Digital System Design with HDL (I)

Lecture 7

Dr. Ming Xu and Dr. Kain Lu Low

Dept of Electrical & Electronic Engineering

XJTLU

In This Session

- Functions and Tasks
- Verilog for Combinational Circuits
  - Multiplexers
  - Decoders
  - Encoders
  - Comparators

Functions

- A function is a part of the code that may be re-used and thus separated from the other parts in a module.
- Syntax:

  function [*size_or_type*] function_name;

  input declarations

  local variable declarations

  procedural statements

  endfunction

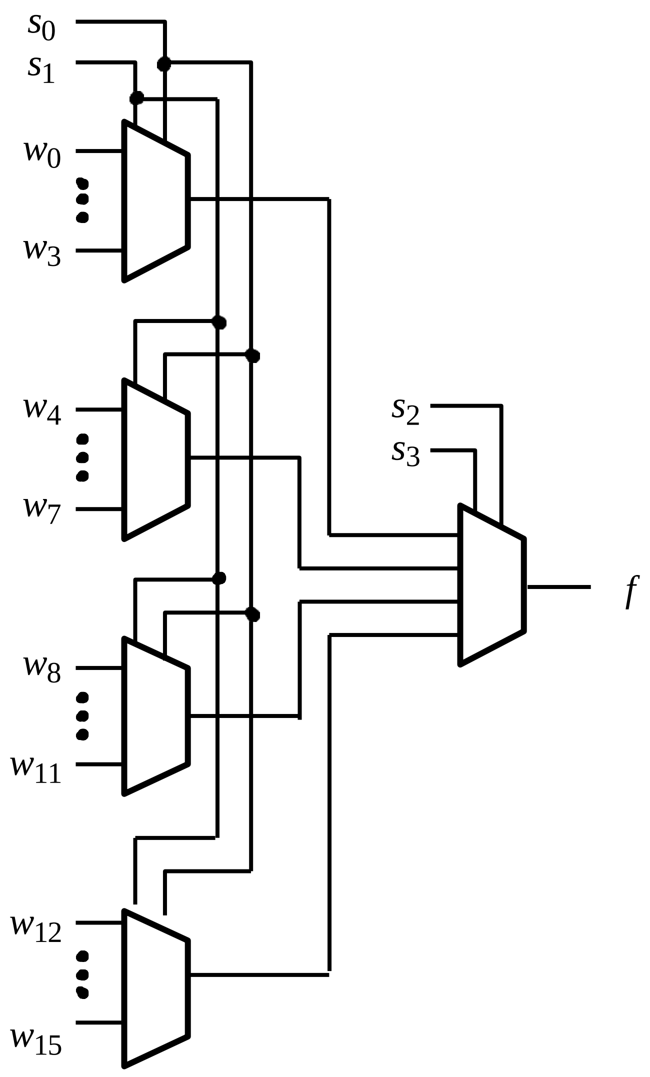
  size_or_type (optional) is the returned type or range as [*msb*:*lsb*]. The default is a 1-bit value.

Functions

- Functions should have at least one input.
- The function name serves as the *only* output variable. No other output or inout is allowed.
- Functions are only defined within a module.

Functions

Example: a 16-to-1 multiplexer



```
module mux_f (W, S16, f);
  input [0:15] W;
  input [3:0] S16;
  output reg f;
  reg [0:3] M;

  function mux4to1;
    input [0:3] W;
    input [1:0] S;

    if (S = = 0) mux4to1 = W[0];
    else if (S = = 1) mux4to1 = W[1];
  else if (S = = 2) mux4to1 = W[2];
  else if (S = = 3) mux4to1 = W[3];
  endfunction

  always @(W, S16)
  begin
    M[0] = mux4to1(W[0:3], S16[1:0]);
    M[1] = mux4to1(W[4:7], S16[1:0]);
    M[2] = mux4to1(W[8:11], S16[1:0]);
    M[3] = mux4to1(W[12:15], S16[1:0]);
    f = mux4to1(M[0:3], S16[3:2]);
  end

endmodule
```

Tasks

- A task is a part of the code that may be re-used and thus separated from the other parts in a module.
- A task may have any number of inputs, outputs or inouts.
- A task does not return any value by its name.
- A task is like a subroutine.

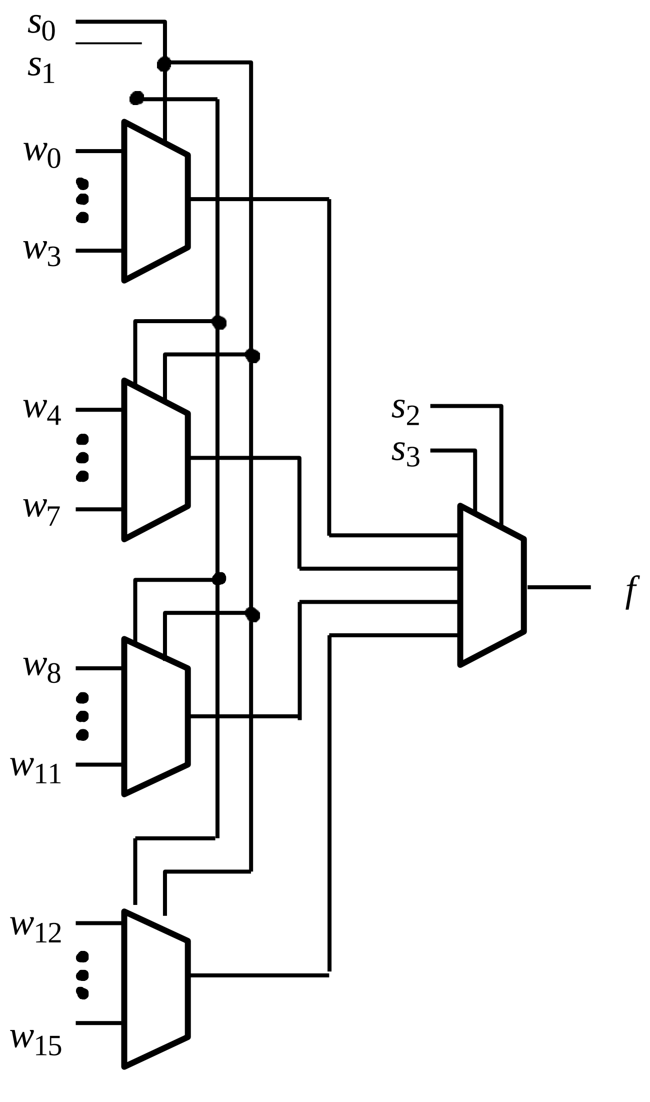
Tasks

Syntax:

```
task task_name;
   input declarations
   output declarations
   local variable declarations
   procedural statements
endtask
```

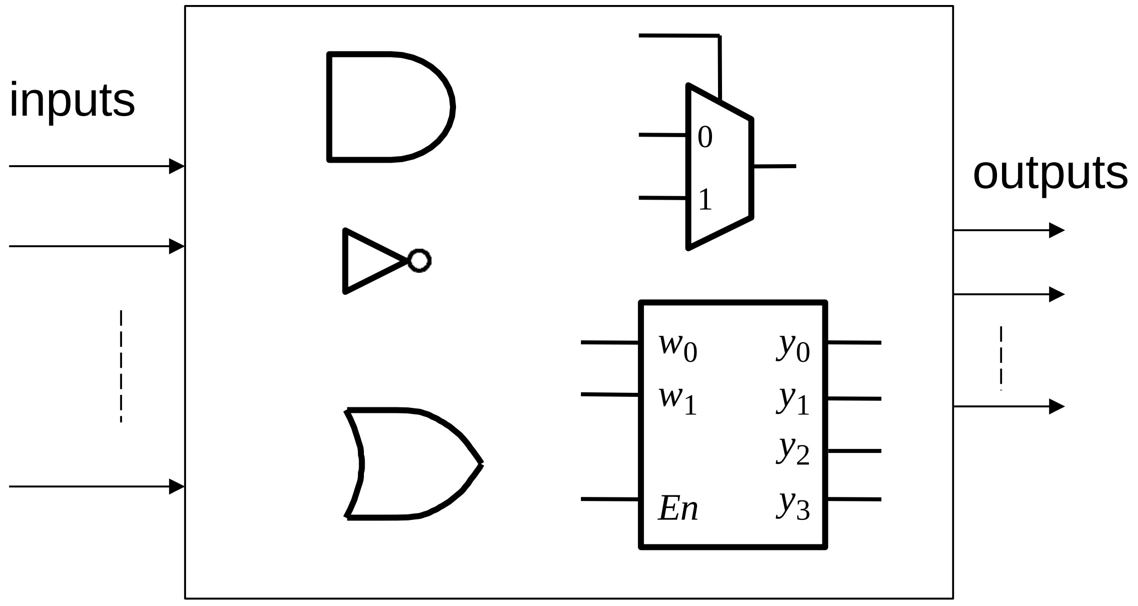
Tasks

Example: a 16-to-1 multiplexer



```
module mux_t (W, S16, f);
  input [0:15] W;
  input [3:0] S16;
  output reg f;
  reg [0:3] M;

  task mux4to1;
    input [0:3] W;
    input [1:0] S;
    output Result;
    begin
      if (S = = 0) Result = W[0];
      else if (S = = 1) Result = W[1];
    else if (S = = 2) Result = W[2];
    else if (S = = 3) Result = W[3];
    end
  endtask

  always @(W, S16)
  begin
    mux4to1(W[0:3], S16[1:0], M[0]);
    mux4to1(W[4:7], S16[1:0], M[1]);
    mux4to1(W[8:11], S16[1:0], M[2]);
    mux4to1(W[12:15], S16[1:0], M[3]);
    mux4to1(M[0:3], S16[3:2], f);
  end

endmodule
```

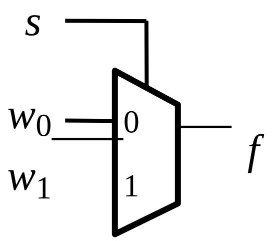
Combinational circuits



- Outputs only depend on inputs
- Could include: gates, multiplexers, encoders, decoders, code converters, comparators …
- Verilog description: gates, logic expression, behaviour

Multiplexers

Using conditional operators



s	f
0	w_0
1	w_1

```
module mux2to1 (w0, w1, s, f);
      input w0, w1, s;
      output f;

      assign f = s ? w1 : w0;

endmodule
```

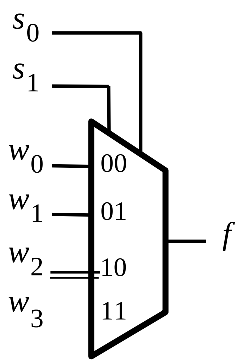
Conditional operator
in continuous assignment

```
module mux2to1 (w0, w1, s, f);
      input w0, w1, s;
      output f;
      reg f;

      always @(w0 or w1 or s)
            f = s ? w1 : w0;

endmodule
```

Conditional operator
in procedural statement

Multiplexers

Using nesting conditional operators



s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

```
module mux4to1 (w0, w1, w2, w3, S, f);
        input w0, w1, w2, w3;
        input [1:0] S;
        output f;

        assign f = S[1] ? (S[0] ? w3 : w2) : (S[0] ? w1 : w0);

endmodule
```

Multiplexers

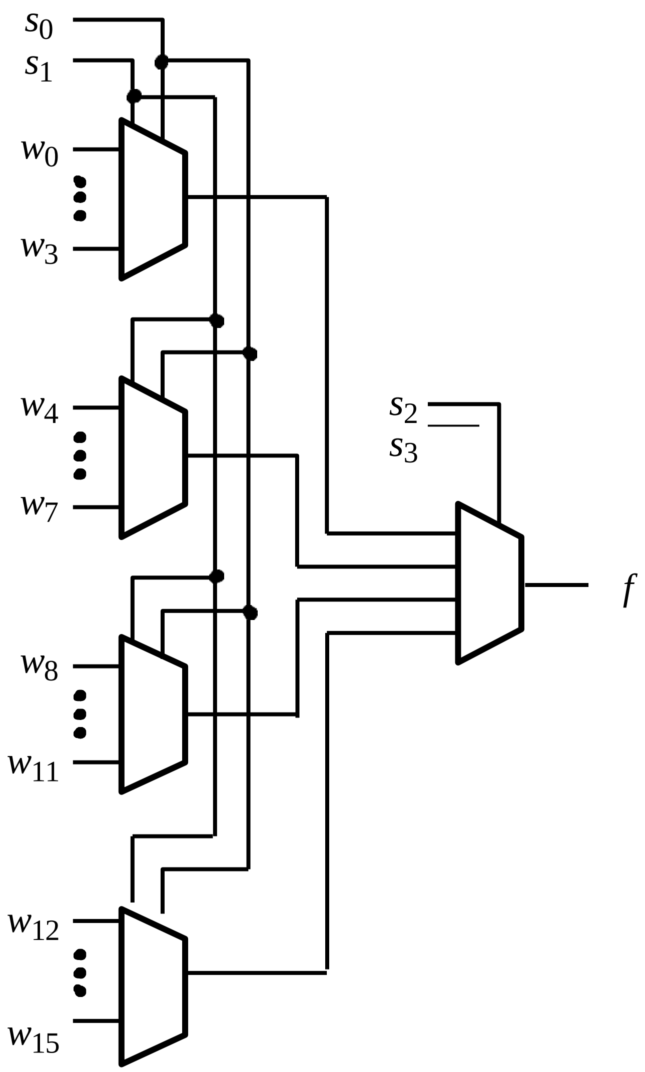
Using If-else statements

```
module mux2to1 (w0, w1, s, f);
        input w0, w1, s;
        output f;
        reg f;

        always @(w0 or w1 or s)
               if (s= =0)
                     f = w0;
               else
                     f = w1;

endmodule
```

Using case statements

```
module mux4to1 (W, S, f);
        input [0:3] W;
        input [1:0] S;
        output f;
        reg f;

        always @(W or S)
                case (S)
                       0: f = W[0];
                       1: f = W[1];
                       2: f = W[2];
                       3: f = W[3];
                endcase

endmodule
```

Multiplexers

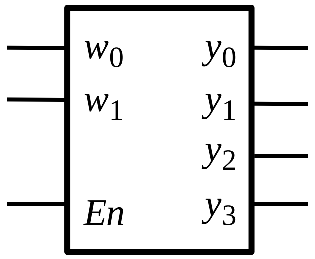


Hierarchical 16-to-1 Multiplexer

```
module mux16to1 (W, S16, f);
        input [0:15] W;
        input [3:0] S16;
        output f;
        wire [0:3] M;

        mux4to1 Mux1 (W[0:3], S16[1:0], M[0]);
        mux4to1 Mux2 (W[4:7], S16[1:0], M[1]);
        mux4to1 Mux3 (W[8:11], S16[1:0], M[2]);
        mux4to1 Mux4 (W[12:15], S16[1:0], M[3]);
        mux4to1 Mux5 (M[0:3], S16[3:2], f);

endmodule
```

Decoders

Decoder using case



En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

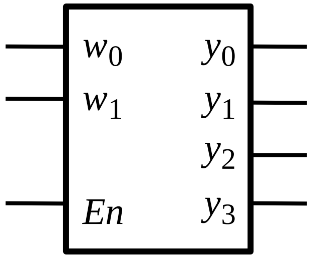
```
module dec2to4 (W, Y, En);
    input [1:0] W;
    input En;
    output [0:3] Y;
    reg [0:3] Y;

    always @(W or En)
        case ({En, W})
            3'b100: Y = 4'b1000;
            3'b101: Y = 4'b0100;
            3'b110: Y = 4'b0010;
            3'b111: Y = 4'b0001;
            default: Y = 4'b0000;
        endcase

endmodule
```

Concatenate operator

Default for En=0

Decoders

Decoder using if-else, case

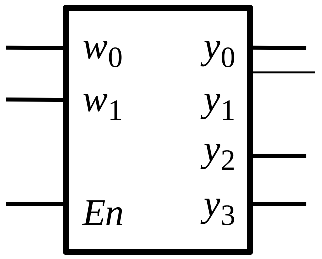


En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

```
module dec2to4 (W, Y, En);
        input [1:0] W;
        input En;
        output [0:3] Y;
        reg [0:3] Y;

        always @(W or En)
        begin
            if (En == 0)
               Y = 4'b0000;
            else
               case (W)
                    0: Y = 4'b1000;
                    1: Y = 4'b0100;
                    2: Y = 4'b0010;
                    3: Y = 4'b0001;
               endcase
        end

endmodule
```

Decoders

Decoder using for loops

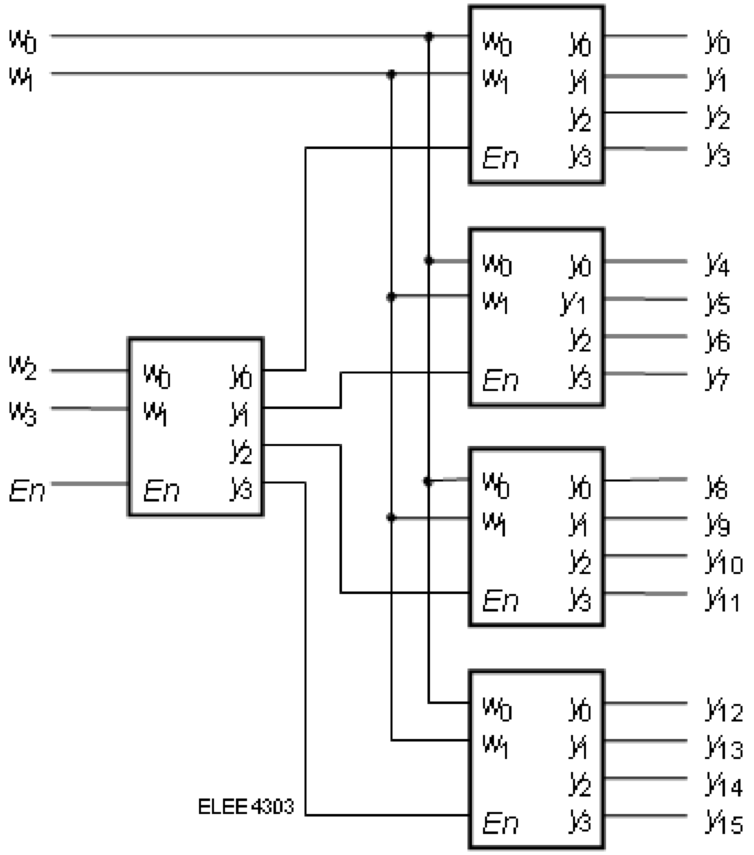


En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

```
module dec2to4 (W, Y, En);
    input [1:0] W;
    input En;
    output reg [0:3] Y;
    integer k;

    always @(W, En)
        for (k = 0; k <= 3; k = k+1)
        if ((W == k) && (En == 1))
            Y[k] = 1;
        else
            Y[k] = 0;

endmodule
```

Decoders

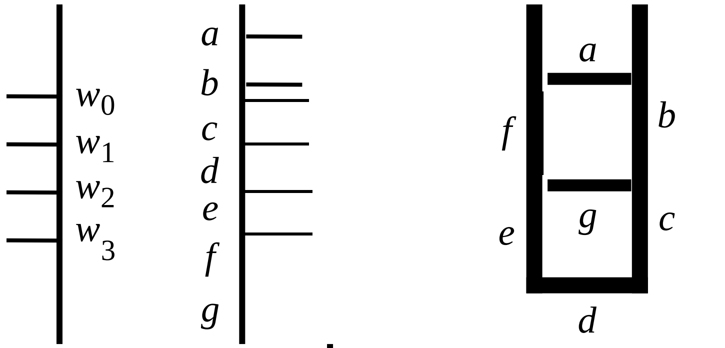
Hierarchical code for 4-16 decoder



```
module dec4to16 (W, Y, En);
        input [3:0] W;
        input En;
        output [0:15] Y;
        wire [0:3] M;

        dec2to4 Dec1 (W[3:2], M[0:3], En);
        dec2to4 Dec2 (W[1:0], Y[0:3], M[0]);
        dec2to4 Dec3 (W[1:0], Y[4:7], M[1]);
        dec2to4 Dec4 (W[1:0], Y[8:11], M[2]);
        dec2to4 Dec5 (W[1:0], Y[12:15], M[3]);

endmodule
```

Decoders

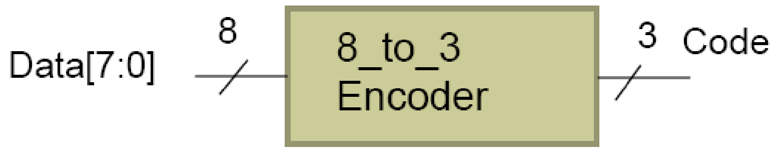
BCD-to-7-segment decoder



w_3	w_2	w_1	w_0	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

```
module seg7 (bcd, leds);
    input [3:0] bcd;
    output [1:7] leds;
    reg [1:7] leds;

    always @(bcd)
        case (bcd)      //abcdefg
            0: leds = 7'b1111110;
            1: leds = 7'b0110000;
            2: leds = 7'b1101101;
            3: leds = 7'b1111001;
            4: leds = 7'b0110011;
            5: leds = 7'b1011011;
            6: leds = 7'b1011111;
            7: leds = 7'b1110000;
            8: leds = 7'b1111111;
            9: leds = 7'b1111011;
            default: leds = 7'bx;
        endcase
endmodule
```

Encoders



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

always@ ( Data )
    case (Data)
        8'b00000001 : Code = 3'b000;
        8'b00000010 : Code = 3'b001;
        8'b00000100 : Code = 3'b010;
        8'b00001000 : Code = 3'b011;
        8'b00010000 : Code = 3'b100;
        8'b00100000 : Code = 3'b101;
        8'b01000000 : Code = 3'b110;
        8'b10000000 : Code = 3'b111;
      default Code = 3'bx;
    endcase

endmodule
```

Encoders

Priority encoder using casex

w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

```
module priority (W, Y, z);
    input [3:0] W;
    output [1:0] Y;
    output z;
    reg [1:0] Y;
    reg z;

    always @(W)
    begin
        z = 1;
        casex(W)
            4'b1xxx: Y = 3;
            4'b01xx: Y = 2;
            4'b001x: Y = 1;
            4'b0001: Y = 0;
            default: begin
                         z = 0;
                         Y = 2'bx;
                     end
        endcase
    end
endmodule
```

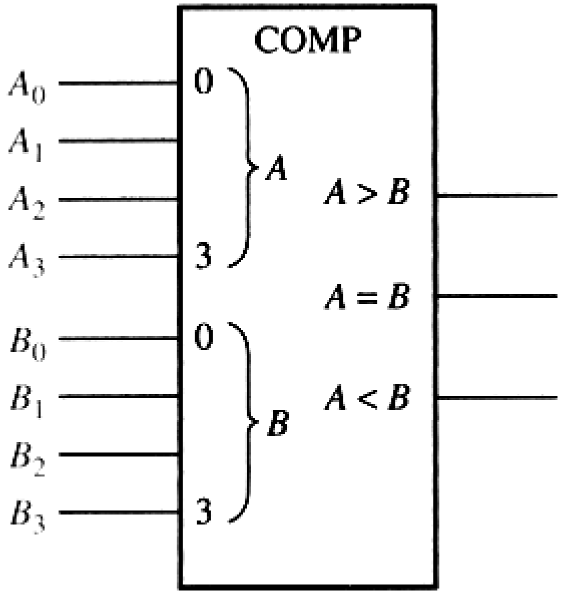
Encoders

Priority encoder using for loops

w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

```
module priority (W, Y, z);
    input [3:0] W;
    output reg [1:0] Y;
    output reg z;
    integer k;

    always @(W)
    begin
        Y = 2'bx;
        z = 0;
        for (k = 0; k < 4; k = k+1)
            if (W[k])
            begin
                Y = k;
                z = 1;
            end
    end

endmodule
```

Comparators



```
module compare (A, B, AeqB, AgtB,AltB);
    input [3:0] A, B;
    output reg AeqB, AgtB,AltB;

    always @(A, B)
    begin
        AeqB = 0;
        AgtB = 0;
        AltB = 0;
        if(A == B)
        AeqB = 1;
        else if (A > B)
          AgtB = 1;
        else
          AltB = 1;
    end

endmodule
```