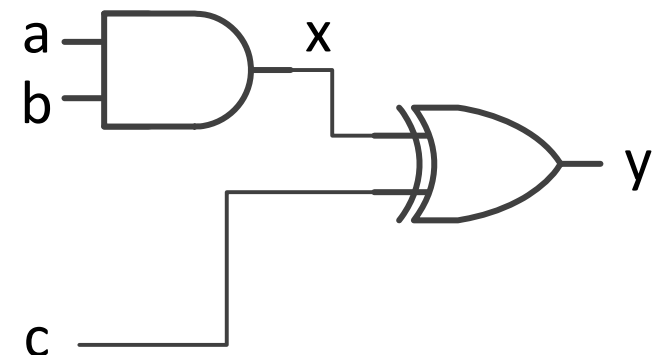
```
always @ *  
    case (sel)  
        2'b00 : y = a;  
        2'b01 : y = b;  
        2'b10 : y = c;  
        default : y = 4'b1010;  
    endcase
```

```
always @ *  
    y = 4'b1010;  
    case (sel)  
        2'b00 : y = a;  
        2'b01 : y = b;  
        2'b10 : y = c;  
    endcase
```

Recap: Behavioral Verilog for Combinational Circuits

- Include all signals that affect the output in the sensitivity list
 - Easiest way is to just use **always @ ***
- If you assign to a signal from inside an **always** block, you must **never** assign to it from **anywhere else**
- Order of statements **matter** within a combinational always block
 - Order of multiple **always** blocks in a module doesn't matter

```
always @ *  
begin  
    x = a & b ;  
    y = x ^ c ;  
end  
  
assign y = a + b ;  
  
always @ (a or b)  
begin  
    u = a | b ;  
    v = b & c ;  
end
```



Recap: Avoiding Latches

- Must assign to the output signal in all possible cases
 - Otherwise a latch may be synthesized

```
always @ *  
begin  
    y = 0;  
    if(valid) begin  
        x = a | b;  
        y = c;  
    end  
    else  
        x = a;  
end
```

```
always @ *  
begin  
    y = 4'b1010;  
    case (sel)  
        3'b000 : y = a;  
        3'b010 : y = b;  
        3'b100 : y = c;  
        3'b110 : y = d;  
    endcase  
end
```

- Use a default assignment at the top of the always block
 - The default is overwritten by any subsequent assignment

Recap: Avoiding Latches

```
module mux (input [3:0] in
            input [1:0] sel
            output reg op );

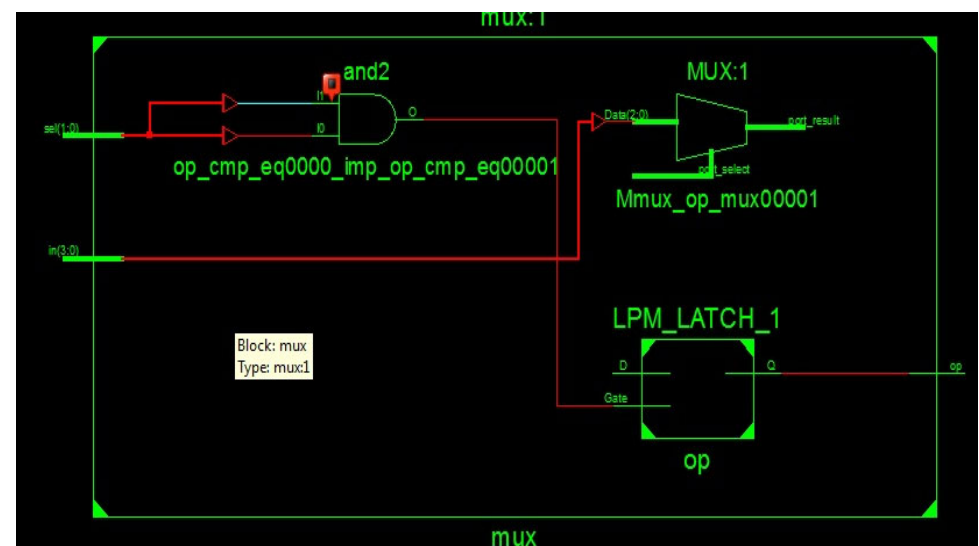
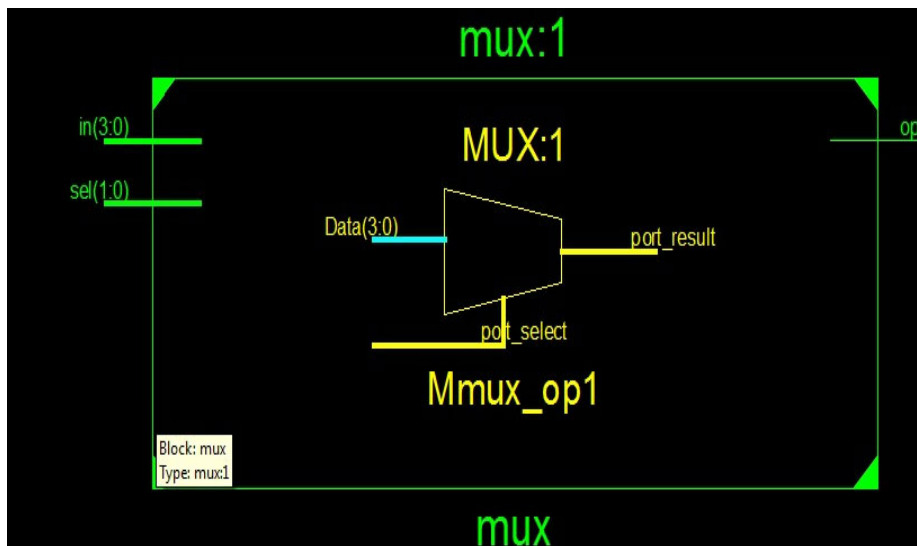
always @ *
begin
    case (sel)
        2'b00: op=in[0];
        2'b01: op=in[1];
        2'b10: op=in[2];
        2'b11: op=in[3];

    endcase
end
endmodule
```

```
module mux (input [3:0] in
            input [1:0] sel
            output reg op );

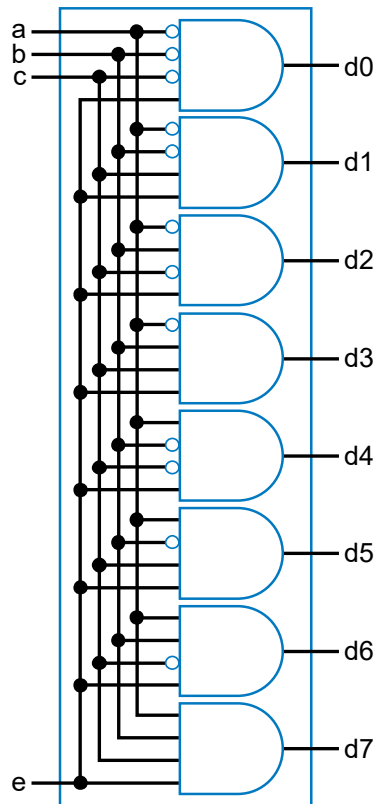
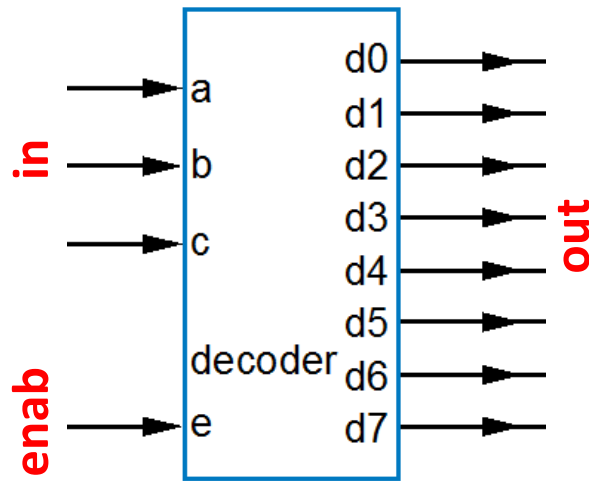
always @ *
begin
    case (sel)
        2'b00: op=in[0];
        2'b01: op=in[1];
        2'b10: op=in[2];

    endcase
end
endmodule
```



Ref: Figures from easyvlsi.wordpress.com

Example: Decoder with enable



```
module decoder (input enab, input [2:0] in,
               output reg [7:0] out);
```

```
    always @ * begin
```

```
        out = 8'h00;
```

```
        if (enab)
```

```
            case (in)
```

```
                3'b000 : out = 8'h01;
```

```
                3'b001 : out = 8'h02;
```

```
                3'b010 : out = 8'h04;
```

```
                3'b011 : out = 8'h08;
```

```
                . . .
```

```
                3'b110 : out = 8'h40;
```

```
                3'b111 : out = 8'h80;
```

```
            endcase
```

```
        end
```

```
    endmodule
```

What is the mistake and how could we fix it?
out is not assigned when *enab* is FALSE. Fix by adding a default assignment (not a **default** statement inside the case as that will not fix it)

Exercise 1

- Which of these Verilog descriptions will result in latches?

X

```
wire A, B;
reg Y;

always @(A or B)
begin
    Y = A|B;
end
```

Y

```
wire[1:0] x;
reg [1:0] q;

always @ (x)
begin
    case (x)
        2'b00: q = 2'b01;
        2'b10: q = 2'b10;
    endcase
end
```

Z

```
wire s;
reg q, r;

always @ (s)
begin
    case (s)
        1'b0: q = 1'b1;
        1'b1: r = 1'b1;
    endcase
end
```

- A. X
- B. Y
- C. Z
- D. X and Y
- E. Y and Z

Ans: Y and Z.

Y: Only 2 of 4 possible x. q not assigned for others

Z: If s=0 r is not assigned (same for q when s=1)

CE/CZ1005 2018-2019 Semester 1 (Nov/Dec 2018)

Q3(a) Given the Boolean function $F(a,b,c) = a'b'c' + ab'c + a'bc + abc'$

(iii) Write the Verilog module for F using a single assign statement.

ANS:

```
module someF (input a, b, c, output F);  
    assign F = ~a&~b&~c | a&~b&c | ~a&b&c | a&b&~c;  
endmodule
```

CE/CZ1005 2018-2019 Semester 1 (Nov/Dec 2018)

Q3(b) Write the Verilog module for a 2-to-4 decoder using a **case** statement. Use *[1:0] in* as the input and *[3:0] out* as the output.

ANS: **module** decoder2_4 (**input** [1:0] in, **output reg** [3:0] out);

always @ *

begin

out = 4'd1;

case (in)

2'b00 : out = 4'b0001;

2'b01 : out = 4'b0010;


2'b10 : out = 4'b0100;

2'b11 : out = 4'b1000;

endcase

end

endmodule



The default assignment is not needed in this instance, but it is good practice to use it.

Selected Past Exam Questions

CE/CZ1005 2018-2019 Semester 2 (Apr/May 2019)

Q3(b) A combinational circuit with a 6-bit input, $X[5:0]$, and output Y is given.

(i) Write the Verilog module for a 2-4 decoder. **ANS: See previous slide.**

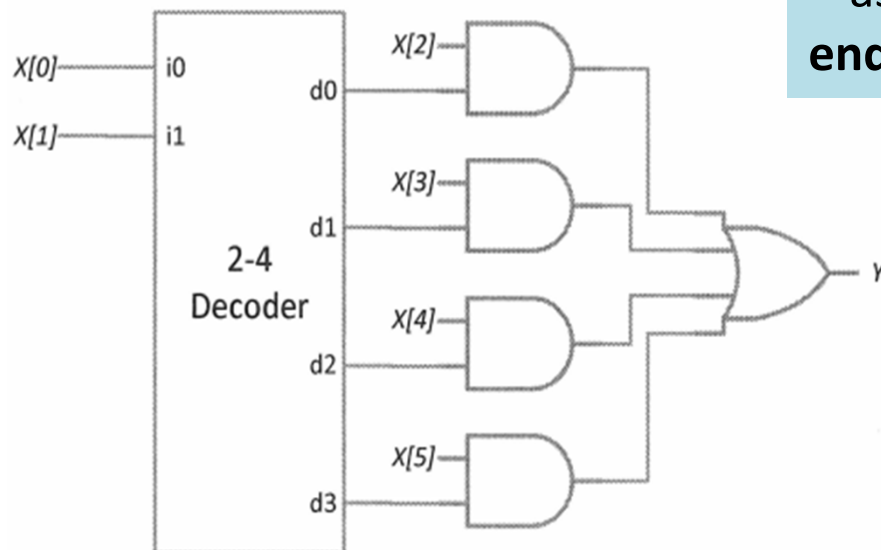
(ii) Write the Verilog for the circuit by instantiating the decoder in (i).

(iii) What is the commonly used circuit with the same functionality.

ANS (iii): It is a 4x1 multiplexer with input, $X[5:2]$, and select, $X[1:0]$.

(iv) Implement (iii) using behavioral Verilog.

```
module someCkt (input [5:0] X, output Y);
    wire [3:0] d;
    decoder2_4 U1 (.in (X[1:0]), .out (d));
    assign Y = d[0]&X[2] | d[1]&X[3] | d[2]&X[4] | d[3]&X[5];
endmodule
```



```
module someCkt (input [5:0] X, output reg Y);
    always @ * begin
        case (X[1:0])
            2'b00 : Y = X[2];
            2'b01 : Y = X[3];
            2'b10 : Y = X[4];
            2'b11 : Y = X[5];
        endcase
    end
endmodule
```

CE/CZ1005 2017-2018 Semester 2 (Apr/May 2018)

Q3(b) Given the sum-of-minterm expression $F(x, y, z) = \sum m(1, 6)$

(ii) Write the conditional assignment statement to implement F.

(3 marks)

ANS: Firstly, the solution to Q3(b)(i) was: $F = x'y'z + xyz'$

Thus one answer is: **assign** $F = z ? \sim x \& \sim y : x \& y ;$

Could also use: **assign** $F = x ? y \& \sim z : \sim y \& z ;$ or

assign $F = y ? x \& \sim z : \sim x \& z ;$

Note that a simple continuous assignment, as in:

assign $F = (\sim x \& \sim y \& z) \mid (x \& y \& \sim z) ;$

is easier to understand, but would not get you any marks.

CE/CZ1005 2017-2018 Semester 2 (Apr/May 2018)

Q3(b) Given the sum-of-minterm expression $F(x, y, z) = \sum m(1, 6)$

- (ii) Write the conditional assignment statement to implement F.
(3 marks)

ANS: As given in the solution to Q3(b)(i), $F = x'y'z + xyz'$

Thus one answer is: **assign** F = z ? ($\sim x \ \& \ \sim y$) : ($x \ \& \ y$);

Could also use: **assign** F = x ? ($y \ \& \ \sim z$) : ($\sim y \ \& \ z$); or
assign F = y ? ($x \ \& \ \sim z$) : ($\sim x \ \& \ z$);

Note that a simple continuous assignment, as in:

assign F = ($\sim x \ \& \ \sim y \ \& \ z$) | ($x \ \& \ y \ \& \ \sim z$);

is easier to understand, but would not get you any marks.

Also, when in doubt, add brackets

CE/CZ1005 2017-2018 Semester 2 (Apr/May 2018)

Q3(b) You have been hired as a design engineer in a chip manufacturing company to implement an arithmetic logic unit (ALU) for a computing system. The ALU has three inputs A , B and OP . Inputs A and B are two operands on which the ALU will perform the following arithmetic and logic operations: addition, subtraction, AND, OR and XOR. Input OP will determine the operation to be performed on A and B . The ALU generates a 32-bit output ALU_OUT . Based on the number of operations and ALU_OUT bit width, determine the correct bit size of A , B and OP . Write a behavioral Verilog module to implement the ALU.

(12 marks)

ANS: Firstly, addition and subtraction require an extra bit for the possible carry. However lets ignore carry.

So A and B are 32 bits, and OP is 3 bits (5 ops)

CE/CZ1005 2017-2018 Semester 2 (Apr/May 2018)

Q3(b) continued. There are many different ways to write the Verilog code. One way is to use case statements.

```
module alu #(parameter SIZE=32) (input [SIZE-1:0] A, B, input [2:0] op,  
                                output reg [SIZE-1:0] ALU_OUT);  
  
    always @ *  
    begin  
        ALU_OUT = 32'd0;  
        case (op)  
            3'b000 :          // NOP, output 0 (covered by default)  
            3'b001 : ALU_OUT = A+B;  
            3'b010 : ALU_OUT = A-B;  
            3'b011 : ALU_OUT = A&B;  
            3'b100 : ALU_OUT = A|B;  
            3'b101 : ALU_OUT = A^B;  
        endcase  
    end  
endmodule
```

CE/CZ1005 2017-2018 Semester 1 (Nov/Dec 2017)

Q3(a) Let $a = 5'b11011$, $b = 3'b101$ and $c = 3'b011$. Determine the value of $x[6:0]$ after executing each of the following Verilog assign statements.

- `assign x = {a[4:0], c[2], b[0]};`
- `assign x = a[4:0];`
- `assign x = b[1] ? {a, c[1:0]} : {b, a[3:0]};`
- `assign x = b & c;`

(6 marks)

ANS: `x = 11011_0_1`

`x = 00_11011`

`x = 101_1011`

`x = 0000_001`

CE/CZ1005 2017-2018 Semester 1 (Nov/Dec 2017)

Q3(c) There are three ants each standing at a different vertex of an equilateral triangle (i.e., the triangle's edges have equal lengths). The ants are free to move along the edges of the triangle. You are required to implement a combinational Verilog module which will detect the likelihood of collision of ants. Assume that the module has three 1-bit inputs and one 1-bit output. Each input represents the direction of movement of an ant on an edge of the triangle. The output goes high whenever there is a likelihood of collision. Briefly explain how you arrive at the solution.

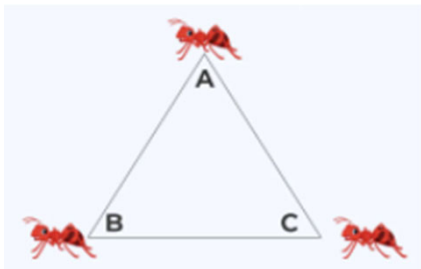
ANS: Lets assume from the perspective of each ant, a movement to the Left is a '0' and Right is a '1'. Then, for NO collision they must all go Left or all go Right, so we need to detect 000 or 111 = no_col = 0.

```
module col_detect (input d1, d2, d3, output col);
```

```
    assign col = ({d1,d2,d3} == 3'b000) ? 0 : (({d1,d2,d3} ==  
        3'b111) ? 0 : 1);
```

```
endmodule
```

Could also use a combinational always block with a case statement (or an if statement, etc).



Selected Past Exam Questions

CE/CZ1005 2016-2017 Semester 2 (Apr/May 2017)

Q3(a) Identify and correct the errors in the combinational Verilog module

```
module finderrors (input [2:0] a, b, c,  
                  input [1:0] sel,  
                  output [1:0] result);  
always @ (a,b,c)  
begin  
    case (sel)  
        2'b00 : result <= a;  
        2'b01 : result = b;  
        2'b10 : result = c;  
    endcase  
  
    end  
endmodule
```

Note: a,b and c being 3-bit and being assigned to a 2-bit result is not an error

1. Reg needed. Use: **output reg** [1:0] result);

2. Incomplete sensitivity list. Use: **always @ ***

3. Non-blocking assignment is incorrect
4. A latch will be inferred. Either add a default statement or add an assignment to **result** above the case

ANS: There are 4 errors

CE/CZ1005 2016-2017 Semester 2 (Apr/May 2017)

Q3(a) What is wrong with the Verilog module. Fix it.

```
module fixit (input p, q, r, z,  
             output reg X, Y);  
    always @ *  
    begin  
        if(p)  
            X = q & r;  
        else  
            begin  
                X = z;  
                Y = q ^ z;  
            end  
        end  
    endmodule
```

The problem with this circuit is that Y is not assigned in the **if** section and the Verilog synthesizer would infer a latch to hold Y. However, without a description stating what the circuit should do, it is impossible to fix. So let's make some assumptions. Let's assume that when $p=1$, $Y=0$. Then the code between the **begin** and the **end** becomes:

Could also use:

```
X = z;  
Y = q ^ z;  
if (p) begin  
    X = q & r;  
    Y = 0;  
end
```

CE/CZ1005 2016-2017 Semester 2 (Apr/May 2017)

Q3(c)(i) $F(a,b,c) = a'b'c' + a'bc' + ab'c' + abc$

Q3(c)(ii) Write the Verilog module for F in Q3(c)(i) using a **case** statement.

(7 marks)

ANS:

```
module someF (input a, b, c, output reg F);  
  
    always @ *  
    begin  
        F = 1'b0;  
        case ({a,b,c})  
            3'b000 :    F = 1'b1;  
            3'b010 :    F = 1;    //easier  
            3'b100 :    F = 1;  
            3'b111 :    F = 1;  
        endcase  
    end  
endmodule
```

